# Intégration de systèmes hétérogènes en termes de niveaux de sécurité

Matthieu Lemerre

UNIVERSITÉ PARIS SUD XI - ORSAY

ÉCOLE DOCTORALE D'INFORMATIQUE

# T H È S E

pour obtenir le titre de

**Docteur en Sciences**

de l'Université de Paris Sud - Orsay
**Spécialité : Informatique**

Présentée et soutenue par

Matthieu LEMERRE

# Intégration de systèmes hétérogènes en termes de niveaux de sécurité

soutenue le 5 octobre 2009

JURY :

| | | |
|---|---|---|
| Examinateur | Albert COHEN | *INRIA* |
| Co-directeur de thèse | Vincent DAVID | *CEA LIST* |
| Rapporteur | Jonathan SHAPIRO | *Johns Hopkins University* |
| Rapporteur | Yvon TRINQUET | *IRCCyN* |
| Examinateur | Jean-Pierre VERJUS | *INRIA* |
| Directeur de thèse | Guy VIDAL-NAQUET | *Supélec et Université Paris XI* |

# Résumé

Cette thèse étudie les principes de mise en oeuvre pour l'exécution sur un même ordinateur, de tâches de niveaux de criticité différents, et dont certaines peuvent avoir des contraintes temps réel dur.

Les difficultés pour réaliser ces objectifs forment trois catégories. Il faut d'abord prouver que les tâches disposeront d'assez de ressources pour s'exécuter ; il doit être ainsi possible d'avoir des politiques d'allocations et d'ordonnancement sûres, prévisibles et simples. Il faut également apporter des garanties de sécurité pour s'assurer que les tâches critiques s'exécuteront correctement en présence de défaillances ou malveillances. Enfin, le système doit pouvoir être réutilisé dans une variété de situations.

Cette thèse propose de s'attaquer au problème par la conception d'un système hautement sécurisé, extensible, et qui soit indépendant des politiques d'allocation de ressources. Cela est notamment accompli par le prêt de ressource, qui permet de décompter les ressources indépendamment des domaines de protection. Cette approche évite d'avoir à partitionner les ressources, ce qui simplifie le problème global de l'allocation et permet de ne pas gâcher de ressources. Les problèmes de type inversion de priorité, famine ou dénis de service sont supprimés à la racine. Nous démontrons la faisabilité de cette approche à l'aide d'un prototype, Anaxagoros.

La démarche que nous proposons simplifie drastiquement l'allocation des ressources mais implique des contraintes dans l'écriture de services partagés (comme les pilotes de périphériques). Les principales difficultés consistent en des contraintes de synchronisation supplémentaires. Nous proposons des mécanismes originaux et efficaces pour résoudre les problèmes de concurrence et synchronisation, et une méthodologie générale pour faciliter l'écriture sécurisée de ces services partagés.

# Summary

This thesis studies design and implementation principles to execute tasks of different criticity levels onto the same computer. Additionally, some of these tasks may have hard real-time constraints.

This requires to prove that tasks will get enough resources to execute properly, through the use of predictible and still simple allocation policies. Moreover, ensuring that critical tasks will execute correctly in presence of faults is needed. In particular, providing guarantees on resource allocation should be possible. At last, the system should be easily adaptable to different situations.

This thesis tackles these issues through a design proposal for a highly secure and extensible system, which is also independent of resource allocation policies. This is accomplished in particular by systematic use of resource lending, which allows to account for resources independently of protection domains. This approach avoids partitionning resources into pools, simplifying the global allocation problem and deleting every waste of resources. We demonstrate that this approach is feasible using a prototype implementation.

This methodology dramatically simplifies resource allocation, but implies addi-

tional constraints when writing shared services (e.g. device drivers). In particular, specific new synchronization problems occur. Original mechanisms to solve these problems are proposed, and a methodology that helps writing these shared services.

# Remerciements

Je tiens à remercier toutes les personnes qui ont contribué à la réussite de ce travail de thèse.

Je remercie vivement Vincent David et Guy Vidal-Naquet pour leur encadrement et leurs conseils, et pour la confiance qu'ils m'ont accordée pendant la réalisation de ces travaux. Merci également pour cette ambiance de travail agréable et propice à mener à bien ces recherches.

Je remercie Jonathan Shapiro et Yvon Trinquet pour avoir pris le temps d'évaluer minutieusement ces travaux de recherche, et pour leurs nombreuses corrections qui ont permis d'améliorer la qualité de ce rapport.

Je remercie également Albert Cohen et Jean-Pierre Verjus pour m'avoir fait l'honneur d'accepter de faire partie de mon jury de thèse.

Merci à tous les membres (présents et passés) du LaSTRE pour leur accueil, leur compagnie et les discussions scientifique que nous avons pu avoir durant ces trois ans. C'est avec plaisir que je continuerai à travailler avec eux. Merci aux personnes qui ont accepté de relire tout ou partie de ce long document. Merci enfin aux autres doctorants pour m'avoir permis de décompresser, et en particulier Fabien pour ses « enseignements ».

Je tient à remercier toutes les personnes qui ont contribué à me former un esprit scientifique : mon père, qui m'a donné très jeune un esprit curieux et initié à l'informatique ; tous les enseignants qui n'ont pas du toujours avoir la tâche facile ; et mes encadrants de thèse pour m'avoir appris à mieux structurer mes idées.

Je remercie tous mes amis pour m'avoir changé les idées, Assia, Cécile, Christophe, Doan-Y-Nhi, Émilie, Émmanuel, Fabien, Farid, Florian, Jean-Paul, Jérome, Grégoire, Guy, Ilias, Jean-Thomas, Jean-Sylvain, Julien, Michel, Nathalie, Nicolas, Pablo, Renaud, Rim, Samer, Sergiu, Stéphane, Thomas, Victor, Vincent, Yvain, et toutes les personnes que j'ai oublié de citer.

Merci enfin à mes parents, à Julie et Gwenn, et à Nathalie, pour leur soutien continu.

# Sommaire

Chapitre 1

# Introduction

## 1.1 Problèmatique

Les capacités de calcul des ordinateurs modernes sont souvent supérieures aux besoins individuels des logiciels que l'on souhaite faire fonctionner. Il est donc souhaitable, pour des raisons de coût, d'y faire fonctionner de manière concommittante plusieurs tâches logicielles. Ce rôle est assuré par le système d'exploitation (OS), qui supervise l'exécution des tâches dans le calculateur.

L'amélioration des performances des calculateurs soulève de nouveaux besoins. Il devient par exemple possible d'intégrer dans un seul calculateur des tâches qu'on devait auparavant placer sur des calculateurs différents. Il faut donc être capable d'intégrer efficacement et de manière sécurisée des tâches très différentes.

Ces tâches diffèrent par leurs spécifications. Elles peuvent être d'importance ou de degrés de criticité différentes. Leur usage des ressources peut être différent. Une contrainte particulièrement difficile à prendre en compte est le *temps réel*, qui pose des contraintes de temps d'exécution sur les tâches en raison des procédés physiques qui sont connectés au calculateur. Elles diffèrent également par leur implémentation, leurs environnements d'exécution (e.g. les APIs), ou les politiques d'allocations utilisées.

Le système d'exploitation doit fournir certaines garanties permettant de s'assurer que les tâches s'exécutent correctement. Plusieurs aspects qui rendent cette tâche difficile sont pris en compte dans cette thèse :

- la présence de tâches temps réel (dur). Cette contrainte implique que le système puisse garantir que l'exécution des tâches se déroulera dans les intervalles spécifiés, donc sache contrôler les instants où les différents traitements du système vont s'exécuter, et comment les ressources sont attribuées. Il est très difficile de réaliser cela sans vider le système de ses fonctionalités : on a souvent le choix entre un système temps réel (en abrégé RTOS) disposant de peu de fonctionnalités, et un système « généraliste (general-purpose) » incapable de garantir que les tâches seront exécutées à temps ;
- la présence dans le système de tâches critiques. Pour ces tâches, il ne suffit

pas que le système fonctionne correctement : il faut fournir la garantie d'une exécution correcte malgré les dysfonctionnements des autres tâches ou de l'OS lui-même : c'est la *sécurité* et *sûreté de fonctionnement*. Ces dysfonctionnements peuvent provenir de bugs, de défaillances transitoires dans l'exécution, ou de code malveillant ;

- la présence de tâches non-critiques. Il s'agit de tâches dont le fonctionnement est partiellement inconnu, et il ne faut pas présumer de leur bon comportement pour garantir la sécurité des tâches critiques. On peut même présumer, pour la preuve de sûreté, que les tâches non-critiques sont malicieuses, et vont tenter d'empêcher l'exécution normale des tâches critiques. En particulier, la présence de tâches non-critiques peut perturber l'exécution du système de sorte que les échéances de tâches critiques temps réel ne soient pas respectées.

  Cependant, le fait que ces tâches soient non-critiques ne doit pas les empêcher de s'exécuter correctement : ainsi une tâche non-critique temps réel devrait respecter son échéance si cela est possible.

Cette thèse propose une méthode d'écriture de système d'exploitation qui permette de simultanément prendre en compte ces trois aspects. Elle comporte une étude qui identifie différents concepts importants pour ce but ; une conception générale de l'ensemble du système ; et d'une méthode de conception détaillée des services de l'OS ; ainsi que de mécanismes nombreux et divers permettant la mise en œuvre de ces conceptions.

**Motivation et applications** Les utilisations potentielles d'un tel système d'exploitation sont nombreuses. La motivation à l'origine de cette thèse est l'industrie du transport (avionique, automobile, etc.), dont une grande partie du contrôle-commande est maintenant faite par logiciel afin de profiter de lois de commande mieux adaptées. Ces logiciels ont des contraintes temps réel dur, et sont souvent critiques. Les systèmes de transports intègrent également des logiciels moins critiques, par exemple pour gérer la communication avec les passagers (dans l'avionique), ou pour la gestion de la radio (dans l'automobile).

Auparavant dans les systèmes avioniques, chaque fonction était assurée par un calculateur dédié différent, ce qui posait de nombreux problèmes : partage de réseau, matériel de communication (cables), et nombre de calculateurs nécessaires, occasionnant ainsi un coût élevé. D'où l'idée de minimiser le nombre de calculateurs utilisés en intégrant différents logiciels dans le même calculateur, sans pour autant compromettre la sûreté du système. Rushby synthétise l'intérêt de cette approche pour l'avionique [Rus98]. L'automobile utilise désormais également beaucoup de calculateurs, et cherche aussi à les mutualiser.

Il y a beaucoup d'autres applications possibles. Par exemple, les systèmes embarqués dans les appareils médicaux ont des fonctions d'utilisabilité complexes (e.g. interfaces graphiques) et des fonctions critiques pour la vie des patients. Il est important de garantir que les fonctions critiques ne peuvent pas être impactées. Un autre exemple d'application est la téléphonie mobile : les téléphones portables exécutent à la fois des fonctions temps réel (pour la communication), et des tâches

de confort (e.g jeux) qui ne le sont pas. Il y a également des contraintes de sécurité : les téléphones peuvent maintenant exécuter des programmes qui viennent d'Internet, et il faut assurer la confidentialité des données de l'utilisateur (e.g. carnet d'adresse).

Plus généralement, les systèmes d'exploitations généralistes actuels souffrent d'un déficit en sécurité (en témoigne le nombre de solutions d'« antivirus ») et en contrôle des ressources (pas de comptage précis des ressources utilisées, difficultés d'exécuter des applications multimédia). Il y a donc besoin d'un système qui puisse résoudre ces problèmes.

**Approches usuelles pour le temps réel** Il est possible d'améliorer un OS généraliste pour qu'il améliore sa prise en compte des tâches temps réel, mais cela est loin d'être suffisant pour l'exécution sûre de tâches temps réel critiques. L'ordonnancement n'est pas le seul problème : l'allocation des autres ressources comme la mémoire est également trop complexe et imprévisible.

Pour le temps réel dur, l'approche usuelle consiste en l'utilisation d'un système dédié (RTOS), organisé autour d'un algorithme d'ordonnancement CPU précis [SR04], les autres ressources étant allouées statiquement. Ces systèmes sont le plus souvent des noyaux monolithiques optimisés pour avoir une préemption rapide avec l'emploi d'un algorithme d'ordonnancement par priorité fixe.

La raison de l'utilisation de ces systèmes est qu'ils permettent des méthodologies de developpement relativement proches des systèmes habituels (e.g. Ravenscar profile [BDV03]), tout en permettant des analyses pour prédire si les tâches critiques pourront atteindre leurs échéances. Les problèmes de concurrences sont résolus par des verrous, comme dans la programmation parallèle classique.

Il y a de nombreux inconvénients à cette approche ; les principaux étant un manque de sécurité (il n'y a aucune garantie pour les tâches peu prioritaires, dès lors que les tâches prioritaires peuvent boucler[1]), un manque de performance (la politique d'ordonnancement n'est pas optimale), une complexité élevée due à l'utilisation d'une pléthore de verrous (problèmes d'inversion de priorité), qui rendent le système imprévisible ou difficile à analyser. Ces défauts rendent difficile l'utilisation de cette méthodologie pour la construction de gros systèmes, ou de systèmes disposant de fonctionnalité avancées.

Cette incapacité est contournée par l'ajout d'une sous-couche de virtualisation (e.g. ARINC 653[Aer]), qui apporte à la fois une protection temporelle entre les partitions, et la possibilité de faire cohabiter des tâches temps réel dur avec des tâches requierant des services de l'OS complexes. Mais la protection temporelle est apportée uniquement à la granularité d'une partition, rend difficile le partage de services complexes de l'OS entre partitions, et rajoute une couche qui augmente la complexité du système et nuit aux performances.

[1] l'addition d'un mécanisme de quota de temps CPU résoud le problème, mais cela ne constitue plus un mécanisme de priorité fixe au sens strict

## 1.2 Buts et approche proposée

L'analyse des approches usuelles et des besoins auxquelles elles répondent nous permettent d'évaluer les propriétés nécessaires à assurer dans un système répondant aux problèmatiques de la thèse.

L'idée générale est que pour garantir qu'une tâche critique puisse s'exécuter correctement, il faut et suffit de *réserver* les ressources nécessaires à ses besoins. Et cela indépendemment de la présence d'autre tâches, critiques ou non. Mais cela pose des problèmes à plusieurs niveaux. Nous les analysons et proposons des solutions pour améliorer les performances, la sécurité, la simplicité de système temps réel (aux niveaux de sécurités hétérogènes).

**Prédictibilité des besoins en ressource** Tout d'abord, il faut pouvoir analyser précisément les besoins en ressources des tâches. Il y a des difficultés qui ne concernent pas le système d'exploitation, par exemple l'analyse des temps d'exécution ou WCET. Nous ne nous préoccupons donc pas de ces problématiques, et supposons disposer d'un modèle des besoins des tâches qui majore les besoins réels. Nous ne supposons rien sur ce modèle des besoins des tâches.

Le seul point qui nous concerne est la prédictibilité des ressources dépensées par l'OS. C'est à dire essentiellement le temps CPU et la mémoire utilisée par l'OS pour servir les tâches. Il faut faire en sorte que ces besoins soient facilement prévisibles, pour qu'ils puissent être pris en compte dans les modèles de besoins en ressource des tâches. Il est beaucoup plus difficile de faire cela lorsque l'OS est dynamique : quand le nombre de tâches et l'allocation de ressources est statique, l'OS a peu de gestion à faire et ses besoins sont donc facilement prévisibles.

**Prise en compte fine des besoins** Il faut ensuite que ces besoins puissent être pris en compte par l'OS, i.e. qu'il sache distribuer les ressources selon les besoins. Il y a de nombreuses manière de modeliser les besoins en ressources des applications : rien que pour les besoins en temps CPU, on trouve les tâches périodiques, sporadiques, multi-périodiques, ou des modèles basés sur les automates comme OASIS [AD98].

Usuellement, l'OS fournit des abstractions que les développeurs doivent utiliser pour faire prendre en compte leurs besoins en temps CPU par l'OS. Des exemples de telles abstractions sont les priorités, les quotas cycliques de temps CPU (i.e. un quota garanti de $x$ secondes toutes les $y$) ou l'allocation d'un slot sur un cycle (ordonnancement dit statique). Ces abstractions ne permettent pas de prendre en compte des tâches aux modèles de besoins plus précis comme OASIS. Cela conduit à réserver plus de ressources que nécessaire, donc à gâcher des ressources.

Plutôt que d'essayer de trouver une abstraction qui convienne à tous les modèles de besoin de tâche, nous proposons que le concepteur puisse définir *entièrement* la politique d'allocation, en nommant explicitement les ressources physiques, sans abstraction. C'est à dire que le temps CPU est allouée selon la notion d'intervalle de temps CPU ; la mémoire est allouée à la granularité d'une page sur machine à base de pagination, selon la notion d'intervalle de mémoire sur les architectures à base de segmentation. Plusieurs systèmes ont proposé de supprimer les abstractions

(e.g. [EKJO95]), mais nous allons plus loin en proposant de rendre l'allocation des ressources complètement indépendante du reste de l'OS.

**Prédictabilité des allocations** Pour s'assurer que les tâches critiques reçoivent assez de ressource pour leur exécution (et atteignent leurs échéances pour les tâches temps réel), il faut pouvoir prédire la quantité de ressource qui sera attribué à la tâche avant l'exécution. Nous disons que l'OS supporte les allocations prévisibles. Cette fonctionnalité est la raison d'être des RTOS.

Nous allons plus loin, en faisant en sorte que lorsqu'une politique décide de l'allocation de ressources à une tâche, cette décision d'allocation soit respectée et ne puisse pas être changée par le reste de l'OS. Nous disons que l'OS permet une *allocation déterministe* de la ressource. Un contre exemple est l'utilisation de sémaphores dans le noyau, qui change les décisions d'ordonnancement de manière imprévisible.

allocation déterministe

Cette propriété est très importante, car il devient simple d'écrire une politique d'allocation qui puisse garantir que chaque tâche a assez de ressource pour s'exécuter. Cela élimine également le besoin d'utiliser une approche « conservative » de l'allocation nécessaire du fait que l'allocation est non-déterministe. Par exemple, il devient inutile d'ordonnancer les tâches critiques prioritairement aux tâches non-critiques (comme le font certains systèmes commerciaux comme Trango, VLX ou Adeos), ce qui conduisait à une utilisation sous-optimale du CPU.

Généralement, on ne sait faire une allocation déterministe que lorsque les ressources sont allouées statiquement, mais ce n'est pas une obligation. OASIS est un système qui alloue le temps CPU dynamiquement en gardant une allocation déterministe. Cette propriété est obtenue par une conception système minutieuse qui permet de ne pas utiliser de « verrou ». Seul l'ordonnanceur détermine les allocations dans le système, et cet ordonnanceur est facilement remplaçable (ce que nous avons fait dans [LDAVN08, LDAVN06, Lem06]).

**Simplicité et efficacité des allocations** La seule manière connue pour réaliser des allocations déterministes est d'avoir un système entièrement statique. Cela est souvent trop rigide, complexifie le développement et résulte en des gâchis de ressources, et cette allocation statique est faite au niveau de « partitions » (e.g. [Rus98, BDRS08]).

On peut faire un constat simple : à chaque fois que des ressources sont divisées statiquement dans deux « pools » $P_1$ et $P_2$, alors on peut se retrouver dans une situation où $P_1$ est vide, $P_2$ ne l'est pas, mais où une demande allocation ne peut se faire que depuis $P_1$ et sera donc refusée. Par exemple, si on réserve statiquement de la mémoire pour la constitution de buffers réseaux, il y a gâchis de mémoire si le réseau est peu utilisé et que le système manque de mémoire en général, ou que le réseau est très utilisé mais que beaucoup de mémoire est disponible.

De plus, pour chacun de ces pools de ressources (que nous dénommons *sous-pools*), il est nécessaire de définir comment est réalisée l'allocation à l'intérieur de ce sous-pool. Cela conduit à multiplier les politiques d'allocations, complexifiant le système.

Nous proposons de réaliser un système dont l'allocation est déterministe, mais sans pour autant diviser les ressources en sous-pools statiques. Cela conduit à plus d'efficacité, plus de simplicité, et finalement à plus de sécurité puisqu'on peut appliquer une allocation déterministe à une granularité plus fine que la « partition ».

**Haute sécurité** Avoir des politiques d'allocations qui prennent en compte les besoins des ressources et des allocations qui suivent les décisions d'allocations est suffisant pour que les tâches critiques s'exécutent correctement, mais seulement en l'absence de défaillances ou malveillances. Il faut également fournir l'assurance de cette exécution correcte, ce qui passe par une haute sécurité et sûreté de fonctionnement.

En plus des problèmes classiques de confidentialité et d'intégrité des données, nous devons nous attaquer à la protection contre les dénis de service, que Gasser considère le problème le plus difficile [Gas88, p. 4]. Nous devons même nous protéger contre le ralentissement de service, qui permettrait à une tâche de faire en sorte qu'une autre puisse manquer son échéance.

Nous considérerons toute modification non autorisée de l'attribution des ressources, et tout ralentissement dans l'exécution d'une tâche, comme une faille de sécurité. Il en résulte un *système invulnérable au ralentissement de service*.

**Support des application « legacy »** Pour des raisons économiques, il est important de pouvoir continuer à exécuter des applications écrites pour des systèmes antérieurs. Nous proposons différentes solutions à cette approche, du support de la virtualisation (qui permet de réutiliser un système avec un travail minimal et une compatibilité maximale, mais sans pouvoir profiter pleinement des avantages du nouveau système) à la réécriture de librairies aux API compatibles (notre prototype implémente ainsi une partie de l'API POSIX, ce qui permet la rétrocompatibilité tout en profitant de la forte sécurité et de l'allocation déterministe.)

## 1.3 Concepts présentés

Notre conception est basée sur deux principes fondamentaux (§ 3.1).

**Haute sûreté et sécurité** Le premier est la structuration pour un haut degré de sûreté et sécurité, dont les principes ont été résumés notamment par l'article de Saltzer et Schroeder [SS74]. Le système est constitué de petits domaines de protection séparés, qui disposent d'une quantité minimale de privilèges ; les mécanismes communs à plusieurs tâches, sont spécialement minimisés. Cela permet de réduire la probabilité et l'étendue de l'impact d'un dysfonctionnement.

services Certains de ces domaines de protections, appelés *services*, sont partagés entre plusieurs programmes. Certaines opérations ne sont réalisables que par leur intermédiaire, ce qui leur permet de garantir l'absence de certains mauvais fonctionnements [Rus89]. Ces services partagés représentent les principaux angles d'attaques du système, et le système doit être conçu afin qu'ils soient bien protégés.

Enfin, la communication et l'évolution des droits d'accès des différents domaines doivent être strictement contrôlés, ce qui est réalisé par l'emploi de capacités (§ 3.3).

**Indépendance des politiques** Le deuxième est le principe nouveau d'indépendance des politiques (§ 3.1.1), qui énonce que les décisions d'allocations devraient toutes êtres prises par des modules de politiques distincts du reste du système. Un système qui permet cela permet également des allocations déterministes, et le remplacement des politiques d'allocation pour l'adapter au besoin. De plus, l'allocation est faite indépendamment de la présence des domaines de protection, ce qui contribue à l'efficacité et la simplicité de l'allocation. Ceci est réalisé par l'utilisation extensive du mécanisme de prêt de ressource.

Cette propriété permet de gagner en efficacité sur plusieurs niveaux :

1. il devient inutile de sur-réserver des ressources de manière « conservative » ;
2. il devient possible d'allouer les ressources pour coller au mieux aux besoins des tâches ;
3. les sous-pools dus à la séparation en domaines de protections sont éliminés.

**Conception du noyau et méthodologie de développement des services** Réaliser l'indépendance des politiques et obtenir une haute sécurité implique de nombreuses contraintes, sur la conception générale, celle du noyau, et celle des services. Beaucoup de détails spécifiques d'implémentation ont du être étudiés.

Une partie de ces contraintes vient du besoin du service de résister aux attaques, et d'assurer lui-même des contraintes de sécurité. Une autre partie provient du fait que l'exécution du service provient de ressources qu'ils ne contrôle pas, et peuvent disparaître à tout moment. Il faut à la fois trouver des principes pour l'écriture des services, et des mécanismes du noyau qui permettent d'assurer la sécurité des services (et du système).

En particulier, il y a un certain nombre de contraintes pour la gestion des problèmes de concurrence et synchronisation. Les services sont multithreadés par nature, et ne peuvent utiliser ni sleeplock (car cela compromettrait l'indépendance de l'ordonnancement CPU) ni masquer les interruptions (car cela compromettrait la sûreté). Il a fallu développer des techniques pour minimiser le besoin en synchronisation, et résoudre les problèmes restants. Un corollaire de ces techniques est le haut niveau de parallélisme du système, qui laisse envisager une exécution très efficace sur architectures multiprocesseurs.

**Analyse des besoins et état de l'art** Nous avons commencé par analyser de manière approfondie les besoins pour répondre à notre problématique et identifié les problèmes rencontrés par l'état de l'art et les concepts qui ressortissaient des différentes solutions proposées. En particulier, c'est là que nous avons identifié les concepts d'allocation déterministe et de prêt de ressource.

**Plan** La thèse est structurée comme suit : le Chapitre 2 présente les besoins spécifiques du sujet, identifie les concepts importants, et présente l'état de l'art en rapport avec le sujet. Le Chapitre 3 présente une vue d'ensemble de la thèse,

la structuration en domaines et les moyens de communication. Le Chapitre 4 étudie en détail les besoins particuliers en synchronisation à l'intérieur du système d'exploitation, et propose différents mécanismes de synchronisation qui répondent à ces besoins. Le Chapitre 5 présente un ensemble de principes et de techniques de conception pour la réalisation de services partagés de l'OS. Le Chapitre 6 conclut.

Chapitre 2

# Besoins, solutions actuelles et problèmes restants

Pour exécuter plusieurs tâches sur une même machine, il faut et suffit que chaque tâche puisse utiliser les ressources dont elle a besoin pour s'exécuter. Cela entraîne cependant des difficultés, que l'ont peut classer en trois sortes :

- l'*allocation de ressources*, i.e. la détermination de la quantité de ressources à attribuer aux différentes tâches. Ce problème est rendu plus complexe lorsque les ressources ne sont pas en nombre suffisant, que certains programmes ont des contraintes temporelles à respecter, et qu'ils ont des degrés d'importance divers ;
- la *sécurité*, i.e. la protection du système contre les défaillances ou malveillances de certaines tâches (ou du matériel). En particulier, il faut protéger les tâches critiques contre les erreurs des tâches non-critiques ;
- la *flexibilité*, i.e. l'adaptation du système aux différents besoins : API différentes, allocations de type différents, etc. Le système doit être facilement réutilisable dans des contextes différents, voire pouvoir exécuter simultanément des tâches dont les environnements d'exécution sont très différents.

Il y a des relations entre ces catégories de problèmes ; par exemple la question de l'ajout d'extensions concerne à la fois la sécurité et la flexibilité. Nous nous sommes particulièrement attachés aux relations entre les problèmes de sécurité et d'allocation de ressources. Par exemple le système doit protéger l'attribution des ressources aux différentes tâches ; certaines ressources doivent être partagées entre plusieurs tâches ; des dénis de service peuvent être occasionnés par une mauvaise allocation des ressources, etc.

Cette thèse se concentre sur le support d'exécution (i.e. le système d'exploitation) pour des tâches de niveaux de criticité différents. Ce chapitre en présente la problèmatique, i.e. présente et identifie les problèmes auxquels il faut apporter une solution, les concepts importants liés à ces problèmes, ainsi que les différentes réponses issues de l'état de l'art.

Nous avons choisi de structurer ce chapitre selon les différentes notions relatives aux différentes problématiques. Certaines notions étaient précedemment identifiées ; d'autres ne l'étaient pas en tant que tel mais étaient présentes dans des systèmes existants ; et certaines sont entièrement nouvelles. Pour chaque notion, nous avons donné lorsque c'était possible des exemples de systèmes dans lesquels cette notion était présente. Nous esquissons également des solutions d'ensemble pour l'exécution de tâches de criticité hétérogènes, ce qui nous permet d'identifier les problèmes à résoudre au niveau du système d'exploitation.

**Contributions** Ce chapitre apporte plusieurs contributions :

- il identifie le concept d'allocation déterministe, qui facilite l'allocation dynamique pour des tâches temps réel dur critiques ;
- il étudie l'antagonisme entre sécurité et performance pour toute politique d'allocation, et propose des solutions pour le dépasser. Il propose une technique générale pour l'ordonnancement de tâches de criticité différentes ;
- il redéfini le problème de déni de ressource sur un service partagé (identifié par KeyKOS [Key88, chap. 4]) sous la forme d'un problème d'allocation des ressources. Il identifie les deux techniques permettant de les éviter : la définition de sous-politique, ou le prêt de ressource. Le prêt de ressource permet de simplifier les politiques d'allocation et d'éviter la perte de ressources par fragmentation ;
- il établi la distinction entre la comptabilisation des ressources et le respect d'un ordonnancement, et indique les avantages de ce dernier point ;
- il catalogue et classifie les problèmes se posant pour écrire un système d'exploitation qui permette de faire cohabiter des tâches temps réel critiques avec des tâches arbitraires, et les solutions existantes associées.

## 2.1 Allocation des ressources

### 2.1.1 Notions et définitions

**Ressource** Dans cette thèse nous différencions la notion de *ressource* de celle d'*objet*. Dans le contexte d'un système d'exploitation, un *objet* est une entité identifiable dans le système. Des exemples sont les fichiers, programmes, répertoires, page mémoire, port I/O, etc. Les objets sont regroupés par *type*.

Les objets peuvent être créés à partir d'autres objets : ainsi un fichier est créé à partir d'un ensemble de secteurs de disques ; un socket réseau nécessite de la mémoire vive et un numéro de port TCP/IP. Nous appelons *ressources* les objets qui ne peuvent pas être créés à partir d'autres objets. Parmi les ressources, il y a des ressources physiques (mémoire, CPU) ou non (port TCP/IP, numéro d'interruption).

En général, le nombre de ressources est limité, ce qui pose le problème de leur répartition. Mais le nombre d'objets en général n'est limité que par le nombre de

ressources disponibles : ainsi le nombre et la taille des fichiers dépendent du nombre de secteurs disques. Inversement, lorsque par exemple le nombre de processus est limité à 1000, nous disons qu'il y a en fait 1000 ressources « numéro de processus », et que la création d'un objet processus nécessite cette ressource.

**Allocation et ordonnancement** Il y a deux manières de diviser l'accès à une ressource entre différentes tâches. La division *spatiale* permet d'attribuer des parties d'un type de ressource à différentes tâches. Les ressources sont généralement composées d'éléments identiques que l'on peut numéroter, de 0 à « nombre d'élément moins un ». Lorsqu'il n'y a qu'un seul élément, on dira par abus de langage que la ressource n'est pas divisible spatialement. La mémoire, l'espace disque, les ports TCP sont des exemples de ressources divisibles spatialement. La division spatiale est appelée *allocation*.

allocation

La division *temporelle* permet de faire varier l'allocation selon l'instant courant. Cette sorte de division est appelée *ordonnancement*, et nous appelons les changements dans l'allocation des *décisions d'allocation*. On parle généralement d'ordonnancement pour les ressources qui ne sont pas divisibles spatialement, comme le CPU, l'accès au disque, ou au réseau ; mais l'attribution de la mémoire peut également suivre un plan d'ordonnancement.

ordonnancement

Nous désignons parfois abusivement allocation à la fois la division spatiale (allocation) et temporelle (changement d'allocation). Ainsi, une politique d'allocation s'occupe à la fois de l'allocation et de ses changements.

Cette distinction est particulièrement importante quand on établira la distinction entre allocation déterministe et comptabilisation des ressources ; voir section 2.1.3.3.

**Politique d'allocation** Les différents programmes présents dans le système sont en compétition pour obtenir les ressources du système. L'OS doit "arbitrer" cette compétition en choisissant comment est faite l'allocation d'un type de ressource. Ce choix est défini par la *politique d'allocation* de ce type de ressource.

politique d'allocation

Nous définissons qu'une politique d'allocation est *statique* lorsque l'allocation n'évolue pas au cours du temps. En d'autre termes, il n'y a pas de changement d'allocation, et il n'y a donc pas besoin de code pour modifier l'allocation de la ressource[1]. L'allocation statique est plus simple, mais l'allocation dynamique permet de ne pas gâcher de ressources et permet donc d'être plus efficient : pour certains programmes (de type best-effort) la demande en ressources peut beaucoup varier [DH66]. Notons qu'il est nécessaire de faire confiance à tout code qui peut modifier l'attribution des ressources d'une tâche.

L'allocation dynamique peut se faire à la demande (c'est souvent le cas dans les systèmes à temps partagé comme UNIX [Tho78]), auquel cas les programmes font des *demandes d'allocation* ; ou selon un plan préétabli.

[1] Notons que selon cette définition, les ordonnancements sont toujours dynamiques, y compris l'ordonnancement dit statique. En effet, un ordonnancement nécessite par nature qu'il y ait modification de l'attribution de la ressource.

**temps réel** Pour certaines applications (contrôle/commande, communication...) les calculateurs sont utilisés pour interagir avec leur environnement. Les instants où les *entrées-sorties (IO)* qu'ils font sont importants et donc spécifiés. Ces spécifications de temps se reportent ensuite sur les logiciels que ces calculateurs exécutent : ce sont des logiciels *temps réel*. Un exemple important de telle contrainte de temps est l'échéance, qui est une date avant laquelle il faut réaliser un certain travail.

Lorsque la violation d'une contrainte de temps est considérée comme une défaillance grave du système, ils sont *temps réel dur* ; c'est le cas dans certains logiciels pour le contrôle-commande dans l'avionique ou le nucléaire par exemple. Un des objectifs de la thèse est l'intégration dans le système, de tâches dont certaines peuvent être temps réel dur, et de tâches au comportement non spécifié.

Certaines politiques d'allocation sont particulièrement adaptées à l'exécution de logiciel temps réel (section 2.1.3), mais il est fondamental de noter que le caractère temps réel d'une application découle uniquement de ses spécifications, pas de sa conception, de son implémentation ou de sa politique d'ordonnancement.

**Difficultés** La problématique de l'allocation des ressources consiste à évaluer les besoins en ressources des différentes applications et de distribuer les ressources selon les besoins.

Il y a de nombreuses difficultés liées à l'identification de ces besoins, en particulier du besoin en temps CPU (i.e. analyse de WCET). Nous ne nous préoccupons pas de ces aspects certes très importants, mais indépendants du système d'exploitation.

Les difficultés qui concernent le système d'exploitation sont de deux sortes. La première est le fait qu'il est difficile de modifier certains comportements du système sans que cela impacte tout le système. Ce problème est abordé en détail en section 2.4. La deuxième est le fait qu'il est difficile d'écrire un système d'exploitation qui respecte la politique d'allocation souhaitée (même si le système d'exploitation est écrit pour une politique donnée). Nous examinons les raisons pour lesquelles cela est difficile.

### 2.1.2 Systèmes généralistes

#### 2.1.2.1 Systèmes généralistes usuels

Usuellement, la méthode d'écriture de système d'exploitation est peu éloignée de celle de programmes ordinaires. Or, l'emploi de certains idiomes de programmations communs peut poser problème. En voici quelques exemples.

**Ressources non comptabilisées** Le noyau est un programme et a donc besoin de ressources ; mais étant un programme privilégié, il est possible d'allouer des ressources sans passer par l'algorithme d'allocation du noyau. Ces ressources sont soit non comptabilisées, soit comptabilisées de manière spéciale.

Par exemple, le temps passé à traiter une interruption est souvent considéré spécialement (voir par exemple les trois mécanismes de "softirq", "work queues" et "tasklets" de Linux [BC05]). Si de plus ce temps n'est pas proprement comptabilisé, le noyau est vulnérable aux "rafales d'interruptions" (interrupt storm) [MR97]. Dans

les micronoyaux, le temps passé dans un serveur à traiter les requêtes des clients est souvent mal décompté, à moins de dispositions spéciales [MST93, SWH05].

Le principal exemple de ressource comptabilisée spécialement est la mémoire, par l'utilisation de primitives de type `malloc()` dans le noyau (exemple : le `kmalloc` dans Linux [BC05]). Cet appel ne permet pas d'attribuer la mémoire allouée à un programme (la mémoire est donc attribuée au noyau). Lorsqu'un tel mécanisme est utilisé en réponse à l'appel système d'un client, cela crée un déni de ressource sur la mémoire : le client peut faire allouer au noyau autant de mémoire qu'il le veut sans être détecté, en en privant les autres programmes.

En résumé : *toutes les ressources allouées par le noyau devraient être prises en compte par l'algorithme d'allocation de cette ressource*, sinon une partie de la politique reste en dehors de l'algorithme. Cela rejoint le principe d'économie de mécanisme (§ 2.2.2.5).

**Changements imprévisibles de l'exécution** Dans les systèmes actuels, il est impossible de prévoir le temps passé dans un appel système ou les instants où une tâche est exécutée.

L'exemple le plus flagrant est le fait que l'exécution d'une tâche peut se bloquer de manière imprévisible lors d'appel systèmes. Cela peut se produire notamment par utilisation de sleeplocks, i.e. de verrous pour lesquels l'attente de la libération se fait en appelant l'ordonnanceur. Par exemple, un processus qui demande l'agrandissement de son espace d'adressage sous Linux peut se bloquer de manière imprévisible à cause d'un sémaphore de descripteur mémoire [BC05, p. 224]. Enfin, même si la tâche ne se bloque pas, le fait qu'elle puisse exécuter un traitement plus long de manière imprévisible est également problématique (e.g. l'allocation dans les slabs Linux est généralement rapide, sauf lorsque ceux-ci sont vides [BC05, p. 337] ) : ceci augmente par exemple de beaucoup l'évaluation du temps maximum d'exécution d'une tâche temps réel.

Un autre exemple est l'utilisation de primitives de communication synchrone dans un micronoyau pour communiquer avec un serveur : on ne peut pas prévoir si la communication avec le serveur va bloquer (à moins d'utiliser des threads dédiés par client dans le serveur, comme préconisé par Härtig [HHL+97, § 7.1]).

Toutes ces attentes modifient de fait la politique d'ordonnancement et sont problématiques pour l'exécution de tâches temps réel dur. *L'exécution lors d'un appel de service devrait être prévisible, et surtout ne pas se bloquer de manière imprévisible* (sauf si l'appelant permet de se bloquer de manière explicite).

**Politiques "codées en dur" et dénis de services** Les politiques d'allocation sont le plus souvent imposées et non configurables. Tout au plus peut-on "conseiller" le système, avec des fonctions comme `nice` et `madvice` de POSIX. Et on ne peut généralement pas facilement remplacer la politique, car la politique et les mécanismes sont intimement imbriqués. Au point qu'on écrit des politiques d'allocation sans s'en rendre compte : beaucoup de ressources finies ne sont pas identifiées comme telles.

L'exemple le plus classique de politique « codée en dur » est l'algorithme d'allocation *FCFS* (first come first served), i.e. premier arrivé, premier servi. On emploie en effet cette politique à chaque fois où un service fait une allocation en réponse à FCFS

la demande d'un client : par exemple en prenant le premier élément dans une liste (FCFS sur les éléments de ce type), ou en faisant un appel de type `malloc` (FCFS sur la mémoire). Cette politique a l'avantage de satisfaire tout le monde quand les ressources sont en nombre suffisant ; mais quand les ressources sont limitées, les derniers arrivés se voient refuser l'accès aux ressources (et donc au service) : c'est le *déni de ressource* [Key88, chap. 4], qui est une forme de déni de service. Il est facile pour un attaquant de provoquer cette situation, en envoyant beaucoup de requêtes.

déni de ressource

Par exemple, certains systèmes comme UNIX ne permettent l'ouverture simultanée que d'un nombre limité de fichiers, et l'allocation des descripteurs de fichiers se fait selon la politique FCFS. D'où un déni de service : certains clients se verront recevoir une erreur `ENFILE` lorsque trop de descripteurs seront ouverts (cf la page `errno` de la norme SUSv3 [Ope]). Ce problème touche aussi les file locks (erreur `ENOLCK`), les buffers IO (erreur `ENOBUFS`), ou l'allocation de ports TCP (erreur `EADDRINUSE`). Ces problèmes ne sont pas nouveaux : les « plafonds » du système CAL/TSS en sont également des exemples [LS76, § 6.3].

Une des raison du choix de FCFS est qu'il est difficile de choisir une politique d'allocation satisfaisante pour des file locks ou les descripteurs de fichiers. Imposer des limites par utilisateur est une manière de résoudre ces problèmes de déni de service, même si elle n'est pas pleinement satisfaisante (la restriction est forcément arbitraire).

*Le choix d'une politique d'ordonnancement devrait donc être explicite* et non implicite, sinon on ne pourra pas en changer l'algorithme. Cela demande que *les ressources en nombre limité soient toutes identifiées.*

**Résumé** Dans les systèmes traditionnels, les politiques d'allocation des différentes ressources ne sont pas définies de manière précises. Elles sont au contraire extrêmement dépendantes de l'implémentation, et l'allocation des ressources comme le temps CPU ou la mémoire dépend quasiment de l'intégralité du code du système d'exploitation. Les autres ressources limitées ne sont pas bien identifiées, ce qui est source de déni de services. Notons que ces problèmes se trouvent aussi bien dans les noyaux monolithiques, que dans les micro ou exonoyaux.

Nous estimons qu'il est nécessaire de définir des principes structurant qui permettent d'éviter ces problèmes, et qu'il est possible de faire cela de manière indépendante des politiques d'allocation. Cependant, il est également possible de structurer un système d'après un choix de politique d'allocation précis : c'est ce que nous voyons par la suite.

#### 2.1.2.2 Approche par cache

**Motivation** Les systèmes dynamiques permettent à de nouveaux objets noyaux, comme les domaines de protections ou les threads, d'être créés dynamiquement. L'extension de la taille d'un espace d'adressage demande également la création de table des pages. Ces opérations demandent au noyau de maintenir des données les concernant. *Si on ne donne pas de limite statique au nombre de tâches, nombre de tables de pages etc., il y a normalement un besoin d'augmentation de la mémoire accessible par le noyau.*

Or, l'allocation de mémoire (non contrôlée) par le noyau peut engendrer des dénis de ressource. Pour éviter toute allocation dynamique de mémoire, certains systèmes utilisent une approche par cache. La taille mémoire du noyau est de taille fixe, les données du noyau sont copiées ailleurs quand elles concernent des objets inutilisés, et recopiés vers le noyau quand ces objets sont réutilisés.

**Exemple de systèmes** Dans certains systèmes, l'extérieur peut être le disque dur. C'est le cas par exemple pour KeyKOS [BFF+92, §2.1] et EROS [Sha99, p. 116], [SH02], ainsi que pour CAL/TSS [LS76]. La présence d'une « object table » centrale de taille fixe associant les capacités aux objets (e.g. [Sha99, p. 103], [LS76, §3.5], [Lev84, p. 192]) encourage ce genre d'approche[2].

Sur d'autres systèmes, l'extérieur peut être géré par des applications sans confiance en espace utilisateur. C'est le cas par exemple pour le Cache kernel [CD94], ou pour certaines version de L4 [HE03].

**Problèmes** Cette approche est pratique pour des systèmes généralistes qui ont besoin de faire du swapping.

Un problème de cette approche est qu'il est typiquement très difficile de prévoir ce qu'il y a dans le cache, surtout lorsque les applications présentes ne se font pas mutuellement confiance (et ne se connaissent pas). Cela impacte fortement la prévisibilité des temps d'exécution, et cette approche n'est donc pas bien adaptée au temps réel dur. Elkaduwe et al. font une analyse similaire [EDE08, § 2.2].

L'autre problème est le coût engendré par les copies vers et hors de la mémoire, même lorsque la copie se fait vers de la mémoire vive [CD94]. Ces copies sont rares, mais prennent du temps, ce qui pose problème pour la prédictibilité des applications temps réel.

**Autre solution : le prêt de mémoire au noyau** L'allocation dynamique de mémoire par le noyau peut ne pas causer de déni de ressource : il suffit de comptabiliser cette mémoire comme allouée par l'application qui a provoqué l'allocation par le noyau. Cela demande une conception adaptée (e.g. seL4 [EDE08]). Le mécanisme de prêt de ressources (§ 2.3.2.4) est l'extension de ce principe à toutes les ressources et tous les services du système.

Notons que cette solution peut être difficile à implémenter. Si Liedtke [LIJ97] avait eu l'idée de cette technique, il n'expliquait pas comment gérer la récupération de la mémoire prêtée au noyau. Elphinstone donne différentes raisons pour lesquelles le prêt de ressource est difficile à implémenter (en particulier dans L4) [Elp], ce qui explique la solution de Haeberlen et Elphinstone pour contourner ce problème [HE03].

### 2.1.3 Support du temps réel

Dans cette section, nous nous intéressons aux approches permettant d'exécuter des tâches qui ont des contraintes temps réel dur. Nous ne nous occupons pas

[2] la section 3.3.5 explique comment se débarrasser d'une telle table.

des approches consistant à améliorer un OS généraliste pour faire du traitement multimédia ou temps réel mou (voir [MST93, Ros95, JRR97] pour des exemples de tels systèmes). Tout comme un système assurant une haute sécurité doit être pensé dans ce but dès le départ, nous considérons qu'on ne peut pas transformer un système pour qu'il puisse exécuter des applications temps réel dur. Un tel système doit être conçu spécifiquement pour fournir l'assurance que les tâches s'exécuteront dans le temps imparti.

Les systèmes conçus pour le temps réel dur (RTOS) sont généralement conçus autour du support d'un algorithme d'ordonnancement particulier, avec quelques additions pour préserver un comportement prévisible en présence de verrous [SR04]. Les ressources sont généralement allouées statiquement, et l'utilisation de ressources partagées se fait simplement par un mutex[Bak91] [3]. Nous en donnons quelques exemples.

#### 2.1.3.1 Ordonnancement par priorité

L'ordonnancement par priorité consiste à toujours exécuter la tâche prête dont la priorité est la plus grande. On distingue l'ordonnancement par priorité *fixe* de celui par priorité *dynamique* selon que les priorités des tâches peuvent ou non évoluer. Nous commençons par la priorité fixe.

L'intérêt de cette approche est qu'on peut simplement définir les interactions entre les tâches lorsqu'une d'entre elles est bloquée (e.g. l'héritage de priorité) ; cela permet de prédire si une tâche va pouvoir atteindre ses échéances même en présence de sémaphores. On peut ainsi réutiliser les techniques de conception parallèle classique pour faire du temps réel. À l'extrême, il suffit de rajouter pour un OS générique le support des priorités fixes, ce qui est simple à faire, pour pouvoir se proclamer système temps réel « mou ».

Une majorité de systèmes industriels suit cette approche [SR04], malgré ses multiples problèmes :

**Coût des analyses d'ordonnancement** Si on peut faire des analyses, elles sont souvent imprécises voire très pessimistes en présence de sémaphores, ce qui limite le taux d'utilisation du processeur et induit ainsi de mauvaises performances pour les systèmes temps réel dur [SRL90].

**Augmentation du TCB** En l'absence de quota de temps, les tâches de priorités inférieures doivent faire confiance aux tâches de priorité supérieure : si celles-ci se mettent à boucler, les tâches de priorité inférieure n'auront plus accès au CPU. Ainsi, pour une tâche donnée, toutes les tâches de priorité supérieure font partie de son TCB (le *TCB* d'une tâche est l'ensemble du code dans lequel une malfonction peut empêcher l'exécution normale de cette tâche).

TCB

C'est la raison pour laquelle on affecte aux tâches critiques une plus grande priorité, puisque les tâches non-critiques sont moins qualifiées et plus sujettes aux bugs.

[3] A tel point que les mutex sont appelées « shared ressources » dans les papiers d'ordonnancement théorique, e.g. [BF08]

**Interférences avec la politique d'ordonnancement** Mais les théories d'ordonnancement de type RMA/DMA [LL73] enseignent que la priorité doit être attribuée en fonction de l'urgence d'une tâche, non en fonction de sa criticité. La solution à priorités fixes empêche donc l'exécution correcte de tâches non-critiques, mais urgentes (un exemple d'une telle tâche pourrai être la réponse aux « acknowledges » TCP dans une pile réseau). La notion de priorité est ainsi problématique, car elle confond les notions de criticité et d'urgence[4] en un seul attribut.

On peut cependantrésoudre ce problème, en utilisant un mécanisme de budget de temps pour limiter le temps CPU utilisable par une tâche en une période (e.g. [HDF+07, II.C]).

**Non-optimalité** L'ordonnancement par priorité fixe n'est pas optimal, y compris pour des tâches très simples [LL73]. Pour un système de tâches égales, il faut donc surdimensionner le processeur lorsqu'on utilise un algorithme par priorité fixe (de l'ordre de 20% sur monoprocesseur). De plus, cette approche devient particulièrement inefficace sur systèmes multiprocesseurs, la perte de puissance CPU pouvant se situer à 50 voire 70% [And03].

Ces chiffres ne concernent que des systèmes de tâches simples (tâches périodiques), et sans la présence de lock, qui font encore baisser ce chiffre. On peut penser que l'utilisation de lock est particulièrement néfaste en multiprocesseur : quand on attend pour un lock en monoprocesseur, on peut utiliser l'héritage de priorité pour résoudre de conflit ; mais il est difficile de transposer cette notion en multiprocesseur. La tâche qui est en attente peut soit attendre en attente active (ce qui fait perdre du temps CPU), soit donner la main à une tâche de priorité plus basse en attendant que le verrou se libère. L'analyse d'ordonnancement par priorité fixe en multiprocesseur est certainement très difficile, ou utilise des approximations très grossières.

**Priorité dynamique, EDF et LLF** Devant tous ces problèmes, on se demande pourquoi on n'utilise pas des algorithmes plus efficaces, par priorité dynamique, tels qu'EDF, qui est optimal en monoprocesseur [Der74]. C'est le sujet de débats actuels [But05].

Mais l'utilisation d'EDF seul ne résout que les problèmes de performance de la priorité fixe. En particulier, cela ne résout ni le problème du coût des analyses d'ordonnancement (surtout en présence de verrous), ni la protection contre des applications qui prennent trop de temps CPU (même s'il n'est plus possible de le monopoliser comme en priorité fixe). Pour cela, il faut rajouter à EDF un mécanisme de budget de temps, comme cela est fait dans OASIS ou dans RBED [BBLB03].

L'utilisation de priorité seule ne peut donc pas résoudre le besoin de protection temporelle. Des ordonnancements pour ce cas sont étudié ci-après.

#### 2.1.3.2 Ordonnancement par réservation

La *réservation* consiste à préserver une partie du temps CPU pour une ou plusieurs tâches données. L'idée est de prédire à l'avance que ce temps CPU est suffisant pour

[4] Baruah et Fischer notent également ce problème [BF08]

l'exécution de ces tâches, ce qui garantit l'exécution correcte de la tâche (si tant est que le temps CPU réservé est effectivement alloué, et que les tâches disposent également de toutes les autres ressources dont elles ont besoin). La réservation ne s'applique pas qu'au temps CPU, mais à toute autre ressource : mémoire, etc.

Elle n'est pas utile seulement dans un cadre temps réel : les serveurs ont aussi typiquement besoin de contrôler l'usage de leurs ressources. Les VMMs sont entre autre utilisés pour pallier au fait que les systèmes généralistes utilisent FCFS et ne savent pas faire de réservation.

**Ordonnancement statique** On appelle *ordonnancement statique* un ordonnancement qui se répète de manière cyclique. Outre le fait que le temps CPU non utilisé est ainsi gâché, et que les tâches sont contraintes à être cycliques, ce mode de fonctionnement contraint fortement l'utilisation des ressources IO. Par exemple, ce système demande une préemption immédiate qui empêche de partager les autres ressources avec un sémaphore. Cela conduit à construire un ordonnancement à la main pour partager l'usage des ressources, long et fastidieux. Pour éviter cela, les autres ressources sont également partagées de manière statique.

L'avantage de ces systèmes est leur extrême simplicité, ce qui permet de les faire facilement certifier (bien qu'ils soient difficiles à réaliser). L'allocation de toutes les ressources est *déterministe* , dans le sens où l'allocation de ressource effective est égale à l'allocation de ressource prévue par l'ordonnanceur. Le noyau est relativement simple, et il peut même ne pas y avoir d'appel système nécessaire.

Mais c'est un gâchis de ressources (surtout mémoire et temps CPU), surtout quand l'usage en ressources peut beaucoup varier (fragmentation interne). De plus, toutes les tâches sont contraintes à être exécutées selon un multiple du cycle ; et il faut découper l'exécution des tâches manuellement, ce qui est pénible pour le concepteur. De plus, cette allocation pose problème lorsque certains I/O sont partagés (§ 2.3.2). Ce type d'allocation est donc souvent réservé aux systèmes les plus critiques (en sécurité ou pour le temps réel). Notons que l'avionique l'utilise également pour séparer les différentes partitions [Rus98, Aer].

Certains systèmes sont conçus poru faciliter le calcul de cet ordonnancement statique, comme MARS [KDK+89, p. 32]. AUTOSAR implémente une version moderne de l'ordonnancement statique, avec la notion de « schedule table », qui permet d'envoyer des notifications à des applications selon un plan cyclique statique [HDF+07, II.E].

Il est cependant possible de faire de l'ordonnancement dynamique en ayant une garantie de sécurité aussi forte que pour l'ordonnancement statique. C'est notamment une des raisons du développement d'OASIS [AD98].

**Ordonnancement par bande passante de CPU** Comme la notion de priorité ne fournit aucune garantie de sécurité, une partie de la recherche s'est focalisée sur l'allocation par division de bande passante. L'idée est de fournir à chaque application un certain pourcentage du temps CPU en fonction de ses besoins. Cette idée a été introduite par Mercer et al. [MST93]. Un ordonnanceur hiérarchique pratique basé sur cette idée est présenté par Goyal et al. [GGV96].

La notion de pourcentage de temps CPU n'est cependant pas bien adaptée au temps réel. La granularité de l'allocation est en effet importante (e.g. donner 10 secondes toutes les 1000 secondes est bien différent de donner 10 millisecondes toutes les secondes). Les quantas de temps doivent donc être donnés en fonction des échéances des applications.

L'abstraction passe dans ce cas d'un simple pourcentage de temps CPU à un couple $(e, p)$ où on garantit aux tâches qu'elles recevront le CPU pendant $e$ toutes les $p$ secondes [JRR97, BBLB03, RJMO98]. Les échéances sont prises en comptes, ce qui permet à cette approche d'être utilisée pour des tâches temps réel.

Bien que cette abstraction soit indéniablement meilleure que l'utilisation de priorité fixes, il reste des problèmes. Tout d'abord, le fait qu'on continue à utiliser le même modèle de programmation pour le système d'exploitation et les applications (en particulier les verrous) imposent toujours une analyse complexe, moins étudiée que celle pour la priorité fixe. Enfin, toutes les applications ne rentrent pas dans cette abstraction, et il est toujours contraignant de forcer toutes les tâches temps réel à rentrer dans des conteneurs périodiques.

#### 2.1.3.3 Allocation déterministe

**Analyse résumée des solutions présentées** Si on analyse ces systèmes, on se rend compte que leur but est de réaliser une allocation des ressources qui soit prévisible. C'est à dire qu'il faut pouvoir prédire comment les ressources seront allouées aux tâches critiques afin de pouvoir affirmer qu'elles respecteront leurs échéances.

Mais elles réalisent cela de la manière suivante. Tout d'abord, la quasi totalité des ressources est partagé de manière statique (l'exception notable étant le CPU, qui ne peut pas être partagé statiquement dès lors qu'il y a plus d'une tâche). Cela simplifie drastiquement l'OS et est sécurisé, mais au prix d'un sacrifice sur la flexibilité et la performance. De plus, cela permet de facilement prédire la consommation en ressources de l'OS.

Le deuxième point est l'utilisation systématique de verrous, pour partager des ressources. Cela conduit à des problèmes de type inversion de priorité, qui sont résolus par des « protocoles » spécifiant comment l'ordonnancement doit être modifié pour éviter ces problèmes (e.g. [Bak91]). Mais cela conduit à des analyses d'ordonnancements très pessimistes, et qui peuvent également être très complexes. Cette solution conduit à utiliser l'ordonnancement à priorités fixes qui permet de faire ces analyses en pratique, bien qu'étant non-optimal.

Enfin, à cause de l'allocation statique des ressources ou du désir de rester prévisible malgré l'utilisation de verrous, il est difficile de permettre aux tâches temps réel dur critiques d'utiliser des fonctionnalités complexes, comme l'emploi d'une pile TCP/IP.

**Proposition : l'allocation déterministe** Nous proposons à ce problème une réponse conceptuellement simple : l'*allocation déterministe*. Nous énonçons qu'*une allocation est déterministe lorsque l'allocation effectivement obtenue est égale à*

allocation déterministe

*l'allocation prévue*. Un contre exemple est l'utilisation dans l'OS de sleeplock, qui modifient l'ordonnancement de manière « imprévisible ».

module de politique

Le terme « prévu » ne signifie pas « entièrement prévu » à l'avance, comme l'est l'allocation statique. Elle signifie « prévu par le *module de politique* », qui est l'entité en charge de l'allocation pour une ressource donnée. Ainsi, l'allocation déterministe signifie que *toutes* les décisions d'allocations soient prises par le seul module de politique. L'allocation est indépendante du reste de l'OS.

Dans un OS classique, les décisions d'allocations peuvent être prises dans des endroits divers. Par exemple, dans Linux l'allocation de mémoire au noyau peut être déclenchée par le code de gestion du réseau. L'exemple le plus courant est l'utilisation de sleeplock dans le noyau, qui provoquent une décision d'ordonnancement à des endroits inattendus.

La majorité des ressources sont classiquement allouées statiquement, et il n'y a donc aucune décision d'allocation de prise. Mais cela ne peut pas être le cas pour le temps CPU. La solution classique de spécification de « protocole » contre l'inversion de priorité consiste en fait à spécifier dans la politique d'ordonnancement que faire en présence de ces locks. Cela permet un ordonnancement *prévisible*, mais pas *déterministe*.

Notre proposition, quand appliquée au temps CPU, consiste tout simplement à éliminer ces sleeplocks. Il n'y a donc plus d'inversion de priorité. Cette solution n'est pas utopique : OASIS est un système qui fait de l'ordonnancement dynamique (avec EDF), qui permet de réaliser des systèmes complexes aux tâches communiquantes, et cela sans nécessiter un seul verrou [5]. Le chapitre 5 contient des techniques et méthodes de conception pour y arriver.

On peut faire un parallèle entre cette proposition de centraliser les décisions d'allocation dans un module de politique avec le principe de médiation complète (§ 2.2.2.2) pour la sécurité, pour laquelle les droits de communications sont centralisées dans le « référence monitor ».

Les allocations déterministes connues sont réalisées quasiment exclusivement lorsqu'on utilise un système statique (i.e. dont toutes les ressources sont allouées statiquement, sans création dynamique de tâches etc.). Les systèmes statiques signifie un OS minimaliste, qui n'interfère donc pas dans l'allocation des ressources. Il se pose alors la question de savoir comment avoir des politiques d'allocation qui sont indépendantes des interactions de la tâche avec l'OS. Cela est résolu par l'usage général du prêt de ressource (décrit section 3.4), et des principes de conception des services partagés (e.g. pas de sleeplock).

**Avantage de l'allocation déterministe** Les avantages de l'allocation déterministe sont nombreux. Comme toutes les décisions d'allocation sont prises par un module de politique indépendant, la politique d'allocation des ressources est facilement lisible. En écrivant un module de politique adapté, on peut vérifier simplement que les tâches disposeront de ressources correspondant à leurs besoins exprimés.

[5] Les travaux de Massalin [Mas92] et Greenwald [GC96] avaient également montré qu'il était possible de réaliser des OS de manière lock-free, mais utilisaient tous les deux l'instruction DCAS qui ne se trouve que sur peu de processeurs

Pour l'ordonnancement CPU, en évitant complètement le problème d'inversion de priorité, on simplifie drastiquement l'analyse d'ordonnancement. Cela permet d'utiliser des algorithmes d'ordonnancement mieux adaptés aux problèmes à résoudre que l'ordonnancement statique ou par priorité fixe, donc des performances accrues (i.e. plus de tâches peuvent être exécutées). En particulier, les analyses pour algorithmes d'ordonnancements multiprocesseurs ne savent pas prendre en compte les verrous en général (e.g. [And03]). En n'ayant pas besoin de prendre en compte les verrous, on peut exécuter les tâches sur multiprocesseur de manière très performante [LDAVN08, LDAVN06, BCPV96].

Pour les autres allocations, il devient possible d'allouer les ressources dynamiquement de manière aussi sécurisée que l'allocation statique. Cela offre un avantage certain en terme de performance.

De plus, en mettant toutes les décisions dans un module de politique indépendant, cela facilite naturellement le remplacement des politiques d'allocations du système, qui peut être adapté aux besoins des applications.

Notons que l'allocation déterministe est également importante pour la sécurité, car l'indéterminisme est souvent une source de canal caché : une application peut influencer l'allocation des ressources pour transmettre des informations.

Finalement, le seul désavantage de l'allocation déterministe est sa réalisation : comment structurer un système pour y arriver ? C'est l'une des principales question à laquelle répond cette thèse.

Dans le chapitre suivant, nous voyons des réalisations dont certaines profitent du fait que l'allocation est déterministe.

**Allocation déterministe contre comptabilisation des ressources** Beaucoup de systèmes supportent le temps réel (mou) par la comptabilisation des ressources. Nous souhaitons établir une distinction entre cette notion et celle d'allocation déterministe.

Dans les systèmes généralistes, l'allocation n'est pas fixée à l'avance mais évolue au grès des demandes des applications. On utilise un mécanisme séparé, la *comptabilisation des ressources*, pour enregistrer la consommation en ressource de chaque tâche. Cette comptabilisation est souvent utilisée en entrée des algorithmes d'allocation (pour assurer une équité d'allocation), mais est un mécanisme séparé.

Pour les ressources partagées essentiellement spatialement, comme la mémoire, il n'y a pas de grande différence entre l'allocation déterministe et le comptage des ressources utilisées. Quand l'application nécessite d'allouer de la mémoire, la comptabilisation incrémente la mémoire utilisée. On peut passer à une allocation déterministe en rajoutant simplement à la comptabilisation la vérification que la demande ne dépasse par la quantité pré-allouée[6].

Pour le partage temporel des ressources (dont le CPU), la situation est bien différente. La comptabilisation du temps CPU utilisé intervient nécessairement *après* que le temps CPU ait été alloué à une tâche ; un ordonnancement déterministe doit au contraire limiter le temps CPU *avant* qu'elle soit allouée (en paramétrant

[6]Notons cependant que le partage des ressources (comme les librairies dynamiques, la mémoire partagée lors d'un fork, le copy on write) rend difficile de comptabiliser la mémoire précisément dans ces systèmes : toute la mémoire n'est pas forcément bien décomptée.

un timer pour que l'ordonnanceur se fasse réveiller à ce moment là). De plus, la comptabilisation de temps CPU ne peut qu'aggréger l'information sur le temps utilisé (e.g. temps d'exécution depuis le lancement de la tâche), alors que l'ordonnancement temps réel est également concerné par les moments précis où une tâche est exécutée.

Enfin, la plupart des systèmes ne savent même pas comptabiliser le temps CPU de manière exacte. Outre l'utilisation d'échantillonage pour se renseigner sur l'utilisation de temps CPU, qui rend d'ailleurs le système vulnérable à certaines attaques [TEF07], certaines activités réalisées pour le compte d'une tâche ne sont souvent pas bien décomptée (la section § 2.3.2 s'étend sur ce sujet).

En modifiant l'algorithme d'ordonnancement du système en un algorithme par priorité fixe, et en ayant une comptabilisation du temps CPU relativement précise, on peut permettre à ces systèmes de faire du temps réel mou ; *mais cela ne suffit pas pour faire du temps réel dur*. Les systèmes pour le temps réel dur doivent être conçus spécialement pour permettre des ordonnancements déterministes. Notons que dans ce cas, la comptabilisation du temps CPU est possible, mais non nécessaire.

### 2.1.4 Dépasser le compromis sécurité/performance

Une fois admise la possibilité d'allocation déterministe, on peut changer la méthodologie de développement de système temps réel. Il ne s'agit plus de réaliser un système selon des abstractions proposées, puis d'en faire l'analyse ; mais au contraire de d'abord regarder les besoins des tâches, puis de trouver comment allouer les ressources aux mieux pour prendre en compte ces besoins. Cela permet d'être sûr (i.e. garantir que les tâches critiques ont assez de ressources pour s'exécuter), et performant (en évitant de gâcher des ressources par une allocation inadéquate). Nous déclinons dans cette section les possibilités que cela ouvre.

#### 2.1.4.1 Allocation selon les besoins

**Conception classique de programmes temps réel** La quasi totalité des travaux sur le temps réel présument que les tâches sont périodiques au sens strict (i.e. qu'à chaque multiple de la période $p$ il faille exécuter un travail de durée $e$ avant le début de la période suivante) ou sporadiques (i.e. qu'il y ait un temps minimum $p$ entre les évènements qui déclenchent des travaux de durée $e$, et que chaque travail aie un temps $p$ pour être terminé). Ceci explique les approches par priorité ou réservation que nous avons vu précédemment.

**Problème des modèles de tâches trop simplifiés** Il est généralement cru que se restreindre ainsi à des tâches simples facilite l'analyse et la conception d'applications temps réel dur. Nous pensons qu'il n'en est rien : pour ces applications, les besoins des tâches sont connus précisément (ils doivent l'être si on veut pouvoir garantir que les exécutions se termineront à temps). Si les approches précédentes sont convenables lorsque les besoins sont mal connus ou pas impératifs (cas des tâches best-effort et temps réel mou), elles ne permettent pas de prendre en compte les besoins qui doivent être spécifiés de manière plus précise.

Par exemple, une tâche périodique permet aux travaux d'être exécutés soit à la fin, soit au début de la période de la tâche. Cette variation d'exécution est appelée *jitter*. Les techniques conventionnelles de conception de programmes temps réel consistent à analyser le jitter après que le système soit conçu [But05]. Mais l'analyse est complexe, et comment corriger le système si le jitter est trop grand ? Au contraire, spécifier le jitter dans la tâche (e.g. en utilisant des tâches périodiques dont l'échéance est plus petite que la période) apporte une solution simple à ce problème, et l'analyse reste faisable.

**Modèles plus expressifs** La solution consiste donc à rapprocher les contraintes à l'exécution des tâches de leurs spécifications, tout en conservant la possibilité de faire des analyses de faisabilité de l'ordonnancement. Une possibilité est d'étendre les modèles de tâches périodiques et sporadiques usuelles pour permettre davantage d'expressivité des besoins des applications (e.g. systèmes multipériodiques ou multiframes [BCGM99]). Mais cela n'est pas suffisant pour prendre en compte les tâches les plus complexes, spécifiés par exemple sous la forme de « timed automata » [AD94] et qui ne sont pas cycliques.

**OASIS** OASIS [AD98, DAL$^+$04, CDA$^+$05]est une méthodologie de développement et une chaîne d'outils originale pour la conception et réalisation de systèmes temps réel dur. La conception d'applications avec OASIS apporte des avantages significatifs par rapport aux techniques de conception habituelles.

Nous pensons ainsi qu'OASIS représente un bon équilibre entre possibilité d'analyse et expressivité des tâches. La seule contrainte sur les tâches OASIS est que toutes leurs spécifications doivent être exprimées relativement au temps (les tâches sporadiques, qui sont cadencées par les évènements arrivant dynamiquement, ne rentrent donc pas dans ce modèle). Cela nous a permis de montrer en particulier qu'OASIS était théoriquement très efficace pour le multiprocesseur [LDAVN06, LDAVN08].

De plus, les spécifications fines des contraintes temporelles permettent d'éviter les problèmes de la programmation classique (on évite ainsi l'utilisation de verrou grâce à la possibilité de synchroniser par le temps). Enfin, la chaîne d'outils intégrée permet de continuellement vérifier que les contraintes à l'exécution sont vérifiées.

En résumé, nous estimons donc important de connaître au mieux les besoins en ressource des tâches à exécuter pour améliorer l'efficacité de l'ordonnancement. Stankovic et Ramamritham sont également partisans de cette approche [SR89, § 2]. Cela est en contraste avec l'approche usuelle consistant à faire rentrer les tâches dans un modèle prédéfini. Cette approche est encore plus justifiée dans le cadre de tâches qui ne se font pas confiance, comme le démontre la section suivante.

#### 2.1.4.2 Ordonnancement sûr et performant de tâches de différents niveaux de criticité

**Notion de performance** Pour un ensemble d'applications temps réel dur, la performance peut être définie comme la possibilité de respecter toutes les contraintes de toutes les applications à faire tourner. Cela définit bien la performance, car si

un système est performant, on peut soit lui accorder moins de ressources (moins de mémoire, moins de CPUs ou CPUs moins puissants) ou lui faire tourner plus de tâches. Le but du système est de respecter toutes les contraintes de temps, pas de terminer longtemps avant chaque échéance.

**Problèmes des priorités** Selon cette notion de performance, les systèmes pour lesquels la priorité des tâches est accordée selon leur criticité (i.e. les tâches critiques sont prioritaires sur les tâches non-critiques) ne sont pas performants. En effet, une tâche non-critique mais urgente, peut être préemptée au profit d'une tâche critique non-urgente. Ainsi, le système peut ne pas satisfaire toutes les échéances alors qu'elles auraient pu toutes l'être.

Pour que le système soit performant, il faut pouvoir exécuter en priorité des tâches moins critiques (en particulier les plus urgentes ; les algorithmes EDF [LL73] et LLF sont ainsi tous deux optimaux en monoprocesseur [DM89] en exécutant les tâches les plus urgentes, bien qu'ils aient tous deux une notion différente de l'urgence). Pour le rendre sûr, il suffit de lui adjoindre un système de contrôle des temps d'exécution (i.e. vérifier qu'une tâche ne s'exécute pas plus que le temps maximal prévu dans l'analyse).

**Algorithme sûr et efficace** Pour que le système soit sûr, il faut pouvoir garantir que toutes les tâches (en particulier les tâches critiques) peuvent correctement s'exécuter. Ainsi pour être à la fois sûr et efficace, il faut *permettre aux tâches non-critiques mais urgentes de s'exécuter prioritairement sur les tâches critiques, tant que subsiste la garantie que les tâches critiques peuvent s'exécuter à temps.* La Figure 2.1 présente une exécution basée sur cet algorithme.

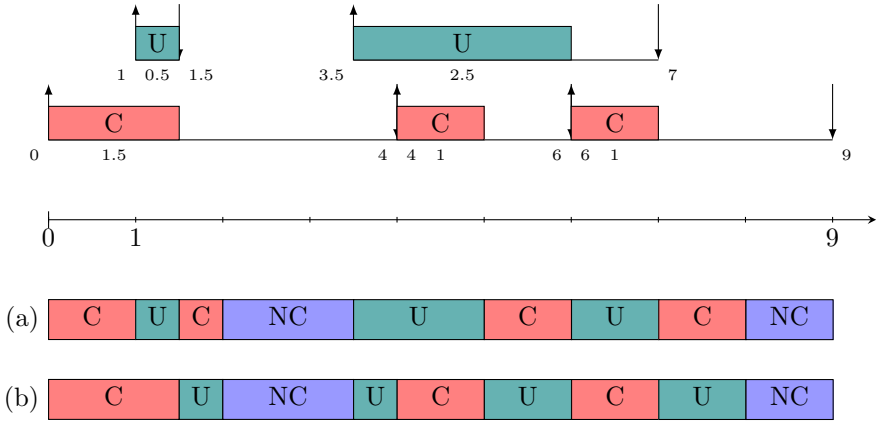



FIG. 2.1 – Exécution de tâches critiques (C), non critiques urgente (U) et non-critiques non-urgentes (NC), selon deux algorithmes : (a) en ayant les tâches non-critiques urgentes prioritaires sur les tâches critiques, sauf lorsque ces dernières ne peuvent plus terminer à temps (b) en ayant les tâches critiques prioritaires sur les non-critiques (ou par allocation statique). Les lignes du haut présentent les besoins temporels des tâches (date de début, date de fin, temps d'exécution). Les échéances des tâches non-critiques urgentes ne sont atteintes que dans le cas (a).

On peut modifier n'importe quel algorithme d'ordonnancement pour permettre cette garantie : il suffit qu'un « oracle » provoque une décision d'ordonnancement dès qu'il est possible qu'une application critique manque une échéance[7]. Si l'oracle est *exact*, i.e. ne provoque une décision que si nécessaire, cette modification n'a aucun impact sur les performances : un algorithme optimal (i.e. qui produit un plan qui respecte toutes les échéances) reste optimal ainsi transformé. En effet, si l'algorithme est optimal, il produit un plan pour lequel il est continuellement possible d'exécuter toutes les tâches (et *a fortiori* les tâches critiques), donc la présence de l'oracle ne modifie pas l'algorithme.

En d'autres termes, en ajoutant un oracle on peut rendre n'importe quel algorithme d'ordonnancement sûr ; et plus cet oracle est précis plus l'algorithme est performant. Notons que l'oracle sera d'autant plus précis qu'il en connaît plus sur les tâches critiques. La pire imprécision est d'exécuter en priorité les tâches les plus critiques, qui revient à ce que l'oracle préempte toujours les tâches non critiques au profit des tâches critiques. La meilleure précision n'a aucun impact sur les performances[8].

**Applications** Par exemple, si un système doit exécuter des applications best-effort d'un système UNIX-like, et des applications critiques, les applications peuvent être exécutées selon ces priorités :

$$\text{non-critique urgent} \quad >^* \quad \text{critique} \quad > \quad \text{non-critique non-urgent}$$

où $>^*$ signifie « plus prioritaire tant que le critique pourra terminer à temps » et où le caractère urgent et non-urgent des applications est choisi par le système UNIX (par exemple, les interruptions du système, ou les applications interactives ou d'une certaine priorité). Tant que les applications non-critiques urgentes sont peu nombreuses, elles s'exécuteront en priorité sur ce système et il n'y aura donc pas d'impact sur la performance (e.g. le throughput d'un serveur web ne s'écroulera pas parce que les « acks » ne sont pas reçus à temps). Les applications critiques peuvent être ordonnancées entre elles avec n'importe quel algorithme, e.g. EDF. La Figure 2.1 présente cette application, où le critique pourrait être une tâche OASIS.

Une autre application est l'ajout d'un oracle à l'algorithme optimal EDF. L'ajout de quota de temps à EDF est un exemple de tel oracle, comme cela est fait dans RBED [BBLB03] ou OASIS.

Notons que l'oracle doit en pratique faire une analyse de faisabilité sur les tâches critiques de manière continue ; or cette analyse a généralement une complexité assez forte, sauf pour des tâches assez simples. Il faut donc faire des approximations de cette analyse pour que le temps de calcul soit peut important, sans perdre trop de précision.

[7] on peut faire le parallèle avec la notion de laxité : la laxité représente le temps où on peut ne pas exécuter une tâche, avant qu'on soit obligé de l'exécuter si on veut qu'elle respecte son échéance. Cet oracle calculerait donc une laxité pour le groupe des tâches critiques, et préempte le processeur au profit des tâches critiques quand cette laxité est nulle.

[8] mis à part le temps de calcul nécessaire à l'oracle, qui peut être important s'il est précis

#### 2.1.4.3 Politique d'allocation : soit inefficientes, soit insécurisées

L'idée selon laquelle il faut faire passer les applications moins importantes en priorité si nécessaire peut être généralisée en cette constatation : *Si les besoins en ressources des tâches ne sont pas parfaitement connus, une politique d'allocation est soit inefficiente, soit insécurisée.*

Imaginons que $B$ soit une tâche importante, et qui puisse avoir besoin d'une ressource $r$ pour s'exécuter. Une autre tâche $A$, moins importante, a également besoin de $r$ pour s'exécuter. La politique peut alors :

1. Soit accéder aux besoins de $A$, et lui accorder $r$. Mais alors, $B$ ne peut plus accéder à $r$ s'elle en a besoin (à moins que l'on puisse immédiatement préempter $r$ au profit de $B$). Elle est donc insécurisée.

2. Soit refuser la demande de $A$ pour la réserver à $B$. Mais si $B$ n'avait pas besoin de $r$, $A$ a été privé d'exécution pour rien.

La politique FCFS est l'application systématique du choix 1, tandis que la réservation statique de ressource choisit toujours le cas 2. Mais toute politique entre ces extrêmes peut prendre une mauvaise décision, sauf :

- si $r$ peut être immédiatement préemptée au profit de $B$, ou indéfiniment partagée (i.e. $r$ n'est pas exclusive) ;

- si les besoins de $B$ sont exactement connus, comme dans l'algorithme que nous avons présenté.

Sur les systèmes généralistes habituels, rien n'est connu du comportement des tâches. L'algorithme FCFS est utilisé puisque c'est le plus efficace : toutes les ressources sont ainsi utilisées. Cependant, c'est également le moins sécurisé : les derniers arrivés sont tous privés de ressource, de manière assez arbitraire. Il est simple d'organiser un déni de service sur un type de ressource ainsi alloué : il suffit de faire une demande d'allocation répétée pour ce type de ressource[9] En pratique, des techniques sont utilisées pour cacher ce problème, comme le swapping, ou le fait de tuer une tâche [BC05, chap. 17].

Ce résultat démontre en particulier que l'algorithme parfait n'existe pas : le seul moyen d'améliorer à la fois les performances et la sécurité, c'est d'améliorer la connaissance des besoins des tâches exécutées. En particulier, il faut sortir du modèle conceptuel de programme où les ressources par processus sont virtuellement infinie [Ros95, p. 12].

#### 2.1.4.4 Conclusion sur l'allocation de ressources

Nous avons vu dans cette section que les OS généralistes avaient généralement des politiques d'allocation imprévisibles, ce qui ne leur permettait pas d'exécuter des

[9] La technique consistant à limiter le nombre de ressource par tâche peut être facilement contournée en créant une nouvelle tâche par demande d'allocation ; il est difficile de se protéger contre les dénis de services distribués, comme les « fork bomb ».

tâches temps réel dur critiques. La conception classique pour éviter cela est de construire un système d'exploitation temps réel dédié, un système dont les fonctionnalités sont simplifiées pour que l'allocation des ressources devienne prévisible. Généralement ils sont construits autour d'une politique d'allocation fixée, souvent allocation statique pour toutes les ressources et par priorité pour le temps CPU. La méthodologie standard consiste à écrire des tâches conformément à la politique d'allocation fournie par l'OS, puis d'analyser les tâches pour vérifier qu'elles s'exécutent bien à temps. Ces analyses sont toutefois très complexes, notamment à cause de la présence de sémaphores ou sleeplocks.

Notre proposition est de rendre l'allocation déterministe, c'est à dire de faire en sorte que l'allocation prévue par une politique soit égale à l'allocation effective. C'est à dire que tous les changements d'allocations sont décidés uniquement par un module de politique. En particulier pour l'ordonnancement, cela signifie que l'OS ne peut pas bloquer les threads de manière imprévisible (e.g. pas de sleeplock). Cette approche permet de simplifier drastiquement les analyses d'ordonnancement.

L'allocation déterministe ouvre de nombreuses possibilités. En s'appuyant sur la connaissance des besoins en ressource des applications critiques (nécessaire pour démontrer qu'elles atteignent leurs échéances), on peut obtenir des politiques d'allocations pour les niveaux de sécurité hétérogène qui sont à la fois sûres et efficace. Notons que cette approche a un autre avantage : la connaissance des besoins temporels des tâches a priori est indispensable pour faire un ordonnancement optimal de tâches temps réel en multiprocesseur [DM89, HL92]. Cette approche nous a permis de développer des algorithmes performants pour l'exécution temps réel multiprocesseur [LDAVN08, LDAVN06]. Nous estimons ainsi que même si EDF est optimal sans connaître les besoins temporels d'une application, l'utilisation de ces besoins est nécessaire pour la sûreté et/ou l'efficacité de l'allocation.

Mais ces considérations dépendent de la politique d'allocation choisie. Cette thèse se concentrera essentiellement sur les moyens de réalisation d'une allocation déterministe, tout en permettant des fonctionnalités complexes et une allocation des ressources dynamique.

## 2.2 Sécurité

Avec une allocation des ressources adaptée, on peut garantir aux tâches qu'elles disposeront d'assez de ressources pour s'exécuter. Mais cela ne suffit pas : il faut également garantir qu'elles pourront s'exécuter correctement, sans être affectées par des défaillances ou malveillances d'autres tâches ou de l'OS. C'est l'objet de cette section.

### 2.2.1 Besoins en sécurité

La notion de sécurité est une notion vaste et générale, qui dépend de ce qu'on cherche à protéger. Nous présentons dans cette section ce que nous entendons précisément par sécurité, i.e. quelle protection on cherche à assurer.

**Isolation** Le but de cette thèse est l'intégration de différentes tâches au sein d'un même système, qui peuvent être liées ou indépendantes. Le système doit être structuré de manière à ce que les défaillances d'une tâche soient contenues et n'affectent pas les autres tâches[10]. Cela est réalisé par l'*isolation*[11], i.e. le fait qu'un programme ne peut pas en affecter un autre (sauf par le biais de canaux de communications bien définis). Il faut en particulier protéger les programmes critiques des programmes non-critiques. Mais il ne faut pas assurer seulement cette protection [Rus98, p. 6] : on doit protéger les tâches critiques entre elles, et si possible les tâches non-critiques entre elles. La présence de tâches non-critique implique seulement qu'on ne peut pas faire confiance à l'ensemble des tâches, comme c'est le cas pour beaucoup de systèmes temps réel. Ce que l'on appelle sécurité est ainsi l'assurance du respect de cette isolation, malgré la présence de tâches malicieuses.

**Sûreté de fonctionnement et tolérance aux fautes** Un autre but de cette thèse est de réaliser des systèmes *sûrs de fonctionnement*, i.e. qui fournissent une *garantie de bon fonctionnement* du système.

Un système peut défaillir à cause de la présence de *fautes*, permanentes ou transitoires. Les systèmes sont généralement trop complexes pour qu'on puisse les réaliser sans faute. Et même si des techniques comme la preuve de programme permettent de tendre vers le zéro-défaut, cela n'empêchera pas le système de défaillir si le processeur commet lui-même une faute (par exemple à cause du vieillissement). La sûreté de fonctionnement n'est pas l'absence de fautes, mais la tolérance aux fautes [Den76, p. 361]. Nous ne nous interesserons donc pas aux techniques d'élimination de fautes ; tout au plus nous suivons des « bonnes pratiques » pour écrire du code sûr (e.g. [Ber07]).

Nous nous concentrons sur les mécanismes de l'OS pour permettre la tolérance aux fautes. Ce qui nous concerne majoritairement sont les mécanismes de confinement des erreurs [Den76] (et donc d'isolation) : on souhaite structurer l'OS de manière à minimiser l'impact des erreurs. La détection des erreurs se fait lors de la tentative de viol de confinement (i.e. tentative de faire une action non autorisée, comme lire une case mémoire non mappée).

Mais nous excluons du champ de cette thèse les mécanismes de détection et recouvrement d'erreurs tels que la redondance modulaire triple (TMR), les codes correcteurs d'erreur etc., qui peuvent être implémentés sans grand support de l'OS. Les seules mesures de recouvrement qui nous intéressent sont les possibilités de supprimer une tâche et récupérer les ressources associées ; et la possibilité d'en créer une nouvelle. En d'autres termes, nous nous intéressons seulement aux mécanismes de l'OS pour permettre le recouvrement d'erreur, mais pas aux techniques de recouvrement elles-mêmes : le recouvrement doit pouvoir être implémenté par l'utilisateur de l'OS, sans que l'OS ne force une méthode de recouvrement particulière.

**Sécurité de l'information** Ce qu'on entend par sécurité est le plus souvent la sécurité de l'information sur les systèmes informatisés. La sécurité de l'information

[10] à moins qu'une tâche dépende de la bonne exécution de la tâche défaillante
[11] Rushby [Rus98] appelle cela partitionnement (partitioning)

est un but premier de la thèse, car elle est très fortement reliée à la notion d'isolation.

La sécurité de l'information ne semble à l'origine n'être qu'un objectif secondaire pour la problèmatique de cette thèse. Elle semble n'être utile que pour des systèmes qui ont un besoin fort en isolation mais où la sécurité de l'information est également très importante, comme les téléphones portables connectés à Internet.

Il y a cependant beaucoup de points communs entre la sécurité de l'information et l'isolation (voir e.g. [Lin76, Den76]). Cela va permettre de réutiliser les résultats des systèmes conçus pour protéger l'information, et d'assurer simultanément isolation et protection de l'information sans travail supplémentaire. En résumé la sécurité de l'information cherche à protéger l'accès à l'information, tandis que nous cherchons à protéger l'accès aux ressources du systèmes. La différence principale est que l'information est duplicable tandis que les ressources ne le sont pas. La sécurité de l'information devient ainsi finalement un élément important de la thèse.

La sécurité de l'information demande le respect de trois propriétés : l'intégrité, la confidentialité, et la disponibilité de l'information.

L'intégrité est l'assurance que l'information n'a pas été altérée ; en particulier un des moyens de l'assurer est d'empêcher la modification de l'information sans autorisation[12]. Similairement nous cherchons à protéger les ressources et les tâches de modifications sans autorisation.

La confidentialité est l'assurance que l'information ne peut pas être accédée sans autorisation. La confidentialité ne semble a priori pas importante pour notre but de respect de l'isolation. Cependant, comme le note Rushby [Rus98, p. 42], le fait que deux tâches puissent communiquer par un canal non autorisé montre qu'elles peuvent interférer l'une avec l'autre[13]. Ainsi, l'identification des canaux de communication permet de vérifier que l'isolation est effective. De plus, les techniques assurant la confidentialité de l'information sont souvent les mêmes que celles pour empêcher la modification de l'information [Gas88, p. 4].

La disponibilité est l'assurance que l'information peut être accédée quand un utilisateur autorisé en a besoin. Le contraire de la disponibilité est le déni de service. Comme le note Gasser, la protection contre le déni de service est le problème de sécurité le plus compliqué [Gas88, p. 4]. En présence de temps réel, la protection contre les dénis de service a des contraintes supplémentaires.

**Déni de service et protection du temps réel** Le *déni de service* consiste à empêcher l'exécution d'une tâche en la privant d'un service dont elle a besoin pour s'exécuter. Dans un système d'exploitation, cela consiste à empêcher une tâche de profiter des services normaux de l'OS. L'attaque se fait sur le service partagé, bien qu'il vise au final une autre tâche qui a besoin de ce service pour son exécution. Par exemple, on peut inonder de requêtes le service d'affichage pour empêcher les autres tâches de s'afficher correctement [SVNC04, FH05]. Ou on peut faire des demandes d'allocation répétées au service d'allocation mémoire afin de priver les

déni de service

[12] Bien que certains auteurs [Gas88] réduisent la définition de l'intégrité à ce dernier cas, l'intégrité de l'information peut être assurée par des moyens divers, e.g. par signature numérique

[13] Il y a cependant des cas où une faille de confidentialité peut être sans grande incidence pour l'isolation, comme le fait de ne pas nettoyer de la mémoire par le système. Mais même dans ce cas, un bug dans une tâche peut rendre une faute indéterministe par ce biais.

déni de ressource

autres tâches de l'usage de cette mémoire (attaque appelée *déni de ressource*). Une attaque insidieuse consiste à priver le service de ressource, afin que le service ne puisse pas s'exécuter, bloquant ainsi toutes les tâches qui dépendent de ce service [LIJ97].

Dans le cadre de tâches temps réel dur critiques, lutter contre les dénis de service n'est pas suffisant. Si une tâche $A$ réussi à seulement *ralentir* l'exécution d'une autre tâche $B$, alors $B$ pourrait ne pas atteindre son échéance (et serait donc en dysfonctionnement). Pour éviter cela, *nous considérerons comme problème de sécurité tout ralentissement par une tâche de l'exécution d'une autre tâche.*

Ces problèmes de dénis de service ne sont pas traités dans cette section, mais dans la suivante, qui traite des problèmes à la frontière entre sécurité et allocation des ressources. En effet, *nous considérons que le problème de déni de service (dans un système d'exploitation) peut être réduit à un problème d'allocation de ressource* : pour empêcher un déni de service, il « suffit » d'avoir paramétré convenablement toutes les politiques d'allocations pour toutes les ressources[14]. Cela présuppose que toutes les ressources ont bien été identifiées, et que les politiques d'allocations sont bien paramétrables, et utilise le fait que les allocations sont déterministes.

Les problèmes traités dans cette partie concernent donc principalement intégrité et confidentialité, c'est à dire l'isolation, la protection des tâches, et le contrôle des communications.

### 2.2.2 Principes de conception sécurisée

Nous voyons maintenant les différents principes pour la conception de systèmes pour la protection de l'information. Comme nous l'avons dit précédemment, les objectifs de sécurité de l'information et d'isolation sont éminemment proches, et la structure des systèmes pour assurer ces buts le sont également. On peut citer par exemple l'usage de TCB, utilisés pour la sécurité, qui peut être étendu pour prouver différentes autres propriétés négatives (e.g. de sûreté de fonctionnement) [Rus89] ; l'usage de systèmes à capacités pour faire de la tolérance aux fautes [Den76] ; les correspondances entre l'architecture IMA aéronautique et l'architecture de sécurité MILS [BDRS08]...

**Intérêt des principes de sécurité** Écrire des programmes sans faille de sécurité, tout comme sans bugs, est une tâche très difficile ; surtout pour un système d'exploitation, par nature de bas niveau et exposé à toute la complexité du matériel. Les failles de sécurité sont des bugs particulièrement difficiles à détecter : alors que les bugs ordinaires causent des dysfonctionnements quand le programme est utilisé en condition normale, les failles de sécurité n'apparaissent souvent que lorsque le programme est "mal utilisé" par intention. Il est donc difficile de trouver des failles de sécurité par test. De plus, même si on arrivait à écrire un programme sans bug, cela ne signifie par pour autant que le système est sûr de fonctionnement [Den76, p.

[14] Notons que cela est possible car le système d'exploitation contrôle l'ensemble des tâches, i.e. le système est « fermé ». Cette approche n'est pas suffisante pour protéger la disponibilité de l'information dans un serveur de données branché sur un réseau ouvert comme Internet, par exemple.

361] : une faute peut par exemple résulter d'une défaillance matérielle. Ce qu'il faut, c'est structurer le système afin d'être paré au mieux à l'éventualité d'une erreur.

Pour aider cette tâche, un ensemble de "bonnes pratiques" à suivre est le meilleur outil dont on dispose. Ces pratiques visent à structurer le système et à se focaliser sur les endroits importants. Cela permet de réduire à la fois le nombre de failles et la portée de ces failles. Dans cette section, nous présentons les principes de sécurité de base, rassemblés par Saltzer et Schroeder [SS74]. Nous reformulons leur explication en définissant des notions qui s'y rapportent au fur et à mesure, et nous complétons ces principes par des développements ou interprétations postérieures.

#### 2.2.2.1 Séparation des privilèges

Un *privilège* est la possibilité d'accomplir une certaine action, par exemple pouvoir programmer la carte réseau, modifier les paramètres de sécurité du système, pouvoir ouvrir un fichier, etc.. La séparation des privilèges consiste à n'attribuer ces privilèges qu'à certains *domaines de protection* séparés. Ainsi, la compromission d'un de ces domaines ne résulte pas en la compromission de l'ensemble du système.

Les processus UNIX ont des droits différents, et implémentent donc la séparation des privilèges. Mais certains privilèges sont accessibles à tout le monde (par exemple le droit de créer un socket réseau). De plus, les privilèges du noyau ne sont pas séparés, au contraire de la structure de type micronoyau, pour laquelle les services réseau, écran, et disque, qui ont besoin de privilèges différents (pour accéder aux différents périphériques), sont placés dans des domaines de protection séparés.

#### 2.2.2.2 Médiation complète

Un privilège se présente sous la forme de l'autorisation d'un accès à un objet ou ressource du système. Ce principe énonce qu'il faut systématiquement vérifier que chaque accès soit bien autorisé.

Ceci se fait par un mécanisme appelé *contrôle d'accès*. À un instant donné, ce mécanisme peut être représenté par une matrice [Lam71] représentant quel domaine a quel *droit* sur quel objet. Il y a deux manières principales de gérer ces droits : soit on associe à chaque objet la liste des domaines qui ont le droit d'y accéder (modèle par liste de contrôle d'accès, ou ACL). Soit on associe à chaque domaine la liste des objets auxquels il a le droit d'accéder (modèle par capacité). La comparaison entre ces modèles est étudiée en section 3.3.1. Les permissions d'accès de chaque domaine vers chaque objet sont régies par des règles qui forment la *politique de sécurité*.

Saltzer et Schroeder mettent particulièrement en garde contre tout mécanisme qui conserverait une copie locale des autorisations (comme un "cache" des accès) : une copie mal mise à jour permettrait des accès non autorisés. Plus généralement, il y a une faille de sécurité (appelé bug *TOCTTOU* [BD96], pour time of check to time of use) lorsqu'un programme vérifie une caractéristique sur un objet, et présume que cette caractéristique est toujours vérifiée alors qu'elle a pu changer. Le schéma de synchronisation use/destroy que nous avons développé (§ 5.2.4) sert en particulier à éviter ces problèmes.

#### 2.2.2.3 Défaillance sûre par défaut

Consiste à interdire tout ce qui n'est pas explicitement autorisé, plutôt qu'autoriser tout ce qui n'est pas explicitement interdit. Autrement dit, les programmes n'ont par défaut accès à rien, et on doit explicitement leur autoriser l'accès. Bien que les deux soient fonctionellement équivalents, le refus d'accès par défaut est plus sûr si les permissions à accorder à un programme sont mal considérées : oublier de fournir un doit d'accès à un programme qui en a besoin sera vite détecté, tandis qu'oublier de retirer un droit d'accès à un programme qui n'en a pas besoin est dangereux et indétectable. Denning appelle un système qui suit ce principe système fermé [Den76].

Les systèmes à capacités purs implémentent naturellement la fermeture du système. Au contraire de systèmes comme UNIX, où encapsuler un programme doit se faire de manière explicite (e.g. [Ber07, §5.2]), et où on n'est jamais sûr que l'encapsulation soit complète (e.g. au vu des nouveaux appels systèmes régulièrement ajoutés à Linux).

#### 2.2.2.4 Principe du moindre privilège (POLP)

Ce principe énonce que toutes les tâches doivent se voir accorder les privilèges minimums nécessaires pour accomplir leur travail. Cela limite les dommages causés par une éventuelle compromission. Par exemple, un programme qui décompresse des fichiers sur le disque n'a pas besoin de pouvoir accéder au réseau.

Ce principe est encore plus intéressant dans le cadre d'un système fermé, car cela encourage à donner une liste de privilège courte.

Miller et Shapiro distinguent deux notions de privilège, la permission et l'autorité [MS03]. La permission autorise un domaine à accomplir une action directement, tandis que l'autorité représente la possibilité de le faire après communication avec d'autres tâches. Les deux doivent être minimisées.

#### 2.2.2.5 Économie de mécanisme

Ce principe consiste à garder la conception aussi simple et petite que possible. Essayer par exemple de trouver des mécanismes généraux plutôt que d'accumuler les exceptions. Ce principe doit être en particulier utilisé pour tout ce qui touche les mécanismes de protection ; Rushby insiste ainsi sur le besoin d'avoir un système de protection unique [Rus84, Rus89].

Garder la conception simple et claire facilite la compréhension du code (et donc sa revue). L'économie de mécanisme permet aussi souvent d'éliminer du code, source de bugs.

Une idée proche de l'économie de mécanisme est celle de l'économie de code en général. En minimisant la taille du code, on minimise le nombre de bugs[15], donc de failles [Ber07]. Les idées sont proches, car en général une conception simple engendre une implémentation simple, ou en tout cas avec peu de lignes de code.

[15] ce d'autant plus que le ration bug/nombre de lignes de code augmente avec la taille du programme [Lip82] : intuitivement quand le programme est petit, on peut en comprendre tous les aspects simultanément, ce qui élimine tous les bugs d'« interfaçage »

Ce n'est cependant pas toujours le cas ; mais nous estimons qu'il faut toujours préférer une conception simple, même au prix d'une implémentation *localement* complexe. Par exemple, une bibliothèque permettant l'accès parallèle à des structures de données parallèles de manière lock-free (comme dans [Fra04]), sera plus simple à utiliser mais d'implémentation beaucoup plus compliquée, qu'une bibliothèque dont l'implémentation est basée sur les locks.

#### 2.2.2.6 Minimisation des mécanismes communs

Il faut minimiser la quantité de mécanisme commun à plus d'un programme ou utilisateur (typiquement les services de l'OS, comme le service d'accès au réseau). Les bugs dans ces mécanismes empêchent tous les programmes qui en dépendent de fonctionner correctement, affectant ainsi une grande partie du système ; ou peuvent involontairement servir d'intérmédiaire pour des communications non-autorisées (canal caché). Minimiser ces mécanismes réduit les possibilités de bugs, donc de failles, dans ces mécanismes. Finalement, un mécanisme non minimal impose des contraintes supplémentaires aux programmes qui l'utilisent : il vaut mieux mettre du code dans des librairies facilement remplaçables, que dans des mécanismes communs fixés.

Un programme utilise généralement un mécanisme commun parce que cela lui est nécessaire pour accomplir une action ; et ce mécanisme commun est généralement le seul moyen pour ce faire. Le mécanisme est donc de confiance. On peut étendre le principe de minimisation des mécanismes communs en la « minimisation des mécanismes de confiances » (appelé réduction ce de qui doit être digne de confiance par Neumann [Neu04], ou minimisation du code de confiance par Bernstein [Ber07]).

Il y a plusieurs réalisations modernes de ce principe. Par exemple, les exonoyaux [EKJO95], en supprimant du noyau le code qui implémente les « abstractions », minimisent les mécanismes communs. Les VMMs (e.g. [BDF+03]) font de même.

Les micronoyaux réalisent une autre forme de minimisation des mécanismes communs. En permettant aux programmes de choisir de quels services ils ont besoin, ils permettent de réaliser une minimisation du *TCB* (i.e. ensemble des tâches de confiances) par tâche [HPHS04]. La notion de TCB est en effet relative : un service peut être de confiance pour une tâche et ne pas l'être pour une autre, suivant que ces services sont nécessaires ou non au bon fonctionnement de la tâche.

TCB

#### 2.2.2.7 Autres principes

Il existe d'autres principes de sécurité qui concernent plutôt des contraintes de génie logiciel, et sont moins importantes dans le cadre de cette thèse. Nous les mentionnons brièvement.

**Conception ouverte** Il ne faut pas que le fait que la conception soit tenue secrète soit considéré comme un mécanisme de sécurité. Ouvrir la conception permet de ne pas se reposer sur ce fait.

La publication des mécanismes de sécurité d'Anaxagoros dans le présent rapport de thèse témoigne du fait que notre conception est ouverte. De plus lorsqu'un

programme n'est pas considéré comme de confiance, nous le considérons comme malicieux et omniscient (cette hypothèse est appelée « presumed collusion » par Shapiro [Sha99, p. 60]).

**Sécurité prévue au départ** On ne peut pas modifier un système existant pour le rendre sécurisé [Gas88]. La sécurité doit être prise en compte lors de toutes les étapes du cycle de développement, de la conception à l'implémentation.

**Acceptation psychologique** L'interface doit être conçue pour être facile à utiliser, pour que les utilisateurs se servent des mécanismes de protection correctement. Elle doit par exemple correspondre à l'image que se font les utilisateurs des mécanismes de protection. Nous n'avons pas vraiment d'utilisateurs dans notre système, mais nous considérons les développeurs d'applications comme les utilisateurs du système. Cela signifie simplement que les concepts de sécurité doivent être simple à appréhender. Par exemple, la structuration objet est maintenant relativement bien connue des programmeurs ; les systèmes à capacités structurés de manière orientée objet [Lev84] sont donc a priori simple d'utilisation pour ces programmeurs.

### 2.2.3 Structurations pour un système sécurisé

S'il y a consensus sur les principes à respecter pour une conception sécurisée, les méthodes de structuration de tels systèmes divergent. Rushby écrit ainsi [Rus84] que le Orange Book[16] n'est (intentionellement) pas un manuel décrivant comment concevoir des systèmes sûrs. Nous décrivons dans cette section les principales approches de structuration.

#### 2.2.3.1 Systèmes d'exploitation généralistes

**Historique des systèmes industriels** La multiprogrammation apparu rapidement en raison du coût des premiers ordinateurs, qui furent ainsi utilisés par de nombreux utilisateurs simultanément. Cela força les concepteurs de systèmes d'exploitation à rendre leur système fortement sécurisé. Un exemple proéminent de tel système est Multics [CV65].

Sur la génération suivante de « mini-ordinateurs », aux capacités plus restreintes, se sont développés des systèmes d'exploitations simplifiés (et avec des mécanismes de protection plus basiques), dont l'archétype est UNIX [RT74, Tho78]. Enfin, les micro-ordinateurs apparurent, qui étaient destinés à un usage personnel, et des systèmes comme DOS ou les premières versions de Windows n'avaient quasiment aucune notion de sécurité [Tan01]. La situation s'est légèrement améliorée depuis, notamment à cause des menaces liées à la connexion de ces systèmes à Internet, mais ces systèmes sont régulièrement sujets à des vulnérabilités aux conséquences graves. Comme indiqué précedemment, pour qu'un système soit sécurisé il faut qu'il soit conçu pour la sécurité depuis le départ.

[16] le document du département de la défense américain décrivant la certification des systèmes sécurisés ;

**Noyaux monolithiques** La manière classique pour structurer un système d'exploitation comme UNIX est le *noyau monolithique* : toutes les fonctions régissant la protection, la communication, et le partage des ressources se trouvent dans le noyau. Le noyau peut ainsi devenir assez gros (le noyau Linux fait ainsi plus de 6 millions de lignes de code [BC05]), et comme le noyau est privilégié, le moindre bug peut impacter l'ensemble du système.

**Micronoyaux** Pour obtenir un système plus fiable, on a sorti du noyau certaines parties, comme la gestion du swapping ou des systèmes de fichier, pour les mettre dans des tâches privilégiées (appelées serveurs). Le noyau plus petit est ainsi appelé *micronoyau*. Le système est plus fiable, car le noyau est plus petit, et que les différents composants peuvent défaillir sans impacter les autres ; et plus modulaire, car les composants peuvent être facilement remplacés.

Les premiers micronoyaux, comme Mach [ABG+86] ou Chorus [RAA+91], ressemblaient à des noyaux UNIX dont quelques composants étaient sortis. Les performances de ces systèmes étaient mauvaises, ce qui montre qu'il ne faut pas les structurer comme un noyau monolithique. Les performances se sont grandement améliorées dans des systèmes plus modernes, dont le design était refait depuis le début, notamment par les travaux de Ford et Liedtke [FL94, Lie93, Lie95a, HHL+97].

Ces noyaux restent cependant relativement gros (e.g. 8700 lignes de code pour seL4 [KEH+09]), notamment en comparaison avec les separation kernels (§l̃abelsec :separation-kernel-dss-mils)[17], ce qui les rend relativement complexes à certifier. De plus, les tâches de confiance hors du noyau sont également relativement grosses, donc sujettes à bugs, qui impactent tous les programmes qui dépendent de cette tâche. Une des raisons pour cela est la présence d'*abstraction* dans le noyau et les tâches de confiance, i.e. le noyau et les tâches de confiance présentent aux applications une interface de haut niveau relativement indépendante du matériel sous-jacent. De plus, il y a souvent des problèmes liés à la gestion des ressources et aux dénis de service (e.g. [WB07], § 2.3).

Ainsi, le système PikeOS [KW07] a simplifié certains aspects de L4 pour arriver à faire certifier le noyau. Notons cependant que des efforts sont en cours pour obtenir la preuve formelle de tels micronoyaux (e.g. seL4 [KEH+09], Coyotos [SDD+04]). Enfin, notons que les micronoyaux récents se rapprochent des systèmes à capacités afin d'obtenir un modèle formel de sécurité, notamment sous l'impulsion des travaux de Shapiro dans EROS [Sha99, SSF99].

**Virtualisation** Les systèmes traditionnels n'étant pas sécurisés, une solution pour assurer l'isolation est la *virtualisation*. Il s'agit d'encapsuler des systèmes d'exploitation entiers dans une *machine virtuelle*. On rajoute ainsi une couche, le *moniteur de machine virtuelle (VMM)*, qui permet de faire fonctionner plusieurs systèmes simultanément.

Dans cette approche, la protection n'est assurée qu'à gros grain : même si le VMM est très sécurisé, la protection n'est pas assurée pour les différentes tâches

[17] notons cependant que ces noyaux permettent beaucoup plus de fonctionnalités, comme la création dynamique de tâches

dans une même machine virtuelle. De plus, les VMMs ont également tendance à grossir , et peuvent être eux même victimes de failles de sécurité [Orm07]. Enfin, le dernier désavantage de cette approche est l'overhead associé à l'ajout d'une nouvelle couche, même si des systèmes comme Xen [BDF+03] ont fortement réduit ce coût, notamment par l'emploi de la *paravirtualisation* (i.e. la modification du code du système d'exploitation à virtualiser), et l'intégration par le matériel de mécanismes facilitant la virtualisation.

La virtualisation est donc surtout intéressante pour assurer la compatibilité avec des applications développées pour d'autres systèmes, mais pas dans l'optique de développer un nouveau système sécurisé.

**Exonoyaux** Certains systèmes d'exploitations proposent de retirer toute abstraction du noyau, et de placer ces abstractions dans des librairies (appelées libOS) : ce sont les *exonoyaux*. Ils ressemblent à des moniteurs de machines virtuelles paravirtualisées, mais exécutent des tâches directement au lieu d'exécuter des systèmes d'exploitation entiers. L'interface est proche, mais généralement différente de celle du matériel nu, ce qui la simplifie.

Entre autre systèmes basés sur cette approche, on trouve Aegis et Xok [EKJO95, KEG+97], Nemesis [Ros95], ou Denali [WSG02]. Ces systèmes ont montrés qu'ils pouvaient atteindre, voire surpasser les performances des systèmes traditionnels, et étaient plus flexibles.

En supprimant les abstractions et en les mettant dans des librairies, ces systèmes réduisent de manière drastique la taille du noyau, et ne sont pas plus difficiles à concevoir et implémenter que des systèmes monolithiques traditionnels. Notons cependant que le partage des ressources est toujours de la responsabilité du noyau, alors que les micronoyaux l'avaient exporté.

**Protection logicielle** La manière traditionnelle d'implémenter la séparation des domaines est par la protection matérielle, i.e. l'utilisation d'un mécanisme matériel qui vérifie que chaque accès mémoire fait par le processeur est autorisé. Ce mécanisme est appellé *MMU* ou *MPU*, la différence étant que la MMU implémente également une translation d'adresses.

Certains programmes sont écrits dans des langages « sûrs », qui ne peuvent accéder qu'à des adresses autorisées par construction. Lorsqu'on exécute un programme écrit dans un langage sûr, la protection matérielle est ainsi superflue. Certains systèmes proposent alors de la supprimer entièrement [HAF+07].

Les approches pour s'assurer qu'un code est sûr sont diverses ; il y a l'utilisation d'un langage « type-safe » [BSP+95], la compilation en bytecode, ou le scan de code [Rip03]. Ces approches souffrent de défauts variés :

- même si un code a été écrit dans un langage type-safe, il est difficile de montrer qu'un binaire a été produit depuis un tel langage. Il faut pour cela que le compilateur soit inclut dans le noyau, ou que l'ensemble des binaires à exécuter soit connu avant le début de l'exécution. L'approche consistant à signer des binaires [LMR00] n'empêche pas ces binaires de mal fonctionner ;

- l'utilisation de bytecode permet de simplifier le « compilateur dans le noyau », mais même ainsi celui-ci reste complexe ;
- même pour un « simple » scanneur de code, il est difficile de s'assurer qu'on a vérifié toutes les attaques possibles. En particulier, il est difficile d'empêcher du code de créer et d'exécuter un autre code malicieux après le scan (sans restriction sur les processus [HAF+07]).

Enfin, toutes ces approches souffrent de problèmes communs :

- l'utilisation de code sûr cause un overhead à l'exécution, à cause des vérifications des indices de tableaux, etc. De plus, cela cause également un overhead à la création/initialisation du programme, pour vérifier que le code est bien sûr ;
- le fait que la protection logicielle demande de faire confiance à un compilateur complexe ne permet pas l'évaluation de ces systèmes aux plus hauts niveaux de sécurité [Rus84] ;
- même si la protection logicielle se faisait en zéro-défaut, cela ne rend pas le système sûr pour autant[Den76]. Ainsi un changement de bit intempestif (bitflip) pendant le calcul d'adresse par le processeur serait détecté avec une grande chance par un système à protection matérielle, tandis que cela pourrait passer inaperçu dans un système à protection logicielle ;
- enfin, la protection logicielle ne permet pas en général l'exécution de tout code qui ne soit pas écrit dans un langage sûr (sauf par utilisation de certains scanneurs de code, qui affectent beaucoup les performances [Rip03, § 4.2.3.2]). Ils ne peuvent donc être utilisé pour exécuter du code existant.

Finalement, le coût de changement d'espace d'adressage n'est pas si important sur des processeurs qui le supportent efficacement. Malheureusement, la domination des systèmes monolithiques (Windows et les variantes d'UNIX) sur le parc de machines actuels n'incite pas les fabriquants de processeurs x86 à optimiser ce coût.

**Conclusion** La domination des noyaux monolithiques dans les systèmes non-académiques démontre l'effort de conception nécessaire pour arriver à écrire un système sécurisé. Les systèmes modernes (micronoyaux, exonoyaux, VMMs) se concentrent essentiellement sur les aspects pratiques de l'implémentation de systèmes plus fiables, et ne s'intéressent traditionellement pas à une assurance plus formelle de la sécurité (bien que cela soit en train de changer avec la convergence des micronoyaux avec les systèmes à capacités, comme dans EROS et seL4 [SSF99, KEH+09]). On y trouve néanmoins quantité de mécanismes intéressants pour la performance, la flexibilité et l'extensibilité, dont certains seront décris dans les sections suivantes.

#### 2.2.3.2 Security kernels

L'approche historique pour construire des systèmes sécurisés est le « security kernel » [Gas88] : l'idée est de minimiser le noyau pour qu'il incorpore toutes les fonctions

de sécurité, et uniquement celles-là. Tout ce qui n'est pas nécessaire pour assurer la sécurité doit alors sortir du noyau[18].

Les fonctions de sécurité assurées par le security kernel sont de deux types [Rus81] : la séparation en différents domaines de protection, et l'application de la *politique de sécurité*. En pratique, quasiment tous les systèmes basés sur cette approche ont implémenté la politique de sécurité multi-niveaux [Gas88, p. 140].

Le problème de cette approche est qu'une seule politique de sécurité peut être prise en compte dans le système. Or un système a en général besoin de « tâches de confiance », qui ne peuvent pas rentrer dans la politique de sécurité [Rus81] (surtout MLS, très restrictive). Des exemples sont les tâches qui font un backup du système, ou des filtres permettant de déclassifier des informations. Ceci contribue à rendre complexe le noyau, ces tâches de confiances et leurs interactions, ces tâches étant en dehors de toute politique de sécurité. Ceci augmente la probabilité de failles de sécurité.

#### 2.2.3.3 Separation kernels, DSS, et MILS

**Separation kernel et approche distribuée de la sécurité** En comparant les security kernels à d'autres architectures pour la sécurité, Rushby a noté que cette approche n'était pas adéquate. Cela l'a conduit à définir une approche basée sur l'émulation de systèmes distribués [Rus81]. Dans cette approche, un *separation kernel* est utilisé dans le seul but d'assurer l'isolation entre les différents domaines, et de contrôler les communications entre ces domaines (cette politique de sécurité du noyau est appellée « channel control »[BDRS08]). Toutes les autres fonctions de sécurité sont assurées par des tâches de confiance.

L'intérêt de l'approche est de séparer les politiques de sécurité en modules indépendants, qui ne font qu'un travail simple. La composition et les relations entre les modules est formalisée [BDRS08] par la présence (ou l'absence) de liens vérifiés par le separation kernel ; on peut représenter le système par un diagramme « boites - flêches », où les boites représentent des domaines et les flèches les liens de communication. La politique de sécurité est assurée en vérifiant que l'ensemble des liens de communication entre deux domaines passe par des domaines de confiance, qui filtrent la communication chacun selon une politique locale simple spécifique. Rushby indique que la différence formelle entre cette approche et celle des security kernels est la non-transitivité de l'interférence [Rus92] : si $a$ ne peut communiquer avec $c$ qu'en passant par $b$, cela est bien différent de si $a$ peut directement communiquer avec $c$ si $b$ agit comme un filtre de confiance.

Cette architecture a été déclinée de plusieurs manières. Dans [Rus84], Rushby propose un système à trois niveaux, constitué d'un separation kernel, de différents ressources managers, et d'un dernier niveau applicatif. Les systèmes à micronoyaux reposeront plus tard sur une architecture de ce type.

Le système DSS [RR83] a développé l'approche de Rushby et montré comment structurer un système sécurisé distribué. En particulier, c'est DSS qui a montré

[18] ceci peut se faire en réécrivant un système, ou en rajoutant un « VMM de sécurité » en sous-couche [Gas88, § 10.7]

comment utiliser les « trusted mediators » pour assurer une politique de sécurité multiniveau (*MLS*).

**MILS** MILS [AFHOT06] est une méthodologie d'écriture de systèmes fondée sur cette approche distribuée de la sécurité. L'idée est de décomposer un système récursivement, afin de minimiser la taille des domaines de confiance (e.g. ceux qui gèrent les données de différentes sources simultanément). Ces domaines deviennent facilement vérifiables. Cette approche peut être étendue pour obtenir une certification compositionelle [BDRS08] : une « policy architecture » formelle permet de montrer que la composition des politiques de sécurité locales permet d'assurer la politique de sécurité globale du système. De plus, cette approche de structuration du système pour faciliter l'assurance de la sécurité est indépendante de la manière dont les tâches sont placées : elles peuvent ainsi être sur différents ordinateurs ou sur le même sur lequel est exécuté un separation kernel. Cette approche semble très prometteuse pour une certification composable, et permettrait même la réutilisation de composants certifiés.

Notons que dans l'architecture MILS, le but n'est pas d'avoir differents niveaux de securité de différentes sensibilités, mais de pouvoir traiter simultanément des informations séparées en plusieurs « classes » relativement indépendantes. Ces classes sont également appellée niveaux (ce qui prête à confusion). MLS est donc un cas particulier de MILS.

**Conclusion** Les separation kernels et la structuration avec MILS sont des approches de choix pour la réalisation de système hautement sécurisés, et qui pourraient même permettent une approche compositionelle de la certification [BDRS08]. Le point clé est que la protection entre domaines est indépendante de la politique de sécurité ; cette dernière est réalisée seulement en paramétrant les liens entre domaines.

Cependant, il y a un point qui n'est pas pris en compte par cette approche, c'est la dynamicité. Même si, actuellement, les applications les plus critiques sont statiques, il est dommage de ne pas pouvoir faire profiter des systèmes plus dynamiques d'une structuration sécurisée de ce type. Les systèmes dynamiques sont généralement confinés dans des compartiments (e.g. souvent un OS virtualisé) qui ne sont pas sécurisés. Les systèmes suivants proposent des solutions à ces problèmes.

#### 2.2.3.4 Les systèmes à capacités

**Introduction** On appelle *système à capacité* un système dont la protection est principalement basée sur l'utilisation de capacités. Ces systèmes sont le plus souvent généraliste, mais leur conception est relativement différente de celle des systèmes traditionels.

Une capacité est une sorte de « clé virtuelle », dont la possession par une tâche indique son autorisation à utiliser une ressource. Bien que ce mécanisme soit commun dans beaucoup de systèmes (les descripteurs de fichiers UNIX en sont un exemple), les systèmes à capacité, dont le contrôle d'accès est fondé uniquement sur les capacités, sont plus rares.

Bien qu'il existe plusieurs modèles pour les capacités, la majorité de ces systèmes utilisent le modèle des capacités-objet [MYS03, MS03], que nous décrivons par la suite. Basiquement, l'usage des capacités implémente la politique de « channel control » des separation kernels ; et permet donc de réaliser des systèmes comme MILS. Mais c'est bien plus que cela, car les droits peuvent être passés pour évoluer dynamiquement, tout en permettant de prédire et restreindre cette évolution ; de nouvelles propriétés de sécurité peuvent être facilement construites ; le tout dans un modèle formalisé et facile à comprendre, car fortement relié à la programmation orientée objet. Les capacités permettent donc la séparation entre protection et politique de sécurité [WCC+74] sur un système dynamique.

**Historique** Jusqu'au début des années 80, les systèmes à capacités étaient une conception de choix pour la réalisation de systèmes sécurisés (comme en témoigne la retrospective de Levy [Lev84]), ou sûrs de fonctionnement [Den76]. Elles ont été par la suite plus ou moins abandonnées, mis à part dans quelques rares systèmes comme KeyKOS [Har85][19], avant de retrouver un (relatif) regain d'intérêt depuis les travaux de Shapiro et Miller [SSF99, MYS03]. Il y a plusieurs raisons qui expliquent cette perte d'intérêt :

- les systèmes à capacités tournaient en majorité sur des systèmes dont l'architecture mémoire était compliquée, ce qui rendait ces systèmes lents [Sha99, p. 11] ;
- le modèle théorique était caractérisé de manière impropre, ce qui a engendré des résultats d'impossibilités très négatifs [MYS03]. En particulier, il n'avait jamais été prouvé que les systèmes à capacité pouvaient assurer le confinement, étape nécessaire pour obtenir une politique MLS [SSF99] (alors qu'il existait des systèmes à capacité qui implémentaient MLS [Key89]). Le problème est que les preuves théoriques ne prenaient pas en compte l'existence de programmes de confiance [MS03].

On peut ainsi voir les systèmes à capacités comme une extension dynamique de l'architecture MILS.

**Résultats** Parmis les résultats théoriques les plus intéressants sur la conception de systèmes à capacités, on peut citer celui de Dennis et Van Horn [DH66] (qui a introduit le concept), le papier de Saltzer et Schroeder [SS74] (qui décrit comment les principes qu'ils énoncent sont mis en place dans un système à capacité), la thèse de Redell [Red74] (qui a entre-autre décrit comment révoquer des droits avec des capacités, et analysé beaucoup de problèmes dans la construction pratique de tels systèmes), la revue de Denning [Den76] (qui explique comment utiliser un système à capacité pour la tolérance aux fautes), celle de Linden qui lie capacités et orientation objet pour la sécurité [Lin76], les travaux de Shapiro, qui ont montré que le confinement pouvait être réalisé [Sha99], et les travaux de Miller et Shapiro

[19] Il est cependant à noter que tous les systèmes d'exploitation actuels utilisent les capacités comme mécanisme de nommage et protection pour représenter les fichiers ou socket ouverts [Sha99, p. 11]. Les ports de Mach sont également des capacités [ABG+86].

[MS03] (qui ont montré qu'un système à capacité ne se formalisait pas juste par un graphe de contrôle d'accès, mais par un langage de programmation).

Différents systèmes à capacités ont été créés, apportant différentes contributions « pratiques » (que nous utilisons dans cette thèse) : nous en donnons quelques exemples. CAL/TSS [LS76] a inventé l'utilisation d'identifieurs uniques pour résoudre le problème des références invalide. Hydra [WCC+74] a utilisé l'orientation objet pour structurer le système, donné un type à chaque objet, et une opération `typecall` pour appeller le domaine responsable de chaque objet d'un même type [Lev84, §6.7.2]. Son successeur StarOS [Lev84] a montré que `typecall` était suffisante pour toutes les opérations sur un objet. CAP [NW77, WN79] utilisait les capacités pour représenter le droit d'usage des ressources mémoire, et controllait l'usage des ressources hiérarchiquement. Keykos [Har85, BFF+92] réussit à comptabiliser les ressource des processus de manière précise, et résister aux dénis de services. EROS [SSF99, Sha99] a effectué une jonction entre systèmes à capacités et micronoyau et démontré que les systèmes à capacités pouvaient être performants sur les architectures contemporaines.

### 2.2.4 Conclusion sur la sécurité

Nous avons vu que les notions de sûreté de fonctionnement, sécurité de l'information et partitionnement étaient proches, ce qui nous permet de les nommer collectivement « sécurité ». Nous avons vu les principes de base permettant l'écriture de système sécurisé, puis un ensemble de méthodes de structurations de systèmes afin d'assurer cette sécurité. Globalement, cette approche consiste à séparer les fonctions en petits domaines de protection aux privilèges restreints, avec un contrôle strict des communications entre ces domaines, et une minimisation des points de défaillance communs. Le contrôle d'accès est plus simple et sûr lorsqu'il est réalisé de manière distribuée, que ce soit statiquement (comme avec MILS) ou dynamiquement (avec les capacités).

## 2.3 Intéractions entre sécurité et partage des ressources

Les deux principaux rôles du système d'exploitation sont la sécurité et l'allocation des ressources. Ces rôles ne sont pas indépendants, et les solutions proposées pour chacun de ces problèmes peuvent entrer en conflit. Dans cette section, nous identifions ces conflits et examinons les solutions possibles.

### 2.3.1 Inadéquation des unités d'allocation et de protection

L'unité de protection élémentaire est le domaine de protection. Mais pourquoi l'allocation des ressources se ferait-elle à la granularité du domaine ? On peut en effet très bien imaginer une allocation à granularité plus petite (e.g. pour gérer différentes activités au sein d'un même processus), ou plus grosse (e.g. le logiciel

qmail est divisé en plusieurs domaines de protection par sécurité, mais on pourrait vouloir gérer les ressources qu'il occupe d'un seul bloc).

Cette unification des unités d'allocation et de protection (et en particulier, des threads et des process) est une simplification, notamment faite dans UNIX et les systèmes qui s'en inspirent, les VMMs, CAL/TSS [SH02, p. 3] ou des systèmes académiques comme Nemesis. Cette simplification pose cependant problème [BDM99] ; nous en donnons deux exemples précis.

**Multithreading** Dans le modèle UNIX original, le processus est l'unité de dispatch de temps CPU. Ce n'est plus vrai dans les systèmes actuels, pour lesquels le receptacle du temps CPU est le thread. La raison de ce changement est qu'il a été trouvé coûteux de devoir dupliquer tout un processus lorsque plusieurs traitements indépendants doivent être faits simultanément (notons que cette affirmation est disputée par [ELS88]), et que cela permet l'extension du modèle aux systèmes multiprocesseur.

**Changement d'espace d'adressage** Un programme, suivant les activités qu'il a à faire, peut être amené à changer d'espace d'addressage pour ces différentes activités. Par exemple, la sécurisation des communications dans OASIS est basée sur ce principe : lors de leurs opérations normales, les tâches ont accès à leurs données propres et leurs buffers de communication ; lorsqu'elles veulent communiquer, elles entrent dans un espace d'adressage particulier qui leur donne accès aux buffers de communications des autres tâches (mais plus à leurs données propres). Ce mécanisme garantit la sûreté des communications, et pourrait être utilisé pour communiquer dans d'autres systèmes. Les *paths* de Scout [MP96] effectuent également des changements d'espace d'adressage.

Beaucoup de systèmes, y compris antérieurs à UNIX, permettaient cette opération utile à la sécurité ; nous y reviendrons quand nous parlerons du thread tunneling (§ 2.3.2.3). On peut émuler ce comportement avec UNIX, en utilisant plusieurs processus qui se synchronisent avec des pipes, mais cette émulation cause un gros overhead par rapport à l'utilisation d'un mécanisme dédié [20]. De plus, le temps est décompté séparément pour chacun de ces processus, alors qu'il s'agit de la même activité logique. Redell notait déja en 1974 que cette simplification complexifiait les couches logicielles du dessus [Red74, p. 16].

**Solutions proposées** Les solutions proposées à ce problème sont relativement peu nombreuses, la simplification « unité de dispatch = unité de protection » étant communément admise.

Banga et al. [BDM99] fournissent une solution à ce problème, en définissant des « resource containers », receptacles des différentes ressources, distincts des processus, qui représentent les domaines de protection. Mais cette solution ne convient qu'aux noyaux monolithiques. Rialto fournit une notion d'activité [JLDJB95, §3.1], similaire aux resources containers, mais également applicable aux micronoyaux. Notons que si ces deux systèmes permettent une comptabilisation des ressources, ils ne sont pas utilisés pour permettre une allocation déterministe.

[20] Cette émulation est notamment utilisée dans l'emulation d'OASIS sur UNIX

Escort, basé sur Scout, a deux unités de dispatch des ressources : le domaine et le chemin [SP99, §2.4]. Mais ce système ne permet pas d'aggréger les ressources consommées par plusieurs chemins.

**Utilisation de la structuration hiérarchique** La structuration hiérarchique du système (abordée plus en détail § 2.4.3.2) est également une solution à ce problème. Si on désire comptabiliser les ressources à une granularité plus grosse que celle du domaine de protection, il suffit de regrouper les différents domaines de protections avec un parent commun, et d'utiliser ce parent pour compter les ressources. Si au contraire, on veut les comptabiliser avec une granularité plus fine, on peut toujours créer des plus petits domaines de protection dans le domaine concerné. Le seul problème dans ce dernier cas étant l'overhead de la création et utilisation de ces plus petits domaines de protection ; et la nécessité du système de supporter la création de nouveaux niveaux de hiérarchie.

### 2.3.2 Communication avec un service partagé

Même si on suppose le problème de l'inadéquation entre unité de protection et receptacle de ressource résolu, il reste le problème de la communication entre entités situées dans des pools de ressources (et domaines de protection) différents.

En particulier, nous avons vu l'usage de services partagés pour s'assurer de toute propriété de sécurité « négative » [Rus89] (i.e. prévention de mauvais comportements), par exemple de l'utilisation correcte d'une ressource. Lorsqu'un programme a besoin d'effectuer une action protégée par le service, il se passe une opération appellée *appel de service*. Cette opération consiste à ce que le client envoie sa requête, que le service la traite, et qu'(éventuellement) le service répond au programme lorsqu'il a terminé. Les appels systèmes envers les noyaux monolithiques en sont un exemple simple ; les RPC des micronoyaux en sont un autre exemple.

appel de service

Ce besoin d'appeller des services partagés pose différents problèmes. L'un d'entre eux est la performance : comme l'appel de service est un mécanisme employé de manière extrêmement fréquente, ses performances sont cruciales [Lie93]. Il y a ainsi eu beaucoup de recherche sur l'optimisation de ce mécanisme (e.g. [Che84, BALL89, Lie93, FL94, SFS96]), et les performances de ce mécanisme sont dorénavant satisfaisantes. Un autre est le contrôle des communications, mais ce problème peut être simplement réduit au problème de contrôle d'accès à un objet « lien de communication ».

Le dernier problème important et peu étudié concerne l'allocation des ressources : pour satisfaire une requête, le service consomme des ressources (au minimum, le temps CPU nécessaire pour exécuter la requête). Comment faut il comptabiliser et attribuer les ressources utilisées pour traiter ces requêtes ? Lorsque ces ressources sont mal comptabilisées, on peut se retrouver en situation de déni de ressource : lorsque toutes les ressources attribuées au service sont utilisées, celui-ci ne peut plus traiter les requêtes des clients. Ce type d'attaque a été identifié par différents auteurs [Key88, LIJ97, Sha03].

Nous avons catalogué les solutions existantes à cette problèmatique en trois. La présentation est la suivante : nous présentons chaque type de solution appliqué au

temps CPU, avant de généraliser l'approche aux autres ressources. La Figure 2.2 propose un résumé. À noter également, la solution originale de Nemesis, qui adopte une structure à la exokernel pour minimiser les ressources à utiliser dans les service partagé [Ros95, p. 16](ce qui réduit considérablement le problème, mais sans complètement l'éliminer).

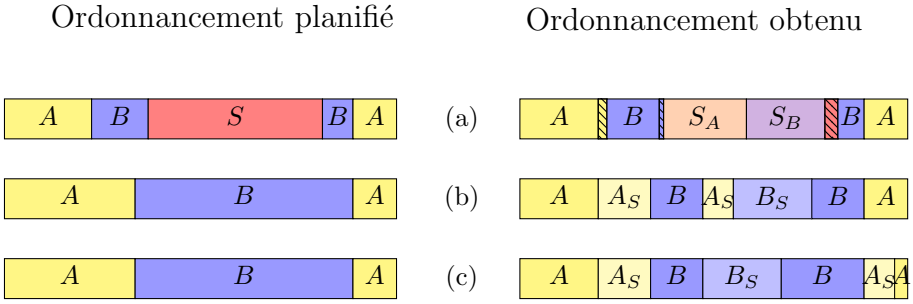



FIG. 2.2 – Comptage du temps CPU passé dans un service, soit (a) par communication asynchrone, ce qui demande de prévoir du temps pour le service et l'introduction d'une sous-politique d'ordonnancement (§ 2.3.2.1) ; (b) par communication synchrone, i.e. en permettant aux ordonnancements prévus et obtenus de différer (donc en rendant l'ordonnancement indéterministe) (§ 2.3.2.2) ; (c) en servant les requêtes sur le temps alloué au demandeur. $S_A$ est le temps de $S$ passé à exécuter des requêtes pour $A$, tandis que que $A_S$ est le temps de $A$ passé à exécuter des requêtes dans $S$ (§ 2.3.2.3).

#### 2.3.2.1 Communication asynchrone/sous-pools et sous-politiques

**Présentation** La première méthode est la *communication asynchrone*. Ce terme désigne la possibilité pour une tâche d'envoyer un message à une autre tâche, qui sera traitée à un moment postérieur à l'envoi ; en particulier l'emetteur n'attend pas immédiatement la réponse.

Ce modèle est bien connu car utilisé pour communiquer entre machines distribuées. Dans un système à mémoire partagée, la communication se fait par l'intermédiaire d'un buffer mémoire partagé : l'émetteur écrit dans le buffer, qui est par la suite accessible au recepteur. Ce buffer peut être une simple zone de mémoire partagée directement accessible aux deux tâches, ou peut passer par le noyau pour la communication.

La communication asynchrone est extrêmement utile pour la communication entre deux tâches en général. Les problèmes surviennent lorsqu'on l'utilise pour l'appel de service.

**Communication asynchrone stricte avec service partagé** Dans la communication asynchrone « basique », ou « stricte » (Figure 2.2(a)), l'émetteur utilise son temps CPU pour envoyer le message, et le recepteur son temps CPU pour lire et traiter le message. Lorsque le recepteur est un service partagé, il faut donc provisionner du temps CPU (temps pour $S$ sur la figure) pour que le service puisse être exécuté.

Cette provision de temps CPU est ce que nous appellerons plus génériquement un *sous-pool*.

Pire, puisque le service peut avoir reçu plusieurs messages entre les différents moments où il est exécuté, il faut définir un ordre de traitement de ces requêtes (sur la figure, on a choisi d'exécuter les requêtes de $A$ avant celles de $B$, mais l'inverse aurait été possible). On peut voir ceci comme un ordonnancement dans l'ordonnancement, ce qu'on appellera plus génériquement *sous-politique*. Ce problème est rarement remarqué puisque la politique FCFS est généralement appliquée par défaut. Par exemple, les ports de Mach sont des queues FIFO [ABG+86, p. 10]. Cette politique rend notamment possible les dénis de service : un ou plusieurs clients peuvent inonder le service de requêtes (et remplir les files FIFO), empêchant les autres clients d'y accéder.

**Problèmes** L'utilisation de la communication asynchrone pour faire des requêtes à un service partagé est pénible pour différentes raisons.

Tout d'abord, il faut prendre en compte la provision de temps CPU pour le service dans l'ordonnancement. Ensuite, il faut définir l'ordonnancement des requêtes. Ces deux paramètres influent sur la latence de la réponse, mais celle-ci est généralement élevée. De plus, quand l'ordonnancement est statique, du temps CPU est perdu à cause de la fragmentation interne (en hachuré sur la Figure 2.2). De plus, la communication asynchrone nécessite des copies dans des buffers intermédiaires, et a un overhead assez important [FL94], surtout si les buffers sont alloués dynamiquement[21]. Enfin, pour paralléliser le traitement des requêtes (utile en multiprocesseur) il faut que les services partagés soient multithread, ce qui complexifie la question du provisionnement de temps CPU.

Mais, lorsque le nombre de clients est fixe (système statique), il permet d'avoir un ordonnancement prévisible et même déterministe, bien qu'il soit rendu plus compliqué par la présence de l'ordonnancement des requêtes. C'est pourquoi on l'utilise pour les systèmes critiques. Des exemples de systèmes qui fonctionnent de cette manière sont le partitionnement pour l'avionique [Rus98, Aer], les systèmes basés sur des separation kernels et MILS [Rus81, BDRS08], les VMMs [BDF+03], ou OASIS [CDA+05]. Attention, tous ces systèmes n'utilisent cette méthode que pour les ressources partagées ; celles qui ne le sont pas sont accédées directement.

**Généralisation : sous-pool et sous-politiques** L'effet de la communication asynchrone sur l'ordonnancement CPU peut être généralisé à toutes les ressources. Nous appellons cette approche « sous-pool et sous-politiques ».

Nous dénommons *sous-pool* la provision d'un ensemble de ressources que le service utilise pour servir les requêtes de ses clients. Par exemple, la mémoire que provisionnerait un service de partage du réseau pour créer des buffers réseau pour ses clients. Le problème principal des sous-pool est la *fragmentation interne* : le fait que si une ressource est reservée pour l'usage du service, elle ne peut pas être mieux utilisée ailleurs.

fragmentation interne

[21] De plus, l'allocation de buffers par le noyau permet une attaque par déni de resource sur l'ensemble du système [Sha03]

Nous appellons *sous-politique* la politique d'allocation définissant la distribution des ressources du sous-pool entre les différents clients du service. Les sous-politiques sont un exemple parfait de politique qui est soit inefficiente, soit insécurisée (§ 2.1.4.3) : en effet *les services ne savent en général rien de leurs clients, surtout quand ceux-ci peuvent être crées dynamiquement*. En pratique, cette sous-politique est souvent FCFS, ce qui explique pourquoi les systèmes actuels sont souvent vulnérables aux dénis de services. Il arrive cependant qu'une politique différente soit utilisée : ainsi PikeOS permet un partitionnement statique de la mémoire du noyau [KW07, p. 55]. Dans ce cas, la mémoire libre réservée pour les allocations noyau relative à une partition est perdue, car elle ne peut pas être utilisée pour un autre usage.

Enfin, même pour un système statique critique où tous les clients sont connus, ce mécanisme complexifie l'analyse de sûreté puisqu'il faut vérifier pour chaque service si chaque client a assez de ressource pour s'exécuter convenablement.

Notons en particulier que beaucoup de ressources, allouées par les services en réponse aux requêtes des clients, ne sont pas perçues comme telles à cause de l'utilisation de l'algorithme FCFS. Ainsi, utiliser `malloc` pour stocker une requête d'un client, insérer un élément dans un tableau fini ou une table de hachage, retirer un élément dans une liste, ou augmenter son espace d'adressage sont autant d'utilisations de FCFS sur des ressources différentes. Les ceilings de CAL/TSS [LS76, §6.3] en sont un exemple flagrant.

**Communication asynchrone modifiée** Dans beaucoup de systèmes, la latence de la communication asynchrone stricte est bien trop importante. De plus ces systèmes n'utilisent pas un ordonnancement statique. C'est pourquoi ils couplent la communication avec l'ordonnancement.

Par exemple, dans Mach l'émetteur peut se bloquer lorsque la queue FIFO des messages est pleine [ABG+86, p. 11]. Les serveurs multithreads se bloquent quand la liste des messages est vide. Nemesis couple la communication par mémoire partagée avec des mécanismes d'évènements [Ros95, § 4.6]. Le problème de ces approches est que cela modifie l'ordonnancement de manière imprévisible, donc ne supporte pas de politiques d'ordonnancement déterministe.

Dans OASIS, l'émission de messages ne contraint pas l'ordonnancement : au contraire, c'est l'émission de messages qui est contrainte par l'ordonnancement de la tâche (i.e. l'émission de message ne peut se produire qu'à des instants prédeterminés) [CDA+05]. Ainsi, l'ordonnancement et l'allocation de ressource reste déterministe (et l'ensemble des ressources nécessaires à la communication, comme la taille des buffers, peut même être calculée à la compilation). Mais cette approche est reservée à des tâches temps réel.

Le couplage entre ordonnancement et communication a été simplifié, rendu plus rapide, et avec une sémantique plus claire, en transformant la communication asynchrone des micronoyaux en communication synchrone [Lie93]. Chorus est un système où cette évolution s'est faite petit à petit [RAA+91, § 2.6]

#### 2.3.2.2 Communication synchrone

**Présentation** La communication synchrone se base sur la notion de *rendez-vous*. Lorsqu'une tâche souhaite communiquer, elle exécute un appel système pour indiquer qu'elle est « prête pour la communication ». Si une autre tâche est en attente pour communiquer, la communication s'initie ; autrement cette tâche se bloque en attendant qu'une autre tâche soit prête à communiquer.

Des systèmes implémentant des mécanismes de communication synchrone sont L4 [Lie95a], EROS [SFS96] ; mais on peut faire remonter ces mécanismes aux travaux de Brich Hansen [Han70].

Cette approche à de nombreux avantages en terme de performance sur l'IPC asynchrone [Lie93, Sha03]. En particulier, ce procédé élimine tout besoin d'allouer et de faire des copies dans un buffer intermédiaire : les données peuvent être directement copiées d'un espace d'adressage à un autre.

**Communication synchrone avec service partagé** Pour communiquer de façon synchrone avec un service partagé, il suffit que celui-ci se mette en attente de reception d'un message. Lorsqu'un client veut faire une requête, il essaie de communiquer avec le serveur ; celui-ci est réveillé, effectue la requête, répond, et se remet en attente de message. Les micronoyaux qui emploient cette techique utilisent des optimisations supplémentaires, comme le regroupement des appels systèmes d'envoi et de reception [Lie93, SFS96].

Communiquer de manière synchrone avec un service partagé a plusieurs avantages en terme de performance. Outre un coût moins important de la communication, la latence devient aussi plus faible. Notamment, il est courant pour ces systèmes que le client donne la fin de son quanta de temps au service pour qu'il puisse exécuter la requête immédiatement [Lie93].

**Problèmes liés à l'ordonnancement et héritage de priorité** La communication synchrone séquentialise les requêtes faites au service : chaque client attend à l'entrée du service que sa requête soit exécutée. Elle est donc équivalente d'un point de vue de l'ordonnancement à l'utilisation d'un mutex partagé entre tous les utilisateurs du service.

Comme on l'a vu précédemment (§ 2.1.2.1), l'utilisation de sémaphores ou autre mécanisme bloquant perturbe l'ordonnancement. Une des perturbations les plus graves est l'inversion de priorité, qui se résoud habituellement par l'héritage de priorité. Steinberg et al. [SWH05] ont pour pallier à cela implémenté l'héritage de priorité dans L4. Ils ont de plus fait en sorte que le temps passé dans le service à exécuter la requête du client soit décomptée du temps du client, ce qui fait que l'utilisation du service se déroule, du point de vue de l'ordonnancement, exactement comme si un mutex partagé était utilisé. Le résultat est présenté Figure 2.2(b). QNX implémente également l'héritage de priorité lors de l'envoi de message [Hil92].

Le problème de cette technique est qu'elle limite de facto la communication synchrone à l'ordonnancement par priorité fixe pour le temps réel. Même si on peut imaginer des techniques d'héritage de priorité dynamique (e.g. comme le CPU inhéritance scheduling de Ford et Susarla [FS96]), les analyses pour ces ordonnance-

ments sont moins connues que pour l'analyse par priorité fixe. Enfin, même si ces analyses étaient abouties, on est encore loin d'aboutir à des analyses en présence d'ordonnancements simultané de types différents, ce qui rendrait impossible l'utilisation de politiques d'allocation hiérarchiques arbitraires. Par exemple, PikeOS suit cette approche, mais impose un ordonnancement par priorité fixe dans les partitions [KW07].

Ces problèmes sont mitigés par Härtig et al. qui proposent d'avoir un thread par client temps réel dans les service partagés ; cependant cela consomme beaucoup de mémoire.

**Vulnérabilités et autres problèmes** Les IPC synchrones souffrent également d'autres problèmes de sécurité.

Liedtke a montré que les services implémentés de cette manière étaient sensibles aux dénis de services [LIJ97]. En particulier, les « pulsar attacks » consistent à envoyer des requêtes en boucle au service, qui perd du temps à exécuter ou rejeter ces requêtes. Les solutions proposées ne sont pas satisfaisantes : utiliser un thread par client demande trop de ressources, et la solution de « clan and chief » a été abandonnée car peu performante [Elp]. Les différentes attaques par déni de ressource sont résolues en préréservant des ressources, ce qui est inefficace comme nous l'avons démontré (voir sous-pools et sous-politiques, § 2.3.2.1).

Shapiro [Sha03] fait une analyse des conséquences du bloquage dans les IPC synchrones lorsqu'utilisés pour communiquer avec un service. En particulier, un service ne doit pas se bloquer en attente d'un client lorsqu'il lui envoie sa réponse ; il ne doit pas se bloquer non plus lorsque des données sont copiées d'un domaine à l'autre et que cela provoque un défaut de page chez le client.

Enfin, l'approche synchrone, bloquante, pose problème pour l'exploitation des systèmes multiprocesseur, car les services sont en général monothreadés. Pour rendre le système multithreadé, il faudrait que le noyau dispatche automatiquement les requêtes sur les threads disponibles (comme le fait Spring [HK93]) : c'est une utilisation de FCFS sur les threads disponibles, et le client peut donc se bloquer à des moments imprévisibles.

L'idée d'avoir un thread par client existe cependant naturellement pour un type de communication : le *thread tunneling*.

#### 2.3.2.3 Thread tunnelling

**Présentation** Ce mécanisme de communication existe sous une myriade de noms différents ; nous reprenons dans cette thèse le nom choisi par Roscoe, le thread tunneling [Ros95, p. 85].

Cette technique consiste à ce que l'exécution d'un thread dans un domaine de protection se poursuive dans un autre domaine de protection (si les espaces d'adressages des deux domaines sont différents, le thread passe d'un espace d'adressage à l'autre). Pour qu'il s'agisse véritablement d'un mode de communication, il faut que des données soient transférées entre les domaines : cela peut se faire simplement par les registres processeur, par copie de messages ou toute autre technique.

**Communication avec service partagé en utilisant le thread tunnelling** Le thread tunnelling est utilisé essentiellement pour communiquer avec un service partagé, même si certains systèmes l'utilisent comme leur mécanisme d'IPC (e.g. Scout [MP96], Pebble [GSB+99]).

Ce type de communication implémente un appel de fonction protégé : le client fait l'appel, le thread change de domaine de protection et revient, avec éventuellement une valeur de retour. Du point de vue du client, cet appel est fonctionnellement équivalent à un appel de fonction.

L'exemple moderne le plus classique de l'utilisation de cette technique est sans nul doute *l'appel système* (system call) vers un noyau monolithique. Dans UNIX [Tho78, §2], le processus s'exécute normalement dans son domaine utilisateur jusqu'à ce qu'il effectue un appel système. À ce moment il s'exécute dans le domaine de protection du noyau, jusqu'à ce que celui ci retourne l'exécution dans le domaine utilisateur. Le modèle monolithique UNIX a la particularité qu'un thread ne peut s'exécuter que dans son domaine de protection ou dans celui du noyau.

Beaucoup de systèmes anciens proposaient ces mécanismes par le hardware. On peut citer les rings de Multics [SS72], l'instruction `CALL` des processeurs x86 [Int09], et beaucoup de systèmes à capacité (e.g. CAP [WN79, p. 7] ou Hydra [WCC+74, p. 339])

Ce mécanisme fut ensuite utilisé pour améliorer les performances de la communication dans les micronoyaux. Dans ce cas, c'est le noyau qui implémente le changement de domaine de protection. Des exemples de tels mécanismes sont LRPC [BALL89], les doors de Spring [HK93, §2.2], les migrating threads de Mach 4.0 [FL94], les synchronous protected control transfers des exokernels [EKJO95, §5.5], les portals de Kea [VH96, §2.4] ou de Pebble [GSB+99, §3.1].

Ce mécanisme a beaucoup d'avantages sur les mécanismes de communication précédents [FL94] : latence extrêmement faible, overhead faible, simplification du code de RPC dans le noyau, et support automatique des services multithreadés. Ses désavantages sont la nécessité d'allouer des ressources de manière transitoires (dont on fait l'analyse et propose des solutions par la suite), et le fait qu'il ne peut être utilisé que sur des machines à mémoire partagée[22].

**Relation avec l'ordonnancement et multithreading** La principale propriété de cette technique est que la *communication est indépendante de l'ordonnancement* : le thread est exécuté selon l'ordonnancement prévu, indépendemment du domaine de protection dans lequel il se trouve. La Figure 2.2(c) en donne un exemple. Notons que le thread peut être préempté et reprendre l'exécution dans le service ; comme d'autres clients peuvent faire une requête à ce moment, *le service est multithread par nature*. Cela permet de tirer parti du matériel multiprocesseur[23].

Cette indépendance vis à vis de l'ordonnancement contribue à ce que l'overhead de la technique soit faible : l'appel de service n'a pas besoin de toucher aux structures

[22] Il n'y a donc pas « transparence réseau » comme sur un système distribué ; cette transparence devra être réalisée à des niveaux plus haut.

[23] L'argument selon lequel cela augmente la complexité du service ne tient pas, puisqu'on peut prendre un mutex couvrant tout le service à son entrée si nécessaire.

liées à l'ordonnancement ([VH96, §2.3])[24].

Mais la principale conséquence de l'indépendance avec l'ordonnancement est que *le temps passé à exécuter la requête du client est fournie par le client.* Ceci offre beaucoup d'avantages pour l'ordonnancement de tâches dans le système :

- Il n'y a plus besoin de provisionner de temps pour l'exécution du service ;
- Il n'y a plus besoin de définir une sous-politique d'ordonnancement ;
- L'instant où la requête est exécutée devient simple et prévisible ;
- Le service devient invulnérable aux dénis de services sur le temps CPU, e.g. à l'inondation de requêtes.

En bref, l'ordonnancement du système devient beaucoup plus simple, plus sûr, et plus performant.

Ces avantages sont cependant conditionnés à la manière dont le thread tunnelling est réalisé :

- le temps passé dans le service doit être décompté comme si le thread était dans le client. Cela implique que le receptacle de temps CPU soit le thread, et non le domaine. En particulier, Roscoe a rejetté le thread tunnelling pour Nemesis car il décomptait le temps CPU avec les domaines [Ros95, p. 86] ;

- le service ne doit pas modifier l'allocation du temps CPU. Par exemple, si on demande à ce qu'un mutex soit acquis avant de pouvoir exécuter le service, on se retrouve avec le même ordonnancement que celui obtenu avec la communication synchrone Figure 2.2(b)[25].

Peu de systèmes obéissent à ce deuxième point, ce qui fait que le tunnel de thread n'a (à notre connaissance) jamais été utilisé pour des systèmes exécutant des tâches temps réel dur critiques. Il y a deux raisons pour cela :

1. Le service est multithread, ce qui demande l'usage de primitives de synchronisation (e.g. sémaphores) qui modifient généralement l'ordonnancement ;

2. L'appel au service consomme également d'autres ressources (e.g. de la mémoire pour la pile), et le client peut se bloquer si elles sont en nombre insuffisant. C'est ce qui se passe par exemple dans Spring [HK93, §3.2].

Le premier point sera étudié en profondeur aux chapitres 4 et 5. Nous nous intéressons maintenant au deuxième.

**Allocation de mémoire transitoire** Pour chaque requête traitée par le service, il est nécessaire d'allouer de la mémoire pour stocker des informations de manière temporaire (il faut en particulier une pile). La mémoire allouée augmente donc en fonction du nombre de requêtes en train d'être simultanément traitées.

[24] Notons néanmoins que le « Lazy Scheduling » [Lie93, §5.3.4] dans L4 rend l'IPC synchrone aussi compétitif que le thread tunnelling [HHL+97, §7.1]

[25] En fait, on peut émuler la communication synchrone avec un service partagé en combinant le thread tunneling avec la prise d'un mutex par service.

Dans Spring, les clients se bloquent quand il n'y a plus assez de mémoire (ou de « server thread »). Cela résulte encore une fois de l'application de la politique FCFS sur la ressource « server thread » : les « server thread » sont alloués au fur et à mesure des appels de service, et on a un déni de service lorsque tous les « server threads » ont été alloués.

Considérons les différentes solutions à ce problème.

1. Changer la politique d'allocation des « server threads » : comme on l'a vu (§ 2.3.2.1), il est difficile de trouver une autre sous-politique qui convienne, et cela sera soit inefficace, soit non sécurisé.

2. Avoir autant de pile que de clients : cela demande, lors de la création de chaque client, la création d'une pile dans chaque service. Autant cela est parfaitement applicable si le noyau est le seul service (c'est ce qui est fait dans les noyaux monolithiques « process model » [FHL+99]), autant cela résulte en un gros gâchis de mémoire lorsqu'il y a plusieurs services.

3. Faire en sorte que le client fournisse une pile au service pour la durée de l'appel, par exemple comme le fait Pebble [GSB+99, §5.4].

Cette dernière proposition est clairement la plus intéressante : il n'y a plus de déni de ressource sur la mémoire, et ce sans avoir besoin de sous-politique d'allocation dans le service ni de gâcher de la mémoire. Nous allons maintenant généraliser cette technique.

#### 2.3.2.4 Transfert et prêt de ressources

**Présentation** Nous avons vu que lorsqu'un service traitait les requêtes soumises par ses clients, il consommait des ressources. Typiquement on résoud le problème en provisionnant des ressources pour le service (constituant ainsi une sous-pool), qu'on distribue entre les requêtes (selon une sous-politique).

On évite complètement ce problème si le service utilise les ressources du client pour exécuter les requêtes de ce dernier. Nous appellons cette technique *transfert de ressources*. En évitant ces problème de sous-pool et de sous-politique, on combine efficacité, performance, et simplicité des politiques d'allocation.

Cette technique existe sous différentes formes dans la littérature, bien que n'étant jamais identifiée en tant que principe général. Nous présentons ici l'état de l'art relatif à ce sujet, en catégorisant le transfert de ressources selon différentes propriétés.

**Transfert temporaire contre prêt** Considérons la situation où un programme client souhaite recevoir des paquets en provenance du réseau. Pour cela, il doit contacter le service réseau, et lui fournir de la mémoire que ce dernier utilisera comme buffer réseau. On peut voir deux manières pour ce faire.

La première approche est de modifier l'allocation mémoire pour qu'un peu de la mémoire du client soit désormais allouée au service. Lorsque le client n'a plus besoin du service réseau, cette mémoire est restituée au client. Cette approche à deux problèmes :

- le transfert de mémoire doit notifier le service gérant la politique d'allocation mémoire, ce qui cause un certain overhead. Cet overhead devient très significatif lorsque les politiques sont structurées hiérarchiquement (§ 2.4.3.2) : il faut alors notifier tous les services de politiques situés sur le chemin du client vers le service. Cette approche est notamment prise par Bastei/Genode [FH06] ;

- l'implémentation des modules gérant la politique d'allocation mémoire est rendu plus complexe, car les modules doivent retenir qu'une partie de la mémoire allouée au service doit en fait être décomptée du client.

La deuxième consiste à donner au service *l'usage* de la mémoire prêtée, mais sans lui en concéder la *propriété*. Le client n'a pas le droit de se servir de la mémoire, mais en est propriétaire, ce qui lui permet de récupérer cet usage quand bon lui semble. Cette technique n'a pas besoin de modifier l'allocation ni de faire intervenir les politiques d'allocations : il n'y a donc pas d'overhead, et les politiques d'allocations restent simples. La gestion du réseau dans EROS [SSS04] est réalisée de cette manière. L'opération « retype » dans seL4 [EDE08] peut également être vue comme un prêt au service « noyau ».

prêt de ressource

Formellement, nous définissons le *prêt de ressource* comme tout transfert d'une ressource ne nécessitant pas l'intervention d'un domaine tiers responsable de la ressource. Lorsque ce tiers intervient, on parle de *transfert temporaire*. On peut faire l'analogie avec deux clients d'une banque : le transfert temporaire consisterait à transférer de l'argent du compte d'un client à un autre (la banque est tenue au courant), tandis que le prêt est fait quand un client fournit du liquide à l'autre (la banque n'est pas impliquée dans l'opération). Par cette définition, le thread tunneling présenté section 2.3.2.3 est un *prêt* de temps CPU, même si le temps CPU ainsi prêté n'est jamais « rendu ».

Si le prêt de temps CPU est relativement répandu, le prêt d'autres ressources « statiques », comme la mémoire, l'est beaucoup moins. En effet ce prêt nécessite que le client soit capable de déléguer le droit d'usage de la ressource à un service sans en notifier d'autorité centrale. Les seuls systèmes dans lesquels cette opération est possible sont les systèmes à capacités. De plus le droit d'usage des ressources doit être représenté par une capacité. Peu de systèmes permettent cela ; il y a les space banks de KeyKOS et EROS [BFF+92, SSF99]. Sel4 permet également la donation de mémoire, mais seulement au noyau [EDE08].

Les systèmes basés sur les ACL ne peuvent pas implémenter le prêt de ressource : l'ACL pour une ressource est gardée par le service qui gère la ressource. La ressource ne peut donc être transférée qu'en contactant ce service.

**Prêt transitoire et semi-permanent** Nous apportons une dernière distinction dans les types de transferts ou prêt de ressources. Nous disons qu'un transfert est *transitoire* lorsque ce prêt ne dure que pour la durée de l'appel. Il est *semi-permanent* lorsque ce transfert dure entre les appels.

Un exemple de prêt de ressource transitoire est le droit fourni au service de pouvoir répondre au client (e.g. resume capability dans EROS [SFS96] ou reply port dans Mach [Loe92]). Un exemple de prêt de ressource semi-permanent est le prêt de

mémoire pour en faire des buffers réseau dans EROS [SSS04]. Un autre est le prêt de mémoire au noyau dans seL4 [EDE08].

Un autre exemple de prêt de ressource transitoire est le prêt d'une pile par le client dans Pebble [GSB+99]. Lorsque l'appel de service se fait en prêtant son temps CPU, i.e. par thread-tunnelling, il y a besoin de prêt transitoire de la mémoire car sa consommation augmente avec le nombre de clients simultanéments présents dans le service. Pour les communications synchrones et asynchrones, le nombre de threads dans le service est fixe, donc l'allocation peut se faire en utilisant les ressources du service. En général, le thread tunneling augmente le besoin en prêts transitoires, comme nous le verrons en section 3.4.

Notons que vu l'overhead causé, il est difficile de faire du transfert temporaire transitoire.

**Sélection des ressources prêtées** Un autre critère de catégorisation est le mode de sélection des ressources prêtées. Nous pouvons en distinguer trois :

- implicite : le service puise directement dans les ressources du client ;
- par conteneur : le service puise les ressources depuis un conteneur transféré par le client ;
- explicite : le client indique exactement les ressources qu'il prête au service.

La différence entre types de sélection concerne d'abord la sécurité : dans le transfert implicite, le service peut utiliser toutes les ressources du client. Cependant, le client doit généralement faire confiance au service : s'il lui fait une requête, c'est généralement parce qu'il a besoin de la fonctionnalité qu'il fournit. Cette sécurité est donc généralement seulement une sécurité supplémentaire contre un bug du service.

La deuxième différence est le degré de connaissance nécessaire des besoins du service. Dans l'implicite, aucune connaissance n'est nécessaire ; pour le transfert par container, une connaissance d'une borne supérieure est nécessaire ; pour l'explicite, il faut connaitre précisément quelles sont les ressources dont le service a besoin. Par exemple, pour un transfert de mémoire, il faudrait connaitre exactement le nombre de pages à transférer.

Enfin, la dernière différence est l'accès à la connaissance « bas-niveau » de l'utilisation des ressources. Pour un transfert explicite de mémoire, il faut savoir quelles sont les pages libres que l'on peut transférer ; ces connaissances sont disponibles sur un système où les ressources sont « autogérées » comme les exonoyaux, mais pas dans un système qui ne gère que des abstractions de haut niveau comme UNIX.

Un exemple de transfert de ressource implicite est l'allocation de table des pages dans UNIX : si une application augmente la taille de son espace d'adressage (e.g. avec `sbrk`), la mémoire pour les tables de pages éventuellement nécessaire serait directement décomptée de l'application.

Des exemples de transferts par container sont les containeurs de Banga et al. [BDM99] les activités de Rialto [JLDJB95], ou les space bank de KeyKOS et EROS [SA02].

Un exemple de transfert explicite est le prêt de mémoire au noyau dans seL4 [EDE08] ou la transformation de pages dans Xen [BDF+03, § 3.3.3].

Notons enfin que le prêt par conteneur peut conduire à de la fragmentation interne lorsque toutes les ressources prêtées ne sont pas utilisées. Notons également que le transfert explicite de temps CPU consisterait à ce que le client indique les cycles où le service peut s'exécuter. Ce n'est ni pratique, ni possible sur les processeurs à cache actuels. Le modèle de thread tunneling présenté section 2.3.2.3 est un transfet implicite de temps CPU ; un transfert par containeur pourrait être obtenu en rajoutant des timeouts.

**Différences avec le décompte au client des ressources utilisées par le service** Le décompte au client des ressources utilisées par le service est un mécanisme similaire au transfert de ressource. Des exemples de tels mécanismes existent pour Mach [MST93] et L4 [SWH05]. En résumé, cette technique permet de prolonger la technique de comptabilisation des ressources pour que les ressources utilisées par le service pour servir un client soient décomptées comme appartenant au client, mais sans pour autant conserver la décision d'allocation. Elle ne permet donc pas de garder l'allocation déterministe. La différence entre décompte au client des ressources utilisées par le service et prêt de ressource, est la même que celle entre comptabilisation des ressources et allocation déterministe.

Comme on l'a vu en section 2.1.3.3, les mécanismes utilisés pour la comptabilisation ou l'allocation déterministe de la mémoire (ou autre ressource partagée spatialement) sont relativement proches.

Par contre, ils sont différents pour le temps CPU. Ainsi, les mécanismes pour décompter le temps passé par le service à traiter les requêtes clients [MST93, SWH05] ne prennent pas en compte le moment où la requête est traitée. C'est la raison pour laquelle ces systèmes nécessitent de définir séparément la politique d'ordre de traitement des requêtes, ce qui complexifie l'analyse.

Par opposition, le prêt de temps CPU permet de conserver les décisions d'allocation et de garder l'allocation déterministe. Notons que l'utilisation du thread tunnelling n'implique pas d'avoir une allocation déterministe ; ce mécanisme peut également être utilisé pour faciliter la comptabilisation du temps CPU [FL94].

#### 2.3.2.5 Conclusion sur l'appel de service et le prêt de ressource

Cette section a présenté, pour les trois types de communications avec un service partagé, les manières de garder l'allocation prévisible. La première consiste à provisionner le service partagé en ressource, puis à définir avec précaution comment distribuer ces ressources entre les différents clients. La deuxième est fonctionellement équivalente à protéger le service avec un mutex. Ces deux solutions complexifient l'allocation, ont un overhead important sans l'emploi de techniques complexes, et imposent un compromis entre performance et sécurité.

Au contraire, la troisième solution, le thread tunnelling, rend l'utilisation du service partagé indépendante de l'ordonnancement, ce qui simplifie grandement ce dernier. Elle est simple, a un overhead petit, et permet l'exécution parallèle des requêtes. Plus généralement, nous avons identifié la technique du prêt de ressource,

qui rend l'utilisation d'un service indépendant des décisions d'allocation, et est à la fois sécurisée *et* efficace. Mais elle va poser des problèmes d'ingénierie qui occuperont une grande partie de cette thèse.

En effet, il existe des systèmes qui ont implémenté des mécanismes de prêt de ressource (par exemple pour la mémoire, les exonoyaux permettent de prêter de la mémoire au noyau [EKJO95], L4 permet de prêter de la mémoire pendant les appels de service [Lie95a], ou par l'intermédiaire des pagers [LIJ97], et surtout EROS permet de prêter de la mémoire (et d'autres ressources) grâce son modèle de contrôle d'accès par capacités [SSS04]). Mais aucun système n'est fondé sur l'utilisation *systématique* du prêt de ressource, et n'a eu a affronter les difficultés spécifiques associées à cette contrainte (comme la gestion de la révocation des ressources prêtées pendant que le service est en train de l'utiliser), dont nous traiterons au chapitre 3.

## 2.4 Flexibilité et extensibilité

Sécurité et allocations sont les principales préoccupations du système d'exploitation. Mais comme tout programme, l'écriture d'un OS pose des problèmes de génie logiciel ; en particuler il est important de pouvoir adapter l'OS aux différents besoins, i.e. de le rendre *flexible*. Cela passe par l'*extensibilité* de l'OS, i.e. le rajout d'extensions qui ajoutent les fonctionnalités nécessaires.

La flexibilité est un thème de recherche important dans le domaine des OS. Son but est de permettre le changement des *politiques*, qui sont les choix du système d'exploitation qui affecte tout le système. Un système flexible peut plus facilement être réutilisé, ce qui induit des économies pour la conception, l'implémentation, et la vérification.

politiques

La flexibilité concerne l'allocation des ressources : nous avons vu qu'il y avait de nombreux choix de politiques d'allocations possibles. Il n'y a pas de politique d'allocation qui convienne à toutes les situations. Il est important que le système permette de changer de politique d'allocation, afin qu'il puisse être adapté aux différents besoins. Éventuellement, il pourra supporter plusieurs politiques simultanément, pour permettre l'exécution simultanée d'applications aux besoins divers.

La flexibilité concerne également la sécurité, car il existe également différentes politiques de sécurité. De plus, il faut garantir quand on ajoute une extension que leur portée est limitée, i.e. qu'elle ne perturbera pas le reste du système.

Enfin, la flexibilité et l'extensibilité offrent des avantages concernant le génie logiciel : l'OS peut avoir un code de base réduit [SESS97], qui permet seulement l'ajout d'extensions, et le reste du système est une composition de ces extensions.

Les principaux types d'extensions qui nous intéressent sont la prise en compte de nouvelles ressources (i.e. nouveau matériel), de nouvelles politiques d'allocations et de sécurité, et l'émulation d'environnement ou API pour pouvoir exécuter des applications existantes (e.g. on veut pouvoir réutiliser des applications POSIX[Ope], ARINC[Aer], ou OASIS [DAL$^+$04] sans avoir à (trop) les modifier).

Cette section présente les solutions proposées pour ce problème de l'extensibilité. Des revues plus complètes peuvent être trouvées dans la revue de Seltzer et al. [SESS97] ou la thèse de Rippert [Rip03].

### 2.4.1 Notions

#### 2.4.1.1 Degrés de flexibilité

Seltzer et al. [SESS97] classifient les systèmes extensibles en trois degrés de flexibilité :

- systèmes paramétrables : le code et le comportement peuvent être modifiés au moment de la compilation du système (i.e. le changement nécessite un reboot) ;
- systèmes reconfigurables : le code est fixé, mais le comportement peut varier dynamiquement. C'est à dire que le système possède plusieurs choix de comportements, que l'ont peut sélectionner dynamiquement ;
- systèmes extensibles : on peut créer des nouveaux comportement dynamiquement.

Ces choix permettent différents degrés de flexibilité. Pour des systèmes spécialisés comme dans l'avionique ou l'automobile, le choix d'un système paramétrable est largement suffisant ; mais pour des systèmes plus dynamiques, comme des PDAs ou ordinateurs personnels, où de nouvelles applications peuvent être ajoutées dynamiquement, le système extensible est mieux adapté.

Le problème de cette classification est qu'elle ne prend pas en compte l'étendue des modifications de comportement possible. Par exemple, il sera très difficile de modifier UNIX pour faire de l'ordonnancement statique, ou de l'allocation mémoire statique, parce qu'il n'est pas prévu pour cela (`mallocs` et sémaphores dans le noyau par exemple). Il est également difficile d'en changer la politique de sécurité.

#### 2.4.1.2 Sécurité

La possibilité de modifier le comportement du système d'exploitation pose naturellement un problème de sécurité : par exemple une extension pourrait modifier le système pour ne plus qu'il exécute les tâches critiques, ou faire planter le système. Il faut distinguer deux problèmes.

**La limitation de la portée de l'extension** Il faut par exemple éviter que les applications non-critiques puissent modifier l'OS d'une manière qui affecte une tâche critique. Cette question, assez peu étudiée, rejoint finalement la notion de contrôle d'accès : la possibilité de modifier le comportement du système vis à vis d'une application donnée est un privilège. Usuellement, l'extension se fait soit par le biais du superutilisateur qui peut modifier l'OS a volonté, soit par l'ajout d'extensions par une application dont la portée est limitée à cette application. Cependant, la structuration hiérarchique de système (§ 2.4.3.2) permet d'avoir des extensions qui portent sur une partie seulement des applications.

**Protection contre les erreurs** Il faut s'assurer que l'extension n'a pas d'impact en dehors de son champ d'application ; e.g. qu'une extension pour une tâche seule ne puisse pas faire tomber tout le système. Les techniques pour cette protection se regroupent en 3 [SESS97] : faire confiance au code, protection logicielle (utilisation

de langages sûrs généraliste (e.g. [BSP+95]) ou ad-hoc (e.g. [EKJO95, §5.6]), de techniques de scan de code [Rip03]) ou protection materielle (utilisation de la MMU pour protéger le système contre l'extension, comme c'est le cas pour les services des micronoyaux, les libOS des exonoyaux, ou les systèmes virtualisés des VMMs).

**Application aux environnements critiques** Ces deux critères sont importants pour les environnements critiques : lorsqu'ils sont garantis, ils devraient permettre une validation indépendante des composants. En effet, quand une extension ne modifie pas le comportement du reste du système, son ajout ne demande pas à revalider le reste du système. Les extensions peuvent ainsi être certifiées de manière indépendante.

Notons cependant qu'en général, cette sécurité ne concerne que le comportement du système, et pas l'allocation des ressource. Ainsi il est toujours possible pour une nouvelle extension de modifier l'allocation des ressources. Pour les systèmes pour lesquels l'allocation des ressources est importante, cette certification indépendante demande à ce que l'ajout de nouveaux composants ne modifie pas l'allocation des ressources, ce que nous proposons au chapitre 3.

### 2.4.2 Approches pour assurer l'extensibilité

#### 2.4.2.1 Modification du noyau

L'approche traditionnele de l'extensibilité est de modifier le code du noyau pour y ajouter des nouvelles fonctionnalités, puis de recompiler le noyau. Les systèmes UNIX-like ont longtemps utilisé cette méthode. Cette méthode à le désavantage qu'avec le temps, le code du noyau grossit énormément, ce qui augmente la quantité de bug (et rend impossible la certification), et complexifie la maintenance. C'est ce qui à motivé la recherche sur les systèmes extensibles [SESS97].

#### 2.4.2.2 Chargement de code dans le noyau

Puisque dans beaucoup de systèmes, le noyau est le coeur du système d'exploitation, pour changer le comportement du système il faut charger du code dans le noyau. Cette approche est relativement classique : Windows et Linux permettent ainsi à leur administrateur d'ajouter dynamiquement des modules, i.e. du nouveau code au noyau [BC05]. D'autres systèmes utilisent des mécanismes de protection logicielle pour ce faire (e.g. [BSP+95, EKJO95, SESS96]).

Notons que pour les micronoyaux, beaucoup de fonctionnalités se trouvent déportées dans des tâches de confiance au lieu du noyau ; mais on peut également envisager charger du code dans ces tâches de confiance.

Ces approches sont difficilement envisageable pour du code de haute sécurité. Elles souffrent des problèmes liées à la protection logicielle (§ 2.2.3.1) : la difficulté à certifier les logiciels (compilateurs de confiance, etc.) qui fournissent la protection logicielle. De plus, les codes de confiance posent le problème de la certification, pour un binaire donné, que ce code est bien de confiance, ce qui pose problème (on est souvent obligé d'incorporer un compilateur dans le noyau). Il est donc difficile de

permettre une telle approche, sauf éventuellement pour certaines créations de codes simples (e.g. les quajects de Synthesis [Mas92], ou pour des packet filters [EK96]).

#### 2.4.2.3 Micronoyau

Les micronoyaux représentent une approche pour l'extensibilité : l'idée est qu'on peut modifier une fonctionnalité fournie par le noyau en changeant le service correspondant. Le système Mach a ainsi été construit dans ce but [ABG+86, §2].

L'avantage du micronoyau est que l'ajout d'extension est sécurisé. Un bug dans une extension n'affecte que les utilisateurs de cette extension. Notons toutefois que le nombre d'utilisateurs affectés peut être élevé : un bug dans un service gérant l'écran, le clavier ou le réseau peut compromettre la mission du système.

Notons que l'utilisation d'un micronoyau ne facilite *pas* en elle-même la flexibilité : tout ce qu'on peut faire avec un micronoyau est réalisable avec un noyau monolithique qui charge du code dynamiquement. L'intérêt du micronoyau est seulement la sécurité dans l'ajout d'extension, et cela est plus difficile (à cause des besoins d'interfaçage) à faire qu'avec des modules chargeables dans un noyau monolithique. En contrepartie, cela force la conception à être modulaire.

L'extensibilité est également limitée en ce que les programmes clients sont forcés d'utiliser un certain nombre de services. Ainsi, un service réseau qui fournit une interface socket BSD force tous les programmes clients à utiliser cette interface. Les partisans de l'approche micronoyau estiment qu'il faut trouver la bonne abstraction comme interface pour les services systèmes [Lie95a, p. 12] ; les partisans de la paravirtualisation et des exonoyaux estiment que l'interface doit être proche (mais différente) du hardware.

#### 2.4.2.4 VMM et émulation

Les machines virtuelles sont une approche de l'extensibilité. En exposant l'interface hardware, elles permettent d'exécuter simultanément différentes applications dont les besoins diffèrent, en exécutant tout le système d'exploitation pour lequel elles ont été conçues. L'avantage de cette technique est qu'elle permet de faire tourner différentes applications aux besoins différents sans travail de portage. Les inconvénients sont le coût en performance que cette technique impose, et la difficulté de coordonner les différents systèmes, par exemple pour combiner les différents ordonnanceurs [26]. De plus, les politiques d'allocations des VMMs eux-mêmes sont elles-mêmes souvent peu paramétrables.

L'usage de la paravirtualisation (e.g. [BDF+03, WSG02]) permet de diminuer l'overhead de l'usage d'un VMM, mais demande de porter l'OS cible pour le VMM. Cette technique ne permet pas non plus d'atteindre le même degré d'intégration qu'un OS traditionel, en particulier concernant le partage des ressources entre les différents OS virtualisés. Ces problèmes ont été détaillés section 2.3.

Notons qu'on peut utiliser un micronoyau comme VMM : le micronoyau L4 a notamment été utilisé dans ce but [HHL+97]. Il est également possible d'émuler une

[26] [EKJO95, §8] fournit également un exemple de mauvaise coordination des mécanismes de swapping

API de haut niveau (comme POSIX) système grâce à l'usage de librairies ou de tâches systèmes dédiées (e.g. [KEG+97, ABG+86, BFF+92]).

### 2.4.3 Principes pour la flexibilité

Nous présentons maintenant deux principes qui nous semblent fondamentaux pour la flexibilité du système. Cette liste n'est pas exhaustive ; en particulier nous en avons exclu toutes les techniques qui demandent une modification dynamique du code d'un domaine de protection. Ces techniques posent un problème de sécurité : la moindre faute dans le code qui installe la modification du code peut avoir des conséquences imprévisibles. Cela les rend difficile à certifier.

#### 2.4.3.1 Separation de la politique et du mécanisme

**Présentation** Le principe de *séparation de la politique du mécanisme* a été énoncé par Levin et al. pour le système Hydra [LCC+75]. Mais cette idée remonte au moins à Brich Hansen [Han70].

Le système d'exploitation définit des règles qui s'appliquent à toutes les tâches du système. Ce principe demande la séparation entre les politiques « spécifiques », qui peuvent changer selon l'utilisation qu'on a du système, et les mécanismes « génériques » qui s'appliquent à toute utilisation et servent de support à l'implémentation des politiques.

**Application à la sécurité** Une application de ce principe à la sécurité est la séparation de la protection et du mécanisme [WCC+74]. Nous avons vu (§ 2.2.2) que la sécurité reposait sur des principes communs, comme la séparation en domaines, le contrôle d'accès... Tout système d'exploitation pour la sécurité doit donc implémenter ces mécanismes. Au contraire, le contrôle sur les communications entre les tâches dépend des utilisations.

Ainsi, on peut séparer le système en une partie « mécanismes protection » (qui implémente les domaines de protection, le contrôle d'accès et les liens de communication) et une partie « politique de sécurité » (qui peut être de la politique multiniveau militaire, ou tout autre politique). L'approche « distribuée » de Rushby [Rus81] et les systèmes à capacités [MS03] implémentent cette séparation, au contraire des « security kernels ».

**Application à l'allocation des ressources** Pour l'allocation des ressources, cela consiste à ce que les applications utilisateur puissent redéfinir la politique d'allocation pour une ressource, en ne gardant dans le noyau que les mécanismes permettant d'implémenter ces politiques.

Mais cette séparation est souvent incomplète. Ainsi, dans Hydra, certains mécanismes ne sont que des politiques paramétrables. Pire, chaque programme "définit" sa politique, donc le noyau contient toujours une politique pour garantir l'équité de l'allocation entre applications. Nous énonçons que la définition de sa propre politique est impossible, et ne peut être réellement qu'un moyen évolué de faire des demandes d'allocation.

Nous estimons que pour que la séparation politique et mécanisme soit complète, il est nécessaire que les politiques soient implémentées dans des domaines de protection privilégiés. La structuration hiérarchique que nous voyons dans la section suivante, est un moyen pour cette réalisation.

**Suppression des abstractions** Les exonoyaux [EKJO95, KEG+97, Ros95] et les VMMs [BDF+03] représentent une avancée dans la séparation de la politique et du mécanisme. L'idée est qu'une interface proche du hardware permet la plus grande flexibilité dans la définition de politique d'allocation des ressources [EKJO95, §8]. De plus, cela contribue à amoindrir la taille du noyau, ce qui tend à diminuer le nombre de politiques en général que le noyau impose aux applications.

Mais même ainsi, il reste toujours en pratique des politiques dans le noyau. Ainsi, la "politique racine" de chaque ressource reste gérée par le noyau [EKJO95, §3.1]. De plus, comme pour les systèmes traditionnels (§ 2.1.2.1), les idiomes de programmation usuels rendent le code et l'allocation de ressources très interdépendents (e.g. usage de FCFS, `malloc` et sémaphores dans le noyau, etc.).

#### 2.4.3.2 Structuration hiérarchique

**Présentation** Dans un exonoyau, on peut changer une décision de conception relative à une tâche en changeant sa "libOS", et changer une décision de conception (i.e. politique) relative à toutes les tâches en changeant le noyau. On se demande maintenant comment changer une politique relative à un sous-ensemble de toutes les tâches.

La structuration hiérarchique (voir Figure 2.3) apporte une réponse à cette question. L'idée générale est de structurer les tâches ou domaines de protection du système selon un arbre. Chaque noeud de l'arbre peut déterminer une politique, qui s'applique à tous les descendants de ce noeud (inversement, chaque noeud a sa politique définie par l'ensemble de ses ancêtres).

L'intérêt de la technique est de pouvoir faire coexister des politiques différentes sur des groupes de tâches différentes. Ces politiques sont elles-même coordonnées par le noeud qui est l'ancêtre commun à ces deux politiques. De plus, un noeud ne dépend que de ses ascendants directs, et pas de ses frères, oncles, grand-oncles... Cela permet de limiter, pour chaque tâche, l'ensemble des tâches auxquelles il faut faire confiance pour fonctionner correctement.

**Mono et multi-hiérarchie** La manière classique de procéder est de créer une hiérarchie de tâches : c'est ce que nous appellons le système mono-hiérarchique. Dans ce système, chaque tâche a un seul parent, qui peut changer à volonté les politiques de tous ses enfants. Les travaux de Brich Hansen [Han70], CAP [NW77], Fluke [FHL+96] ou Genode/Bastei [FH06] sont des systèmes structurés de cette manière. Les resource containers [BDM99] sont aussi structurés mono-hiérarchiquement, mais les noeuds de la hiérarchies ne sont pas des tâches.

Le problème de cette structure est que cela force des politiques indépendantes à coïncider : la relation hiérarchique doit être la même pour les domaines de protection, l'allocation mémoire, ou l'ordonnancement CPU. Cela limite la flexibilité du système

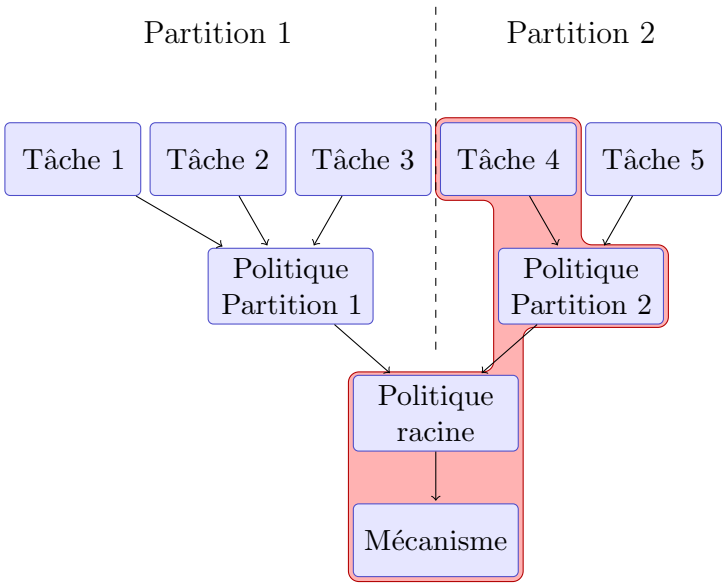



FIG. 2.3 – Structuration hiérarchique. La partie rouge est l'ensemble du code auquel le client 4 fait confiance pour correctement implémenter sa politique. En particulier, le code implémentant la politique de la partition 1 n'en fait pas partie.

[WCC+74, p. 338]. On obtiendrait donc une plus grande flexibilité en ayant des hiérarchies indépendantes pour chaque politique, ce que nous appellons *multi-hiérarchie*. Nous ne connaissons pas de système structuré de cette manière.

**Cas de l'allocation des ressources** La structuration hiérarchique est beaucoup utilisée pour la définission des politiques d'allocation de ressources. Par exemple, il est commun de réserver différents « pools » de ressources (e.g. mémoire), et d'allouer les ressources dans ces pools de manière différente. L'intérêt de cette technique est que l'allocation dans chaque pool se fait de manière indépendante les unes des autres, ce qui apporte sécurité et flexibilité dans l'allocation de ces pools. Mais la séparation en plusieurs pools peut également causer des problèmes de fragmentation, et donc une perte d'efficacité. Cela dépend de la coopération des politiques entre différents niveaux. Il peut donc y avoir compromis entre performance et sécurité/flexibilité.

**Ordonnancement hiérarchique** Les types de conditions sur les temps d'exécutions peuvent beaucoup varier d'une tâche à une autre, comme en témoigne l'ensemble des solutions d'ordonnancement vues en section 2.1.3. C'est pourquoi la structuration hiérarchique s'applique souvent à l'ordonnancement.

La plupart des systèmes ne permettent cependant de composer que deux niveaux d'ordonnanceurs de type fixé. Dans l'avionique, on utilise en allocation racine un ordonnancement statique, et un ordonnancement par priorité fixe dans les partitions [Rus98, Aer, KW07]. Les VMMs permettent n'importe quel ordonnanceur en deuxième niveau, et fournissent une allocation équitable pour le niveau principal

[BDF+03][27]. Certains VMMs utilisent la priorité fixe (approche priorité du critique sur le non critique) comme premier niveau. SPIN permet également une allocation à deux niveaux [BSP+95, §4.2]

Les systèmes permettant de composer plus de niveaux d'ordonnanceurs sont plus rares. Nous analysons comme un des principaux problèmes la nécessité de notifier toute une branche d'ordonnanceur lorsqu'un thread redevient prêt (ce qui est fait dans [GGV96, §4] et [FS96, §3.2]). De plus ces systèmes ne permettent souvent pas de choisir le type d'ordonnancement pour tous les niveaux. L'ordonnanceur par bande passante CPU de Goyal et al. [GGV96] impose ainsi d'ordonnancer par découpage de bande passante sur les premiers niveaux.

Les travaux de Ford et Susarla [FS96] permettent une vraie composition d'ordonnanceurs arbitraires. Ils ont eu l'idée de rendre n'importe quel thread ordonnanceur en lui permettant de donner de son temps à un autre thread, et des techniques comme le CPU inheritance protocol. Cependant, beaucoup de temps n'est pas compté dans leur système (e.g. le temps de demande de temps CPU), ce qui ne convient pas pour du temps réel dur).

#### 2.4.3.3 Conclusion sur la flexibilité

Cette section a vu différentes approches qui permettent à un système d'exploitation d'être flexible et extensible de manière sécurisée, i.e. facilement adaptable à des besoins différents.

Entre autres techniques, on a vu que les micronoyaux permettaient de rajouter des extensions de manière sécurisée ; que la suppression des abstraction facilitait l'extensibilité de chaque application et supprimait une majorité de politiques de l'OS ; que la séparation de la politique et du mécanisme permettait de conserver un même mécanisme pour différentes politiques ; et que la structuration hiérarchique permettait d'avoir simultanément plusieurs politiques, en les faisant se coordonner. Toutes ces techniques ont été utilisées dans la conception de notre prototype.

Dans cette retrospective, nous nous sommes intéressé seulement à la possibilité de créer des extensions, et pas à la problèmatique de génie logiciel concernant l'interfaçage entre ces extensions. Roscoe [Ros95] et Rippert [Rip03] traitent de ce sujet en plus amples détails.

## 2.5 Résumé et conclusion

Dans ce chapitre, nous avons présenté tout un éventail de problèmes et solutions qui se posent pour l'intégration de système hétérogènes. Nous avons vu qu'il existait des solutions mûres concernant la sûreté de fonctionnement, et les aspects confidentialité et intégrité de la sécurité ; ainsi que pour l'ajout sécurisé d'extensions au système. Les systèmes basés sur les capacités ou MILS permettent d'assurer un très haut niveau de sécurité.

[27] cet ordonnanceur serait facilement modifiable, mais cela implique de modifier le code source du VMM [BDF+03, § 3.3.1]

Les questions de l'allocation de ressources et de la protection contre les dénis de service sont par contre insuffisants pour une intégration éfficace de systèmes hétérogènes. Les solutions actuelles utilisent généralement une politique d'allocation fixée, non-optimale, et souffrent de problèmes variés comme l'inversion de priorité. Cela empêche les systèmes d'exploitation de proposer une allocation des ressources simple, sûre et flexible.

Dans ce but, nous avons proposé deux concepts majeurs : l'allocation déterministe et le prêt de ressource, qui apportent des solutions à ces problèmes. Un système avec de telles propriétés pourrait répondre de manière optimale au problème de l'intégration de systèmes hétérogènes, comme nous l'avons vu en section 2.1.4. Encore faut-il concevoir et implémenter un tel système, et l'intégrer avec un grand respect de la sécurité.

Chapitre 3

# Structure et conception générale

Nous présentons dans ce chapitre la structure et conception générale d'un système d'exploitation pour l'intégration d'applications hétérogènes avec un haut niveau de sécurité, Anaxagoros. Le nom de ce système vient du philosophe grec Anaxagoros de Clazomènes, qui a dit « Rien ne naît ni ne périt, mais des choses déjà existantes se combinent, puis se séparent de nouveau »(formule reprise plus tard par Lavoisier). Cette phrase résume bien la manière dont sont gérées les ressources dans Anaxagoros ; en particulier les concepts de prêt de ressource et d'allocation déterministe, décrits dans le chapitre précédent.

**Contributions** Les contributions principales de ce chapitre central sont les suivantes :

- le principe d'indépendance des politiques d'allocation, qui fait de l'allocation des ressources une préoccupation séparée. Ce principe simplifie drastiquement les problèmes d'allocation en éliminant par exemple les problèmes classiques d'inversion de priorité et de déni de service, permet à l'allocation des ressources de devenir modulaire, et permet d'obtenir une allocation déterministe même en utilisant des algorithmes d'allocation complexes ;
- le design et l'implémentation du mécanisme de prêt de thread, qui implémente une communication où aucune ressource n'est utilisée par le service, lui permettant d'être invulnérable aux dénis de service ;
- le principe d'indépendance entre politiques de ressources différentes ; sa mise en œuvre par le découplage entre threads, espace d'adressages, et domaines de protection ;
- l'idée de partitionnement multi-hiérarchique (hiérarchies d'allocations indépendantes, et reposant sur la notion de partitionnement au sens mathématique), permettant de comptabiliser les ressources de manière modulaire et sécurisée ;
- l'implémentation de notre mécanisme de capacités, au design simple et flexible, économe en mémoire, qui ne nécessite pas de table d'objets centrale et dont toutes les opérations (y compris la révocation) se font en temps constant ;

- le principe d'utilisation de ressources par le système prédictible, et son implémentation où tous les objets systèmes sont de taille une page, et toutes les opérations systèmes sont en temps constant ;
- l'utilisation systématique du principe de report d'erreur paramétrable, sécurisé et flexible ;
- la méthode d'organisation des services pour lesquels le prêt de ressource est utilisé de manière systématique, et qui respectent le principe d'indépendance des politiques d'allocations.

## 3.1 Principes de conception

Notre conception est fondamentalement basée sur un ensemble de principes. Pour nous, ces principes ne sont pas seulement des guides pour la conception, mais des règles qui sont appliquées de manière systématique et intransigeante. Neumann note l'importance de suivre des principes avec discipline pour les systèmes qui fournissent une haute assurance en sécurité [Neu04, p. ix]. Shapiro et Hardy notent que le respect strict de principes permet d'avoir un système plus simple et plus consistant [SH02].

Nous présentons dans cette section la liste des principes que nous avons suivi, et les raisons pour lesquelles ils répondent à la problématique de cette thèse.

### 3.1.1 Indépendance des politiques d'allocations

Nous définissons le principe général d'*indépendance des politiques d'allocations*, qui regroupe toutes nos préoccupations de sécurité et de flexibilité dans l'allocation des ressources. Il énonce qu'une politique d'allocation devrait être définie par un module séparé (qu'on appelle module de politique), *et uniquement par ce module*.

Cela permet au problème de l'allocation des ressources de devenir, sinon simple, au moins une préoccupation séparée. En particulier, il devient possible d'avoir une politique d'allocation simple et lisible, et donc sûre.

#### 3.1.1.1 Principes

Ce principe général est mis en œuvre par différents principes, qui suppriment chacune une contrainte quant à l'application du principe d'indépendance des politiques d'allocations. Cette liste ne se prétend pas exhaustive, mais reprend les problèmes les plus courants.

**Principe : Identification de toutes les ressources** Il est en premier lieu important d'identifier tous les types de ressources, qui sont en quantité limitées, et qu'il faut donc allouer. Lorsqu'un type de ressource n'est pas identifié, cela signifie que son algorithme d'allocation est codé en dur dans le code, souvent avec FCFS ; et donc que le système est vulnérable au déni de service par accaparation de cette ressource.

Il faut également identifier toutes les ressources au sein d'un type. En particulier, les noyaux monolithiques peuvent allouer des ressources sans passer par l'algorithme d'allocation (par exemple, allocation de mémoire pour le noyau, ou de temps CPU pour les interruptions). En conséquence, ces ressources ne sont pas comptabilisées par l'algorithme d'allocation.

**Principe : Séparation complète du mécanisme et de la politique** Nous avons vu (§ 2.4.3.1) le principe de séparation de la politique et du mécanisme, appliqué à l'allocation des ressources. Il s'agit de séparer ce qui dépend de la manière dont sont attribuées les ressources (la politique) de ce qui en est indépendant (le mécanisme). Mais nous avons également vu que cette séparation était la plupart du temps incomplète ; il s'agit souvent seulement de politiques paramétrables ; ou alors le système demande aux applications de « définir elles-mêmes leur politique », tout en étant soumise à une « politique racine » garantissant l'équité fixée.

Nous proposons d'aller plus loin en définissant que la séparation de la politique et du mécanisme est *complète* lorsque le mécanisme permet d'implémenter n'importe quelle politique d'allocation. En d'autres termes, la séparation ne rend pas certaines politiques impossibles (par rapport à un OS où les politiques seraient intégrées).

En particulier, tout mécanisme dont l'interface est basée sur une abstraction ne peut pas implémenter une séparation mécanisme/politique complète. Par exemple pour le temps CPU, les priorité ou réservations sont des abstractions qui limitent la manière dont les ressources sont allouées. Pour implémenter une séparation complète, il faut que la politique puisse contrôler les intervalles d'exécutions de chaque tâche. Pour la mémoire, il faut que l'allocation se fasse à la granularité de la page, comme le font Xen [BDF+03] ou les exokernels [EKJO95, KEG+97].

Notons que la séparation entre politique et mécanisme n'indique pas où le module de politique doit être placé. Cela dépend des possibilités d'extensibilité du système. Le module peut ainsi être dans le même domaine de protection que le mécanisme (l'interface du mécanisme est simplement définie par les appels de fonctions), ou dans un domaine séparé, ou éclaté dans plusieurs domaines.

**Principe : Pas de modification des allocations par l'OS** Il arrive souvent qu'en accédant à une ressource, l'allocation d'une ressource non reliée en soit modifiée. Par exemple, une demande de socket dans un noyau monolithique peut créer une allocation mémoire par le noyau, qui peut à son tour causer l'activation du processus de swap : l'allocation mémoire et CPU s'en trouvent modifiées.

Un autre exemple courant est le besoin de synchronisation dans le noyau : un tâche voulant accéder à une structure protégée par un mutex peut bloquer si ce mutex est déjà pris, de manière tout à fait imprévisible. De la même manière, l'accès à un service partagé par IPC synchrone peut également bloquer de manière imprévisible.

D'où ce principe, qui interdit à l'OS d'être à l'initiative des modifications de l'allocation. Les modifications de l'allocation doivent toutes être contrôlées par le module de politique, qui est seul à décider de ces modifications.

Mais l'application peut être à l'initiative de la modification dans l'allocation, en faisant par exemple des demandes d'allocation au module de politique, ou en décidant

de bloquer volontairement. Notons que l'emploi d'appels systèmes bloquants (comme `read` dans UNIX) peut être vu comme une décision consciente de l'application de bloquer (l'application, en utilisant cet appel système, sait qu'il y a possibilité de bloquer). En revanche, `sbrk` (demande d'allocation mémoire) n'est pas sensé bloquer.

**Principe : Indépendance des politiques d'allocation de ressources de type différent** Les politiques gérant les ressources de type différent doivent également être indépendantes les unes des autres. Un bon contre-exemple est le processus UNIX, qui est à la fois l'unité de comptabilisation du temps CPU et de la mémoire (la situation est devenue plus floue depuis l'avènement du multithread).

Historiquement, les politiques d'allocation de la mémoire et du CPU étaient liées. On peut voir deux raisons à cela. La première est que la notion de processus vient de la volonté de découper une machine réelle en plusieurs « machines virtuelles », qui contiennent chacune une partie de la mémoire, une partie du CPU, et une partie des autres ressources, de manière liée. L'autre, plus pragmatique, était liée à la nécessité de faire du swapping sur les premiers systèmes interactifs (e.g. CTSS [CMDD62]), quand la quantité de mémoire disponible était extrêmement faible : ordonnancer un programme impliquait qu'il soit sorti du swap.

Mais le swapping est à prohiber pour les systèmes temps réel. Sur les machines modernes, il n'y a plus aucune raison pour que les réceptacles des différentes ressources aient des politiques liées.

**Principe : Indépendance par rapport aux domaines de protection** Il faut pouvoir allouer les ressources de manière indépendante des domaines de protection. Le modèle UNIX assume que le processus est l'unité de partage du temps CPU, du partage de la mémoire, et l'unité de protection du système. Or, on a vu (§ 2.3) qu'il était important de pouvoir comptabiliser les ressources de manière indépendante du domaine de protection (à une échelle inférieure ou supérieure), et surtout lors de la communication avec un service partagé (mécanisme de prêt de ressource).

Ce dernier principe contribue à la simplicité et efficacité de l'allocation des ressources, en supprimant beaucoup de besoin de sous-pools et de sous-politiques (§ 2.3.2.1), et en permettant des politiques transverses indépendantes des appels de service.

En particulier, on évite les « plafonds » sur les créations d'objets : les objets de haut niveau sont toujours créés à partir de ressources de plus bas niveau (par exemple, les domaines de protections sont créés à partir de mémoire). Ainsi, il n'y a pas besoin de se préoccuper de l'allocation de ressources pour ces objets de haut niveau (par exemple, pas besoin de limiter la création de tâches) : cette création n'est limitée que par le nombre de ressources fournies par les modules de politique à l'application.

#### 3.1.1.2 Intérêts

**Allocation déterministe** Grâce à l'application de ce principe, l'allocation des ressources devient déterministe. Cela permet de prédire qu'une tâche critique disposera toujours d'assez de ressources pour s'exécuter (§ 2.1.3.3). Il est par exemple

possible d'utiliser une politique d'allocation des ressources statique, tout en permettant à l'usage de ces ressources de varier dynamiquement.

Notons cependant qu'il est toujours possible de faire de l'allocation de manière indéterministe : par exemple en exécutant un système existant virtualisé. Mais ce qui importe est la *possibilité* d'avoir des décisions d'allocations qui soient déterministes, surtout pour celles concernant les tâches critiques.

**Simplicité et lisibilité** Le deuxième intérêt est la simplicité et lisibilité des politiques. Comme toute la politique pour une ressource est définie par le module de politique, cela permet de savoir simplement comment les ressources seront allouées : l'allocation des ressources devient une préoccupation séparée. Par opposition, dans les systèmes classiques, il faut connaître des informations sur l'ensemble du système (en particulier, les appels qui peuvent bloquer) pour pouvoir faire une prévision.

De plus, comme on évite beaucoup de sous-pools et sous-politiques, cela simplifie la gestion des ressources. Ainsi, on pourra dire qu'on a attribué 100 ko à une application, plutôt que de totaliser qu'une application utilise 50ko pour ses données, 12ko de table des pages, 8ko pour le domaine de protection, et 20ko de buffers réseaux.

**Flexibilité** Enfin, cette indépendance permet une grande flexibilité dans les politiques d'allocation. Même s'il était déjà possible de définir des politiques dans des modules externes (§ 2.4.3.1), on était nécessairement limité dans l'étendue des politiques possibles par le fait que toute la politique n'était pas définie par ce module ; l'indépendance des politiques résout ce problème. On obtient ainsi une approche modulaire de l'allocation des ressources.

### 3.1.2 Assurance d'une exécution correcte

Les principes que nous abordons maintenant permettent d'assurer à une tâche une exécution correcte, une fois que la tâche dispose d'assez de ressources et ne commet pas de fautes. En particulier, ces principes assurent que la présence d'autres tâches (critiques ou non) ne peut pas faire défaillir cette tâche.

#### 3.1.2.1 Respect des principes fondamentaux de sécurité

Les principes de sécurité fondamentaux tels qu'énoncés par Saltzer et Schroeder (§ 2.2.2), doivent être scrupuleusement suivis pour atteindre une haute garantie d'exécution. Nous accordons une importance particulière aux principes de séparation des privilèges, médiation complète, défaillance sûre par défaut, principe du moindre privilège, et minimisation des mécanismes communs, que nous prenons en compte dans la structure du système.

Nous prenons également en compte le principe de conception ouverte (cette présente thèse, étant publique, en est la preuve), et de sécurité prévue au départ. L'acceptation psychologique est moins importante puisque dans les systèmes embarqués, on n'a pas vraiment d'interface avec l'utilisateur ; cependant nous avons

fait en sorte que les concepts du système soient naturels et faciles à appréhender pour un développeur d'application.

#### 3.1.2.2 Principe : Ressources nécessaires prévisible

Pour pouvoir prévoir à l'avance que l'exécution d'une tâche va bien se dérouler, le fait de pouvoir choisir l'allocation des ressources ne suffit pas : il faut également pouvoir prévoir de combien de ressources la tâche a besoin pour s'exécuter. Cette question concerne essentiellement les besoins des tâches ; mais il se pose également la question du nombre de ressources dont ont besoins les services partagés utilisés par ces tâches.

**Mémoire** Pour la mémoire, il est nécessaire que la tâche sache répondre à des questions comme « combien de mémoire est nécessaire pour créer une nouvelle tâche ? un nouveau thread ? un nouvel espace d'adressage ? un buffer réseau ? ». La mémoire nécessaire peut ainsi être calculée en fonction des besoins de la tâche, ce qui évite de sur-réserver des ressources. Dans notre prototype Anaxagoros, quasiment tous les objets systèmes sont de taille 1 page ; sauf les buffers prêtés aux services (pour le réseau, etc.) qui sont de taille arbitraire, choisie par l'application.

**Temps CPU et protection contre les « ralentissements de service »** Pour le CPU, il faudrait idéalement qu'on puisse prévoir le temps CPU nécessaire pour exécuter les différents appels de services. Prévoir le temps CPU utilisé est quelque chose de complexe, notamment à cause de la présence de caches sur les processeurs. Pour ne pas se soucier de ce problème pendant l'implémentation de l'OS, nous dirons que *le nombre d'instructions exécutées doit être prévisible*, et non le nombre de cycles. L'idée est que si on peut donner un majorant au nombre d'instructions exécutées, c'est que l'on connait les différents chemins d'exécutions, et qu'une analyse poussée permettrait de donner un majorant du nombre de cycles utilisés.

Cela ne signifie pas pour autant qu'il faille compter le nombre d'instructions traversées par chaque appel système ; cela signifie qu'il doit être possible de donner un majorant à ce nombre d'instructions si nécessaire.

En particulier, *le temps d'exécution d'une tâche ne devrait pas dépendre de l'exécution des autres tâches* (ou de manière bornée). C'est à dire que *les tâches doivent être non seulement protégées contre le déni de service, mais également contre tout « ralentissement » de service.* Cette condition est donc très stricte. Un autre exemple de temps d'exécution qui dépend des autres tâches est l'usage d'un cache logiciel global. Les tâches temps réel ne devraient pas dépendre de l'utilisation d'un tel cache (et devraient par exemple se constituer leur cache elle-même).

Notons que les majorants sur le nombre d'instructions traversées dépend de la complexité théorique de l'appel système. Si l'appel système est en $\mathcal{O}(1)$, ce majorant sera une constante $a$. Si il est en $\mathcal{O}(n)$, il sera de la forme $a*n+b$, si en quadratique de la forme $a*n^2+b*n+c$, etc. Dans Anaxagoros, *le temps d'exécution de quasiment tous les appels de service est en temps constant (*$\mathcal{O}(1)$), donc hautement prévisible ; y compris pour la récupération de ressources ou terminaison d'autres tâches.

Notons qu'il faut également prévoir le temps CPU nécessaire pour les « processus systèmes » (par exemple, le filtre de paquets réseaux) ; tout comme il faut prévoir la mémoire pour les services du système.

La question de la prévisibilité des caches est pour nous une préoccupation séparée. Elle est de la responsabilité du logiciel par les politiques d'allocation CPU (e.g. quotas de temps assez longs), et mémoire (e.g. page coloring [LHH97]), et du calcul de WCET.

**Révocation immédiate** Une opération particulière qui devrait se faire en temps prévisible est la suppression d'une tâche, i.e. la récupération de toutes ses ressources. C'est un cas particulier de la récupération d'une partie des ressources accordée à une tâche. La récupération d'une ressource d'une tâche consiste à révoquer à la tâche l'usage de cette ressource.

Dans Anaxagoros, nous avons choisi qu'autant que possible la révocation se fasse en temps constant court. Cette décision offre plusieurs avantages : le redémarrage d'une tâche se fait en temps court, ce qui est important pour le recouvrement d'erreur [Den76] ; il est également important que la suppression de programmes qui tentent un déni sur les ressources se fasse en un temps court [SP99, § 1]. De plus, cette décision simplifie et rend plus efficace l'allocation des ressources : par exemple, on peut donner beaucoup de mémoire aux tâches non-critiques si on sait qu'on pourra reprendre cette mémoire rapidement si une tâche critique en a besoin.

Ce principe apporte cependant des difficultés d'implémentation ; en particulier, lorsqu'on supprime une tâche qui avait prêté des ressources à un service partagé, le service partagé doit être prêt à ce que ces ressources soient soudainement révoquées. Nous appelons cela la « révocation forcée ». Cette décision impacte fortement la conception des services (§ 3.5).

#### 3.1.2.3 Principe : Erreurs prévisibles

Ce principe énonce tout simplement qu'une application devrait pouvoir prédire si un appel système va réussir. En particulier, cela ne devrait pas dépendre des agissements des autres tâches. La seule exception, importante, est lorsqu'une demande d'allocation est refusée.

Regardons l'appel système `mmap` sous Linux : il peut échouer pour une quantité de raison, certaines prévisibles (file descriptor invalide, protection incorrecte), certaines non (nombre maximum de mappings du processus excédé, limite sur le nombre de fichiers ouverts dans le système excédée).

Dans Anaxagoros, les seules demandes qui peuvent échouer sont les demandes d'allocation. On sépare les appels de service pour les demandes d'allocation et d'utilisation des ressources. Ainsi, lorsqu'on a toutes les ressources nécessaires, *tous les appels de service service réussissent*, sauf en cas d'utilisation incorrecte (mauvais paramètres, appels dans le mauvais ordre, etc.). Cela permet de garantir à une tâche critique dont toutes les ressources ont été allouées qu'elle va toujours exécuter son traitement correctement, même lorsque celui-ci demande de faire des appels de service.

Ce principe permet également d'exécuter plusieurs appels systèmes en lot plus facilement, puisqu'on sait que les appels vont réussir. Nous exploitons cette propriété pour le mécanisme de multicall (§ 3.3.3.2).

### 3.1.3 Détection et recouvrement des erreurs

La détection d'erreurs est du domaine du système d'exploitation ; en particulier chaque service est le mieux à même de détecter les erreurs le concernant. Par contre, le recouvrement des erreurs est spécifique aux tâches et doit être déporté de manière sécurisée à l'espace utilisateur pour plus de flexibilité. D'où ce principe, qui fournit à l'OS la possibilité de le faire.

#### 3.1.3.1 Principe : Report d'erreur paramétrable

Il est fréquent pour le système d'exploitation de pouvoir détecter une erreur du programme. Dans Anaxagoros, nous vérifions systématiquement que le programme agit en conformité avec ce qu'il est censé faire. Notamment, on vérifie la conformité de tous les arguments des appels de service (valeur bien comprise dans un intervalle, pointeur correct, etc.), ou que certaines actions se font de manière séquentielle.

Lorsque le système d'exploitation détecte une telle erreur, on peut avoir différentes approches :

- *terminer le programme.* Pour certains systèmes critiques, cela peut ne pas être une option, et on préférerait mettre en œuvre un recouvrement (voir e.g. [Den76]) ;

- *corriger immédiatement l'erreur* par l'OS (lorsque c'est possible). Mais l'OS ne peut corriger les fautes que localement ; souvent quand le programme commence à faire des fautes, il va avoir un comportement complètement incorrect, et on pourrait préférer passer dans un mode dégradé par exemple. De plus, passer sous silence les erreurs prive le programmeur d'un outil pour la conception des programmes ; celui-ci aura plus de mal à trouver l'origine d'une erreur ;

- *signaler l'erreur au programme.* Par exemple les appels systèmes POSIX renvoient `EINVAL` lorsqu'un argument est incorrect, et reçoivent un signal `SIGILL` lorsqu'ils exécutent une instruction incorrecte. Les exokernels reçoivent leurs propres exceptions [EKJO95]. Cette approche permet au programme de faire un recouvrement adapté si nécessaire. Mais cela est problématique si le programme est dans un état complètement chaotique, car il est alors incapable de s'auto-réparer. Denning [Den76, p. 361] énonce ainsi que les « algorithmes de vérification doivent être indépendant du système vérifié, ou une erreur pourrait empêcher sa propre détection » ;

- *renvoyer l'erreur à un gestionnaire d'erreur externe.* Ce gestionnaire d'erreur peut être une tâche de confiance, qui met en œuvre le mécanisme de recouvrement choisi (redondance, mode dégradé, etc.). On peut également modifier la gestion de l'erreur en modifiant seulement ce gestionnaire (et sans modifier le

programme qui peut commettre l'erreur), ce qui rend la gestion d'erreurs plus modulaire.

Cette approche est la préférable pour les tâches critiques, qui ne sont pas censées commettre d'erreur.

Même si la dernière option est préférable pour des tâches critiques, de nombreux programmes sont conçus pour gérer eux-mêmes leurs erreurs, ou que leurs erreurs soient corrigées par l'OS. Nous proposons pour cela que la gestion des erreurs soit paramétrable, i.e. qu'on puisse configurer pour chaque « type » d'erreur la tâche qui en soit avertie. Par exemple, pour chaque exception (défaut de page, division par zéro, instruction illégale ...), un pseudo appel de service est exécuté qui renvoie le thread dans le gestionnaire correspondant. Ce mécanisme est similaire aux « domaine keeper keys » dans Keykos [Har85]. Les « pagers » de Mach [ABG+86] suivent une approche similaire, mais limitée à la seule exception « défaut de page ».

Comme nous l'avons vu précédemment, si un appel de service renvoie une erreur c'est parce que la tâche n'utilise pas le service de manière correcte. Des exemples de telles erreurs sont `EBADARG` (argument invalide) ou `ECONCURRENCY` (détection de concurrence interdite). Il faudrait pouvoir rajouter optionellement un appel à un gestionnaire d'erreur lorsqu'une telle erreur est détectée (nous ne l'avons pas implémenté, bien que cela soit simple à effectuer).

Notons qu'il peut parfois être plus efficace (en temps CPU) de corriger l'erreur de manière silencieuse plutôt que de la détecter. Ainsi, la vérification qu'une valeur $x$ appartient à un intervalle peut parfois être remplacée par la transformation de $x$ à l'aide d'un masque. Dans la mesure du possible, il faut éviter de recourir à cette technique ; ces micro-optimisations peuvent éventuellement être faites de manière conditionnelles.

### 3.1.4 Flexibilité et extensibilité sécurisée

Les principes suivants sont importants pour pouvoir adapter le système aux besoins, et l'étendre sans compromettre l'existant.

#### 3.1.4.1 Principe : Indépendance des politiques de sécurité

Tout comme le système est indépendant des politiques d'allocation, il devrait également être indépendant des politiques de sécurité. Rushby a montré qu'on pouvait exprimer des politiques de sécurité selon une formulation basée sur la « non-interférence » dans laquelle la relation d'interférence (e.g. possibilité de communiquer) pouvait être non-transitive [Rus92]. En pratique, cela peut être réalisé par le contrôle des canaux de communications (channel control) et l'utilisation de tâches de confiances qui agissent comme des filtres intermédiaires. Si on suppose que toute politique peut être exprimée à l'aide de cette formulation, il suffit au noyau d'assurer la séparation des tâches et de fournir un moyen à l'espace utilisateur de contrôler les communications.

Les politiques de sécurité peuvent être construites statiquement en insérant des programmes de confiances qui jouent le rôle de médiateur [Rus81, AFHOT06,

BDRS08], ou en créant dynamiquement des nouveaux canaux et domaines d'isolation [WCC+74, MS03].

#### 3.1.4.2 Principe : Garantie de limitation de la portée des extensions

Il faut fournir l'assurance que l'ajout d'une extension a une portée limitée : i.e. où on puisse prédire quelles sont les tâches affectées par l'extension, et lesquelles ne le sont pas. En particulier, il doit être possible d'assurer que l'ajout d'une extension ne modifie pas le comportement des tâches pré-existantes à cet ajout.

Ce principe est important, car il permet de fournir des garanties sur l'exécution des programmes de manière compositionelle : lorsqu'une extension est ainsi ajoutée, la re-certification ou re-qualification ne concerne que les tâches qui sont affectées par cette extension. Les autres tâches sont garanties de conserver le même comportement.

Des approches qui suivent ce principe sont l'extensibilité par les libOS dans les exonoyaux (portée limitée à l'application), par la modification de serveurs dans les micronoyaux ou de domaines de protections dans MILS (portée limitée aux tâches qui en dépendent [HPHS04, BDRS08]), ou la modification d'un noeud dans les structures hiérarchiques (§2.4.3.2) (portée limitée aux descendants de ce noeud). Notons cependant qu'à part pour MILS, ces extensions peuvent induire des modifications dans l'allocation des ressources (e.g. présence de sémaphores possibles), et donc modifient le comportement de l'existant.

Des contres exemples sont l'ajout de modules au noyau dans Linux ou Windows[1]. Il est potentiellement possible de construire des extensions dont la portée est limitée avec des approches qui chargent du code dans le noyau de manière contrôlée (e.g. SPIN [BSP+95]) ; mais il est certainement très difficile d'arriver à l'assurer.

#### 3.1.4.3 Principe : pas de contraintes de temps imposées par le système aux applications

Certains systèmes imposent de faire certaines actions dans un temps limité. Certains systèmes peuvent demander à des tâches de faire des actions, et doivent limiter le temps pour faire cette action pour qu'il n'y ait pas accaparation de temps CPU. Par exemple, L4 permet de spécifier des timeouts pour ses RPC [Sha03] ; Genode/Bastei demande aux services de libérer les ressources en temps limité [FH06, p. 8] ; certains systèmes ont une notification de préemption imminente, ou accordent un sursis d'exécution limité (§ 4.1.2.1).

Le problème est qu'il est très difficile de prévoir si un traitement peut être fait en un temps donné. Si il faut le faire pour les tâches temps réel critiques, il est déjà assez difficile de prendre en compte les contraintes de temps imposées par les spécifications ; il vaut mieux éviter d'imposer ainsi des contraintes supplémentaires.

De plus, ces temps dépendent de différents facteurs dont la vitesse d'exécution du processeur ; ainsi le code n'est pas portable entre processeurs de vitesses différentes.

Ce type de contraintes est donc à proscrire ; il faut trouver un moyen d'organiser la conception pour ne pas en avoir besoin.

[1] y compris signés numériquement, cette signature n'apportant rien à l'assurance de sécurité

### Conclusion sur ces principes

Nous avons proposé un certain nombre de principes à respecter pour un OS qui puisse exécuter de manière performante et sécurisée des applications de criticité hétérogènes. En particulier, les trois premiers groupes de principes vont :

1. Fournir l'assurance que les tâches critiques disposeront d'assez de ressources pour s'exécuter, par l'allocation déterministe et la simplicité dans l'allocation ;
2. Fournir l'assurance d'une exécution correcte aux applications qui disposent d'assez de ressources ;
3. Pouvoir agir en cas d'erreur, en implémentant une procédure de recouvrement sécurisée.

Mis ensemble, ces principes permettent d'assurer pour une tâche une exécution correcte à priori (une fois les ressources nécessaires pour la tâche connues), et une surveillance du bon fonctionnement de la tâche. Ce sont ces principes qui permettent au système de supporter des tâches critiques.

Les principes de flexibilité et d'extensibilité sécurisée facilitent la réutilisation et l'adaptation du système aux besoins. Ce sont eux qui permettent l'exécution de tâches critiques et non critiques arbitraires. L'indépendance des politiques d'allocation permet également d'apporter performance et flexibilité, et aurait donc pu être rattaché à ce groupe de principes.

Une fois ces principes énoncés, il reste à trouver une conception qui permette de tous les combiner. Même si les principes réduisent les choix possibles, les conceptions qui les suivent peuvent être variées, comme le donne l'exemple des systèmes pour la sécurité. Dans le reste de ce chapitre (et de cette thèse), nous présentons une conception qui suit simultanément tous ces principes.

Notons que les deux rôles du système d'exploitation étant la sécurité et l'allocation des ressources, un système étant indépendant des politiques pour ces deux rôles est extrêmement flexible. La seule contrainte que l'OS impose aux applications est l'interface à utiliser pour utiliser les mécanismes (i.e. la manière de spécifier les droits d'accès à une ressource).

## 3.2 Structure

### 3.2.1 Structure générale

**Présentation** Le système est structuré à la fois comme un exonoyau et comme un micronoyau (Figure 3.1 page 86). Lorsqu'une tâche seule a besoin d'une extension, elle devrait pouvoir le faire en modifiant simplement une librairie "libOS" ; on ne crée un nouveau service partagé que lorsque plusieurs tâches ont besoin d'un service commun, et ne se font pas mutuellement confiance. Seules les tâches qui en ont besoin accèdent à ce service partagé. De plus, ce service partagé est de la taille minimale nécessaire pour remplir le service commun. En particulier il faudra rajouter du code à un client utilisant un service commun pour qu'il puisse l'utiliser. Ce code pourra être fourni par une librairie spécifique à l'utilisation de ce service commun.

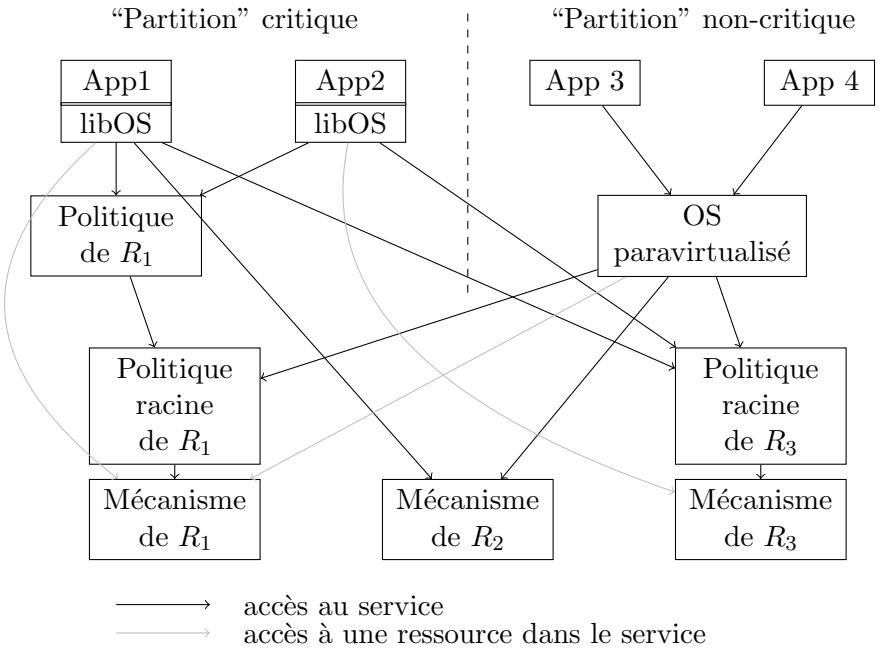



FIG. 3.1 – Exemple de structure : le système permet de créer des hiérarchies pour l'allocation, mais n'en impose pas. Ici $R_1$, $R_2$ et $R_3$ sont trois types de ressources. La politique de $R_2$ n'a pas besoin d'être explicitée (e.g. accès à un terminal). L'allocation de $R_1$ se fait sur deux niveaux, l'OS paravirtualisé pouvant jouer le rôle de module de politique (e.g. accès au temps CPU par priorité dans la partition critique, en round-robin dans l'environnement paravirtualisé). Pour $R_3$ (e.g. de la mémoire), l'OS paravirtualisé détermine encore la politique d'allocation pour ses applications, tandis que les applications critiques peuvent directement accéder à la politique racine. Pour $R_3$ et $R_1$, l'allocation de ressources passe par une politique, mais une fois celle-ci allouées, les programmes peuvent accéder au mécanisme directement, comme App1 et App2. Les applications App3 et App4 ont encore besoin de faire des appels systèmes à leur OS pour accéder aux ressources, comme dans un VMM classique.

Un système d'exploitation remplit trois rôles :

1. *abstraction* : ce qui facilite la programmation d'applications ;
2. *partage des ressources* : ce qui concerne la répartition des ressources entre les tâches ;
3. *sécurité* : ce qui permet à l'OS d'assurer toutes sortes de propriétés “négatives” [Rus89], i.e. qui protège et garantit l'absence de certains comportements non désirables.

Dans notre système, l'abstraction est le rôle des librairies (i.e. la libOS), tandis que les deux derniers rôles sont assurés par les services partagés (et le noyau).

**Intérêts** Le principal intérêt de cette structure est la sécurité. En effet, elle suit scrupuleusement le principe de minimisation des mécanismes communs (§ 2.2.2.6) : les

tâches n'utilisent un mécanisme commun que lorsque c'est nécessaire, et ce mécanisme commun est de taille minimale. Nous réalisons la synthèse de la minimisation des mécanismes communs des micronoyaux et de celle des exonoyaux.

Cette structure permet également de garantir la limitation de la portée des extensions (§ 2.4.1.2). La modification d'une libOS a une portée garantie d'être limitée à une application, tandis que la modification d'un service partagé à une portée limitée aux tâches qui en dépendent. La limitation de la portée est ainsi garantie de manière simple, par la structure même du système. L'ajout d'extensions est simple, et le système facilement extensible.

Un autre intérêt est que cette structure minimise le besoin pour les différents services systèmes de communiquer entre eux. Dans un OS monolithique ou micronoyau traditionel, il y a beaucoup de communication interservice (e.g. le service terminal communique avec le service écran et le service clavier). En minimisant les services et en liant les applications à une libOS, la majorité de ces communications peut être faite par l'intermédiaire de l'application (e.g. Anaxagoros peut ne pas avoir de service partagé de terminal). Cette minimisation des communications inter-services est intéressante car c'est ce qui rend l'écriture de systèmes à base de micronoyau difficile (les services sont fortement couplés), et contribue à l'imprévisibilité dans l'exécution du système (e.g. lorsqu'il y a des notifications en cascade).

Enfin, on peut profiter de tous les avantages qu'offre une structuration en exonoyau en termes de performances [KEG+97, Ros95]. L'interface de bas niveau d'abstraction, avec le nommage physique des ressources, peut aider à implémenter un OS (para)virtualisé en tant que service partagé, pour exécuter le code pré-existant. Elle va également faciliter l'implémentation du transfert explicite des ressources (§ 2.3.2.4), plus sécurisé.

**Comparaison** La différence avec les exonoyaux sont la séparation des services de base en différents domaines de protection ; celle avec les micronoyaux sont que les services partagés sont minimaux et que l'abstraction est faite par une librairie.

Cette structuration ressemble à celle conseillée par Rushby pour atteindre les plus hauts niveaux d'assurance de sécurité pour les embedded systems [Rus84], mais ce système est statique. Elle ressemble également à celle qu'il conseille pour la réalisation d'un système MILS [BDRS08] ; mais dans cette dernière les applications sont toutes exécutées dans des partitions virtualisées, et il n'y a donc pas d'assurance de protection entre applications de la même partition.

Enfin, la différence principale vient de la structuration de la gestion des ressources, présentée dans la prochaine section.

### 3.2.2 Services pour le partage des ressources

#### 3.2.2.1 Services de politique et de mécanisme

On a vu que l'OS assurait trois rôles : abstraction, partage des ressources, et sécurité. Nous avons vu que l'abstraction pouvait être placée dans des librairies, ce qui laisse aux services partagés la sécurité et le partage des ressources.

**Séparation politique-mécanisme** Nous implémentons le principe de séparation du mécanisme et de la politique (§ 2.4.3.1) en plaçant pour chaque ressource l'un dans un *service de mécanisme*, l'autre dans un ou plusieurs *services de politique* (d'allocation). Un service de mécanisme assure toutes les propriétés de sécurité « négatives » sur un type de ressource (par exemple, garantit qu'une application ne peut pas utiliser les ressources qui ne lui sont pas allouées, ou utilise les ressources correctement), et fournit de quoi permettre aux services de politiques de définir l'allocation des ressources. Il y a donc communication entre ces services.

**Sécurité des services de partage des ressources** Les prochains chapitres de la thèse traitent de la réalisation des services et principalement des services de mécanismes. En effet ceux-ci sont accédés par des applications qui ne se font pas confiances, à la fois critiques et non critiques : *la sécurité dans la réalisation des services de mécanisme est essentielle*, car ce sont les seuls points d'attaques du système par une tâche malicieuse.

La sécurité des services de politique peut être obtenue par des propriétés de l'implémentation, mais également de manière structurelle. En particulier, l'allocation hiérarchique (§ 3.2.4.1) permet d'isoler les politiques d'allocations. De plus, l'allocation statique n'a pas besoin de module « actif » gérant une politique, et donc n'a pas besoin de service de politique. Nous nous sommes donc davantage attachés à la réalisation des services de mécanismes qu'aux services de politiques.

Il y a deux types d'attaques contre lesquelles un service partagé doit se protéger. Le premier type concerne la confidentialité : en temps normal des tâches indépendantes ne doivent pas communiquer par l'usage d'un service partagé (notons que cela rend l'utilisation de ces services indépendante de la politique de sécurité (principe 3.1.4.1)). Une brèche dans la confidentialité indique que les tâches peuvent communiquer par l'intermédiaire d'un service, et donc qu'elles peuvent interférer l'une sur l'autre ; cela peut indiquer qu'une tâche peut provoquer la défaillance d'une autre tâche [Rus98].

Le deuxième type concerne la disponibilité : i.e. il y a un déni de service (ou un ralentissement de service) sur le service partagé, ce qui empêche une tâche d'utiliser le service.

La prévention de ces attaques est assurée en partie par notre mécanisme de communication et le prêt de ressource (§ 3.4), et en partie par des principes de structuration des services (section 3.5 et chapitre 5).

#### 3.2.2.2 Gestion de l'allocation des ressources

**Tableau des ressources** Une majorité de types de ressources peuvent se partager spatialement (§ 2.1.1) : i.e. la ressource est décomposable de manière à ce qu'on puisse en donner une partie à une tâche de manière « permanente » (jusqu'à la prochaine décision de modification de l'allocation). Les seules ressources qui ne se partagent pas spatialement sont le CPU ou tout autre unité capable d'un traitement parallèle au CPU (e.g. DMA, réseau, disque dur ; mais pas ligne série puisque celle-ci ne peut pas fonctionner sans être directement pilotée par le CPU).

En général ces ressources spatiales peuvent être énumérées de 0 à $N-1$, $N$ étant leur nombre. Les éléments peuvent être indifférenciés (e.g. pages mémoires) ou non

(e.g. port TCP/IP). Les applications utilisent les ressources en faisant un appel au service de mécanisme de cette ressource. Le numéro de la ressource est transmis en paramètre de l'appel, et le service de mécanisme exécute la requête sur la ressource au numéro correspondant.

Le service de mécanisme doit gérer des données relatives à chaque ressource. On utiliser pour cela une structure appelée *tableau des ressources*, qui est un simple tableau indexé par le numéro de ressource. Il pourrait se présenter sous une forme différente que celle d'un tableau ; mais cette forme est la plus simple et efficace. Ce choix est justifié section 3.5.2.1.

tableau des ressources

**Protocole d'accès aux ressources** L'allocation et l'utilisation des ressources se déroule de la manière suivante. L'allocation d'une ressource à une tâche se fait par le service de politique, qui crée une association tâche/numéro de ressource[2] et la communique au service de mécanisme. Lorsque la tâche veut utiliser la ressource, elle fait un appel au service de mécanisme pour cette ressource, en lui désignant le numéro de la ressource. Le service vérifie que la tâche a bien le droit d'utiliser cette ressource en regardant si l'association tâche/numéro existe, avant d'exécuter la requête pour cette ressource. La Figure 3.2 fournit un exemple.

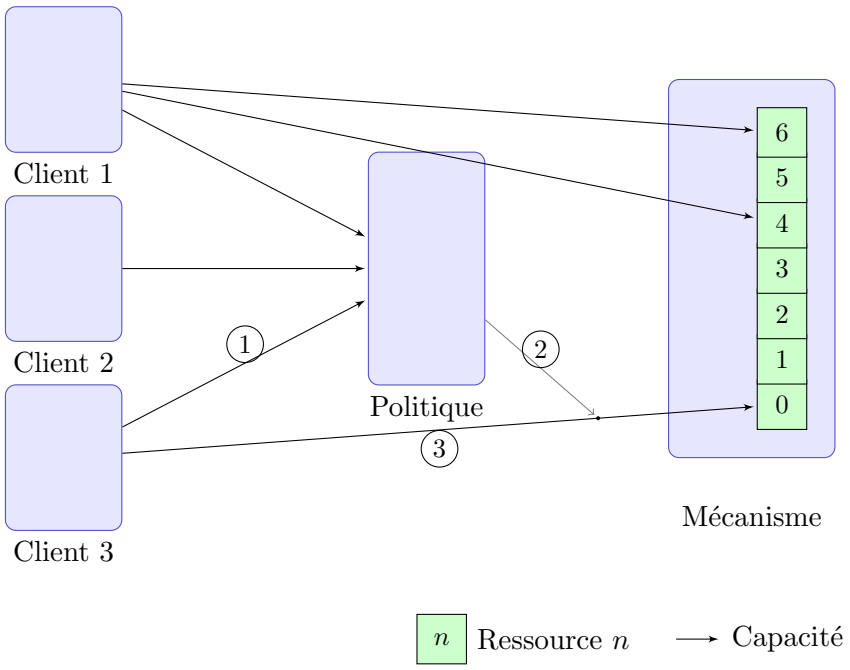



FIG. 3.2 – Séparation politique/mécanisme pour les ressources spatiales. Le protocole est le suivant : (1) la tâche demande à la politique une ressource ; (2) la politique crée une capacité vers une ressource dans le service de mécanisme et la renvoie à la tâche (nota bene : cette flèche ne représente pas une capacité), après avoir révoqué les éventuels autres accès à la ressource. (3) la tâche communique directement avec le service de mécanisme par cette capacité.

[2] appelée secure binding dans les exokernels [EKJO95, §3.2]

**Implémentation avec les capacités** Pour suivre le principe d'économie de mécanisme (§ 2.2.2.5) et d'unicité du mécanisme de protection, et le principe de médiation complète (§ 2.2.2.2), *il est important que l'installation et la vérification de l'association tâche/ressource soit faite en réutilisant le mécanisme de contrôle d'accès*. Notre mécanisme de capacité (présenté en détail § 3.3) est particulièrement adapté pour implémenter efficacement ce protocole.

Une capacité correspond à une ressource[3]. Le noyau fournit une opération `invoke` qui prend en argument une capacité, qui permet de faire un appel vers le service de mécanisme de la ressource. Le noyau communique alors au service le numéro de la ressource correspondant à la capacité. *On ne peut appeler le service de mécanisme avec le numéro de ressource n que si on a une capacité vers cette ressource de numéro n*. Ainsi, le mécanisme de vérification de droit d'accès à une ressource est uniforme, sécurisé et simple d'utilisation à la fois pour le service et pour la tâche cliente. La désignation d'une ressource se fait tout simplement en sélectionnant la capacité correspondante, ce qui permet de faire l'économie d'un mécanisme de désignation.

Ce mécanisme de sécurité est flexible ; par exemple parfois une tâche peut utiliser un grand nombre de ressources d'un même type (e.g. page mémoire, ou secteurs de disque). Dans ce cas, une capacité peut désigner un ensemble de ressources, plutôt qu'une seule. Le service mémoire fonctionne de cette manière.

La création et révocation de capacités vers un domaine $d$ peut se faire depuis un domaine $d'$ différent de $d$. Ainsi, lorsqu'une tâche contacte un service de politique $d'$ pour faire une demande d'allocation de ressources, ce dernier peut créer une capacité vers le service de mécanisme $d$ sans qu'il fasse lui-même un appel de service à $d$. Cela contribue à l'efficacité de l'allocation.

**Avantages et notes** Ce mécanisme a surtout un avantage en terme de simplicité et de sécurité. En utilisant le mécanisme de contrôle d'accès pour accéder au ressource, on dispose d'un moyen formel pour savoir qui peut accéder à quelles ressources. Il est ainsi simple de s'assurer qu'il est impossible pour une tâche d'accéder à des ressources auquel il n'a pas le droit.

Le deuxième avantage est l'efficacité. L'allocation se fait sans contacter le service de mécanisme, et l'utilisation se fait sans contacter le service de politique. Une fois l'allocation faite, l'accès aux ressources est très rapide, la vérification se faisant par le mécanisme de capacité optimisé dans ce but. La séparation des services de politique et de mécanisme permet de spécialiser et optimiser chaque service pour son opération.

Enfin, l'allocation des ressources est également rapide : il n'y a a priori qu'un seul appel de service à réaliser.

#### 3.2.2.3 Gestion de la répartition temporelle des ressources

On différencie deux notions dans la répartition temporelle des ressources.

Lorsqu'on parle simplement d'un changement de la possession des ressources (e.g. changement d'allocation mémoire), cette répartition est librement implémentable

[3] en général ; parfois à un groupe de ressources.

par le service de politique pour cette ressource. Il lui suffit de changer les capacités des tâches vers les ressources.

Par contre, les ordonnancements d'accès à des périphériques (CPU, réseau, disque), ne peuvent se faire ainsi. Pour le temps CPU, nous avons un prototype préliminaire de protocole d'ordonnancement hiérarchique similaire à celui de Ford et al [FS96], décrite plus en détail en section 3.2.4.2 ; la version non-hiérarchique consiste seulement à ce que le service d'ordonnancement CPU puisse programmer des timers pour être réveillé à certains instants.

Nous n'avons pas encore d'ordonnancement disque et l'ordonnancement réseau est encore très préliminaire ; nous n'avons donc pas encore étudié comment ces politiques d'ordonnancement pouvaient être séparées du mécanisme.

### 3.2.3 Protection mémoire, temporelle, et des ressources

#### 3.2.3.1 Unités de protection et d'allocation

**Domaine, threads et espace d'adressage** Nous découplons la protection et l'allocation des ressources en découplant les objets systèmes (le thread, le domaine, et l'espace d'adressage) en entités indépendantes (Figure 3.3).

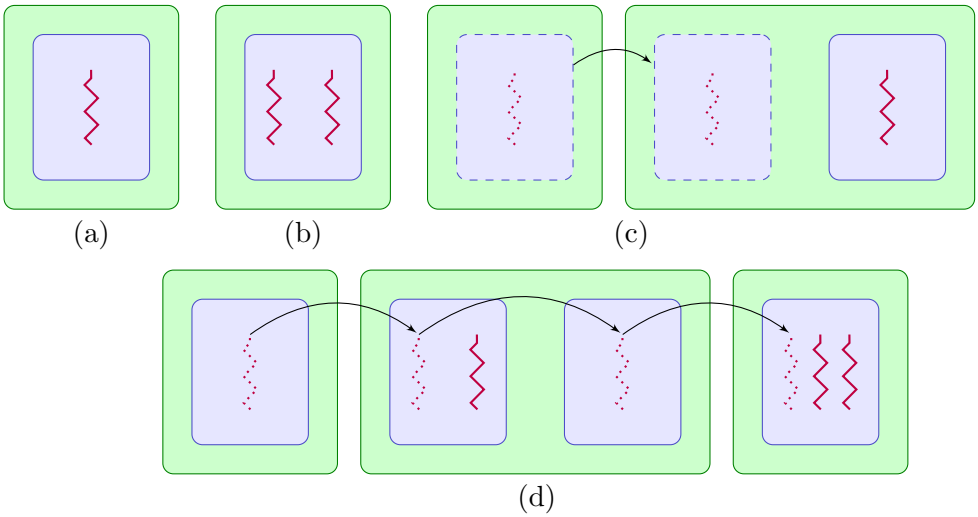



FIG. 3.3 – Découplage des threads, domaines, et espace d'adressages. Différents cas de figures sont représentés (a) Processus UNIX. (b) Processus UNIX multithread. (c) Domaine changeant d'espace d'adressage ; un espace d'adressage devient inutilisé, l'autre devient partagé. (d) Thread se déplaçant entre domaines (similairement aux chemins de Escort [MP96]).

Le *domaine* (de protection) est l'unité principale de protection du système, et un *principal*, i.e. une entité détentrice de droit, ou encore le sujet du contrôle d'accès. Comme nous implémentons un système à capacité, les capacités sont donc attribuées à un domaine. Comme l'accès aux ressources spatiales se fait par des capacités, le domaine est également le principal pour le droit d'utilisation de ces ressources. domaine principal

Le *thread* (noyau) est principalement le réceptacle du temps CPU (i.e. l'unité thread

d'allocation et de protection temporelle). C'est également un principal, mais est utilisé ainsi seulement pour la transmission de droits entre domaines ; cet aspect est développé en section 3.4.3. Threads et domaines sont découplés : un thread peut changer de domaine, et un domaine peut exécuter plusieurs threads. Les seules contraintes sont qu'un thread ne peut être au plus que dans un domaine, il qu'il soit dans un domaine quand il est exécuté.

espace d'adressage

Le domaine est également découplé de la notion d'espace d'adressage. L'*espace d'adressage* est l'entité qui contrôle la translation adresse physique/adresse virtuelle et les droits d'accès direct à la mémoire : c'est donc le principal pour les droits d'accès mémoire (et l'unité de protection mémoire). Ce découplage est surtout assuré pour des raisons pratiques : l'implémentation de l'espace d'adressage dépend en effet fortement du hardware. Un domaine est toujours dans un espace d'adressage, mais l'espace d'adressage auquel un domaine est rattaché peut changer. De plus, il peut y avoir plusieurs domaines pour le même espace d'adressage.

Le découplage est réalisé grâce à l'utilisation de pointeurs universels (§ 3.3.3.1) entre les différents objets. Ce découplage est difficile à réaliser dans beaucoup de systèmes, mais ne pas l'avoir pose des difficultés (voir par exemple Cache [CD94, § 4.2]).

**Intérêts du découplage** Cette organisation permet un maximum de flexibilité. Il est ainsi possible d'émuler les processus UNIX classiques (en fixant l'espace d'adressage, le domaine, et le ou les threads), les chemins de Scout (en permettant aux threads de changer de domaine et d'espace d'adressage), ou de faire de la paravirtualisation (les appels systèmes peuvent être émulés par un changement de domaine et d'espace d'adressage).

On peut également comptabiliser les ressources à une échelle « inférieure au processus » en créant de nouveaux domaines qui partagent le même espace d'adressage. Il reste cependant à voir comment attribuer les ressources entre ces différents domaines, et comment comptabiliser les ressources à une échelle supérieure à celle du processus, pour avoir résolu le problème de l'inadéquation entre unités de protection et d'allocation (§ 2.3.1). Ce sera l'objet de la prochaine section.

Le découplage entre capacité et espace d'adressage est particulièrement intéressant. Il permet au système d'être indépendant du mécanisme de protection mémoire utilisé (et donc plus facilement portable), et de pouvoir manipuler ce mécanisme de protection mémoire sans imposer d'abstraction. Beaucoup de systèmes ont combiné capacité et adressage mémoire ; selon Shapiro c'est souvent source de problème de performance [SSF99, § 1.4.1]. EROS [SSF99] et seL4 [EDE08] ne sont pas lent, mais imposent des abstractions sur l'espace d'adressage (utilisation de cache pour l'un, organisation hiérarchique par puissance de 2 pour l'autre).

#### 3.2.3.2 Implémentation

Nous avons su réaliser une implémentation très simple de ces mécanismes sur l'architecture x86. La protection mémoire sur cette architecture est basée sur la pagination, avec des pages de 4ko. Le format des tables des pages est défini par le hardware, qui se charge d'aller récupérer les entrées dans le TLB. Les tables de pages

sont définies sur deux niveaux. Nous appelons *table des pages de premier niveau* la table racine, et tables des pages de deuxième (ou dernier) niveau les autres. Les tables des pages de premier et deuxième niveau font également 4ko.

table des pages de premier niveau

**Taille unique d'objets systèmes** Tout comme les pages de données et les tables des pages des différents niveaux, nous avons défini que *tous les objets systèmes étaient de taille une page*. Ce choix de conception a permis de simplifier drastiquement l'implémentation du système de mémoire virtuelle ; il suit également notre principe de ressources nécessaires prévisibles (principe 3.1.2.2).

Ce que nous appelons de manière générale espace d'adressage est sur x86 une table des pages de premier niveau (donc de taille une page).

Un domaine est représenté dans le noyau par une structure (appelée KDCB, kernel domain control block), de taille une page. Cette structure contient l'adresse physique *estampillée*[4] de l'espace d'adressage courant, ainsi que le program counter d'entrée lors d'un appel de service ; tout le reste est utilisé comme *C-list*, i.e. contient les capacités du domaine.

C-list

Le thread est représenté par deux structures qui font chacune une page : une est appelée *KTCB* (kernel thread control block), n'est accessible que par le noyau. Elle contient des données pour l'ordonnancement, quelques données relatives à l'appel de service en cours, dont l'adresse physique estampillée du domaine courant, et des capacités (le thread est également une C-list). L'autre est appelée *UTCB* (user thread control block), et est accessible en écriture par le thread courant. Elle est utilisée pour passer des arguments en paramètres lors de l'appel de service (§ 3.3.3.2), comme pile, ou pour contenir les registres sauvegardés lors de la dernière préemption (§ 4.2.3)).

UTCB

**Services du noyau** Le noyau contient les services de mécanisme pour la manipulation de la mémoire, des domaines et des threads. Il fonctionne en attribuant à chaque page physique un des types suivants : KDCB, KTCB, UTCB, page de table de premier niveau, page de table de deuxième niveau, page de données, page blanche.

Il assure certaines propriétés sur les pages (e.g. il est impossible d'écrire directement dans le KDCB, KTCB, ou une table des pages ; il est impossible d'utiliser une page de données en tant que table des pages, etc.), dont le respect de la politique d'allocation. Ces propriétés assurent l'intégrité (et la confidentialité) des tâches du système (étant donné une allocation des ressources appropriée).

L'implémentation de ce mécanisme, que nous avons su rendre extrêmement scalable en multiprocesseur, est décrite en annexe C.

**Autres architectures** Cette implémentation est facilement adaptable sur d'autres architectures à mémoire paginée, surtout si elle basée sur des tables des pages multi-niveaux. Comme on découple la notion d'espace d'adressage de celle de domaine de protection, on peut facilement utiliser n'importe quelle méthode pour décrire les espaces d'adressages ; il suffira de modifier le mécanisme en fonction.

[4] voir § 3.3.3.1

Si la protection mémoire est basée sur la segmentation, le problème est plus difficile, sauf dans le cas ou le système est statique (e.g. MPU pour un système contraint). Dans ce cas, domaine, KTCB etc. sont des données qui font partie du noyau ; comme elles ne sont pas contraintes à être de taille une page, on ne perd plus de place due à la fragmentation interne.

### 3.2.4 Comptabilisation des ressources

Nous avons vu que les services de politiques étaient responsables de l'allocation et comptabilisation des ressources ; dans cette section nous voyons en détail comment ces services implémentent cette comptabilisation.

#### 3.2.4.1 Partitionnement multi-hiérarchique

Cette section traite de la division des ressources par les politiques. Notons que les règles que nous édictons ici peuvent ne pas être suivies (e.g. si on utilise le système pour paravirtualiser un OS), mais sont importantes pour obtenir une allocation des ressources sûre et sécurisée.

**Notion de partitionnement** Nous définissons le *partitionnement* pour formaliser la notion de division et de partage d'une ressource (que la division soit faite temporellement ou spatialement). Le partitionnement doit être compris au sens mathématique du terme, et non au sens « avionique » (e.g. [Rus98]).

Mathématiquement, un partitionnement d'un ensemble de ressources divise les ressources en sous-ensembles, appelés *partitions*, tels que

- l'ensemble des ressources est divisé (i.e. il n'y a pas de ressource non prise en compte dans la division) ;
- il n'y a pas d'intersection entre les partitions (i.e. une ressource ne peut pas appartenir à deux partitions).

Le partitionnement permet ainsi de diviser les ressources de sorte que *chaque ressource appartient à une, et une seule, partition.*

**Application à la comptabilisation des ressources** Le partitionnement est important pour deux raisons. Tout d'abord, comme toutes les ressources appartiennent à une partition, il n'y a pas de ressource non comptabilisée. Cela permet ainsi de respecter le principe d'identification de toutes les ressources (§ 3.1.1.1). Ensuite, chaque ressource n'appartient qu'à une seule partition, et ne peut pas être décomptée plusieurs fois. Le partitionnement crée ainsi une notion de *propriété* sur les ressources, au sens où chaque partition est l'unique propriétaire de la ressource. Cela facilite la comptabilisation et l'attribution des ressources.

Notons que la notion de propriété est découplée de celle d'utilisation : il est possible d'utiliser une ressource sans en être le propriétaire. Il est par exemple possible qu'une partition soit utilisée qui contienne le code des librairies partagées du système. Les domaines peuvent utiliser ce code dans leur espace d'adressage, sans

qu'il leur soit décompté. Également, la notion de propriétaire est découplée de celle de principal, ce qui évite un certain nombre de problèmes souvent présents dans les systèmes à capacités [Lev84, p. 198].

L'annexe C.3.1.1 présente à titre d'exemple comment nous avons mis en place la comptabilisation de la mémoire.

**Partitionnement hiérarchique** On peut étendre la notion de partitionnement pour prendre en compte la structuration hiérarchique, qui permet de disposer de plusieurs politiques simultanément et de manière sécurisée (§ 2.4.3.2). Il suffit pour cela de permettre à chaque partition d'être elle-même partitionnée, et ainsi de suite récursivement. Ce partitionnement hiérarchique peut être mis sous la forme d'un arbre, où la racine représenterait la totalité de la ressource, et où les enfants représentent les partitions d'un ensemble. La Figure 3.4 fournit un exemple.

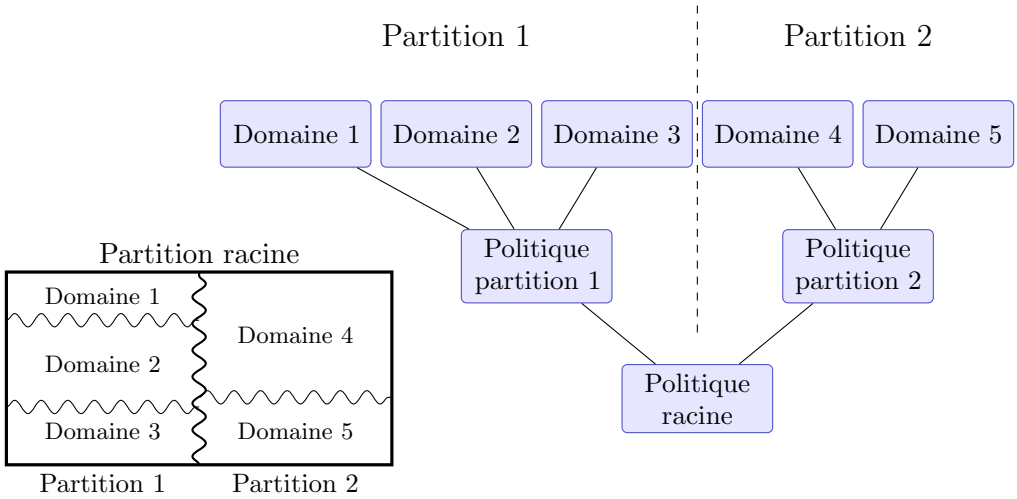



FIG. 3.4 – Partitionnement hiérarchique : représentation des partitions hiérarchiques (à gauche) et des domaines responsables de leur gestion respective (à droite).

Une ressource appartient désormais à une et une seule partition *à chaque niveau*. Il n'y a donc pas un seul propriétaire de la ressource : toute une branche de l'arbre est propriétaire de la ressource.

**Comparaison à l'état de l'art** Certains systèmes à capacités, comme Hydra, rejettent la notion de propriété d'une ressource [WCC+74, p. 340]. D'autres, comme System/38, font du droit de propriété un droit spécial [Lev84, p. 147]. Nous estimons que la propriété est une notion importante pour pouvoir comptabiliser les ressources proprement ; mais la notion de propriété doit être découplée de la notion de droit d'utilisation, et la notion de propriétaire découplée de celle de principal. Cela permet une grande liberté dans la manière d'utiliser les ressources tout en permettant de maintenir une comptabilité précise et d'utilise le mécanisme de capacité de manière unique.

Une autre différence par rapport à l'état de l'art est la notion de partitionnement, qui définit strictement la possession de ressources. Ceci est en contraste avec d'autres travaux qui implémentent une allocation hiérarchique des ressources, mais où les nœuds de la hiérarchie sont plutôt vus comme des « gestionnaires hiérarchiques » des ressources (comme [FS96]). Nous estimons que le partitionnement est nécessaire pour pouvoir comptabiliser les ressources de manière exacte.

Notons cependant que d'autres travaux ont implémenté une disposition ou le parent « contenait » les enfants, comme Fluke [FHL+96] ou les travaux de Brich Hansen [Han70], ou encore les space banks d'EROS [SSF99].

#### 3.2.4.2 Implémentation

L'idée générale est que les partitions à chaque niveau soit sous la responsabilité d'une entité (thread pour le temps CPU, domaine pour les autres ressources). Si la partition est une feuille dans l'arbre, l'entité est une simple utilisatrice des ressources. Si c'est un noeud intérieur, elle remplit le rôle de politique d'allocation, et a la possibilité de modifier le partitionnement en fonction des besoins des entités filles. Les noeuds parents ont prééminence sur leurs enfants, i.e. décident de leur allouer ou révoquer une ressource.

Pour respecter le principe d'indépendance des politiques d'allocation de ressources de types différents (§ 3.1.1.1), nous avons choisi d'implémenter un système multi-hiérarchique. Les différentes hiérarchies sont complètement indépendantes ; ainsi une entité responsable de la politique racine de l'allocation de ports TCP pourra-t-elle être simple utilisatrice de la mémoire.

**Mémoire et ressources partagées spatialement** Dans le prototype, nous nous sommes limités à une allocation à deux niveaux pour toutes les ressources ; le partitionnement à la racine est statique, tandis que le partitionnement du deuxième niveau peut être dynamique.

La raison est la suivante : lorsque l'allocation est dynamique, les ressources attribuées à chaque partition ne sont pas contigües. Pour des facilités de programmation, il est plus facile de les considérer comme contigües, d'où un « tableau de translation » qui associe à chaque numéro de ressource « physique » un numéro de ressource « virtuel ». Il y a besoin d'un tel tableau pour chaque niveau d'allocation dynamique. L'incorporation de ces différents niveaux de translation est coûteux en temps CPU et en mémoire, et relativement compliqué à implémenter. Surtout, en se limitant à un niveau de translation, le mécanisme n'a pas à s'occuper de ces translations, ce qui le simplifie.

Il est par contre possible de rajouter, sous ce niveau d'allocation dynamique, différents niveaux d'allocation statique, car il n'est pas difficile de rendre ceux-ci contigüs. Ainsi, il n'y a pas besoin de table de translation : une simple addition ou vérification d'intervalle est suffisante. Comme il est absolument inutile d'avoir plusieurs niveaux d'allocation statique, nous les avons regroupé en un seul.

Notons qu'il est possible de rajouter d'autres niveaux d'allocation dynamique : il suffit pour pour cela d'une politique dynamique qui soit adaptée pour le permettre. C'est ce qu'on obtient e.g. si on virtualise un système qui virtualise un autre système.

Les deux niveaux de translation sont suffisants pour beaucoup de situations (en particulier pour émuler le partitionnement avionique (e.g. [Rus98]) ou celui des separation kernels (e.g. [BDRS08])), et évite trop de translations coûteuses.

**Temps CPU** Le temps CPU est différent des ressources allouées spatialement. Tout d'abord, les décisions de changement de l'allocation doivent être prises beaucoup plus souvent. De plus, nous ne voulons pas qu'une tâche puisse faire une demande d'allocation de temps CPU (comme e.g. [FS96]), car on ne peut pas proprement comptabiliser le temps CPU utilisé pour une telle demande. Enfin, comme les threads sont les réceptacles du temps CPU (et donc les feuilles de l'arbre de partitionnement), il est donc naturel que chaque noeud de l'arbre soient des threads, et non des domaines.

On a donc un arbre d'ordonnancement dont chaque noeud est un thread. À la base, il y a l'ordonnanceur racine, qui dispose de tout le temps CPU. Chaque thread peut se comporter comme un ordonnanceur, et donner du temps à un autre thread. Nous donnons la possibilité de limiter le temps donné de deux manières : soit en spécifiant une échéance, soit en spécifiant un temps maximum d'exécution. Un thread ne rend la main à son ordonnanceur que si l'une de ces conditions est remplie, ou qu'il décide de rendre la main en avance. Ces deux paramètres permettent de composer des hiérarchies d'ordonnancement efficacement. En particulier, lorsqu'un thread $A$ décide d'ordonnancer un thread $B$ qui ordonnance un thread $C$, l'exécution passe directement de $A$ à $C$ : nul besoin d'exécuter $B$. Il n'y a donc pas de réveils en cascade.

La plus grande difficulté dans l'ordonnancement hiérarchique est la gestion du réveil des tâches, i.e de la faire passer de l'état bloqué à l'état prêt. Cela peut se produire parce qu'une interruption matérielle survient, qu'un service prévient une tâche que ses données sont prêtes... Dans ces cas, il y a besoin de prévenir (par une notification) l'ordonnanceur de la tâche. Si l'ordonnanceur n'avait plus rien à faire, il faut également prévenir son ordonnanceur etc. Nous avons choisi de limiter le temps CPU en bornant le nombre de niveaux hiérarchiques, et de décompter ce temps (court) de l'entité qui en fait la demande (interruption ou service). Notons que les notifications n'ont pas encore été implémentées dans notre prototype.

Il y a enfin quelques points à noter :

- l'utilisation exclusive du polling (comme dans OASIS) permet d'éviter le besoin de notification. Mais cela occasionne une perte de temps, temps qui est mieux utilisé à exécuter une tâche non-critique ;

- une question à résoudre est que faire lorsqu'un thread $T$ ordonnance un autre thread, et reçoit une notification disant qu'un troisième thread est prêt. Nous permettons à $T$ de choisir s'il est réveillé par ces notifications ;

- pour des raisons de performance, nous laissons la possibilité au hardware d'agir comme l'ordonnanceur racine (i.e. pour exécuter un petit traitement lorsqu'une interruption est reçue). Nous permettons cependant à l'ordonnanceur racine de spécifier pour chaque interruption, le nombre d'interruptions recevables par

unités de temps ; lorsque ce nombre est dépassé l'interruption est désactivée. Cela permet d'éviter les problèmes du genre interrupt livelock [MR97] ;

- du à des contraintes de temps, l'algorithme d'ordonnancement hiérarchique, implémenté dans un premier prototype, n'a pas été incorporé dans le prototype actuel. Nous disposons dans le présent prototype seulement d'un ordonnanceur à un niveau, qui spécifie au mécanisme seulement les dates de réveils.

### 3.2.5 Conclusion sur la structure

#### 3.2.5.1 Résumé

Dans cette section nous avons vu l'organisation des services, en particulier pour le partage des ressources. Cette organisation suit les principes énoncés précédemment, concernant l'indépendance des politiques d'allocations, la haute sécurité, et la limitation garantie de la portée des extensions.

Nous avons ainsi combiné la structure « micronoyau » et la structure « exonoyau » pour obtenir des services minimaux indépendant et très sécurisés. Nous avons implémenté la séparation de la politique et du mécanisme en mettant chacun dans un service séparé, en les faisant communiquer par l'intermédiaire d'un « tableau des ressources » partagé. Les politiques sont de plus structurées hiérarchiquement, pour que les services pour plusieurs politiques pour une même ressource soient séparés.

Nous avons défini la notion de partitionnement, qui permet de s'assurer que toutes les ressources sont bien prises en compte. Nous avons découplé les différentes entités du système et rendu toutes les politiques indépendantes entre elles, ce qui permet l'obtention d'un système multi-hiérarchique.

Ces points de conception sont réellement originaux. L'implémentation de la séparation politique/mécanisme est souvent incomplète, et ne permet pas de définir arbitrairement sa politique. Les politiques hiérarchiques existent, mais les noeuds sont vus comme des gestionnaires de ressources, et il n'y a pas de notion formelle de comptabilisation des ressources comme nous le faisons avec le partitionnement. Enfin, nous ne sommes au courant d'aucun autre système multi-hiérarchique existant.

#### 3.2.5.2 Séparation en domaines et performances

Notre implémentation décompose l'OS en de nombreux domaines, situés dans des espaces d'adressages séparés ; or le changement de contexte MMU est très coûteux dans les processeurs actuels, surtout sur des processeurs comme le Pentium où cette opération flushe entièrement[5] le TLB. Nous apportons des éléments de réponse à ce problème.

D'abord, le surcoût apporté par ces changements de contexte n'est pas si élevé, comme démontré par les systèmes à micronoyau récents (voir les évaluations de L4 [HHL+97] ou d'EROS [SSF99]). Ce coût est comparable à celui de l'usage de virtualisation [BDF+03], qui est de plus en plus utilisée en partie à cause du problème

[5] même si en réalité, le Pentium offre un bit « global » qui permet d'éviter de flusher les entrées partagées par tous les espaces d'adressages, cette optimisation est surtout utile pour les noyaux monolithiques.

de manque de contrôle sur l'allocation des ressources, à laquelle on répond avec Anaxagoros. Ce surcoût peut de plus être optimisé par des astuces [Lie95b, WTUH03], mais cela complexifie l'implémentation.

Nous croyons, tout comme les concepteurs de Keynix/KeyKOS [BFF+92, p. 13], que la séparation en domaines de protection simples permet de fortement réduire les coûts de gestion, et que cela peut compenser la perte de performance introduite par les changements de contextes additionnels. Ainsi, notre séparation en domaines de politiques et de mécanismes permet à chacun des domaines d'être hautement spécialisé, et plus efficace.

Si les changements de contextes posaient de réels problèmes de performance, il serait facile de rassembler des domaines de protection (alors qu'il est au contraire difficile de séparer du code existant). Cela permettrait de profiter des avantages en performance de notre structuration, sans avoir à payer le coût des changements de contextes. Une fois cela fait, le système ressemblerait à un exonoyau : un noyau petit dans lequel les coûts de gestion sont éliminés. Notons que notre méthode de structuration peut également être utilisée avec des techniques de protection logicielle.

Enfin, il est possible d'utiliser (voire de développer) un hardware adapté pour des changements de contextes MMU rapides.

## 3.3 Contrôle d'accès

Nous avons déjà indiqué que le contrôle d'accès se faisait par des capacités, et que ce contrôle d'accès était utilisé pour accéder aux ressources. Dans cette section, nous justifions ce choix et décrivons notre implémentation.

### 3.3.1 Choix des capacités

#### 3.3.1.1 Matrice de contrôle d'accès

Les principes de séparation des privilège et du moindre privilège (§ 2.2.2) imposent de séparer les différents *privilèges* en différents domaines, qui sont les entités qui détiennent ces privilèges. Entre autres privilèges, on compte le fait d'accéder à certaines ressources, dont la mémoire physique ; différents domaines ont donc (usuellement) des espaces d'adressages distincts. Un exemple classique de domaine de protection est le processus UNIX.

Une fois les ressources partagées entre différents domaines, il est nécessaire d'enregistrer qui peut accéder à quelle ressource. Ou de manière équivalente, de disposer d'un oracle qui sache répondre à la question : est ce que le domaine $D$ peut accéder à la ressource $R$ ? Cet oracle implémente le *contrôle d'accès*.

À un instant donné, cet oracle peut se présenter sous la forme d'un tableau, appelé *matrice de Lampson* ou *matrice de contrôle d'accès* [Lam71]. Dans cette matrice (voir Figure 3.5), chaque ligne représente un domaine de protection et chaque colonne une ressource. Les cases peuvent représenter un booléen indiquant si le domaine a accès à la ressource, mais en pratique un domaine peut avoir plusieurs *droits* distincts sur une ressource : ainsi on pourra n'avoir le droit que d'écrire, ou que de lire, dans un fichier.

Ce tableau permet de représenter les situations les plus variées où plusieurs domaines peuvent avoir accès à une même ressource, et où les domaines ont accès à plusieurs ressources. Mais il pose des problèmes concernant son stockage en mémoire : tout d'abord, le nombre de lignes et de colonnes peut être relativement important, donc le tableau prend une taille énorme alors qu'il est relativement peu dense (dans un système fermé (§ 2.2.2.3), où tout est interdit par défaut, la majorité des entrées contiennent $\emptyset$ (aucun droit)). Ensuite, dans un système dynamique il faut le faire évoluer dynamiquement, et surtout rajouter lignes et colonnes, ce qui demande des structures de données adaptées.

#### 3.3.1.2 Capacités et ACL

En pratique, on est alors contraint d'utiliser l'une ou l'autre des solutions suivantes [Lam71, SS74] :

- soit on associe à chaque domaine la liste des ressources auxquels il a le droit d'accéder ; c'est le modèle par *capacité* [DH66]. Une capacité est un pointeur sur une ressource accompagnée des droits que le domaine a sur cette ressource ; lorsque le domaine veut se servir d'une ressource il indique quelle capacité il utilise. Les capacités ont donc deux rôles : elles servent aussi bien à montrer qu'on a le droit d'accéder à la ressource qu'à *désigner* la ressource [MYS03, Red74] ;

- soit on associe à chaque ressource la liste des domaines qui peuvent y accéder ; c'est le modèle par *liste de contrôle d'accès (en abrégé ACL)*. Le domaine qui gère la ressource (e.g. un service de mécanisme) maintient pour chaque ressource la liste des clients qui peuvent y accéder ; quand un client veut accéder à une ressource, on regarde s'il est présent dans la liste. Le mécanisme d'ACL ne permet donc pas la désignation, et doit être accompagné d'un mécanisme de nommage associé (comme le système de fichier).

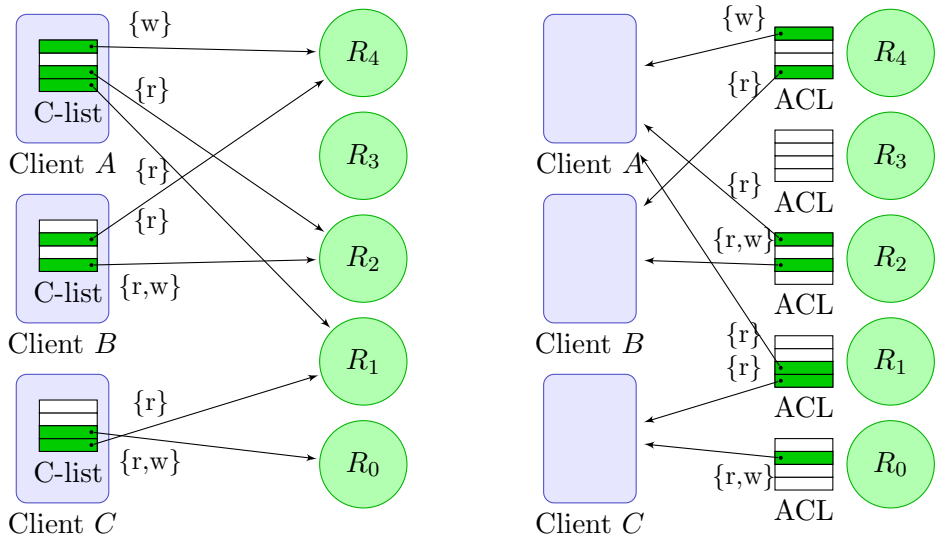
Beaucoup de systèmes utilisent une combinaison des deux : ainsi l'ouverture de fichier UNIX est-t-il contrôlé par des permissions par fichier qui sont un mécanisme de type ACL ; mais une fois le fichier ouvert, on s'y réfère par un descripteur de fichier qui est une sorte de capacité.

#### 3.3.1.3 Avantages des capacités sur les ACLs

Le principe d'unicité du système de protection (§ 2.2.2.5) stipule de n'utiliser qu'un seul mécanisme pour assurer le contrôle d'accès pour l'ensemble du système. Dans ce cas, notre choix se porte assez naturellement sur un mécanisme basé sur les capacités plutôt que sur les ACL pour les raisons suivantes :

- l'implémentation de la vérification du contrôle d'accès par capacité peut se faire en temps constant[6], donc prévisible (principe § 3.1.2.2). Cette vérification

[6] Il existe cependant des implémentations qui ne le sont pas, comme seL4




| | $R_0$ | $R_1$ | $R_2$ | $R_3$ | $R_4$ |
|---|---|---|---|---|---|
| A | {} | {r} | {r} | {} | {w} |
| B | {} | {} | {r,w} | {} | {r} |
| C | {r,w} | {r} | {} | {} | {} |

FIG. 3.5 – Contrôle d'accès par capacité (à gauche) et par ACL (à droite) (Figure inspirée de [MYS03]). Avec les capacités les domaines pointent vers leurs ressources, tandis que pour les ACL c'est l'inverse. En bas est indiquée la matrice de Lampson [Lam71] correspondante.

consiste simplement à regarder que le pointeur entre la capacité et la ressource est valide et dispose des bon droits.

La vérification par ACL nécessite forcément une recherche du client dans la liste, et se fait donc en temps non-constant (souvent linéaire). Pire, le temps dépend du nombre de clients qui ont accès à la ressource dans la liste, un paramètre imprévisible et qu'un attaquant peut potentiellement faire augmenter ;

- comme les capacités permettent aussi de désigner des ressources, on économise un mécanisme de désignation des ressources séparé (e.g. l'arborescence des fichiers UNIX) (principe § 2.2.2.5) ;

- pour les capacités, les données du contrôle d'accès sont associées au client (Figure 3.5). On peut ainsi facilement comptabiliser la mémoire utilisée pour stocker ces données pour les capacités : un domaine ayant accès à beaucoup de ressources aura simplement une grande liste de capacités[7].

  Pour les ACL, les données de contrôle sont associées aux ressources. Ainsi, une ressource partagée devra avoir autant d'entrée dans la liste de contrôle que de

[7] Engler réfute cet argument [Eng95, p. 41] en disant que les politiques doivent traquer la propriété des ressources de toute manière ; c'est faux dans Anaxagoros, car nous distinguons droit d'usage de droit de propriété.

domaines qui peuvent y accéder. Il faudrait alors soit interdire le partage, soit utiliser des techniques de comptabilisation de la mémoire particulière pour les entrées dans les ACL, ce qui est source de bugs et de problèmes de sous-pools potentiels. Nous n'avons pas connaissance de l'existence d'un tel mécanisme ;

- les capacités permettent de transmettre des droits sans notifier de service central, en effectuant juste une copie de ces droits. Cela va permettre d'implémenter le prêt de ressource avec un coût faible. Au contraire, avec les ACL la transmission de droits nécessite de notifier le service qui gère l'ACL de la ressource, pour que ces ACL soient modifiées[8] ;

- les systèmes à capacité ont une granularité de protection plus fine, et par conséquent suivent mieux les principes de sécurité tels que le principe du moindre privilège [MYS03] : il est facile dans un tel système de créer de nouveaux domaines avec moins de droit. Denning [Den76] et Linden [Lin76] ont dans leurs revues de littérature beaucoup insisté sur ce principe, d'où leurs choix de système à capacité. Enfin, les systèmes à capacité permettent d'éviter certains problèmes de sécurité comme le problème du député confus [Har88, MYS03] ;

- les capacités permettent d'implémenter n'importe quel type de politique de sécurité (principe § 3.1.4.1). Les capacités entre différents objets peut en effet être vu comme l'instauration d'une politique « channel control »[9], qui permet d'implémenter toutes sortes de politique de sécurité, dont les politiques à multiniveau (cf KeySAFE [Key89]).

  Mais les capacités sont bien plus que cela, car la modification dynamique des droits permet de créer dynamiquement des schémas de sécurité adapté à tout type de situation [MS03]. De nombreux systèmes à capacités ancien ou récents ont démontré qu'on pouvait construire des systèmes complexes réels avec cette seule approche, et dont les propriétés de sécurité pouvaient être prouvées formellement [Sha99, KEH+09].

  Par contraste, il est difficile de trouver des systèmes ACL pur en pratique : dû à leur rapidité pour le contrôle d'accès, on utilise souvent également les capacités dans les systèmes à base d'ACL (e.g. les descripteurs de fichiers dans UNIX).

Certains modèles de capacité souffrent de problèmes de sécurité ; mais le modèle utilisé en pratique, le modèle des capacité-objet [MYS03], en sont exempts. Dans ce modèle, les capacités ne sont pas de simples données, et transmettre des capacités est

[8] Engler réfute également cet argument [Eng95, p. 41], arguant que cela permet à d'autres domaines de récupérer une capacité et de l'utiliser pour perturber l'exécution du premier. Cela provient probablement de l'implémentation des capacités dans les exokernels ; dans un système à capacité « normal », cet argument ne tient pas puisqu'un domaine contrôle à qui il donne des capacités, et ne les donnerait qu'à des domaines en lequel il a confiance.

[9] Les capacités permettent en fait l'échange d'information dans les deux sens et est donc un canal bidirectionel, tandis que MILS par exemple est unidirectionel [BDRS08]. Cela n'a pas d'importance, puisqu'on peut si nécessaire rajouter un médiateur qui s'assure qu'un canal soit unidirectionel.

une opération distincte de celle de transmettre des données. Cela est nécessaire pour pouvoir assurer la confidentialité pour certaines règles de sécurité comme l'**-Property* de la politique de sécurité multi-niveaux. De plus, les domaines de protection sont considérés comme un type de ressource. Cela aide à contrôler la communication potentielle de capacités, et permet de considérer la délégation d'autorité comme une simple invocation de capacité, donc économise un mécanisme. Cela permet de prouver que des capacités ne peuvent pas s'échapper d'un ensemble de domaines, et donc d'obtenir une preuve formelle de l'isolation des ressources.

#### 3.3.1.4 Résumé

Le modèle de protection par capacité-objets est approprié pour construire des systèmes de haute sécurité ; il permet des vérifications de contrôle d'accès rapide, en temps constant ; il permet de comptabiliser plus facilement la mémoire utilisée pour stocker les données de contrôle d'accès ; et on a des exemples de systèmes complexes pouvant être construits en se reposant sur ce modèle.

### 3.3.2 Opérations à supporter

**Invocation** Nous avons vu que lorsqu'une tâche (le client) avait le droit d'utiliser une ressource, elle devait envoyer une requête au service de mécanisme pour cette ressource, par l'intermédiaire d'une capacité pointant sur cette ressource. L'utilisation de ce service de mécanisme permet de garantir que l'usage de la ressource est fait de manière correcte.

Cette requête est envoyée par l'*invocation* de capacité, qui est une opération permettant de faire un appel de service au « gestionnaire » de la capacité (dans le cas de l'utilisation d'une ressource, le gestionnaire de la capacité est le service de mécanisme pour la ressource). Cette opération doit successivement :

- contacter le service « gestionnaire » ; la capacité est utilisée pour que le noyau puisse trouver le service concerné, et vérifier que le client a bien le droit d'utiliser le service. C'est le rôle « channel control » ;
- exécuter la requête ; la capacité permet de désigner au service la ressource concernée, et de lui démontrer que le client a bien le droit d'utiliser la ressource. C'est le rôle « désignation de ressource ».

Un point fort de notre implémentation est la séparation de ces deux rôles, le premier étant implémenté par le noyau, le deuxième par le « service gestionnaire ». Cela permet une grande flexibilité tout en minimisant le noyau, suivant le principe de minimisation des mécanismes communs (§ 2.2.2.6).

Nous ne permettons l'utilisation d'une capacité que par l'invocation. Cela rend le système *orienté objet* : il faut voir une capacité comme un pointeur sur un objet, l'invocation comme un appel de méthode sur cet objet, et le service appelé comme la classe implémentant l'objet. D'autres conceptions sont possibles : Hydra a notamment implémenté un modèle basé sur des procédures plus génériques ; mais ses concepteurs se sont rendu compte que l'opération d'appel au service qui gère la

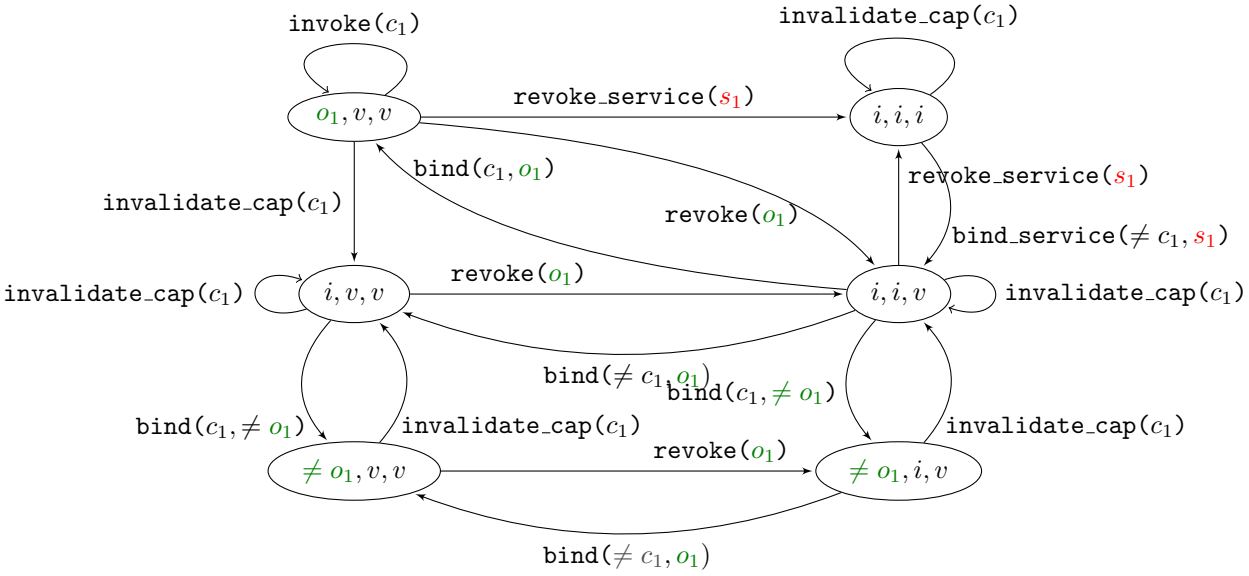



FIG. 3.6 – Automate des transitions possibles du système de capacité, pour une capacité $c_1$, un objet $o_1$, et un service $s_1$. Une capacité peut invoquer un objet, ssi depuis sa création (`bind`) la capacité n'a pas été supprimée (`invalidate_cap`), qu'il n'y a pas eu révocation sur l'objet (opération `revoke`), et que le service implémentant l'objet n'a pas été détruits (opération `revoke_service`).

ressource (appelé `typecall` dans Hydra) était plus simple et suffisante [Lev84, p. 123].

**Révocation** La plus grande difficulté dans l'implémentation des capacités est l'implémentation de la *revocation*. On a vu que le service politique pouvait lier une tâche à une ressource, mais il doit également pouvoir supprimer ce lien. La tâche peut avoir copié la capacité dans beaucoup d'autres domaines : toutes les droits conférés par ces capacités doivent être également révoqués. Il faut de plus que le temps de l'opération « révoquer » soit prévisible (principe 3.1.2.2) : cela signifie révoquer un nombre non-borné de capacités en un temps borné. Cela interdit notamment la traversée de toutes les capacités, comme le font certains systèmes [EDE08].

Il y a d'autres opérations à supporter. Lorsqu'un service est supprimé, il ne faut plus que les capacités puissent accéder aux objets qu'il sert. De plus, lorsqu'une C-list est détruite, toutes les capacités qu'elle contient doivent être invalidées.

Nous avons formalisé toutes ces opérations en annexe A ; nous reproduisons la Figure 3.6 qui représente un automate modélisant la séquence des opérations autorisées. L'annexe A fournit également une preuve de la conformité de notre implémentation à ce modèle.

**Création et copie** Les deux dernières opérations sur les capacités concernent leur création et leur copie. La sécurité dans les systèmes à capacités repose sur le fait

que les capacités ne peuvent être contrefaites. Ainsi, en contrôlant la propagation de capacités (i.e. la création et la copie de capacités), on contrôle l'accès à la ressource à laquelle elle permet d'accéder.

Pour empêcher la contrefaçon, il faut que seul le noyau (ou le hardware) puisse créer de nouvelles clés. Il y a plusieurs moyens d'implémenter cela [Red74]. Nous avons choisi d'implémenter les capacités de manière segmentée : les capacités sont stockées dans des zones mémoires particulières, les C-lists, qui ne sont accessibles qu'au noyau ; les capacités et les données ne sont pas mélangées. Ce partitionnement assure que des domaines ne peuvent pas se créer des capacités pour pouvoir accéder à de nouvelles ressources.

D'autres possibilités d'implémentation des capacités sont : par typage, mais cela requiert que les applications soient implémentées dans un langage sûr ; par tag, mais cela demande du hardware particulier ; ou par mot de passe ou cryptographie, auquel cas les capacités prennent de la place, du temps de traitement, et on n'a pas de contrôle sur la propagation des capacités puisqu'elles peuvent être transmises en tant que données. En particulier, les capacités à mot de passe ne peuvent pas être confinées, ni assurer une politique de sécurité multi-niveaux [MYS03].

Notons que les capacités, une fois créée, sont non mutables : on ne peut que les remplacer par une nouvelle capacité. Cela permet en particulier de se prémunir contre des race condition, dont une faille de sécurité de type TOCTTOU : si les droits d'accès sont vérifiés dans un thread $A$, puis que la capacité est modifiée dans un thread $B$, alors $A$ pourrait utiliser une ressource sans plus en avoir la permission. Le fait que les capacités soient non mutables permet de récupérer les données de la capacité et de vérifier les droits de manière simple dans le service, tout en étant prémuni contre ces bugs.

### 3.3.3 Invocation

#### 3.3.3.1 Pointeurs estampillés et universels

Notre mécanisme de capacité est basé sur ce que nous dénommons *pointeur universel*, qui permet au noyau de référencer de manière unique tous les objets qu'il gère. Le pointeur universel est un cas particulier de ce que nous appelons *pointeur estampillé*.

**Pointeur estampillé** Le problème est le suivant. Dans un système dynamique à mémoire finie, il faut réutiliser la mémoire. Lorsqu'on libère un emplacement mémoire, les pointeurs vers cet emplacement deviennent incorrects et ne doivent plus être utilisés. Pour cela, il faut invalider tous ces pointeurs (i.e. faire en sorte qu'ils ne peuvent plus être utilisés). Le pointeur estampillé permet de faire cela en temps constant. Levy crédite l'invention de cette technique aux concepteurs de CAL-TSS [Lev84, p. 54].

Pour cela, nous associons à toute adresse qui peut contenir un objet la « date » de création de l'objet. Lorsque l'adresse physique ne contient pas d'objet, cette date est placée à $+\infty$. Un *pointeur estampillé* est un couple (adresse, date). Un pointeur estampillé $p$ pointe vers un objet $o$ si l'adresse contenue dans $p$ est celle de $o$, et si la date contenue dans $p$ est celle de l'adresse de $o$. En résumé :

$$p \text{ pointe vers } o \iff p.\texttt{adresse} = o \ \wedge \ p.\texttt{date} = o.\texttt{date}$$

Lorsqu'on crée un pointeur estampillé $p$ sur un objet $o$, on stocke dans $p$ le couple (adresse de $o$, date de création de $o$). Le mécanisme ci-dessus garantit que si $o$ est détruit ou remplacé, la date correspondant à l'adresse de $o$ sera toujours strictement supérieure à la date de création de $o$, donc tous les pointeurs sur $o$ auront la mauvaise date et seront invalides. Le pointeur estampillé permet ainsi d'identifier un objet de manière unique dans le temps.

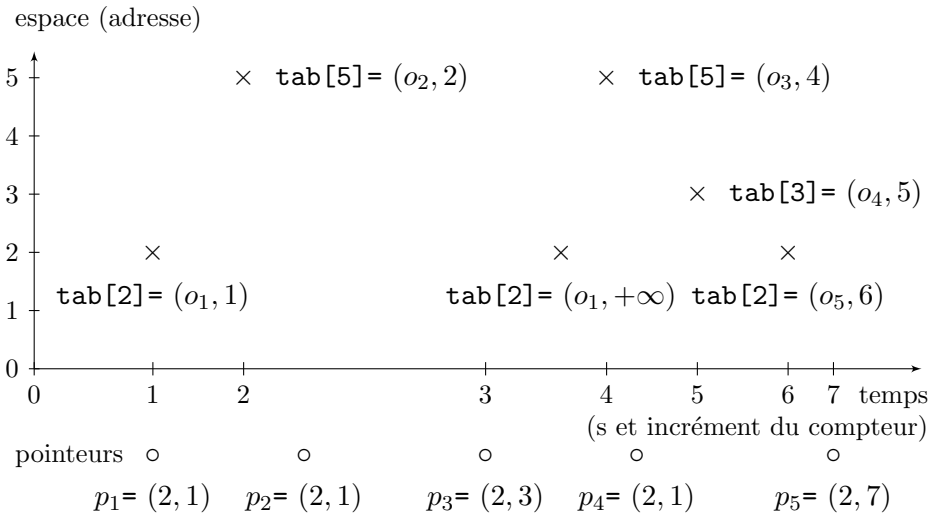



FIG. 3.7 – Création d'objets et de pointeurs estampillés. L'abscisse représente le temps réel, mais les unités sont les instants où le compteur de date doit être incrémenté. Les créations d'objets sont représentées par un $\times$, et placent dans un tableau un objet et une date de création. Les pointeurs contiennent un couple (indice de l'objet dans le tableau, date). $p_1$, $p_2$ et $p_4$ contiennent la date de création de $o_1$ et pointent tous sur $o_1$. $p_3$ et $p_5$ sont créés avec la "date actuelle" : $p_3$ pointe sur $o_1$ car $o_1$ existe encore au moment de la création, alors que $p_5$ pointe sur $o_5$. Lorsque $o_1$ est détruit entre les instants 4 et 5, tous les pointeurs présents et futurs sur $o_1$ sont invalidés (sur la figure, $p_1$, $p_2$, $p_3$ et $p_4$).

Le pointeur estampillé est très intéressant car il permet d'invalider tous les pointeurs qui pointent vers un objet $o$ en temps constant, ce qui suit le principe d'exécution en temps prévisible (§ 3.1.2.2). Cela évite notamment d'avoir à utiliser la technique du « pointeur arrière », i.e. de traverser tous les objets qui pointent vers $o$ pour invalider ces pointeurs, comme cela est fait dans EROS, [Sha99, p. 103], seL4 [EDE08, § 3.2.2], ou plus généralement les inverted page table présentes dans de nombreux systèmes [SSF99, § 4.2.3]. Par contre, l'utilisation du pointeur estampillé rajoute un léger surcoût à chaque déréférencement du pointeur dû à la vérification des dates.

Notons que ce mécanisme nécessite de pouvoir toujours trouver la date associée à une adresse. Un cas dans lequel cela est vrai est lorsque tous les objets pointables

par un pointeur $p$ font la même taille, et peuvent être stockés dans un tableau. Dans ce cas, on associe une date à chaque entrée dans le tableau. Tous nos pointeurs estampillés sont utilisés de cette manière.

Le système est incorrect si, par exemple, un objet $o_1$ est créé, puis un pointeur $p$ est créé qui pointe sur $o_1$, puis $o_2$ est créé à la place de $o_1$, tout cela au même moment. Cela peut se produire si on utilise une horloge système comme date : il suffit que ces opérations s'effectuent entre deux ticks successifs d'horloge. Pour contrer ce problème, nous utilisons un compteur : à chaque fois qu'une nouvelle date est nécessaire (e.g. création d'un nouvel objet), elle est copiée de ce compteur qui est également incrémenté. Nous utilisons des dates sur 64 bits afin qu'il ne puisse pas y avoir d'overflow. Comme il ne s'agit pas réellement de date au sens temporel, nous appelons ces dates « estampilles ». Nous conservons cependant dans cette description le terme date.

**Extension du pointeur estampillé** On peut étendre le mécanisme de pointeur estampillé ainsi. Dorénavant, un pointeur estampillé $p$ pointe vers un objet $o$ si l'adresse physique contenue dans $p$ est celle de $o$, et si la date contenue dans $p$ est *supérieure ou égale* à celle de l'adresse physique de $o$ :

$$p \text{ pointe vers } o \quad \Longleftrightarrow \quad p.\texttt{adresse} = o \ \wedge \ p.\texttt{date} \geq o.\texttt{date}$$

Cette extension permet de stocker dans le pointeur estampillé la « date actuelle » (i.e. la date de création du pointeur), au lieu de la date de création de l'*objet* pointé [10]. Nous utilisons cette extension pour l'implémentation des capacités, qui utilisent une même date pour pointer sur un objet « nouvellement créé » et un sur un objet « ancien ».

Le principe est le même : quand $o$ sera détruit, la date associée à son adresse sera toujours supérieure à la date actuelle, et tous les pointeurs seront toujours invalidés (comme $p_3$ sur la Figure).

Il y a une subtile différence : lorsqu'un pointeur estampillé est créé ainsi, il faut s'assurer que l'objet pointé « est le bon » au moment de la création du pointeur. Si un nouvel objet $o_2$ a remplacé $o$ à la même adresse physique au moment de la création du pointeur, on risque de créer un pointeur valide sur $o_2$, ce qui est problématique (sur l'exemple Figure 3.7, $p_5$ ne pointe ainsi pas sur $o_1$, mais sur $o_5$).

**Pointeur universel** Notre principale utilisation des pointeurs estampillés est pour stocker des liens entre objets systèmes. De tels liens sont fréquents : par exemple le KTCB pointe sur son domaine courant, qui pointe lui-même sur son espace d'adressage courant. Ces objets peuvent ne pas être accessible dans l'espace d'adressage courant ; le seul moyen de les identifier de manière unique (dans l'espace) est d'utiliser leur adresse physique. Un *pointeur universel* est un couple (adresse physique, date) ; c'est un cas particulier de pointeur estampillé. Un pointeur universel identifie un objet de manière unique dans l'espace et le temps. C'est grâce à

[10] Il est même possible de créer un pointeur avec n'importe quelle date entre la date de création de l'objet et la date actuelle, bien que cela n'aie pas grand intérêt.

l'utilisation du pointeur universel pour stocker les relations entre objets que l'on peut implémenter le découplage entre thread, domaine et espace d'adressage (§ 3.2.3.1).

Comme on implémente un système dynamique, et que la mémoire est finie, il faut la réutiliser. Ainsi, la mémoire utilisée pour stocker un domaine peut être réutilisée pour stocker un thread, une table des pages, ou de la mémoire pour l'espace utilisateur. Le pointeur universel permet de vérifier que le pointeur correspond bien à l'objet pointé initialement. Le mécanisme exploite la propriété que tous nos objets systèmes sont de taille une page. Cela permet de stocker la date de l'objet dans un tableau associant à chaque page physique sa date (ce tableau est décrit en annexe C.1.2.2).

**Notes**

- Un prérequis pour l'utilisation du pointeur universel (ou estampillé) est que les objets restent fixes en mémoire. C'est notamment le cas dans notre système car la mémoire utilisée par le noyau n'est pas swappée. Les noyaux monolithiques, dont la mémoire est swappée, doivent en général continuer à utiliser la méthode des « pointeurs arrières ».
- Les pointeurs universels sont à rapprocher de la notion d'identificateur unique de Liedtke [Lie95a, § 2.3], utilisé pour établir une communication. Cependant, les pointeurs universels ne sont pas accessibles aux tâches, et restent dans le noyau ; les exporter aux tâches est en effet une brèche de confidentialité [Sha03].
- Un problème qui se pose est celui de la lecture et écriture des objets pointés par des adresses physiques. Idéalement, notre noyau n'utiliserait que des adresses physiques directement. Mais on ne peut pas accéder directement aux adresses physiques sous Pentium, sauf si le système a peu de mémoire, auquel cas on peut la mapper en entier dans l'espace d'adressage du noyau.

  Pour pallier à ce problème, nous installons des mappings temporaires lorsque cela est nécessaire. Nous arrangeons les données pour minimiser ces mappings, mais cela a néanmoins un impact sur les performances. Ce problème est commun à beaucoup de systèmes d'exploitations sur Pentium (e.g. [ECCZ05, BC05]). Il est dommage que ce processeur ne fournisse pas de mécanisme pour l'accès à la mémoire physique : il semble simple d'installer une extension qui permette de ne pas passer par le mécanisme de MMU, surtout étant donné que la majorité des caches de ces architectures sont adressés physiquement.

#### 3.3.3.2 Invocation de capacités

Nous discutons à présent de notre mécanisme de capacité proprement dit, qui s'appuie sur les mécanismes de pointeurs estampillés vus précédemment. Notre mécanisme de capacité est à la fois simple, efficace en mémoire et en temps CPU, flexible, et a des propriétés intéressantes résumées en fin de section.

**Format** Notre capacité est la combinaison d'un pointeur universel sur un service, d'un pointeur estampillé sur un objet, et de droits. Chaque capacité contient 4

champs :

- `destination`, un pointeur vers un domaine (22 bits)
- `rights`, les droits de la capacité (10 bits)
- `object`, un pointeur vers un objet (32 bits)
- `date`, une date de pointeur estampillé (64 bits)

Soit un total de 128 bits, ou 4 mots de 32 bits. Ce qui en fait une capacité partitionnée particulièrement économe en mémoire. Une des clés de ce résultat est qu'on n'utilise qu'une seule date pour les pointeurs estampillés vers la destination et l'objet.

Un domaine occupe nécessairement une page, donc l'adresse d'un domaine est alignée sur une page (4 ko), donc les 12 derniers bits du domaine sont toujours zéro et n'ont pas besoin d'être stockés. Un pointeur sur un domaine peut être stocké sur 20 bits. `destination` est stocké sur 22 bits car certains services ne sont pas implémentés dans des domaines : certains sont dans le noyau ,auquel cas `destination` contient l'adresse du point d'entrée correspondant dans le noyau[11] ; et on peut envisager utiliser les « small address space » [Lie95b] ou autre mécanisme. Les 2 bits restants servent à selectionner entre ces mécanismes.

Les 10 bits de libre du mot servent à encoder des droits d'accès à l'objet.

**Invocation : partie noyau** Le noyau ne propose qu'un seul appel système, qui permet d'invoquer une capacité. L'appelant fournit en paramètre l'index de la capacité qu'il veut invoquer (et un bit, indiquant si la capacité est stockée dans la C-list du domaine courant, ou dans celle du thread courant). La partie noyau se contente de vérifier si la capacité pointe bien vers le service, et de poursuivre l'exécution dans le service.

Le noyau accède au contenu de la capacité, et lit `destination`. Si `destination` est égal à 0 (capacité non présente), il retourne avec une erreur. Si `destination` pointe vers un service du noyau, il saute simplement à l'adresse contenue dans `destination` (les services du noyau ne peuvent pas être détruits, il n'y a donc pas besoin d'utiliser la date).

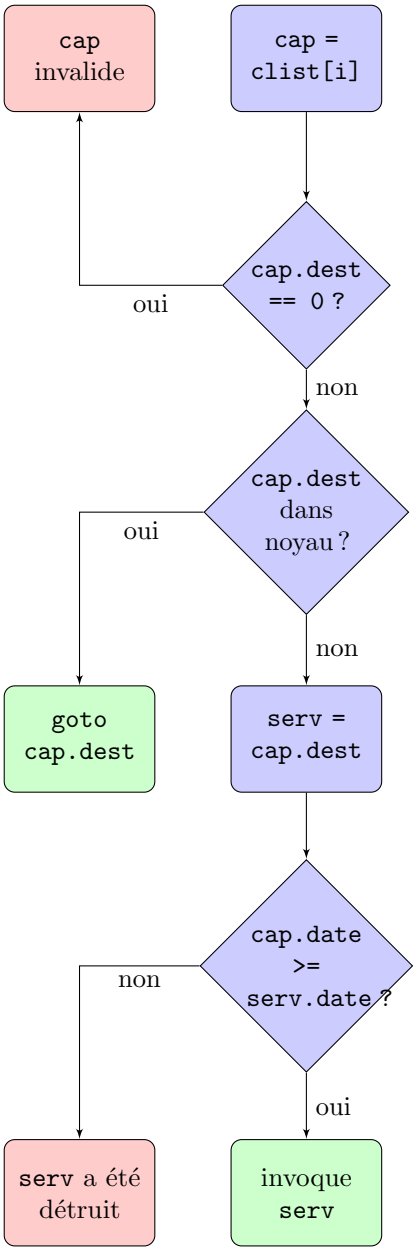
Si c'est un domaine, alors il compare la date de la capacité avec celle du service[12]. Le couple (`destination`, `date`) forme donc un pointeur universel vers un service. Le service est invoqué ssi la capacité a été créée après le service. Le service continue alors les vérifications sur la capacité. La Figure 3.8 (a) présente un organigramme de cet algorithme.

Cet algorithme repose sur le fait que seul le noyau peut écrire dans des capacités. Ainsi, il peut faire confiance à ce qui y est entreposé. Si on arrive à l'étape d'invocation du service, cela signifie qu'une capacité a bien été créée vers ce service, et que le service n'a pas été détruit depuis.

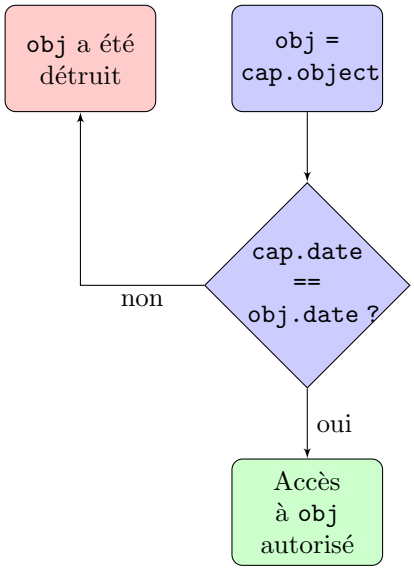
Cet algorithme permet au noyau de contrôler la communication entre un programme et un service : *il n'est possible d'invoquer un service que si on dispose d'une*

[11] Il y a 20 bits pour cette adresse, ce qui limite la taille du noyau à 1Mo de code, bien au delà du code de notre noyau qui fait moins de 60ko

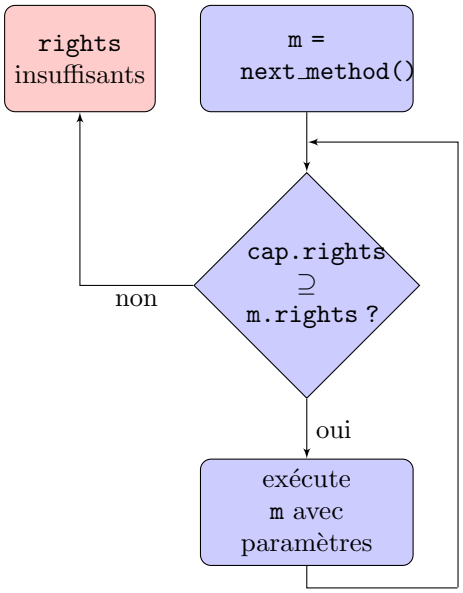
[12] Le noyau maintient un tableau avec une entrée par page physique, qui est utilisée pour stocker la date du service.




(a) Partie noyau




(b) Partie par invocation du service




(c) Partie par méthode du service

FIG. 3.8 – Étapes dans l'invocation d'une capacité. L'invocation passe d'abord par le noyau (à gauche), puis par le service (à droite). La partie (b) est exécutée au début de l'invocation du service, mais aussi quand l'exécution reprend dans le service après une préemption. La partie (c) est exécutée avant chaque nouvel appel de méthode. Les cadres rouges signifie une erreur, les cadres verts signifient que l'accès est autorisé pour l'étape suivante. Le retour se fait par la demande d'exécution d'une méthode de retour.

*capacité vers ce service*. Le noyau implémente donc une politique « channel control », où un channel est représenté par une capacité. Notons que la création de capacité vers un service n'est pas libre, et est contrôlée par des droits ; nous y reviendrons lors de la création de capacités. C'est cela qui permet à l'espace utilisateur de contrôler l'établissement des liens de communication, et de définir ainsi sa politique de sécurité.

Notons que `rights` et `object` ne sont pas interprétés par le noyau lors de l'invocation[13]. Ils sont juste rendus accessibles au service invoqué. Leur interprétation est exportée au service. Cela permet d'éviter au noyau le besoin de lire un tableau des objets dans le service, ce qui simplifie son implémentation et permet plus de flexibilité.

Notons que la date ne sert pas lorsque le service est dans le noyau. Cela repose sur le fait que les services du noyaux ne peuvent pas être détruits. Quand le système est statique et que les services ne peuvent pas être détruit, on peut également supprimer cette étape de vérification de la date.

**Invocation : partie service** Une fois le service invoqué, celui ci implémente le contrôle d'accès de la manière qu'il le désire. Certains service de mécanisme, qui ne servent qu'une seule ressource, n'ont pas besoin de faire de vérification. Ainsi, si on a accédé au service « affichage console », il est inutile pour le service de faire des vérifications supplémentaires.

Pour les services qui gèrent plusieurs objets, et en particulier pour les services de mécanisme pour un type de ressource, il faut servir la ressource à laquelle donne droit la capacité, et seulement celle-là. On utilise alors le champ `object` de la capacité, qui contient le numéro de la ressource servie. Le champ `date` est à nouveau utilisé pour la comparaison. Notre mécanisme de création de capacité assure qu'un objet et sa capacité sont créés en même temps, et avec la même date ; il faut ainsi juste vérifier que les deux dates sont bien égales. La révocation d'un objet est également extrêmement simple : il suffit de modifier la date de l'objet.

L'exécution de threads dans le service est préemptible ; ainsi quand l'exécution reprend, il est possible que la capacité ne puisse plus utiliser la ressource. C'est pourquoi cette vérification est refaite quand l'exécution reprend dans le service. En SMP, cela ne suffit pas ; on rajoute de plus une interruption entre processeur lorsqu'un objet est révoqué (§ 5.2.4.2).

Il est à noter que nous fournissons un « stub serveur » qui permet au concepteur de service de ne pas avoir à se soucier de tout celà. Le stub serveur fournit également le mécanisme de multicall.

**Multicall** Une fois que l'accès à la ressource est garanti, le mécanisme de *multicall* prend le relais. Le service permet de faire différentes opérations sur une ressource, qu'on appelle *méthode* (comme dans la programmation orientée objet). Des exemples de méthodes, pour un service de fichiers, sont `read` ou `write` ; pour le service de domaine, il y a `copy_cap` et `delete_cap`. Le multicall permet d'appeler plusieurs

[13]Certains droits sont interprétés par le noyau pour d'autres opérations, par exemple pour le droit de copier une capacité.

méthodes de manière successive, ce qui évite d'avoir à repasser par tout le mécanisme d'invocation de capacité pour chaque appel de méthode.

On peut autoriser l'accès à une ressource seulement pour certaines opérations. Pour cela, certaines méthodes vérifient que la capacité dispose des bons *droits d'accès* [Lin76, p. 412]. Par exemple, l'appel des méthodes `read` et `write` ci-dessus est autorisé par des droits d'accès différents (resp. `READ` et `WRITE`) ; par contre les méthodes `copy_cap` et `delete_cap` sont autorisées par un même droit (`MODIFY_CLIST`).

Ces droits sont encodé dans le champ `rights` de la capacité. Un bit correspond à un droit distinct ; chaque service peut encoder jusque 8 droits distincts (2 bits sont réservés par le noyau). La convention est qu'un bit est à 1 si le droit est présent, et à 0 si le droit est absent. Cette convention permet au noyau de garantir qu'il n'y a pas d'augmentation des droits (e.g. lors de la copie de capacité, § 3.3.4.2) sans avoir à interpréter ces droits. Hydra utilisait une convention similaire [WCC+74, p. 341][14]

L'annexe C.3.4.4 présente le multicall plus en détail.

### 3.3.4 Obtention de capacités

Nous nous intéressons maintenant à l'origine des capacités. Le système assure qu'une capacité ne peut pas être « forgée » par l'utilisateur : toute capacité est obtenue soit par le noyau créant une nouvelle capacité vers un objet, soit par le noyau copiant une capacité existante.

Le noyau assure également que les capacités ne sont plus modifiées une fois créées. Ces propriétés sont implémentées en segmentant la mémoire : les pages mémoires contenant les C-lists ne sont accessibles que par le noyau, jamais directement.

#### 3.3.4.1 Création de capacités

On ne crée une nouvelle capacité que lorsque l'on créée un nouvel objet. Toutes les autres capacités vers un objet sont obtenues en copiant cette capacité. Inversement lorsqu'on crée un objet, on est assuré que tout accès à cet objet se fait par la capacité créée simultanément (ou une de ses copies). Cette correspondance est importante : on s'en sert pour garantir formellement qu'un domaine n'a pas l'autorisation d'accéder à un objet (il ne peut accéder à un objet que s'il y a un moyen par lequel on lui aurait copié une capacité vers cet objet). Cela permet d'assurer l'isolation des ressources entre différentes tâches.

La création de capacité vers un service `serv` se fait en invoquant la méthode `create_cap` sur une capacité sur le domaine `serv`, qui dispose du droit `CREATE_CAP`. En particulier, la création de capacité n'est pas forcément faite depuis `serv`. Cela permet à un service de politique de créer de nouvelles capacités sans notifier le service de mécanisme, comme dans le protocole de la section 3.2.2.2.

L'implémentation est la suivante. Cette méthode prend en paramètre l'index `i` où on désire que la nouvelle capacité soit stockée, un paramètre `rights`, un paramètre `data`, et une adresse virtuelle `vaddr`. La méthode crée une capacité dans la c-list à

[14] Hydra utilisait cette convention pour vérifier des droits lors de l'invocation sans avoir à les interpréter, tandis que nous l'utilisons pour restreindre les droits lors de la copie.

l'endroit `i` indiqué par l'appelant, en remplissant les champs `destination`, `data`, et `rights` par respectivement `serv`, `object` et `rights`.

Le système incrémente un compteur d'estampille global 64 bits pour récupérer une nouvelle date, et la stocke dans le champ `date` de la capacité, et à l'adresse `vaddr`[15].

Pour le protocole de séparation politique/mécanisme, cela demande juste à ce que les dates des objets soient accessibles en écriture aux services de politique ; nous mettons en place de la mémoire partagée entre les services de politique et de mécanisme pour ce faire[16].

Ce mécanisme est simple, flexible et efficace, mais nous nous sommes rendus compte par la suite qu'il posait un problème de confidentialité. Il est possible de communiquer par canal caché en incrémentant le compteur d'estampille à un certain rythme. Cela n'a néanmoins pas d'importance pour la sécurité au sens partitionnement avionique.

Il y a différents moyens d'éviter cela. Un premier serait de resynchroniser le compteur à chaque décision d'ordonnancement : les premiers bits contiendraient la date, et seuls les derniers seraient incrémentés à chaque utilisation. Cela efface les informations sur les précédents incréments du compteur, et donc coupe le canal caché. Une autre possibilité est d'utiliser un compteur processeur, qui garantit être incrémenté à chaque fois qu'il est lu. Le compteur de cycles (timestamp counter) du Pentium présente cette garantie. Enfin, on pourrait faire la « partie service » de l'invocation par le noyau pour cacher les valeurs du compteur d'estampille.

#### 3.3.4.2 Copie de capacité

La copie de capacité est le deuxième mécanisme par lequel une capacité peut être obtenue. C'est notamment grâce à lui qu'on va pouvoir implémenter le partage des ressources.

La copie d'une capacité $c$ nécessite plusieurs conditions. Il faut que :

1. $c$ soit accessible (i.e. soit dans la C-list du domaine courant ou celle du thread courant)
2. $c$ aie le droit `COPY_RIGHT`, qui est un des droits réservés par le noyau ;
3. que l'on dispose d'une capacité $c'$ sur la C-list de destination (e.g. sur un objet thread ou domaine), qui aie le droit `MODIFY_CLIST`.

Quand ces conditions sont réunies, on invoque la méthode `copy_cap` sur $c'$ avec en paramètre l'index $i$ où se trouve $c$, l'indice $i'$ où placer la capacité une fois copiée, et un masque de droit `mask`. La méthode copie $c$ avec tous ses champs à l'emplacement $i$, sauf le champ `rights`, pour lequel la valeur de l'expression `rights & mask` est stockée.

[15] Nous arrangeons l'espace d'adressage afin que le noyau puisse s'assurer facilement qu'il n'est pas dangereux d'écrire à une adresse virtuelle donnée.

[16] Cette mémoire partagée est la raison principale pour laquelle nous avons restreint la hiérarchie des politiques à deux niveaux, dont le premier est statique

Le paramètre `mask` permet d'affaiblir les droits donnés à une capacité copiée. Par exemple, une capacité sur un buffer en lecture/écriture pourra être transmis en lecture seule. Cet affaiblissement des capacités suit le principe du moindre privilège (§ 2.2.2.4). Notons qu'il est important que la copie ne puisse qu'affaiblir des privilèges existants, et non les augmenter ; sinon un domaine pourrait accéder à des privilèges non prévus par simple copie de capacité. Le fait que la présence de privilège soit toujours représentée par un 1, permet au noyau de garantir que les privilèges sont toujours affaiblis sans que lui-même aie à interpréter ces privilèges (§ 3.3.3.2).

#### 3.3.4.3 Contrôle de la propagation des capacités

Du fait que l'utilisation d'une des deux méthodes ci-dessus est nécessaire pour obtenir une capacité, il résulte que toute capacité vers un objet est obtenue soit au moment de la création de l'objet, soit par copies successives de cette capacité initiale. Cette propriété va notamment permettre de prouver qu'une ressource n'est ni partagée, ni partageable, entre deux partitions. Il suffit pour cela que la politique ne fournisse une capacité qu'à une des partitions, et de s'assurer que cette partition ne puisse transmettre sa capacité à l'autre.

Avoir une politique qui ne fournisse une ressource qu'à une partition est simple. Nous nous intéressons au contrôle de la propagation.

La méthode principale consiste à limiter les canaux permettant de copier une capacité. Si une capacité $c$ est présente dans une C-list $C$, elle ne peut être copiée que de trois manières :

1. directe, si $C$ possède une capacité vers une autre C-list ;
2. indirecte, si $C$ possède une autre capacité $c'$, l'invocation de $c'$ permet la transmission de capacités vers le service gestionnaire de $c'$ ;
3. indirecte inverse, s'il est possible d'invoquer le domaine qui contient $C$.

En dessinant la configuration initiale du système, on peut trouver par transitivité quels domaines peuvent accéder à quelle capacité [MYS03]. En particulier, notre système étant structuré sous la forme d'un arbre (Figure 3.1 page 86), la propagation de capacités entre partitions n'est possible que par la complicité des services partagés.

Une autre manière pour restreindre la copie d'une capacité est de retirer son droit `COPY_RIGHT`. Mais on peut désirer qu'une capacité puisse être copiée librement dans une partition, mais pas à l'extérieur de cette partition (par exemple pour une politique de sécurité MLS) ; dans ce cas cette méthode est trop restrictive. On l'utilise surtout par sécurité, pour éviter que des capacités transmises lors d'une invocation ne puisse se propager (permet d'éviter la propagation indirecte).

La dernière méthode consiste à assurer qu'il n'y a pas de capacité existante sur une C-list avec le droit `MODIFY_CLIST`. Ainsi, il est impossible d'ajouter une capacité vers cette C-list. Cette méthode est également très restrictive, mais il arrive souvent qu'on sache qu'un domaine n'utilisera pas plus de capacités, en particulier pour la plupart des services ; ainsi suivant le principe du moindre privilège, on peut retirer tout droit d'ajouter une capacité à ce service. Cela garantit l'absence de propagation par l'intermédiaire des services partagés.

Ces deux dernières méthodes permettent de garantir l'absence de propagation des droits entre partitions sans avoir à faire confiance au code des services partagés.

### 3.3.5 Comparaison et conclusion sur le contrôle d'accès

Levy fournit une retrospective des difficultés dans l'implémentation des systèmes à capacité [Lev84, chapter 10], que nous catégorisons en deux parties : la recherche d'un objet et sa destruction.

**Recherche d'un objet** Une capacité doit contenir un identifiant pour un objet, autrement dit l'adresse, d'un objet. Les premiers systèmes à capacités n'étaient pas orientés objets, et les capacités contenaient alors des descripteurs de segments. Mais cela pose des problèmes lorsqu'un objet doit être déplacé en mémoire.

Les systèmes plus récents utilisent une table centralisée intermédiaire, utilisée pour stocker différentes informations [Lev84, § 10.4]. En particulier, ces tables sont utilisées pour retrouver le type de l'objet, qu'il faut connaître pour implémenter l'opération `typecall`. Ces tables fournissent également un degré d'indirection, permettant de faire du swapping des objets de manière transparente. Mais le problème principal de ces tables est qu'elles sont de taille fixe : ainsi l'allocation d'entrée dans cette table doit être régie par une politique d'allocation. CAL/TSS utilisait FCFS [LS76, § 6.3] ; d'autres considèrent les entrées dans cette table comme un cache (e.g. EROS [Sha99], CAl/TSS [LS76]).

Notre mécanisme peut être vu comme l'implémentation d'une table distribuée sur chaque service, en stockant le service de destination dans la capacité. Chaque service implémente le format de table qu'il désire, ou aucune table s'il le souhaite : les tables ne sont pas interprétées par le noyau. L'adresse d'un objet est représentée par le couple (service, numéro d'objet)[17]. Ainsi le noyau se contente-t-il de transmettre les requêtes au type indiqué par la capacité, toute autre opération étant déléguée à ce type. Cela suit le principe de minimisation des mécanismes communs (§ 2.2.2.6), simplifiant le noyau et rendant le mécanisme de capacités plus flexible. Notons que ce système ne demande pas à ce que les capacités aient une taille importante : nos capacités font chacune 16 octets. À titre de comparaison sur même matériel, les capacités d'EROS font 32 octets [SSF99, § 4.1] et celles de seL4 font également 16 octets [EDE08, § 3.2.2].

Le désavantage est que le service est désigné par son adresse physique, ce qui empêche son déplacement en mémoire ou swapping du domaine. Cela ne pose pas de problème sur les applications que l'on vise : soit le système est très contraint en mémoire (e.g. automobile), et en général statique et tient entièrement en RAM ou ROM ; soit il est destiné au hardware moderne, qui disposent de plus d'1Go de mémoire et où la perte de quelques kilo-octets n'est pas un problème.

Notons que le contenu du service peut être swappé. Par exemple, toutes nos capacités sur des objets systèmes (threads, espaces d'adressages, etc.) pointent directement sur des adresses physiques. Il semble faisable d'implémenter un service en espace utilisateur de « proxy noyau », qui constituerait un cache mémoire de

[17] Notons que l'adresse physique du service sert ainsi de « type » dans notre système

certains objets en plaçant les autres sur le disque, et agirait comme un proxy transparent sur les requêtes des applications.

Notons également notre mécanisme de multicall, qui permet d'appeler plusieurs méthodes successivement sans avoir à ré-exécuter tout le mécanisme de recherche d'objet. Ce mécanisme permettra au service d'implémenter des méthodes plus simples (§ 5.4.3).

**Destruction d'un objet** Une deuxième difficulté est la destruction d'un objet. Lorsqu'un objet $o_1$ est détruit, les ressources (mémoires, place dans une table des objets) sont réutilisées pour un autre objet $o_2$. Il ne faut pas que les capacités pour $o_1$ puissent être utilisées pour accéder à $o_2$.

Il y a différentes méthodes pour éviter cela [Lev84, § 10.4]. Certains systèmes interdisent la destruction d'un objet tant qu'il reste des capacités pointant sur cet objet (i.e. avec des compteurs de références) ; mais cela permet à des programmes malicieux d'empêcher la réutilisation de ressources. D'autres détruisent toutes les capacités qui pointent vers un objet pour le détruire (backpointers), mais cette opération prend un temps non borné (e.g. seL4 [EDE08, § 3.2.2]). Ces deux méthodes ne permettent pas d'obtenir une destruction des ressources en temps prévisible (§ 3.1.2.2).

Certains systèmes (e.g. Hydra et Systeme 38 [Lev84, p. 194], CAL/TSS [LS76, § 3.5]) utilisent un identifiant unique pour chaque objet : ainsi quand un objet est détruit, les capacités existantes ont un identifiant qui ne sera jamais réutilisé. Le problème est que le nombre de ces identifiants est large par rapport aux nombres d'objets présents dans le système, d'où l'utilisation de tables de hachages, qui rajoutent un surcoût pour l'utilisation de capacités.

Notre mécanisme suit cette méthode d'identifiant unique pour chaque objet ; la différence est que cet identifiant n'est pas un seul nombre, mais un triplet (service, numéro d'objet, date). En d'autres termes, l'identifiant unique se fait tout simplement en rajoutant une date aux capacités et aux objets. Nous avons généralisé ce mécanisme par la notion de pointeur estampillé (§ 3.3.3.1). Enfin, notre mécanisme utilise la même date pour le contrôle de l'accès au service par le noyau, et le contrôle de l'accès à l'objet par le service, réduisant ainsi la place nécessaire pour les capacités.

Cette idée de rajout d'une date pour identifier uniquement un objet peut être rapprochée des « allocation counts » d'EROS [Sha99, p. 102]. Les différences sont que dans EROS ces dates sont stockées dans une table centrale ; les entrées sont allouées par caching contrôlé par le noyau, donc l'allocation count peut ne faire que 32 bits ; et il y a toujours besoin de « backpointers » lorsqu'un objet est sorti du cache [Sha99, p. 103], l'allocation count servant seulement à éviter de modifier les capacités situées sur le disque dur.

**Conclusion sur le contrôle d'accès** Nous avons terminé l'explication de la conception de notre mécanisme de capacités, utilisé pour tout le contrôle d'accès dans le système. Il a comme caractéristiques d'être simple et flexible, de ne pas nécessiter de table centrale, et de permettre la révocation d'objet (et invalidation de toutes les capacités qui s'y rapportent) en temps constant, ce qui suit les principes énoncés pour cette thèse.

L'implémentation et sa preuve sont critiques pour la sécurité du système, mais ne sont pas nécessaires pour la compréhension de la thèse, aussi nous les avons placés en annexe A. Elle contient la formalisation du modèle du mécanisme de capacité, la preuve que l'implémentation correspond au modèle, les détails d'implémentation, et en particulier une implémentation efficace (quasi-lock free) en multiprocesseur.

## 3.4 Appel de service et prêt de ressource

Nous avons vu la nécessité de faire des appels de service à des services partagés, et comment l'accès à ces services partagés était contrôlé par les capacités. Nous nous intéressons maintenant à notre mécanisme d'appel de service, indépendemment des considérations précédentes de contrôle d'accès. Nous avons pu implémenter un mécanisme qui fasse en sorte qu'aucune ressource ne soit dépensée par le service : toutes les ressources sont décomptées par le client. La communication avec un service est ainsi complètement invulnérable aux dénis de service. De plus, l'appel d'un service n'a aucune conséquence sur l'ordonnancement, respectant ainsi le principe d'indépendance des politiques d'allocations (§ 3.1.1).

Le prêt de ressource est à la base de ce mécanisme, que nous appellons *prêt de thread*. Nous rappelons que nous appelons prêt de ressource un transfert de ressource qui ne nécessite pas l'intervention d'un domaine tiers responsable de la ressource (§ 2.3.2.4). Le temps CPU est prêté selon cette définition, même si le temps CPU « prêté » ne peut pas être récupéré. Nous allons décrire l'implémentation du prêt selon les différentes ressources.

### 3.4.1 Prêt de temps CPU

**Présentation** L'appel de service doit se faire sans que l'ordonnancement CPU n'en soit modifié. L'exécution de la requête du client dans le service ne doit donc se faire que sur le temps alloué au client, i.e. que lorsque le client aurait du être ordonnancé, comme dans la Figure 2.2(c) page 54(c). Le client prête donc de son temps CPU au service pour exécuter la requête.

L'appel de service demande donc à changer d'espace d'adressage (de celui du client à celui du service) sans modifier l'ordonnancement : la communication par *tunnel de thread* (§ 2.3.2.3) est donc appropriée pour l'implémenter. Le prêt de temps CPU est fait simplement en prêtant le thread, qui est le réceptacle du temps CPU. Différents systèmes ont par ailleurs démontré que ce mécanisme pouvait être implémenté efficacement. Or, l'appel de service est un mécanisme employé de manière extrêmement fréquente, ce qui rend ses performances cruciales [Lie93].

**Implémentation** Le prêt de thread justifie notre découplage des notions de threads, des domaines et d'espace d'adressage (§ 3.2.3.1) : prêter un thread signifie seulement lui faire changer de domaine (et le plus souvent d'espace d'adressage, car la plupart des domaines sont dans des espaces d'adressages différents).

Le prêt est implémenté de manière très simple. on associe à chaque domaine une adresse d'entrée. Lorsqu'une capacité est invoquée (et que l'invocation réussi), on

fait pointer le thread courant vers le domaine du service, le program counter est simplement mis à cette adresse d'entrée, on utilise l'espace d'adressage du domaine du service, et l'exécution se poursuit.

Lorsqu'il y a préemption puis retour, le retour se fait dans le domaine où était le thread au moment de la préemption ; donc le thread continue l'exécution dans le service s'il s'est fait préempter dans le service. Les services sont ainsi *multithreadés*.

Le retour dans l'espace d'adressage d'origine se fait en invoquant une capacité de retour transmise avec l'appel. Cette capacité de retour contient le domaine d'origine, avec le program counter et le pointeur de pile de retour. L'invocation de la capacité de retour restaure ces valeurs.

**Sélection du temps CPU prêté** Le prêt de thread va se fait de manière implicite (§2.3.2.4) : dès qu'on fait l'appel de service, le service peut utiliser tout le temps CPU du client pour faire la requête, jusqu'à ce qu'il aie terminé et retourne. Une donation explicite demanderait au client de spécifier les intervalles de temps pendant lesquels le service devrait faire la requête, au cycle prêt. Ce serait pénible, parce que le temps nécessaire dépend de la puissance de la machine, qu'il est difficile à prédire (statiquement ou dynamiquement), et qu'il est inutile de gérer le temps CPU au cycle prêt.

Nous permettons le prêt par containers par émulation : il suffit pour cela de créer un nouveau thread qui fera l'appel de service, et d'agir comme un ordonnanceur en lui donnant de son temps CPU, de manière limitée. Notons que comme le temps est compté exactement, des timeouts sur la durée d'exécution dans le service sont indépendants de l'exécution des autres tâches, et ne souffrent donc pas de problème de reproductibilité [Sha03, p. 2].

**Invocation de capacité aveugle et notification** Une conséquence intéressante du prêt de temps CPU est qu'il permet ce qu'on appelle *invocation de capacité aveugle*. Le client peut transmettre des capacités au service, que ce dernier va devoir normalement invoquer. Quand l'appel de service n'est pas fait par prêt de temps CPU, le service ne peut pas invoquer une telle capacité sans s'assurer du destinataire[18], car ce destinataire pourrait rentrer dans une boucle infinie et bloquer le service. Au contraire, cette opération est sans risque lorsqu'il y a un prêt de temps CPU : en fournissant une capacité malicieuse, le client ne peut que perdre de son propre temps CPU.

Nous permettons cependant au service d'utiliser de son propre temps CPU (i.e. d'avoir ses propres threads, et non un utilisé par le client). On ne peut pas dans ce cas invoquer une capacité à l'aveugle. Nous prévoyons de restreindre l'invocation de capacité dans ce cas à une *notification*. Cela consisterai à ce qu'une opération soit effectuée par le noyau (comme écrire à une adresse, réveiller la tâche ou notifier un ordonnanceur...) qui soit garantie d'être en un petit temps borné. Ce mécanisme est utile pour réveiller des clients pour l'implémentation d'appels systèmes bloquants.

notification

[18] EROS dispose pour cela d'un moyen d'identifier le service à appeler (par les "brand slots" [Sha99, page 25])

Ce problème de confiance se pose aussi dans les systèmes basés sur les communication synchrone ou asynchrone ; d'autant plus qu'ils n'ont que des thread propres. En conséquence, les mécanismes de notifications sont courants, par exemple Mach [Loe92], `send` dans EROS [SFS96]). Notons qu'EROS semble avoir plus tard supprimé le `send` (ou ne l'utilise pas pour envoyer des notifications), ce qui demande une émulation plus compliquée [SSS04].

**Autres ressources temporelles** Il existe d'autre ressources temporelles que le temps CPU : le partage de l'accès au disque ou à la carte réseau est fait par ordonnancement. Mais le transfert de ces ressources temporelles est inutile en pratique, et n'est pas implémenté dans Anaxagoros. Nous allons maintenant étudier le transfert des ressources partagées spatialement, et en premier lieu de la mémoire.

### 3.4.2 Prêt de mémoire transitoire

#### 3.4.2.1 Besoin de prêt transitoire

Avec le transfert de temps CPU, il faut aussi transférer de la mémoire. En effet, à moins que la requête ne soit triviale, l'exécution de cette requête consomme une pile[19]. Comme l'exécution de la requête peut être préemptée dans le service, il peut y avoir plusieurs clients simultanément, et il faut une pile par client servi. Et à moins que le nombre de clients soit borné, la mémoire pour ces piles ne peut pas provenir d'une sous-pool statique dans le service ; ce serait l'application de la politique FCFS sur la mémoire du service, et entraînerait notamment des blocages incontrôlés. Le prêt de mémoire transitoire (i.e. intra-appel) est donc nécessaire pour que l'ordonnancement soit indépendant des domaines de protections. Le prêt de mémoire semi-permanent (i.e. inter-appel) est abordé en section 3.4.4.

#### 3.4.2.2 Implémentation

**Problème de la consommation d'espace d'adressage** Transférer de la mémoire au service consiste à installer dans l'espace d'adressage du service un certain nombre de mappings. Le problème est que l'espace d'adressage du service est limité : si trop de threads prêtent de la mémoire au service, tout son espace d'adressage sera consommé, ce qui empêchera l'entrée de nouveaux threads. L'espace d'adressage d'un service est en fait une ressource limitée, et nous venons donc de décrire encore une utilisation de FCFS, sensible à un déni de ressource.

La solution naturelle à ce problème est de « prêter de l'espace d'adressage ». Nous implémentons cela grâce aux *thread-local mappings*, qui sont une partie de l'espace d'adressage propre au thread.

**Implémentation du prêt transitoire** Concrètement, tout espace d'adressage est divisé en deux parties : une partie propre au domaine, et une partie propre au

[19] et même si la requête ne consomme pas de pile, il faut toujours pouvoir stocker les registres lorsqu'une préemption survient.

thread[20]. À chaque fois qu'un thread change de domaine (i.e. appel de service), ou lors de toute décision d'ordonnancement, on change d'espace d'adressage. La Figure 3.9 présente cette implémentation.

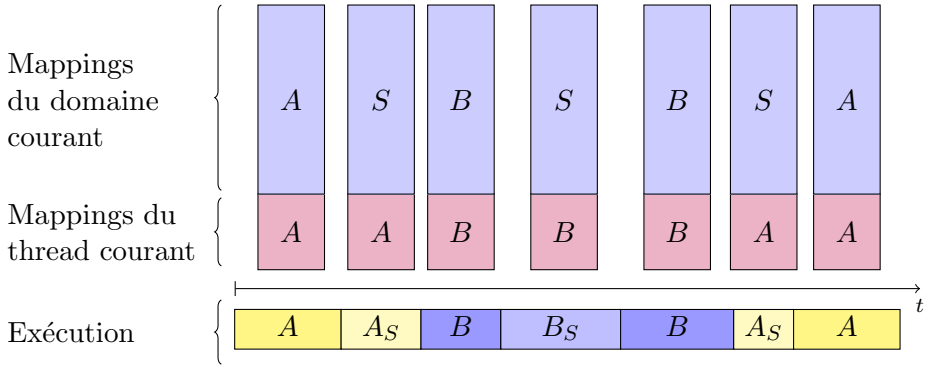



FIG. 3.9 – Évolution de l'espace d'adressage de la tâche. La partie domaine de l'espace d'adressage change lors des appels de services, mais par la partie thread. Lorsque le thread change, l'espace d'adressage change pour accueillir les mappings du thread choisi et du domaine dans lequel est ce thread. Sur IA32, le mapping du noyau est présent dans tous les espaces d'adressages.

Ce mécanisme n'est pas lent. Sur x86, tout changement d'espace d'adressage nécessite un flush complet du TLB. Mais nous avons arrangé le code et les structures de données afin que la modification de l'espace d'adressage se fasse avant ce flush ; ainsi seule une écriture dans une table des pages est nécessaire pour ajouter un mapping. Notons cependant que sur d'autres types d'architectures, de meilleures solutions sont envisageables. En particulier, sur les architectures pour lesquelles la gestion du TLB est déléguée au système d'exploitation, ce mécanisme peut être facilement et efficacement implémenté.

Ce mécanisme a également un avantage en sécurité : il est impossible pour un thread d'accéder directement aux données d'un autre thread. Ainsi, pour qu'il y ait communication entre deux threads, il faut nécessairement passer par de la mémoire propre au service. Cela permet d'augmenter la garantie de non-interférence entre clients différents.

Notons que dans le cas de single-space operating systems, dans lequel les adresses virtuelles de toutes les tâches sont identiques, un mécanisme alternatif peut être envisageable, qui permette également au service de ne pas consommer d'espace d'adressage. Il suffirait de modifier le service pour qu'il puisse accéder à la mémoire du client. Il faut faire attention à ce que les droits d'accès ne fassent pas consommer de mémoire au service.

**Problème du partage des mappings** Le prêt de mémoire accorde au service le droit d'accéder à une partie de la mémoire allouée au client. Mais rien n'empêche le client d'accéder à cette mémoire par ailleurs, sans que le service puisse le savoir.

[20]Les processeurs x86 requièrent également une troisième partie, propre au noyau, qu'on ne représente pas

Cela pose en particulier problème pour la pile, puisque cela permet de modifier l'exécution dans le service.

Pour la pile, nous avons résolu le problème de manière simple : la pile utilisée est l'UTCB, qui est toujours située à adresse fixe du thread-local mapping du thread courant, et ne peut pas être mappée à un autre endroit. Cela garantit que l'UTCB d'un thread ne peut être modifiée que par ce thread[21]. Cette utilisation est similaire à la « pile noyau » dans les noyaux monolithiques.

Nous n'avons pas encore eu besoin d'implémenter une telle garantie pour d'autres mappings. Généralement, on pourra le faire en permettant au client de fournir au service la preuve que le mapping qu'il utilise est unique. Une garantie similaire est nécessaire pour le prêt semi-permanent (§ 3.4.4).

Un des avantages des thread-local mappings est qu'ils facilitent la preuve que le service ne fait pas communiquer deux clients : toute communication entre deux threads dans le service doit passer par un buffer intermédiaire. Cette preuve sera établie en montrant que deux threads n'accèdent que rarement aux même données, et que les données partagées ne permettent pas de communication.

**Utilisation des thread-local mappings** Pour le moment, le thread-local mapping fait 4ko d'UTCB utilisé comme pile, pour le passage des paramètres/multicall, pour la sauvegarde des registres lors de la préemption (et autres mécanismes de contrôle de la préemption présenté en section 4.2.3).

Les autres mappings correspondent chacun à une entrée dans la table des pages de premier niveau (donc à une plage de 4Mo chacun), et peuvent être installés tout simplement par la modification d'une entrée dans la table des pages du premier niveau. Ils permettent de passer de gros volumes de données. Notons que ces mappings peuvent être simultanément présents dans l'espace d'adressage du client : cela permet la copie directe de données du service dans l'espace d'adressage du client.

En rétrospective, il serait plus judicieux d'avoir un autre mapping de 4ko et un mapping de 4Mo. La transmission de données par tranche de 4ko est fréquente (les appels `read` et `write` UNIX fonctionnent ainsi), et ce mapping pourrait être installé dans la même table des pages que l'UTCB, réduisant le coût du TLB miss.

#### 3.4.2.3 Extension au SMP

**Extension naturelle** L'extension naturelle au SMP est de réserver, dans chaque espace d'adressage, une partie de l'espace d'adressage pour le thread-local mapping pour chaque processeur. L'espace d'adressage est ainsi divisé, pour $M$ processeurs, en $2M + 1$ parties.

Outre la consommation d'espace d'adressage, cette extension pose des problèmes de sécurité. Plus rien n'empêche, dans un espace d'adressage, un thread de pouvoir accéder aux thread-local mappings d'un autre thread. Et cela, même quand l'autre thread est parti ou détruit : en effet, même si on retire le mapping thread-local de l'espace d'adressage, il peut être resté dans le cache TLB des autres processeurs.

[21] Un autre avantage, dans notre implémentation, est qu'il n'y a même pas besoin de modifier une entrée dans la table des pages lors d'un appel de service.

Cela demanderait de faire un TLB shootdown à chaque appel de service, et serait extrêmement coûteux en performances.

Même si on suppose faire confiance aux services, le même problème se retrouve pour des clients multithreads, ce qui leur permet de modifier la pile sur un autre processeur en train d'exécuter un service, par exemple.

Pour se protéger contre ce comportement, on peut penser utiliser le mécanisme de segmentation du Pentium, qui permet de restreindre l'accès à une plage d'adresses virtuelles. Malheureusement, ce mécanisme ne fait pas seulement de la protection, mais également de la translation d'adresses. Ce qui implique que les adresses utilisées par différents segments ne correspondent plus. Les compilateurs actuels ne gèrent pas les architectures segmentées, et rajouter une API pour les translation d'adresses entre segments ne serait pas commode et surprenant pour le programmeur.

**Autres solutions** Nous avons donc envisagé d'autres solutions. L'une d'elle consisterait à avoir un espace d'adressage par service par CPU (i.e., une table des pages de premier niveau différente pour chaque processeur ; le reste serait partagé). Ainsi, chaque processeur utiliserait la même plage d'adresses pour les thread-local mappings, mais ne pourraient plus accéder à celui des autres. Le principal problème de cette solution est le coût en mémoire occasionné. De plus, elle oblige chaque domaine à tenir compte du nombre de processeurs qui peuvent l'exécuter simultanément.

Ce coût peut être éliminé en générant l'espace d'adressage à la volée au moment de l'appel. Concrètement, le noyau garderait une page des tables de premier niveau par CPU. Lorsqu'un thread invoque un service, les entrées nécessaires, peu nombreuses, sont copiées dans cette table des pages avant qu'elle soit utilisée. Cette technique permet d'éviter tout surcoût en mémoire, mais rajoute un surcoût en temps CPU à chaque invocation de service ou préemption de thread. Sur Pentium, il faut copier une entrée par tranche de 4Mo d'espace d'adressage. La majorité des services et programmes font moins de 4Mo, mais il y en a qui peuvent être relativement gros (systèmes de fichiers ou Linux paravirtualisé)[22].

Finalement, la meilleure solution est peut être une solution mixte entre les deux : suivant la taille de l'espace d'adressage, un domaine pourra choisir l'une ou l'autre des solutions.

Notons que ce problème est spécifique aux systèmes pour lesquels les tables des pages sont gérées en hardware, et ne se pose pas pour les architectures pour lesquelles le TLB est géré en logiciel. Dans ce dernier cas, chaque processeur maintient tout simplement dans son TLB, les thread-local mappings de son thread, et tous les processeurs partagent la même plage d'adresses virtuelles pour ces mappings.

Certains processeurs ARM ont également un mécanisme qui permet de rediriger une région de l'espace d'adressage vers un autre endroit [WTUH03], qui pourrait également être mise à profit.

[22] On peut penser faire l'écriture de certaines entrées de manière paresseuse : l'entrée est installée lorsqu'un défaut de page se produit. Il n'est pas certain que cela améliore les performances, même pour les tâches non temps réel.

### 3.4.3 Prêt de capacités

**Motivation** Nous voyons maintenant comment prêter les autres ressources (partagées spatialement). L'accès à ces ressources se faisant par capacités, il suffit de prêter des capacités pour prêter une ressource. Le prêt de capacités consiste à donner au service la possibilité d'invoquer la capacité pendant la durée du prêt.

Le prêt de capacités a d'autres utilités que le prêt de ressource. Il permet par exemple de donner au service la possibilité de communiquer avec une troisième entité (on utilise par exemple cela pour permettre au service de notifier l'ordonnanceur du client). On fournit également au service une capacité de retour, pour que le thread puisse retourner au client. On peut également penser utiliser les capacités comme jeton d'authentification. La transmission de capacité est quelque chose de très naturel dans les systèmes à capacités, et tous les systèmes le permettent.

**Allocation d'entrées dans la C-list du service** La solution naturelle pour prêter une capacité est de la copier dans la C-list du domaine du service depuis la C-list du domaine du client, et de la supprimer une fois le prêt terminé. Mais procéder ainsi pose différents problèmes :

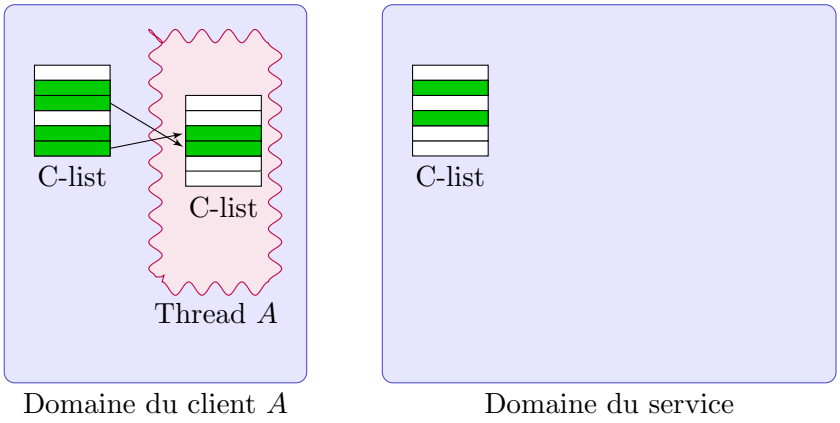
- *de stockage* : les capacités prêtées remplissent la C-list du service, qui est de taille finie. Les allouer au fur et à mesure serait un exemple typique d'utilisation de FCFS, et donc de vulnérabilité aux dénis de ressource. Il faut donc une politique d'allocation sur les entrées de la C-list, qui est pénible à écrire ;

- *de gestion* : si on met en place une politique des entrées dans la C-list, cette politique est implémentée par le service. Or, c'est le noyau qui place les capacités dans la C-list, et ce au moment de l'invocation, i.e. sans que le service ne sache encore « qui est le client ». Pour gérer convenablement les entrées dans la C-list, il faudrait mettre en place un protocole d'échange entre le noyau et le service, qui risque d'être complexe et peu efficace ;

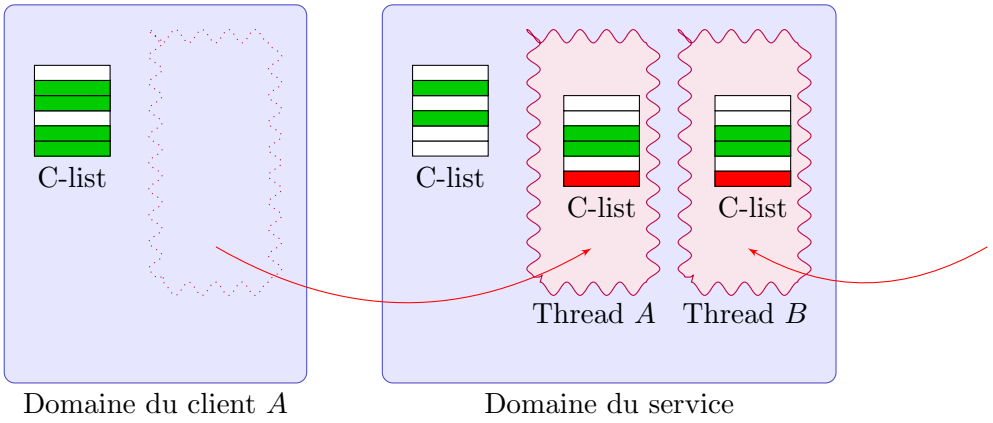
- *de sécurité* : rien n'empêche un service en train d'exécuter la requête d'un client $C_1$ d'utiliser la capacité donnée par un client $C_2$, ce qui contredit le principe du moindre privilège (§ 2.2.2.4).

**C-list dans les threads** Le meilleur moyen de résoudre ce problème est de prêter des places de C-list. C'est la raison pour laquelle le thread contient également une C-list dans notre système.

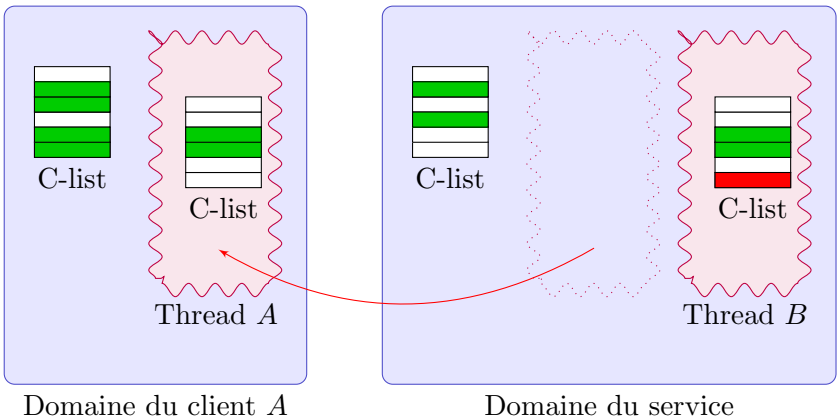
Le prêt de ressources se déroule ainsi (voir Figure 3.10) : le client copie tout d'abord ses capacités depuis son domaine vers le thread courant. Puis il transmet son thread au service, ce qui signifie que le service peut utiliser les capacités présentes dans le thread (en plus du temps CPU et des mappings mémoires du thread déjà mentionnés). Enfin, le service rend le thread au client, en invoquant la *capacité de retour* [SFS96] insérée par le noyau lors du prêt du thread (dans la C-list du thread, et non celle du domaine du service). Toute capacité transmise du service au client peut également être transférée par l'intermédiaire de la C-list du thread. Cette




(a) Le client copie les capacités à prêter de la C-list de son domaine vers celle de son thread




(b) Le client prête son thread au service. Une capacité de retour est insérée par le noyau dans la C-list du thread prêté. Si le thread est preempté, l'exécution recommence dans le service. D'autres clients peuvent simultanément prêter leur thread au service.




(c) Le service invoque la capacité de retour, et le thread retourne au client. Le service peut éventuellement avoir copié des capacités dans le thread avant de ce faire.

FIG. 3.10 – Etapes de l'appel de service par prêt de thread

solution résout tous les problèmes mentionnés ci-dessus pour les prêts de capacités transitoires.

Mais on ne peut pas prêter un thread entre plusieurs appels de service par nature, donc cette solution ne se prolonge pas pour les prêts semi-permanents. Pour ces derniers il est toujours nécessaire d'écrire une politique d'allocation des places dans la C-list du domaine. Cependant, cette politique est maintenant beaucoup plus simple à écrire : comme le service a la connaissance du client, il lui suffit de demander au noyau de copier les capacités de la C-list du thread vers celle de son domaine lorsque c'est nécessaire. Notre mécanisme résout donc aussi le problème de gestion pour les prêts semi-permanents.

Un problème dont nous n'avons pas parlé est la révocation des capacités prêtées au service. En effet à la fin de l'appel, une capacité prêtée de manière transitoire à un service ne doit plus être utilisée par le service ; suivant le principe du moindre privilège (§ 2.2.2.4), le service ne devrait même plus invoquer cette capacité. Ce droit est retiré automatiquement par le prêt de thread ; faire de même par copie de capacité entre domaines requiert l'utilisation de schémas plus complexes comme le « caretaker pattern » de Redell [Red74, p. 40].

**Analogie avec le modèle objet** Le modèle par prêt de thread est analogue à l'implémentation de l'appel de méthode passif du modèle objet, par exemple dans des langages comme C++. Dans celui-ci, la pile contient des données locales au calcul courant, et l'objet encapsule des données partagées entre plusieurs calculs. L'appel de méthode utilise la pile pour passer des arguments entre les objets. Dans notre modèle par passage de thread, les capacités sont analogues aux données, le thread à la pile, les domaines aux objets, et l'appel de service à l'appel de méthode. Dans les langages sûrs, l'analogie va encore plus loin : un objet doit disposer d'une référence pour pouvoir appeler un autre objet, tout comme une capacité vers un service est nécessaire pour pouvoir l'appeler.

Le modèle de programmation objet est bien connu des programmeurs, ce qui permet au modèle de sécurité par prêt de thread de suivre le principe d'acceptation psychologique (§ 2.2.2.7). Il leur est plus naturel qu'un appel par communication synchrone, qui serait l'équivalent dans le modèle objet d'un appel actif d'une méthode protégée par un « monitor ».

Notre modèle d'invocation est donc extrêmement naturel pour des systèmes à capacités, qui sont étroitement liés à la conception orientée objet (voir par exemple [Lin76, Lev84, MYS03]).

**Analogie avec le prêt de mémoire** Le prêt de mémoire transitoire que nous avons décrit précédemment est également proche du prêt de capacités. Une entrée dans une table des pages peut en effet être considérée comme un droit d'accès à la mémoire, et la table des pages est ainsi une sorte de C-list. Les différences sont qu'il y a des garanties supplémentaires à fournir pour la mémoire (le fait qu'un mapping n'est pas partagé), et le fait que les mappings mémoire sont interprétés directement par le matériel, au contraire des capacités.

### 3.4.4 Prêt de mémoire semi-permanent

**Présentation et applications** Nous abordons maintenant un dernier type de prêt de ressource, le prêt de mémoire semi-permanent. Ce type de prêt est utile lorsqu'un service a besoin d'un buffer de données pour servir un client ; en ce cas le client fournit toute la mémoire nécessaire pour ce buffer. Un exemple de tel prêt de mémoire peut être vu dans l'implémentation de la pile réseau dans EROS de Sinha, Sarat et Shapiro [SSS04]. D'autres utilisations peuvent être l'établissement de communication par mémoire partagée entre deux tâches.

**Cas particulier du noyau** Un cas particulier de prêt de mémoire semi-permanent est le prêt de mémoire au noyau, qui permet de résoudre le problème de l'allocation de mémoire noyau (§ 2.1.2.2). Notre solution pour ce problème consiste tout simplement à demander au noyau de convertir certaines pages en d'autres (table des pages, domaines, threads). Cette opération est simplifiée par le fait que tous les objets systèmes sont de taille une page (§ 3.2.3.2). Cette conception est similaire à celle Xen [BDF+03] ou encore de seL4 [EDE08]. L'annexe C en détaille l'implémentation.

**Trusted mapping object** La mémoire prêtée se fait en utilisant un « trusted mapping object » (TMO), qui fait office d'intermédiaire de confiance entre le client et le service. Le service ne peut accorder aucune confiance au client ni aux capacités qu'il envoie. On pourrait demander au client de faire confiance au service, mais le service de politique mémoire qui fournit la mémoire au client ne peut pas faire confiance au service pour qu'il lui rende de la mémoire si nécessaire. Il doit donc y avoir un moyen de forcer la récupération de la mémoire. À partir de cela, il est aussi simple de permettre au client lui-même de récupérer la mémoire. Le TMO permet ainsi au service d'utiliser la mémoire prêtée en toute confiance, et fournit au client (ou son fournisseur de mémoire) l'assurance de pouvoir récupérer la mémoire à tout moment.

Le fonctionnement général est le suivant. Le client sélectionne un certain nombre de pages, et demande au service de mécanisme mémoire de les « transformer » en TMO. Il reçoit en échange une capacité. Il transmet cette capacité au service (en retirant quelques droits). Le service invoque alors cette capacité[23] pour installer les pages prêtées par le client ; une fois installé, le mapping n'est plus modifiable. Le service vérifie que les droits d'accès sur les mappings sont paramétrés comme il le souhaite, puis peut commencer à les utiliser. Si le client veut récupérer la mémoire prêtée, il invoque sa capacité (non amoindrie) pour supprimer le TMO, ce qui supprime également le mapping du service, et invalide toutes les capacités vers le TMO.

#### 3.4.4.1 Conception

Le trusted mapping object est seulement une table des pages de deuxième niveau particulière, qui permet seulement de fournir des assurances sur le nombre de mappings installés.

[23] il s'agit d'un appel de service aveugle (§ 3.4.1).

Basiquement, le client choisi les pages qu'il veut prêter en les inscrivant dans une table des pages, et en paramétrant les droits en lecture et écriture qu'il souhaite, puis la transforme en TMO. Il prête le TMO par capacité au service, qui l'installe en modifiant une entrée dans sa table des pages de premier niveau. La destruction du TMO ne fait que modifier cette entrée dans l'espace d'adressage du service.

**Droits d'accès du client** Une entrée dans une table des pages normale contient les droits de lecture et d'écriture vers une page (donc les droits d'accès du service). En plus, les entrées du TMO contiennent les droits de lecture et d'écriture du client (en utilisant 2 bits libres des entrées de la table des pages). Ces bits sont choisis par le client quand il crée le TMO, et assurent les deux propriétés suivantes :

- si une page du TMO ne peut pas être écrite par le client, personne ne peut plus la mapper en écriture tant que le TMO n'est pas détruit ;
- si une page du TMO ne peut pas être lue par le client, personne ne peut plus la mapper en lecture tant que le TMO n'est pas détruit.

Ces droits du clients ne concernent donc pas seulement le client, mais toutes les tâches. Cela évite que le client permette à un tâche tierce de modifier les mappings, même si lui-même n'y a pas le droit (les droits du clients sont ainsi improprement nommés). Ces deux restrictions sont implémentées en marquant les pages concernées comme « non-mappables » lors de la transformation en TMO. Tous ces bits sont facilement vérifiés par le service, qui peut accéder au TMO en lecture. Enfin, le TMO n'est plus modifiable une fois mappé par le service. Ces propriétés assurent au service qu'il est seul à pouvoir accéder à la mémoire prêtée.

**Récupération de la mémoire prêtée** Imaginons que le client fournisse un TMO au service, et que celui ci crée une myriade d'espaces d'adressages dans lequel il installe dans tous le TMO. Cela pose deux problèmes : la récupération de la mémoire se ferait alors dans un temps long non borné ; et il faudrait une certaine gestion pour pouvoir retrouver la liste des table des pages qui mappent le TMO. Ce problème est similaire à celui des pointeurs estampillés et « backpointers », mis à part que le format des tables des pages est fixé par le hardware, et qu'on ne peut pas utiliser de dates dans les entrées de table des pages[24].

Pour résoudre ces problèmes, le TMO restreint le nombre de fois où les pages prêtées peuvent être mappées par le service à 1. Tout d'abord, le TMO ne confère pas le droit d'installer les pages qu'il contient de manière individuelle ; seul le TMO peut être installé dans son intégralité. Chaque page du TMO ne peut donc être mappée qu'une fois par le service. De plus, le TMO lui même ne peut être mappé qu'une fois (donc dans une seule table des pages de premier niveau). Lorsque le TMO est installé, l'adresse physique de l'entrée dans la table des pages de premier niveau où est mappée le TMO est enregistrée. Ainsi, lorsque le TMO est détruit, il suffira de modifier cette entrée pour assurer que toute la mémoire a été récupérée.

[24] Et même si on le pouvait, cela ferait grossir les tables des pages.

La gestion est donc simple, et la récupération de toute la mémoire se fait en temps constant borné.

Ainsi, le service peut transmettre et copier sa capacité vers le TMO sans aucune restriction ; cela ne pose aucun problème pour la récupération de la mémoire prêtée.

**Conclusion** Le TMO permet le partage de ressources par un protocole relativement simple. Son implémentation est dans un stade préliminaire, mais nous nous attendons à ce que la mise en place de ce mapping partagé se fasse de manière relativement rapide. Une fois mis en place, il est évidemment extrêmement rapide.

Le prêt transitoire est à préférer lorsqu'il suffit. Il est en effet plus rapide à mettre en place, et ne consomme pas d'espace d'adressage, contrairement au prêt semi-permanent. L'allocation d'espace d'adressage du service peut être fait en en limitant le nombre de clients, et en accordant une entrée dans la table des pages de premier niveau du service par client (ce qui permet tout de même plus de 1000 clients, avec 4Mo par client).

On peut comparer ce mécanisme avec l'utilisation de space banks dans EROS (e.g. [SSS04]), où la révocation se fait également immédiatement. La principale différence est que notre sélection de la mémoire est explicite, et que nous n'utilisons pas de caching de la mémoire vive.

On peut également comparer notre mécanisme au mécanisme de quota de Bastei/Genode, une architecture basée sur L4 : dans celle-ci la récupération des ressources se fait en demandant au service de libérer les ressources. L'idée est d'imposer une durée maximale pour la libération des ressources, et de tuer le service s'il agit mal [FH06, § 2.2]. Cela est contraire à nos principes d'opération en temps CPU prévisible (§ 3.1.2.2), et le refus d'avoir du temps codé « en dur » (§ 3.1.4.3)

### 3.4.5 Évaluation et conclusion

**Évaluation** Une première évaluation indique que qu'un appel de service avec retour dans le client demande environ 1500 cycles (c'est à dire environ 750 cycles pour l'appel de service). Ces chiffres sont de même ordre de grandeur que pour les appels de service optimisés d'autres systèmes, comme L4 ; et donc bien plus rapide que les appels de services de micronoyaux de première génération, comme Mach.

**Résumé** Cette section a présenté un mécanisme de communication pour l'appel de service pour lequel toutes les ressources utilisées sont prêtées par le client. Grâce à cela, ce mécanisme de communication est invulnérable aux attaques par déni de service : l'exécution des requêtes des clients est insensible à la présence des autres clients. De plus, le fait d'appeller un service est complètement indépendant des décisions d'allocation.

Ce dernier point est l'avantage majeur par rapport à la communication synchrone, utilisée par L4 [Lie93, Lie95a] ou EROS [SFS96]. L'autre avantage est que cela permet au service d'être multithread, donc de profiter du parallélisme des machines multiprocesseurs[25]. C'est le fait que les services sont multithread qui a

[25]On pourrait objecter que notre mécanisme de communication force les services à être multi-

rendu nécessaire le développement de techniques pour le prêt transitoire des autres ressources ; toutes ces techniques se basent sur un modèle simple, le prêt de thread.

## 3.5 Structure et implémentation des services

Nous avons expliqué en quoi l'intégration de systèmes hétérogènes demandait l'application de deux principes : la haute sécurité et l'indépendance des politiques d'allocation (§ 3.1). La méthode de structuration et les mécanismes de communication et de prêt de ressource que nous avons présenté sont très importants, mais ne suffisent pas pour réaliser ces systèmes. Par exemple, un service qui pourrait envoyer dormir un thread de manière imprévisible ne permettrait pas d'obtenir un système indépendant de l'ordonnancement CPU, même si la communication est basée sur le prêt de ressource.

Dans cette section, nous présentons les contraintes et des éléments de solution pour que les services suivent les principes énoncés. Une méthodologie complète pour l'écriture de ces services sera présentée par la suite, qui répond aux problèmes identifiés dans cette section. Nous nous concentrons essentiellement sur les service de mécanisme, qui sont partagés par des clients ne se faisant pas confiance. La raison est qu'on a toujours besoin du service de mécanisme, tandis qu'on peut supprimer le besoin d'un service de politique en allouant les ressources de manière statique.

Nous avons concentré l'analyse sur la manière de supporter l'indépendance des politiques, et structuré cette analyse autour des trois types de ressources : CPU, mémoire, et autres ressources. Pour chaque type de ressource, nous séparons l'analyse entre ressources propres et prêtées.

### 3.5.1 Ressources propres et prêtées

Le mécanisme du prêt de ressource permet au service d'utiliser des ressources prêtées par le client. Mais le service dispose aussi de ressources qui lui sont propres (i.e. qui lui ont été allouées), que nous appelons *ressources propres*.

ressources propres

Les problèmes liés à l'emploi des ressources sont nettement différents selon qu'elles sont propres ou prêtées.

**Allocation des ressources propres** Le service doit être conçu pour ne pas "consommer" de ses ressources en réponse à des requêtes du client (i.e. ne doit pas réserver des ressources à un client uniquement), car cela conduirait à un déni de service (§ 2.3.2.1). Pour éviter cela, nous nous efforçons de maximiser l'usage des ressources prêtées, et de structurer le système pour identifier toutes les ressources (principe § 3.1.1.1).

threadé, ce qui les complexifie ; mais rien n'empêche de placer un mutex à l'entrée et la sortie du service pour reproduire le comportement de la communication synchrone.

**Révocation des ressources prêtées** Les problèmes liés à l'emploi de ressources prêtées viennent du fait que le contrôle de l'allocation de ces ressources est fait par une entité extérieure au service, et en laquelle le service ne peut pas avoir confiance. Le service peut se voir *révoquer* le droit d'utiliser ces ressources au beau milieu d'une opération, laissant ainsi derrière lui un état inconsistant (structures à moitié écrites, verrou non libéré, etc...).

Cette révocation n'est pas forcément une décision du client : elle peut venir de son service d'allocation qui lui retire ses ressources, ou du fait que le client s'est fait détruire entre temps. Ainsi toute ressource prêtée, que ce soit de manière transitoire ou semi-permanente, peut être révoquée à tout moment.

Une manière générique de procéder pour éviter que les ressources soient révoquées au milieu d'une opération est l'*épinglage (pinning)*, qui consiste à interdire la révocation de ressources pendant un certain temps (jusqu'à ce que le service les "dépingle"). Mais cela occasionne des complications dans l'algorithme d'allocation (qui doit attendre que les ressources ne soient plus épinglées), et impose de faire confiance au service pour les dépingler dans un laps de temps court. Pour suivre le principe de 3.1.2.2, nous éviterons l'emploi de ce mécanisme autant que possible.

L'exception courante est le masquage des interruptions, qui est un épinglage de temps CPU pour un temps court réservé aux services de confiance. Il est également possible qu'il faille interdire la révocation de mémoire quand un transfert DMA est en cours sur cette mémoire ; dans ce cas le service en question serait de confiance et s'arrangerait pour minimiser le temps d'épinglage.

### 3.5.2 Mémoire

Nous divisons la présentation de la gestion de la mémoire selon qu'elle est propre ou prêtée. La Figure 3.11 fournit un schéma récapitulatif des types de données, avec leur types de mémoire et de la manière dont ces données sont partagées.

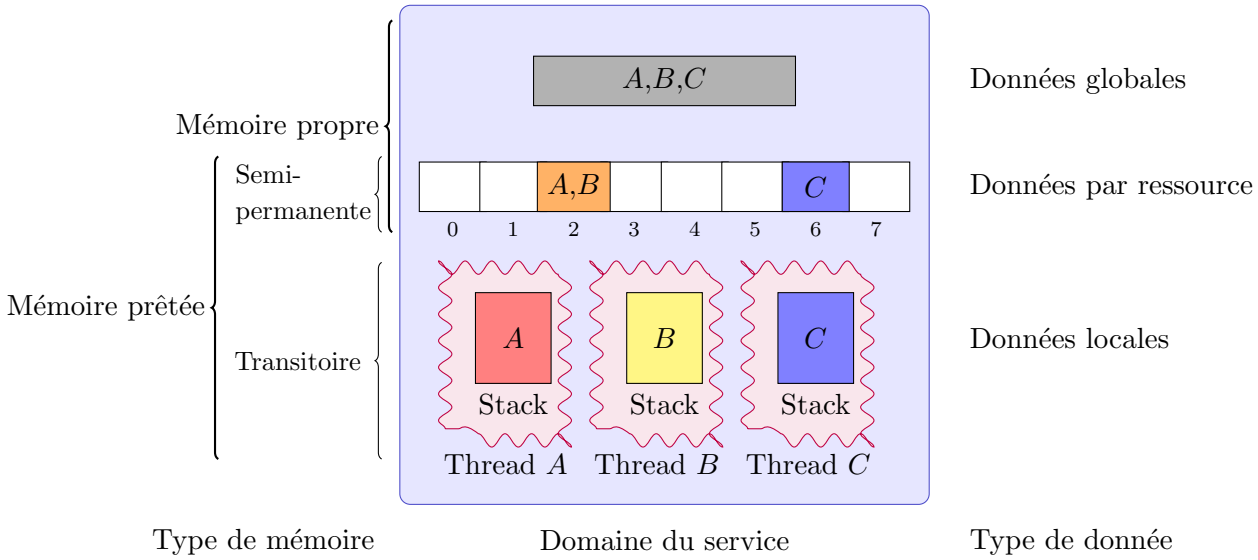
Cette différenciation des types de données et de mémoire est en pratique très simple à implémenter, et est fondamentale pour empêcher l'utilisation de mémoire propre pour répondre à des requêtes du client.

#### 3.5.2.1 Mémoire propre

Le service dispose de mémoire propre, qui sert par exemple à stocker son code et
données globales des variables globales (que nous appelons *données globales*). L'utilisation de cette mémoire doit faire l'objet de la plus grande attention pour éviter les déni de services.

**Données par client ou connexion et statelessness** La mémoire propre ne peut pas être utilisée pour stocker des informations relatives à un client ou une « connexion » : comme le nombre de clients est non borné[26], cela serait une opportunité de déni de service. Toutes ces informations sont stockées dans de la mémoire

[26]En effet, rien n'empêche un grand nombre de clients d'utiliser simultanément la même ressource : un client peut librement copier sa capacité à beaucoup d'autres clients.




Selon leur type, les données peuvent être stockées dans des mémoires d'origines différentes, et peuvent être partagées par différents clients.

- Les données *locales* (i.e. par client ou par "connection" ) ne sont ni partagées ni stockées dans le service, car elles sont placées dans de la mémoire transitoire. Elles ne sont accessibles que par le client. Comme il n'y a pas de données par client non-transitoire, le service est stateless.
- Les données *par ressources* peuvent être soit stockées dans de la mémoire propre (tableau des ressources), soit dans de la mémoire prêtée semi-permanente (essentiellement pour des gros buffers de données utilisés de manière asynchrone par les services). Elles peuvent être accédées seulement par les clients de cette ressource.
- Les données *globales* sont stockées dans de la mémoire propre au service, et peut être accédée par tous les clients.

FIG. 3.11 – Organisation de la mémoire dans un service. Chaque bloc de donnée est colorié et étiqueté en fonction des threads qui peuvent y accéder. $A$ et $B$ sont clients de la ressource 2, et $C$ est client de la ressource 6.

prêtée ; en pratique uniquement dans de la mémoire prêtée de manière transitive[27].
stateless Le service est donc *stateless* : aucune information relative à une connexion ne peut être retenue entre deux appels. Les données associées à une connexion dans les systèmes classiques seront soit placées dans le client (§ 5.3.1), soit associées à la ressource sur laquelle agit le client.

Notons que cela ne concerne que les données par connexion qui restent entre plusieurs appels : durant une connexion, l'état de la connexion est stocké dans la pile, qui est stockée dans de la mémoire prêtée de manière transitoire (§ 3.4.2).

Les file descriptors UNIX sont un exemple typique de données associées à une connexion. Dedans sont stockées différentes données ; dans Linux [BC05] on trouve les flags passé à `open`, l'offset courant, l'UID et GID du processus... Dans Anaxagoros, les seules données relatives à une connexion qui restent sont stockées dans la capacité du service (qui est immutable), et permettent seulement de pointer vers une ressource en indiquant les droits qui y sont associés.

Cette contrainte de statelessness va dans la même direction que le principe de minimisation des données que l'on verra en section 5.3.1, qui offre en fait un certain nombre d'avantages.

**Données par ressource** Contrairement aux données par connexion, il est possible de stocker entre appels les informations relatives à une ressource, dont le nombre est borné (quand le service ne fournit pas de ressources à proprement parler, comme le service réseau, on peut créer des ressources artificielles "connexion au service réseau" dont le nombre est borné, voir Section 3.5.3). Ainsi le problème de déni de service est évité : au maximum, la mémoire consommée sera de "nombre de ressources" fois "taille des données associées à chaque ressource".

Nous faisons le choix d'allouer cette mémoire associée aux ressources de manière statique. Ainsi on peut structurer les données associées à chaque ressource sous la
tableau des ressources forme d'un tableau : c'est le *tableau des ressources*. Cela permet un accès rapide à ces données, et simplifie la conception (structure de données, relations avec les politiques d'allocation) par rapport au choix d'une allocation dynamique. Afin de limiter le gâchis de mémoire, qui se produit lorsque toutes les ressources ne sont pas utilisées, la taille des entrées de ce tableau est minimisée.

**Allocation statique** La mémoire propre d'un service peut toujours être divisée comme présenté, en données "par ressource" ou "globale" (e.g. code et variables globales). La taille des données globales étant généralement fixe[28], le service n'a le plus souvent pas besoin de demander d'augmentation de la taille de sa mémoire propre (allocation mémoire entièrement statique). Cela simplifie drastiquement l'implémentation du service, et permet de facilement contrôler qu'on a pas d'allocation de mémoire (e.g. en interdisant l'utilisation de `malloc`, on s'assure qu'il n'y a pas d'utilisation implicite de FCFS sur la mémoire).

[27] On pourrait utiliser de la mémoire prêtée de manière semi-permanente pour stocker des informations relatives à une connexion. Mais cela consomme de l'espace d'adressage (qui est une ressource propre), cela ne fait que reporter le problème ailleurs.

[28] Il y a des exceptions, comme la constitution de caches.

L'allocation statique de la mémoire dans le service semble être un gâchis mémoire, en particulier pour le tableau des ressources, où l'on pourrait préférer allouer de la mémoire au fur et à mesure que différentes ressources sont utilisées. Il y a néanmoins beaucoup d'arguments en faveur de ce choix :

- ce sont les buffers de données qui occupent le plus de place mémoire, et ceux ci sont prêtés (§ 3.4.4), ainsi seule la place pour les « données de gestion », bien plus petites, est perdue. Le design des services demande par ailleurs de minimiser ces données de gestion (§ 5.3.1) pour perdre un minimum de place ;

- le service doit faire l'association entre une ressource et les données qui lui sont associées. Le tableau est la structure la plus efficace en terme de temps CPU pour faire cette association, et cela sans compter le temps passé à faire les allocations mémoires. Les autres structures implémenteraient un « tableau peu dense », par exemple avec une table de hachage. Il y a un surcoût en temps CPU pour calculer ces hash et faire les allocation dynamiques de mémoire ;

- de plus, quand quasiment toutes les ressources sont utilisées, alors cette structure plus complexe représente également un gâchis mémoire, à cause des pointeurs additionnels etc. Le tableau est la structure la plus économe en mémoire quand presque toutes les ressources sont utilisées ;

- le tableau des ressources est une structure simple. La structure statique facilite la compréhension fine du code, ainsi que la preuve de programme (par exemple, on n'a pas de problème d'aliasing). De plus, on évite quantité d'erreurs de programmation (e.g. problème de double free, utilisation de mémoire après free, etc.), ce qui rend le service plus sûr. La garantie que la mémoire ne « s'en va pas » est utile dans beaucoup de situations. Cela permet par exemple de simplifier certaines synchronisations, comme le notent également Greenwald et al. [GC96]. Cela permet aussi d'utiliser des pointeurs estampillés (§3.3.3.1) ;

- l'allocation statique de la mémoire aide à identifier toute allocation dynamique d'une autre ressource. En général, quand une ressource est allouée dynamiquement (e.g. place dans la c-list), on utilise de la mémoire pour conserver des opérations sur cette ressource (e.g. à quoi sert la capacité placée dans la c-list), qui est également allouée dynamiquement. Si la mémoire est allouée statiquement, on minimise les chances de faire une telle allocation par erreur ;

- l'allocation dynamique risque de transformer un déni de service local (sur la mémoire d'un service) en un déni de service global (sur la mémoire de tous les services). L'allocation statique permet donc de confiner les erreurs ;

- enfin, l'allocation dynamique de mémoire pose différents problèmes : l'allocation mémoire est rarement faite en temps constant et potentiellement long ; que faire s'il n'y a plus beaucoup de mémoire dans le système, sachant qu'on ne peut pas facilement attribuer ces allocations à petite granularité (en dessous de la taille d'une page) aux clients ; le service de politique mémoire fait dorénavant partie du TCB du service...

Ainsi l'allocation statique résulte en un petit gâchis mémoire (quelques kilo-octets) par service lorsque toutes les ressources ne sont pas utilisées, mais offre un grand nombre d'avantages.

#### 3.5.2.2 Mémoire prêtée

Nous avons vu comment est prêtée la mémoire, et comment est fournie l'assurance que certains mappings ne sont pas partagés. Nous expliquons seulement l'usage de cette mémoire prêtée, et la gestion de sa révocation.

**Usage de la mémoire transitoire** La mémoire passée de manière transitoire est à emplacement fixe de l'espace d'adressage dans notre implémentation, et est facilement accessible. Elle est utilisée pour les *données locales*, i.e. la pile et les arguments passés.

données locales

**Usage de la mémoire semi-permanente** Comme on utilise de la mémoire transitoire pour les données locales, et de la mémoire propre pour les données globales, la mémoire prêtée de manière semi-permanente ne sert que pour des grosses données par ressource. On l'utilise typiquement pour des données à communiquer avec le client, et que le service utilise de manière asynchrone (e.g. données envoyées ou reçues sur disque dur ou sur le réseau). Mis à part la problématique de gestion de l'espace d'adressage, qui est une denrée finie (et est abordée en section 3.5.3), la mémoire semi-permanente ne pose pas de problème de gestion.

**Gestion de la révocation** La principale difficulté concerne la tentative d'accès par le service à de la mémoire qui a été révoquée.

Si la pile est révoquée, l'exécution du service va devenir très rapidement impossible. Ce cas est donc similaire à celui de la révocation de temps CPU. C'est pourquoi, dans l'implémentation, pile et temps CPU sont tous deux liés au thread, et on ne peut révoquer la pile sans détruire le thread. De même, tous les thread-local mappings ne peuvent être révoqués sans détruire le thread. Le problème de révocation est donc réduit à la mémoire semi-permanente.

Pour la mémoire semi-permanente, on voudrait que le code puisse gérer le fait que la mémoire aie été révoquée. On va pour cela s'aider du fait qu'un emplacement mémoire (mis à part la pile) est toujours accédé de manière explicite dans le code (du moins en C). On ne peut pas tester une adresse pour savoir si la mémoire a été révoquée : ce serait lent, pénible, et sujet à race conditions. On va faire cela par un mécanisme similaire aux exceptions.

La première contrainte est de détecter que la mémoire accédée à été révoquée. Cela se fait par le mécanisme d'*autopagination* [Han99, EKJO95], i.e. en faisant en sorte que les services soient notifiés lorsqu'ils font un défaut de page (nous avons vu que les exceptions étaient paramétrables (§ 3.1.3.1) ; les services sont paramétrés pour recevoir eux même leurs défauts de page, i.e. leur défaut de page changent leur program counter à une adresse fixée). Et en s'assurant par le noyau que lorsque la mémoire est révoquée, cela crée effectivement un défaut de page. Une technique similaire est utilisée dans le système de fenêtrage d'EROS [SVNC04, § 5].

```
// set_page_fault_handler est similaire à setjmp : elle renvoie 1
// lorsqu'elle retourne la première fois, et retourne nouveau avec 0
// lorsqu'il y a eu défaut de page.
#define TRY_ACCESS_REVOKABLE_MEMORY if( set_page_fault_handler())
#define CATCH_REVOKED_MEMORY else

TRY_ACCESS_REVOKABLE_MEMORY
  {
   // Code qui accède à la mémoire prêtée
   ...
  }
CATCH_REVOKED_MEMORY
  {
   // Gestion des problèmes ; par exemple annule l'opération en cours,
   // et passe à la suivante.
   ...
  }
```

Listing 3.1 – Exemple de gestion de la révocation mémoire par exception

Pour remonter l'information qu'il y a eu un défaut de page et que la mémoire prêtée a été révoquée, on peut utiliser le mécanisme d'exception présent dans beaucoup de langages. En C, on utilisera une interface similaire au couple `setjmp/longjmp`, illustré par le Listing 3.1.

Notons que lorsque l'exécution se passe dans le thread prêté par le client, on peut ramener le cas de la révocation de mémoire à la révocation de temps CPU, en faisant simplement en sorte que lorsqu'une page fault est reçue, le thread retourne dans le client avec une erreur. Cela limite drastiquement la nécessité d'employer le mécanisme décrit ci-dessus : il faut le faire uniquement pour du code exécuté dans des threads propres au service, et donc uniquement pour des prêts de mémoire semi-permanents.

#### 3.5.2.3 Conclusion

Cette section a montré comment gérer la mémoire dans les services. La pile et les arguments sont stockés dans de la mémoire prêtée transitoires ; les gros volumes de données (buffers) par ressource dont a besoin le service sont prêtés par le client de manière permanente ; les petits volumes par ressource et les données globales sont alloués statiquement par le service.

### 3.5.3 Capacité et autres ressources

**Capacités et autres privilèges propres** Le service est contacté parce qu'il permet de réaliser des actions que ses clients ne peuvent pas faire ; aussi dispose-t-il généralement de privilèges sous la forme soit de capacités qui lui sont propres, soit d'accès direct à des périphériques (par memory-mapped IO ou ports IO). Ces

privilèges propres sont stockés dans les domaines à la manière de capacités, mais ils ne peuvent pas être copiés entre domaines.

**Allocation d'entrées dans des tables** Un problème similaire à celui de l'allocation de mémoire propre est celui de l'allocation d'entrées dans les C-lists propres et table des pages propres du service. Une entrée dans la C-list (resp. table des pages) est nécessaire pour que le client puisse prêter au service une capacité (resp. un mapping mémoire) de manière semi-permanente. Ces entrées sont en nombre fini (e.g. l'espace d'adressage est fini), et sont donc une ressource propre du service, et doivent être allouées avec soin pour éviter les DoS. Ce problème est largement évité grâce au prêt de thread et en minimisant l'utilisation du prêt de ressource semi-permanent.

Pour le résoudre, on réserve tout simplement un certain nombre d'entrées dans la table pour chaque ressource.

Parfois, il n'y a pas vraiment de ressources, mais le nombre de ces entrées limite quand même le nombre de clients simultané. C'est le cas des services d'accès au réseau et au disque, qui ne servent qu'à ordonnancer les requêtes des clients[29]. Dans ce cas, on limite le nombre de clients artificiellement en définissant une ressource "connexion au service".

Ainsi, pour le service de réseau, pour chaque ressource "connexion", on réservera une entrée dans la table des pages de premier niveau (permettant de prêter 4Mo de manière semi permanente), une entrée dans la C-list du service (pour pouvoir notifier les clients quand leurs données sont prêtes), et une entrée dans le tableau des ressources. Le nombre de connexion est limité à 1000, représentant environ le nombre d'entrées libres dans ces tables.

Notons que finalement, imposer un nombre fini de client permet aussi de faciliter la définition de politiques d'ordonnancement pour ces services.

**Révocation de capacité prêtée** Certains micronoyaux envoient des notifications à tous les clients lorsque leur capacité est révoquée (dead name notification dans Mach [Loe92, page 33]). Cela pose problème pour correctement comptabiliser le temps passé à envoyer ces notifications, et nous avons choisi de ne pas implémenter (et donc économiser) ce mécanisme.

En effet, on n'utilise les capacités qu'à certains endroits du code. Il suffit donc que le noyau renvoie une erreur au client qui utilise une capacité révoquée, pour que ce client s'en "aperçoive". Le service qui utilisera une capacité prêtée révoquée s'en apercevra, et pourra donc agir en conséquence si nécessaire.

### 3.5.4 Gestion du temps CPU

La gestion du temps CPU (et du multithreading) est très certainement la plus grande difficulté dans la réalisation de services suivant les principes énoncés section 3.1. Rappellons le principe général : lorsqu'un thread client a besoin d'un service, il y a changement de domaine et l'exécution de ce thread continue dans le service, i.e. le

[29] Le service disque sert en fait aussi à partager les secteurs du disque

temps CPU du client est utilisé pour exécuter le code du service. L'exécution de ce thread peut être préempté et reprendre dans le service, et le service est donc multithreadé.

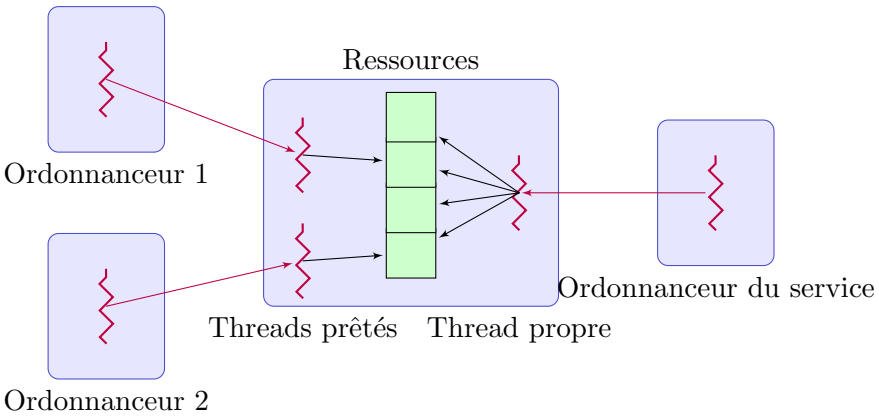



FIG. 3.12 – Threads prêtés et l'éventuel thread propre. Les threads prêtés n'accèdent qu'à la ressource sur laquelle ils font leur appel, tandis que le thread propre fait des opérations sur toutes les ressources. L'ordonnancement des thread prêtés est fait par des ordonnanceurs inconnus du service, tandis que celui du thread propre est l'ordonnanceur du service, qui est de confiance.

#### 3.5.4.1 Temps CPU propre

**Intérêt du thread propre** Bien que la majorité des traitements se fait sur du temps CPU prêté par les clients, le service peut aussi disposer de temps CPU qui lui est propre, sous la forme d'un thread propre. Son utilité est double :

- il permet d'exécuter toute action qui ne peut être faite sur le temps CPU d'un client. Par exemple, pour recevoir des paquets réseau avant que le client récepteur ait été identifié. Ou pour lancer des écritures sur le disque : comme l'ordonnancement du disque est différent de l'ordonnancement CPU, on ne peut pas faire ces requêtes pendant le temps CPU du client ;

- soit quand on a des besoins temporels précis, et qu'on doit faire confiance à son ordonnanceur pour y répondre. Par exemple, le service d'affichage demande à son ordonnanceur de lui donner du temps régulièrement pour que l'affichage ne reste pas "à moitié actualisé" longtemps.

Le thread propre exécute du code typiquement différent de celui exécuté dans les thread prêtés. Le code des thread prêtés effectue des opérations spécifique à une ressource, tandis que le thread propre agit pour toutes les ressources. Ainsi, le thread propre du service d'affichage mettra à jour le framebuffer en utilisant les données de chaque client, le thread propre du service réseau enverra les données de tous les clients sur le réseau, etc.

**Contraintes sur le thread propre** Une bonne conception devra faire en sorte que le temps passé dans le thread propre pour faire une itération sur l'ensemble des ressources soit borné. Comme le nombre de ressources est borné, cela se fait généralement en s'assurant que le temps passé par ressource est borné. Cela permet d'exécuter les threads propres de manière périodique, et ainsi de s'assurer que chaque ressource est prise en compte en un temps maximum. Ainsi, si le thread d'affichage s'exécute avec une période de 10 millisecondes, et prend au maximum 1 milliseconde pour s'exécuter, chaque client est assuré que son affichage sera actualisé dans les 20 prochaines millisecondes [30]. Ce type de propriété est important pour les systèmes temps réel.

L'utilisation de thread propre pour répondre aux requêtes ressemble fortement à la communication asynchrone que nous avons présentée section 2.3.2.1. Il y a cependant une différence essentielle : les requêtes traitées par le thread propre ont déjà été « pré-traitées » par un thread prêté. C'est ce pré-traitement qui va permettre d'assurer que le temps passé par le thread propre par ressource se fait en temps borné.

#### 3.5.4.2 Temps CPU prêté

La gestion du temps CPU prêté est de loin le problème le plus contraignant pour la conception des services. Nous dressons ici une vue d'ensemble de la complexité du problème.

**Des services multithreadés** Le client prête au service du temps CPU sous forme de thread. L'exécution d'un thread prêté peut s'arrêter et reprendre dans le service, suivant l'ordonnanceur de ce thread : c'est la *préemption*, sorte de révocation temporaire du prêt de temps CPU.

Pendant qu'un thread est préempté dans le service, d'autres clients sont exécutés qui peuvent à leur tour appeler le service. Ainsi, les exécutions de plusieurs threads différents s'entrelacent dans le service, qui est par nature *multithreadé*.

L'inconvénient est que cela demande une attention particulière pour l'accès aux données partagées. Les entrelacements entre écritures et lectures peuvent déboucher sur des données lues qui sont incohérentes, et donc des bugs, augmentations de privilège, etc.

Les avantages sont multiples. D'abord, contacter un service ne modifie pas le plan d'ordonnancement du client, ce qui est un prérequis à l'indépendance entre l'implémentation du service et la politique d'ordonnancement du client. De plus, le système d'exploitation est quasiment prêt pour le parallélisme matériel : entre le multithreadé monoprocesseur et multiprocesseur, il y a relativement peu de différences (essentiellement des problèmes de consistance mémoire, et dans l'implémentation des primitives de synchronisation).

**Des ordonnanceurs incontrôlables, voire malicieux** Pour respecter le principe d'indépendance des politiques d'ordonnancement (§ 3.1.1), les services ne requièrent

[30] Ce cas le pire se produit lorsque le thread d'affichage s'exécute au début de sa période, puis à la fin.

aucune coopération de la part des ordonnanceurs.

Le service ne peut ainsi pas empêcher le thread d'un client de s'arrêter au milieu d'une section critique protégée par un verrou, même si cela bloque l'exécution de tous les autres client pendant ce temps. Le thread peut même ne jamais reprendre l'exécution, par malice, bug, ou destruction normale du client.

Ce problème est exacerbé du fait que lorsqu'un thread est préempté ou révoqué, les données prêtées au threads ne sont plus consultables par le service. La section 5.2.2 sur les rollforward lock dans les services partagés détaille ce problème.

Les ordonnanceurs peuvent poser d'autre problèmes, comme "oublier" un thread pendant un certain temps, puis reprendre l'exécution dans le service après que le client se soit fait révoquer le droit d'utiliser ce service[31].

Ainsi, les services doivent être conçu de manière à ne faire aucune hypothèse sur le fonctionnement des ordonnanceurs : l'ordonnancement des threads prêtés doit être considéré, du point de vue du service, comme totalement indéfini.

Notons que ce n'est pas le cas des thread propres, qui peuvent être préempté à des instants indéfinis, mais pour lesquels on peut faire confiance à leur ordonnanceur pour les exécuter selon leurs besoins. Ces threads peuvent ainsi être assurés de pouvoir faire certaines actions "à temps". On peut parfois se servir de cette propriété (comme vu Section 3.5.4.1).

**Des contraintes fortes** Le service ne peut pas s'appuyer sur les services de politique d'ordonnancement, mais doit en plus leur fournir certaines garanties, nécessaires pour l'exécution de clients temps réel.

Ainsi le service ne doit pas modifier l'ordonnancement de manière imprévisible, comme cela est fait en utilisant des sleeplock. Il ne peut pas non plus masquer les interruptions ou utiliser de spinlock, car il devrait pour cela être privilégié. Dans la mesure du possible, les opérations devraient se faire en un temps borné, ce qui permet de les prendre en compte dans l'établissement de majorants d'exécution[32].

### 3.5.5 Conclusion sur la gestion des ressources dans les services

Le prêt de thread permet d'établir une communication entre le client et le service sans que ce dernier n'utilise de ressources propres. Mais le service doit continuer à veiller à ne pas utiliser de ses ressources, en particulier de mémoire, en réponse aux requêtes de clients. Nous avons vu qu'en général, cela n'était pas très difficile à gérer.

La gestion des ressources prêtées se résume essentiellement au fait que l'ordonnancement des threads prêtés n'est pas contrôlable, dû au principe d'indépendance des politiques d'ordonnancement. Il va s'agir essentiellement problèmes de concurrence et de cohérence des données, avec des contraintes particulières sur l'utilisation des verrous (ni sleeplock, ni masquage des interruptions).

[31] La protection contre cette attaque de type TOCCTOU est faite en re-vérifiant que la capacité est valide quand le thread revient dans le service

[32] On peut avoir des opérations qui prennent un temps non-borné, L'ouverture d'un fichier d'un système de fichier requiert ainsi un parcours d'arbre. Mais les opérations "récurrentes", qui ne peuvent pas se faire à l'initialisation, devraient se faire en temps constant.

Il y a deux axes qui permettent de résoudre le problème ; le premier consiste à examiner et concevoir différentes primitives de synchronisations à la lumière du problème particulier d'ordonnancement non contrôlé. Mais on peut aussi trouver des moyens de concevoir le service dans le but de fortement réduire le besoin d'utiliser ces primitives de synchronisation. Ces deux axes sont successivement étudiés dans les deux chapitres suivants.

## 3.6 Conclusion

**Résumé** Dans cette section, nous avons proposé un ensemble de principes pour l'intégration d'applications hétérogènes en niveau de sécurité. L'idée première est que pour exécuter des tâches critiques, il faut

1. connaître leurs besoins en ressource ;
2. définir une politique d'allocation adaptée qui garantisse que ces besoins seront remplis (indépendemment de ce qu'est la politique) ;
3. exécuter les tâches, avec des mesures pour la sûreté de fonctionnement.

Les principes que nous avons défini permettent l'obtention d'un support d'exécution qui aide à implémenter tous ces points.

Le reste du chapitre explique comment concevoir le système selon ces principes ; structuration et décomposition du système en domaines de protection, contrôle d'accès pour les communications entre les domaines de protection, prêt de ressource, et gestion des ressources dans les services. Les deux premiers points concernent davantage la sécurité-intégrité et sécurité-confidentialité, tandis que les deux derniers points sont plus précisément tournés vers le principe nouveau d'indépendance des politiques d'allocations.

Ce chapitre n'a cependant abordé que la structure générale et la conception du noyau. On obtient finalement un système où les politiques d'allocations sont beaucoup plus simples et compréhensibles, mais au prix de contraintes supplémentaires dans le développement des services partagés. Il reste donc à voir comment concevoir des services pour respecter ces contraintes ; cela est présenté au chapitre 5. Une contrainte particulière est le respect de l'indépendance des politiques d'ordonnancement, qui va poser des problèmes de synchronisation abordés au chapitre suivant.

**Remarques** Une propriété très intéressante de l'OS que nous avons implémenté est qu'il n'impose quasiment aucun choix. Nous nous sommes toujours arrangé pour permettre une fonctionnalité (comme la sécurité ou l'allocation déterministe), mais sans l'imposer. C'est cela qui permet d'être assez flexible pour pouvoir exécuter des tâches critiques et des tâches non critiques.

Cette propriété est obtenue principalement par le respect très strict du principe de minimisation des mécanismes communs (§ 2.2.2.6). Ce principe est à la fois un principe de sécurité, mais également un principe pour la flexibilité. Ainsi, la gestion des ressources par l'OS est minimaliste (il n'impose que le mécanisme, la politique est remplaçable). Le contrôle d'accès est minimaliste (seule la communication avec

le service est contrôlée par le noyau ; le contrôle d'accès à l'objet est remplaçable par le service), etc..

Cela permet une réutilisation maximale du système, puisqu'à la fois les politiques d'allocation et de sécurité sont libres. Le système n'impose finalement que les APIs d'utilisation.

Chapitre 4

# Concurrence et synchronisation

Le chapitre précédent a montré que non seulement les services dans Anaxagoros sont multithreadés, mais on leur impose des conditions supplémentaires : mémoire prêtée révocable, ordonnancement indépendant du service, consommation de mémoire propre bornée, clients ayant des contraintes temps réel dur... Ces conditions rendent la synchronisation dans les services particulièrement difficile, voire impossible, avec les mécanismes usuels. En particulier, l'indépendance des politiques d'ordonnancement interdit au service de mettre des threads en sommeil ; de plus les services n'étant pas de confiance[1], il leur est également interdit de masquer les interruptions ; ces deux mécanismes étant les primitives généralement utilisées pour implémenter des mécanismes de synchronisation.

Dans ce chapitre, nous analysons pourquoi les techniques usuelles de résolution des problèmes liés à la concurrence ne peuvent pas s'appliquer à nos service. Nous en déduisons un ensemble de mécanismes de synchronisation aux propriétés satisfaisantes (non-bloquant, ne nécessitant pas d'allocation dynamique de mémoire, overhead faible...). Ces mécanismes ne sont pas réservé aux services, mais bénéficient à tous les programmes. Ils sont particulièrement intéressants pour l'écriture de programmes temps réel parallèles.

Dans le chapitre suivant, nous développerons de manière plus détaillée la résolution des problèmes de concurrence dans les services d'Anaxagoros, en s'appuyant sur les mécanismes développés ici.

**Contributions** Les principales contributions de ce chapitre sont les suivantes :

- un formalisme simple pour décrire exactement les problèmes de cohérence dans un programme parallèle (§ 4.1.3.2) ;

[1] Dans certains systèmes, il est possible de supposer que les services sont de confiance, auquel cas le masquage des interruptions est envisageable. Cette solution implique qu'un bug dans un service impacte l'ensemble du système, ce qui réduit la sécurité. De plus, cette solution est particulièrement pénible si on souhaite installer des services uniquement à l'usage de tâches non-critiques (e.g. paravirtualisation d'un système Linux), puisque ce service partagé devient critique alors qu'il pourrait ne pas l'être.

- la catégorisation des problèmes de la programmation parallèle en quatre types (§ 4.1.1.3) ;
- une catégorisation de mécanismes nouveaux ou existants qui offrent des garanties complètes ou partielles contre les problèmes ci-dessus ; et de la manière de les utiliser ;
- un mécanisme permettant la sauvegarde et la restauration du contexte processeur dans son intégralité en espace utilisateur, la préemption programmable ;
- un nouveau type de lock non-bloquant, le recoverable lock, implémenté efficacement grâce à la préemption programmable ; et les autres verrous qu'il permet d'implémenter, le rollback lock et le rollforward lock.

## 4.1 Problématique de la concurrence

### 4.1.1 Concurrence dans les programmes

Avant d'utiliser des primitives de synchronisation, il faut connaître les problèmes qui se posent en situation de concurrence. Et pour cela, il faut savoir ce que doit faire le programme, i.e. sa *spécification*. Cette section précise ces problèmes.

Ces relations entre problème posé et synchronisations à effectuer sont importantes, car elle permettent d'utiliser des primitives légères qui correspondent au mieux à la situation : en particulier passent mieux à l'échelle et prennent moins de place. La petite taille des services dans Anaxagoros facilite cette démarche.

#### 4.1.1.1 Propriétés et spécification

Un programme infini, se caractérise par la suite des observations faites sur ce programme. Pour un service de système d'exploitation, ce qui est observable l'est par le hardware (I/O, état de la mémoire, registres spéciaux) ou par les clients (mémoire partagée service/client et valeurs de retour des appels de services).

Concrètement, cela signifie que ce qu'on peut spécifier d'un service (ou d'un programme en général) se met sous la forme :

- de séquencements valides de certaines actions du service, spécifiés sous la forme d'automates ;
- de prédicats vrais sur les données du service, qui peuvent dépendre de l'état actuel de l'automate, et qu'on appelle *invariant* ;

invariant

- de propriétés sur les valeurs envoyées au hardware ou aux programmes (e.g. des postrequis sur les valeurs retournées aux appels de service).

En particulier les propriétés de sécurité, que le service a pour rôle d'assurer (propriétés « négatives » de Rushby [Rus89]) se trouvent sous cette forme.

Pour remplir ces propriétés sur les spécifications, le programme implémente des propriétés supplémentaires, qui apparaissent notamment lors de la preuve de

programme. L'annexe A sur la preuve du système de capacité fournit un parfait exemple de spécification de problèmes de sécurité faite par un automate et d'invariants à prouver pour son implémentation. D'autres méthodes comme Dingo/Termite [RCKH09, RCK+09] montrent que les drivers (qui sont des services) se spécifient bien sous la forme d'automate.

Notons que ces propriétés sur le séquencement des actions sont importantes pour d'autres programmes, i.e. pour les programmes de contrôle-commande. Le respect des invariants est également important pour tout programme.

#### 4.1.1.2 Impact du parallélisme

**Définitions : concurrence, cohérence, séquencement** Quand à un moment donné, tous les invariants sur un ensemble de données sont respectés, on dit que le programme respecte la *cohérence* pour cet ensemble, et cet ensemble est cohérent. cohérence

Quand un programme fait ses actions dans un ordre autorisé[2], nous disons qu'il est respecte le *séquencement*, ou qu'il est correctement séquencé. Cette propriété est notamment importante quand le programme fait des entrées/sorties. séquencement

Le respect du séquencement et de la cohérence forment l'ensemble des problèmes de *concurrence* qu'on va retrouver ; les problèmes de séquencement étant les problèmes temporels, et les problèmes de cohérence les problèmes spatiaux. concurrence

Il est facile pour un programme séquentiel de respecter ces propriétés. Le programme est structuré pour suivre les actions comme requis par l'automate. Le programme modifie la mémoire pour passer d'un état cohérent à un autre. Notons que comme le matériel ne permet de modifier la mémoire que mot à mot, il est courant pour un programme séquentiel de passer temporairement par des états incohérents.

**Cas du multithreading et linearizabilité** La difficulté du multithreading est de maintenir ces deux propriétés. L'entrelacement des traitements fait qu'on ne peut plus savoir dans quel état on est juste par la position dans le code ; il faut prendre des mesures particulières pour rester correctement séquencé. De plus, il prendre des mesures pour éviter que des threads puissent lire des données dans un état incohérent.

Ce problème des états mémoires incohérents est le plus souvent traité. Généralement, on fait cela en modifiant le code des opérations qui modifient ces états mémoires pour les rendre *atomiques*. Mais ce n'est pas parce qu'on fait des mises à jour de manière atomique qu'on résout tous les problèmes dûs au parallélisme ; il peut rester des problèmes de séquencement. C'est à dire qu'en général, les concepteurs se concentrent sur la cohérence des états mémoires du système à un instant donné, mais plus rarement à la séquence des comportements autorisé.

Le problème de séquencement correct est pourtant aussi important. Les services partagés sont là pour assurer certaines propriétés, et en particulier que certaines actions sont effectuées dans un ordre correct. La propriété permettant d'effectuer

[2] si toutes les actions sont représentées par une lettre dans un alphabet, les actions autorisées sont un langage pour cet alphabet. Généralement, ce langage est régulier, et la suite des actions autorisées peut être représentée sous la forme d'un automate.

cette garantie en situation de concurrence est la linéarizabilité [HW90], qui ne se réduit pas à l'atomicité des écriture de données.

La linéarizabilité modélise les accès à un objet en ayant chaque thread qui peut envoyer des couples (requête/réponse). Un historique est une séquence de requêtes et de réponses ; en situation de concurrence, ces couples peuvent être entremêlés. Un historique est linéarizable si on peut le réordonner de manière à obtenir un historique séquentiel (où chaque requête est immédiatement suivi de sa réponse) correct (selon la définition de l'objet), et tel que si une requête suivait une réponse dans l'historique original, cet ordre est préservé dans l'historique séquentiel réordonné. Un objet est linéarizable si tous les historiques possibles sont linéarizables.

Cette notion est donc relativement fine, et permet certaines optimisations, dont on peut profiter (voir en particulier la section A.1.3 qui donne un exemple, mais aussi la section 5.2.1.1).

Une autre notion est la serializabilité [FR82, FR85], plus généralement utilisée dans les bases de données. Cette notion est moins contraignante que la linéarizabilité, car elle n'impose pas de contrainte d'ordre sur les différentes requêtes et réponses. Il est possible que cette notion plus faible soit parfois mieux adaptée.

#### 4.1.1.3 Problèmes en programmation parallèle

Nous définissons quatre catégories de problèmes que l'on peut rencontrer lors de l'écriture de programme parallèle :

**L'observation des états intermédiaires** Le thread, en effectuant la mise à jour de données d'un état cohérent vers un autre, passe par des états intermédiaires observables par les autres threads. En particulier, comme vu précédemment, ces états peuvent être incohérents.

**La non-terminaison** Le thread peut commencer d'exécuter une opération sans jamais la terminer, parce qu'il est supprimé ou plus jamais exécuté. Par exemple, pour déplacer un objet d'une case à une autre, on peut d'abord retirer l'objet de la première case, puis le mettre dans la deuxième ; si l'exécution stoppe entre ces opérations, l'objet est perdu à jamais. Un autre exemple est d'incrémenter un compteur du nombre de ressources possédées sans jamais exécuter l'augmentation de ce nombre de ressources. Les programmes multithreads habituels ne se posent habituellement pas la question de la non-terminaison, car ils contrôlent quand leurs threads sont définitivement arrêtés ; ce n'est pas le cas en programmation système.

**Les opérations retardées** Ces problèmes concernent des opérations qui s'arrêtent et reprennent à un moment inattendu ou incontrôlé. Par exemple, l'acquisition d'un pointeur par un thread sur de la mémoire qui a été entre-temps libérée. Ces problèmes peuvent causer des incohérences et des séquencements incorrects.

Les *bugs TOCTTOU* (time of check to time of use) [BD96] en sont un exemple particulier. Ce type de faille de sécurité se produit lorsqu'un thread $A$ fait une vérification sur une condition avant de faire une opération, qu'un autre thread $B$

modifie cette condition, et qu'enfin le thread $A$ effectue l'opération alors que la condition ne tient plus.

**Les opérations simultanées** Enfin, des problèmes surviennent pour des opérations simultanées, bien qu'on connaisse les instants où elles sont exécutées.

Par exemple beaucoup d'algorithmes fonctionnent seulement s'il y a un seul thread écrivain, et plusieurs threads lecteurs ; mais pas s'il y a plusieurs écrivains simultanés. Ainsi l'algorithme de concurrent reading and writing of clocks de Lamport [Lam90] permet d'augmenter la valeur d'un compteur sur plusieurs mots. L'algorithme suivant :

```
clock_t global_clock ;

void add_to_clock( unsigned int val) {
    clock_t local_clock = read_clock( global_clock) ;
    write_clock( &global_clock, local_clock + val) ;
}
```

où `read_clock` et `write_clock` sont ceux décrits par Lamport. Cet algorithme marche parfaitement bien lorsqu'un seul thread l'exécute (même s'il y a plusieurs autres lecteurs), mais peut être incorrect s'il y en a plusieurs : pour différentes valeurs de `val`, on peut se retrouver à remplacer `global_clock` par une valeur plus petite.

**Autres classifications** Harris et Fraser [HF05] classifient les problèmes du parallélisme en seulement deux catégories, les mises à jour partielles (correspondant à nos deux premières catégories) et les opérations retardées (correspondant à nos deux dernières). C'est naturel car ils inventent une primitive de synchronisation qui s'occupe de l'une, mais pas de l'autre.

Notre travail est une poursuite du leur au sens où nous avons analysé plusieurs autres primitives de synchronisation, qui résolvent des problèmes de parallélisme qui ne rentrent pas dans les catégories définies par Harris et Fraser, d'où le besoin de créer ces nouvelles catégories.

Cette liste ne se prétend pas exhaustive, et on pourrait bien découvrir par la suite d'autres primitives de synchronisation qui résolvent des catégories de problèmes non répertoriées ici.

#### 4.1.1.4 Problèmes liés au parallélisme matériel

Les problèmes de parallélisme existent déjà sur un monoprocesseur qui peut exécuter plusieurs threads de manière préemptive. Mais des problèmes additionnels se posent en multiprocesseur.

**Cohérence des caches** Le premier est lié à la cohérence des caches. Il faut s'assurer que chaque processeur a la même vue de la mémoire : à cause des caches, différents processeurs pourraient avoir des valeurs différentes pour une même adresse. Généralement ces problèmes sont résolu automatiquement par le matériel [AB86]

(e.g. protocole MESI sur les processeurs Intel Pentium [Int09]), le seul impact pour le programmeur est la question de performance (il doit éviter les invalidations en "ping-pong" des différents caches)[3].

**Consistance des caches** L'autre vient du fait que du à la présence de cache (et au réordonnancement des instructions), l'ordre des accès à la mémoire par le processeur peut être différent de ceux spécifiés dans le programme ; c'est le problème de consistance des caches [Lam79]. Pour le résoudre, les processeurs offrent des instructions particulières, appelées *barrières mémoires*, qui permettent au programmeur de contraindre l'ordre des lectures et écritures faites par le processeur.

Ce problème ajoute donc une contrainte pour le programmeur, même si l'addition de ces barrières n'est souvent pas très difficile. Notons que le programmeur doit souvent également faire attention à ce que le compilateur ne réordonnance pas les opérations ; il est possible de définir des macros qui ajoutent les deux types de barrières (processeur et compilateur) simultanément.

### 4.1.2 Techniques usuelles de résolution

Nous analysons maintenant différentes techniques usuelles de résolution des problèmes de concurrence, et analysons si elles sont compatibles avec le principe d'indépendance des politiques d'allocation (§ 3.1.1).

#### 4.1.2.1 Les mécanismes bloquants

Les techniques de résolution des problème de concurrence peuvent être cataloguées
bloquant en deux : bloquantes ou non-bloquantes [Fra04]. Un mécanisme est *bloquant* lorsque l'arrêt d'un thread peut retarder ou indéfiniment empêcher l'exécution d'un autre ;
non-bloquant *non-bloquant* lorsque ce n'est pas le cas[4]. Les techniques bloquantes, les plus usitées, posent problème dans Anaxagoros car elles ne respectent pas le principe d'indépendance de l'ordonnancement. Mais c'est en les analysant en détail que nous trouvons les racines des problèmes que nous résolvons par la suite.

section critique **Section critique** L'utilisation de *section critique* permet de répondre aux 4 problèmes du même coup. Il s'agit de synchroniser les différents threads afin qu'au plus un d'entre eux soit dans une certaine portion de code, qui est la section critique. Ainsi, cette portion de code s'exécute de manière séquentielle.

On utilise souvent une variable partagée pour synchroniser ces différents accès,
verrou appelé *verrou*, ou *mutex lock* (verrou d'exclusion mutuelle).
mutex lock Il y a plusieurs façons d'implémenter une section critique, chacune ayant ses avantages et ses défauts.

spinlock **Spinlock** La manière la plus simple de créer une section critique est le *spinlock*,

[3]Une exception courante est le cache des entrées de la table des pages (TLB) : le Pentium, par exemple, n'invalide pas les entrées automatiquement, ce qui requiert une synchronisation entre les processeurs, appelée TLB shootdown.

[4]Notons que non-bloquant était un synonyme de lock-free dans des papiers plus anciens, mais nous utilisons le vocabulaire actuel [Fra04, p. 9]

ou verrou par attente active. Le thread qui rentre dans la section critique prend le verrou en modifiant la valeur du spinlock, et le libère à la fin de la section ; les autres bouclent en lisant la variable et attendant que le verrou se libère. Il y a quantité de possibilités d'implémentation de ce mécanisme, comme le « Bakery algorithm » de Lamport [Lam74] ou l'algorithme de Peterson [Pet81] ; les implémentations modernes utilisent des instructions processeurs comme Compare-and-swap qui permettent une implémentation simple et efficace.

Le problème principal des sections critiques par attente active est la préemption pendant la section critique. Les autres threads sont alors bloqués jusqu'à ce que le thread reprenne la main pour terminer cette section critique, donc pour un temps long, voire infini. De plus, comme les threads sont en attente active, le temps où ils sont bloqués est perdu.

Les deux approches classiques pour résoudre ce problème sont soit d'empêcher les préemptions, soit d'éviter de perdre le temps où le thread est bloqué en faisant autre chose (appel à l'ordonnanceur).

**Empêchement des préemptions** Une section de code pour laquelle on empêche la préemption de survenir est appelée *section non préemptible*. Lorsque le code exécuté pour la prise d'un spinlock et sa section critique sont exécutés à l'intérieur d'une section non préemptible, le thread qui exécute la section critique ne peut plus être préempté, et les autres threads ne seront donc bloqués que pour un temps borné (sous réserve qu'il n'y a pas de défaillance lors de l'exécution de la section critique).

section non préemptible

Plus généralement, une section non préemptible permet en un certain sens de transformer des algorithmes bloquants en algorithmes non-bloquants.

Notons qu'en monoprocesseur, une section non préemptible est toujours une section critique, puisque le thread est garanti d'être seul à utiliser le processeur. De plus, *un spinlock est toujours inutile en monoprocesseur* : lorsqu'un thread attend pour un spinlock, donc monopolise le processeur, le spinlock ne peut pas se libérer.

Il y a deux moyens pour empêcher les préemptions, *a priori* ou *a posteriori*.

**Masquage des interruptions** On empêche les préemptions *a priori* en masquant les interruptions. Ce mécanisme a un overhead faible, mais demande de faire confiance au programme qui l'emploie puisque celui-ci peut bloquer le processeur indéfiniment[5]. Dans Anaxagoros, cette technique est réservée au (petit) noyau.

**Sursis d'exécution** L'autre moyen est *a posteriori* : si une préemption doit survenir, on l'enregistre mais on accorde au programme un délai supplémentaire. Cela peut être implémenté sous différentes formes : les exonoyaux [EKJO95, § 5.1.1] accordent un sursis de manière systématique. Psyche [MSLM91, § 3.2] utilise une notification d'une préemption imminente : le temps entre cette notification et la préemption peut être vu comme un sursis d'exécution. Symunix [ELS88, § 3.2] utilise

[5]On pourrait éventuellement rajouter un timeout, mais cela est contraire à notre refus de coder du temps en dur (§ 3.1.4.3).

une variable par thread permettant de signaler à l'ordonnanceur qu'il a besoin d'un sursis ; ce sursis est accordé lorsque cette variable est à 1.

Le problème qui se pose est celui de la durée du sursis d'exécution.

S'il est fini, on prend le risque que la section critique ne soit pas terminée quand le thread est réellement préempté. S'il est long, ce risque diminue, mais le temps de préemption augmente, ainsi que l'interférence avec les décisions d'ordonnancement, ce qui pose problème pour les programmes temps réel. De plus on ne respecte pas notre refus de coder du temps en dur (§ 3.1.4.3). S'il est infini, cela demande de la confiance comme pour le masquage des interruptions.

Dans Anaxagoros, tout le noyau s'exécute dans une section non préemptible, implémentée à l'aide d'un sursis d'exécution infini ; on fait confiance au code du noyau pour vérifier régulièrement qu'il n'y a pas eu préemption. Cette implémentation permet à la routine d'interruption de continuer à s'exécuter (e.g. pour compter le temps passé dans le noyau, ou implémenter un watchdog), et permet une implémentation efficace des points de préemption explicites[6] (il suffit de vérifier une variable).

**Sleeplock** Pour éviter de perdre du temps à attendre que la section critique soit libérée comme pour le spinlock, il est commun que les threads bloqués effectuent un
sleeplock appel à l'ordonnanceur (e.g. par l'appel système `sleep()`, d'où le nom de *sleeplock*). En particulier, si le thread dans la section critique était préempté, l'ordonnanceur pourra ordonnancer ce thread là, et ainsi libérer le verrou.

Dans le cas général, l'utilisation de sleeplock perturbe l'ordonnancement de manière imprévisible et empêche l'ordonnancement déterministe. Cela va à l'encontre du principe d'indépendance des politiques (§3.1.1.1).

Cependant, certains systèmes permettent de "diriger" l'ordonnanceur afin qu'il ordonnance spécifiquement le thread qui détient le verrou, le temps qu'il le libère. Dans les systèmes par priorité, on appelle cela héritage de priorité (e.g. [SWH05]), mais ce système peut être implémenté de manière générale, comme l'ont montré Ford et Susarla[FS96]. Ainsi, l'attente pour le verrou est borné à la longueur de la section critique [7]. Ce mécanisme semble donc acceptable pour des sections critiques courtes, et des verrous non récursifs.

Mais même ainsi, il y a des désavantages : il y a perturbation de la décision d'ordonnancement ; lorsqu'un processus peut détenir de multiples lock, l'ordonnancement peut devenir incompréhensible (en plus de pouvoir souffrir de deadlocks) ; enfin l'overhead de ce mécanisme est haut : un appel à l'ordonnanceur pour commuter vers le thread qui détient la section critique, et un appel pour en revenir.

Dans Anaxagoros, nous avons décidé d'éviter ce mécanisme au maximum (il n'est actuellement pas utilisé, mais envisagé en section 4.5). Si nécessaire, nous l'implémenterons, et l'utiliseront en prenant soin de n'utiliser qu'un seul lock simultanément, et pour des sections critiques très courtes afin que la perturbation de l'ordonnancement ne soit pas remarquable.

[6] [FHL+99] utilise également des points de préemption explicites.
[7] en monoprocesseur

**Autres problèmes des section critiques** Les sections critiques ont également d'autres problèmes. Lorsqu'utilisées pour des sections de code longs, elles restreignent le parallélisme et perturbent l'exécution des threads bloqués. Il faut pour contrer cela utiliser des sections critiques courtes, ce qu'on appelle *verrouillage à grain fin*. Mais cela augmente l'overhead pour entrer et sortir de la section critique ; de plus la multiplication des verrous augmente le risque d'*interblocage*. Dans Anaxagoros, on limite toutes les sections critiques à des séquences d'instruction très courtes pour ne pas perturber l'exécution des autres tâches. De plus, on interdit la prise de plusieurs verrous simultanément pour empêcher les interblocages. Cela ne nous empêche pas d'être scalable.

**Problème de contention et famine** Le fait de bloquer en attendant l'exécution de la fin de la section critique (quelque soit le mécanisme) génère des délais dans l'exécution et des possibilités de famine, que l'on peut analyser théoriquement.

S'il y a $M$ threads en compétition pour rentrer dans une section critique dont l'exécution dure au plus $t$, alors le temps d'attente maximal pour un thread pour rentrer dans la section critique est de $(M-1)*t$. Il y a possibilité de famine si un thread a besoin de moins de $(M-1)*t$ pour être à nouveau en attente du verrou (on suppose que le thread qui obtient le verrou est choisi aléatoirement). Dans ce cas, certains threads peuvent ne jamais obtenir ce verrou. Pour éviter la famine, il faut minimiser le temps passé dans les sections critiques.

Notons que dans une section non préemptible ou pour un algorithme lock-free, il y a au plus "nombre de processeur" threads simultanément en train d'accéder au verrou, donc un nombre petit.

**Autres mécanismes bloquants** Il y a d'autres mécanismes bloquants, donc inutilisables dans les services d'Anaxagoros. On peut brièvement citer :

- le *readers-writer lock*, qui est simplement une version optimisée du mutex lock classique. Il permet à plusieurs lecteurs un accès concurrent, mais souffre des mêmes problèmes. On considérera donc ce verrou comme analogue au mutex lock ;
- les *sémaphores* [Dij68], qui déclenchent par nature un appel à l'ordonnanceur, ce qui est contraire à notre but d'indépendance de la politique d'ordonnancement (principe 3.1.1.1) ;
- les *compteurs de version de Lamport*[Lam77]. Ce mécanisme permet de synchroniser un écrivain et plusieurs lecteurs (les écrivains peuvent être synchronisés autrement). On utilise deux compteurs de versions $v_1$ et $v_2$, qui sont modifié en début et en fin d'écriture ; le lecteur recommence à lire tant que $v_1 \neq v_2$, et recommence donc tant que l'écrivain est préempté. Il s'agit donc d'une synchronisation sans verrou, mais bloquante. Le problème est évité si l'écrivain ne peut pas être préempté vis à vis du lecteur (e.g. s'exécute dans une section non préemptible, ou est de plus haute priorité quand l'ordonnancement est à priorité fixe).

#### 4.1.2.2 La programmation non-bloquante

La technique générale pour obtenir des algorithmes non-bloquants est d'agencer son code de façon à ce qu'il soit fonctionnel quelque soient les entrelacements des différents threads.

Un exposé complet sur l'état de l'art des différentes techniques de programmation non-bloquante sortirait du cadre de cette thèse ; nous nous contenterons de pointer vers l'excellent état de l'art de Fraser [Fra04] et d'expliquer les raisons pour lesquelles ces techniques ne peuvent pas s'appliquer dans les services d'Anaxagoros. Concrètement, les problèmes qui se posent sont le fait de recommencer l'exécution, qui introduit une interférence dans l'exécution des tâches, et l'allocation dynamique de mémoire, qui peut permettre des dénis de ressource sur la mémoire.

**Typologie** Il existe trois catégories d'algorithmes non-bloquants :

- un algorithme *wait-free* garantit à tout moment que tous les threads présents finiront leur opération dans un temps maximum ;
- un algorithme *lock-free* garantit à tout moment qu'un thread parmi tous les threads présents finira son opération en un temps maximum (et pas simplement qu'il n'y a pas de lock) ;
- un algorithme *obstruction-free* garantit à tout moment qu'un thread terminera son opération en un temps maximum, s'il est seul.

Les algorithmes lock-free et obstruction-free demandent des précautions pour un usage pour le temps réel, car le temps d'exécution des algorithmes doit être borné. Mais en monoprocesseur, l'emploi de ces primitives ne pose pas de problème pourvu que les threads soient ordonnancés pour un quantum de temps suffisamment long (supérieur à la durée d'exécution de l'algorithme), car un thread est toujours tout seul à s'exécuter.

En multiprocesseur, les algorithmes obstruction-free peuvent souffrir du problème du livelock, et ne jamais terminer. Il existe des solutions pour éviter ce problème, comme d'attendre pendant un temps probabiliste, mais nous ne souhaitons pas employer ces solutions pour des programmes temps réel critiques. Les algorithmes lock-free souffrent de la contention (§ 4.1.2.1) et éventuellement de famine. Ce problème n'est pas remarquable si on boucle sur des sections courtes et qu'on utilise ces primitives de manière "peu fréquente", ce qui est généralement le cas donc ces algorithmes peuvent en général être employés.

En résumé, la programmation non-bloquante semble bien adaptée pour résoudre les problèmes de synchronisation dans les services d'Anaxagoros.

**Problème de l'allocation mémoire** Malheureusement, une bonne partie des algorithmes non-bloquants demande une allocation mémoire dynamique.

Beaucoup fonctionnent sur ce principe : « alloue de la mémoire, accessible seulement au thread, écrit les nouvelles valeurs dedans, et remplace atomiquement le pointeur sur ces valeurs ». Dans les services partagés, cela demande de l'allocation

de mémoire propre par un client, et est donc un déni de ressource mémoire potentiel (le nombre de clients étant non borné). Entre autre algorithmes basés sur ce principe, on peut citer les constructions universelles de Herlihy [Her90, Her93], ou la STM de Fraser et Harris [Fra04].

L'autre utilisation de mémoire allouée dynamiquement vient des lecteurs simultanés. Quand une nouvelle version est installée, il peut y avoir des threads qui lisent en parallèle l'ancienne version. Souvent, on ne récupère pas la mémoire tant qu'il en reste un lecteur, ce qui occasionne aussi une possibilité de déni de service. Entre autres algorithmes qui sont basés sur ce principe, il y a aussi les constructions universelles de Herlihy [Her90, Her93], et le mécanisme RCU utilisé dans le noyau Linux [BC05, p. 207]. Ce problème est plus difficile à résoudre, mais il existe également des solutions (par exemple en notifiant les lecteurs de ne plus utiliser la donnée, comme le fait notre schéma de synchronisation use/destroy 5.2.4).

Ces problèmes empêche la réutilisation directe de beaucoup d'algorithmes existants, dès qu'ils demandent une allocation dynamique de la mémoire.

**Problème spécifique aux systèmes d'exploitation** Quand on écrit un service de système d'exploitation, on doit communiquer avec le matériel qui choisit ses structures de données (e.g. format des tables de pages). Souvent la programmation non-bloquante demande à réorganiser ses données afin de pouvoir faire une mise à jour atomique, mais ce n'est souvent pas possible. Entre autre, l'utilisation de ports IO (communication avec des instruction particulière) ou de buffers à des emplacement mémoires fixes restreignent la possibilité d'utiliser les algorithmes habituels.

Peu de systèmes d'exploitations ont été implémenté de manière non bloquante. Cache [GC96] et Synthesis [Mas92] utilisent tout deux l'instruction DCAS, qui était présente sur le processeur 68k mais ne l'est plus dans les processeurs actuels. OASIS est non-bloquant, mais la majorité des ressources sont allouées statiquement.

**Exception notables** Il y a des exceptions notables, pour lesquels on peut faire de la programmation parallèle sans verrou ni allocation dynamique. Entre autres choses que nous avons utilisées, on trouve le "single-word protocol" de Herlihy [Her90], la lecture et écriture concurrente de Lamport [Lam90], et une modification de l'algorithme pour les listes de Harris [Har01] pour l'utiliser comme pile pour pouvoir faire de l'allocation FCFS. Enfin, avec un peu d'astuce on peut facilement trouver des algorithmes ad-hoc (e.g. annexe A.1.3).

#### 4.1.2.3 Un chemin intermédiaire

En résumé, les mécanismes usuels pour la synchronisation dans les programmes parallèles demandent souvent à choisir entre plusieurs maux : les algorithmes bloquants posent des problèmes lorsqu'ils sont préemptés dans leur section critique, ce qui demande soit de faire confiance au service soit qu'ils puissent modifier l'ordonnancement. Ils restreignent aussi souvent inutilement le parallélisme. Les algorithmes non bloquants réclament souvent une allocation dynamique de mémoire, ce qui conduit très facilement à un déni de ressource mémoire.

À cause de ces problèmes, nous avons décidé d'explorer également un chemin intermédiaire : i.e. de se prémunir contre certains des problèmes liés à la programmation parallèle par le code, et des autres par l'emploi de primitives de synchronisation. Notre but est de fournir des mécanismes pour permettre des synchronisations qui ne demandent pas d'allocation dynamique, qui soient non-bloquantes, et qui permette au programmeur d'écrire facilement du code parallèle efficace.

Cette voie a été ouverte notamment par le mécanisme de revocable lock de Harris et Fraser [HF05] ; nous avons mené une recherche plus exhaustive sur cette voie intermédiaire.

### 4.1.3 Respect continu des invariants de cohérence

#### 4.1.3.1 Introduction

On a vu qu'un des problèmes de concurrence venait du fait que les données pouvaient passer par des états incohérents. Nous définissons ici formellement cette notion de cohérence des données, et analysons comment la cohérence peut être continuellement respectée (dans le cas où il n'y a qu'un seul écrivain).

Il y a plusieurs applications à cela. Le respect continu d'invariants est souvent une condition nécessaire[8] pour assurer la cohérence lorsqu'il y a écritures et lectures de même données en parallèle. Ainsi, même si cette technique demande de sérialiser les écritures (par exemple avec un lock classique), les lectures peuvent être faites en parallèles.

Une autre application est pour l'utilisation des revocable lock, vu plus tard. La terminaison de l'exécution de sections critiques couvertes par ces locks n'est pas garantie ; donc ils doivent laisser la mémoire dans un état cohérent.

Nous avons utilisé extensivement ces propriétés pour obtenir le système de mémoire virtuelle parallèle ; dans ce cas les « lecteurs » sont les processeurs directement lorsqu'ils accèdent aux tables des pages.

#### 4.1.3.2 Forme canonique des prédicats sur les données

On définit un *invariant de cohérence* comme un prédicat sur des adresses (i.e. de variables). Un ensemble d'adresses est cohérente si tous les invariants de cohérences sur ces adresses sont vrais.

Le matériel permet la modification d'un mot de manière atomique ; ainsi tout invariant portant sur un seul mot peut être facilement respecté.

Soit $a$ et $b$ deux ensembles d'adresses disjointes ; et $P_a$ et $P_b$ des prédicats portant sur les valeurs possibles de resp. $a$ et $b$. On peut écrire un *prédicat composé* $P_{a\cup b}$ à l'aide d'une formule logique sur $P_a$ et $P_b$. Il est clair que tout prédicat portant sur un ensemble de données $a \cup b$ peut s'écrire comme la composition inductive de prédicats sur des ensembles de données $a$ et $b$ disjoints et strictement plus petits.

Nous analysons maintenant comment un prédicat $P_{a\cup b}(d)$ composé des 2 prédicats $P_a(a)$ et $P_b(b)$ peut rester vrai, quand $P_a(a)$ et $P_b(b)$ changent suite aux modifications de $a$ et $b$. La Table 4.1 donne une liste des compositions possibles de $P_a(a)$ et $P_b(b)$ ;

[8] mais pas suffisante, car la façon dont sont faites les lectures est également importante

| $P_a(a)=0$ $P_b(b)=0$ | $P_a(a)=0$ $P_b(b)=1$ | $P_a(a)=1$ $P_b(b)=0$ | $P_a(a)=1$ $P_b(b)=1$ | Formule équivalente |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | $P_a(a) \wedge P_b(b)$ |
| 0 | 0 | 1 | 0 | $P_a(a) \wedge \neg P_b(b)$ |
| 0 | 0 | 1 | 1 | $P_a(a)$ |
| 0 | 1 | 0 | 0 | $\neg P_a(a) \wedge P_b(b)$ |
| 0 | 1 | 0 | 1 | $P_b(b)$ |
| 0 | 1 | 1 | 0 | $P_a(a) \iff \neg P_b(b)$ |
| 0 | 1 | 1 | 1 | $\neg P_a(a) \Rightarrow P_b(b)$ |
| 1 | 0 | 0 | 0 | $\neg P_a(a) \wedge \neg P_b(b)$ |
| 1 | 0 | 0 | 1 | $P_a(a) \iff P_b(b)$ |
| 1 | 0 | 1 | 0 | $\neg P_b(b)$ |
| 1 | 0 | 1 | 1 | $\neg P_a(a) \Rightarrow \neg P_b(b)$ |
| 1 | 1 | 0 | 0 | $\neg P_a(a)$ |
| 1 | 1 | 0 | 1 | $P_a(a) \Rightarrow P_b(b)$ |
| 1 | 1 | 1 | 0 | $P_a(a) \Rightarrow \neg P_b(b)$ |
| 1 | 1 | 1 | 1 | 1 |

Tab. 4.1 – Ensemble des compositions possibles de 2 prédicats.

ces compositions peuvent être regroupées ($P_a(a)$ et $P_b(b)$ jouant des rôles symétriques, tout comme $P_a(a)$ et $\neg P_a(a)$, $P_b(b)$ et $\neg P_b(b)$).

- Cas 0 : $P_{a \cup b}$ ne peut jamais être satisfait.
- Cas 1 : $P_{a \cup b}$ est toujours satisfait.
- Cas $P_a(a)$ : $P_a(a)$ doit toujours être satisfait, et $P_b(b)$ est inutile. Notons que ces trois premiers cas ne sont pas vraiment des compositions.
- Cas $P_a(a) \wedge P_b(b)$ : les deux prédicats doivent chacun être toujours satisfaits, indépendamment l'un de l'autre. Ainsi, toute modification de $a$ ou de $b$ doit continuellement respecter resp. $P_a(a)$ ou $P_b(b)$.
- Cas $P_a(a) \Rightarrow P_b(b)$ : pour satisfaire $P_{a \cup b}$, il est possible de ne pas respecter l'un ou deux des deux prédicats. Cependant, il est interdit de passer par l'état $P_a(a) \wedge \neg P_b(b)$, ce qui implique que les modifications doivent se faire dans un certain ordre (Figure 4.1).
- Cas $P_a(a) \iff P_b(b)$ : les deux prédicats doivent tous les deux être simultanément satisfaits, ou simultanément insatisfaits. Cependant, comme $a$ et $b$ sont à des adresses disjointes, passer d'un état à l'autre implique de passer par l'un des états intermédiaires $P_a(a) \wedge \neg P_b(b)$ ou $P_b(a) \wedge P_b(b)$ qui sont tous les deux interdits. On ne peut donc pas passer d'un état à l'autre, comme le monter la Figure 4.1 ; ce qui fait que ce cas se ramène au cas $P_a(a) \wedge P_b(b)$. Le

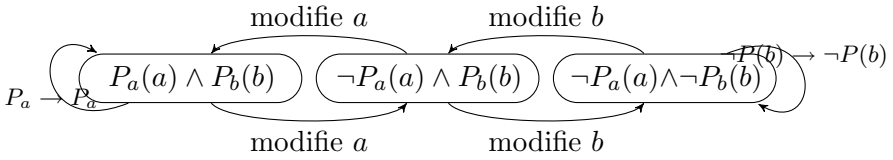



FIG. 4.1 – Ordre des modifications qui respectent l'invariant $P_a(a) \Rightarrow P_b(b)$.

fait que cette composition n'est pas utilisable est l'inconvénient majeur par rapport aux sections critiques habituelles.

En utilisant la forme canonique des invariants, il est ainsi possible de savoir si un invariant de cohérence peut être ou non continuellement respecté.

**Équivalences cachées** Il faut prendre garde à ce que nous appelons des "équivalence cachées". C'est à dire que si l'invariant est $P_a(a) \Rightarrow P_b(b)$, il faut que l'état $\neg P_a(a) \wedge P_b(b)$ soit explicitement autorisé.

Par exemple, on ne peut écrire l'invariant $i = j$ sous forme canonique ainsi :

$$\forall n, \quad i = n \;\Rightarrow\; j = n$$

Si on appelle $P_n$ le prédicat $i = n$, et $P'_n$ le prédicat $j = n$, il est en fait impossible d'avoir $\neg P_n \wedge P'_n$ : si $i = m \neq n$, alors en appliquant $P_m \Rightarrow P'_m$ on a forcément $j = m$. Ainsi, il faut écrire l'invariant comme :

$$\forall n, \quad i = n \iff j = n$$

Par la suite, nous supposons avoir retiré toutes les équivalences cachées.

#### 4.1.3.3 Pratique : affaiblissement des invariants

Certaines propriétés ne peuvent donc pas être assurées de manière continue, dès qu'il y a une équivalence à assurer. Nous voyons maintenant différents moyens par lesquels ces contraintes peuvent être contournées. On peut généralement modifier les invariants afin de les mettre sous la bonne forme, ce que nous appelons *affaiblissement des invariants*.

**Copie partiellement synchronisée** Ainsi, tout invariant de la forme $f(a) = g(b)$, où $a$ et $b$ sont des ensembles d'adresses disjoints, s'écrit sous forme canonique

$$\forall n, \quad f(a) = v \iff g(b) = v$$

et ne peut être assuré par un revocable lock. Par contre, on peut souvent transformer ces invariants en :

$$\forall v \neq 0, \quad f(a) = v \Rightarrow g(b) = v$$

I.e., soit $a$ contient une copie de $b$, soit $a$ n'est pas synchronisé avec $b$. Ce dernier invariant peut être respecté de manière continue.

**Inégalité** Les égalités ne peuvent pas être continuellement respectées, mais les inégalités le peuvent. I.e. l'invariant

$$s \;=\; |\{\, x \in B \mid P(x) \,\}|$$

ne peut être respecté, au contraire de l'invariant

$$s \;\geq\; |\{\, x \in B \mid P(x) \,\}|$$

que l'on écrit sous forme canonique par :

$$\forall n, \quad s \geq n \;\Rightarrow\; |\{\, x \in B \mid P(x) \,\}| \leq n$$

Pour respecter cet invariant, il faut que les modifications à $s$ et $B$ soient faites dans cet ordre :

| | |
|---|---|
| `s++` | $\lvert\{\, x \in B \mid P(x) \,\}\rvert$`--` |
| $\lvert\{\, x \in B \mid P(x) \,\}\rvert$`++` | `s--` |

En général, on a envie que l'invariant avec inégalité soit continuellement respecté, mais sans éviter que $s$ soit complètement en décalage par rapport à ce qu'il représente. On peut s'assurer de cela lorsqu'on a un mécanisme qui donne une garantie de terminaison, comme la section non préemptible ou le rollforward lock (ou le lock classique). $s$ et sa valeur sont donc seulement temporairement en décalage, et ce décalage ne peut être que d'un seul coté.

On utilise cette propriété pour des compteurs de références, par exemple pour le compteur du nombre de fois où est mappé une page en mémoire : ce compteur est toujours supérieur au nombre de mappings effectifs. Ainsi, si ce compteur est à 0, cela signifie que la page n'est pas mappée. L'annexe C.2.2.1 présente ce problème et son implémentation en détail.

**Indicateur de validité** Si on veut ne pas respecter un prédicat $P_b$ sur un ensemble de données $b$ de manière temporaire, il suffit de trouver un ensemble de données $a$ et un prédicat $P_a$ tel que $P_a(a)$ soit faux quand $P_b$ l'est. Le plus simple pour cela est que $a$ soit un booléen : si $a$ est vrai, $b$ est cohérent, sinon $b$ peut ne pas l'être. $a$ peut être placé dans le même mot que le verrou pour gagner de la place. Cette technique est très souvent utilisable. Le 2-handed émulation de Greenwald [GC96] en est une application.

#### 4.1.3.4 Conclusion

On peut souvent trouver des moyens pour respecter un invariant de manière continue, ce qui est souvent utile dans les programmes parallèles. Cela permet souvent de ne pas utiliser de primitives de synchronisation (en particulier cela aide pour faire les lectures parallèlement aux écritures), ou des primitives plus légères pour les écritures. Ces primitives sont étudiées dans la section suivante.

## 4.2 Primitives de synchronisation

Dans cette section, nous détaillons les primitives de synchronisations mises à disposition par le hardware et le noyau pour la résolution de ces problèmes de synchronisation. La section suivante montrera en détail comment les utiliser.

Beaucoup de systèmes d'exploitations offrent aux applications des primitives de synchronisation par l'intermédiaire d'appels systèmes (e.g. les sémaphores dans THE [Dij68], les futex dans Linux [FRK02], ou les appels d'IPC `semget` dans UNIX System V [BC05])

Notre approche a été de fournir un support minimal du noyau pour permettre l'écriture de mécanismes variés en espace utilisateur. Cette section décrit les différentes composantes de ce support. Nous commençons cependant par les sections non préemptibles, bien qu'elles soient réservées aux services implémentés dans le noyau.

### 4.2.1 Sections non préemptibles

section non préemptible

La *section non préemptible*, permet l'exécution d'une portion de code sans se faire interrompre. Nous en parlons car nous avons un emploi particulier de ce mécanisme classique.

**Intérêt** Comme on l'a vu (§ 4.1.2.1), ce mécanisme est utilisé pour implémenter une section critique sur monoprocesseur, ou en multiprocesseur en y ajoutant des spinlock. Mais il est également possible de faire un usage beaucoup plus subtil de cette primitive.

Ce que la section non préemptible offre[9], en multiprocesseur, est une garantie contre la non-terminaison et les opérations retardées (§ 4.1.1.3). Voici deux exemples de codes parallèles, exécutés dans des sections non préemptibles, qui tirent partie de ces propriétés :

```
void atomic_move_between_lists(list_t from, list_t to)
{
  void *elem = atomic_pop( from) ;
  atomic_push( to, elem) ;
}
```

Dans ce premier exemple, on fait temporairement disparaître un objet : il n'apparaît plus dans aucune des deux listes. La garantie de non-terminaison permet de s'assurer qu'il ne disparaitra pas à tout jamais : on sait que l'elément va ensuite appartenir à la liste `to`.

```
void atomic_push_and_add_counter( list_t to, void *elem)
{
  atomic_add( to->nb_elem, 1) ;
  atomic_push( to, elem) ;
}
```

[9] dans une moindre mesure, les sursis d'exécution offre également cela

Dans ce deuxième exemple, la liste `to` possède un champ `nb_elem` qui contient le nombre d'éléments de la liste. Ici, même si `nb_elem` ne reflète pas exactement la réalité, il y a juste une désynchronisation temporaire entre la valeur de ce champ et le vrai nombre d'élément : au bout d'un temps court ces valeurs seront égales.

Ce type de garanties faibles est ce qui a permis l'implémentation de notre système de mémoire virtuelle quasiment sans verrou (voir annexe C.2, notre implémentation efficace du système de capacités (annexe A.1.3), ou l'affaiblissement des invariants (section 4.1.3.3)).

Il y a également d'autres usages ; par exemple dans une section non préemptible, le nombre de thread concurrents est au plus égal au nombre de processeurs de la machine. Cela permet de borner la consommation mémoire pour les algorithmes lock-free.

**Implémentation** L'implémentation se fait tout simplement dans le noyau : il suffit de masquer les interruptions.

En fait, nous avons choisi que le noyau soit intégralement exécuté dans une section non préemptible. C'est ce que Ford et al. appellent l'*interrupt model* [FHL+99]. Cela simplifie l'implémentation du noyau, économise de la mémoire (pour toutes les piles noyaux), et permet de meilleures performances.

Pour éviter que les sections non préemptibles aient un impact sur l'ordonnancement, nous utilisons une technique de *point de préemption explicite*, qui consiste à indiquer dans le code les endroits où la préemption est permise. Ces points de préemptions peuvent s'implémenter en démasquant et remasquant les interruptions, mais nous en avons une implémentation plus efficace.

Le noyau s'exécute avec les interruptions démasquées. Lorsqu'une interruption cause un appel à l'ordonnanceur, on écrit dans une variable pour le processeur correspondant. Ainsi, le surcoût du point de préemption explicite est seulement celui de lire et tester une variable. Le fait de ne pas masquer les interruptions permettra également l'usage d'interruptions entre processeurs à l'intérieur du noyau.

Notons que dans un système à noyau monolithique, ou du moins dont le besoin en sécurité est moins élevé et où on peut se permettre d'accorder à tous les services le droit de masquer les interruptions, cette primitive de synchronisation, accompagnée de spinlocks, est suffisante pour tous les besoins en synchronisation. Les primitives étudiées plus loin servent pour la synchronisation dans les cas où on n'accorde pas de confiance au service.

Notons également qu'on pourrait envisager de faire des sections non préemptibles pour du code qui n'est pas de confiance. On peut par exemple fournir par le noyau une portion de code de confiance ; lorsque le program counter se trouve dans cette portion de code, le noyau permet l'exécution du code jusqu'au bout. La section serait non-préemptible sauf si une exception, comme un défaut de page, était déclenché. Cette voie, que nous n'avons pas eu le temps d'explorer, pourrait permettre davantage de primitives de synchronisation légère en espace utilisateur.

### 4.2.2 Les notifications inter-CPU

**Présentation** Un autre mécanisme mis à disposition par le noyau est un mécanisme de notifications inter-CPU. Il s'agit de la virtualisation d'un mécanisme que permet le hardware, l'envoi d'interruption inter-CPU (IPI). Le but est de permettre à un thread dans un domaine d'agir sur les exécutions des autres threads dans le même domaine. Ce mécanisme est utile pour certaines synchronisations.

Il marche de la manière suivante : le thread invoque la méthode `cross_cpu_notification` sur une capacité vers un domaine, avec en paramètre un bitfield indiquant quels sont les CPUs auxquels envoyer une notification. Le noyau fait un « et logique » de ce bitfield avec le bitfield de tous les CPU qui sont présent dans le domaine au moment de l'appel de la méthode, et leur envoie à tous une interruption. Cela garantit qu'on ne peut perturber l'exécution que des processeurs qui s'exécutent dans un domaine dont on a la capacité, i.e. dont on a le droit de perturber l'exécution.

**Utilisation** Notre principale utilisation de cette notification est pour notre schéma de synchronisation use/destroy, décrite section 5.2.4. En particulier, cela permet la prévention de bugs TOCTTOU. Par exemple, lorsque qu'un objet est détruit (et toutes les capacités vers cet objet invalidées), il faut être sûr que tous les threads qui sont dans le service et qui l'utilisent ne puissent plus continuer à l'utiliser. Pour tous les threads au repos (i.e. tous les autres threads en monoprocesseur), il suffit, avant que le thread reprenne son exécution dans le service, de vérifier que sa capacité permet encore d'utiliser la ressource. En multiprocesseur, on y ajoute ce mécanisme, pour les threads en cours d'utilisation de la ressource.

Il y a d'autres usages : par exemple un noyau paravirtualisé utilise fréquemment des interruptions inter-CPU, par exemple pour le "TLB shootdown", i.e. l'invalidation d'entrées du TLB sur les autres processeurs. Cet exemple est particulièrement important puisque notre système de mémoire virtuelle ne garantit par la cohérence des TLBs entre les différents processeurs ; le TLB shootdown doit donc toujours être implémenté à l'aide de ce mécanisme pour ces noyaux paravirtualisés.

### 4.2.3 La préemption programmable

Nous présentons maintenant la *préemption programmable*, le mécanisme qui est à la base de la plupart des primitives de synchronisation légères pour l'espace utilisateur proposées dans Anaxagoros.

#### 4.2.3.1 Motivation

**Recouvrement** Nous avons vu (§ 4.1.2.1) que le spinlock est un mécanisme simple et efficace pour résoudre les problèmes de concurrence pour le multiprocesseur. Mais la possibilité de préemption dans une section critique le rend bloquant. Et empêcher la préemption dans la section critique pose différents problèmes (§ 4.1.2.1).

Nous allons donc plutôt agir a posteriori pour débloquer l'exécution, dans le (rare) cas où le thread est préempté dans une section critique ; c'est ce que nous
recouvrement appelons le *recouvrement*. Cela permettra d'implémenter des sections critiques en

espace utilisateur, non-bloquantes et qui n'impactent pas l'ordonnancement.

**Note :** Note sur la notion de non-bloquant. Le mécanisme décrit est non-bloquant si, dans la définition page 148, par "arrêt d'un thread" on entend "préemption", mais pas « progrès infiniment lent » on implémente donc une notion un peu plus faible. Bien qu'on puisse imaginer des cas où cela fait une différence (processeur qui ne peut plus avancer, à cause d'un bug hardware où d'un problème de bus), il n'y a en pratique pas de grande différence entre ces notions pour des multiprocesseurs "normaux" ; et les autres cas peuvent se gérer par e.g. par des timeouts hardware. [Ber93] a également noté ce problème.

**Besoins pour implémenter le recouvrement** Le recouvrement nécessite de savoir répondre à ces trois questions :

- les threads en attente doivent savoir si un thread exécute la section critique, ou s'il a été préempté. En monoprocesseur c'est évident, mais pas en multiprocesseur[10] ;
- le thread qui était dans la section critique doit savoir qu'il y a eu recouvrement ;
- certaines stratégies de recouvrement consistent à restaurer l'état du processeur (i.e. la pile et les valeurs des registres) pour terminer la section critique, ce qui demande 1/ que cet état soit accessible, et 2/ qu'il puisse être restauré.

Notre contribution consiste en un mécanisme léger et efficace pour répondre à ces trois points. En particulier, nous n'introduisons aucun appel système supplémentaire. En bonus, cela constitue un mécanisme efficace pour faire de l'ordonnancement de thread utilisateur, analogue aux scheduler activations [ABLL92].

#### 4.2.3.2 Action lors de la préemption

**Présentation** Pour prévenir les autres threads lorsqu'on s'est fait préempter, nous exécutions une dernière action au moment de chaque préemption (dans une section critique ou non). Mais au lieu de déclencher un upcall lorsqu'un thread est préempté (e.g. comme dans [ABLL92, MSLM91, EKJO95]), ce qui pose des problèmes de sursis d'exécution (§ 4.1.2.1), nous nous contentons d'effectuer une action fixe. Cela permet de garder la préemption rapide, et de se garder de tous problème qu'on aurait eu en limitant le temps passé dans ce sursis d'exécution.

Le code que nous avons choisi d'exécuter lors de la préemption est le suivant :

[10] Le fait que les threads ne peuvent pas connaître cette information est la raison pour laquelle dans les systèmes classiques, les sleeplocks font une attente active pendant un petit temps avant d'appeler l'ordonnanceur (e.g. [HF05]) ; ce temps CPU est souvent inutilement perdu.

```
if( *addr_lock == expected_value)
{
  // Sauve tous les registres du CPU à addr_dump
  dump_cpu( addr_dump) ;
  test_success = 1 ;
  *addr_lock = new_value ;
}
```

Listing 4.1 – Code simplifié de la préemption programmable

où `addr_lock`, `expected_value`, `new_value`, `test_success` et `addr_dump` sont des variables par thread que le service choisit.

L'idée est que lorsque le thread qui a pris le lock se fait préempter, alors la valeur du lock va changer pour `new_value`, ce qui prévient les autres threads potentiellement en attente active de cette préemption.

La variable `test_success` est mise à 1 pour prévenir le thread qu'il s'est fait préempter (il testera la valeur du lock en complément pour savoir s'il y a eu recouvrement).

Enfin, les registres du processeurs sont écrits en espace utilisateur (fonction `dump_cpu`, afin qu'ils puissent être réutilisés (généralement, les registres processeurs sont sauvegardés dans le noyau).

**Notes**

- Notons que l'ordre des opérations est important ; en particulier il faut libérer le lock après avoir dumpé les registres CPU pour éviter la race condition où on tente d'accéder à ces registres dès que le lock est libéré, et avant qu'ils ne soient écrit.
- Dans le cas d'une exception, il faudrait aussi exécuter ce code. Nous ne l'avons pas implémenté ; a part le défaut de page, les autres exceptions (instruction illégale, division par zéro) sont des erreurs de programmations qu'on peut facilement éviter. Comme l'allocation dans la mémoire des services est statique, le problème des exceptions de défaut de page ne se pose pas non plus.
- Dans l'implémentation, ces variables sont stockées dans l'UTCB, une zone de mémoire par thread dans laquelle un domaine peut écrire ssi le thread est en train d'être exécuté.
- Lors du transfert de thread entre services, des valeurs par défaut sont placées dans ces variables. Ainsi ce mécanisme ne peut pas être utilisé pour une attaque ; et est de plus complètement transparent pour un programme qui ne souhaiterait pas l'utiliser.
- En monoprocesseur, si un thread constate qu'un verrou est pris, il sait que son détenteur a été préempté dans la section critique. Il est donc inutile de modifier `addr_lock`.
- On pourrait rajouter un paramètre `mask` à la préemption programmable, pour permettre à un lock de n'occuper qu'une partie du mot. La préemption programmable exécuterait alors :

```
local = *addr_lock ;
if( local & mask == expected_value) { ...
   *addr_lock = (local & ~mask) | new_value ; }
```

**Implémentation** L'implémentation de ce mécanisme peut varier selon la plateforme.

Dans l'architecture Pentium IA32 [Int09], le noyau et l'espace utilisateur partagent le même espace d'adressage ; le noyau a seulement le droit d'accéder à certaines pages en plus. Ainsi l'implémentation de la préemption ci-dessus dans le noyau est simple, mais pose des problèmes de sécurité : il ne faut pas qu'en utilisant ce mécanisme, l'espace utilisateur puisse arriver à écrire dans des structures réservées au noyau.

Pour contrer ce problème, nous séparons l'espace d'adressage en deux zones contiguës : une zone est réservée à l'espace utilisateur, et l'autre au noyau. Ainsi, il suffit de vérifier que les adresses fournies par l'utilisateur sont dans l'intervalle utilisateur pour écrire sans crainte pour les structures du noyau.

Cette précaution ne suffit pas : l'espace utilisateur peut avoir mappé, dans son espace d'adressage, des pages qui ne doivent pas être modifiées (par ex. des pages contenant du code). Ces pages sont mappées en read-only, et il ne faut pas que le noyau puisse écrire dedans[11].

Le dernier problème est le défaut de page : les adresses fournies par l'utilisateur peuvent pointer sur des adresses non mappées. Plutôt que de vérifier si ces adresses sont correctes (ce qui demanderait de parcourir la table des pages à la main), nous avons fait en sorte de rendre le système robuste aux défauts de page (cela demande des modifications simples à la routine d'exception de défaut de page).

L'implémentation pourrait être complètement différente dans d'autres architectures. On pourrait par exemple faire parcourir par le noyau la table des pages manuellement , récupérer l'adresse physique de la page avant d'y écrire les registres processeurs. Dans un système d'exploitation à espace d'adressage unique, des optimisations sont également possibles...

#### 4.2.3.3 Action lors de la reprise

**Présentation** La reprise d'exécution ne peut plus être assurée par le noyau, qui ne sait plus où sont les registres CPU à charger. À la place, l'exécution reprend en plaçant le program counter à une valeur choisie par le service. Nous appelons cela l'*upcall de reprise*. L'upcall de reprise est paramétrable par la variable `upcall_entry`, placée également dans l'UTCB.

Typiquement, l'upcall de reprise rechargera les registres du thread pour reprendre l'exécution d'où elle en était. Mais on peut aussi revenir en arrière, ou plus tard, dans l'exécution. On peut même utiliser cela pour implémenter du M-on-N scheduling, i.e. multiplexer $M$ threads utilisateur au dessus de $N$ threads noyau.

[11] Il faut sur le Pentium désactiver un flag WP qui permet au noyau d'écrire dans des pages en read-only

Ce mécanisme est différent de certain existants [MSLM91, ABLL92] en ce que *les upcalls de reprise ne s'empilent pas* : si un nouvel upcall est reçu tandis que l'ancien n'avait pas terminé d'être exécuté, l'exécution de l'ancien est tout simplement abandonnée. Cela est implémenté notamment par la zone de non-sauvegarde sur préemption présentée ci-après.

**Implémentation de la restauration des registres** On se pose ici la question de savoir s'il est possible de recharger le contexte d'un processeur (i.e. tous ses registres, y compris le registre d'état) depuis l'espace utilisateur. La difficulté est la suivante : le program counter doit être modifié en dernier, puisque après sa modification, on ne peut plus exécuter de code pour charger les autres registres, qui doivent donc être chargés auparavant.

Charger les registres qui ne sont pas le program counter n'est pas très difficile, c'est le chargement de ce dernier qui pose problème. Nous proposons différentes implémentations selon l'architecture, afin de montrer qu'il existe toujours une solution à ce problème.

**Mode d'adressage complexe** Sur des architectures qui supportent des modes d'adressage complexe comme l'Intel IA32, une solution simple consiste à stocker le program counter à charger à une adresse fixe (dans l'UTCB dans notre implémentation). Le jeu d'instruction permet de charger le program counter avec le contenu d'une adresse fixe. Sur l'IA32, cette solution s'écrit :

```
// Réserve un mot de l'UTCB
#define temp_pc (CURRENT_UTCB_ADDRESS + offset__temp_pc)
movl %eax, temp_pc //eax = program counter à restaurer
...                // Restaure les autres registres
jmp *(temp_pc)     // Restaure le program counter
```

C'est la solution que nous avons retenue pour l'implémentation sur IA32.

**Dépilement du program counter** Certaines architectures peuvent permettre de dépiler le program counter (e.g. ARM, IA32). On peut alors écrire au sommet de la pile à restaurer la valeur du program counter à restaurer ; charger tous les registres, en modifiant le stack pointer ; puis dépiler le program counter.

```
// esp déjà restauré
pushl %eax //eax = program counter à restaurer
...        // Restaure les autres registres
ret        // Dépile le pc, et repositionne sp
```

Le gros problème de cette technique est que l'application doit permettre qu'on puisse écrire au dessus du sommet de sa pile à tout moment. Il y a un risque de bug pour les programmes qui manipulent le stack pointer, comme ceux qui utilisent des librairies d'user-level threading. Ce problème est mitigé par le fait que le noyau Linux utilise également le dessus de la pile pour exécute les signal handlers [BC05] (sauf appel spécifique à l'appel système `sigaltstack`) ; ainsi ce bug potentiel a une

probabilité très faible d'arriver sur des applications usuelles. Mais cela peut poser problème par exemple pour des noyaux paravirtualisés.

**Code auto-modifiant** Une troisième technique est celle du code auto-modifiant. Il suffit d'écrire un opcode qui fasse un saut à l'adresse souhaitée, puis de sauter vers cet opcode pour exécuter le saut. Quand les processeurs ne permettent que des sauts relatifs, un petit calcul d'offset est nécessaire. En voici l'implémentation pour l'IA-32 :

```
// Reserve deux mots de l'UTCB, ecrivible et executable
#define self_mod (CURRENT_UTCB_ADDRESS + offset__self_mod)
// eax = program counter à restaurer
subl $(self_mod+8), %eax // saut relatif à fin de l'instr.
movl %eax, (self_mod+4) // stocke adresse du saut
movl $0xe9000000, self_mod // stocke l'opcode du saut
...                        // restaure les autres registres
jmp self_mod+3             // saute vers l'instruction créée
```

Il faut prendre certaines précautions, pour les architectures qui peuvent avoir des caches d'instruction et de données séparés, détaillées dans le manuel (e.g. [Int09]) ; mais les processeurs devraient généralement supporter cette technique, dont les prérequis sont similaires à celles utilisées pour les langages qui passent des paramètres en call-by-name (comme Algol), ou pour la compilation just-in-time.

**Appel système** Enfin, dans le cas où aucune de ces techniques n'est possible (e.g. le SPARC a besoin d'instruction privilégiée pour changer le register window [MSLM91]), on pourrait toujours utiliser un appel système pour ce faire. Pour minimiser le surcoût lors d'un retour après une préemption, les registres peuvent être par défaut sauvegardés et restaurés par le noyau, sauf s'il y a eu préemption dans la section critique. Ainsi l'overhead n'interviendrait que dans ce cas rare.

Quelque soit la méthode, il est donc possible de charger efficacement un contexte processeur stocké en espace utilisateur.

#### 4.2.3.4 La zone de non-sauvegarde des registres sur préemption

La responsabilité de recharger les registres est déportée à l'utilisateur par la routine de rechargement. Mais l'emploi de cette routine pose cependant problème, lorsqu'on désire recharger des registres à une adresse `addr_dump` et que les futures préemptions stockent les registres également à cette adresse. La routine doit avoir paramétré `addr_dump` avant de reprendre sa précédente exécution, i.e. avant d'avoir rechargé tous les registres. Mais tant que tous les registres ne sont pas rechargés, il peut y avoir une préemption, et la programmable préemption écraserait l'ancien contenu de `addr_dump`.

Pour résoudre ce problème, nous définissons une *zone de non-sauvegarde sur préemption*, dans laquelle nous plaçons la routine de restauration des registres. La zone suit le principe suivant : lorsque le program counter est dans cette zone,

les registres ne sont pas sauvegardés dans `addr_dump`. Le code exécuté lors de la préemption devient donc :

```
if( *addr_lock == expected_value)
{
  // Si le program counter au moment de la préemption
  // n'était pas dans la zone de non-sauvegarde
  if( !is_in_no_save_on_preempt_zone( user_pc)) {
    // Sauve tous les registres du CPU à addr_dump
    dump_cpu( addr_dump) ;
  }
  test_success = 1 ;
  *addr_lock = expected_value ;
}
```

Listing 4.2 – Code exécuté lors de la préemption programmable

Comme les registres ne sont plus sauvegardés, le code dans cette zone doit être conçu pour cette éventualité, e.g. doit faire attention à l'endroit d'où l'exécution reprendra.

Cette zone est utile dans l'implémentation de différents mécanismes de synchronisation plus haut niveau, et pas seulement à la reprise de l'exécution (e.g. l'implémentation du rollforward lock en section 5.2.2).

**Note :** Notre première implémentation permettait à l'utilisateur de configurer les adresses de début et de fin dans son code. C'est toujours possible, mais nous avons trouvé plus commode (pour l'initialisation des programmes, et pour pouvoir modifier l'upcall de reprise lorsqu'un thread change de domaine pour qu'il ait un comportement simple et sûr par défaut) que cette zone soit du code pour l'espace utilisateur fourni par le noyau à une adresse fixe.

#### 4.2.3.5 Résumé

La préemption programmable va nous permettre d'implémenter simplement et efficacement tout un ensemble de primitives de synchronisation de plus haut niveau. Comme les mécanismes décrits ci-dessus ne demandent pas d'appels systèmes, cela permettra des mécanismes extrêmement efficaces, et sans rajout de complexité pour le noyau ; ces mécanismes sont décrits dans la section suivante.

De plus, cela permettra de faire de l'ordonnancement de thread utilisateur à la manière des scheduler activations [ABLL92] si désiré.

## 4.3 Implémentation des sections critiques

Comme on l'a vu, la mise en place de section critique, qui consiste à s'assurer qu'un thread est seul à accéder à une ressource, est un mécanisme usuel pour la gestion de la synchronisation. Aussi nous intéressons nous ici à son implémentation dans Anaxagoros, à l'aide de la préemption programmable vue précédemment.

Il y a trois possibilités pour implémenter des sections critiques non bloquantes ; nous expliquons ici pourquoi, avant de détailler l'examen de ces trois possibilités.

### 4.3.1 Possibilités d'implémentation

**Contraintes** Comme vu en Section 4.1.2.1, l'implémentation usuelle des sections critiques pose des problèmes qui empêche leur utilisation dans Anaxagoros. Une implémentation acceptable des sections critiques devra donc :

- pouvoir être utilisable par du code sans confiance et non privilégié, contrairement aux masquage des interruptions et spinlocks ;
- ne pas bloquer les autres threads s'il y a préemption dans la section critique ;
- ni les réordonnancer de force, ou perturber de quelconque manière les décisions d'ordonnancement prises par ailleurs.

**Stratégies de recouvrement** Le problème principal est celui de la préemption lors de la section critique. En évitant ce problème, le masquage des interruptions fournit une section critique simple, en conjonction avec le spinlock en multiprocesseur.

Comme un programme sans confiance ne peut pas éviter qu'un thread ne puisse pas être préempté dans la section critique, il faut résoudre ce problème. La méthode classique pour ce faire consiste à appeler l'ordonnanceur, pour faire autre chose tant que le verrou n'est pas libéré, solution qui ne convient pas pour les services d'Anaxagoros.

À la place, nous faisons en sorte qu'un thread préempté dans la section critique modifie le verrou pour l'indiquer, afin que les autres threads puissent se débloquer. On peut voir trois manières [12] de faire cela :

- le *rollforward* : l'exécution de la section critique est poursuivie, ce qui libère le verrou. L'exécution se passe comme si la préemption avait eu lieu juste après la section critique ;
- le *rollback* : les écritures en cours sont annulées et les anciennes valeurs sont restaurées par un autre thread. L'exécution se passe comme si la préemption avait eu lieu juste avant la section critique ;
- la *revocation* : un autre thread devient tout simplement détenteur du lock. Le thread qui détenait précédemment le lock sort immédiatement de la section critique. Le recouvrement se fait de manière ad-hoc.

Ces trois solutions permettent d'implémenter des sections critiques non-bloquantes.

Notons que le rollback et la révocation demandent a priori un support de la part du programmeur. Le rollback impose de maintenir un journal des écritures faites pour pouvoir les défaire [13], et le revocable lock de définir une *fonction de recouvrement*. Le rollforward lock n'a pas ces contraintes, et est donc plus facilement utilisable.

[12] [Ber93] s'était précédemment intéressé au recouvrement, mais n'avait pas considéré la révocation.
[13] qui pourrait être maintenu automatiquement avec un langage de programmation adéquat.

**Implémentation** Notons que ces stratégies de recouvrement implémentent quasiment[14] à elles seules une section critique en monoprocesseur. En effet, il suffit alors d'exécuter l'algorithme en pseudo-code suivant :

```
if( !is_free( lock)) recover( lock) ;
take_lock( lock) ;
// section critique
free_lock( lock) ;
```

pour implémenter une section critique. En effet, comme un seul thread peut simultanément s'exécuter, tout lock pris rencontré a forcément été pris par un thread qui s'est fait préempté dans sa section critique. Notons que parfois le recouvrement ne peut pas être exécuté plusieurs fois.

En multiprocesseur, il faut y ajouter des spinlocks pour s'assurer d'être seul à exécuter la section critique. De plus, il faut notifier les threads en attente active que le thread pris est préempté, ce que nous faisons en modifiant la valeur du lock par la préemption programmable.

### 4.3.2 Le rollforward lock

#### 4.3.2.1 Cas général

rollforward lock

**Présentation** Le *rollforward lock* est un verrou non bloquant implémentant la stratégie de recouvrement rollforward (§ 4.3.1) : lorsqu'un thread est préempté dans une section critique protégée par un lock de ce type, les thread bloqués peuvent terminer l'exécution de ce thread jusqu'à ce que le verrou soit libéré, afin de le débloquer. La Figure 4.2 en fournit un exemple.

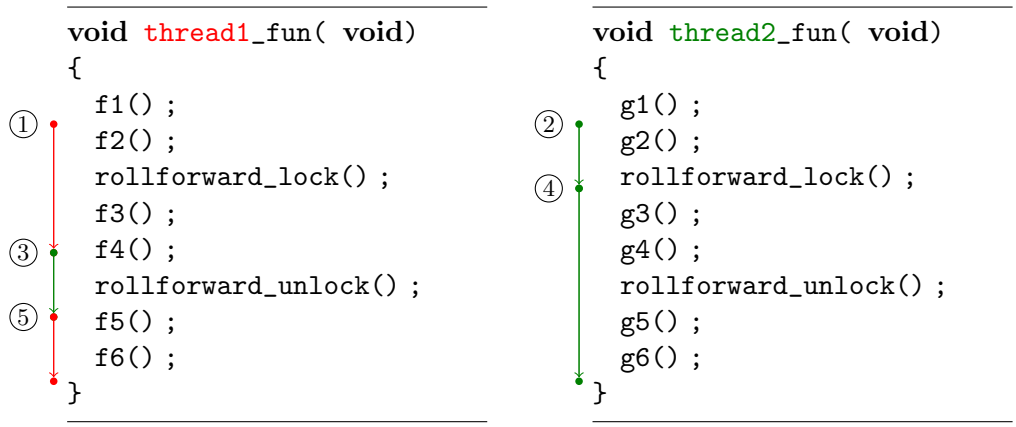



FIG. 4.2 – Exemple d'exécution pour deux threads, en 5 étapes : 1/ Le thread 1 s'exécute, et est préempté dans la section critique. 2/Le thread 2 tente d'acquérir le verrou. 3/ Le thread 2 termine l'exécution de la section critique du thread 1. 4/ Le thread 2 rentre dans la section critique, puis continue son exécution normalement. 5/ Le thread 1 continue son exécution après la section critique.

[14] Il reste quelques problèmes, comme la préemption pendant le recouvrement.

**Implémentation** En multiprocesseur, l'implémentation est centrée autour des états que peut prendre le verrou (Figure 4.3). Celui-ci sert a garantir qu'un seul thread (et un seul processeur en SMP) ne s'exécute dans la section critique, que le thread soit en train d'exécuter la section critique pour son compte ou pour celui d'un autre thread. Le pseudo-code du rollforward lock est donné par le Listing 4.3 ; le code complet se trouve en annexe B.2.

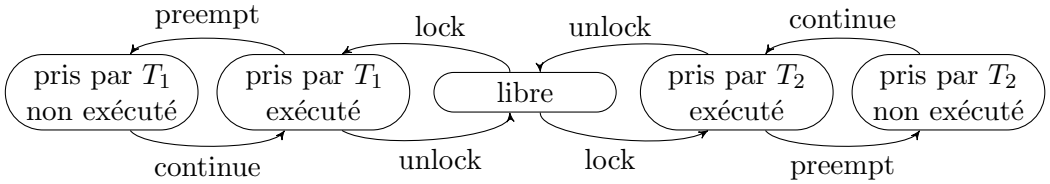



FIG. 4.3 – États possible du rollforward lock, pour 2 threads. Les états changent quand le lock est pris ou libéré, que le thread qui détient un lock est préempté, ou qu'un thread continue l'exécution dans la section critique.

La continuation de l'exécution se fait simplement, en rechargeant tous les registres du thread dont on continue l'exécution. Le reste de l'état du thread est contenu dans sa pile, qui est utilisée quand on charge le registre de stack pointer. On peut voir cela comme un changement temporaire de thread utilisateur. Le lock assure qu'un seul thread utilise une seule stack simultanément.

À chaque fois qu'on essaie d'acquérir le lock (avec un CAS en multiprocesseur, mais une simple écriture en monoprocesseur), il faut précédemment avoir paramétrer les paramètres `addr_lock`, `expected_value`, `new_value` etc. pour qu'ils puissent libérer le verrou s'il y a préemption. Toute la difficulté dans l'implémentation réelle de l'algorithme provient du fait qu'on peut se faire préempter pendant qu'on est en train de modifier ces paramètres, et qu'il faut toujours être paré à ce que cela arrive. L'annexe B.2 présente l'algorithme complet.

**Usage** Le rollforward lock peut s'utiliser pour implémenter une section critique, comme le fait un spinlock dans le noyau d'un OS. Il y a cependant quelques limitations :

1. On ne peut se saisir que d'un verrou simultanément, puisque la programmable préemption ne peut modifier la valeur que d'un seul lock.
2. Un thread ne peut prendre un verrou qu'une seule fois (pas de verrou "récursif")
3. S'il y a un appel de service dans la section critique, elle ne peut pas être continuée sans mécanisme additionnel.

Les cas 1. et 2. ne sont pas des problèmes en pratiques. Le rollforward lock n'est pas un sémaphore, il est donc souvent inutile d'en avoir plusieurs. De même il est inutile de se saisir d'un lock plusieurs fois (même si ce cas pourrait éventuellement être implémenté). Le rollforwardlock est adapté pour des sections critiques extrêmement courtes, où ce genre de problèmes ne survient pas. Il faut le voir comme un spinlock

```
void rollforward_lock(void *lock_addr, void (*critical_section)(void))
{
  start :
  return_upcall = &upcall_fun ;
  retry :
  if( try_take_lock())
    {
      critical_section() ;
      release_lock() ;
      goto end ;
    }
  else( try_continue_execution())
    {
      continue_execution_autre_thread() ;
    }
  else goto retry ;

  end :
  return_upcall = normal_resume_upcall ;
  return ;

  upcall_fun :
  // Si on a réussi a avoir le lock (sans forcément
  // avoir terminé l'execution de la section critique)
  if( lock_aquired())
  {
    retry2 :
    // La section critique a terminé, on peut quitter
    if( lock_is_free() || lock_held_by_other_thread())
       goto end ;
    if( lock_is_held())
       try_continue_execution() ;
    goto retry2 ;
  }
  else goto retry ;
}
```

Listing 4.3 – Pseudo-code du rollforward lock

en multiprocesseur, mais qui résout le problème de la préemption dans la section critique.

Le cas 3. peut parfois poser problème ; nous nous y attachons en section 4.5.

**Évaluation** L'évaluation est faite dans le cas particulier des services partagés d'Anaxagoros, qui demande une commutation de pile (§ 5.2.2), ce qui rajoute une

certain overhead. Mais même ainsi, l'overhead de ce verrou dans le cas favorable est faible (35 cycles). Une évaluation plus complète dans un contexte de programme normal reste à faire, ainsi qu'une évaluation du temps nécessaire pour continuer l'exécution d'un autre thread.

**Conclusion** Dans le cas d'applications normales, le rollforward lock est extrêmement intéressant : très efficace, non bloquant donc utilisable pour le temps réel, il s'utilise comme le modèle simple du spinlock avec masquage des interruptions dans le noyau. On peut le voir comme une implémentation légère et plus générale de l'héritage de priorité pour les ordonnancements à priorité fixe.

Ce lock introduit une légère entorse au principe de ressources CPU prévisibles (§ 3.1.2.2), puisqu'un peu de temps CPU est perdu pour débloquer la section critique. Le service doit donc s'arranger pour que les sections critiques soient petites et que ce temps perdu ne soit pas significatif (et pris en compte dans l'évaluation du temps passé dans le service).

Il est limité en ce qu'il ne permet pas d'appels de services à l'intérieur de la section critique. Également, nous n'avons pas implémenté la prise de lock récursive ou la prise de plusieurs locks simultanément, mais nous n'en avons pas besoin pour les services d'Anaxagoros.

### 4.3.3 Rollback et restart

Le rollforward lock implémente un verrou avec la sémantique usuelle, et une protection contre les 4 types de problèmes de la section 4.1.1.3. Dans cette section, nous voyons comment implémenter d'autres mécanismes.

**Rollback lock** Le rollforward lock permet de débloquer la section critique en permettant à un thread de la terminer, i.e. d'aller jusqu'au `unlock`. Au contraire, le *rollback* permet de retourner en arrière, i.e. de retourner au moment du `lock`.

Le rollback nécessite de maintenir un journal des écritures qui sont faites, afin que le *recouvrement* puisse revenir en arrière. Ce journal peut être simplement une pile de paires (adresse, ancienne valeur) : avant chaque écriture à une adresse de la structure partagée, on ajoute à la pile l'adresse et l'ancienne valeur.

Ce mécanisme de synchronisation est très intéressant. Il demande un support de la part du programmeur, mais qui reste relativement simple. Surtout si on fournit une API simple pour utiliser cela (i.e. une fonction de type `void transaction_write( void *address, uint new_value)` qui remplit le journal et écrit la valeur), voire un support de la part du compilateur. Cette fonction exécuterait :

```
 void transaction_write( uint *address, uint new_value) {
   uint index = log->index ;
   log->array[index].address = address ;
   log->array[index].old_value = *address ;
   log->index++ ;
   *address = new_value ;
}
```

où `log` serait le journal lié au lock[15]. Notons que cette fonction est conçue pour que le rollback puisse intervenir sans danger à n'importe quel moment. De plus le recouvrement peut être exécuté plusieurs fois sans problème. Différentes optimisations sont possibles ; par exemple si les écritures se font toujours dans le même ordre, il est inutile de copier l'adresse ; etc.

Au niveau du désavantage, la nécessité (sans support du compilateur) pour le programmeur de modifier son programme pour utiliser l'API ; la perte de temps qu'occasionne la révocation (pour avoir entamé la transaction, puis pour l'effacer) ; la taille du journal et l'overhead CPU lorsque beaucoup d'écritures sont faites.

Nous n'avons pas eu le temps d'explorer cette option en détail, mais elle est certainement très intéressante ; et très facile à implémenter à l'aide du recoverable lock vu après.

**Restart lock** On peut implémenter une version simplifiée du rollback lock, sans maintenir de journal. Si le thread est préempté dans la section critique, il recommence tout simplement son exécution de juste avant qu'il aie pris le lock. En regardant notre catégorisation des problèmes de concurrence (§ 4.1.1.3), on s'aperçoit que ce verrou garanti contre les problèmes d'opérations simultanées et opérations retardées. On peut alors écrire le code de manière à éviter les problèmes restants d'observation des états intermédiaires et de non-terminaison, mais cela est beaucoup plus simple que d'écrire des programmes complètement lock-free. Bershad [Ber93] utilise ce lock pour pour simuler des instructions atomiques sur des processeurs qui n'en disposent pas, par exemple.

Le restart lock peut être généralisé en le revocable lock, que nous voyons ci-après.

**Sections restart** Avant d'examiner plus en détail le restart lock, on peut également observer qu'on peut faire des sections restart sans lock. Le principe est simple : on définit une certaine section de code. Si on reçoit une préemption dans une certaine section, l'exécution reprend de cet endroit.

Il y a beaucoup de moyens d'implémenter ceci à l'aide de la préemption programmable : en changeant la valeur de l'upcall de reprise `upcall_entry`, en changeant la valeur de `addr_dump`, en changeant la valeur de `addr_lock` ou de `expected_value`...

Les sections restart garantissent contre les problèmes d'opérations retardées. En monoprocesseur, elles garantissent également contre les opérations simultanées, et on peut donc implémenter l'équivalent du restart lock, mais sans lock.

En multiprocesseur, elle permet également de limiter le nombre de threads dans la section restart. S'il y a $N$ processeurs sur la machine, il ne peut y avoir que $N$ processeurs simultanément dans la section restart. On peut par exemple s'en servir pour contrer les problèmes de d'allocation dynamique dans les algorithmes lock-free : comme le nombre de threads est borné, l'allocation l'est aussi. Cela fonctionne bien puisque les algorithmes lock-free sont fait pour "réessayer" lorsqu'ils n'arrivent pas à faire leur "commit". Nous avons ainsi pu écrire une version du "small-object protocol" de Herlihy [Her90] qui se sert de cette propriété.

[15]Pour garder l'API simple, l'adresse de `log` doit être gardée dans des données thread-locales, comme dans l'UTCB

On peut également s'en servir quand on écrit nos ordonnanceurs hiérarchiques (ou même pour un ordonnanceur de thread utilisateur : si un ordonnanceur est préempté avant de pouvoir donner du temps à un thread, le contexte peut avoir été modifié, ainsi que le choix du thread à ordonnancer.

Mais l'usage le plus courant de ces sections est pour l'implémentation des locks de plus haut niveau eux-même. En effet, dès qu'on modifie un paramètre de la préemption programmable, l'exécution ne reprend plus d'où elle en était lorsqu'il y a préemption. Et il y a plusieurs paramètres à changer pour prendre des locks tels que le rollforward lock. Pour contrer le problème, tous nos algorithmes qui modifient les paramètres de la préemption programmable se placent dans des sections restart.

**Note :** Les sections critiques de [BRE92] sont différentes en ce que la dernière opération et la fin de la section restart sont simultanées. On peut également implémenter cette variante en modifiant l'upcall de reprise pour qu'il fasse une comparaison du program counter.

### 4.3.4 Le revocable lock et le recoverable lock

#### 4.3.4.1 Revocable lock

Le mécanisme de restart lock a été généralisé par Harris et Fraser sous le nom de *revocable lock* [HF05]. La différence par rapport au restart lock est que lorsque le verrou est révoqué, le thread révoqué a son program counter modifié pour exécuter une fonction prédéfinie, plutôt que de seulement recommencer. revocable lock

L'interface se constitue donc de deux fonctions :

- `void revocable_lock( lock_t lock, function_t revokee)`
- `void revocable_unlock( lock_t lock)`

**Comparaison** Notre implémentation des revocable lock présente cependant un certain nombre de différences avec celle de Harris et Fraser.

- La plus importante est qu'un lock n'est révoqué que si celui qui le détient ne fait plus aucun progrès, i.e. est préempté ou a déclenché une exception. Par opposition, l'implémentation de Harris et Fraser permettent à un thread de révoquer un lock n'importe quand. Cela a plusieurs conséquences :
  - Leur implémentation révoque des threads qui s'exécutaient correctement, donc perdra du temps à refaire un travail qui était en train de se terminer. Si la contention est forte et la section critique assez longue, l'exécution peut même être complètement bloquée. Pour éviter ce problème, Harris et Fraser ajoutent une attente active de durée bornée avant de révoquer un lock, mais cela ne marche pas dans tous les cas[16].

[16] pour éviter tous les problèmes, il faudrait que l'attente active soit toujours plus longue que le temps maximum passé dans la section critique, ce qui est dur à garantir.

  - Cette attente active est du temps perdu lorsque le thread qui détenait le verrou a été préempté ; dans notre implémentation la révocation est directe.
  - Dans notre implémentation la révocation (ni la prise de lock) ne demande pas d'appel système, donc est aussi plus rapide.

  Ces améliorations rendent le lock plus efficace.

- Notre implémentation est non-bloquante, ce qui n'est pas toujours le cas pour celle de Harris et Fraser.
- Au niveau des inconvénients, le lock occupe un mot entier plutôt qu'un seul bit. Il est cependant possible d'ajouter des masques au mécanisme de préemption programmable pour permettre la même chose.
- Dernier inconvénient, notre implémentation requiert un mécanisme spécial, la préemption programmable ; cependant l'implémentation de Harris et Fraser utilise aussi des mécanismes OS pour implémenter la révocation.

#### 4.3.4.2 Recoverable lock

recoverable lock

Enfin, nous avons généralisé notre implémentation en un verrou que nous appelons *recoverable lock*. Le recoverable lock permet au thread qui fait la révocation d'effectuer une fonction dite de *recouvrement*, qui permet au thread de rétablir un état sain avant d'exécuter sa section critique.

L'interface se constitue donc de deux fonctions :

- `void recoverable_lock( lock, revoked_fun, recover_fun) ;`
- `void recoverable_unlock( lock) ;`

Notons que la fonction de recouvrement peut elle-même être préemptée sans préavis. Notons aussi qu'on peut, dans la section critique, regarder si l'état est sain, et faire le recouvrement depuis là. C'est l'approche utilisée par les revocable locks de [HF05]. Cependant elle souffre de plusieurs problèmes :

- il peut être impossible, ou trop long, de détecter les incohérences par tests sur les données. Par exemple, si la section critique doit insérer un élément dans une liste puis incrémenter un compteur "nombre d'élément" de cette liste ;
- ces tests sont spécifiques à une utilisation ; tandis que le *recoverable lock* permet d'avoir des stratégies de recouvrement génériques, comme le rollforward lock, le *rollback*, le *restart* ou même le revocable lock. Tous ces locks sont implémentés en tant que spécialisations du recoverable lock ;
- ces tests sont des surplus, puisque lorsqu'un thread en révoque un autre, il le "sait" sans avoir besoin de test supplémentaire. Ainsi le recoverable lock permet d'éviter ces tests inutiles.

Les exemples donnés par Harris et Fraser dans [HF05] peuvent être implémentés plus simplement (et efficacement) à l'aide du recoverable lock. En particulier pour le two-handed emulation de Greenwald [GC96]. Aussi préférerons nous ce dernier lock.

Notons que le recoverable lock peut s'implémenter sans la préemption programmable, et constituerait une modification simple de l'implémentation de [HF05].

Notons enfin que la fonction `revoked_fun`, en argument du verrou, est simple, car tout le recouvrement est fait par la fonction `recover_fun`. En pratique, `revoked_fun` ne fait que retourner au début de la section critique ou à la fin, selon que l'opération voulue ait été complétée ou non.

#### 4.3.4.3 Implémentation

Il s'agit d'une simple généralisation du code du rollforward lock présenté Listing 4.3 page 170.

#### 4.3.4.4 Usage

Le verrou est utilisé pour protéger un ensemble de données. Comme le verrou peut être révoqué à n'importe quel moment, il faut que les invariants sur les données soient vérifiés à tout moment.

**Implémentation de locks avec stratégie de recouvrement générique** Le recoverable lock permet d'implémenter des locks avec une stratégie de recouvrement générique. Par exemple, le rollforward lock s'obtient approximativement en ayant pour fonction de recouvrement le rechargement des registres du thread préempté. Le *rollback* peut s'obtenir en utilisant une fonction de recouvrement qui restaure les valeurs dans le journal ; etc...

**Utilisation directe** On peut donc utiliser les recoverable lock directement, comme on le ferait des revocable lock. Ce verrou protège contre les opérations simultanées et retardées(§ 4.1.1.3). Il protège également contre les problèmes de consistances des caches (§ 4.1.1.4), qui complexifient cette programmation pour le programmeur et obligent à l'utilisation de barrières coûteuses[17].

Pour l'utiliser pour protéger une donnée $d$, il faut vérifier une seule propriété : *après chaque écriture, tous les invariants sur d sont encore vérifiés.* L'idée est que les opérations gardent l'état des données continuellement "sain", i.e. cohérent. Cela impose des contraintes sur le code, mais qui sont bien plus simple à respecter (car l'exécution dans le lock est séquentielle) que dans le cadre général d'applications parallèles. Cela a été détaillé en section 4.1.3.

Certains invariants ne peuvent pas être assurées de manière continue, dès qu'il y a une équivalence à assurer. Ce problème peut être contourné, car il existe des solutions universelles pour résoudre ces problèmes (e.g. utilisation du rollback lock). Mais l'avantage d'une solution adaptée est d'être plus efficace pour un problème donné.

[17] Toutefois, il faut prendre garde à ce que le compilateur ne réordonne ni ne supprime des écritures.

**Exemple : la vérification** Ce verrou peut être utilisé pour les lectures, ou lorsqu'on a peu de modifications à faire.

Par exemple, beaucoup de code peut être amené à vérifier une condition sur une ressource avant de faire une action (comme modifier la ressource). Un code concurrent peut poser problème, car la condition pourrait ne plus être vérifiée. Le revocable lock, en assurant la séquentialité des traitements, permet de pallier à ce problème.

Par exemple, on peut utiliser ce lock pour faire une sorte de semi `NCAS`, i.e. faire atomiquement :

```
if( o1 == v1 && o2 == v2 && ... && on == vn) *addr = value ;
```

Cela permet également de résoudre les problèmes de séquencement correct : on vérifie qu'on est dans le bon état de l'automate avant de faire la transition.

**Autres utilisations** Harris et Fraser [HF05] donnent d'autres exemples d'utilisation ; pour implémenter de la software transactional memory ou le "two-hand emulation" de Greenwald [GC96].

On peut également utiliser le revocable lock pour empêcher les problèmes de livelock dans les algorithmes obstruction-free, qui sont simple à écrire selon Herlihy [HLM03], et fournissent donc une application pratique.

## 4.4 Travaux en rapport

La plupart des travaux concernés sur le problème de la préemption lors de section critiques se sont concentrés sur la méthode du sursis d'exécution, que nous avons rejetée. Certains ont un mécanisme analogue à la préemption programmable, et certains non.

### Mécanismes similaires à la préemption programmable

**Scheduler activations** L'upcall de reprise fait ressembler notre mécanisme aux scheduler activations [ABLL92]. La différence principale est que les scheduler activations utilisent un upcall de préemption. Même si ce mécanisme n'est pas implémenté par un sursis d'exécution, il souffre de nombreux problèmes : cet upcall préempte un autre processeur, ce qui perturbe sa décision d'ordonnancement et cause un certain overhead. D'autant plus que le recouvrement des sections critiques est fait de manière immédiate ; dans Anaxagoros le recouvrement est fait de manière paresseuse, ce qui réduit des besoins de recouvrement et réduit les interférences d'exécution de threads non reliés.

L'upcall de préemption leur retire le besoin de la zone de non-sauvegarde sur préemption, ainsi que le besoin de stocker les registres en espace utilisateur (ces derniers sont passés en paramètre de l'upcall). Enfin, il semble plus facile d'implémenter l'user-level scheduling avec, car cet upcall permet la mise à jour de la liste des threads prêts. Nous pensons cependant que cette mise à jour peut se faire de manière paresseuse au moment de la consultation de ces listes.

Une autre différence est le traitement des appels systèmes bloquants. Nous n'implémentons pas d'appel système bloquants ; notre plan est de pouvoir émuler appels systèmes bloquants et non-bloquants par le biais d'un mécanisme de notification, permettant au service de prévenir les clients quand leurs données sont prêtes. Ces notifications permettent également de mettre un thread dans l'état « prêt » en prévenant son ordonnanceur.

Enfin, la dernière différence concerne la motivation : les scheduler activations ont pour but d'améliorer les performances de l'ordonnancement de thread utilisateur sur multiprocesseur (M-to-N scheduling [Can96]). La préemption programmable a pour but de créer un mécanisme de synchronisation efficace et non-bloquant en évitant de modifier l'ordonnancement ; en particulier pour les services, pour lequel l'ordonnancement est 1-to-1. Le fait qu'on puisse faire de l'ordonnancement de thread utilisateur avec n'est qu'un bonus agréable.

**Psyche** Le mécanisme de threads utilisateur de première classe de Psyche [MSLM91], également conçu pour améliorer l'ordonnancement $M$-to-$N$, est similaire au notre en ce qu'il utilise de la mémoire partagée entre le noyau et le thread pour guider ce que doit faire le noyau. Il diffère du notre en ce qu'il utilise l'approche problématique du "sursis d'exécution" pour résoudre les problèmes de préemption dans les sections critiques. Il n'y a pas non plus d'upcall de reprise.

**Exokernel** Les implémentations d'exokernels [EKJO95, KEG+97] ont des similitudes avec les nôtres. La sauvegarde et le rechargement du contexte est faite en espace utilisateur, au moins pour l'implémentation sur Alpha [EKJO95, § 5.3]. Mais la sauvegarde des registres se fait par un upcall [EKJO95, § 5.1.1], ce qui implique un aller-retour noyau/application supplémentaire. Cet upcall peut terminer les sections critiques avec un sursis d'exécution [EKJO95, § 5.1.1], ce qui pose également problème (§ 4.1.2.1)

### Mécanismes de synchronisation

**Symunix** Symunix[ELS88] ne permet pas la gestion du contexte processeur par l'espace utilisateur, mais présente le mécanisme de sursis d'exécution, qui émule le masquage des interruptions en espace utilisateur.

**Restartable atomic sequences** Les restartable atomic sequences [BRE92] permettent de reprendre l'exécution en arrière lorsque le program counter est à une certaine valeur ; cela permet d'éviter l'utilisation d'opérations read-modify-write sur monoprocesseur (de type CAS), qui sont coûteuses.

La zone de non sauvegarde sur préemption a des similitudes avec ce mécanisme, puisque la comparaison se fait également par program counter, et que la reprise se fait au précédent point de sauvegarde si l'upcall de reprise est paramétré pour reprendre l'exécution. Mais cette implémentation n'est pas optimisée, et ne permet pas de faire une écriture en sortant de la section atomiquement, ce qui limite son utilité. On peut cependant implémenter les vraies restartable atomic sequences à l'aide du mécanisme entier de préemption programmable, par exemple en modifiant

l'upcall de préemption pour qu'il reprenne l'exécution conditionellement à la valeur du program counter lors de la préemption.

Notons que l'implémentation des recoverable lock/rollforward lock en monoprocesseur n'a pas besoin d'utiliser d'opération read-modify-write atomique, ce qui est l'optimisation permise par les rollforward atomic sequences.

**Rollforward lock** D'autres papiers ont exploré l'usage du rollforward lock. Le rollforward lock est mentionné dans [Ber93], mais non implémenté compte à cause des problèmes de défaut de page et de la difficulté d'une implémentation sûre.

Mosberger et al. [DM96] ont implémenté un rollforward lock. Leur implémentation diffère de la notre essentiellement en ce qu'ils terminent l'exécution par l'interruption qui provoque la préemption (i.e. utilisent un sursis d'exécution). Cela pose des problèmes de sécurité (e.g. la routine d'interruption doit scanner le code pour vérifier qu'il termine et ne permet pas des opérations interdites), augmente le temps de préemption, et ne marche que pour le monoprocesseur.

Le mécanisme de « donation volontaire » de Ford et al. [FS96, § 3.4] permet de donner du temps à un autre thread pour qu'il termine l'exécution de la section critique ; cela étend le mécanisme classique d'héritage de priorité. La différence principale avec notre mécanisme est que nous n'avons pas besoin de faire intervenir d'ordonnanceur, ce qui est plus efficace et facilite l'écriture des ordonnanceurs.

Le mécanisme de futex de Linux [FRK02] permet d'éviter d'avoir à faire des appels systèmes dans le cas où le lock n'est pas pris. Mais pour débloquer un lock pris, il est toujours nécessaire de faire plusieurs appels systèmes.

**Revocable locks** Les revocable lock de Harris et Fraser [HF05] sont implémentés différemment. Ils ne savent pas quand un thread est préempté dans une section critique et par conséquence, font une attente active dans l'espérance qu'il en sortira, ce qui diminue les chances de révoquer un thread actif sans les éliminer. De plus la révocation ne se fait pas en temps constant, et a besoin d'un appel système coûteux.

Nos recoverable lock implémentent donc une interface améliorée (avec un test générique pour savoir s'il y a eu révocation) de manière plus légère.

## 4.5 Conclusion et travaux futurs

### Conclusion

Écrire des services indépendants de toute politique et basés sur le prêt de ressource pose des problèmes concrets : les services sont multithreadés, les ordonnanceurs sont incontrôlables voire malicieux, et la mémoire rattachée aux threads peut disparaître à tout moment. De plus, il est impossible d'utiliser tel quels les mécanismes de synchronisation habituels que sont le sleeplock et le spinlock. Cela complexifie la résolution des problèmes de cohérence.

Dans le noyau, la possibilité de masquer les interruptions  facilite la résolution du problème, mais cette possibilité n'est pas tolérable pour des services en espace

utilisateur sans confiance. Nous avons donc développé un ensemble de primitives de synchronisations pour ce cas telles que :

- l'implémentation du noyau en soit peu affectée, pour que celui-ci reste simple et sûr ;
- la synchronisation ne demande pas la coopération des ordonnanceurs, qui peuvent être malicieux ;
- les sections critiques soient non-bloquante, ce qui garantit la vivacité du système ; la perturbation dans l'exécution des threads est minimisée ;
- la synchronisation fonctionne sur des machines multicœur, qui seront standard dans le futur, y compris pour les systèmes temps réel.

Le résultat en est un mécanisme et une implémentation simple, élégante et efficace de sections critiques non-bloquantes aux propriétés différentes ; qui sont utiles pour l'ensemble des applications du système, et pas seulement les services. Nous avons donc des techniques de résolution des problèmes de synchronisation qui permettent de minimiser la perturbation des décisions de l'ordonnancement, en monoprocesseur et en SMP.

Leur utilisation systématique nous permet de rejoindre la petite famille des systèmes d'exploitations qui font seulement de la synchronisation non bloquantes ; et cela sans utiliser DCAS (comme [Mas92, GC96]) ni utiliser d'ordonnancement spécifique [SWH05].

### Travaux futurs

Bien que les verrous non-bloquants implémentés dans cette section soient une réelle avancée, il reste néanmoins plusieurs problèmes à régler.

**Appels de service dans une section critique** Les verrous que nous avons implémentés permettent de synchroniser des données pour plusieurs threads dans le même espace d'adressage. Mais nous n'adressons pas le problème de la synchronisation entre plusieurs espaces d'adressages.

Des applications clientes peuvent se synchroniser de manière différentes, en faisant notamment intervenir leur(s) ordonnanceur(s). Les services ne peuvent pas faire cela, car ils doivent être indépendant des décisions d'ordonnancements (§ 3.1.1). Concrètement, le problème qui se pose pour les services est celui de la synchronisation lorsqu'il y a un appel à un autre service dans la section critique. Pour l'instant, nous évitons les appels de service récursifs (qui sont de toute manière lent et une introduction de complexité dans le service), mais c'est parfois un peu contraignant (en général, on en a besoin pour faire des opérations sur la mémoire virtuelle).

Nous entrevoyons deux solutions différentes à ce problème :

- le *CPU inheritance*. Ce moyen, décrit par Ford et Susarla [FS96], est une sorte de généralisation de l'héritage de priorité indépendant des politiques d'ordonnancement. L'idée consiste à faire exécuter la section critique par un

thread propre au service, auquel les threads prêtés donnent du temps pour terminer l'exécution. Il s'agit donc d'une sorte de rollforward lock, mais au niveau des threads noyau ;

- l'*annulation prompte*. Cette technique consisterait à permettre à un thread dans un service d'annuler un appel fait par un autre thread depuis le même service. Cette annulation ferait partie du recouvrement du recoverable lock. Même si ce mécanisme est relativement simple à implémenter (les services étant prêts à tout arrêt dans leur exécution), il reste des problèmes à régler, comme l'exécution de la préemption programmable quand un thread est préempté hors du service.

Enfin, une dernière solution serait de spécifier que certains appels de services, vers le noyau, s'exécutent de manière atomique ; les besoins d'appels récursifs étant plutôt rares et destinés au noyau (généralement utilisés pour installer des capacités ou des mappings mémoire), ce dernier mécanisme peut se révéler être suffisant. Nous avons trouvé que la majorité des communications entre services peut se faire par mémoire partagée ; par exemple les communications entre service de mécanisme et de politique (§ 3.2.2.2).

**Évaluation des locks implémentés** L'évaluation de l'efficacité des verrous implémentés, leur scalabilité, et de leur comparaison, serait intéressante. Nos verrous sont qualitativement efficaces, les microbenchmarks sont très bons, mais une évaluation quantitative est toujours intéressante. Cela nécessiterait de porter un macro-benchmark et de l'exécuter sur une machine disposant d'un nombre important de coeurs.

Chapitre 5

# Méthode de conception des services

Nous avons vu au chapitre 3 comment structurer le système en différents services, comment se déroulait la communication avec un service, et comment structurer les services pour respecter les principes énoncés, en particulier l'indépendance des politiques d'allocation. Ce chapitre complète la méthodologie d'écriture du système, en expliquant comment écrire des services qui suivent les principes édictés en section 3.1. En particulier, il énonce comment éviter et résoudre les problèmes de synchronisations dûs au multithreading tout en restant indépendant des politiques d'allocation.

**Contributions** Les contributions principales de ce chapitre sont :

- La possibilité pour un OS de présumer de la non-concurrence de différentes applications, et ses applications (spinlocks non tournant (§ 5.2.1.1), délégation de la gestion de la concurrence (§ 5.3.2) et exportation des synchronisations (§ 5.4.6) ;
- un schéma de synchronisation important pour l'écriture de systèmes d'exploitations dynamiques parallèle, le *use/destroy pattern* ;
- les principes et techniques de minimisation des problèmes de concurrences (§ 5.3 et 5.4).

## 5.1 Méthode de conception des services

### 5.1.1 Particularités des services d'Anaxagoros

Les services conçus pour suivre les principes méthodologiques développés dans cette thèse ont un certain nombre de points notables, certains contraignants pour la conception et l'écriture du service, d'autres avantageux, que nous récapitulons ici.

#### 5.1.1.1 Gestion de la mémoire

Nous avons choisi (§ 3.5.2.1) d'allouer la mémoire des services statiquement, pour des raisons qui ont été développées précédemment. Cela cause un inconvénient : les services sont stateless dans le sens où on ne stocke pas l'état des requêtes par thread entre plusieurs appels. Mais cela a également beaucoup d'avantages, en terme de simplification du code : identification des partages sur les données, absence d'aliasing, pas de gestion de la mémoire, etc.

#### 5.1.1.2 Indépendance du service et de l'ordonnancement et parallélisme subi

Le service est indépendant de l'ordonnancement des threads des clients qui lui soumettent des requêtes. Cela occasionne deux inconvénients :

- Le service ne doit pas perturber l'ordonnancement, ni même l'exécution, des threads prêtés par les clients ; i.e. le temps passé à faire des requêtes devrait être globalement prévisible ;
- Le service n'a pas de contrôle sur l'ordonnancement des threads, et ceux-ci peuvent même être ordonnancé de manière adverse.

Les services sont des programmes parallèles, non parce qu'on veut en obtenir de bonnes performances, mais parce que c'est une conséquence directe de l'implémentation du principe d'indépendance de l'ordonnancement par le prêt de temps CPU au service. Le parallélisme est donc *subi*, et non choisi. En particulier, nous ne sommes au courant d'aucun programme réel multithreadé dont les threads sont ordonnancés par des entités extérieures sans confiance ; cette notion d'ordonnanceur adverse ne peut être rencontrée que pour des évaluations au pire cas dans des travaux théoriques (e.g. [Her90]).

Les programmes multithreadés standards font généralement soit du calcul parallèle, soit utilisent un thread par tâche distincte (e.g. une GUI, ou un serveur réseau) ; ici nous avons un ensemble de threads qui peuvent concourir pour faire une tâche parallèlement sans que cela n'aie vraiment de sens. Ce type de problème est courant dans les systèmes d'exploitation ; Ryzhyk [RCKH09] identifie ainsi de nombreux bugs dans l'initialisation parallèle de drivers dans Linux.

#### 5.1.1.3 Fonctionnement d'un service, séquencement et automates

Le rôle d'un service est de partager des ressources entre plusieurs clients. Les clients n'accèdent pas à la ressource directement, mais passent par le service en lui faisant des requêtes. Le but de ces requêtes n'est pas de faire un calcul par le service (ce calcul peut tout aussi bien être fait par le client), mais de modifier l'état d'une ressource servie par le service.

Le client ne peut pas modifier l'état des ressources n'importe comment, sinon le service serait inutile. Le but du service est justement d'assurer certaines propriétés "négatives" sur les ressources [Rus89]. Ces propriétés peuvent prendre deux formes :

- la propriété peut spécifier un ordre sur le changement des états, ce qu'on peut spécifier à l'aide d'un automate ;
- la propriété peut être une relation sur l'état mémoire de la ressource, i.e. assure que l'état est cohérent.

Cette spécification sous la forme d'un automate se retrouve dans plusieurs travaux. Ainsi, Denning [Den76, p. 19] annonce qu' "un contrôle des ressources sûr est assuré lorsque les unité d'un certain type de ressource sont forcées à se conformer à un diagramme état-transition bien défini". Plus récemment, la modélisation des interactions OS/device driver dans Dingo [RCKH09] suit également la forme d'un automate.

Le service est donc un programme parallèle dont les actions doivent suivre un ordre défini (ce qu'on appelle être *correctement séquencé*) en plus de devoir travailler sur des états cohérents.

Au niveau des avantages, les requêtes sont faites essentiellement pour modifier l'état interne du service, et renvoient peu de résultats. Cela d'autant plus qu'un maximum d'état est exporté vers le client (§ 5.3). Cette observation est exploitée de plusieurs façons, par exemple pour l'utilisation du rollforward lock dans les services (§ 5.2.2) ou pour le multicall (§ 3.3.3.2).

#### 5.1.1.4 Minimisation, sécurité du service et preuve de programme

Les services partagent des ressources entre différents clients qui ne se font pas confiance. Les clients ont en général confiance dans les services qu'ils utilisent, car ils ont besoin des ressources servies. Les services sont bien souvent les seuls points communs entre deux programmes indépendants, et donc les seuls points d'attaques potentiels pour un programme malveillant. Il faut donc bien les sécuriser, et pour cela il faut minimiser leur taille (principe 2.2.2.6) : i.e. en faire faire un maximum par les application elle-mêmes. En minimisant les services, on diminue la probabilité d'occurrence de bugs [Ber07], et on facilite les preuves de sûreté.

Un dernier avantage d'avoir des services de taille minimale est que cela permet de relier directement l'implémentation aux spécifications (e.g. la relation entre les invariants de l'implémentation et les fonctionnalités et les propriétés de sécurité du service). Nul besoin de fonctions génériques qui peuvent faire plus que ce qui leur est demandé, ce qui se retrouve fréquemment dans des programmes plus gros. On peut ainsi optimiser le programme. Par exemple, c'est parce que nos services sont petits que leur rôle est bien compris et qu'on peut se permettre des choses comme l'affaiblissement des invariants (§ 4.1.3.3), la délégation de la gestion de la cohérence (§ 5.3.2), ou l'utilisation de primitives de synchronisation "exotiques" comme le revocable lock. Enfin, cette relation directe entre l'implémentation et les spécifications est utile pour la preuve de programme.

Il est important de comprendre que la preuve du service concerne seulement les propriétés de sécurité que le service doit assurer. En particulier, il est souvent légitime que par suite d'une mauvaise utilisation par l'espace utilisateur (e.g. appels simultanés), le noyau rende certaines données incohérentes : ceci ne constitue pas un comportement incorrect du noyau.

#### 5.1.1.5 Restrictions sur les primitives de synchronisation et minimisation des problèmes de concurrence

Dans le chapitre précédent, nous avons vu que les primitives de synchronisation habituelles ne pouvaient pas être utilisées quand on veut adhérer au principe d'indépendance des politiques. Les primitives que nous avons développées fournissent une solution, mais nous allons voir qu'il y a des limitations additionnelles dans leur utilisation. De plus, elles ne permettent d'acquérir qu'un seul lock simultanément. Notons qu'avec un seul lock, on peut résoudre tous les problèmes de concurrence ; mais cela peut poser problème pour faire du verrouillage à grain fin.

Notre approche est de minimiser les problèmes de concurrence. Quand on élimine un problème de concurrence, il n'y a plus besoin d'employer de primitive de synchronisation, et on a un passage à l'échelle excellent. La section 5.3 explique cela en détail.

**Conclusion : globalement simple, localement complexe**

Les difficultés principales concernant l'écriture des services sont les problèmes liés à la concurrence et le besoin en synchronisation. Le fait d'avoir des services stateless pose également problème.

L'approche que nous avons suivie peut s'appeler *globalement simple, localement complexe*. L'allocation des ressources est extrêmement simplifiée, les services ne bloquent pas de manière imprévue, les services sont indépendants, etc., tout cela simplifie la vision globale du système. Mais le prix à payer pour cette simplicité globale est une complexité accrue au niveau de l'implémentation des services. L'interface d'accès au service reste toutefois simple.

Cette approche est à l'opposé de l'approche actuelle d'écriture des OS, où la simplicité du code prime [Gab91], même si le système en devient globalement complexe (e.g. allocation des ressources imprévisible).

Dans le reste de ce chapitre, nous voyons comment résoudre cette complexité accrue des services.

### 5.1.2 Méthode pour l'écriture de service

Nous proposons la méthode suivante pour l'écriture d'un service qui partage un type de ressources :

1. Imaginer les fonctionnalités minimums que doit fournir le service pour permettre de partager les ressources de manière sécurisée ;

2. En déduire une interface fournie par le service, les structures de données utilisées par ce service, et les relations entre ces données, à l'aide des sections 5.3 et 5.4 ;

3. En implémenter une première version. Pour ne pas avoir à se préoccuper des problèmes de synchronisation dans un premier temps, on peut n'avoir qu'un seul client, ou développer le service en lui permettant l'accès aux instructions privilégiées comme le masquage des interruptions. En parallèle, implémenter une librairie client qui servira également pour le test.

4. Améliorer l'implémentation en analysant finement les problème de concurrence qui se posent, et en utilisant les primitives pour l'espace utilisateur que nous fournissons en section 5.2. On peut également adapter les structures de données pour faciliter ces synchronisations.

L'implémentation du système de capacité (qui peut être vu comme un « service » du noyau), présentée en annexe A, suit cette démarche.

Ces étapes ne sont pas figées, et permettent de nombreux retours en arrière. Les services étant petits, ils peuvent facilement tenir intégralement dans la tête du programmeur, ce qui facilite grandement sa conception ou modification agile.

Il serait intéressant, dans des travaux futurs, de pouvoir intégrer plus facilement conception du service, preuve, et synchronisation dans un environnement de développement qui aiderait le développeur ; par exemple en lui permettant d'écrire des invariants qui seraient vérifiés automatiquement. Les services d'Anaxagoros sont en général suffisamment simples pour qu'on puisse espérer que la preuve ne soit pas trop difficile (même si l'exécution dans les services est parallèle), et ne demande pas ou peu d'interactions avec le programmeur.

Les sections suivantes détaillent des méthodes et principes pour la conception des services.

## 5.2 Synchronisation dans les services

Nous montrons maintenant la manière dont les services peuvent utiliser différents mécanismes, dont ceux présenté ci-dessus, pour résoudre leurs problèmes de synchronisation.

### 5.2.1 Présomption de non-concurrence

#### 5.2.1.1 Présentation

**Parallélisme subi** Comme le parallélisme est subi (§ 5.1.1.2), il est fréquent d'avoir à se protéger contre des opérations pour lesquelles les faire en parallèle n'a ni intérêt ni sens. Citons par exemple, l'initialisation parallèle d'une ressource, ou l'écriture simultanée de deux capacités au même endroit. Notons que ces problèmes sont présents dans de nombreux systèmes d'exploitations (cf l'analyse des fautes de concurrence par Ryzhyk et al. [RCKH09]).

**Indéterminisme** Si plusieurs threads $A$ et $B$ écrivent simultanément dans la même structure de donnée, le premier problème, le plus évident, est le risque d'avoir des données incohérentes. La plupart des systèmes utilisent des mutex locks à ces endroits, pour se prémunir de ce problème. Mais cela ne résout pas le second problème, qui est l'*indéterminisme*.

Le mutex permet de séquentialiser les écritures, mais pas de déterminer dans quel ordre ces écritures sont faites. Il y a indéterminisme quand la suite d'évènement "écriture par $A$" suivi de "écriture par $B$" est différente de la suite "écriture par $B$"

suivi de "écriture par $A$". La valeur contenue dans l'emplacement mémoire dépend de cet ordre, qui est quasi-aléatoire.

Notons que les tentatives d'écritures simultanées ne sont pas forcément indéterministes. L'incrément d'un compteur sur plusieurs mots en utilisant un lock, par exemple, ne l'est pas, puisque le résultat est le même quelque soit les opérations. Mais il y a indéterminisme dès que dans les écritures faites l'une d'elles est *absolue*, que nous définissons comme "la valeur écrite est indépendante des valeurs précédentes de la structure". En d'autres termes, quand une écriture peut être faite sans avoir à faire de lecture sur les données protégée par le spinlock.

**Spinlocks non-tournants** On peut déjà se servir de cette observation pour supprimer la contention dans certain spinlocks. Si un thread $A$ souhaite faire une écriture et que le verrou est détenu par un thread $B$, il peut juste quitter : cela simule l'exécution séquentielle valide où $A$ a écrit ses valeurs juste avant que $B$ n'écrive les siennes, et où personne n'a eu le temps de lire les valeurs de $A$. Il n'y a donc plus besoin de faire d'attente active, donc plus de contention.

Notons que cette solution suit le concept de linéarizabilité. Notons également que nous n'avons pas décrit comment faire les lectures.

Mais cela ne fonctionne que si les écritures sont toutes absolues (e.g. on ne peut pas le faire si la structure mise à jour par $A$ et $B$ contient un compteur à incrémenter). Nous allons cependant étendre cette technique.

**Pertinence d'écritures indéterministes** Lorsque les écritures sont indéterministes, il y a manifestement un problème. Par exemple, s'il y a écriture simultanée de deux capacités au même endroit, même si les capacités sont intègres car on les fait écrire atomiquement, elles ne peuvent pas vraiment être utilisées car le client ne sait pas quel service il va contacter. De manière générale, les données manipulées sont aléatoires ce qui pose des problèmes d'utilisation. Ainsi, un système bien construit ne devrait pas faire d'écritures indéterministes.

**Détection d'écritures concurrentes** Notre approche est alors celle-ci : plutôt que de s'assurer de l'atomicité des écritures, ce qui laisse le problème de l'indéterminisme, *nous détectons lorsqu'une écriture est indéterministe et retournons une erreur* (`ECONCURRENCY` dans le code).

Pour éviter de remonter une erreur quand une opération est permise (faux positifs), nous concevons le service de manière à ce que la responsabilité de toute écriture indéterministe puisse être remontée aux clients. Pour qu'il y ait erreur, il faut que les clients aient fait des appels de services concurrents avec les même paramètres alors que cela est interdit par le service. C'est la raison pour laquelle nous appelons cette technique *présomption de non-concurrence* : nous faisons l'hypothèse que les programmes bien écrits ne font pas certaines opérations de manière concurrente, et nous vérifions cette hypothèse a posteriori.

Notons que pour ne pas remonter de faux-positif, il faut en particulier du verrouillage à grain fin : par exemple, on ne peut pas retourner `ECONCURRENCY` parce

qu'on utilise un même verrou pour protéger deux opérations sur des emplacements mémoires indépendants.

Cette technique est également étendue dans le principe de délégation de la gestion de concurrence (§ 5.3.2), où il n'y a plus de vérification, lorsque le non-respect des invariants ne pose pas de problème de sécurité.

Il faut faire attention de ne pas interdire toutes les opérations concurrentes. En particulier, les lectures concurrentes devraient être autorisées, ainsi que les écritures concurrentes déterministes. Enfin, il faut autoriser les cas où une écriture est "prioritaire" sur une autre, comme le nettoyage d'une ressource est prioritaire sur son utilisation, car cela permet de forcer la révocation d'une ressource. Le use/destroy lock (§ 5.2.4) implémente ceci.

### 5.2.2 Le rollforward lock

**Le problème** L'implémentation du rollforward lock est performante et peut être utilisée comme un verrou normal par des programmes utilisateurs classiques, mais pas directement par les services. Le problème est que lorsqu'une section critique d'un thread $T_1$ doit être continuée, la mémoire associée à ce thread n'est plus disponible : $T_1$ peut avoir été détruit, ou sa mémoire peut ne pas être accessible quand il n'est pas ordonnancé[1]. En particulier, la pile utilisée par ce thread et ses registres ne sont pas disponibles, et l'exécution ne peut pas être continuée.

**Solution** La solution est simple : il s'agit de réserver, pour chaque verrou, une petite zone de mémoire qui servira de pile pour l'exécution de la section critique ; et qui servira également à sauvegarder les registres lorsque le processeur est préempté. La Figure 5.1 illustre comment cela est réalisé.

Ce mécanisme utilise la préemption programmable (§ 4.2.3), en particulier la possibilité de choisir où ses registres CPU sont placés (c'est l'utilité du paramètre `addr_dump`). Il est non-bloquant, et son overhead est très faible (aucun appel système, même en cas de contention).

**Inconvénients** Dans le cas, même rare, où un thread $T_1$ est continué par un autre thread $T_2$, la mémoire locale de $T_1$ est inaccessible. Cela signifie qu'on ne peut ni écrire, ni lire dans la pile de $T_1$ ; donc on ne peut pas retourner de valeur depuis la section critique.

De plus, tous les paramètres doivent être passés par registres, ce qui limite leur nombre (surtout sur IA-32, où le nombre de registres est très faible)[2]. Le rollforward lock agit donc comme une super-instruction read-modify-write atomique (e.g. comme CAS), mais qui ne peut pas renvoyer de données. Le rollback lock et le revocable lock ne souffrent pas de ce problème.

[1] comme dans notre implémentation IA32 actuelle (§ 3.4.2.2)

[2] Il est néanmoins possible d'augmenter ce nombre de paramètres passé si nécessaire. Un algorithme possible consiste à écrire les paramètres supplémentaire dans de la mémoire propre par CPU avant de rentrer dans la section critique. Nous n'avons pas développé cette solution car nous n'en avons pas rencontré le besoin.

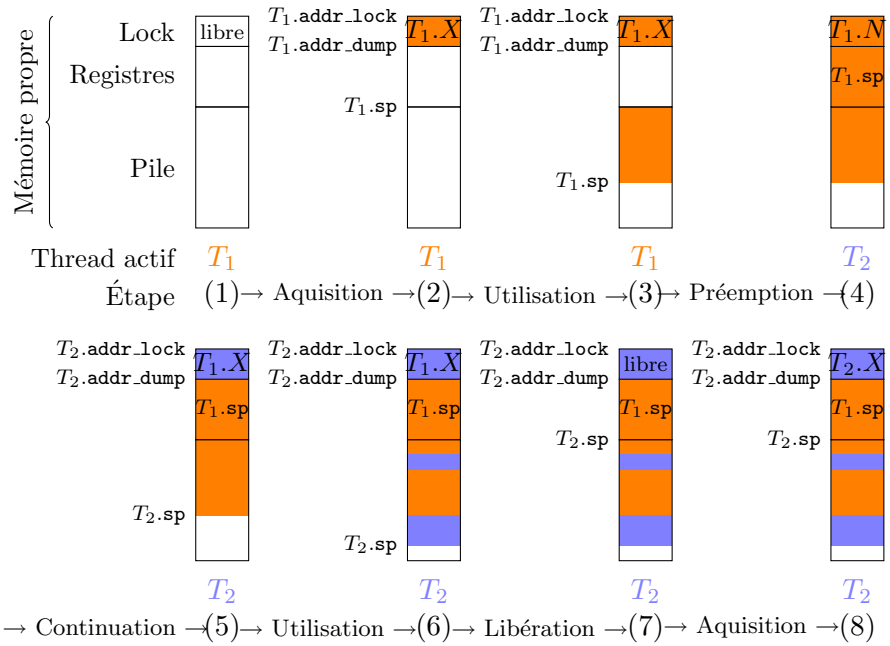



FIG. 5.1 – Étapes du changement de pile pour utiliser le rollforward lock dans les services d'Anaxagoros. La couleur indique qui a fait la dernière écriture. `sp` est le pointeur de pile.

Le dernier inconvénient est la taille prise : il faut réserver une certaine taille de pile pour chaque lock. Cela peut poser problème pour implémenter un verrouillage à grain fin. De plus, il est difficile de mesurer la taille de pile nécessaire ; l'emploi d'outils d'analyse de pile est vivement souhaitée pour ce cas. Le rollback lock souffre également de ce problème, puisque le journal des écritures faites doit également être stocké dans de la mémoire propre.

**Utilisation** Malgré tous ces défauts, on utilise beaucoup le rollforward lock dans les services. La quasi-totalité des requêtes faites au service ne renvoient pas de résultat (§ 5.1.1.3) : les clients conservent la majorité des données, et le service n'est contacté que pour qu'il modifie son état interne. Ce qui est renvoyé par le service sont des données, non des métadonnées, qui n'ont pas besoin d'être synchronisées (§ 5.4.6). La majorité des sections critiques sont ainsi des mises à jour de données[3]. Comme on minimise l'état des services, les mises à jour concernent un petit nombre de données, et le faible nombre de paramètres n'est pas gênant en pratique.

L'utilisation du rollforward lock est quasi-obligatoire : par exemple, les écritures de valeurs dans certains ports I/O doit se faire dans un certain ordre. Il n'y a alors pas de possibilité de "revenir en arrière" comme avec un rollback lock. Le rollforward lock est aussi intéressant pour sérialiser des écritures, avec des lectures qui se font sans utiliser le lock.

[3]Il y a bien sûr quelques exceptions, surtout pour les services politiques qui font de l'allocation dynamique de ressource

```
#include <rollforward_mutex.h>

// 200 est la taille reservée pour la pile.
ROLLFORWARD_MUTEX_DEFINE( test_lock, 200) ;
ROLLFORWARD_MUTEX_SECTION_DEFINE( test_section, test_lock,
                                  (unsigned int param1,
                                   unsigned int param2,
                                   unsigned int param3,
                                   unsigned int param4),
  {
    f3( param1, param2) ;
    f4( param3, param4) ;
  }) ;

void thread1_function( void)
{
  f1() ; f2() ;
  ROLLFORWARD_MUTEX_SECTION_CALL4( test_section,
                                   999, 888, 777, 666) ;
  f5() ; f6() ;
}
```

Listing 5.1 – Exemple utilisation de l'API du rollforward lock

**Interface de programmation** Comme la section critique ne doit pas accéder à la pile du thread qui l'utilise, nous la plaçons dans une fonction séparée, qui est appelée avec une méthode particulière qui acquiert le lock et change de pile. Comme cette fonction ne peut pas communiquer de résultat, elle retourne `void`. Ainsi, la seule manière pour le programmeur d'accéder à la pile est de passer à la fonction "section critique" un pointeur sur cette pile, ce que nous ne pouvons empêcher.

Le Listing 5.1 fournit un exemple réel d'utilisation de l'API.

**Évaluation** Nous avons évalué le coût de l'overhead de prise d'un verrou à 35 cycles en monoprocesseur, et 50 cycles en « multiprocesseur » (en fait monoprocesseur avec le code pour le monoprocesseur). La différence provient de l'utilisation d'un `Compare&Swap` dans le code, qui prend un certain nombre de cycles.

### 5.2.3 Rollback et revocable locks

Dans le cadre d'applications standards, le rollforward lock est le mutex non-bloquant le plus attractif, car il peut être utilisé à la place de sections critiques classiques sans modifier le code ni support du compilateur, et l'overhead est extrêmement faible[4]

[4]Bershad [Ber93] note cependant que le rollforward lock peut être un problème quand il y a possibilité de défaut de page, mais ce n'est généralement pas le cas e.g. pour les applications temps réel dur.

Mais les problèmes de pile non présente lorsque l'exécution est poursuivie par un autre thread imposent des limitations pénibles, ce qui nous fait considérer les autres sections critiques.

Dans celles-ci, une section critique n'est exécutée que par le thread qui l'a demandée. Par conséquent, la pile reste accessible pendant la section critique, et on peut renvoyer des résultats, passer autant d'arguments qu'on veut, et utiliser la même API que pour les programmes ordinaires.

#### 5.2.3.1 Rollback

Il n'y a pas de différence à utiliser le rollback lock dans un service par rapport au cas général. Mis à part que le journal des écritures doit être maintenu dans de la mémoire propre, sinon le recouvrement n'est pas possible. Cela consomme un certain montant de mémoire propre par lock. Mais cette taille est facilement calculable et relativement faible pour des sections critiques courtes, et en général bien inférieure à celle utilisée par un rollforward lock pour la même opération.

Un désavantage est le fait que dans certains cas, on ne peut pas revenir en arrière dans l'exécution (quand il y a communication avec un matériel, on ne peut pas toujours stocker puis restaurer l'état de ce matériel, e.g. les ports IO). Également, maintenir le journal crée un overhead, mais cet overhead est faible.

Néanmoins, ce lock devrait être sérieusement considéré dans le futur.

#### 5.2.3.2 Recoverable

Le recoverable lock ne souffre pas non plus des problèmes de pile qui affectent le rollforward lock, et son utilisation est donc similaire à celle du cas général. Il reste donc souvent difficile à utiliser.

**Ensure alone** Un cas où il est facilement utilisable est lorsqu'on ne veut que vérifier que l'utilisation est séquentielle. On peut souvent déléguer aux clients la responsabilité de s'assurer que la ressource n'est utilisée que par un thread à la fois (§ 5.3.2) ; et utiliser un recoverable lock pour s'en assurer.

L'idée est la suivante : si un thread doit effectuer une action séquentielle, il prend le lock, exécute l'action, puis le relâche. Le client conforme s'assure que personne d'autre n'utilise la ressource au même moment, et il n'y a donc pas de problème.

Si un autre thread tente de prendre le lock, le service constate que le lock a déjà été pris, et quitte immédiatement avec une erreur `ECONCURRENCY` comme dans la section sur les spinlocks non tournants (§ 5.2.1.1). Pour éviter qu'un thread puisse monopoliser une ressource de cette manière, la méthode de recouvrement consiste à nettoyer la ressource et libérer le lock même s'il était pris.

Cette technique permettrait d'implémenter un verrouillage à grain fin, individuellement pour chaque ressource, avec un faible overhead CPU et mémoire. Notons surtout qu'elle ne sert pas tant à synchroniser qu'à s'assurer que la synchronisation est faite correctement. Nous appelons ceci le “ensure alone”.

#### 5.2.3.3 Conclusion

Nous avons vu les trois sortes de verrou mis à disposition des services dans Anaxagoros. Malheureusement, il n'y a pas de bonne solution universelle, chacune ayant ses avantages et inconvénients :

- le rollforward lock consomme de la mémoire propre, et ne renvoie pas de résultat ; mais garantit la sérialisation des sections critiques ;
- le rollback lock consomme également de la mémoire propre, mais moins ; il peut renvoyer des résultats ; mais on ne peut pas toujours retourner en arrière dans l'exécution (communication avec matériel) ;
- le recoverable lock ne consomme quasiment pas de mémoire propre, mais est souvent difficile à utiliser.

Enfin, un dernier problème de synchronisation est le fait que nous n'avons pas encore de moyen de synchronisation inter-service, même si nous avons discuté de la possibilité d'en implémenter.

Une possibilité future serait d'examiner un verrou qui pourrait avoir certaines sections critiques en rollforward et certaines en rollback. Cela devrait couvrir quasiment tous les cas d'utilisation.

Une autre possibilité serait de fournir un environnement de programmation de plus haut niveau pour faciliter le travail du programmeur. Ce framework pourrait être fondé sur la génération automatique de drivers à partir des spécifications de l'OS et du matériel, comme le permet Termite [RCK+09]. Cela permettrait de n'avoir à résoudre les problèmes de synchronisation que par classe de drivers, et que cela s'applique à tous les drivers de la classe.

La solution que nous avons choisi est de minimiser les besoins en synchronisation et de permettre au programmeur de comprendre finement ces besoins, afin qu'il utilise ces primitives judicieusement. Cette minimisation offre par ailleurs d'autres avantages, en terme de passage à l'échelle et de non-perturbation de l'ordonnancement.

### 5.2.4 Le schéma de synchronisation use/destroy

#### 5.2.4.1 Présentation

**Problème** Le problème que nous cherchons à résoudre survient dans tout système multithread où les ressources sont allouées dynamiquement. Plusieurs threads peuvent utiliser une ressource lorsqu'un autre décide de la détruire. Il est nécessaire d'utiliser un schéma de synchronisation pour s'assurer qu'un thread n'utilise pas une ressource détruite.

Ce schéma de synchronisation a des applications dans tout le système. C'est notamment grâce à lui que nous avons pu implémenter notre système de mémoire virtuelle de manière presque entièrement wait-free. Il est également utile pour implémenter la révocation des ressources en espace utilisateur.

**Solution** Notre solution à ce problème consiste à détruire la ressource en plusieurs phases, nommément :

1. empêcher de nouveaux threads d'utiliser la ressource ;
2. attendre ou faire en sorte que les threads qui utilisent déjà la ressource ne l'utilisent plus ;
3. nettoyer ou détruire la ressource ;
4. marquer la ressource comme réutilisable.

On note également que la destruction d'une ressource est un évènement assez peu fréquent comparé à son utilisation. Typiquement, une ressource est allouée/créée, utilisée plusieurs fois, avant d'être désallouée/détruite. En particulier, il n'y a pas de raison d'empêcher l'utilisation parallèle de la ressource ; il faut juste empêcher l'utilisation lorsque la destruction est entamée.

Si on se permet d'attendre pour pouvoir détruire la ressource, il faut toutefois que ce temps d'attente ne soit pas trop long, i.e. qu'il puisse être borné. Ceci est imposé par le principe de révocation immédiate (§ 3.1.2.2). Par ailleurs, si l'attente était infinie, il serait possible pour des threads de monopoliser une ressource sans possibilité de leur retirer, ce qui est une faille de sécurité évidente.

**Théorie** Plus généralement, ce schéma de synchronisation fournit une autre solution au respect d'invariants sous forme canonique $P(a) \Rightarrow P'(b)$ (voir § 4.1.3.2). Les threads « utilisateurs » doivent s'assurer que $P'(b)$ est vrai quand ils modifient $a$, tandis que le thread « destructeur » veut rendre $P'(b)$ faux. Dans l'exemple précédent, $P'(b)$ est donc simplement "la ressource est prête".

**Concepts reliés** Le schéma de synchronisation readers/writer est un concept proche du notre : tous deux permettent à un ensemble de threads de s'assurer d'une condition (respectivement, que la mémoire n'est pas réutilisée pour le use/destroy lock, que les données ne sont pas modifiées pour le reader/writer lock). Nous proposons d'ailleurs différentes implémentation du schéma use/destroy, qui peuvent être utilisées pour implémenter le schéma readers/writer de manière différente que le traditionnel readers/writer lock.

Le typestable memory management (TSM) de Greenwald [GC96] est également un concept rapproché de notre schéma de synchronisation. Cependant dans le cas des TSM, il n'y a pas de synchronisation explicite, mais seulement une assurance (par le temps) que les threads qui utilisent une structure ne sont pas en train d'accéder à de la mémoire libérée (ou réallouée à un autre usage). Ainsi il est impossible de réutiliser de la mémoire en train d'être accédée d'utilisation immédiatement comme dans notre implémentation.

#### 5.2.4.2 Implémentations

**Présentation** Il y a plusieurs implémentations possibles de ce schéma. La première proposée utilise un verrou pour réaliser des exclusions mutuelles conformément au schéma décrit précédemment. En particulier :

- un thread qui veut détruire la structure est en exclusion mutuelle avec tous les autres ;
- les threads qui veulent utiliser la structure ne sont pas en exclusion mutuelle.

Ce schéma ressemble beaucoup au readers/writer lock, avec l'exception que les threads qui utilisent la structure peuvent la modifier. Comme il n'y a pas d'exclusion mutuelle entre les threads qui utilisent la structure, il faut soit utiliser un autre verrou, soit faire les modifications de manière non-bloquante.

**Implémentation théorique** L'idée est la suivante. Chaque ressource possède un compteur d'utilisateurs (i.e. du nombre de threads qui utilisent la ressource), incrémenté de 1 quand un thread utilise la ressource, décrémenté de 1 quand il ne l'utilise plus.

Chaque ressource possède également un indicateur de destruction, un booléen qui dit si un thread veut détruire la ressource. Quand cet indicateur est à 1, alors aucun nouveau thread ne peut toucher la ressource. Le thread qui veut détruire la ressource n'a alors plus qu'à attendre que tous les threads qui utilisaient déjà la ressource partent, i.e. que le compteur d'utilisateurs passe à 0. L'implémentation théorique est présentée en Figure 5.2.4.2 (`atomic` indique des opérations à effectuer atomiquement). Différentes implémentations réelles sont données en annexes B.1 et C.2.2.2.

```
error_t use( resource_t r)
{
  atomic{
    if( r->lock.destroyer)
      return EACCESS ;
    else r->lock.users ++ ; }
  ... // utilise ou modifie
  atomic{ r->lock.users -- ; }
}
```

```
error_t destroy( resource_t r)
 {
   atomic{
     if( r->lock.destroyer)
       return ECONCURRENCY ;
     else r->lock.destroyer = 1 ; }
   while( r->lock.users  != 0) ;
   ... // nettoie
   atomic{ r->lock.destroyer = 0} ;
}
```

FIG. 5.2 – Implémentation théorique du use/destroy lock.

Notons l'usage de `ECONCURRENCY`, qui permet de détecter des accès concurrents en renvoyant un erreur au lieu d'attendre, comme expliqué section 5.2.1.1. `EACCESS` est retourné lorsqu'on tente d'utiliser une ressource en cours de destruction, ou détruite.

Le problème principal de cette implémentation théorique est l'attente pour le thread qui veut détruire la ressource, que tous les threads utilisateurs aient terminé d'utiliser la ressource. Il est en général rare de détruire une ressource en cours d'utilisation (c'est utilisé par exemple pour forcer la destruction d'un programme), donc le fait d'attendre un peu ne crée pas un overhead significatif ; mais une attente infinie occasionne une faille de sécurité. Nous voyons comment ce problème est traité dans différents cas d'utilisation.

**Utilisation dans le noyau** Dans le noyau, on peut demander au thread qui utilise la ressource de le faire dans une section non préemptible, et de ne pas l'utiliser pour très longtemps. Par exemple, lorsqu'un thread veut modifier une entrée dans la table des pages, il acquière un use/destroy lock sur la table des pages, modifie l'entrée de manière atomique, puis relâche le verrou. L'utilisation est extrêmement simple, l'overhead petit, et est l'une des raisons pour laquelle nous présageons d'excellentes performances pour notre système de gestion de la mémoire virtuelle (dont l'implémentation est décrite en annexe C).

Si on doit modifier plusieurs entrées dans la même table, on peut éviter l'overhead de prendre et relâcher le verrou d'utilisation : il suffit de vérifier directement si `lock.destroyer == 1` sans relâcher le verrou. Si c'est le cas, cela signifie que la ressource est en instance de destruction, et on relâche le verrou. Cette technique de "polling" est souvent utilisable lorsque le thread doit accomplir une tâche longue et répétitive, et peut être couplée avec l'introduction des *points de préemption explicite*.

**Utilisation à travers le noyau** Tant que l'utilisation de la ressource se fait dans le noyau, la technique précédente fonctionne : il suffit de vérifier si la ressource doit être détruite, aux moments où on fait des vérifications pour les points de préemption explicite. Mais il peut arriver qu'on utilise une ressource pendant un certain temps sans pouvoir faire cette vérification, lorsqu'on sort du noyau pour exécuter une application utilisateur.

Par exemple, si on veut utiliser un domaine, on va prendre un verrou d'utilisation sur ce domaine, puis poursuivre l'exécution de ce domaine en espace utilisateur. Simultanément, un thread (sur un autre CPU) veut détruire le domaine exécuté.

Si on ne fait rien, le thread qui veut détruire ce domaine exécuté va attendre pendant longtemps. Pour remédier à cela, ce thread va envoyer un IPI au processeur qui exécute le domaine, afin qu'il retourne dans le noyau et arrête d'utiliser le domaine plus tôt. Il faut faire cela avant d'attendre que tous les threads relâchent la section critique.

Ainsi, au « polling » de la section précédente, on substitue un mécanisme « d'interruption » pour prévenir les threads d'arrêter d'utiliser la ressource.

Au niveau de l'implémentation, il ne suffit plus maintenant de compter les processeurs qui utilisent la ressource, mais il faut pouvoir les identifier. Cela peut se faire en remplaçant le compteur `thread.users` par un bitfield, ou en indiquant pour chaque processeur quelles sont les ressources qu'il est en train d'utiliser. Le thread qui veut détruire la ressource envoie alors un IPI à tous ces CPUs entre le moment où il met son indicateur à 1 et celui où il attend que plus personne n'utilise le verrou. Les implémentations réelles sont présentées plus en détail en annexes B.1 et C.2.2.2.

**Utilisation en espace utilisateur** Notons que l'implémentation précédente retire le besoin d'exécuter le code dans une section non préemptible. Cela peut permettre d'utiliser le use/destroy lock en espace utilisateur.

Le fait d'avoir un upcall de reprise suffit pour implémenter le schéma use/destroy en monoprocesseur : il suffit de paramétrer cet upcall pour qu'il regarde si la ressource n'a pas été détruite avant de reprendre son exécution. L'exemple le plus important de

l'utilisation de cette méthode est lorsque nous vérifions que les dates de la capacité et de la ressource correspondent encore lors de la reprise d'exécution dans le service (§ 3.3.3.2).

En multiprocesseur, les IPIs sont nécessaires : si un thread détruit une ressource, il faut empêcher les autres threads de l'utiliser. Cela se fait comme dans le noyau, à la différence que l'appel système de la section 4.2.2 ne permet d'envoyer un IPI qu'aux threads qui partagent le même domaine.

#### 5.2.4.3 Conclusion

Le use/destroy pattern est un schéma de synchronisation extrêmement intéressant. Il permet à différents CPU d'utiliser la même ressource sans aucune restriction sur le parallélisme, tout en permettant une révocation de la ressource en un temps borné (principe 3.1.2.2). Ces deux propriétés sont intéressantes à la fois pour les systèmes temps réel dur, mais aussi pour les systèmes best-effort.

Grâce à lui, notre système de mémoire virtuelle fait toutes ses opérations en parallèle, sans copie, en opérant directement sur les données, et sans boucle ; donc de manière wait-free, sauf pour les quelques attentes lorsqu'un thread veut forcer la récupération d'une page mémoire. Cela devrait le rendre extrêmement scalable pour la plupart des opérations. Cette implémentation pourrait être reprise par d'autres exokernels ou VMMs, dont les implémentations actuelles utilisent un certain nombre de verrous ; ou même dans la couche basse de noyaux monolithiques.

Enfin, cette technique est très importante pour l'implémentation du principe de révocation rapide (principe 3.1.2.2), puisque la révocation peut se faire en temps constant.

**Travaux futurs** L'implémentation du schéma use/destroy pattern ne peut implémenter des ressources qui n'ont que deux états : invalide/inutilisable et valide/utilisable. Il serait intéressant (et a priori simple) d'étendre ce schéma pour gérer des états plus complexes, si nécessaire. C'est ce qui est fait avec les pages mémoires, dont l'automate des états est plus complexe (voir annexe C) ; il serait intéressant de voir comment généraliser cette approche.

### 5.2.5 Conclusion

Nous avons expliqué comment utiliser différentes primitives pour faire la synchronisation à l'intérieur des services. Ces primitives ne sont pas foncièrement difficiles à utiliser pour un programmeur habitué à la programmation système, mais demandent néanmoins une réflexion. C'est la raison pour laquelle nous recommandons de ne pas trop se soucier de l'implémentation des synchronisations dans un premier temps.

Mais avant de tenter de résoudre des problèmes de synchronisation, il vaut mieux les éviter ; i.e. ne pas synchroniser à moins que cela soit nécessaire. Les sections suivantes se penchent sur ces questions.

## 5.3 Minimisation des problèmes de concurrence

Nous énonçons ici trois principes nouveaux, permettant de largement réduire les problèmes de concurrence dans les services. En particulier ils réduisent les problèmes de cohérence, en faisant de sorte que tout accès aux données à faire de manière atomique ne nécessite qu'un nombre faible d'accès mémoire.

Ces trois principes sont : *minimisation et exportation des données*, *exportation des synchronisations* et *décomposition en états intermédiaires cohérents*, et permettent respectivement de réduire la taille des données à synchroniser, le nombre de synchronisations nécessaires, et le nombre d'opérations à réaliser pour passer d'un état cohérent à un autre.

Ces principes sont issus de notre expérience à programmer les services partagés, et facilitent énormément la résolution des problèmes de cohérence dans les services. Nous présentons d'abord les principes, avant de donner en section 5.4 une panoplie de techniques de conception (nouvelles ou non) qui suivent ces principes.

**Considérations générales** L'idée générale est d'exporter un maximum de responsabilité vers le client, dans lequel il est beaucoup plus facile de réaliser une éventuelle synchronisation que dans le service. En effet le client peut faire confiance à ses politiques d'allocations, et est donc un programme (éventuellement multithread) normal. Au contraire, le service n'accorde aucune confiance ni aux clients, ni à leurs politiques d'allocation, ce qui rend la synchronisation bien plus difficile (§ 5.1.1).

Notons que la minimisation des problèmes de concurrence ne se contente pas seulement de simplifier ces problèmes : elle permet également d'augmenter les possibilités de parallélisme, et de réduire les interférences d'exécution entre les différents threads.

Enfin, cette minimisation des problèmes de cohérence va également dans le sens d'une minimisation et simplification des services, ce qui suit le principe de sécurité de minimisation des mécanismes communs (§ 2.2.2.6).

### 5.3.1 Minimisation et exportation des données

#### 5.3.1.1 Présentation

**Principe** Le principe de minimisation des données consiste à faire en sorte que *les données stockées dans le service soient les données minimales nécessaires pour assurer un service fonctionnel*. Un principe majeur qui aide à cette conception est l'exportation des données, qui consiste à ce que *toute donnée qui peut être placée dans le client le soit*. En général, les données non exportées le sont soit parce qu'elles doivent être partagées pour des raisons fonctionnelles (comme la position du curseur du service de terminal), soit parce que le service les protège pour raison de sécurité (comme le contenu des tables des pages).

Notons que ce principe va également permettre le respect de la contrainte de statelessness (§ 3.5.2.1).

**Types de données dans le service** Comme vu en section 3.5.2, les données manipulées par le service sont regroupées en trois types :

- les données *locales* ou *par connexion*, qui sont stockées dans la mémoire prêtée de façon transitoire, comme la pile ;
- les données *globales*, qui sont stockées dans la mémoire propre du service ;
- les données *par ressource*, qui peuvent a priori être stockées sur de la mémoire propre ou prêtée de manière semi-permanente.

La minimisation porte uniquement sur les données globales et par ressource, qui sont celles qui peuvent être accédées simultanément par plusieurs threads. Au contraire, certaines techniques augmentent la taille des données locales.

**Exemple** Par exemple, dans les systèmes POSIX l'offset courant dans un fichier est stocké dans le service (dans le file descriptor). Cet offset est mis à jour par les appels systèmes `read` et `write`. Or, il est facile d'exporter cette donnée pour la maintenir et mettre à jour dans le client, qui la passerai en paramètre de ces appels systèmes.

**Mise en œuvre** Elle est simple : il suffit de se demander pour chaque variable si on a besoin de la garder dans le service, et si oui de voir si on peut la « compresser ». Les différentes techniques de la section 5.4 aident à cela.

#### 5.3.1.2 Avantages

La motivation principale pour ce principe est que toute donnée qui n'est pas stockée par le service, n'a pas besoin d'être synchronisée par celui-ci. Le problème est exporté vers le client. Mais ce principe présente d'autres avantages.

**Réduction de la mémoire consommée** Lorsque les données à stocker par ressource sont nombreuses, et qu'elles sont placées dans le tableau des ressources, beaucoup de mémoire est gâchée lorsque toutes les ressources ne sont pas utilisées. C'est particulièrement important dans notre cas où la mémoire propre des services est allouée statiquement. Minimiser ce tableau peut représenter un gain de mémoire considérable.

Dans le cas où la mémoire des services est reçue dynamiquement (prêt semi-permanent de mémoire), d'autres problèmes de consommation mémoire se posent aussi, comme la fragmentation (on ne peut donner qu'un multiple d'une page). L'exportation des données permet le plus souvent de limiter l'utilisation de la mémoire prêtée de manière semi-permanente à des buffers quand le service fait des envois asynchrones (e.g. service réseau/service de disque dur).

**Réduction des bugs et vulnérabilités** Puisqu'il y a moins de données, il y a aussi moins de code nécessaire pour les manipuler ; tout cela permet de réduire la probabilité d'occurrence de bug dans le service. C'est particulièrement important puisque le service est partagé par plusieurs tâches, qui en dépendent, et que ces bugs peuvent être exploités par des tâches malicieuses. On réduit donc les possibilités d'attaques [Ber07, § 2.3].

Notons que ces données, étant globales et partagées par plusieurs threads, sont d'autant plus sources de bugs [WS73] ; les minimiser réduit l'occurrence de ces bugs d'autant.

**Réduction des canaux cachés** Dès qu'une donnée dans un service est lisible et modifiable par plusieurs clients, cette donnée leur permet de communiquer. Ainsi, minimiser les données gérées par le service limite cette communication, et est intéressante pour la confidentialité. On peut voir cela comme une occurrence particulière de réduction de bugs.

**Amélioration des performances** Déplacer les informations du service vers le client :

- rend l'accès et la modification de ces données plus "facile" pour le client (e.g. il n'y a plus besoin d'appels à `lseek` pour modifier l'offset du fichier) ;
- rend l'accès et la modification de ces données plus "difficile" pour le service (ces données doivent être copiées lors des appels, donc les requêtes sont plus grosses).

Globalement, on diminue le nombre d'appels systèmes, mais on augmente la taille des requêtes. Vu que l'overhead d'un appel système est largement supérieur à celui de l'overhead de la copie d'un paramètre, ce compromis est a priori intéressant du point de vue des performances.

**Adaptation aux besoins** Comme le client gère les données, il peut être adapté selon ses besoins ; par exemple utiliser des librairies différentes suivant que les allocations sont statiques ou dynamiques, ou que le client sera multithreadé ou non (cela permet d'avoir du code plus simple, et donc plus sûr, pour les applications critiques).

On peut aussi utiliser des librairies spécifiques pour maximiser les performances selon les besoins des applications. Les exokernels ont notamment démontré que cette approche permettait d'améliorer les performances [EKJO95, KEG+97][5].

**Amélioration du parallélisme** Quand un appel modifie une donnée dans le service, on ne peut pas faire plusieurs appels en parallèle : il faut attendre qu'une opération soit terminée avant de pouvoir changer les données pour la prochaine opération. Sans cela on risque d'obtenir un comportement indéterministe (§ 5.2.1.1).

[5] ce résultat est nuancé par [HHL+97].

Par exemple, le fait que l'offset soit situé dans le service de système de fichier dans POSIX empêche qu'il y ait plusieurs lectures se déroulant en parallèle sur le même descripteur de fichier (c'est la raison[6] pour l'addition dans POSIX des appels `pread` et `pwrite`, qui prennent l'offset en paramètre sans les modifier. Pour faire des lectures en parallèle, les file descritors devaient sinon être dupliqués, ce qui prend de la place.)

De plus en supprimant les données du service, les synchronisations du service ne concernent que des données essentielles. Ainsi, du point de vue global de l'exécution les sections atomiques sont concentrées et réduites au maximum, ce qui permet d'améliorer le passage à l'echelle en multiprocesseur [GC96].

#### 5.3.1.3 Inconvénients possibles

**Impact sur l'espace pris** Lorsque de multiple clients utilisent la même ressource, on peut penser que les données auparavant maintenues dans le service sont maintenant copiées dans plusieurs clients. Mais en fait, rien n'empêche de multiples clients de partager ces données dans une zone de mémoire partagée si nécessaire.

**Incompatibilité avec les anciens comportements** Exporter les données au client peut générer des problèmes d'incompatibilité, quand des clients multiples "communiquaient" par l'intermédiaire de ces états partagés. La communication doit alors se faire de manière explicite.

Il est normalement relativement simple d'émuler les anciens comportement, car souvent des tâches qui utilisent la même ressource ont un moyen de communication entre elles. Ainsi, le cas de programmes multithreadés UNIX peut se résoudre assez simplement dans la "libOS" UNIX sous la libc. Pour des programmes multiprocessus, la communication est plus difficile, mais peut se faire en établissant une zone de mémoire partagée entre les différents processus. Cela suffit par exemple pour partager les offsets des file descriptors courants.

Pour les autre cas, on peut imaginer un service de "proxy" dans lequel les informations à partager seraient stockées de manière transparente.

Il y a des cas où l'émulation peut poser problème. Par exemple dans UNIX, le répertoire parent d'un répertoire est stocké dans le répertoire. À cause des liens symboliques, `foo/bar/..` n'est peut être pas égal à `foo`, alors que ce serait le cas si la résolution des noms était faite par le client[7]. Mais globalement, ces incompatibilités sont relativement rares et la majorité des programmes ne devrait pas s'en apercevoir, ou devrait pouvoir être modifiés facilement.

#### 5.3.1.4 Comparaison à l'état de l'art

Il existe d'autres systèmes qui proposent de minimiser la taille des services, mais les raisons principales pour cela sont différentes des nôtres (minimisation des problèmes de concurrence, des interférences dans le temps d'exécution entre les différents

[6] cf page man de `pread` de SCO UNIX, `http ://uw714doc.sco.com/en/man/html.2/pread.2.html`.

[7] Ce comportement cause d'ailleurs certains problèmes, et permet par exemple d'échapper à des `chroot`, voir [hKW00].

threads, et du besoin d'allocation de mémoire propre, sécurité). Ainsi, Nemesis réduit les services pour minimiser le temps passé à exécuter des requêtes dans les services (appellé QoS crosstalk) [Ros95, p. 17] ; c'est inutile dans Anaxagoros puisque le prêt de temps CPU permet d'attribuer ce temps CPU correctement. Les exokernels réduisent également la taille de leurs services [EKJO95], mais dans un but de flexibilité et performance. Rushby minimise la taille des services uniquement dans un souci de sécurité [Rus84].

La minimisation des services est donc un bon choix de conception pour tous ces buts : comptabilisation exacte du temps CPU utilisé, sécurité (minimisation des mécanismes communs), flexibilité et performance, et scalabilité pour le multiprocesseur.

### 5.3.2 Délégation de la gestion de la concurrence

**Principe** Une fois les données exportées, il existe encore des moyens de réduire le nombre de synchronisations à faire dans le service. Nous introduisons le principe de délégation de la cohérence, qui est l'un d'entre eux. Ce principe énonce que, *pour tout ensemble de données, on rend (lorsque c'est possible) les utilisateurs du service responsables de la cohérence de ces données*, au lieu que ce soit le service qui assume seul cette contrainte. En fait, non seulement la gestion de la cohérence des données peut être exporté, mais également celle du séquencement des actions du service (par le séquencement des requêtes qui lui sont faites). Ainsi, c'est toute la gestion de la concurrence qui peut exportée.

L'idée est que pour un certain nombre de cas, les clients sont tout à fait en mesure de se synchroniser pour envoyer les requêtes au service dans le bon ordre, et de faire en sorte que les données contenues dans le service soient cohérentes. Dans ce cas, on déleste le service de l'obligation de maintenir cette propriété. On a deux possibilités : le service peut soit vérifier que les appels des clients sont fait dans un ordre tel que les actions soient cohérentes[8] ; soit ne rien faire du tout quand par exemple les données incohérentes ne gênent pas les autres clients du services.

**Conditions d'application** Notons d'abord que ce principe ne s'applique qu'après exportation des données, i.e. il vaut mieux exporter une donnée plutôt que seulement la gestion de sa cohérence.

Considérons les cas principaux où on ne peut pas appliquer ce principe :

**Problèmes de sécurité** Lorsque la violation d'un invariant pose un problème de sécurité. Par exemple, le service mémoire maintient pour chaque page un compteur de mappings, qui indique le nombre de fois où une page de donnée a été mappée (i.e. est dans une table des pages). Cela permet par exemple d'éviter qu'une page écrivible puisse être utilisée comme table de page, lorsque quelque part un client peut écrire dedans (compteur de mappings différent de 0).

[8] Cette idée peut être reliée à la notion de serializabilité dans les bases de données [FR82]. Comme le montrent Fle et Roucairol [FR85], la vérification du séquencement des actions peut se faire à l'aide d'un simple automate.

L'invariant "nombre de fois où la page est mappée = nombre indiqué par le mapping" doit être respecté[9], sinon ce compteur de mapping devient inutile.

Notons que dans ce cas, on peut souvent faire une détection de concurrence, par exemple avec un spinlock non tournant (§ 5.2.1.1) plutôt qu'une synchronisation pleine.

**Clients ne pouvant pas assurer la gestion de la concurrence** Le but d'un service est de partager une ressource entre des clients qui ne se font pas confiance. Quand ces clients peuvent modifier ces données en parallèle, ils ne peuvent pas établir la cohérence de ces données. C'est souvent le cas pour les données globales. Par exemple, on ne peut pas demander à des tâches critiques et non critiques de se synchroniser pour ne pas faire d'écritures simultanément dans le service d'affichage de terminal : les données globales relatives à cet affichage doivent être synchronisées.

**Exemples d'application** Quand on est assuré que les clients se connaissent, on peut au contraire utiliser ce principe : c'est ce que fait la technique d'exportation de la synchronisation, que l'on verra en § 5.4.6. En général, on peut présumer que des clients qui utilisent une même ressource se font confiance.

L'autre utilisation que nous avons eu de ce principe lors de la conception d'Anaxagoros est pour le maintien de la cohérence lors des flushs de TLB dans le service mémoire ; cela permet de demander des flushs TLB par le noyau que si cela peut créer un problème de sécurité. L'annexe C.3.3 détaille cette technique.

**Avantages** Ce principe permet de minimiser le nombre de synchronisations à faire dans le service, et retire souvent des synchronisations inutiles ou redondantes.

### 5.3.3 Décomposition en états intermédiaires cohérents

#### 5.3.3.1 Présentation

**Principe** Même une fois qu'on a exporté toutes les données exportables vers le client, et exporté la synchronisation partout où on pouvait, il se peut qu'il reste des opérations assez importantes à faire "d'un coup" de manière atomique. Par exemple, le nettoyage d'une table des pages nécessite un nombre important d'opérations (mise de l'entrée à 0, diminution de compteurs de références, etc.), et les deux précédents principes ne sont pas applicables. Cela entraîne des sections critiques de taille importante, ce qui augmente les interférences dans les temps d'exécution entre différentes requêtes.

Ce problème peut être résolu en découpant les grosses opérations à faire atomiquement en une succession d'opérations atomiques plus petites. Cela expose des états qui sont *intermédiaires*, mais *cohérents* (Figure 5.3).

**Cohérence des états intermédiaires** La cohérence des états intermédiaires est importante. L'idée est que l'état intermédiaire n'est pas un état où on veut rester,

[9] en réalité, cet invariant peut être relâché un court instant ; voir § 4.1.3.3

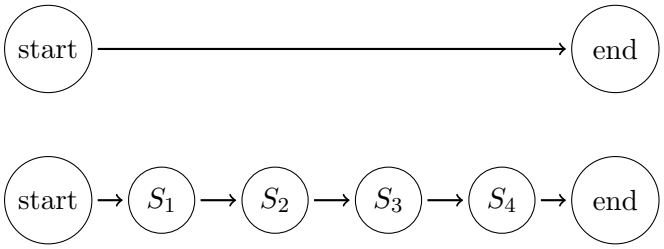



FIG. 5.3 – Décomposition en états intermédiaires cohérents. Chaque flèche représente une opération atomique, chaque cercle un état cohérent. La décomposition crée les états intermédiaires $S_i$.

mais où on peut rester si nécessaire. C'est à dire que si le client est détruit à ce moment là, un autre thread pourra faire des transformations à partir de cet état.

**Exemple** Prenons l'exemple d'un service d'affichage en mode texte. De multiple clients (qui ne peuvent pas se coordonner) envoient à ce service des chaînes de caractères, qui peuvent être de taille relativement longue. L'ordre d'écriture des caractères est indéterministe, mais on veut que tous les caractères soient écrits.

Plutôt que de considérer l'écriture de la chaîne entière comme étant atomique [10], on décompose cette opération comme étant une succession d'écriture de plusieurs caractères, chacun étant écrit de manière atomique.

L'écriture du caractère demande plusieurs opérations : entre autre la mise à jour du buffer video, et la mise à jour des coordonnées $x$ et $y$ (qui indiquent l'endroit de la prochaine écriture) dans ce buffer. Ces opérations sont indivisibles, car sinon on laisserait le service dans un état inconsistant, et des caractères seraient perdus. Ainsi, on ne peut pas décomposer plus cette opération.

On peut observer un autre exemple concernant le nettoyage de pages en annexe C.1.1.

**Contre-indication : opérations atomiques** Il faut parfois ne pas suivre ce principe : i.e. ne pas décomposer une opération en deux, même si l'état intermédiaire entre les deux est consistant. C'est lorsqu'on veut que l'opération se fasse de manière atomique.

C'est surtout le cas pour les opérations sur le système de fichier, qui est globalement partagé. Par exemple, l'appel système POSIX `open` peut avoir la signification "ouvre un fichier, et s'il n'existe pas, crée-le". Cette opération ne peut pas se décomposer en "ouvre un fichier ouvert" ou "crée un nouveau fichier", car cela permettrait l'occurrence d'une race condition si le fichier est créé par un autre programme entre ces deux opérations. L'existence d'une opération qui fait atomiquement un "ouvre ou crée" est donc justifiée.

Mais ces contre-indications sont des exceptions. Il faut voir leur cas comme analogue aux instructions de type "compare-and-swap" dans les processeurs RISC :

[10] ce qui ne rendrait pas l'écriture dans la console déterministe

des exceptions à la décomposition en éléments simples, qui sont là uniquement pour raison d'atomicité.

**Mise en œuvre** Nous décomposons les états intermédiaires dans nos services, par les techniques de "suppression de la logique de contrôle"(§ 5.4.3) et de "suppression des abstractions"(§ 5.4.2). Mais il y a d'autres manières de procéder : par exemple le service pourrait proposer des opérations complexes sur les ressources, analyser dans quel état est la ressource au début d'une opération et agir en conséquence. Ainsi, on peut décomposer en états intermédiaires sans que ceux-ci soient exposés au client.

#### 5.3.3.2 Avantages et inconvénients

**Sections critiques plus courtes** Ce principe permet d'avoir des sections critiques plus courtes, dans lequel sont faites des opérations en temps constant.

L'avantage principal est que cela facilite l'utilisation des primitives de synchronisation légères décrites au chapitre 4 : par exemple, la pile utilisée pour le rollforward lock serait plus court, le journal pour le rollback lock également, et il est plus facile de réarranger les codes courts pour utiliser le revocable lock.

Les sections peuvent même devenir si courtes qu'on peut même éventuellement faire ces opérations en utilisant directement les instructions atomiques du processeur.

Dans certain cas, cela permet également d'augmenter le passage à l'échelle pour le parallélisme : on aura moins de contention ou d'attente active pour les verrous, moins de chance de recommencer pour les mécanismes de concurrency control optimistes, etc.

Le désavantage est que cela cause une certain overhead : il y a plus de verrous à prendre, ou plus d'écritures à faire pour consigner l'état intermédiaire, ou plus de vérifications dans le cas des synchronisations optimistes.

**Retirer le besoin de notifications au service** Comme toutes les opérations se terminent dans un état consistant, les "connections" avec le service peuvent se couper à tout moment. En conséquence, on n'a pas besoin de prévenir le service lorsqu'il n'y a plus de client pour se connecter. Par opposition, les services "stateful" peuvent nécessiter un nettoyage quand la connexion se coupe.

Dans Mach, une notification est envoyée si le thread est terminé pendant l'appel [FL94, p. 8], ou si la tâche n'utilise plus sa capacité (« no-sender notification » [Loe92]). On économise donc l'implémentation de tels mécanismes, qui posent de nombreux problèmes (à qui décompter le temps pour traiter cette notification ?)

Le service suit donc plus un modèle de « procédure partagée », comme dans les anciens systèmes à capacité [Lev84], plutôt que le modèle plus moderne de « serveur partagé ».

## 5.4 Techniques de conception

Dans cette section, nous exposons différentes techniques de conception qui permettent de répondre aux exigences de statelessness (§ 3.5.2.1) et d'indépendance des politiques

d'allocation (§ 3.1.1), et suivent les trois principes de réduction des synchronisations. Elles contribuent ainsi aux avantages des principes qu'elles suivent.

Certaines de ces techniques sont nouvelles, tandis que d'autre existaient déjà, mais n'étaient pas utilisées pour les mêmes raisons.

### 5.4.1 Passage des données en argument

**Présentation** Cette technique est relativement classique pour faire des services stateless. Mais elle sert aussi pour toute exportation des données en général.

Pour faire certaines opérations, le service a besoin de données qui sont présentes dans le client, et non plus dans le service. Quand le service a besoin d'accéder à ces données, il les prend tout simplement en argument des appels du client.

Dans le service, on diminue ainsi les données par ressource et globales, et on supprime les données par connexion, tandis qu'on augmente les données locales (qui ne sont pas partagées, donc ne doivent pas être synchronisées).

**Exemple** Par exemple, dans les systèmes POSIX est stocké dans le file descritor (données par connexion) l'offset courant dans le fichier, les flags utilisé dans l'appel à `open`, etc. Les flags de l'appel à `open` sont utilisés par les appels à `read` ou `write`. Ces deux appels systèmes utilisent et mettent à jour l'offset du ficher. Si cet offset est stocké sur 2 mots, il faut le synchroniser si des lectures et écritures arrivent en parallèle.

Dans un système stateless, ces flags[11] et l'offset seraient passés comme arguments supplémentaires à `read` et `write`. Les seules informations par connexion stockées dans la capacité seraient l'inode du fichier accessible, et les permissions de lire ou d'écrire ce fichier. L'offset n'a plus besoin d'être synchronisé par le service. De multiples clients peuvent accéder en parallèle au fichier en utilisant le même file descriptor[12].

### 5.4.2 Suppression des abstractions

**Présentation** L'idée est que le service devrait présenter une interface de bas niveau, qui soit proche du hardware plutôt que des abstractions manipulées par les programmeurs d'applications. Ces abstractions peuvent être fournies dans des librairies : c'est ce que font par exemple les exokernels [EKJO95, KEG+97] ou Nemesis [Ros95].

En faisant cela, le service n'a pas besoin de maintenir les données relatives à la translation entre l'abstraction et le matériel. Cela permet une réduction drastique des données à maintenir dans le service, qui sont exportées vers le client.

Comme les opérations sur les abstractions sont des agrégats d'opérations sur les ressources matérielles, cette technique tend également à suivre le principe de décomposition des états intermédiaires cohérents (§ 5.3.3). Rendre ces états intermédiaires cohérents est aussi plus facile, vu qu'il y a moins de données.

[11] en fait, seul les flags intéressants pour l'appel courant ; ainsi le flag `O_CREAT` ne sert à rien pour un appel à `read`

[12] l'appel système POSIX `pread`, qui prend l'offset en argument, a été créé dans ce but

**Exemple** L'interface de gestion de la mémoire virtuelle dans POSIX est de haut niveau, et on n'a accès qu'à des fonctions comme `mmap`, qui manipulent des portions de l'espace d'adressage. L'interface que nous proposons est de bas niveau, et permet la manipulation directe des tables de pages.

On peut ainsi spécialiser l'algorithme d'allocation mémoire : sur les systèmes temps réel critiques, l'allocation est faite statiquement et est relativement simple et sûre ; sur les systèmes moins critiques, on émule l'appel à `mmap`, `exec`, etc. par une librairie qui implémente tout cela.

**Autre avantage** En étant plus proche du matériel, cette technique simplifie l'utilisation du système pour la paravirtualisation. En effet, l'OS modifié est conçu pour manipuler des objets de bas niveau, et pas de haut niveau.

### 5.4.3 Suppression de la logique de contrôle : interface “RISC”

**Présentation** Le service offre une interface permettant d'effectuer des opérations privilégiées sur des ressources. Les services traditionnels proposent comme interface des fonctions complexes, comme `mmap` ou `readv`. Par opposition, nous faisons en sorte que les services proposent uniquement des fonctions simples, élémentaires. Les fonctionnalités de haut niveau sont obtenues en composant ces fonctions simples.

La logique de contrôle, i.e. la partie “intelligente” du programme qui va composer les opérations élémentaires pour en faire des opérations de haut niveau, est donc exportée vers le client.

Cette technique suit le principe de la décomposition en états intermédiaires cohérents (§ 5.3.3) : les opérations exportées sont courtes, donc exposent plus d'état intermédiaires ; et il est normal pour une opération d'une interface de modifier l'état de manière cohérente.

**Exemple** Le service mémoire (voir annexe C) est un exemple parfait d'application de ce principe. Par exemple, pour mapper une zone de mémoire, il faut successivement mettre à zéro quelques pages, les transformer en table des pages, y placer des entrées pointant sur des pages de données, puis placer ces tables de pages dans la table des pages de second niveau. On a donc bien composition de fonctions individuelles.

De même, l'appel à `write` dans le service d'affichage est véritablement vu comme une succession d'appels à `putchar`.

**Interface programmable** En suivant ce principe, l'interface du service présente un certain nombre d'“opérations élémentaires”, et le client obtient des fonctionnalités de haut niveau en écrivant un “programme” qui compose ces opérations. On peut ainsi comparer l'interface proposée par les services aux jeux d'instructions des processeur. Les systèmes traditionnels proposent une interface permettant des opérations complexes, comme les jeux d'instructions CISC. L'interface que nous proposons propose des instructions simples et composables, comme les jeux d'instructions RISC.

Notre implémentation va plus loin dans cette analogie. Pour pallier à l'augmentation du nombre de requêtes client-service causée par cette décomposition, nous envoyons les demandes d'opérations par lot.

Le mécanisme de multicall (annexe C.3.4.4) permet au client de soumettre aux services de véritables programmes qui exécutent ces différentes opérations élémentaires. Le jeu d'instruction de ces programmes est optimisé pour que leur écriture soit rapide (puisque ces programmes ne sont exécuté qu'une seule fois). De plus, pour pallier au fait que l'augmentation du nombre de requêtes est combinée à une augmentation de la taille des requêtes (à cause du passage des données en argument (§ 5.4.1)), le jeu d'instructions permet de grouper les arguments passés à plusieurs opérations.

**Relation avec suppression des abstractions** Les principes d'interface programmable et de suppression des abstractions sont complémentaires, mais bien distincts. On peut en effet tout à fait offrir des opérations complexes sur des objets de bas niveau d'abstraction, et au contraire offrir des opérations élémentaires sur des objets de haut niveau d'abstraction.

Les principes sont complémentaires, en ce que les opérations élémentaires sur des objets de bas niveau d'abstraction sont plus courtes et simples que celles sur les objets de haut niveau ; et que la suppression des abstractions augmente les possibilités de ce qu'il est possible de faire en "programmant l'interface".

En résumé, la suppression des abstractions consiste en une simplification des données, tandis que la suppression de la logique de contrôle simplifie les opérations sur ces données.

**Autres avantages**

**Optimisations** Tout comme le RISC pour les processeurs, cette technique facilite l'optimisation des opérations de base. Ainsi, notre système de mémoire virtuelle est presque entièrement wait-free, et la synchronisation des données est faite sans verrou, en utilisant seulement quelques instructions atomiques du processeur. Or, la programmation lock-free est notoirement difficile, et est rendu possible uniquement parce que nous avons décomposé les opérations à faire en opérations simples.

Cela devrait permettre à notre système de mémoire virtuelle de passer à l'échelle lorsqu'il y aura de multiples processeurs. Cela pourrait même permettre d'implémenter le service de mémoire entièrement en hardware : chaque opération faite dans le service est relativement petite et pourrait être directement câblée.

Cette technique permet aussi de limiter la consommation de pile.

**Élimination de code de confiance** Enfin, tout comme la suppression des abstractions, cette technique permet d'éliminer du code de confiance, donc des vulnérabilités.

**Plus de flexibilité** En exposant des opérations petites, le programme a plus de choix de la manière dont il va les composer, et peut faire cela de manière adaptée à

la situation. Par exemple, l'allocation de mémoire dans un programme avec garbage collection pourrait être différente dans un programme qui n'en dispose pas.

**Contrôle du temps de calcul** Les opérations exportées sont simples, souvent en $O(1)$. Ainsi cette conception aide pour calculer ou évaluer des majorants d'exécution du programme et de ses interactions avec les services plus facilement, ce qui est important pour le temps réel dur, et suit notre principe de temps d'exécution prévisibles (§ 3.1.2.2).

**Atomic API** Enfin, cette conception permet d'avoir facilement une API atomique pour les services du noyau. En permettant à des threads de lire l'état actuel du « programme multicall », on pourrait implémenter des mécanismes de checkpoints comme décrit par Ford et al. [FHL+99].

### 5.4.4 Suppression des redondances

**Présentation** Cette technique suit le principe de minimisation des données (§ 5.3.1)

Il s'agit d'une technique générale pour éviter d'avoir à faire des synchronisations. L'idée est que beaucoup d'invariants peuvent être évités en supprimant certaines données partagées, quand ces données peuvent être recalculées à partir d'autres données partagées. En supprimant des redondances, on supprime des invariants.

**Exemple** Dans le service d'affichage textuel, les données globales sont le buffer vidéo et les positions $x$ et $y$. Le pointeur courant dans le buffer vidéo, qui indique où écrire le prochain caractère, n'a pas besoin d'être placé dans les données globales, et peut être recalculé.

En effet, on a la relation :

```
pointeur_courant = adresse_buffer_video + y * NB_COL + x
```

Cette relation est un invariant, donc si `pointeur_courant` était placé dans la mémoire globale, il devrait être synchronisé avec les valeurs de `x` et `y`, ce qu'on évite en le recalculant.

Notons qu'on aurait pu stocker dans la mémoire globale `pointeur_courant`, et calculer `x` et `y` à partir de sa valeur. Mais le calcul, dans ce sens demande des divisions et modulo, et est plus coûteux que la multiplication dans l'autre sens[13].

**Compromis espace/temps** Recalculer ces valeurs prend du temps CPU, mais il est infime la plupart du temps. Il est souvent préférable de supprimer la donnée et de la recalculer lorsque le calcul concerne un petit nombre fixe de données, au vu des synchronisations évitées (qui imposent un overhead) et de la place gagnée (surtout pour les données par ressource). Lorsque le recalcul demande un temps long et variable (e.g. parcourir une liste pour connaître le nombre d'éléments), alors il est préférable de stocker le nombre d'éléments malgré la redondance.

[13] ce qui n'est pas vrai si `NB_COL` est une puissance de 2, mais dans notre cas `NB_COL` vaut 80

### 5.4.5 Regroupement en données contiguës

**Présentation** Cette technique consiste à regrouper des données interdépendantes (i.e. sur lesquelles il y a un même invariant) dans le même mot, ou groupe de mots adjacents. Cela facilite leur mise à jour atomique.

Cela permet également d'économiser de la place, donc fait partie du principe de minimisation des données (§ 5.3.1).

**Exemple** Les coordonnées $x$ et $y$ d'un émulateur de terminal VT100 se trouvent dans des intervalles respectifs de $[\![\, 0, 79 \,]\!]$ et $[\![\, 0, 24 \,]\!]$, et occupent donc moins d'un octet chacune. Dans un mot de 32 bits, on peut faire rentrer ces deux coordonnées ; cela permet de les mettre à jour atomiquement.

Un autre exemple courant est l'utilisation de flags : il est fréquent qu'un mot dispose d'au moins quelques bits de libre (par exemple s'il contient une adresse ou un index), ce qui permet de stocker quelques flags dans les bits restants.

**Intérêt** Le matériel garantit d'écrire les mots de manière atomique. Des invariants sur les données peuvent être facilement respectés grâce à cela, y compris les invariants de forme canonique « $\Longleftrightarrow$ » (§ 4.1.3.2).

Cette technique est limitée à un seul mot, sauf si le processeur permet la mise à jour atomique de plusieurs mots, auquel cas elle devient extrêmement intéressante. Un support de CAS2 (CAS sur 2 mots, dont dispose le Pentium), de hardware transactional memory [HM93] (encore très peu courant), ou d'instructions ll/sc qui couvrent plusieurs mots en sont des exemples.

Cette technique a également d'autres avantages, comme d'améliorer la localité du cache.

### 5.4.6 Exportation de synchronisations

#### 5.4.6.1 Présentation

**Principe** Nous définissons cette nouvelle technique, qui suit le principe de délégation de la gestion de la cohérence (§ 5.3.2). Elle part de l'observation que souvent, certains clients d'un service doivent se coordonner pour accéder au service, sinon ils produisent des résultats incohérents. Le service n'a alors pas à assurer la cohérence de ces données, car cela serait redondant.

**Exemple** Par exemple, la lecture simultanée par deux programmes de l'entrée standard clavier est indéterministe : il est impossible de prévoir quel programme recevra quel appui sur une touche du clavier. Pourtant, les systèmes traditionnels comme UNIX font une synchronisation dans le service de terminal, ce qui permet de s'assurer que chaque caractère ne sera lu que par un seul programme.

Le fait que l'entrée clavier soit lue simultanément par plusieurs programmes est déjà source d'indéterminisme et est un bug, donc assurer cette propriété d'atomicité est inutile. La synchronisation est assurée par les programmes eux-mêmes (en

particulier les shells), qui font en sorte qu'un seul programme lise simultanément l'entrée standard[14].

La synchronisation dans le service de terminal est redondante, et peut donc être éliminée. Elle ne sert qu'à s'assurer de certaines propriétés quand les programmes sont déjà buggués.

**Conditions d'application** Ce sont d'abord les conditions d'applications du principe de délégation de la gestion de la cohérence (§ 5.3.2) : on ne peut pas l'appliquer si cela cause un problème de sécurité où si la cohérence ne peut pas être maintenue par les clients[15].

De plus la majorité des données du service, qui sont partagées mais dont la cohérence peut être déléguée, peuvent être directement exportées au client suivant principe d'exportation des données (§ 5.3.1). Comme ce principe offre d'autres avantages, il vaut mieux exporter toute une donnée plutôt que seulement la gestion de sa cohérence.

Les données qui restent sont principalement des buffers de données d'entrée/sortie (accédé sous UNIX par `read` et `write`), qui ne peuvent pas être exportés, comme le buffer réseau, les buffers disques, etc. Cette technique est fondamentale pour elles, car comme les volumes sont potentiellement important la synchronisation dans ce cas se fait souvent avec des sleeplocks.

#### 5.4.6.2 Conséquences pour les clients

**Mise en place des garanties d'exclusivité** Souvent, l'accès à une "partie" du service doit se faire de manière *exclusive* , c'est à dire qu'il ne peut y avoir qu'un seul client qui fasse des requêtes sur cette partie.

L'idée est de profiter de cette exclusivité pour supprimer des synchronisations dans le service redondantes (voir Figure 5.4).

L'exclusivité dans l'accès aux ressources peut être assuré par les clients de deux manières :

- *spatiale*, lorsque les clients accèdent à des ressources différentes (e.g. utilisent des ports I/O différents), où a des des parties différentes d'une même ressource (e.g. écrivent à des endroits différents du même fichier) ;
- *temporelle*, lorsque les clients accèdent aux mêmes ressources, mais à des instants différents (e.g. différents programmes accédant au service clavier, ou un programme multithreadé écrivant dans un même socket réseau).

Il est facile pour l'espace utilisateur de s'assurer de l'exclusivité.

[14] Avec l'affichage graphique (e.g. serveur X) plusieurs programmes partagent effectivement l'accès au clavier. Mais le demultiplexage se fait à l'intérieur du serveur X : on a ainsi une "ressource clavier" par fenêtre, auquel un seul programme peut simultanément accéder, et l'argument tient encore.

[15] Notons que pour les données dont la cohérence doit être assurée pour raison de sécurité, il est possible de seulement détecter la présence de mises à jours concurrentes sans pour autant faire de synchronisation. On renvoie aux clients une erreur dans ce cas là.

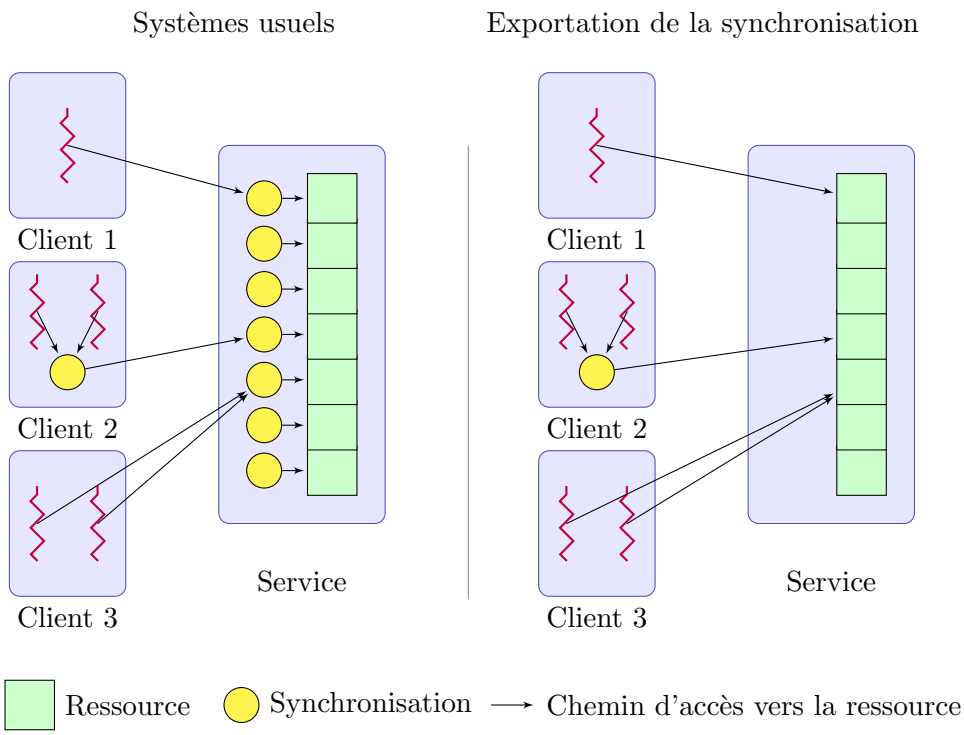



FIG. 5.4 – Dans les systèmes usuel, chaque ressource est synchronisée, parfois à l'aide d'un sleeplock. L'exportation de la synchronisation s'appuie sur le fait que le partage d'une ressource impose de toute façon l'exclusivité de l'accès à la ressource pour obtenir un résultat soit cohérent. Il permet de retirer l'overhead de la synchronisation pour les clients qui n'en ont pas besoin, et d'utiliser des méthodes alternatives de synchronisation. Ici, les ressources du client 1 ne sont pas partagées ; le client 2 est multithreadé et se synchronise avec un mutex ; le client 3 synchronise ses accès aux ressources autrement (e.g. exclusion temporelle ou accès à des parties différentes d'un fichier)

L'exclusivité spatiale peut être assurée par la politique d'allocation (e.g. différents programmes utilisent des ports TCP différents).

L'exclusivité temporelle n'est nécessaire que si plusieurs clients accèdent à la même ressource, et peut alors être assurée par l'emploi de primitives de synchronisations. Cela demande une confiance mutuelle entre les clients, ce qui est généralement le cas quand ils utilisent la même ressource[16]

Les systèmes d'exploitations usuels imposent déjà l'exclusivité ; ainsi Anaxagoros n'impose pas de contrainte supplémentaire aux clients. On tire juste parti de cet exclusivité pour supprimer des synchronisations.

Notons qu'il est impossible pour des programmes situés hors du service de savoir où en est l'exécution dans le service. L'exclusivité temporelle ne peut donc donc se faire qu'à la granularité de l'appel de service.

[16]C'est le cas e.g. dans les systèmes UNIX, où ce cas se rencontre soit dans des programmes multithreadés ou multiprocessus, soit des programmes qui se synchronisent par le système de fichier.

```
void
write_all( unsigned int fd, const char *buf, unsigned int size)
{
  unsigned int nb_to_write = size ;
  const char *src = buf ;

  while( nb_to_write  != 0)
    {
      int n = write( fd, src, nb_to_write) ;
      nb_to_write -= n ;
      src += n ;
    }
}
```

Listing 5.2 – Fonction pour écrire dans un buffer sous UNIX

**Garantie d'atomicité** L'objection que l'on peut faire à ce principe est que l'emploi de primitives de synchronisation dans le service garantit l'atomicité des accès, et que ne pas avoir cette garantie va demander des modifications importantes dans le client. Nous montrons ici que c'est faux, et que cette garantie est inutile pour les programmes courants. Nous concentrons la démonstration sur les buffers de données d'entrée sortie.

Tout ce que peut faire le service en utilisant des primitives de synchronisation, c'est de garantir l'atomicité des appels à `read` ou `write`, mais pas l'ordre dans lequel les appels sont faits. Dans la majorité des cas, le client doit donc séquentialiser les ordres des lectures et écritures (par exemple si plusieurs threads écrivent dans le même socket réseau).

Il y a des cas où cet ordre n'est effectivement pas important, comme pour le fichier de log. L'important dans ce cas est juste que le message apparaisse en entier, et seule la garantie d'atomicité est nécessaire.

Mais regardons les appels systèmes dans UNIX. Les appels systèmes `read` (resp. `write`) prennent en argument le nombre d'octets à lire (resp. à écrire) et rendent le nombre d'octets lus (resp. écrits). Rien ne garantit que ces deux nombres sont égaux. Ainsi, pour écrire l'intégralité du message dans UNIX, il faudra écrire un code comme dans le Listing 5.2, i.e. faire des appels systèmes répétés jusqu'à ce que l'intégralité du message soit écrit.

Rien ne garantit donc que l'écriture du message à l'aide de la fonction `write_all` sera atomique, et c'est bien au client de s'en assurer. Dans tous ces cas, la garantie d'atomicité apportée par le service ne sert à rien.

#### 5.4.6.3 Avantages

**Suppression de synchronisations** Ce principe permet surtout de ne pas synchroniser certaines données, et évite des synchronisations inutiles et difficiles à réaliser.

En particulier, elle supprime les synchronisations de "séquentialisation", i.e. qui

permettent à plusieurs programmes d'accéder de manière séquentielle aux données, comme dans UNIX pour séquentialiser l'accès à l'entrée clavier. Pour ce type de synchronisation, les temps d'attentes peuvent être longs et les systèmes usuels utilisent généralement un sleeplock, ce qui ce qui rend la politique d'ordonnancement imprévisible.

**Liberté dans le choix de la méthode de synchronisation** Ce principe énonce que la conception doit laisser libre la manière dont la synchronisation est faite, i.e. ne pas imposer de type de synchronisation particulière. C'est une conception qui suit le principe d'indépendance des politiques d'ordonnancement CPU (§ 3.1.1.1).

Par exemple, le cas très courant où la ressource n'est pas partagée n'a pas besoin de synchronisation, ce qui économise l'overhead de la synchronisation. De même dans le cas où la synchronisation se fait par le temps (cela est souvent utilisé dans OASIS [CDA+05] ; Roscoe mentionne également l'idée [Ros95, p.96]).

Un autre exemple où cela est utile est l'utilisation de threads utilisateur (e.g. [MSLM91, ABLL92]) : lorsque le verrou utilisateur pour une ressource est utilisé, l'ordonnanceur utilisateur change de thread utilisateur ; nul besoin de rajouter un verrou dans le noyau.

**Simplification de l'implémentation du service** Le dernier avantage est la simplification de l'implémentation du service : ne pas utiliser de verrou fait toujours autant de code d'économisé, de sources de bugs dans un programme de confiance, et d'espace mémoire gagné : avec un verrou par ressource, la consommation mémoire de l'ensemble des verrous n'est pas négligeable.

## 5.5 Conclusion

Notre volonté de simplifier, sécuriser et rendre indépendant l'allocation des ressources aux applications se paie par une augmentation de la complexité au niveau des services : synchronisation difficile, mémoire révoquée, pas d'allocation dynamique... Notre approche est donc *globalement simple/localement complexe*, ce qui est en opposition avec beaucoup de systèmes actuels dont les parties sont simples mais très interdépendentes.

Cette complexité peut être maîtrisée à l'aide des outils adéquats, présenté ici. Tout d'abord, un ensemble de mécanismes de synchronisation adapté aux problèmes de concurrences dans les services. Ensuite une méthodologie et des techniques pour réduire la taille des services et les problèmes de synchronisation qui s'y posent. Le problème d'écriture de service devient alors parfaitement faisable.

Ces méthodes font suivre au service un modèle passif, comme une petite librairie partagée, plutôt qu'un modèle actif de serveur de ressource. Les services effectuent des opérations relativement simples, ce qui simplifie également leur implémentation : allocation statique avec zones de partages bien identifiés (Figure 3.11 page 131) ; minimisation des données à manipuler (§ 5.3.1) ; décomposition en opérations élémentaires simples (§ 5.3.3).

Les services sont petits et peu complexes, et on peut peut tracer la relation entre invariants de cohérence et invariants de sécurité, cela permet de minimiser les besoins en synchronisation (ce qui facilite la programmation non-bloquante et augmente le parallélisme).

Chapitre 6

# Conclusion

## 6.1 Résumé

Nous avons soutenu l'idée que la majorité des problèmes pour l'intégration d'applications hétérogènes en niveaux de sécurité est un manque de support du système d'exploitation. En suivant strictement un ensemble de principes de conception, on peut implémenter un système qui permette de réaliser efficacement (en performance et en coût de développement) et de manière sécurisée cette intégration.

Ces principes sont bien connus pour ce qui concerne la protection de l'intégrité et de la confidentialité (principes de Saltzer et Schroeder [SS74]), même s'il est relativement difficile de construire un système qui les respecte totalement. Ils sont insuffisants pour la protection contre le déni de service, et en particulier pour garantir la sécurité de tâches temps réel dures.

À cette fin, nous proposons un nouveau principe, *l'indépendance des politiques d'allocation*. Ce principe stipule que toutes les décisions d'allocation devraient être prises par un module de politique séparé. Ce principe permet en particulier de complètement choisir sa politique, et d'avoir des allocations *déterministes* (i.e. qui suivent les décisions d'allocations).

Ce principe permet la mise en place d'une méthodologie générale pour l'exécution de tâches temps réel de différents niveaux de criticité. Il suffit dans un premier temps d'évaluer les besoins en ressource de chaque tâche critique, puis de mettre au point une politique qui garantisse que ces besoins seront remplis. L'indépendance des politiques d'allocation permet d'allouer les ressources au plus près des besoins des tâches critiques, ce qui laisse plus de ressources aux tâches non-critiques, et améliore donc in fine les performances. Cette méthodologie est uniforme pour tous les niveaux de criticité, ce qui facilite l'intégration des différentes applications et la rend plus simple à appliquer.

Nous définissons également d'autres principes complémentaires, qui permettent d'obtenir un système où les ressources nécessaires sont prévisibles, où les décisions d'allocations le sont également, et où une surveillance de la bonne exécution des tâches critiques peut être réalisée. Ces principes permettent ensemble un haut niveau de sûreté de fonctionnement pour les tâches critiques, indépendemment de la

présence des autres tâches (critiques ou non).

Nous avons conçu et implémenté un système qui suit ces principes. La structuration et séparation en domaines de protection permet d'assurer le confinement des erreurs et l'extensibilité du système. Le mécanisme de contrôle d'accès par capacité permet de raisonner formellement sur le partage des ressources, et dispose d'une implémentation efficace où toutes les opérations se font en temps constant. Le mécanisme de prêt de ressource permet la communication avec un service partagé sans perturber les autres tâches qui utilisent ce service ; en particulier ce mécanisme rend la communication avec le service partagé invulnérable aux dénis de service.

Une attention particulière a été portée au développement et à l'implémentation des services partagés, qui sont les points d'entrées à des vulnérabilités potentielles du système. Ces services sont d'abord séparés en domaines de protection distincts (structure micronoyau), afin de limiter l'impact d'une défaillance. Ils sont également minimisés en taille, ce qui les rend plus sûrs (et plus flexibles). Enfin, ils doivent gérer leurs ressources (prêtées ou non) de manière correcte pour éviter les failles de sécurité et les dénis de services. Nous avons proposé différents principes de structuration et d'implémentation des services qui permettent de suivre les principes de haute sécurité et de respect de l'indépendance des politiques d'allocation.

Une des principales difficultés concernant l'écriture de service est le problème de la synchronisation. Dans les systèmes usuels, les synchronisations sont résolues en faisant dormir certaines tâches. Cela rend les problèmes d'ordonnancement particulièrement difficiles (problèmes dits d'inversion de priorité), et le principe d'indépendance des politiques d'allocation est en particulier conçu pour les éviter. Certaines synchronisations par exclusion mutuelle sont également faites en masquant les interruptions, une opération privilégiée dangereuse pour tout le système.

La résolution des problèmes de synchronisation dans les services se fait par deux manières. Tout d'abord, les techniques de minimisation des services permettent de retirer un grand nombre de besoins en synchronisations. En particulier, l'exportation de la synchronisation consiste à seulement vérifier que les opérations se font dans un ordre correct (et interdire les opérations incorrectes), plutôt que de l'assurer. Les autres techniques consistent à minimiser les données et les traitements sur ces données.

Les besoins en synchronisation restants sont résolus par l'application de primitives de synchronisation originales, qui ne demandent ni à dormir ni à masquer les interruptions. Nous avons en particulier plusieurs primitives qui implémentent l'exclusion mutuelle, et une qui résout les problèmes d'update sur un objet libéré. Ces primitives sont disponibles à l'ensemble des applications, ce qui permet l'écriture de tâches temps réel multithreads qui ne souffrent pas du problème d'inversion de priorité.

Les techniques de conception et d'implémentation sont issues de l'expérience pour l'implémentation d'un prototype fonctionnel, Anaxagoros, dont les premières mesures de performances sont très encourageantes. La simplicité et le minimalisme global du système nous permettent de penser que le système complet sera très efficace.

## 6.2 Travaux futurs

### 6.2.1 Travaux à court terme

**Démonstrateur POSIX-like** Même si notre prototype est opérationnel, il reste encore à le compléter pour une démonstration plus « visuelle » et des mesures de performances pour faire la comparaison avec d'autres systèmes. Nous souhaiterions terminer l'implémentation d'un petit système analogue à POSIX. Cela demande l'implémentation d'un service de système de fichier (un prototype est en cours), et d'une « libOS » qui permette la gestion des file descriptors UNIX, fork, etc. en s'appuyant sur nos mécanismes de création de tâche et mémoire virtuelle existants.

**Support d'applications critiques de type OASIS** Pour démontrer le support d'application hétérogènes, il serait également intéressant de pouvoir exécuter des tâches temps réel critiques de type OASIS en parallèle avec d'autres tâches. Ce travail de portage peut être fait relativement rapidement.

OASIS est particulièrement intéressant à implémenter, parce que le modèle de tâches est indépendant des politiques d'ordonnancement, sans être aussi simple que le modèle de tâches périodiques. En particulier, il peut être facile d'implémenter et d'évaluer l'amélioration de performance de l'ordonnancement « non-critique prioritaire sur le critique » décrit en section 2.1.4.2.

**Ajout de nouveaux services** Un démonstrateur plus complet demande l'incorporation d'autres services partagés. Les services actuels concernent la mémoire virtuelle et la protection, l'ordonnancement, les ports d'entrées sorties et les numéros d'interruptions pour les services incorporés dans le noyau ; l'affichage en mode VGA texte, et le clavier pour les services en espace utilisateur.

Deux services sont particulièrement intéressants à implémenter. Le premier est le réseau. En particulier, la bande passante avec le protocole TCP/IP dépend de la latence des émissions des « ack » TCP. Ceci est un bon exemple de tâche non-critique qui devrait être prioritaire sur le critique (§ 2.1.4.2). Un premier prototype de service réseau (implémenté dans le noyau) a été réalisé en stage par Sébastien Hocquet l'été 2008. L'implémentation finale devrait ressembler à celle de Sinha et al. pour EROS [SSS04].

Le deuxième est l'affichage. Les services d'affichages comme X sont typiquement vulnérables aux attaques par déni de service, et il est difficile de prendre en compte le temps passé à exécuter des requêtes d'affichage [FH05, § 2.5]. Notre mécanisme de prêt de temps CPU peut aider à résoudre ce problème, et constituerai donc un bon exemple.

### 6.2.2 Travaux à moyen terme

**Version multiprocesseur** Un travail extrêmement intéressant à moyen terme consisterait à porter le système sur multiprocesseur. Notre conception des services semble propice à d'excellentes performances sur multiprocesseur, car notre conception réduit le nombre de mécanismes de synchronisation nécessaire et la taille des

sections critiques. En particulier, notre système de mémoire virtuelle est quasiment entièrement non-bloquant (wait-free), les seules attentes occasionnées étant lors de la « destruction » d'objets mémoires en cours d'utilisation. Cela signifie que des opérations comme `mmap` ou `sbrk` peuvent très bien passer à l'échelle de l'augmentation du nombre de cœurs.

Un problème à plus long terme est que notre modèle de communication est basé sur la mémoire partagée, qui n'est peut-être pas le modèle le plus approprié pour les machines massivement multiprocesseur ou distribuées. On pourrait cependant imaginer avoir des grappes de processeurs qui partagent une même mémoire et communiquent via les moyens que nous avons défini ; et ces services qui feraient de la communication en envoyant des messages.

Enfin, il serait intéressant de voir quelles transformations apporter sur des machines à mémoire partagée NUMA.

**Ajout du support des notifications** Les notifications sont importantes pour l'implémentation d'appels systèmes bloquants. Pour l'instant, nous n'en avons pas besoin grâce à l'utilisation du polling, mais celui-ci est relativement inefficace (surtout pour des tâches qui seraient en attente de beaucoup d'évènements, comme un serveur web qui attendrait avec `select`). Nous pensons que c'est l'une des dernières modifications du noyau qui manque pour pouvoir implémenter un service POSIX-like complet (sans virtualisation).

**Utilisation pour la paravirtualisation** Notre système peut assez rapidement être employé comme VMM pour virtualiser d'autres systèmes. À moyen terme, nous visons la paravirtualisation de Linux. Cela permettrait l'exécution de tâches critiques de manière concomitante avec celle de processus Linux.

**Prototype pré-industriel** Le prototype a été implémenté avec rigueur. Nous nous sommes en particulier efforcés d'utiliser la fonction `assert` pour tester chaque hypothèse sur laquelle nous nous reposons dans le code (e.g. les prérequis des fonctions). En conséquence durant ces trois années de développement, tous les bugs rencontrés étaient des erreurs de codage triviales qu'on retrouvait immédiatement.

Nous avons également systématiquement utilisé une fonction `check` (qui s'utilise comme `assert`, et lance une exception en cas d'erreur) pour vérifier les prérequis des données issues des autres tâches (e.g. si les arguments sont dans les intervalles corrects). Cette fonction est extrêmement utile pour produire sans effort du code sécurisé et lisible. En particulier, comme le noyau et les services ne font pas d'allocation dynamique de ressource, en faisant les vérifications au début de chaque méthode il n'arrive quasiment jamais d'erreur temporaires, simplifiant beaucoup le code [Ber07, § 4.2].

Mais certains passages sont manquants (en ce cas, la fonction `unimplemented` a été appelée pour s'en rendre compte). En particulier, lorsqu'on rencontre une erreur, le système bloque plutôt que d'envoyer l'erreur à un moniteur paramétrable comme décrit section 3.1.3.1.

Un dernier problème peut être celui de la portabilité avec d'autres compilateurs. Les primitives pour la programmation concurrente non-bloquante n'est pas standardisée en C ; pour être performant, il a fallu utiliser des fonctions non incluses dans le standard. Nous les avons systématiquement implémentées avec des macros, qui peuvent être facilement remplacées par des appels à des fonctions assembleurs pour obtenir du code conforme C99 si nécessaire.

### 6.2.3 Travaux à long terme

**Confidentialité** Un axe de développement du système est le support d'applications qui demandent un gros niveau de confidentialité. Le support de la dynamicité, le respect des principes de sécurité pourrait permettre au système de prétendre à un haut niveau de sécurité. Il serait intéressant d'étendre le système pour réaliser une politique de sécurité multiniveaux.

**Méthode formelle de développement de services partagés** Un axe de développement très intéressant serait une méthodologie formellement fondée de développement des services partagés. On a vu l'importance des services partagés, car le partage de services entre deux tâches indépendantes sont les seuls moyens pour une de ces tâches d'impacter la deuxième. Il est donc très important de les sécuriser.

Une voie vers cela est la preuve de programme. Certains points développés dans cette thèse premettent de présager que cela est possible en suivant les techniques de développement proposées au chapitre 5. En effet :

- les services partagés sont relativement petits (en ligne de code) ;
- ils proposent un petit nombre de méthodes simples, et courtes ;
- ils ne font pas d'allocation de mémoire dynamique, et ne sont pas récursifs ;
- les services sont parallèles, mais les zones de partages des données sont petites et surtout bien identifiées.

Ces restrictions facilitent grandement la tâche de preuve de programme.

On a vu également qu'il était relativement difficile d'écrire un service partagé. On a vu enfin que comme nos services partagés étaient petits, il existait une relation directe entre les spécifications du programme et les propriétés à assurer par les synchronisations. Une autre voie (complémentaire) consiste en la génération automatique de pilotes à partir de spécifications du matériel, et de l'OS, similairement à Termite [RCK+09].

**Étude d'un hardware adapté pour l'architecture** Enfin, l'étude de mécanismes hardware adaptés à l'architecture pourrait être menée.

Une piste de réflexion est la prédictabilité des temps d'exécution en liaison avec la réalisation matérielle. En effet si l'architecture logicielle de l'OS que nous avons décrite élimine quasiment toute imprévisibilité dans les temps d'exécution des applications, celle liée au hardware reste. Il est cependant possible de contrôler le

comportement du hardware, en allouant des quotas de temps CPU assez long, ou en allouant la mémoire de manière à partitionner les caches [LHH97]. Ces pistes sont à explorer.

Une autre piste consiste en l'optimisation des coûts liés aux fréquents changements d'espace d'adressages, dus à la décomposition en domaines. Un processeur adapté pourrait minimiser les coûts des appels de services et des commutations de contexte MMU.

Enfin, il serait également intéressant de s'assurer que les IOs exportés à un service ou système paravirtualisé sont sûrs pour le reste du système (e.g. la programmation d'un DMA peut avoir des conséquences sur tout le système). Des mécanismes comme les IO/MMU représentent un pas dans la bonne direction.

Annexe A

# Implémentation et preuve du système de capacité

Nous décrivons ici les preuves de validité de notre mécanisme de capacité, décrit section 3.3.

## A.1 Implémentation bas-niveau

Nous allons commencer par montrer comment sont implémentés les capacités dans le noyau. Les opérations que nous allons décrire dans cette sous-section concerne des opérations de bas niveau du noyau, invisibles à l'espace utilisateur. Ces fonctions de bas niveau seront ensuite utilisées pour construire l'interface de haut niveau.

Nous présentons d'abord un modèle de l'implémentation bas-niveau, puis nous montrerons que notre implémentation est conforme à ce modèle. Cela donnera une connaissance précises des propriétés à respecter dans l'implémentation, ce qui servira pour l'implémentation multiprocesseur.

### A.1.1 Modèle d'implémentation

**Définition A.1** (Système de capacité)**.** On définit un système à capacité par

- un ensemble d'*objets*, qui peuvent être dans l'état *valide* ou *invalide* ;
- un ensemble de *services*, qui sont des objets particuliers. Chaque objet $o$ est servi par un unique service $s$ appellé le *le service de* $o$ ;
- un ensemble de *capacités*, qui soit sont invalide, soit pointent vers un objet.

Et par 5 opérations `bind( cap, objet)`, `invoke( cap)`, `revoke( objet)`, `revoke( service)` et `invalidate( cap)`, qui sont *atomiques* et liées de la manière suivante :

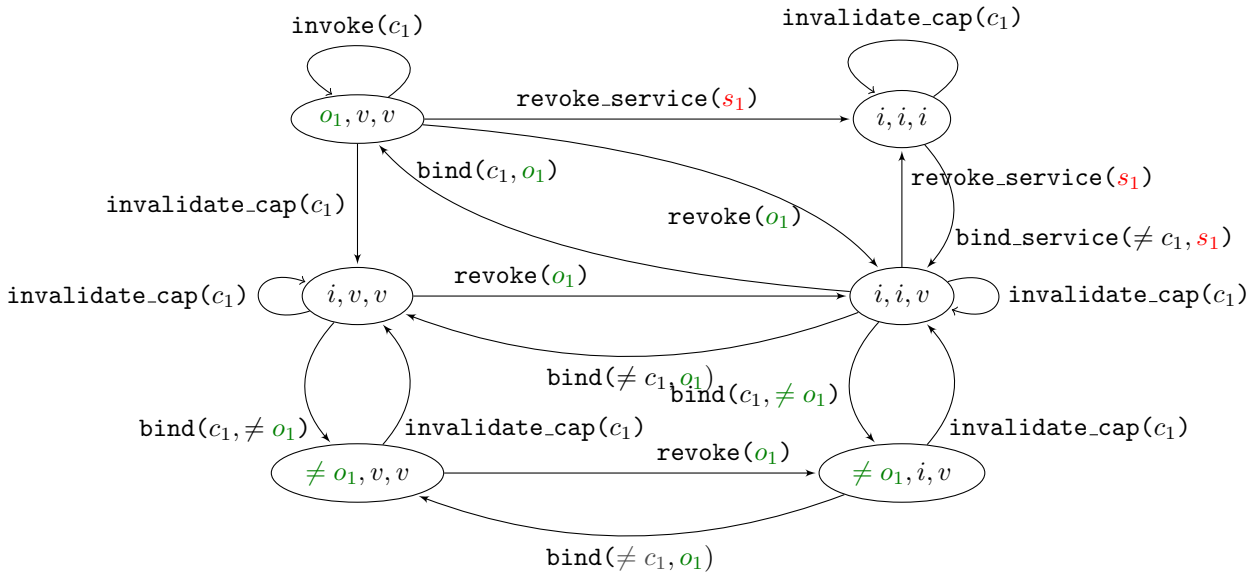



FIG. A.1 – Automate des transitions possibles du système de capacité, pour une capacité $c_1$, un objet $o_1$, et un service $s_1$. Chaque état contient un triplet contenant l'état respectif de $(c_1, o_1, s_1)$. $i$ signifie invalide, $v$ valide, et $o_n$ que la capacité pointe vers $o_n$. $\neq$ signifie que le paramètre est un autre paramètre valide, différent de $o_1$ ou $c_1$. Lorsque le paramètre `objet` est un service, on le met en valeur en rajoutant `_service` et en mettant le paramètre en rouge.

- `bind( cap, objet)` peut être appellé seulement si le service de `objet` est valide, `objet` est invalide, et `cap` est invalide ; après appel `objet` devient valide et `cap` pointe vers `objet` ;
- `revoke( objet)`, lorsque `objet` n'est pas un service, fait passer `objet` de l'état valide à l'état invalide, et fait passer toutes les capacités qui pointaient vers `objet` vers l'état invalide
- `revoke( service)` fait passer `service` de l'état valide à l'état invalide, fait passer toutes les capacités qui pointaient vers `service` vers l'état invalide, et fait passer tous les objets servis par `service` vers l'état invalide ;
- `invalidate( cap)` fait passer `cap` vers l'état invalide ;
- `invoke( cap)` réussi (i.e. renvoie `true`) ssi `cap` pointe vers `objet`, si `objet` est servi par `service`, et si `objet` et `service` sont tous les deux valides.

**Notes**

- Pour se représenter un système de capacités, on peut prendre l'exemple de système de fichiers UNIX : les descripteurs de fichier sont des capacités, les fichiers sont des objets, et les systèmes de fichiers sont des services.

- La création d'un objet et d'une capacités sont liées, car faites simultanément par l'opération `bind`. I.e. on ne peut pas créer un objet sans créer la capacité qui lui est associée.
- Ces opérations sont naturelles si on fait l'analogie avec le monde réel : une capacité est une clé, un objet est une serrure sur une porte et un service est l'immeuble dans lesquelles sont les portes. Les clés ne fonctionnent plus si on les détruit (i.e. les invalide), si modifie la serrure d'une porte (i.e. révoque l'accès à la porte) ou si on interdit l'accès à l'immeuble (i.e. révoque l'accès à l'immeuble).
- Un objet est toujours servi par le même service : la relation « être le service de » reste immuable dans le temps.
- Il y a une opération de bas niveau importante que nous avons omise par souci de simplification, qui est la copie de capacités, `copy( capdest, capsrc)`, qui copie l'état de `capsrc` dans `capdest`. Nous en parlons section A.1.2.4.
- L'implémentation est simple lorsqu'on n'appelle jamais `bind`, `revoke` ou `invalidate` : le contrôle d'accès peut être stocké dans un ensemble de données statiques (e.g. matrice de Lampson [Lam71]). Cette implémentation serait adaptée à des systèmes statiques.
- Lorsque `bind` est appellé, c'est conceptuellement un nouvel objet qui est créé à la même place. En particulier, si on créée un nouveau service $s$, les objets servis par $s$ restent invalides.

**Implémentation naïve** La manière directe d'implémenter ce modèle consiste à avoir un champs « valide » ou « invalide » dans les objets et capacités du système. Mais cette approche nécessite de parcourir toutes les capacités pointant vers un objet pour les invalider lorsque l'objet devient invalide ; ainsi la révocation se fait en temps proportionel en nombre de capacités. Cette implémentation requiert de maintenir des listes chainées entre les capacités et l'objet pointé, pour les parcourir quand l'objet est révoqué. seL4 [EDE08][KEH+09, § 3.2] implémente une variante de ce modèle.

Cette approche a pour principal problème que la révocation se fait en temps proportionel au nombre de capacités, ce qui peut être source de vulnérabilité et va à l'encontre de notre principe de temps d'exécution prédictible. Dans l'implémentation seL4, l'invocation se fait également en temps non-constant.

Ces problèmes ne se posent pas lorsqu'il n'y a pas de révocation (ou de création) d'objets ou de capacités, i.e. quand le système est statique. Mais dans ce cas, on peut considérer que toutes les capacités et objets du système sont valides, et il devient inutile de stocker le caractère valide ou invalide d'un objet.

Ceci motive notre implémentation plus avancée, dont toutes les opérations sont faites en temps constant court, et qui va avoir une implémentation multiprocesseur extrêmement scalable.

### A.1.2 Implémentation monoprocesseur

#### A.1.2.1 Structures de données

**Pointeurs estampillés** Notre approche est fondée sur la notion de pointeurs estampillés (§ 3.3.3.1). L'idée est la suivante : une capacité est un pointeur sur un objet, accompagné de droits. Par pointeur, on entend un identifiant d'un objet qui est unique dans tout le système. Un exemple commode de tel identifiant unique est l'adresse physique.

Mais lorsque l'objet est supprimé ou remplacé, la capacité ne doit plus pointer vers l'objet, ce que l'adresse physique seule n'assure pas. L'idée est alors d'associer à la capacité et à l'objet un même mot de passe, et de ne permettre à la capacité d'accéder à l'objet pointé que si le mot de passe est égal. Pour révoquer l'accès à toutes les capacités qui pointent vers l'objet, il suffit de changer le mot de passe de l'objet. Pour s'assurer qu'on ne réutilise pas un ancien mot de passe, on peut utiliser une date (qui augmente toujours) : c'est l'estampille.

**Note :** Attention au sens de date : il faut que le mot de passe pour un objet soit unique. Si on utilise la seconde courante, plusieurs objets créés dans la même seconde auraient la même date, et l'identifiant ne serait plus unique pas. On peut par contre utiliser le nombre de cycles processeurs depuis le démarrage de la machine par exemple ; ou plus simplement, un compteur qui s'incrémente à chaque fois que l'on crée une capacité. Pour éviter un overflow, on prend pour ce compteur une valeur de 64 bits.

Ce mot de passe n'est pas dur à deviner pour un attaquant, mais cela ne lui servira pas, car seul le noyau peut créer des capacités. On garantit pour cela qu'il est impossible pour un programme de créer une capacité sans passer par l'appel `bind`[1] du noyau.

**Vérifications à effectuer** Quand une capacité est invoquée, on a deux choses distinctes à vérifier.

- Que le client peut accéder au service. Cette propriété doit être vérifiée par le noyau, car cela créerait sinon une brêche dans la confidentialité.
- Que le client peut accéder à l'objet. Cette propriété n'a pas besoin d'être vérifiée par le noyau, et on peut donc déléguer l'opération de vérification au service.

Les systèmes à capacité font usuellement cette vérification en une seule fois ; pour cela un tableau des ressources global est utilisé, qui contient la liste des objets utilisables. Lorsqu'il a été vérifié (par le noyau ou le matériel) que le client pouvait accéder à l'objet, on initie la connexion entre le client et le service.

Le problème de cette méthode est qu'elle demande d'utiliser une entrée dans le tableau des ressources global, qui est fini. Cela entraîne soit un déni de service

[1] ou `copy` lorsque la copie de capacité est supportée

global, ou soit de définir une politique d'allocation pour les entrées dans la table, ce qui n'est pas pratique.

C'est la raison pour laquelle notre système implémente des tableaux des ressources locaux à chaque service, qui ne souffrent pas de ce problème. La vérification est ainsi décomposée en deux étapes :

1. Le noyau vérifie que la capacité du client lui permet d'accéder au service

2. Le service vérifie que la capacité du client lui permet d'accéder à la ressource

La vérification 1., faite par le noyau, compare l'estampille de création de la capacité à celle de création de l'objet. Si l'estampille de la capacité est supérieure à celle du service, alors la capacité a été créée après le service. Par conséquent, le service n'a pas été révoqué depuis que la capacité est créée.

La vérification 2. est libre ; un service peut très bien ne pas implémenter cette vérification si il le préfère. La révocation des objets est aussi libre. Pour ce faire, on permet au service de modifier directement l'estampille de l'objet qu'il révoque (c'est la raison pour laquelle on a besoin d'une vérification séparée par le noyau). Si l'estampille de la capacité est supérieure à celle de l'objet, alors la capacité a été créée après l'objet, et l'objet n'a pas été révoqué depuis que la capacité est créée, et la capacité pointe bien vers l'objet.

**Structures de données** Le Listing A.1 fournit des structures de données pour un modèle de l'implémentation simplifié.

```
// low représente les bits de poids faible, et up ceux de poids fort.
typedef struct { uint32_t low ; uint32_t up ; } timestamp_t ;

typedef struct { timestamp_t timestamp ; service_t service ; } *object_t ;
typedef object_t service_t ;

typedef struct capability {
 service_t service ;
 object_t object ;
 timestamp_t timestamp ;
} *capability_t ;
```

Listing A.1 – Structures de données

On suppose qu'on ne peut accéder aux données que par des mots de 32 bits. Les timestamps sont des compteurs sur 64 bits, mais pour lesquels on a besoin d'accéder à chacun des deux mots individuellement.

On considère souvent les `timestamp_t` comme une donnée scalaire (de type `unsigned long long`), et on permet des opérations (non-atomiques), comme l'incrément ou l'affectation, sur ces données. L'implémentation réelle demanderait l'utilisation d'une `union`, que nous avons retiré pour plus de lisibilité.

#### A.1.2.2 Implémentation

Voici le code (simplifié) des différentes opérations telles qu'effectuées par le noyau pour créér ou détruire des capacités. Le code est basé pour une implémentation en 32 bits ; elle serait plus simple en 64 bits.

L'atomicité est assurée par le fait que ces opérations sont faites dans le noyau, et toutes appellées avec des interruptions masquées.

On suppose l'existence d'une fonction `service` qui donne pour chaque objet $o$ son service $s$. Dans l'implémentation réelle, cette fonction n'existe pas : le service d'un objet est toujours connu dans le contexte des appels à ces opérations.

#### A.1.2.3 Preuves

Différentes propriétés peuvent être prouvées, mais la plus intéressante que nous souhaitons montrer est que *cette implémentation respecte bien le contrôle des invocations* ; i.e. qu'on peut invoquer une capacité ssi elle pointe sur un objet, et qu'il n'y a pas eu de révocation de l'objet, de son service, ou d'invalidation de la capacité entretemps.

**Définitions** Il est assez facile de comprendre la raison pour laquelle cette implémentation fonctionne, une fois comprise la notion de pointeur estampillé (§ 3.3.3.1). Cependant la preuve formelle présente quelques difficultés.

Une de ces difficultés est la question du vocabulaire, car on doit distinguer certaines propriétés qu'on prouve être finalement équivalentes.

Une autre difficulté est qu'on ne peut pas définir l'état valide ou invalide d'une capacité ou d'un objet en fonction de son contenu, parce que cet état dépend des invocations précédentes.

On considère donc un historique (i.e. une suite ordonnée) $\mathcal{H}$ d'appels à `invoke`, `invalidate`, `revoke`, ou `bind` sur les objets, capacités ou services du système (on suppose l'ensemble des objets, capacités ou services constant pour cette preuve).

**Définition A.2.** Soit $o$ un objet, et $s$ son service. $o$ est *valide* si :

1. il y a eu une opération `bind(.,`$o$`)` ;
2. cette opération n'a pas été suivie d'opération `revoke(`$o$`)` ou `revoke(`$s$`)`.

$o$ est sinon dit *invalide*.

Notons que comme `bind(.,`$o$`)` suppose que $s$ est valide au moment de l'appel, on en déduit que $o$ valide implique que $s$ est également valide.

Notons que suivant cette définition, initialement tous les objets sont invalides [2]

**Définition A.3.** Soit $c$ une capacité, $o$ un objet, et $s$ le service de $o$. $c$ pointe vers $o$ si :

1. la dernière opération de type `bind` concernant $c$ est `bind(` $c$`,` $o$`)` ;

[2] En réalité, il faut qu'il existe un service initial valide $S$ tel que tous les services ont pour service $S$ ; en particulier $S$ a pour service $S$, sinon on ne peut créer aucun objet.

```
// Marche pour les services comme pour les autres objets.
void revoke( object_t obj) { obj->timestamp.up = 0xffffffff ; }

void invalidate( capability_t cap) { cap->timestamp.up = 0 ; }

timestamp_t fetch_timestamp( void)
{
  static timestamp_t current_timestamp = 0x100000000 ; // = 2^32
  return timestamp++ ;
}

// Prérequis : cap est invalide.
void write_to_cap( capability_t cap, service_t service,
                   object_t object, timestamp_t timestamp)
{
  cap->timestamp = timestamp ;
  cap->service = service ;
  cap->object = object ;
}

// Prérequis : le service de obj est valide.
void bind( capability_t cap, object_t obj)
{
  timestamp_t ts = fetch_timestamp() ;
  obj->timestamp = ts ;

  invalidate( cap) ; // Permet pas avoir pour prérequis que cap soit invalide.
  write_to_cap( cap, service( obj), obj, ts) ;
}

// Renvoie true ssi la capacité peut être invoquée.
// Si true est renvoyé, local_copy contient une copie de cap
bool invoke( capability_t cap, capability_t local_copy)
{
  if( cap->timestamp.up == 0) return false ;

  local_copy->service = cap->service ;
  local_copy->object = cap->object ;
  local_copy->timestamp = cap->timestamp ;

  if( local_copy->timestamp < local_copy->service->timestamp) return false ;
  // Cette vérification est fait en espace utilisateur dans l'implémentation réelle.
  if( local_copy->timestamp  != local_copy->object->timestamp) return false ;
  return true ;
}
```

Listing A.2 – Implémentation du système de capacité sur monoprocesseur

2. cette opération n'est pas suivie par une opération `invalidate(`$c$`)`, `revoke(`$o$`)` ou `revoke(`$s$`)`.

$c$ est sinon dit *invalide*.

En particulier, les capacités sont initialement toutes invalides.

**Définition A.4.** Une capacité $c$ *peut invoquer* un objet $o$ lorsqu'un appel à `invoke(`$c$`)` renvoie `true`, et qu'après appel on a `local_copy->object ==` $o$.

Notons qu'additionellement, il faut que `local_copy->service` soit égal au service de $o$.

**Objectif** Ce que l'on va chercher à prouver devient :

$$\text{une capacité } c \text{ peut invoquer un objet } o \iff c \text{ pointe sur } o$$

i.e. que l'implémentation répond bien aux spécifications.

Par abus de notation, on désigne souvent par `variable` la *valeur courante* de la variable `variable`.

Pour simplifier cette preuve, on ne se préoccupe pas des états initiaux (qui seront supposés respecter les propriétés énoncées).

On prend comme hypothèse que les prérequis du code sont respectés (i.e. les opérations appellées avec des arguments corrects et dans un ordre respectant les prérequis.)[3].

**Propriétés sur `current_timestamp`**

**Hypothèse A.1.** Toutes les valeurs de `current_timestamp` sont strictement inférieures à `0xffffffff0000000`

*Justification.* La seule écriture de `current_timestamp` est dans `fetch_timestamp`. Pour que `current_timestamp` atteigne `0xffffffff0000000`, il faudrait plus 500 ans même si `fetch_timestamp` était appellé toutes les nanosecondes, et on suppose que cela n'arrivera jamais. □

**Propriété A.1.** *`current_timestamp` croit toujours : si à un moment on a `current_timestamp` $= t$, après ce moment on aura `current_timestamp` $\geq t$.*

*Démonstration.* L'hypothèse A.1 implique qu'il n'y a pas d'overflow ; et la seule écriture de `current_timestamp` est dans `fetch_timestamp` qui l'incrémente. □

**Propriété A.2.** *On a toujours `current_timestamp` $\geq$ `0x100000000`*

*Démonstration.* Car `current_timestamp` croit toujours et vaut initialement cette valeur. □

[3]En particulier, on suppose que des appels successifs à `bind` sur une même capacité $c$ sont séparés par des `invalidate(`$c$`)`, même si l'implémentation réelle permet d'omettre cette hypothèse en invalidant $c$ inconditionellement avant chaque `bind`

### Propriétés sur les capacités

**Propriété A.3.** *Pour qu'une capacité* `cap` *puisse invoquer un objet* `obj`*, il faut que le dernier appel à* `bind` *avec* `cap` *en premier argument, aie eu* `obj` *en deuxième argument.*

*Démonstration.* Pour que `cap` puisse invoquer `obj`, il faut que `local_copy->object` soit égal à `obj` par définition. Or `cap->object` ne peut être modifié que par `bind`, et doit donc prendre `obj` en deuxième argument. □

**Propriété A.4.** *Pour toute capacité* `cap`*, on a toujours* `cap->timestamp` $<$ `current_timestamp`

*Démonstration.* Immédiat en considérant les écritures dans `cap->timestamp` sont faites seulement par `bind` et `invalidate`. `bind` incrémente `current_timestamp` et met dans `cap->timestamp` l'ancienne valeur. `invalidate` met `cap->timestamp` à une valeur inférieure à `0x100000000`, qui est inférieur ou égal à `current_timestamp`. □

**Propriété A.5.** *Si le dernier appel sur* `cap` *est* `invalidate`*, alors* `invoke(cap)` *rendra toujours faux.*

*Démonstration.* Immédiat : dans ce cas il n'y a pas eu d'autre d'écriture dans `cap->timestamp.up` de puis `invalidate` qui y a mis `0`. Ainsi le premier test de `invoke` retournera faux. □

**Propriété A.6.** *Si on a appellé* `invalidate` *sur une capacité* `cap` *qui pointait vers* `obj`*, alors* `cap` *ne peut plus invoquer* `obj` *à moins d'un nouvel appel à* `bind( cap, obj)`*.*

*Démonstration.* Si le dernier appel sur `cap` était `invalidate`, ce cas est couvert pas la propriété A.5. Les autres écritures de `cap` sont des appels à `bind(cap, ·)` ; pour que `local_copy->object` puisse valoir `obj` après l'appel à `invoke` il faut que ce deuxième argument ait été `obj`. □

### Propriétés sur les objets et services

**Propriété A.7.** *Soit* $t$ *la valeur de* `current_timestamp` *lors d'un appel à* `revoke(obj)`*. Alors à partir de ce moment, on a toujours* `obj->timestamp` $> t$*.*

*Démonstration.* Si `obj` est invalide, alors la dernière opération dessus est `revoke`, qui a mis dans `obj->timestamp` une valeur supérieure à `0xffffffff00000000`, qui est supérieur à `current_timestamp` par l'hypothèse A.1.

Si non, il y a eu en dernier un appel à `bind` avec `obj` en deuxième argument. Dans ce cas, `obj` a été mis à `current_timestamp` qui croit toujours (propriété A.1). □

Note : un service étant un objet, la propriété A.7 est également vraie sur les services.

**Propriété A.8.** *Si on a appellé* `revoke` *sur un objet* `obj`*, alors une capacité* `cap` *ne peut pas ou plus l'invoquer à moins d'un nouvel appel à* `bind( cap, obj)`*.*

*Démonstration.* Soit $t$ la valeur de `obj->timestamp` au moment de l'appel à `revoke(obj)`. Toutes les capacités `cap` telles que `cap->object` = `obj` ont été créées à l'aide de `bind(cap, obj)` (propriété A.3). Comme `current_timestamp` est croissant, cela signifie que `cap->timestamp` $\leq t$.

Les propriétés A.7 et A.1 indiquent qu'à partir de ce moment, on aura toujours `obj->timestamp` > `current_timestamp` > $t$. Ainsi, le troisième test de `invoke` rendra false tant que `cap->timestamp` n'est pas modifié. Examinons comment `cap->timestamp` peut être modifié.

Si on appelle `invalidate( cap)`, `invoke` ne réussira pas (propriété A.5). Si on appelle `bind( cap, obj2)` sur un autre objet, le test pourrait réussir mais permettra d'invoquer `obj2` et pas `obj` (propriété A.3).

Donc `cap` ne peut plus invoquer `obj` à moins d'un nouvel appel `bind( cap, obj)`. □

**Propriété A.9.** *Soit `cap` une capacité. Si on a appelle `revoke` sur un service `serv`, alors pour tout objet `obj` servi par `serv`, `cap` ne pourra invoquer `obj` que si `serv` est recréé, puis qu'il y a un appel à `bind( cap, obj)`.*

*Démonstration.* Si $t$ est la valeur de `current_timestamp` lorsque `serv` est révoqué. Après l'appel à `revoke(serv)`, on aura toujours `serv->timestamp` > $t$ (propriété A.7).

De plus on a `cap->timestamp` < $t$ (propriété A.4) tant que `cap` n'est pas modifié. Ainsi, le deuxième test échouera toujours, tant que la capacité n'est pas modifiée.

Pour que ce deuxième test réussisse, il faut augmenter la valeur de `cap->timestamp`, ce qui nécessite un appel à `bind(cap,·)`. Pour pouvoir invoquer un objet `obj` servi par `serv`, il faut que le deuxième argument soit `obj` (propriété A.3).

Les prérequis de la fonction `bind` assure que cet appel n'est fait que si `serv` est valide, et donc recréé. □

### Propriétés finales

**Propriété A.10.** *Une capacité `cap` qui ne pointe pas sur `obj` ne peut pas invoquer `obj`.*

*Démonstration.* Si `cap` ne pointe pas sur `obj`, cela signifie qu'il y a eu un ou plusieurs appels à `invalidate(cap)`, `revoke(obj)` ou `revoke(serv)` avec `obj` servi par `serv`, sans qu'il y ait ensuite eu d'appel à `bind(cap,obj)`. Les propriétés A.6,A.8 et A.9 addressent ces cas respectifs. □

**Propriété A.11.** *Si une capacité `cap` pointe sur `obj`, alors elle peut invoquer `obj`.*

*Démonstration.* Si `cap` pointe sur `obj`, cela signifie qu'il y a eu en dernier appel à `bind(cap,obj)` qui ne soit suivi d'aucun appel à à `invalidate(cap)`, `revoke(obj)` ou `revoke(serv)` (où `serv` est le service de `obj`). Dans ce cas, on va avoir `cap->timestamp` = `obj->timestamp`; `cap->object` = `obj` et `cap->service` = `serv`.

En particulier, le troisième test de `invoke` va réussir; comme `obj->timestamp >= 0x100000000` (car `current_timestamp` est croissant par la propriété A.1), le premier test va également réussir.

De plus, suivant les prérequis de `bind`, `serv` est valide et l'appel `bind(·,serv)` s'est fait avant celui à `bind(·,obj)`. Comme `current_timestamp` est croissant, on a donc `serv->timestamp < obj->timestamp`. Comme de plus `cap->timestamp = obj->timestamp`, le deuxième test va également réussir.

Donc `cap` peut invoquer `obj`. □

**Theorème A.12.** *Une capacité `cap` peut invoquer `obj`, ssi elle pointe sur `obj`.*

*Démonstration.* Prouvé par les propriétés A.10 et A.11. □

#### A.1.2.4 Implémentation réelle

L'implémentation réelle diffère de cette implémentation modèle par plusieurs aspects.

Tout d'abord, la C-list, qui contient les capacités, peut également être invalidée. Cette opération peut être assimilée à un `invalidate` sur toutes les capacités de la C-list, bien qu'il n'y ait pas d'écriture dans les capacités. Pour prendre en compte cette opération dans la preuve, il faudrait dire qu'une capacité est invalide lorsque sa C-list est invalide.

L'implémentation de `invoke` est en réalité partagée entre l'espace utilisateur et l'espace noyau. Le dernier test est réalisé uniquement par le service en espace utilisateur, et se fait sur la copie (`local_copy`) réalisée par le noyau.

L'implémentation de `invoke` différencie également les capacités vers des services en espace utilisateur de ceux en espace noyau. Les services dans l'espace noyau sont supposés irrévocables, et certains tests ne sont donc pas à faire. Le test `cap->timestamp.up == 0` n'est pas non plus présent : l'invalidation de capacités se fait en modifiant un champs `destination_type` de la capacité, qui indique si la capacité est vers un service noyau, un service utilisateur, ou est invalide.

L'invocation et la révocation d'objets (avec `revoke`) font, dans l'implémentation réelle, appels au schéma de synchronisation use/destroy (§ 5.2.4). L'invocation d'un objet est suivi d'un « use » sur l'objet, tandis que la révocation fait un « destroy ». Ainsi, tous les utilisateurs sont révoqués. Ceci a été décrit en section 3.3.3.2. Notons que la révocation de capacité, elle, ne fait pas d'appel à destroy : i.e. si la capacité pour accéder à un service est supprimée pendant l'appel de service, l'appel de service continue normalement.

Enfin, l'implémentation réelle permet la copie de capacités, `copy(`$c_d$`,`$c_s$`)` qui copie $c_s$ vers $c_d$. L'implémentation de cette fonction en monoprocesseur est triviale : il s'agit d'un simple `memcpy` de $c_d$ vers $c_s$. La preuve est impactée par l'ajout de cette opération : une capacité $c$ peut désormais pointer vers $o$ aussi lorsque que la dernière opération sur $c$ est `copy(`$c$`,`$c'$`)`, avec $c'$ qui pointe vers $o$ au moment de la copie. Les modifications de la preuve sont simples mais laborieuses.

### A.1.3 Implémentation en multiprocesseur

En multiprocesseur, on pourrait évidemment transposer la solution monoprocesseur par l'ajout d'un spinlock global du noyau pour toutes les opérations sur les capacités. Mais une telle solution constituerait un goulot d'étranglement sur ces opérations, et réduirait le passage à l'échelle.

Cette section permet de présenter les intéractions entre invariants de sûreté, de fonctionnalité, et de cohérence ; et montre pas à pas comment l'analyse fine des besoins de cohérence, permet de réaliser une implémentation très efficace. L'idée est que la réalisation de la preuve (ou du schéma de preuve) donne une compréhension fine du programme, qui permet de réaliser une implémentation où les besoins en synchronisation sont minimaux. Notons qu'il n'est pas toujours nécessaire de faire la preuve pour réaliser cette implémentation performante, mais celle-ci est le meilleur moyen de comprendre le programme intégralement.

L'implémentation proposée repose sur le fait que les interruptions sont masquées (ce qui permet une garantie de terminaison, voir chapitre 4), pour réaliser une implémentation des capacités de manière entièrement wait-free et très efficace. Elle donne donc un exemple concret d'utilisation de ces garanties partielles.

On suppose l'existance d'une fonction `CAS`, qui effectue un compare and swap de manière atomique. Voir [Her90] pour une description de compare and swap.

**Note :** Note implémentation suit le principe de linéarizabilité ([HW90]), qui stipule que le comportement du système est toujours analogue à un comportement séquentiel "peu éloigné" du comportement réel. Pour autant, il est difficile de voir dans le code un instant atomique où les changements se font.

**Analyse des besoins** La principale différence en multiprocesseur est que l'exécution des différentes fonctions n'est plus atomique.

Dans les propriétés de la preuve précédente, certaines sont dérivées d'autres propriétés, certaines ne traitent que de l'ordre du programme. Ces propriétés sont toujours respectées en multiprocesseur.

Les propriétés qui peuvent ne plus tenir sont celles qui concernent les valeurs des variables, surtout lorsqu'elles font plusieurs mots ou concernent plusieurs variables.

Il faut en particulier veiller à la croissance et l'unicité de la valeur renvoyée par `fetch_timestamp` ; l'intégrité de `local_copy` ; le respect continu des relations entre les timestamps, alors que ceux-ci font plusieurs mots (un problème similaire a été exposé par Lamport [Lam77]).

**Adaptation des hypothèses** Notre implémentation monoprocesseur initiale utilisait la valeur de timestamp `0xffffffffffffffff` pour indiquer un objet invalide et `0` pour une capacité invalide. Mais le timestamp fait deux mots (sur une architecture 32 bits), et l'affectation de valeur aux timestamps nécessitait donc deux écritures, et était non-atomique.

Pour faciliter l'écriture, nous avons adapté certaines hypothèses ou propriétés. Ainsi, tous les timestamps au dessus de `0xffffffff00000000` indiquent un objet invalide, et tous les timestamps en dessous de `0x100000000` une capacité invalide.

Cela permet de passer une capacité ou un objet de l'état valide à invalide (ou réciproquement) en changeant uniquement le mot de poids fort de timestamp. On peut ainsi invalider une capacité ou un objet de manière atomique, en une simple écriture (cela est également plus efficace pour la version monoprocesseur).

Cela change l'hypothèse A.1 et la propriété A.2, mais les preuves peuvent être facilement adaptées pour supporter cela.

Pour la version multiprocesseur, nous réadaptons nos hypothèses : désormais, tous les timestamps au dessus de `0xfffffffe00000000` indiquent un objet invalide, et tous les timestamps en dessous de `0x200000000` une capacité invalide. Les adaptations pour prendre en compte ces changements sont triviales (`fetch_timestamp` et `invoke` sont affectés).

Cette adaptation des hypothèses, possible par la connaissance de la preuve, et qui permet de faire des invalidations atomiquement, va être très utile pour tous les algorithmes présentés par la suite.

### A.1.3.1 Accès aux capacités

**Écriture de capacité** Pour garantir l'intégrité des écritures de capacité, on se base sur la garantie de terminaison procurée par le masquage des interruptions, pour implémenter un « lock » (en réalité un spinlock non tournant (§ 5.2.1.1)) qui empêche les écritures simultanées au même endroit. Pour maximiser le parallélisme, on met un lock par capacité ; et pour économiser de la place on met le lock en utilisant les valeurs invalides du timestamp.

Notons que si on ne réussit pas à acquérir ce lock, on quitte en retournant une erreur au lieu de rentrer en attente active, comme pour un spinlock classique. Une autre possibilité était de simplement retourner, i.e. d'oublier l'opération d'écriture (§ 5.2.1.1). Nous préférons retourner une erreur à simplement retourner ici car un cas d'écriture parallèle dans des capacités rend la valeur indéterministe, ce que nous considérons être une erreur.

```
// Prérequis : cap est invalide
void write_to_cap( capability_t cap, uint32_t service,
                   uint32_t object, timestamp_t timestamp)
{
  // Tente d'acquérir un "lock" sur l'emplacement de la capacité
  if( !CAS( &cap->timestamp.up, 0, 1))
    {
      // La capacité n'était plus invalidée, il y a opérations concurrentes
      // ce qui est une erreur.
      throw( ECONCURRENCY) ; // gestion d'exception.
    }
   cap->service = service ;
   cap->object = object ;
   cap->timestamp.low = timestamp.low ;
   // Libère le "lock" et valide la capacité atomiquement.
   cap->timestamp.up = timestamp.up ;
}
```

Ce lock garantit qu'une capacité est toujours écrite en entier lorsqu'elle est validée. Comme on écrit dans une capacité (à part pour l'invalider) qu'ainsi, on est garantit que les capacités valides ont toujours été correctement écrites par un seul processeur.

**Invalidation de capacité** L'invalidation de capacité peut se faire atomiquement, en modifiant `cap->timestamp.up`. Mais il faut prendre garde à ne pas écraser le lock d'écriture de capacité que cette variable peut contenir.

```
void invalidate( capability_t cap)
{
  uint32_t up = cap->timestamp.up ;
  if( up > 1) CAS( &cap->timestamp.up, up, 0) ;
}
```

Si `cap->timestamp.up` est supérieur à 1, la capacité était valide, et on l'invalide. Si `cap->timestamp.up` vaut 0, la capacité est déjà invalidée et l'invalidation est inutile. Si `cap->timestamp.up` vaut 1, il y a une écriture en cours : ne rien faire est équivalent à la séquence d'évènements où l'appel à `invalidate` aurait eu lieu avant cette écriture, donc est une action correcte. Si `CAS` échoue, il y a eu une écriture entre la lecture et l'exécution de `CAS` : on n'a donc rien écrit, ce qui est équivalent à la séquence d'évènement où l'invalidation à lieu avant l'écriture par l'autre thread à `cap->timestamp.up`.

Dans tous les cas, on peut transformer l'historique des évènements en un historique séquentiel valide qui respecte le principe de linéarizabilité [HW90].

**Lecture de capacité** On désire maintenant pouvoir lire les capacités de manière concurrente aux écritures. La lecture de capacité se fait uniquement dans `invoke`, où il faut surtout respecter l'intégrité de `local_copy`, et renvoyer le bon résultat.

L'idée est de réutiliser les écritures sur `cap->timestamp` pour détecter les écritures concurrentes. Si il y a une écriture concurrente, on renvoie faux : une capacité ne peut passer de l'état "pointer vers $o_1$" à "pointer vers $o_2$" qu'en passant par un état "invalide", donc on considère tout état intermédiaire comme invalide.

On peut faire cette détection sans modifier l'écriture et l'invalidation, avec un peu d'astuce. On remarque que la fonction `write_to_cap` modifie `cap->timestamp.up` en début d'écriture, et `cap->timestamp.low` en fin d'écriture. Si `timestamp` n'a pas été modifié pendant la lecture, alors le reste de la capacité non plus, et la lecture est intègre. Si non, il y a une écriture concurrente, la capacité est dans un état intermédiaire et est donc invalide.

On utilise ici `cap->timestamp.low` comme compteur de version (de manière similaire à [Lam77], mais avec un seul compteur). L'usage de `cap->timestamp.up`, qui sert également de lock pour sérialiser les écritures dans la capacité, est également subtil.

La détection des écritures concurrentes ne coûte presque rien : une lecture mémoire et une comparaison supplémentaires (dont le résultat est presque toujours vrai, donc le branchement est facilement prévisible) suffisent.

```
bool invoke( capability_t cap, capability_t local_copy)
{
  uint32_t old_low = cap->timestamp.low ;
  uint32_t service = cap->service ;
  uint32_t object = cap->object ;
```

```
  uint32_t new_up = cap->timestamp.up ;
  uint32_t new_low = cap->timestamp.low ;

  if( new_up <= 1) return false ;
  if( new_low  != old_low) return false ;

  local_copy->object = object ;
  local_copy->service = service ;
  local_copy->timestamp.up = new_up ;
  local_copy->timestamp.low = new_low ;

  if( local_copy->timestamp < local_copy->service->timestamp) return false ;
  if( local_copy->timestamp  != local_copy->object->timestamp) return false ;
  return true ;
}
```

**Propriété A.13.** *`invoke` renvoie true* $\iff$ *`cap` est valide* $\wedge$ *`local_copy == cap`.*

*Démonstration.* Si la capacité est valide et qu'il n'y a pas eu d'écriture concurrente, le fait que l'algorithme renvoie `true`, est trivial (la seule addition par rapport au monoprocesseur étant une lecture et une comparaison vraie dans ce cas là).

On montre maintenant que si `invoke` retourne `true`, alors `cap` est (ou était) valide, et `local_copy == cap` (pas d'écriture concurrente).

Si `old_low` = `new_low`, soit il y a eu des écritures dans `cap->timestamp.low` et les valeurs sont à nouveau les même. Comme on n'écrit dans cette variable qu'avec la fonction `write_to_cap`, cela signifie qu'on a rempli `cap->timestamp.low` à l'aide des 32 bits de poids faible de `current_timestamp`. Donc `current_timestamp` a été augmenté $2^{32}$ fois, ce qu'on prend par hypothèse impossible si `invoke` est exécuté avec les interruptions masquées (le temps est trop grand). Donc il n'y a pas eu d'écriture dans `cap->timestamp.low`.

Si il y a eu une écriture ou invalidation concurrente entre la lecture de `old_low` et celle de `new_up`. Alors soit il y a eu en dernier une invalidation, auquel cas `new_up` = `0`. Soit il y a eu en dernier un début d'écriture, auquel cas `new_up` = `1`. Dans ces deux cas on renvoie `false`.

Soit il y a eu en dernier une fin d'écriture, auquel cas on a bien `new_up` > `1` mais on aurait eu aussi une écriture dans `cap->timestamp.low`, ce qu'on a montré impossible.

Comme aucune de ces opérations n'a pu se passer, il n'y a donc pas eu d'invalidation ou d'écriture concurrente jusqu'à la lecture de `new_up`.

Notons qu'il peut y avoir invalidation ou un début d'écriture avant la lecture de `new_low`, mais toutes les données sont déja lues, sauf `new_low` qui n'a pas encore été touché. On est donc garantit de l'intégrité de la capacité.

Donc `cap` est valide (prouvé en monoprocesseur) et la lecture est correcte. □

**Implémentation réelle** Ces algorithmes présupposent que les emplacements mémoires auxquels on accède contiennent bien des capacités. En réalité, tous ces

algorithmes sont donc encapsulés dans une section « use » d'un schéma de synchronisation use/destroy (§ 5.2.4) sur la C-list qui contient les capacités. C'est à dire que le code a une structure comme suit :

```
begin_use(clist) ;
// Invoque, lit ou écrit des capacités
end_use(clist) ;
```

Notons que la majorité des capacités auxquelles on accède sont celles du domaine ou du thread courant, pour lesquels on est déjà dans une section use (voir 5.2.4.2, utilisation à travers le noyau).

L'implémentation réelle nécessite également, bien sûr, l'ajout des différentes barrières de synchronisation en écriture et lecture.

#### A.1.3.2 Accès au timestamp et aux objets

**Récupération d'un nouveau timestamp** La récupération du nouveau timestamp peut se faire en rajoutant un petit spinlock autour de la fonction `fetch_timestamp`.

Le Pentium offrant une instruction CAS2 (CAS sur deux mots), cette instruction peut être facilement utilisée pour implémenter cette instruction de manière lock-free. Cet algorithme implémente une variante du single-word protocol de Herlihy [Her90], sur deux mots en utilisant CAS2, et en ayant la première lecture non-atomique (cela n'a pas d'importance).

```
timestamp_t fetch_timestamp( void)
{
  static timestamp_t current_timestamp = 0x200000000 ;

  do {
  // Note : lecture non atomique, mais c'est sans importance.
  timestamp_t timestamp = current_timestamp ;
  } while( !CAS2( &current_timestamp, timestamp, timestamp+1))

  return timestamp ;
}
```

Il est certainement possible de réduire la contention sur `current_timestamp`, par exemple en l'incrémentant de manière paresseuse (seul l'ordre relatif à la création de service (e.g. de domaines) compte). Ces travaux pourront être entrepris si cela cause un problème réel.

Enfin, d'autres valeurs peuvent être utilisées. Par exemple, si le compteur du nombre de cycle (TSC) des différents processeur est synchronisé, il semble possible d'utiliser ce compteur en ajoutant en suffixe le numéro du processeur.

**Révocation d'un objet** La révocation d'un objet se fait de manière atomique tout simplement :

```
// Marche pour les services comme pour les autres objets.
void revoke( object_t obj)
{
  obj->timestamp.up = 0xffffffff ;
}
```

C'est aux services de s'assurer que les objets sont complètement écrits lorsqu'ils sont validés. Le noyau l'assure pour ses services ; mais cette propriété est à la charge des services pour les objets standard. C'est une forme de délégation de la gestion de la consistance (§ 5.3.2).

**Note :** L'écriture des capacités est sensible : un attaquant peut se servir d'une défaillance peut violer des conditions de confidentialité (et permettre la communication non autorisée avec un service). Par contre, la gestion des timestamps des objets n'est sensible que pour le service qui sert ces objets, c'est la raison pour laquelle on peut les déleguer.

**Création d'un objet** Nous implémentons la "création" d'un objet dans le noyau (i.e. la mise à jour de son timestamp) de cette manière :

```
obj->timestamp.low = timestamp.low ;
obj->timestamp.up = timestamp.up ;
```

La raison est la même que pour l'écriture de capacité : l'espace utilisateur peut utiliser la valeur `0xfffffffe` en tant que lock dans `obj->timestamp.up`. Il peut ainsi sérialiser les écritures à un objet en utilisant un recoverable lock :

```
// Si le lock est révoqué, appelle la procédure CANCEL.
recoverable_lock( &obj->timestamp.up, 0xfffffffe, CANCEL) ;
obj->... = ... ;// Ecrit dans les autres champs d'objets
bind(obj, cap) ; //Appel système crée la capacité vers l'objet et libère le verrou.
```

### A.1.3.3 Code multiprocesseur complet

Nous rassemblons pages 238 et 239 le code complet pour la version multiprocesseur.

```
// Marche pour les services comme pour les autres objets.
void revoke( object_t obj) { obj->timestamp.up = 0xffffffff ; }

void invalidate( capability_t cap)
{
  uint32_t up = cap->timestamp.up ;
  if( up > 1) CAS( &cap->timestamp.up, up, 0) ;
}

timestamp_t fetch_timestamp( void)
{
  static timestamp_t current_timestamp = 0x200000000 ;

  do {
  // Note : lecture non atomique, mais c'est sans importance.
  timestamp_t timestamp = current_timestamp ;
  } while( !CAS2( &current_timestamp, timestamp, timestamp+1))

  return timestamp ;
}

// Prérequis : cap est invalide
void write_to_cap( capability_t cap, uint32_t service,
                   uint32_t object, timestamp_t timestamp)
{
  // Tente d'acquérir un "lock" sur l'emplacement de la capacité
  if( !CAS( &cap->timestamp.up, 0, 1))
    {
      // La capacité n'était plus invalidée, il y a opérations concurrentes
      // ce qui est une erreur.
      throw( ECONCURRENCY) ; // gestion d'exception.
    }
   cap->service = service ;
   cap->object = object ;
   cap->timestamp.low = timestamp.low ;
   // Libère le "lock" et valide la capacité atomiquement.
   cap->timestamp.up = timestamp.up ;
}
```

Listing A.3 – Implémentation du système de capacité sur multiprocesseur (1)

```
// Prérequis : le service de obj est valide.
void bind( capability_t cap, object_t obj)
{
  timestamp_t ts = fetch_timestamp() ;
  obj->timestamp.low = ts.low ;
  obj->timestamp.up = ts.up ;

  invalidate( cap) ; // Permet pas avoir pour prérequis que cap soit invalide.
  write_to_cap( cap, service( obj), obj, ts) ;
}

// Renvoie true ssi la capacité peut être invoquée.
// Si true est renvoyé, local_copy contient une copie de cap
bool invoke( capability_t cap, capability_t local_copy)
{
  uint32_t old_low = cap->timestamp.low ;
  uint32_t service = cap->service ;
  uint32_t object = cap->object ;
  uint32_t new_up = cap->timestamp.up ;
  uint32_t new_low = cap->timestamp.low ;

  if( new_up <= 1) return false ;
  if( new_low  != old_low) return false ;

  local_copy->object = object ;
  local_copy->service = service ;
  local_copy->timestamp.up = new_up ;
  local_copy->timestamp.low = new_low ;

  if( local_copy->timestamp < local_copy->service->timestamp) return false ;
  if( local_copy->timestamp  != local_copy->object->timestamp) return false ;
  return true ;
}
```

Listing A.4 – Implémentation du système de capacité sur multiprocesseur (2)

#### A.1.3.4 Ajout des copies

La copie de capacités consiste en une lecture suivie d'une écriture. L'écriture peut être faite avec la fonction `write_to_cap`. La lecture est faite de manière similaire à `invoke` ; si on détecte une écriture concurrente, on écrira tout simplement une capacité invalide (e.g. avec `timestamp == 0`) à l'emplacement de la nouvelle capacité.

#### A.1.3.5 Conclusion sur l'implémentation multiprocesseur

Nous avons montré dans cette section comment notre système de capacités pouvait être implémenté en multiprocesseur, de manière efficace et avec des synchronisations minimales. Notons en particulier, que mis à part pour récuperer un nouveau `timestamp`, toutes les opérations de lecture et d'écriture de capacités se font en temps court borné dans le temps, et il n'y a aucune attente active.

Cette section montrait également comment l'usage du masquage des interruptions seul pouvait amener à une implémentation très efficace.

Enfin, nous avons montré comment la preuve de la tâche permettait de relâcher certaines hypothèses, et était la clé pour permettre cette implémentation.

## A.2 Implémentation haut-niveau

Nous avons décrit l'implémentation haut-niveau (i.e. au niveau de l'interface avec l'espace utilisateur) en section 3.3.4. Il pourrait être intéressant de faire des preuves de certaines propriétés de cette implémentation haut-niveau, comme le fait qu'il est impossible d'augmenter les droits d'une capacité existante, de créer une capacité, sans passer par les « canaux officiels ». À leur tour, ces preuves pourraient être utilisées pour montrer des propriétés de l'espace utilisateur, comme l'absence de communication entre plusieurs partitions, ou le confinement [Lam73] d'un programme.

Cependant, les preuves de genre « il n'est pas possible que » ne peuvent être obtenues qu'en considérant l'intégralité du code du noyau (en effet, il suffit d'une défaillance dans n'importe quel appel service pour que cette propriété ne tienne plus). La preuve complète ne pourrait donc s'obtenir que sur un noyau petit (comme l'est Anaxagoros), et gagnerait à l'utilisation de prouveurs automatique (à la manière de Klein et al.[KEH+09]).

Annexe B

# Implémentations des verrous

Nous décrivons ici en plus amples détails l'implémentation des verrous de synchronisations, présentés au chapitre 4.

## B.1 Implémentations du schéma use/destroy

Nous avons présenté succinctement le schéma de synchronisation use/destroy en section 5.2.4. Ce schéma fournit une primitive de synchronisation pour les cas où un thread veut « libérer » de la mémoire en cours d'utilisation par un ou plusieurs autres threads ; sans restreindre le parallélisme des modifications apportées à cette mémoire.

Nous détaillons ici différentes implémentations de ce verrou avec quelques optimisations que nous avons mises en place. Sauf mention contraire, toutes les implémentations sont exécutées sur $M$ processeurs et avec les interruptions masquées (il y a donc au maximum $M$ threads qui s'exécutent en parallèle).

Ce verrou n'est utile qu'en multiprocesseur ; en monoprocesseur, si le processeur utilise de la mémoire, il n'est pas en train de la libérer, et le masquage des interruptions suffit. Il faut par contre toujours faire en sorte de ne pas utiliser de la mémoire déjà libérée.

### B.1.1 Implémentations du use/destroy lock

#### B.1.1.1 Implémentation de base

L'implémentation de base est donnée Figure B.1.1.1. Cette implémentation, pour le noyau, doit être exécutée avec les interruptions masquées. Les sections `atomic` indiquent que le code doit être exécuté atomiquement ; nous décrirons comment réaliser (ou relâcher) cela par la suite.

Il y a différents points à noter sur cette implémentation de base.

Tout d'abord, l'utilisation de spinlocks non tournants (§ 5.2.1.1) pour `lock.destroyer`. Le lock assure qu'il n'y a pas de destructions concurrentes des

```
error_t use( resource_t r)
{
  atomic{
    if( r->lock.destroyer)
      return EACCESS ;
    else r->lock.users ++ ; }
  ... // utilise ou modifie
  atomic{ r->lock.users -- ; }
}
```

```
error_t destroy( resource_t r)
 {
   atomic{
     if( r->lock.destroyer)
       return ECONCURRENCY ;
     else r->lock.destroyer = 1 ; }
   while( r->lock.users  != 0) ;
   ... // nettoie
   atomic{ r->lock.destroyer = 0} ;
}
```

FIG. B.1 – Implémentation théorique du use/destroy lock.

ressources ; mais plutôt que de sérialiser les destructions, il rapporte une erreur. En règle générale, détruire plusieurs fois à la suite une même ressource n'a pas d'intérêt, et est donc une erreur[1].

Les erreurs rapportées sont différentes. Dans le cas de destructions parallèles, nous rapportons une erreur de concurrence (`ECONCURRENCY`). Dans le cas d'une tentative d'utilisation pendant une destruction, nous rapportons une erreur d'accès (`EACCESS`) : cette erreur est la même que celle rapportée pour une tentative d'utilisation d'une ressource non initialisée. Autrement dit, le fait qu'une destruction soit en attente est équivalente au fait que la ressource est déjà détruite du point de vue des utilisateurs de la ressource. La prise du « destroy lock » est ainsi l'instant atomique de la destruction de la ressource selon la définition de Herlihy et Wing [HW90].

**Propriétés** Nous présentons la liste des propriétés que le verrou apporte de manière formelle, sous forme de preuve. Pour ces preuves, nous appellons « utilisation » la phase entre les deux sections `atomic` de la fonction `use`, « destruction » la phase entre les deux sections `atomic` de la fonction `destroy`, et « nettoyage » la phase entre la boucle d'attente et le dernier `atomic` de la fonction `destroy`.

**Propriété B.1.** *`lock.destroyer` est à 1, ssi il y a un thread en train de détruire la ressource.*

**Propriété B.2.** *Il y a au plus un thread en train de détruire la ressource.*

*Démonstration.* `lock.destroyer` est accédé en écriture seulement par la fonction `destroy`, qui implémente un spinlock classique. Ces deux propriétés viennent de là. □

**Propriété B.3.** *Tant que `lock.destroyer` est à 1, il n'y a plus de nouveaux utilisateurs.*

*Démonstration.* Ceci est assuré par le test à `lock.destroyer` au début de `use`. □

[1] Dans notre implémentation, il peut y avoir des cas de destructions parallèles quand un thread veut libérer une ressource, et que le service de politique d'allocation pour cette ressource lui a retiré et veut également libérer la ressource. Nous ne présentons pas ces cas dans un souci de lisibilité.

**Propriété B.4.** *`lock.users` contient le nombre d'utilisateurs en train d'utiliser la ressource.*

*Démonstration.* Garantit par l'atomicité des sections `atomic` de la fonction `use`. □

**Propriété B.5.** *Si un thread est en train de nettoyer la ressource, il n'y a pas de thread en train d'utiliser la ressource.*

*Démonstration.* Si un thread nettoye la ressources, il a attendu que `lock.users` soit à 0, donc qu'il n'y ai plus aucun utilisateur de la ressource (prop. B.4).

Or, `lock.destroyer` est à 1, car le thread est seul à détruire la ressource (prop. B.2), et il n'a pas encore mis `lock.destroyer` à 0. Donc il ne peut y avoir de nouvel utilisateur de la ressource (prop. B.3).

Ainsi, si un thread nettoye la ressource, il n'y a plus d'ancien utilisateur de la ressource et il ne peut y en avoir de nouveau tant que la ressource n'est pas nettoyée. □

**Propriété B.6.** *Si la durée d'utilisation de la ressource est bornée, alors la durée d'attente pour la destruction pour la ressource est également bornée.*

*Démonstration.* Cette propriété vient du fait que la destruction se fait en plusieurs phases. Dans la première, le bit destroyer est positionné à 1, ce qui empêche toute nouvelle utilisation (prop. B.3). Dans la deuxième, on attend que toutes les utilisations se terminent, ce qui prend un temps borné si toutes les utilisations se font en temps borné. Dans la troisième, le nettoyage se fait proprement dit ; et dans la quatrième, le verrou pour la destruction est libéré. □

**Propriété B.7.** *Si un thread est en train d'utiliser la ressource, il n'y a pas de thread en train de la nettoyer.*

*Démonstration.* Si un thread utilise la ressource, alors `lock.users` est différent de 0 (prop. B.4), et le restera tant que le thread n'aura pas fini d'utiliser la ressource. Il ne peut donc pas y avoir de nouveau nettoyage, puisque le nettoyage requiert que ce compteur soit à 0.

De plus, comme `lock.destroyer` est à 0, il n'y avait pas de destruction, donc de nettoyage en cours (prop. B.1). Donc il n'y a aucun nettoyage pendant la durée de l'utilisation. □

#### B.1.1.2 Implémentation réelle

La difficulté de l'implémentation réelle est la réalisation des sections `atomic`, et en particulier le fait qu'il faille vérifier une condition et faire une modification de manière atomique.

On peut résoudre ce problème en mettant `lock.destroyer` et `lock.users` dans un seul mot. `lock.destroyer` ne nécessite que 1 bit ; `lock.users` dépend du nombre de threads qui utilisent simultanément la ressource, mais ne peut dépasser le nombre de processeurs $M$ lorsqu'utilisé dans une section non-préemptible. On peut donc généralement mettre ces quantités dans un seul mot.

Il faut de plus disposer d'une instruction de type `CAS` (compare and swap), ou `LL/SC`. `FAA` (fetch and add) n'est pas obligatoire, mais permet d'éviter de boucler.

La Figure B.2 présente l'implémentation réelle, réalisée avec `CAS` et `FAA`.

```
error_t use( resource_t r) {
 do {
   uint old = r->lock ;
   if (old.destroyer)
      return EACCESS ;
   uint new = old ; new.users++ ;
 } while( !CAS(&r->lock, old, new)) ;
 ... // utilise ou modifie
FAA( &r->lock, -1) ; }
```

```
error_t destroy( resource_t r) {
 do {
   uint old = r->lock ;
   if (old.destroyer)
      return ECONCURRENCY ;
   uint new = old ;
   new.destroyer = 1 ;
 } while( !CAS(&r->lock, old, new)) ;
 while( r->lock.users  != 0) ;
 ... // nettoie
 r->lock.destroyer = 0 ; }
```

FIG. B.2 – Implémentation réelle du use/destroy lock

**Notes**

- L'utilisation de `FAA` ne modifie que la partie `users` du mot `lock` (on suppose ici que ces bits sont les moins significatifs). Ainsi, `FAA` ne modifiera pas la partie `destroyer`.
- Quand on écrit `r->lock.destroyer = 0` à la fin de `destroy`, on sait que `r->lock.users` vaut également 0. Ainsi, on pourrait écrire tout simplement `r->lock = 0`, et économiser ainsi plusieurs instructions.

Nous allons voir comment optimiser ce code dans une écriture plus élégante.

#### B.1.1.3 Économie du bit « destroyer »

Quand le use/destroy lock est exécuté avec les interruptions masquées, `lock.users` peut aller de 0 à $M$, le nombre de processeurs sur une machine. Si $M$ est une puissance de 2, ce qui est fréquent, le stockage de `lock.users` requiert $log_2(M) + 1$ bits, ce qui fait que le bit de poids fort est « sous-utilisé » : les valeurs de `lock.users` de $M + 1$ à $2M - 1$ ne sont jamais utilisées. Nous allons utiliser ces valeurs pour y stocker `lock.destroyer`. La Figure B.3 présente cette implémentation.

```
error_t use( resource_t r) {
 do {
   uint v = r->lock ;
   if (v >= M) //r en destruction
      return EACCESS ;
 } while( !CAS(&r->lock, v, v+1)) ;
 ... // utilise ou modifie
 FAA( &r->lock, -1) ; }
```

```
error_t destroy( resource_t r) {
 do {
   uint v = r->lock ;
   if (v >= M) //r en destruction
      return ECONCURRENCY ;
 } while( !CAS( &r->lock, v, v+M)) ;
 while( r->lock  != M) ;
 ... // nettoie
 r->lock = 0 ; }
```

FIG. B.3 – Implémentation optimisée du use/destroy lock

Dans cette implémentation, les valeurs de `lock` de 0 à $M$ indiquent que `lock` threads utilisent la ressource ; les valeurs de $M$ à $2M-1$ indiquent qu'un thread est en attente de destruction ou détruit la ressource, et que `lock`$-M$ threads l'utilisent.

La valeur `lock` $= M$ signifie donc à la fois « tous les CPUs utilisent la ressource » et « il n'y a aucun utilisateur de la ressource, un CPU la détruit ». Cela ne cause pas de problème, car si on fait le test `if( lock >= M)`, cela signifie qu'on n'est pas en train d'utiliser la ressource, donc qu'il y a au plus $M-1$ utilisateurs de la ressource. Le test ne réussi donc que si il y a une destruction en cours.

Ainsi, le « use/destroy lock » peut se stocker en $log_2(M)+1$ bits au lieu de $log_2(M)+2$. Cette implémentation simplifie également l'écriture du code réel, et le rend plus efficace (on élimine le besoin de manipuler des champs de bits).

## B.1.2 Ajout de signaux inter-threads

### B.1.2.1 Principe général et implémentation théorique

Pour assurer que la destruction se fasse en un temps court, il faut borner l'attente de destruction ; or les utilisations d'une ressource peuvent se faire sur un temps long. Dans le noyau (et en particulier dans le service mémoire), certains traitements « use » prennent un certain temps et se font dans une boucle. Dans ce cas, on peut vérifier dans la boucle si une destruction est en attente, et libérer le verrou à ce moment là.

Mais il est parfois peu pratique ou impossible de faire cela, et en particulier lorsqu'on exécute du code utilisateur (qui n'a aucune connaissance des ressources qu'il utilise, et qui ne peut donc pas vérifier périodiquement les destructions en attentes de ces ressources). Il faut alors que le destructeur « signale » aux utilisateurs que la ressource va être détruite, et que l'utilisateur traite ce signal en arrêtant d'utiliser la ressource.

Nous donnons Figure B.1.2.1 le code général permettant d'implémenter cela, avant d'observer les modifications et optimisations que l'on peut apporter.

```
error_t use( resource_t r)
{
  atomic{ if( r->destroyer)
            return EACCESS ;
          else add_to_set( r->users,
                           my_cpu()) ;}
  ... //utilise
  atomic{ remove_from_set( r->users,
                           my_cpu()) ;}
}
```

```
error_t destroy( resource_t r)
 {
   atomic{
     if( r->lock.destroyer)
       return ECONCURRENCY ;
     else r->lock.destroyer = 1 ;}
   send_signal( r->users) ;
   while( !empty( r->users)) ;
   ... // nettoie
   atomic{ r->destroyer = 0} ;
 }
```

FIG. B.4 – Implémentation du schéma use/destroy avec signaux

La modification majeure dans ce code est l'utilisation de la fonction `send_signal`, qui va envoyer le signal à tous les utilisateurs de la ressource. Cela nécessite de pouvoir retrouver ces utilisateurs ; ainsi on utilise l'ensemble `r->users`, et non

plus un compteur du cardinal de cet ensemble comme dans les implémentations précédentes.

L'ordre des opérations pour détruire la ressource est important : il faut d'abord positionner le destroy lock, qui empêche l'arrivée de nouveaux utilisateurs ; prévenir les utilisateurs courants ; puis attendre qu'il n'y ait plus d'utilisateurs. Un autre ordre pourrait conduire à des race conditions.

Notons que `send_signal` n'est pas atomique, et qu'on peut signaler à des threads d'arrêter d'utiliser une ressource qu'ils ont déjà arrêté d'utiliser. La routine de réception de signaux doit savoir gérer ce cas.

#### B.1.2.2 Implémentation avec un bitfield

Un bitfield, associé à chaque ressource, peut être utilisé pour représenter les CPUs qui utilisent la ressource : le bit numéro $n$ est à 1 ssi le CPU $n$ utilise la ressource. Le bitfield est une représentation compacte de l'ensemble des utilisateurs de la ressource.

**Bitfield et « bit destroyer » dans un même mot** Si le nombre de CPU est petit, on peut faire tenir tout le bitfield (un bit par processeur) plus le bit « destroyer » dans un seul mot. L'implémentation est donnée Figure B.1.2.2.

```
error_t use( resource_t r)
{
  do{
      uint old = r->lock ;
      if( old.destroyer)
        return EACCESS ;
      uint new = old ;
      new.users |= my_cpu() ;
  } while( !CAS( &r->lock, old, new)) ;
  ... // utilise
  FAA( &r->lock, - (1 << my_cpu())) ;
}
```

```
error_t destroy( resource_t r)
{
  do {
    uint old = r->lock ;
    if (old.destroyer)
      return ECONCURRENCY ;
    uint new = old ;
    new.destroyer = 1 ;
  } while( !CAS( &r->lock, old, new)) ;
   send_IPI( r->lock.users) ;
   while( r->lock.users != 0) ;
   ... // nettoie
   r->lock = 0 ;
}
```

FIG. B.5 – Implémentation du schéma use/destroy avec signaux et bitfield

Cette implémentation est la même que celle avec les compteurs, sauf qu'on utilise des bitfields, et qu'on rajoute l'envoi d'une IPI (interruption inter-processeur) aux processeurs qui utilisent la ressource au lieu de seulement attendre passivement qu'ils arrêtent de l'utiliser.

Notons l'emploi de `FAA` en fin d'utilisation de la ressource, qui permet d'éviter une boucle avec `CAS`.

**Bitfield et « bit destroyer » dans des mots différents** Il y a une difficulté supplémentaire si le bitfield et le « destroy bit » ne peuvent pas tenir dans un même

mot (par exemple si il y a plus de 32 processeurs). Le problème est que l'aquisition du lock user demande de vérifier la présence du bit « destroyer », et de modifier le bitfield, atomiquement, ce qui ne peut se faire si ils sont dans des mots différents.

Sur IA32, on peut utiliser CAS2 pour résoudre ce problème. Mais nous allons voir une solution plus générale à ce problème, qui relâche le besoin d'atomicité, et qui va ouvrir la voie à l'implémentation du use/destroy lock en espace utilisateur.

### B.1.3 Séparation de `lock.destroyer` et des utilisateurs de la ressource

#### B.1.3.1 Implémentation théorique

Si on veut placer `lock.destroyer` et l'ensemble des utilisateurs de la ressource dans des mots séparés, il faut remplacer l'opération atomique réalisée sur ces mots au début de la fonction `use`. La Figure B.1.3.1 fournit une solution.

```
error_t use( resource_t r)
{
  if( r->destroyer) return EACCESS ;
  atomic{ add_to_set( r->users,
                      my_cpu()) ;}
  if( r->destroyer) {
    atomic{ remove_from_set( r->users,
                             my_cpu()) ;}
    return EACCESS ; }
  ... //utilise
  atomic{ remove_from_set( r->users,
                           my_cpu()) ;}
}
```

```
error_t destroy( resource_t r)
 {
   atomic{
     if( r->destroyer)
       return ECONCURRENCY ;
     else r->destroyer = 1 ; }
   send_signal( r->users) ;
   while( !empty( r->users)) ;
   ... // nettoie
   atomic{ r->destroyer = 0} ;
 }
```

FIG. B.6 – Implémentation théorique du use/destroy pattern avec séparation

Dans cette nouvelle implémentation, les opérations atomiques ne concernent plus simultanément `r->destroyer` et `r->users`, ce qui permet de les séparer.

Son fonctionnement est le suivant. Le premier test à `r->destroyer` permet d'empêcher qu'il y aie de nouveaux utilisateurs quand un nettoyage est en cours ou en attente. Le deuxième test permet de contrer la race condition où un nouvel utilisateur et une nouvelle destruction sont faites simultanément. En d'autres termes, l'utilisation de la ressource est séparée en deux phases : une phase d' « inscription de demande d'accès », et une phase d'utilisation réelle (tout comme la destruction est faite en plusieurs phases).

Les propriétés B.3 et B.4 de la preuve changent légèrement, et deviennent :

**Propriété B.8.** *Tant que `r->destroyer` est à 1, il n'y a plus de thread qui utilise la ressource*

*Démonstration.* Ceci est assuré par le deuxième test à `r->destroyer` au début de `use`. □

**Propriété B.9.** *Si `r->users` est vide, il n'y a plus de thread en train d'utiliser la ressource.*

*Démonstration.* Pour utiliser la ressource, il faut « s'être inscrit » dans `r->users`. Notons cependant que contrairement à précedemment, `r->users` peut être non-vide sans qu'aucun thread n'utilise la ressource. □

Une nouvelle propriété importante est :

**Propriété B.10.** *Si `r->destroyer` est à 1, au bout d'un certain temps `r->users` sera vide.*

*Démonstration.* Après un certain temps, tous les utilisateurs auront terminé leur exécution, et aucun ne peut s'inscrire dans `r->users` grâce au premier test à `r->destroyer` au début de `use`.

Sans ce premier test, la correction serait assurée, mais il pourrait en théorie y avoir continuellement de nouveaux threads qui s'inscrivent dans `r->users` avant de s'en retirer ; et le nettoyage pourrait être bloqué pour un temps infini. □

Les preuves des autres propriétés sont facilement adaptées pour prendre en compte ces nouvelles propriétés.

Notons qu'on peut maintenant envoyer des signaux à des threads qui se sont déjà vu refuser l'accès à la ressource. Ceci est traité facilement par la routine de reception des signaux.

#### B.1.3.2 Implémentations pratiques

Différentes implémentations pratiques peuvent être dérivées de cet algorithme, suivant la manière dont on implémente `add_to_set`, `remove_from_set` et `empty`. Ces fonctions sont exécutées en parallèles, et doivent donc être prêtes à supporter ce parallélisme.

L'implémentation avec un bitfield est très simple à partir des éléments déjà montré, et ne mérite pas qu'on s'y attarde. L'implémentation avec un compteur du nombre d'éléments de l'ensemble `r->users` est également très simple.

**Utilisation d'un tableau par processeur** Une autre implémentation consiste à avoir, pour chaque processeur, une liste des ressources qu'il utilise. `add_to_set` et `remove_from_set` sont juste des insertions et suppressions dans cette liste ; `empty` demande un parcours de la liste pour chaque processeur.

Une propriété intéressante de cette implémentation est que les listes manipulées par `add_to_set` et `remove_from_set` ne sont écrites que par un processeur, ce qui simplifie leur implémentation parallèle (pas besoin d'utiliser de `CAS`, par exemple). Par contre, elles peuvent être lues par les autres processeurs (par la fonction `empty`).

On peut utiliser des listes différentes par type de ressource, ce qui permet un parcours plus rapide par la fonction `empty`. Un cas particulier usuel est lorsque les processeurs ne peuvent modifier au plus qu'une ressource d'un même type, auquel cas la liste est une simple case mémoire. Nous présentons l'implémentation dans ce cas simple en Figure B.1.3.2.

```
error_t use( resource_t r)
{
  if( r->destroyer) return EACCESS ;
    used_resource[my_cpu()] = r ;
  if( r->destroyer) {
    used_resource[my_cpu()] = 0 ;
    return EACCESS ; }
  ... //utilise
  used_resource[my_cpu()] = 0 ;
}
```

```
error_t destroy( resource_t r)
 {
   do {
    uint old = r->lock ;
    if (old.destroyer)
      return ECONCURRENCY ;
   uint new = old ; new.destroyer = 1 ;
  }while( !CAS(&r->lock, old, new)) ;

   for( uint i = 0 ; i < M ; i++)
     if( used_resource[i] == r)
        send_IPI( i) ;
   for( uint i = 0 ; i < M ; i++)
     while( used_resource[i] == r) ;

   ... // nettoie
   r->destroyer = 0 ; }
```

FIG. B.7 – Implémentation du use/destroy pattern avec tableau

**Implémentation pour les services en espace utilisateur** L'implémentation pour les services en espace utilisateur pose des problèmes supplémentaires, en particulier le fait que les threads peuvent arrêter leur exécution à n'importe quel moment (on ne peut pas masquer les interruptions). On ne peut donc pas compter sur les utilisateurs pour écrire le fait qu'ils n'utilisent plus la ressource.

On dispose par contre d'un autre mécanisme : à chaque fois qu'un thread accède à une ressource (après un appel de service ou un retour après préemption dans le service), le thread vérifie qu'il a toujours le droit d'accéder à la ressource (en comparant le timestamp de la ressource à celui de sa capacité). Ceci permet d'éviter les bugs TOCTTOU en monoprocesseur.

```
error_t use( resource_t r)
{
  used_resource[my_cpu()] = r ;
  if( cap.timestamp < r.timestamp)
    return EACCESS ;
  ... //utilise
}
```

```
error_t destroy( resource_t r)
 {
   r.timestamp.up = 0xffffffff ;
   uint mask = 0 ;
   for( uint i = 0 ; i < M ; i++)
     if( used_resource[i] == r)
        { mask |= (2 << i) ; }
   domain_send_ipi( cur_dom, mask) ;
   ... // nettoie

   // (Plus tard : réutilisation)
   domain.create_cap( cur_dom, r, ...) ;
}
```

FIG. B.8 – Implémentation du use/destroy pattern en espace utilisateur

L'implémentation est présentée Figure B.1.3.2. L'idée est la suivante : lorsqu'on

veut détruire une ressource, on révoque son accès en modifiant son timestamp[2]. Puis on force tous les threads présents dans le domaine à revérifier que leur capacité est toujours valide (e.g. en simulant une préemption/retour d'exécution dans le service), en leur envoyant une IPI particulière (présentée section 4.2.2).

Cette IPI n'affecte que les processeurs en train d'être exécutés dans le domaine courant. Ainsi un domaine ne peut pas perturber l'exécution des autres domaines, et si on envoie une IPI à un processeur qui s'est déjà fait préempter, elle est tout simplement ignorée.

Pour rendre le système plus efficace, et ne pas perturber l'exécution des autres threads, on n'envoie des IPIs qu'aux threads susceptibles d'utiliser la ressource. C'est la raison pour laquelle on utilise le tableau `used_resource`.

La propriété importante à assurer est que lorsque le nettoyage est amorcé, il n'y a plus de thread en train d'utiliser la ressource. Plusieurs faits sont nécessaires pour assurer cette propriété :

- La modification du timestamp de la ressource se fait avant l'envoi d'IPI, ce qui permet de bloquer l'arrivée des nouveaux utilisateurs et assure que si un thread utilise la ressource, la destruction n'a pas encore commencé.

- `used_ressource` est modifié avant de vérifier si l'accès à une ressource est autorisé ; ainsi si la vérification d'accès réussi, il n'y a pas de destruction de commencée, et le thread est certain de recevoir l'IPI si il y a destruction.

- `domain_send_ipi`, implémenté dans le noyau, assure que tous les processeurs ont bien reçu leur IPI avant de retourner. Ainsi, quand le nettoyage est entamé, tous les utilisateurs de la ressource se sont arrêtés.

Notons que cette implémentation ne sérialise pas les demandes de destructions d'une ressource. Cela est acceptable si toutes les créations et destructions pour une ressource fait par un seul thread (i.e. par le service de politique). Dans le cas contraire, on peut facilement utiliser un revocable lock (§ 4.3.4). Dans ce cas, on peut utiliser la valeur `0xfffffffe` de `r.timestamp.up` pour indiquer la présence du lock, de manière similaire à ce qui a été fait pour les capacités (§ A.1.3.1). Une implémentation est donnée Figure B.1.3.2.

### B.1.4 Conclusion

Nous avons présenté différentes implémentations du use/destroy lock, adaptées à différentes situations rencontrées dans le noyau. On remarque qu'avec assez de travail sur les algorithmes, on arrive toujours à trouver une manière de contourner les difficultés et d'aboutir à un algorithme efficace.

[2]atomiquement, comme vu en section A.1.3.2

```
error_t destroy( resource_t r) {
 //met 0xfffffffe dans r.timestamp.up
 //et recommence si révoqué
 revocable_lock( r.timestamp.up,
                 RETRY) ;
 uint mask = 0 ;
 for( uint i = 0 ; i < M ; i++)
   if( used_resource[i] == r)
      { mask |= (2 << i) ; }
   domain_send_ipi( cur_dom,
                    mask) ;
 ... // nettoie (doit être révocable)
 memset( r, 0, sizeof(r)) ;
 //met 0xffffffff dans r.timestamp.up
 revocable_unlock( r.timestamp.up) ;}
```

```
error_t create( resource_t r) {
 //met 0xfffffffe dans r.timestamp.up
 //et recommence si révoqué
 //et vérifie atomiquement que
 //r.timestamp.up est à 0xffffffff
 revocable_lock( r.timestamp.up, RETRY) ;
 ... // initialise l'objet
 // crée la capacité et libère le lock
 domain.create_cap( cur_dom, r, ...) ; }
```

FIG. B.9 – Sérialisation des destructions avec un revocable lock.

## B.2 Implémentation des rollforward et recoverable locks

L'implémentation du rollforward lock a été longuement décrite en section 4.3. Nous nous contentons de placer le code complet du rollforward et du recoverable lock ici. Ce code présente en particulier l'ordre dans lequel l'UTCB doit être modifié.

Les parties du code qui sont en rouges sont celles spécifiques au rollforward lock ; celle en bleu est spécifique au recoverable lock. Nous présentons en page 253 les deux implémentations simultanément car elles sont très proches.

Ce code repose sur des hypothèses sur quelques constantes utilisées :

- `UNUSED_NUMBER` n'est jamais une valeur utilisée par le lock. (-1 dans l'implem)
- `PREEMPTED` est différent de tous les numéros de CPU
- `FREE & thread_id_mask` est différent de tous les numeros de threads
- `FREE & cpu_mask == PREEMPTED`
- Tous les `my_thread_id` sont différents par thread, et les derniers bits sont libres pour y mettre un numéro de CPU.

**Notes**

- Pour le rollforward lock, il faut mettre toute la zone de `retry` à `end` et tout `upcall_fun` dans la nopreempt zone.
- On n'a pas besoin, après le `next`, de libérer le lock pour le reprendre : on peut le prendre directement en changeant sa valeur. Nous ne l'avons pas présenté pour éviter de rajouter de la complexité à ce code déjà complexe.
- Certaines optimisations sont possibles en prenant pour `PREEMPTED` la valeur 0.

- En monoprocesseur, de nombreuses optimisations sont possibles. En particulier, on n'a pas besoin de faire de `CAS`, car on est seul à s'exécuter ; et seuls certains cas sont possibles (e.g. soit le lock est pris, soit on réussit a continuer son exécution pour le rollforward).
- Malgré la taille de ce code, chaque chemin est individuellement rapide ; et en particulier, la prise d'un lock libre est très rapide. Dans les tests que nous avons réalisés, la prise/libération de verrou ne causait qu'un overhead de 50 cycles en multiprocesseur, 35 en monoprocesseur (du à l'utilisation du CAS). De plus, il n'y a de contention que pour le lock, les autres écritures étant toutes privées par processeur. Cette contention peut être améliorée en ajoutant une comparaison avant de faire le `CAS` (optimisation du « test and test and set »)

Ce code est relativement complexe ; nous avons tenté de le prouver en monoprocesseur, ce qui demande essentiellement la réalisation d'un automate pour chaque instruction assembleur, en regardant pour chaque état ce qu'il se passe en cas de préemption. Le diagramme résultant est une machine à état très grande (qui ne tient pas dans ces pages de manière lisible). Comme ce code est facilement sujet à erreur, il serait intéressant (et je pense possible) de réaliser un prouveur ad-hoc de ce code (écrit en assembleur), qui créerait cet automate automatiquement ; et qui permettrait en particulier de démontrer la vivacité du code (i.e. qu'on peut toujours sortir de la section critique). La création automatique de l'automate aiderait notamment à la mise au point du code.

```
// Sauve les registres.
// (l'exemple ici ne sauve que esp)
resume.esp = esp ;
upcall_entry = upcall_fun ;

retry :
// Ne change pas le lock si preemption
expected_value = UNUSED_NUMBER ;
addr_lock = lock_addr ;
test_success = 0 ;
addr_dump = dump ;

/* Essaye de prendre le lock. */
new_value = my_thread_id | PREEMPTED ;
expected_value = my_thread_id | my_cpu ;
if( CAS( lock_addr, FREE, new_value))
  {
    /* Le lock est pris. */
    test_success = 1 ;
    esp = stack ;
    // Le retour de la section critique
    // se fait à end_unlock
    *esp-- = &end_unlock ;
    goto section_critique ; }

/* Essaye de revoquer un lock pris. */
local = *lock_addr ;
other_thread_id = local & thread_id_mask ;
new_value = other_thread_id | PREEMPTED ;
expected_value = other_thread_id | my_cpu ;
if( CAS( lock_addr, FREE, new_value))
  {
    /* Le lock est pris. */
    recover_fun() ;
   // Rollforward : dès le CAS, on
   // doit être dans la nopreempt zone.
   stack.ret_address = next
   reload_regs(dump)
next :
  *lock_addr = FREE ; /* Libere le lock */
   goto retry ; }
// Le lock est occupé ; réessaie.
goto retry ;
```

```
end_unlock :
*lock_addr = FREE ;
// La section critique est exécutée ; sort.
end :
resume.eip = end ;
upcall_entry = resume ;
// Restaure les registres.
// Il y a au minimum esp.
esp = resume.esp ;

addr_lock = &always0 ;
addr_dump = upcall_resume ;
new_value = 0 ;
expected_value = 0 ;
return ;

upcall_fun :
// Si le thread n'a pas acquis le lock
if( test_success  != 1
    || (expected_value & thread_id_mask)
        != my_thread_id)
goto retry ;
// Si le thread a été révoqué
while(1) {
  uint local = *lock_addr ;
  uint local_id = local & thread_id_mask ;
  // Si la section critique est terminée
   if( local_id  != my_thread_id) goto end ;
  /* if( local_id  != my_thread_id) strong_revokee (); */

  // Si elle n'est pas terminée, et
  // qu'aucun thread ne la recouvre
  if( (local & cpu_mask == PREEMPTED))
    {
     new_value = my_thread_id | PREEMPTED ;
     expected_value = my_thread_id | my_cpu ;
     if( CAS( lock_addr, FREE, new_value))
       {
        revoked_fun() ;
        stack.ret_address = end_unlock
        reload_regs(dump)
       }
     }
  // Un autre thread est en train de
  // recouvrer ; spinne en attendant.
}
```

Annexe C

# Implémentation et preuve du service de mémoire virtuelle

Le service mémoire est au cœur de l'implémentation du micronoyau Anaxagoros. Il est utilisé par les applications pour allouer de la mémoire dynamiquement, modifier ou créér des espaces d'adressages, créer ou détruire des objets systèmes comme les domaines ou les threads, etc.

Il est l'exemple le plus important de services conçu selon les principes énoncés dans cette thèse : interface RISC, exportation des données, pas d'abstraction, exportation de la gestion de la consistance, pas d'allocation dynamique, etc. Il dispose néanmoins de la particularité d'être exécuté dans le noyau, et son implémentation profite à ce titre de l'opération privilégiée de masquage des interruptions.

Les idées dans l'implémentation de ce service pourrait servir à d'autres systèmes, que ce soient d'autres micronoyaux, des exonoyaux, ou des hyperviseurs.

**Architecture cible** L'implémentation a été réalisée sur architecture IA32, où les pages, et les tables de pages des deux niveaux font 4ko[1]. Le format des tables de pages est fixé, et c'est le matériel va chercher les entrées dans la table de pages pour les mettre dans le TLB. On peut réserver une partie de l'espace d'adressage au mode superviseur ; c'est à cet endroit qu'on place les mappings du noyau, dont on ne parle pas dans ce chapitre (il suffit de considérer que les répertoires des pages ont des entrées indisponibles). Enfin, le TLB de l'IA32 est non-tagué, signifiant qu'il est vidé à chaque changement d'espace d'adressage (e.g. appel de service, préemption, etc.)

L'implémentation actuelle se base seulement sur l'hypothèse où les pages et les tables des pages font la même page, ce qui est raisonnablement portable. Cette hypothèse pourrait être levée si nécessaire.

[1] des extensions permettent des tailles de page différentes.

# C.1 Modèle et implémentation monoprocesseur

## C.1.1 Diagramme des états d'une page physique

L'implémentation, sur processeur Intel IA32, est centrée sur une idée simple : tous les objets systèmes sont de taille une page (4096 octets sur IA32). Cette idée part de l'observation que les objets « matériels » (i.e. pages de données et tables de pages des deux niveaux) sur ce processeur sont de taille une page. Il suffit de définir les objets « logiciels » (threads, processus) de taille une page pour que tous les objets systèmes soient de cette taille.

Ainsi, l'implémentation consiste à réaliser les transitions entre les utilisations d'une page physique (en anglais frame). Le diagramme des transitions possibles, que nous allons présenter, est donné Figure C.1.

### C.1.1.1 Définitions et contexte

Chaque état de ce diagramme représente un *type* de la page. Le passage d'un type à un autre est appellé *transformation* de la page. La présentation fonctionelle de ce diagramme présente les différents types et transformations par boucle, ainsi que le pseudo-code correspondant. Une transformation vers l'état zéro est appellée *nettoyage* ; une transformation depuis l'état zéro est appellée *création*.

Notons que d'autres systèmes, comme Xen [BDF+03] ou seL4 [EDE08] ont également une organisation par typage de la mémoire physique.

Une tâche peut être supprimée à tout moment, y compris si elle était en train d'effectuer un traitement dans le noyau. Lorsqu'une tâche est supprimée, toutes les données associées à une tâche sont perdues, y compris celles concernant son éventuelle exécution dans le noyau. Pour autant, il faut que l'état du système de mémoire virtuelle reste dans un état cohérent.

C'est la raison pour laquelle nous avons choisi pour le noyau une API « atomique » et une structuration selon l'« interrupt model » [FHL+99] (i.e. une pile par processeur). Par ce choix, le noyau ne dépend pas pour son exécution de mémoire qui peut lui être révoquée, ce qui permet de simplifier la suppression d'une tâche et de toujours laisser le noyau dans un état cohérent.

En résumé, le noyau doit transformer les pages d'un type à l'autre de manière *atomique*, en laissant toujours le système de mémoire virtuelle dans un état cohérent.

Nous allons maintenant décrire les différentes boucles de transformations.

### C.1.1.2 Boucle `Zéro` → `Dataframe` → `Zéro`

**Type `Dataframe`** Ce type représente une page qui peut contenir des données. Ces pages sont les seules[2] qui puissent être écrites librement par l'espace utilisateur : i.e. pour qu'une table des pages pointe vers une page physique avec les permissions en écriture, il faut que cette page physique soit de type `Dataframe`.

[2] avec l'UTCB, voir plus loin

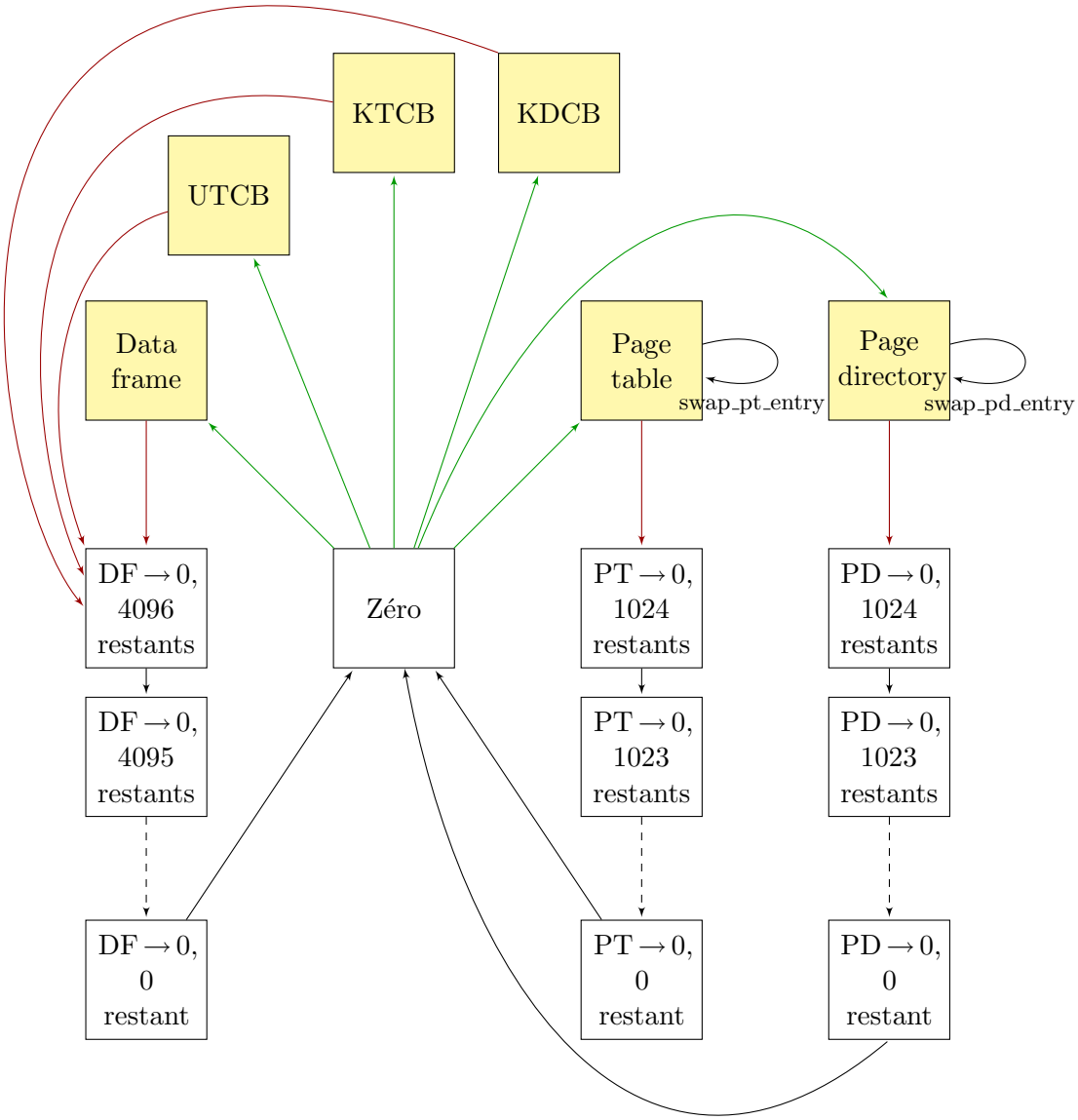



FIG. C.1 – États possibles d'une page physique, et transitions atomiques entre ces états. Une transition verte indique une création d'objet, et une transition rouge une destruction ; un état jaune indique que la page contient un objet système.

**Type `Zéro`** Le type `Zéro` représente une page physique remplie par la valeur 0, et qui est prête à être utilisée.

Le type `Zéro` n'est pas simplement une page `Dataframe` qui serait remplie par des zéros : il y a la condition supplémentaire qu'aucune table des pages (d'aucun niveau) ne pointe vers la page `Zéro`. Ainsi cette page n'est pas accessible par l'espace utilisateur.

**Transformation** `Dataframe` → `Zéro` Ainsi, pour qu'une page $p$ passe du type `Dataframe` au type `Zéro`, il faut que toutes les entrées de table de page qui pointent vers $p$ soient supprimées, et que $p$ ne contienne que des zéros. La suppression des entrées qui pointent vers $p$ peut être déléguée à l'espace utilisateur (il suffit de compter les mappings vers une page), mais pas la mise à zéro de $p$.

En effet, une fois tous les mappings rendus, comment le noyau peut il savoir que la page ne contient que des zéros ? Il faudrait pour cela que le noyau parcoure toute la page, pour vérifier que chaque mot contient 0. En ce cas, autant laisser au noyau le soin de remplir la page de zéros, ce qui élimine le besoin de vérification.

La transition du type `Dataframe` au type `Zéro` demande donc un certain temps ; or les transitions doivent être faites atomiquement dû à notre choix d'une API atomique. Cela signifie dans l'implémentation une longue section non-préemptible. Pour éviter cela, nous définissons des états intermédiaires[3] où le nettoyage n'est que partiel. Ainsi l'état `(DF→ 0, 2000 restants)` signifie que les 2096 premiers octets de la page ont été mis à zéro, et qu'il en reste 2000 pas encore mis à zéro.

Ces états sont visibles et cohérents. Concrètement, cela signifie qu'il est possible pour un thread de « terminer » le nettoyage entreprit par un autre, ce qui est fort utile dans le cas de suppression forcée d'un thread. Pour ne pas avoir à se soucier du nombre d'octets qui a été nettoyé, la méthode `zero_page(p)` du noyau reprend le nettoyage d'où il s'était arrêté, sans qu'on aie besoin de spécifier l'état actuel du nettoyage.

Quand une page $p$ est dans un de ces états intermédiaires de nettoyage, aucune table des pages ne doit pointer vers elle ; sinon le nettoyage pourrait être défait par une écriture concurrente. Ainsi, la vérification que toutes les entrées de table de page qui pointent vers $p$ sont supprimées se fait lors de la transition `Dataframe` vers `(DF → 0, 4096 restants)`.

Notons que la définition de 4096 états correspond seulement au modèle. Dans l'implémentation réelle, la mise à zéro de la page physique se fait au minimum mot par mot, mais plus généralement par tranche de plusieurs mots. Il y a donc beaucoup moins d'états intermédiaires réels : le diagramme montre le maximum possible d'états intermédiaires. Le nombre d'états définis est en fait un compromis entre la latence de prise d'interruption et le surcoût en temps CPU qu'engendre la définition d'états intermédiaires (voir §C.3.4.1).

#### C.1.1.3 Boucles `Zéro` → {KDCB, KTCB, UTCB} → `Zéro`

**Type** `KDCB`, `KTCB`, `UTCB` Ces types représentent les différents objets présentement gérés par le système.

- Le KDCB est l'objet système qui contient les informations relatives à un domaine de protection, comme le pointeur universel vers l'espace d'adressage courant, les capacités accessibles au domaine, et les ports IOs (i.e. « espace d'adressage IO ») auxquels le domaine peut accéder ;
- Le KTCB est l'objet système qui contient les informations relatives à un thread noyau, comme le pointeur universel vers le domaine courant, des

[3] cette idée est à l'origine du principe de définition des états intermédiaires consistants (§ 5.3.3)

```
(Dataframe) → (DF->0,4096 restants) :
   // Assure que la page est du bon type
   check( p.type == Dataframe) ;
   // Assure que la page n'est plus mappée en mémoire
   check( p.mappings.all == 0) // Implique p.mappings.rw == 0
   p.type = (DF->0,4096 restants) ;

(DF->0,n restants) → (Zero) :
   check( p.type & typemask == DF->0) ; // masque la composante n
   int n = p.type & nmask ; // récupère n du type
   do {
     p.contents[n] = 0 ; // Mise à zéro de l'octet
     if( pending_preemption) // Flag volatile mis à 1 si attente de préemption
       p.type = DF->0,n restants
   } while(n--)
   p.type = Zero :

(Zero) → (Dataframe) :
   check( p.type == Zero) ;
   p.type = Dataframe ;
```

Listing C.1 – Pseudo-code de la boucle `Dataframe`

informations sur l'ordonnancement, et les capacités du thread. Il contient également des entrées de table ou répertoire de pages, pour les thread-local mappings (§ 3.4.2.2) ;

- L'UTCB est l'objet système utilisé pour la communication asynchrone entre un thread et le noyau, et contient notamment les registres sauvegardés lors d'une préemption (§ 4.2.3). L'UTCB est également utilisé comme pile pour les services auquel il est prêté, et comme moyen pour passer des paramètres entre client et service.

La taille de ces différents objets est modulable, en modifiant le nombre de capacités ou la taille de la pile. Nous avons choisi que tous ces objets soient de taille une page, afin de pouvoir définir le comportement du système de mémoire virtuelle d'après le diagramme de la Figure C.1. Cela fournit dans l'implémentation actuelle environ 250 capacités par thread et par domaine[4].

**Transformations `Zéro` → {`KDCB`, `KTCB`, `UTCB`}** Ces transformations ne font qu'initialiser les champs des objets à leurs valeurs initiales. Le seul point remarquable est que pour de nombreux champs, on utilise la convention « mise à zéro implique valeur

[4]Ce nombre de 250 capacités peut sembler petit, mais il est en fait relativement gros si on utilise le principe du moindre privilège (§ 2.2.2.4). Pour le contourner, il faut créer plus de domaines, qui ne sont pas chers vu qu'on peut avoir plusieurs domaines pour un même espace d'adressage ; et stocker les capacités dans ces différents domaines.

initialisée », ainsi cette transformation est normalement assez courte. En particulier, pour les capacités, une valeur de 0 signifie que la capacité est invalide.

**Transformations {KDCB, KTCB, UTCB} → Zéro** Le nettoyage d'un objet système est fait de deux opérations

1. le changement de type de la page, afin que la page ne soit plus utilisable en tant qu'objet système ;
2. la mise à zéro du contenu de la page.

Comme pour les pages de données, la mise à zéro du contenu de la page est une opération assez longue, qu'on ne veut pas faire de manière atomique. Cela demande une conception par laquelle on passe par des états de nettoyage intermédiaire, comme pour les pages de données. Une première conception créerai de nouveaux états intermédiaires ({KTCB,UTCB,KDCB}→0,$n$ restants) pour faire ces nettoyages de manière fractionnée.

On peut cependant faire l'observation suivante : une page ayant contenu un objet système n'est finalement qu'une page de donnée ; on peut donc la nettoyer de la même manière qu'une page de données. Ainsi, on factorise le nettoyage des objets systèmes et des pages de données ; toutes les opérations de nettoyages éventuelles spécifiques à chaque objet sont fait lors de la transition (objet système) → (DF→0,4096 restants).

Notons que cela fonctionne bien lorsque cette transition est courte. En particulier, on profite pour le KTCB et le KDCB de la propriété selon laquelle une capacité peut être détruite sans « prévenir » l'objet vers laquelle elle pointe (e.g. on n'implémente pas de compteur du nombre de capacités vers un objet, contrairement aux ports de Mach [Loe92, p. 34], ou aux capacités dans seL4 [KEH+09, § 3.2]). Les tables et répertoires de pages ont besoin de faire un nettoyage pour chaque entrée, et ont donc une procédure de nettoyage qui leur est propre.

Notons que plusieurs champs du type KTCB pointent vers des pages ou des table des pages pour les thread-local mappings. Ces entrées sont gérées de manière similaire à celles des types tables et répertoires de pages, que nous voyons maintenant.

#### C.1.1.4 Boucles Zéro → Page{table,directory} → Zéro

**Types Pagetable et Pagedirectory** Deux types supplémentaires que le noyau doit gérer sont les table des pages des différents niveaux, appellés table des pages et répertoire des pages sur Pentium [Int09]. C'est par leur gestion que le noyau va assurer des propriétés de sûreté telles que « on ne peut pas écrire directement dans les objets systèmes », « on ne peut pas forger de capacités », etc.

Les différentes tables des pages doivent être naturellement protégées contre l'écriture directe par l'espace utilisateur, puisque cela permettrait par ce biais à l'espace utilisateur d'accéder à n'importe quelle page du système, et plus aucune propriété de sûreté ne pourrait être assurée. Par contre, on peut permettre l'accès à ces pages en lecture : cela est utile notamment lors de la paravirtualisation de systèmes (voir e.g. [BDF+03]).

```
(Zero) → (Pagetable) :
   check( p.type == Zero) ;
   p.type = Pagetable ;

void swap_pt_entry( page_t p, index_t n, entry_t e) {
   check( p.type == Pagetable) ;
   entry_t old = p.table[n] ; // recupère l'ancienne entrée
   if( e.present) { // si la nouvelle entrée est valide
     page_t to = e.pointed_frame ;
     // On ne mappe pas les frames qui sont dans un "état de nettoyage"
     check( to.type  != Zero && to.type  != (*->0,n restants)) ;
     to.mappings.all ++ ;
     if( e.writable) { // si l'entrée contient le champs writable
        check( to.type == Dataframe) ;
        to.mappings.rw ++ ; }}
    p.table[n] = entry ;
    page_t old_to = old.pointed_frame ;
    if( old.present) {
      old_to.mappings.all -- ;
      if( old.writable) { old_to.mappings.rw -- ; }}}

(Pagetable) → (PT->0,1024 restants) :
   check( p.type == Pagetable) ;
   p.type = (PT->0,1024 restants) ;

(PT->0,n restants) → (PT->0,0 restant) :
   check( p.type & typemask == PT->0) ; // masque la composante n
   int n = p.type & nmask ; // récupère n du type
   do {
     swap_pt_entry( p, n, 0) ;
     if( pending_preemption) // Flag volatile mis à 1 si attente de préemption
       p.type = PT->0,n restants
   } while( n--) ;

(PT->0,0 restant) → (Zero) :
   check( p.type == (PT->0,0 restant)) ;
   check( p.mappings.all == 0) ;
   p.type = Zero ;

// Les transitions sur les Pagedirectory sont les même avec Pagetable remplacé
// par Pagedirectory. Seule la fonction swap_pd_entry change significativement :
void swap_pd_entry( page_t p, index_t n, entry_t e) {
   check( p.type == Pagedirectory) ;
   entry_t old = p.table[n] ; // recupère l'ancienne entrée
   if( e.present) {
     page_t to = e.pointed_page ;
     check( to.type == Pagetable) ;
     to.mappings.all ++ ; }
   p.table[n] = entry ;
   page_t old_to = old.pointed_frame ;
   if( old.present) { old_to.mappings.all -- ; }}
```

Listing C.2 – Pseudo-code des boucles `Pagetable` et `Pagedirectory`

**Modification des entrées de la table des pages** Les opérations les plus intéressantes concernant les tables des pages sont les modifications des entrées dans les tables des pages. C'est à cette occasion que l'on va contrôler ce qui est mis dans les table des pages et ainsi, assurer des propriétés de sûreté pour la mémoire virtuelle. Ainsi dans le pseudo-code (Listing C.2), nous nous assurons ainsi que seules les pages de données peuvent être mappées en écriture (dans cet exemple, les autres pages peuvent toutes être mappées en lecture ; il est notamment utile de pouvoir accéder aux tables des pages pour la virtualisation).

C'est également à cet endroit que va être mis à jour le *compteur de mappings* des pages. Ce compteur de mappings compte pour chaque page le nombre de fois où elle est pointée par une table des pages. Pour les pages physiques utilisées comme table des pages, ce compteur compte le nombre de fois où la page est pointée par une table des pages de niveau supérieur (i.e. par un répertoire de pages). C'est grâce à ce compteur qu'on peut s'assurer que plus aucune table des pages ne pointe vers une page, et qu'on peut donc changer son type sans crainte. C'est à l'espace utilisateur de faire ensuite en sorte de respecter les contraintes mises dans ce code (e.g. de ne pas transformer une page de donnée tant qu'il existe des tables de pages qui pointent sur elle).

**Notes**

- La modification d'une entrée dans la table des pages ne modifie le type d'aucune page. Nous la considérons néanmoins comme une transition (et elle est représentée dans la Figure C.1), car elle fait partie intégrante du mécanisme de mémoire virtuelle et de sa preuve.
- Une alternative à ce compteur aurait été l'utilisation d'estampilles dans les tables de pages ; c'est à dire que les entrées de la table des pages seraient devenues des pointeurs universels (§ 3.3.3.1). Cela permet de changer le type d'une page mémoire $p$ même si il reste des tables de pages qui pointent vers $p$, et supprime le besoin de maintenir le compteur de mappings. En contrepartie, chaque entrée dans la table des pages prend beaucoup plus de place.

  Cette alternative peut s'implémenter lorsque le TLB est géré par logiciel, mais cela est plus difficile lorsque le format des tables est fixé comme sur le Pentium. Ceci dit, le fait d'interdire certaines transitions lorsque le compteur de mapping n'est pas nul n'est pas si contraignant pour l'espace utilisateur en général, et l'intérêt de cette alternative semble donc minime.
- La modification des entrées dans la table est également un bon endroit pour installer des mécanismes de programmation défensive. Par exemple, on peut rajouter des assertions pour vérifier que lorsqu'on supprime un mapping vers une page, le compteur de mappings de cette page est non-nul ; ou des assertions sur le type de cette page.
- Notons que le compteur de mapping ne compte pas le nombre de fois où la page est mappée ; mais le nombre de fois où elle est pointée par une table des pages de niveau supérieur. Ainsi, si une table des pages non-mappée contient deux entrées vers une page, le compteur de

mappings de la page sera de 2, et non de 0. De plus, les tables de pages peuvent être mappées en tant que donnée en lecture ; dans ce cas le compteur comptient la somme du nombre de fois où la page est mappée par une table des pages et du nombre de fois où elle est mappée par un répertoire des pages.

- Xen dispose d'un compteur de mappings similaire à celui présenté ici [BDF+03, § 3.3.3].

**Transformations** `Zéro` → `Page{table, directory}` **et** `Page{table, directory}` → `Zéro` Il est courant pour les architectures mémoire paginées, et c'est le cas pour l'IA32, qu'une entrée de table des pages à 0 signifie que l'entrée est vide. Ainsi, en changeant seulement le type d'une page de `Zéro` vers `Pagetable`, on obtient une table des pages vide et valide. Dans l'implémentation réelle, changer le type est tout ce qu'il y a a faire pour les tables des pages ; pour les répertoires de page, c'est à ce moment qu'on installe les mappings du noyau.

La transformation du type `Pagetable` vers `Zéro` est similaire au cas de la boucle de données, i.e. on expose des états intermédiaires consistants qui permettent de décomposer le nettoyage des tables de pages. Cependant, le nettoyage des tables et répertoires de pages nécessite des précautions spéciales, qui empêchent de réutiliser les états (`DF→0,`$n$ `restants`) comme pour les autres objets systèmes. Ce mécanisme est spécial pour deux raisons.

- Le nettoyage de chaque entrée d'une table ou d'un répertoire nécessite de décrémenter les compteurs de mappings des pages pointées correspondantes ; on ne peut donc pas simplement mettre l'entrée à zéro comme on le fait pour les autres objets systèmes.

  Notons que l'opération de nettoyage d'une entrée est équivalente à l'appel d'une fonction `swap_pt_entry` ou `swap_pd_entry` avec 0 pour troisième argument ; l'implémentation réelle utilise cependant une version optimisée[5].

- Il est préférable que la vérification que le nombre de mappings est nul se fasse après le nettoyage des entrées, et non avant comme pour les pages de données.

  Cela est utile lors du nettoyage d'un ensemble de pages séparé[6] ; faire la vérification à la fin permet de faire le nettoyage en seulement deux passes. La première nettoye toutes les entrées de toutes les tables de pages et répertoires, mettant ainsi tous les compteurs de mappings à zéro ; la deuxième nettoye toutes les pages de données et les objets systèmes, et fait passer les tables de pages et de répertoire vers le type `Zéro`.

  Si la vérification était faite au début, il faudrait trois passes : d'abord nettoyer les répertoires de pages, ce qui permet de nettoyer les tables de pages, et enfin les pages. Il faudrait rajouter d'autres passes si la pagination était sur plus de

[5] qui est en fait le résultat de l'évaluation partielle de ces fonctions avec 0 en dernier argument.

[6] dans le sens où il n'y a pas de tables de pages ou répertoires de pages externe à cet ensemble qui pointe vers cet ensemble

deux niveaux, et il y aurait également des problèmes dans le cas où des tables de pages se référencent mutuellement.

Notons que faire cette vérification à la fin du nettoyage est possible parce que toutes les écritures sur les tables de pages sont faites par le service mémoire, et qu'il garantit qu'il n'y a plus d'écriture une fois le nettoyage commencé (on ne peut appeller `swap_p{t,d}_entry` que sur une page de type `Page{table,directory}`.

On définit pour ces raisons de nouveaux types de nettoyage, `(PT→0,` $n$ `restants)` et `(PD→0,` $n$ `restants)`. Ces types peuvent être utilisés en tant que table des pages et répertoire des pages, mais leurs entrées ne peuvent pas être modifiées autrement que par nettoyage.

#### C.1.1.5 Propriétés assurées et schéma de preuve

L'implémentation que nous avons décrite assure un certain nombre de propriétés, énoncées ci-après. La plus importante est que seules les pages de type `Dataframe` sont accessibles en écritures[7] : ainsi seul le noyau écrit dans les autres types de données.

Les propriétés sont prouvées sur un modèle du système qui est le diagramme de la Figure C.1, auxquelles sont rajoutées deux transitions symbolisant les lectures et écritures sur une page. Les lectures et écritures respectent les informations contenues dans les tables et répertoires de pages, mais ne vérifient pas le type de la page.

On considère l'évolution des états du système, où un changement d'état se fait par une transition exécutée de manière atomique, et partant de l'état où toutes les pages sont remplies de zéro et de type `Zéro`. Les conditions des `check`s sont des gardes sur les transitions, i.e. les transitions ne peuvent se faire que si ces conditions sont remplies.

La preuve générale des propriétés consiste à montrer que si on part d'un état où toutes ces propriétés sont assurées, alors quelque soit la transition prise, les propriétés sont encore assurées. Ainsi, les propriétés sont continuellement respectées. Les schémas de preuves que nous donnons expliquent comment chaque propriété est assurée pendant les transitions.

**Propriétés sur les compteurs de mappings** Ces propriétés démontrent que les valeurs des compteurs de mappings correspondent bien aux nombres de mappings dans le système.

**Propriété C.1.** *Pour toute page* `p` *dont le type est* `Zéro`*, son compteur de mappings* `p.mappings.all` *est égal à zéro.*

*Démonstration.* `swap_pt_entry` et `swap_pd_entry` vérifient qu'on ne peut pas installer de mappings dans un « type de nettoyage ». Ces fonctions sont les seules à manipuler le compteur, donc une fois dans un type de nettoyage, le compteur de mappings ne peut plus s'incrémenter.

[7] Pour cette preuve, on met de coté le fait que l'UTCB soit accessible en écriture, car son mapping est géré de manière spéciale par le noyau.

De plus, à un certain endroit de chaque boucle de nettoyage, il y a une vérification que le nombre de mappings est égal à zéro :

- à la fin pour les tables et répertoires de pages ;
- au début pour les autres objets.

Ainsi dans le type `Zéro`, les pages ont un nombre de mappings `p.mappings.all` à zéro, et qui ne peut pas s'incrémenter.

Notons que cela implique que `p.mappings.rw` soit égal à zéro (prop. C.6). □

**Propriété C.2.** *Pour toute page `p`, les transitions de création de table de pages ou de répertoire `(Zéro)→(Pagetable)` et `(Zéro)→(Pagedirectory)` laissent invariant le nombre de mappings vers `p`.*

*Démonstration.* Une page de type `Zéro` ne contient que des zéros (prop. C.10), et ces transitions ne modifient pas leur contenu. Cette transition crée donc une table ou répertoire de page qui ne pointe vers aucune page, laissant invariant le nombre de mappings vers toute page.

Cette propriété s'appuye sur le fait qu'une entrée mise à zéro ne pointe vers aucune page. □

**Propriété C.3.** *Pour toute page `p` dont le type est différent de `Pagetable` et de `(PT→0,`$n$ `restants)`, `p.mappings.all` est égal au nombre de fois où `p` est présente dans une table des pages.*

*Démonstration.* La transition qui manipule les mappings de table de page est `swap_pt_entry`, car toutes les tables de page sont de type `Pagetable` ou `(PT→0,`$n$ `restants)` (prop. C.8) et les pages de ces types ne sont pas accessibles par écriture directe (prop. C.9).

À chaque mapping installé, la page correspondante a son compteur incrémenté, à chaque mapping désinstallé le compteur est décrémenté. Donc après ces transitions, le compteur et le nombre de mappings restent égaux.

Cette égalité n'est pas respectée lorsque la page est de type `Pagetable` ou (`PT→0,`$n$ `restants`), mais pour revenir vers un autre type les pages doivent passer par le type `Zéro`, où le compteur et le nombre de mappings sont tous deux égaux (à zéro, prop. C.1). Ainsi, la propriété reste vraie avec les transitions de changement de type de `p`.

Enfin, la création de nouvelles tables de pages laissent cette égalité invariante (prop. C.2).

Les autres transitions ne modifient ni le compteur de mappings ni le nombre de mappings de tables de page. Donc pour toute transition, la propriété reste vraie. □

**Propriété C.4.** *Pour les pages `p` dont le type est `Pagetable` ou `(PT→0,`$n$ `restants)`, `p.mappings.all` est égal au nombre de fois où elle est présente dans une table des pages plus le nombre de fois ou elle est présente dans un répertoire de pages.*

*Démonstration.* Les transitions qui manipulent les mappings de table de page (resp. de répertoire de page) sont `swap_pt_entry` (resp. `swap_pd_entry`), en utilisant les propriétés C.8,C.9 et C.7 comme pour la preuve de la propriété précédente.

À chaque mapping installé (que cela soit dans une table des pages ou répertoire des pages), la page correspondante a son compteur incrémenté, à chaque mapping désinstallé le compteur est décrémenté. Donc après ces transition, le compteur et le nombre de mappings restent égaux.

L'égalité est respectée lors des changement de types : la transition de `Pagetable` vers `(PT→0,`$n$ `restants)` ne modifie pas le compteur de mappings, et la création de nouvelles tables de pages passe par le type `Zéro`, où le compteur et le nombre de mappings sont tous deux égaux (à zéro, prop. C.1).

Enfin, la création de nouvelles tables ou répertoires de pages laissent cette égalité invariante (prop. C.2).

Les autres transitions ne modifient ni le compteur de mappings ni le nombre de mappings. Donc pour toute transition, la propriété reste vraie. □

**Propriété C.5.** *Pour toute page* `p`*,* `p.mappings.rw` *est égal au nombre de fois où* `p` *est présente dans une table des pages avec accès en écriture.*

*Démonstration.* Les transitions qui manipulent les mappings de table de page (resp. de répertoire de page) sont `swap_pt_entry` (resp. `swap_pd_entry`), en utilisant les propriétés C.8 et C.9 comme pour la preuve des propriétés précédente.

À chaque mapping en écriture installé, la page correspondante a son compteur incrémenté, à chaque mapping en écriture désinstallé le compteur est décrémenté. Donc après toute transition, le compteur et le nombre de mappings en écriture restent égaux.

Enfin, la création de nouvelles tables ou répertoires de pages laissent cette égalité invariante (prop. C.2).

Les autres transitions ne modifient ni le compteur de mappings en écriture ni le nombre de mappings en écriture. Donc pour toute transition, la propriété reste vraie. □

**Propriété C.6.** *Pour toute page* `p`*, si* `p.mappings.all` *vaut zéro, la page n'est présente dans aucune table des pages ou répertoires des pages.*

*De plus,* `p.mappings.rw` *est égal à zéro.*

*Démonstration.* Soit `p` est de type `Pagetable` ou `(PT→0,`$n$ `restants)`, auquel cas un nombre de mappings de 0 signifie qu'il n'est dans aucune table de pages ni aucun répertoire de pages (prop. C.4).

Soit `p` n'est pas de ces deux types, auquel cas un nombre de mappings de 0 signifie qu'il n'est dans aucune table de pages (prop. C.3). De plus, comme seules les pages de type `Pagetable` ou `(PT→0,`$n$ `restants)` peuvent être utilisées comme table de pages (prop. C.8), elle n'est également présente dans aucun répertoire des pages.

Enfin, comme `p` n'est dans aucune table des pages, elle n'est dans aucune table des pages avec accès en écriture, donc `p.mappings.rw` vaut zéro (prop. C.5). □

**Propriétés sur les types de pages et leur utilisation** Ces propriétés démontrent que les types des pages correspondent bien aux utilisations qui en sont faites.

**Propriété C.7.** *Pour être utilisée en tant que répertoire de pages, une page doit être de type* `Pagedirectory`*.*

*Démonstration.* Cette propriété dépend de code que nous n'avons pas présenté. Pour implémenter cette propriété, il suffit qu'avant de changer d'espace d'adressage, le code de chargement vérifie que le type de la page est bien `Pagedirectory`.

Il faut également assurer que si une page de type `Pagedirectory` est nettoyée, elle ne soit plus utilisée. La section C.1.2.3 sur l'autorévocation décrit comment cela est réalisé.

Note : le type `(PD→0,`$n$ `restants)` ne peut pas être utilisé comme répertoire des pages ; cela empêche que des mappings du noyau soient retiré pendant l'exécution du noyau. □

**Propriété C.8.** *Pour être utilisée en tant que table de pages, une page doit être de type* `Pagetable` *ou* `(PT→0,`$n$ `restants)`*.*

*Démonstration.* Pour être utilisé comme table de pages, une page doit être présente dans un répertoire de pages. Or les répertoire de pages sont de type `Pagedirectory` (prop. C.7) qui ne sont pas accessibles en écriture (prop. C.9). Ainsi, la seule transition qui installe un mapping dans un répertoire de page est `swap_pd_entry`, qui vérifie que la page est de type `Pagetable` avant d'être installée.

La page peut alors changer de type, mais uniquement vers `(PT→0,4096 restants)`. Pour cela, il faut que son compteur de mapping soit égal à zéro, ce qui signifie (prop. C.4) que plus aucun répertoire de page ne pointe vers elle. □

**Propriété C.9.** *Seules les pages de type* `Dataframe` *sont accessibles en écritures*[8]*.*

*Démonstration.* Pour être accessible en écriture, il faut être présent dans une page utilisée comme table de pages. Or les tables de pages sont de type `Pagetable` ou `(PT→0,`$n$ `restants)` (prop. C.8), qui ne sont pas accessibles en écriture (prop. C.9). Ainsi, la seule transition qui installe un mapping en écriture dans une table de page est `swap_pt_entry`, qui vérifie que la page est de type `Dataframe` avant d'être installée.

Une fois installée, la page peut changer de type, mais cela nécessite de passer par la transition `(Dataframe)` → `(DF->0,4096 restants)`, qui vérifie que le compteur de mappings est à zéro, et donc que la page n'est plus accessible (prop. C.6), donc n'est plus accessible en écriture. □

**Propriété C.10.** *Lorsqu'une page est de type* `Zéro`*, elle ne contient que des zéros.*

*Démonstration.* Lors qu'une page passe dans un état de nettoyage, elle n'est plus accessible en écriture, car seules les pages de type `Dataframe` sont accessibles en écriture (prop .C.9), et ne le sont plus lorsqu'elles changent de type. Ainsi toutes les écritures dans les états de nettoyage (dont `Zéro`) sont faites par le noyau, et seulement par le code des transitions concernées.

[8] Dans la réalité, les UTCB sont également accessibles en écriture ; mais leurs mappings sont gérés de manière spéciale par le noyau.

Le fait que chaque nettoyage est fait correctement se montre par simple récurrence : si une page est de type `(*->0,`$n$ `restants)`, alors seules les $n$ dernières entrées ne sont pas nettoyées. Ainsi, lors de la transition vers le type `Zéro`, la page est intégralement nettoyée. □

**Propriétés de vivacité** Ces propriétés démontrent que la mémoire peut toujours être récupérée, et qu'il n'y a pas d'état du système où des opérations peuvent être bloquées.

**Propriété C.11.** *On peut passer depuis n'importe quel état du système vers l'état « tout à zéro ».*

*Démonstration.* Ce nettoyage des pages du système peut se faire en deux passes. Lors de la première, tous les répertoires et tables de pages passent dans les types respectifs `(PD→0,0 restants)` et `(PT→0,0 restants)`. Il n'y a rien qui empêche cette opération ; après elle il n'y a plus aucun mapping installé dans le système.

Dans le deuxième temps, toutes les pages sont passées vers le type `Zéro` (on le peut puisque les compteurs de mappings sont tous égaux à zéro). Et lorsqu'une page est à `Zéro`, elle ne contient que des zéros (prop. C.10).

Notons que cette technique est utilisée pour nettoyer un sous-ensemble « isolé » (dans le sens où il n'y a pas de mappings extérieurs qui pointent vers l'ensemble) de pages. □

**Propriété C.12.** *On peut passer de n'importe quel état vers n'importe quel autre.*

*Démonstration.* Par définition de notre système, tout état de notre système est issu d'une succession de transitions depuis l'état « tout à zéro ».

Pour passer de $e_1$ à $e_2$, il suffit de passer de $e_1$ à « tout à zéro » (prop. C.11), puis de répéter les transitions qui permettent d'arriver à $e_2$. □

**Remarques sur la preuve**

**Notes**

- Pour terminer la preuve, il reste seulement montrer que toutes les propriétés sont vérifiées à l'état initial « tout à zéro ». Cela est très simple.
- Il faudrait idéalement compléter la preuve pour prendre en compte la gestion de la consistance du TLB. Cela passe par l'ajout dans le modèle d'un TLB de taille infinie, qui n'oublie les anciennes entrées que sur invalidation explicite. Cela assure que toutes les invalidations sont faites au bon moment. Nous expliquons comment est gérée la consistance du TLB en section C.3.3.
- Il est facile d'oublier des conditions dans cette preuve, qui feraient en sorte que le système ne marche pas. Une preuve formelle vérifée par un programme serait intéressante.
- On remarque que `mappings.rw` est inutile pour le fonctionnement de l'algorithme, mais c'est une information utile en pratique (e.g. si la confidentialité n'est pas importante, on peut permettre des changements de types dès que `mappings.rw` vaut 0).

### C.1.2 Considérations importantes non présentées

Il y a des considérations complémentaires, que l'on ne présente que maintenant afin de faciliter la compréhension du système de mémoire virtuelle.

#### C.1.2.1 Autres types

Notre mécanisme est facilement extensible à d'autres types d'objets, pourvus qu'ils soient de taille une page[9]. Xen a par exemple des types supplémentaires pour la gestion des pages de segment [BDF+03, § 3.3.3] dont nous pourrions avoir besoin pour aider à la virtualisation.

Le TMO (§ 3.4.4) est également un autre type que nous n'avons pas présenté, qui se nettoye comme un table de pages (et en partage la boucle de nettoyage).

#### C.1.2.2 Tableau des pages physiques

Une structure système centrale, que nous n'avons pas présentée, est le *tableau des pages physiques*. C'est là que sont stockés les attributs de chaque page physique, comme leur type ou le nombre de fois où elle est mappée. Ce tableau contient une structure avec les différents champs par frame, et est initialisée au démarrage du système par le noyau.

Ses champs peuvent également être utilisés pour la gestion de la propriété des pages, que l'on voit en section C.3.1 ; ainsi que pour un champs de bit pour l'implémentation d'un schéma de synchronisation use/destroy, détaillé en section C.2.2.2.

On utilise également le tableau des pages physiques pour faire certaines optimisations sur IA32. L'architecture mémoire de l'IA32 ne donne pas au noyau d'espace d'adressage séparé : il faut réserver certaines plages dans chaque espace d'adressage. Cela a pour conséquence que la taille de l'espace d'adressage réservée au noyau empiète sur celle des autres processus, ce qui rend difficie l'accès direct à la mémoire physique (sauf si il y a peu de mémoire physique). En conséquence, il y a fréquemment besoin d'installer des mappings temporaires de la mémoire (§ C.3.4.3), qui ont un coup important [ECCZ05]. En plaçant certaines données de certains objets systèmes dans le tableau des pages physiques (qui est mappé de manière permanente dans l'espace d'adressage du noyau), on peut économiser certains mappings temporaires, notamment lors de l'appel de service, et faire significativement baisser l'overhead de certaines opérations.

Le tableau des pages physiques contient enfin les estampilles utilisées pour la vérification des pointeurs universels (i.e. les pointeurs estampillés vers des pages physiques (§ 3.3.3.1)), dont nous détaillons le fonctionnement en section C.3.2.

[9] Il est toujours possible de prendre en compte des objets de taille supérieure à une page, comme nous l'avons fait pour l'objet thread, que nous avons séparé en UTCB et KTCB. La technique consiste à avoir des objets liés entre eux, de manière à ce que le nettoyage de l'un entraîne le nettoyage des autres.

#### C.1.2.3 Auto-révocation et révocation pendant l'utilisation

Un dernier aspect non abordé est le problème de l'auto-révocation, i.e. lorsqu'un thread demande au service mémoire de nettoyer la page contenant l'espace d'adressage ou le domaine dans lequel il est, ou de nettoyer la page de son KTCB ou UTCB. Après une telle opération, il devient impossible au thread de retourner de l'appel de service.

Une manière simple pour résoudre ce problème en monoprocesseur consisterait à interdire l'auto-révocation. Mais cette solution est à rejetter car ne se généralise pas bien au multiprocesseur.

L'auto-révocation est en effet un cas particulier de révocation d'un objet en cours d'utilisation. En SMP, on rencontre également la révocation inter-CPU d'objets, et interdire la suppression des objets en cours d'utilisation bloque l'exécution du processeur qui demande la destruction de l'objet pour un temps imprédictible. Ceci est contraire à nos principes de temps d'exécution prédictible et de révocation immédiate (§ 3.1.2.2).

Le problème de la révocation d'un objet en cours d'utilisation est résolu grâce au schéma de synchronisation use/destroy (§ 5.2.4) dans l'implémentation multiprocesseur, que nous voyons dans la prochaine section. L'auto-révocation en monoprocesseur y est un cas particulier.

## C.2 Implémentation multiprocesseur

Notre implémentation de ce système de mémoire virtuelle a été conçue pour une utilisation efficace sur multiprocesseur à mémoire partagée, mais n'a pas encore été testée sur multiprocesseur. Néanmoins, nous listons les différents problèmes de l'implémentation, et les solutions (de différents types) que nous avons apportées.

### C.2.1 Transformations en opérations atomiques

Les transitions du modèle doivent être réalisées de manière atomique, i.e. l'exécution de leur code ne peuvent pas être entrelacées. Un exemble de bug ou d'attaque qui surviendrait à cause d'un tel entrelacement est de type TOCTTOU : un thread $A$ vérifie le type d'une page, le type change dans un thread $B$, et $A$ effectue des modifications interdites sur la page.

En monoprocesseur, l'atomicité des transitions est réalisée par le masquage des interruptions dans le noyau. En multiprocesseur, on pourrait rajouter un spinlock global (comme cela est fait dans beaucoup de systèmes d'exploitations), mais cela rend le système de mémoire virtuelle, fréquemment utilisé, un goulot d'étranglement sur multiprocesseur.

Nous préférons réaliser l'atomicité des transitions de manière lock-free, par l'adjonction de plusieurs techniques. Celles présentées dans cette section vont permettre de regrouper des opérations mémoires afin de les rendre atomiques.

#### C.2.1.1 Groupement des champs

Si on dresse un bilan des métadonnées associées à une page, on trouve seulement :

- `type`, le type de la page. Il y a 6 types (présentement) d'objets systèmes, mais le nombre de types de nettoyages est plus important ;
- `mappings.all` et `mappings.rw`, qui comptent le nombre de fois où une page ou table de page est mappée ;
- `timestamp`, l'estampille utilisée lors de la validation des pointeurs universels ;
- éventuellement un champs permettant d'identifier le propriétaire de la page (voir § C.3.1)).

Nous nous préoccupons pour le moment seulement des champs `type`, `mappings.all` et `mappings.rw`.

**Minimisation de la taille du stockage utilisé** Si on regroupe les différentes étapes de chaque type de nettoyage, on compte au moins 10 types dans le système. Il faut donc 4 bits pour pouvoir les stocker.

Le maximum de mappings `mappings.all` pour une page est obtenu si toutes les pages du système sont des tables de pages, et référencent toutes la même page plusieurs fois. Sur une machine de 4Go de mémoire, cela représente une valeur maximale de $2^{30}$.

On peut estimer que ce scénario n'est pas réaliste, et que le maximum de mappings dans un scénario plausible est lorsqu'une page est présente dans chaque application du système (par exemple le code d'une librairie partagée). Une valeur de 16-20 bits semble raisonnable. Lorsque le nombre de mappings maximum est dépassé, on interdit de créer un mapping supplémentaire vers la page[10] ; dans ce cas l'alternative serait de copier la page.

Pour les mappings en écriture, un partage aussi important semble improbable. En général, pour que beaucoup de tâches se partagent un mapping en écriture, c'est qu'il s'agit d'un ensemble de programme s'exécutant en parallèle et communicant par mémoire partagée ; ainsi un ordre de grandeur raisonnable est le nombre de cœur. Une valeur de 8-10 bits semble suffisante.

Ainsi, si on prend 20 bits pour `mappings.all`, 8 bits pour `mappings.rw`, et 4 bits pour `type`, on n'utilise plus que 32 bits, et on peut mettre à jour tous les champs par l'écriture d'un mot de manière atomique.

[10] On peut se demander si cette inderdiction ne permet pas à un attaquant de provoquer un déni de service : en mappant une page beaucoup de fois, on peut empêcher les autres tâches de mapper cette page. Pour éviter cela, on tire partie du fait qu'on ne peut créer un mapping que vers des pages que l'on possède. Ainsi, si les mappings partagés entre des tâches ne sont possédés par aucune de ces tâches (e.g. est possédée par la politique d'allocation qui contrôle ces tâches), il n'y a pas de déni de service possible.

**Réutilisation de champs** Certains bits peuvent avoir plusieurs utilisations en fonction du contexte.

Par exemple, on constate que mis à part pour les pages de type `Dataframe`, aucune page ne peut être mappée en écriture. Ainsi ce compteur n'est pas utilisé lorsque la page est dans les autres types[11]. On peut alors le réutiliser ; par exemple pour les états de nettoyage, ce compteur peut servir à compter le nombre d'entrées à nettoyer restantes. Ce type de réutilisation est de type « union tagguée », i.e. le même champs a une utilisation différente selon la valeur d'un « tag » (ici le tag est le type).

Un autre type de réutilisation possible est la « somme de compteurs ». C'est ce que nous avons fait pour `mappings.all` lorsque qu'une page est de type `Pagetable` : le compteur compte la somme du nombre de mappings dans une table des pages et du nombre de mappings dans un répertoire des pages. Cette réutilisation est possible car nous ne sommes pas intéressés par les valeurs individuelles de ces compteurs, mais uniquement par le fait qu'ils valent tous les deux zéros.

#### C.2.1.2 Atomicité des changements de type

Si on regarde le code des transitions du diagramme, la majorité d'entre elles consistent à vérifier le type, éventuellement le nombre de mappings, puis de modifier ce type. Comme nous avons regroupé ces métadonnées en un seul mot, cette vérification/mise à jour peut être atomiquement réalisée à l'aide de l'instruction `CAS` (l'instruction compare and swap, qui compare le contenu d'une adresse à une valeur, et met une nouvelle valeur à cette adresse si la comparaison a réussi. On suppose ici que `CAS` renvoie un booléen vrai ssi la comparaison a réussi).

Généralement sur ces transitions, la vérification ne porte que sur certains champs, ce qui force l'implémentation à se faire de manière lock-free :

```
do {
    word_t old = *p_addr ;// p_addr est l'adresse des métadonnées de la page p
    check( ....) ; // vérifications sur old, les anciennes métadonnées
    word_t new = ...(old) ; // Création de la nouvelle valeur
} while( !CAS( p_addr, old, new)) ;
```

car les autres champs ont le droit de changer de manière concommittante au `CAS` ; ainsi le `CAS` a le droit d'échouer dans certains cas. C'est ainsi qu'on réalise les transitions de `Pagetable` vers `(PT→0,1024 restants)` par exemple, car le nombre de mappings peut être modifié de manière concurrente à la transition.

Cependant, parfois dans les transitions présentées, tous les champs sont connus (`mappings.all` à zéro implique que `mappings.rw` est également à zéro, et certains états de nettoyages impliquent également que `mappings.all` est à zéro. Dans ce cas, la comparaison peut se faire de manière wait-free :

[11] Notons que le compteur de mappings en écriture n'est pas non plus nécessaire pour le fonctionnement du système de mémoire virtuelle, et pourrait être supprimé. On peut parfois modifier le système de mémoire virtuelle pour que les mappings en lecture puissent rester malgré les changements de type, ce qui est utile pour la virtualisation ; dans ce cas les vérifications se font exclusivement auprès du compteur de mappings en écriture.

```
check( CAS( p_addr, expected, new)) ;
```

#### C.2.1.3 Atomicité des modifications de mappings

Les modifications que nous avons vu précedemment ne concernaient que les transitions « triviales », qui ne faisaient que changer de type. Nous voyons maintenant comment rendre atomique des opérations plus complexes, comme la modification de mappings.

**Échange atomique de mappings** Dans le code présenté Listing C.2, la lecture et l'écriture d'entrées dans la table des pages est faite de manière séparée. Cela peut conduire à des inconsistances sur les compteurs de mappings, par exemple si un thread $B$ modifie une entrée entre sa lecture et son écriture par un thread $A$.

Pour y remédier, il faut que la lecture et l'écriture d'entrées de table de pages soient faites atomiquement. Cela peut être réalisé sous IA32 par une instruction d'échange atomique. Sous d'autres architectures, on pourrait également boucler en utilisant des instructions de type `Load-Linked/Store-Conditional`, ou des lectures suivies de `Compare and Swap`.

```
entry_t old = xchange( &p.table[n], e) ;
```

**Modifications atomiques des compteurs de mapping** Les compteurs de mapping sont modifiés par incrémentation ou décrémentation. Il est nécessaire de faire ces opérations de manière atomique, sinon certaines race conditions pourraient faire que certains incréments apparaitraient comme « oubliés ». `FAA` (fetch and add) est une instruction qui permet d'incrémenter ou décrémenter une valeur mémoire de maniere atomique. Mais l'emploi de cette instruction ne suffit pas.

Tout d'abord, les incréments du compteur de mappings de la page mappée `to` dans les fonctions `swap_p{t,d}_entry` sont précédées d'une vérification du type de `to`, et les deux doivent être réalisées de manière atomique (sans cela, on pourrait casser certains invariants, comme le fait qu'une page de type `Zéro` implique que son compteur de mappings est à 0). On peut utiliser `CAS`, de la même manière que pour les transitions atomiques :

```
do {
    word_t old = *to_addr ; // to_addr sont les métadonnées de to
    check( old.type == ... )) ; // vérifie que to est du bon type
    new = ...( old) ; // modifie en local les métadonnées pour incrémenter
                      // mappings.all et éventuellement mappings.rw
} while( !CAS( to_addr, old, new)) ; // Remplace atomiquement old par new
```

Pour les décréments de compteur de mappings, il est inutile d'utiliser `CAS`, car les décréments sont fait sans aucune vérification sur `old_to`. On peut donc utiliser `FAA` tout simplement.

### C.2.2 Transitions avec opérations non-atomiques

Nous avons jusqu'ici réussi à transformer le code pour regrouper des accès mémoires afin de rendre certaines opérations atomiques. Malheureusement, cela ne suffit pas pour rendre certaines transitions atomiques, et en particulier l'opération de modification des mappings en entier atomique. Nous allons voir pourquoi et comment faire en sorte que les transitions restantes soient atomiques (au sens linéarizables [HW90]).

#### C.2.2.1 Modifications de mappings

**Impossibilité de rendre les modifications de mappings atomiques** Un autre problème concernant les modifications de mappings est le respect de la propriété « `p.mappings.all` = nombre de mappings de `p` ». La modification d'un mapping vers `p` augmente le nombre de mappings, et augmenter le compteur de mappings requiert forcément un autre accès mémoire. Pour pouvoir faire conserver l'égalité et modifier le nombre de mappings de manière atomique, il faudrait pouvoir modifier deux adresses disjointes simultanément. Cela est possible sur certains systèmes (l'instruction `DCAS` du Motorola [GC96, Mas92]), mais c'est assez rare, et n'est pas permis par l'IA32. L'ajout de mappings ne peut donc pas se faire en une opération atomique[12].

**Affaiblissement des invariants** La solution que nous avons retenue s'appuie sur l'affaiblissement des invariants (§ 4.1.3.3). Si on regarde la preuve, on n'a pas besoin de l'égalité entre le nombre de mappings et le compteur de mappings ; la seule propriété utilisée est

$$\texttt{p.mappings.all} = 0 \quad \Rightarrow \quad \text{nombre de mappings} = 0$$

Cette propriété peut être respectée lorsque les modifications se font dans un certain ordre, donné dans le pseudo-code qui suit :

```
swap_p[dt]_entry :
to.mappings.all++ ; // note : incrément atomique, comme vu précedemment
old = xchange( &p.table[n], new) ;
old_to.mappings.all -- ; // décrément atomique, comme vu précédemment
```

Dans ce cas, on a la relation :

| | | |
|---|---|---|
| `to.mappings.all` | = | nombre de mappings de `to` |
| | + | nombre de threads qui doivent mettre le mapping[a] |
| | + | nombre de threads qui doivent décrémenter le compteur[b] |

[a] i.e. les threads qui ont incrémenté `to.mappings.all`, mais pas encore installé l'entrée vers `to`
[b] i.e. les threads qui ont retiré l'entrée vers `to`, mais pas encore décrémenté `to.mappings.all`

[12] Normalement, on peut toujours faire les modifications et les valider atomiquement, comme le font les mécanismes de STM [Fra04]. C'est impossible ici car la structure de table de pages est fixée par le matériel. La solution que nous proposons fait en sorte de ne nécessiter aucune modification des structures de données.

et on a donc bien l'inégalité de la propriété. Ainsi le compteur de mappings est « décalé » par rapport au nombre réel de mappings, mais le décalage est toujours fait dans le même sens ; et en exécutant les transitions avec les interruptions masquées, ce décalage ne dure jamais longtemps.

**Prise en compte de la vérification** Avec cette dernière technique, nous avons presque rendu les transitions `swap_p{t,d}_entry` atomiques. Il reste cependant une dernière opération que nous n'avons pas prise en compte : pendant l'exécution de la fonction, la page doit continuer à être de type `Pagetable` ou `Pagedirectory`, sinon cela pourrait conduire à des inconsistances.

Vu le nombre d'emplacements mémoires auxquels il faut accéder, on ne pourra pas résoudre ce problème en faisant toutes les opérations atomiquement à l'aide d'un `CAS`, comme on l'a fait précedemment. Ce problème nécessite un autre mécanisme, le schéma de synchronisation use/destroy.

#### C.2.2.2 Schéma de synchronisation use/destroy

Nous avons vu précédemment que l'implémentation monoprocesseur pouvait être sujette à l'« auto-révocation », quand le processeur récupérait la mémoire d'un objet qu'il était en train d'utiliser (§ C.1.2.3). En multiprocesseur, il peut arriver des « révocations croisées », quand un processeur désire récupérer la mémoire d'un objet qu'un autre processeur est en train d'utiliser. La résolution de ce problème nécessite l'emploi d'une primitive spécifique, le use/destroy lock.

**Principe général** Nous allons utiliser le schéma de synchronisation use/destroy (§ 5.2.4) pour s'assurer que certaines conditions (e.g. `p.type == TYPE`) restent vraies pendant que d'autres actions sont entreprises sur `p` (e.g. modification du contenu de `p`).

Le principe est le suivant :

- si on veut entreprendre une action sur une page `p` requiérant qu'une condition $P(\texttt{p})$ soit vraie durant cette action, on « utilise » la page `p` ;
- si on effectue une action sur `p` tel que $P(\texttt{p})$ devienne faux, on « détruit » la page. Concrètement, la destruction se fait au moment des changements de transitions (en rouge sur la Figure C.1).

Voyons en détail où, comment et pourquoi ce schéma est appliqué pour chaque type de page.

**Type `Dataframe`** Pour ce type, nul besoin d'utiliser le schéma use/destroy. Les utilisations faites des pages de type `Dataframe` sont simplement des lectures et écritures, et ne peuvent se faire que si la page est dans une table des pages. Ce qui implique que le compteur de mapping de la page est à 1, et que la page ne peut pas être nettoyée ou changer de type.

**Type `Zéro`** Le type `Zéro` n'a pas besoin non plus d'utiliser le schéma use/destroy, car on n'accède jamais au contenu d'une page de type zéro. Les seules opération sur ces pages sont le changement de type, qui comme on l'a vu peuvent être faites atomiquement.

**Type `Pagetable`** Le type `Pagetable` nécessite l'emploi d'un schéma de synchronisation use/destroy : une table des pages ne doit être modifiée que si elle est de type `Pagetable` (sans cela, il serait possible d'écrire dans des entrées que le noyau a compté comme nettoyée, par exemple). Le plus simple est d'utiliser un use/destroy *lock*, de manière similaire à la Figure C.2.2.2 (reproduite de la Figure 5.2.4.2 page 193). À noter, la présence de « spinlocks non tournants » (§ 5.2.1.1).

```
error_t use( resource_t r)
{
  atomic{
    if( r->lock.destroyer)
      return EACCESS ;
    else r->lock.users ++ ; }
  ... // utilise ou modifie
  atomic{ r->lock.users -- ; }
}
```

```
error_t destroy( resource_t r)
 {
   atomic{
     if( r->lock.destroyer)
       return ECONCURRENCY ;
     else r->lock.destroyer = 1 ; }
   while( r->lock.users  != 0) ;
   ... // nettoie
   atomic{ r->lock.destroyer = 0} ;
}
```

FIG. C.2 – Implémentation théorique du use/destroy lock.

On peut faire un certain nombre d'améliorations par rapport à cette implémentation théorique (Listing C.3).

La première concerne `lock.users`. Le lock étant utilisé dans le noyau, cette valeur va de 0 à « nombre de processeurs - 1 », i.e. est une valeur relativement petite. Elle tiendra dans `mappings.rw`, qui vaut zéro quand la page est de type `Pagetable` ; on peut donc réutiliser ce champs.

La deuxième vient de l'observation que `lock.destroyer` empêche les modifications des entrées dans la table de page quand la page est en train d'être « détruite ». Or, un autre champ indique que la page est en train d'être « détruite », ou nettoyée : c'est le type. Ainsi, les tests et accès à `lock.destroyer` peuvent être remplacés par des accès au champ `type`.

Cela ne permet cependant pas de sérialiser les nettoyages [13], ce qui est l'autre utilité de `lock.destroyer`. Pour ce faire, on pourrait penser ajouter un spinlock, par exemple dans `mappings.rw`. Cela cause des difficultés, car on ne sait pas différencier l'état de la page où « le destroy lock est aquis » de l'état « il reste encore un utilisateur ». Pour éviter cela, nous avons défini un type spécial, `Destroying` ; ce type spécial est mis lorsque le destroy lock est aquis. Toute demande d'accès est refusée tant que ce type est mis.

[13] Le nettoyage en parallèle est inutile, les invalidations de caches risquant de créer plus d'overhead que le gain en parallélisme.

```
error_t swap_pt_entry( page_t p, index_t n, entry_t e) {
  do { word_t old = p.metadata ;
       check( old.type == Pagetable) ; // Pas de destruction en cours
       new = old ; new.mappings.rw ++ ;
  } while( !CAS( &p.metadata, old, new)) ; // Prend le lock atomiquement
  ... // Modifie les entrées (atomiquement)
  fetch_and_add( p.mappings.rw, -1) ; // Libère le "use lock"
}

(Pagetable) → (PT->0,1024 restants) :
  do { word_t old = p.metadata ;
       check( old.type == Pagetable) ;
       new = old ; new.type = (Destroying) ;
  } while( !CAS( &p.metadata, old, new)) ; // Change le type atomiquement ;
  while( p.mappings.rw != 0) ; // Attend qu'il n'y ait plus d'utilisateurs
  p.type = (PT->0,1024 restants) ; // Change de type définitivement

(PT->0,n restants) → (PT->0,0 restant) :
  do { word_t old = p.metadata ;
       check( old.type & typemask == PT->0) ; // masque la composante n
       new = old ; new.type = (Destroying) ;
  } while( !CAS( &p.metadata, old, new)) ; // prend le destroy lock
  do { swap_pt_entry( p, n, 0) ;
       if( pending_preemption) // Flag volatile mis à 1 si attente de préemption
         p.type = (PT->0,n restants)
  } while( n--) ;
```

Listing C.3 – Implémentation pratique du use/destroy lock pour le type `Pagetable`

**Notes**

- Lorsqu'on fait plusieurs modifications successives à la même table des pages, avec `swap_pt_entry`, on peut juste regarder le type de temps en temps pour voir si il n'y a pas de destruction en attente ; il est inutile de relâcher et reprendre le verrou. Cette vérification peut se faire en même temps que celle qui vérifie si il n'y a pas une préemption en attente.
- On ne peut pas utiliser `mappings.all` pour stocker le compteur du nombre d'utilisateurs de la page (i.e. en ayant `mappings.all` contenant la somme du nombre de mappings et du nombre d'utilisateurs de la page). Le problème est qu'on peut changer de type si le nombre de mappings est différent de 0, mais pas si le nombre d'utilisateurs de la page vaut 0, et on ne peut distinguer ce cas qu'en utilisant des compteurs différents pour ces deux valeurs.
- Plutôt que d'utiliser le type `Destroying`, il est également être possible d'utiliser l'implémentation des use/destroy lock « avec économie du bit destroyer » (§ B.1.1.3), qui permet également de sérialiser les destruc-

tions. Nous n'avons pas présenté cette implémentation par homogénéité avec les autres types de page.

**Types `KTCB`, `KDCB` et `Pagedirectory`** Ces types peuvent utiliser un use/destroy lock, de manière similaire à ce que nous avons présenté pour le type `Pagetable`. Mais cela ne suffit pas. Si ces pages peuvent être « utilisées » pour des modifications de capacités ou d'entrées de table de pages, elles peuvent également être « utilisées » lorsque l'objet système est nécessaire à l'exécution sur un processeur (i.e. un KTCB est utilisé si le thread correspondant est exécuté sur un processeur).

Dans ce dernier cas, il est nécessaire de « prévenir » le processeur correspondant pour qu'il cesse de l'exécuter (il est impossible pour ce processeur de vérifier périodiquement si l'objet est en attente de destruction, puisqu'il exécute du code utilisateur). Cela peut se faire en envoyant au processeur des interruptions entre processeur. Mais pour cela, il est nécessaire de connaitre le numéro du ou des processeurs qui utilisent l'objet système.

On peut utiliser deux techniques pour cela : en utilisant un bitfield par page, ou une table des ressources séparées. Les section B.1.2 et B.1.3 décrivent comment réaliser ces implémentations. L'implémentation choisie au final dépendra du nombre de processeurs dans la machine : si on peut placer le bitfield intégralement dans `mappings.rw` (jusqu'à 8 processeurs dans notre implémentation), l'utilisation du bitfield est la meilleure en termes de mémoire et de temps CPU. Par uniformité, on peut alors également utiliser les bitfields pour le use/destroy lock des `Pagetable`.

Lorsque le processeur à prévenir est le même que celui qui détruit la ressource, on est dans un cas d'auto-révocation. Ce cas demande une détection et traitement spécial : la révocation est faite, mais le thread rends la main au thread qui l'ordonnance.

**Type `UTCB`** Le type `UTCB` n'a pas besoin d'utiliser le schéma use/destroy. Il n'est accédé que pour des écritures par le noyau ou l'espace utilisateur, et uniquement quand le thread courant est « actif ». Ainsi, l'UTCB ne peut être utilisé que si le KTCB l'est, et il ne peut être détruit que par destruction du KTCB. Le use/destroy lock sur l'UTCB est donc implicite, et est celui pour le KTCB correspondant.

### C.2.3 Conclusion

L'implémentation multiprocesseur du service de mémoire virtuelle a un fonctionnement complexe. Mais le code généré est extrêmement compact, et les synchronisations entre processeurs sont minimales ; le système devrait donc être très efficace.

Pour s'assurer du fonctionnement correct de ce code, il serait intéressant de le prouver de manière automatique. Cela permettrait de s'assurer qu'il n'y a pas de « race conditions » oubliées. Les preuves manuelles que nous avons faites ne sont pas assez robustes, au sens où il est facile de ne pas voir que certaines propriétés ne tiennent pas dans certaines conditions (par exemple à cause de la réutilisation de certains champs). De plus, le code final en langage C (et non le pseudo-code présenté) est relativement difficile à lire ; l'usage de macros pourrait peut être le simplifier.

Cette implémentation multiprocesseur a également démontré que les garanties de synchronisations « partielles » (i.e. intermédiaires entre la section critique et l'absence de toute synchronisation) sont des outils très puissants. Le masquage des interruptions et l'emploi du use/destroy lock résolvent suffisament de problèmes de concurrence pour qu'on puisse écrire un code qui permette des accès aux données sans contraindre le parallélisme.

Enfin, notons que cette implémentation multiprocesseur est utile non seulement à Anaxagoros, mais également à tout système de virtualisation (ou implémentation bas niveau de la mémoire virtuelle) en multiprocesseur.

## C.3 Autres points importants

### C.3.1 Propriétés des pages et séparation politique/mécanisme

Un point dont nous n'avons pas parlé concerne le contrôle de l'accès aux pages physiques : il est par exemple important d'interdire aux tâches non-critiques de transformer les pages utilisées par les applications critiques. Nous allons ici préciser ces notions et montrer comment nous les avons réalisées.

#### C.3.1.1 Contrôle d'accès et notion de propriété

**Mécanismes de contrôle d'accès** Il a différentes utilisations possibles d'une page. Les droits de lecture, écriture, et éventuellement exécution sont gérés par le matériel (MMU et tables des pages, ou MPU et segments), qui peuvent être vues commes des capacités « spéciales »[14]. Les autres *droits d'utilisation* (utilisation de domaine, de thread) sont gérés par des capacités vers les pages. Le service mémoire gère les *droits de propriété*, i.e. l'utilisation de la mémoire pour créer ou détruire un objet système.

Le fait d'avoir plusieurs mécanismes séparés pour les différents types d'accès peut sembler pénible, et il peut sembler préférable de les « intégrer » (comme le font seL4 [EDE08], EROS [SSF99] ou CAP [NW77]). Cependant, le mécanisme et format de table des pages est fixé sur plusieurs matériel (dont l'IA32), ce qui rend cette intégration difficile. De plus, la séparation offre des avantages : le mécanisme de capacité est indépendant du mécanisme de MMU, ce qui rend le système plus facilement portable et plus à même d'utiliser efficacement le matériel. Enfin, la séparation des mécanismes encourage la séparation entre les droits d'utilisation et droits de propriété, respectant ainsi le principe du moindre privilège.

Dans cette section, nous nous intéresserons plus particulièrement aux droits de propriétés sur la page. Ce droit est essentiel pour des preuves de haut niveau : e.g. pour lister l'ensemble des domaines qui ont le droit de transformer les pages d'une application critique.

[14] Ce sont des capacités au sens où elles sont un couple (pointeur vers une page physique, droit d'accès) ; elles sont spéciales car en l'absence de timestamp on ne peut pas les révoquer en agissant sur la page pointée

**Notion de propriété** La notion de propriété est délicate, et en particulier sur les systèmes à capacité. Beaucoup de systèmes à capacité n'implémentent pas de notion de propriété [Lev84, p. 198]. Ils comptent généralement le nombre d'utilisateur de chaque ressource, et la libèrent quand ce nombre atteint 0. Cela crée des opportunités de déni de service.

System/38 dispose d'un propriétaire [Lev84, p. 147], mais la notion de propriété n'est pas gérée par capacité. Cela met en cause le modèle unique de contrôle d'accès, et rend difficile par exemple le transfert de propriété.

Les systèmes à ACL ne séparent pas l'usage et la propriété. De plus, pour des raisons de partage, il y a souvent plusieurs propriétaires pour un objet, ce qui est problématique pour faire de la comptabilité des ressources.

**Différenciation des concepts de propriétaires et de principal** Notre solution à ce problème est que le concept de propriétaire doit être différente de celui de principal. Dans Anaxagoros, cela signifie que les domaines (ou threads) ne sont pas propriétaires des pages physiques, ou encore que les pages physiques ne sont pas attribuées à des domaines ou threads.

Au lieu de cela, chaque page physique appartient à une *partition* de la mémoire ; et les principals disposent de droit d'utilisation de la partition (sous la forme classique de capacités). Ainsi, le comptage est aisé (chaque page appartient à une et une seule partition), le partage facilité (il suffit de partager le droit d'utilisation d'une partition), et le modèle de contrôle d'accès reste inchangé. Pour plus de souplesse, on peut utiliser un partitionnement hiérarchique de la mémoire (voir § 3.2.4.1).

KeyKOS et EROS fournissent une solution similaire à ce problème, sous la forme des space banks [SSF99, § 5.1]. Dans EROS, on peut voir les space banks comme les véritables propriétaires de la mémoire, et les principals n'ont que le droit d'utilisation de ces space banks. Le schéma d'EROS est différent en ce que les space banks sont également des principals (chaque space bank est un service du système).

### C.3.1.2 Implémentations du concept de propriétaire

La question est la suivante : comment un processus peut-il démontrer qu'il a droit de propriété sur une page physique, ou en d'autre termes, comment implémenter le concept de propriétaire ? Il existe plusieurs manières pour ce faire.

**Intervalles fixes** Le premier moyen consiste à partitionner les pages selon des intervalles contigus fixés. Dans les capacités vers le service mémoire sont alors inscrites l'intervalle correspondant. Certains hyperviseurs partitionnent la mémoire de cette manière [BDF+03, § 3.3.4].

Ce mécanisme a surtout l'avantage de la simplicité, et conviendrait pour tout usage dont les besoins mémoires sont connus par avance. Notons que si les intervalles sont fixés (et donc l'allocation mémoire est statique), l'usage de la mémoire (e.g. création de nouveaux mappings) est elle complètement dynamique. De plus rien n'empêche les droits de propriétés d'être transférés entre les principals.

Dans le cas limite chaque intervalle fait une page, et le système est complètement dynamique. Malheureusement cela conduit à une prolifération de capacités, qui pose

des problèmes de gestion et de consommation mémoire.

Notons que ce système n'implémente pas de hiérarchie de partition ; mais puisque l'allocation est statique cela ne présente pas un grand intérêt.

**Tags** Ce mécanisme est celui actuellement implémenté dans le service mémoire d'Anaxagoros. L'idée consiste à associer à chaque page une séquence de bits de taille fixe, appellé *token*. Les *tags* sont des séquences de bits de taille variable ; un tag est propriétaire d'une page si et seulement c'est un préfixe du token de la page. La Figure C.3 fournit un exemple.

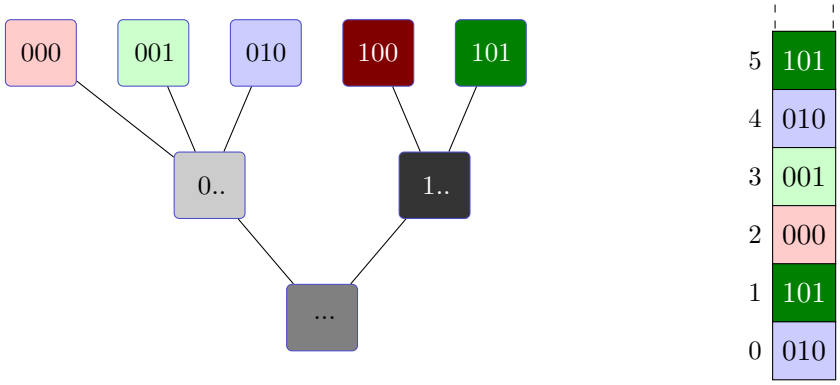



Fig. C.3 – Implémentation de la propriété avec des tags. À gauche se trouve l'arbre d'inclusion des partitions hiérarchiques ; à droite une assignation de la mémoire aux partitions. Dans cet exemple, le tag « 0.. » est propriétaire des pages 0,2,3,et 4.

Dans notre implémentation, les tags sont stockés comme mot de la capacité, et permet donc le droit d'accès aux pages correspondantes. Les tags sont implémentés de manière simple, comme un couple (longueur du préfixe, mot de taille fixe). Le test pour savoir si la capacité a le droit d'utiliser la page se fait de manière simple et efficace (basiquement un masque et une comparaison).

Un tag a le droit de changer le token d'une page qu'il possède tant que le nouveau token correspond aussi au tag. Ainsi, « 0.. » peut changer un token de « 010 » ver « 001 », mais pas vers « 100 ».

Cette implémentation a un dernier avantage, c'est que son utilisation ne demande aucune allocation mémoire par le service : les tags tiennent dans une capacité, et les tokens dans le tableau des pages physique.

Cette implémentation a néanmoins, à l'usage, quelques défauts. Le premier est qu'on ne peut pas révoquer un tag ; une capacité pointe toujours vers un tag (la raison est qu'il est difficile d'associer un timestamp à chaque tag, sans faire d'allocation mémoire dans le noyau). Ce défaut peut être contourné en contrôlant finement la transmission de capacités vers des tags (basiquement, un service de politique interdirait la copie de capacité vers un tag, et la capacité est invalidée lorsque le domaine qui la détient est détruit).

Un autre défaut est la création de hiérarchies. Si un niveau de hiérarchie peut contenir $n$ partitions, il faut lui réserver $log_2(n)$ bits de token. Les tokens sont de

petite taille (si on veut faire rentrer les tags dans un mot de 32 bits, les tokens font au maximum 27 bits) et il n'est pas simple de savoir comment les répartir. En pratique, il faut limiter la hiérarchie à deux niveaux ; garder par exemple 7 bits pour le premier niveau (permettant 128 machines virtuelles concurrentes) et 20 pour le deuxième (chacune peut avoir $10^6$ applications).

Le dernier défaut est que le changement de propriétaire demande d'effectuer un appel service d'un service de politique mémoire vers le service mémoire, i.e. souvent un appel de service récursif. Cela pose des difficultés d'implémentation parce que le rollforward lock ne peut pas marcher lorsqu'il contient un appel de service[15]. Cela ajoute également un certain surcoût aux demandes d'allocation mémoire.

**Mécanisme de « pool page »** Un mécanisme pas encore implémenté, mais qui semble très intéressant, est le mécanisme de pool page. Dans le mécanisme précédent, lorsque qu'un service de politique mémoire réserve un certain nombre de pages pour une application, il doit lui communiquer les pages qu'elle peut utiliser (autrement, l'application a le droit d'utiliser des pages, mais ne sait pas lesquelles (à moins de scanner l'ensemble des pages)). Cette communication se fait en transmettant un tableau contenant la liste des pages utilisables.

L'idée est d'utiliser cette liste de page comme base pour la gestion de la propriété. La liste de pages utilisable est stockée dans une page d'un type spécial `Poolpage`[16]. Lorsqu'une page veut être utilisée, on utilise la capacité vers sa `Poolpage`, et on désigne les pages par un indice dans la `Poolpage`.

Les `Poolpage`s ne peuvent pas être accédées[17] par les tâches qui l'utilisent. Pour comptabiliser les pages utilisées, on peut utiliser un simple bitmap, où un bit est à 1 ssi la page est libre. Afin d'éviter un parcours lent du bitmap, nous avons implémenté un mécanisme de « bitmap multiniveaux », tel qu'un bit dans un niveau est à 1 ssi un des 32 bits du niveau inférieur est à 1. Ce mécanisme permet d'allier le faible encombrement mémoire du bitmap (132 octets pour représenter 4Mo) avec des temps d'allocations/libérations très faibles (15 cycles sur notre machine cible, un Pentium disposant d'une instruction donnant l'index du premier bit à 1 dans un mot).

Les `Poolpage`s peuvent être modifiées par les services de politique directement, sans appel système. Pour cela, on limite la hiérarchie de politiques à deux niveaux. Le premier niveau divise les pages entre les politiques statiquement, par intervalle. Le deuxième permet aux politiques de diviser les ressources entre les applications. Le contrôle de propriété se fait alors en regardant si la page est bien dans la `Poolpage` (contrôle le premier niveau, i.e. que l'application a bien le droit d'utiliser la page), mais également si la page est dans l'intervalle du service (contrôle du second niveau, i.e. que le service de politique a bien le droit de donner l'accès à cette page).

Ces deux niveaux devraient être plus que suffisants pour la majorité des applications. Rajouter d'autres niveaux demanderait la création de tables d'indirections, ce qui est lent et occupe de la mémoire. Notons qu'il est possible pour une tâche

[15] Ces problèmes pourraient être résolus par l'implémentation d'un mécanisme de « CPU inhéritance » similaire à celui de Ford et al. [FS96]

[16] une `Poolpage` représente une zone mémoire accessible de 4Mo

[17] Empêcher l'écriture est nécessaire pour que la notion de propriété soit respectée ; empêcher la lecture peut éliminer des canaux cachés pour la confidentialité.

d'utiliser de la mémoire provenant de services mémoire différent.

Ce mécanisme a différents avantages. Tout d'abord, il combine la liste des pages utilisables avec le stockage de la propriété, fournissant ainsi un gain de temps et d'espace, et simplifiant l'implémentation des services de politiques mémoire. Ensuite, il permet aux services d'accéder directement à la propriété, économisant ainsi l'overhead d'un appel au service mémoire, et permettant l'utilisation de techniques comme le rollforward lock. Enfin, le droit d'utilisation d'une `Poolpage` est révocable, contrairement à la technique par tag.

Un désavantage est que la vérification de la propriété demande plus de temps (avec en particulier le besoin de mapper la `Poolpage` de manière temporaire) ; mais ce coût peut être amorti en utilisant le multicall.

**Implémentation indépendante du mécanisme de propriété** Nous avons vu qu'il y a différents moyens d'implémenter ce mécanisme de propriété, qui ont différents avantages et inconvénients à choisir en fonction de l'utilisation souhaitée. Mais rien n'empêche a priori de rendre l'implémentation du service de mémoire « indépendant » du mécanisme de propriété. La future implémentation de notre service mémoire tentera de réaliser cela, en séparant avec une API claire ce qui dépend du mécanisme de propriété de ce qui en est indépendant.

Un point intéressant à considérer est que le changement de propriétaire n'a pas d'impact sur le service mémoire autre que le changement de droit d'accès. En particulier, il n'y a pas de nettoyage de page ou de « destroy », i.e. les opérations en cours continuent lors du changement de propriétaire. Le service politique peut décider de réaliser ces opérations de lui-même, si il les réalise dans cet ordre :

1. Retirer la propriété de la page à la tâche $A$
2. Nettoyer la page (la mettre dans l'état `Zéro`, ou juste dans un état de nettoyage si la confidentialité n'est pas importante)
3. Donner la propriété de la page à la tâche $B$

Dans l'étape 2., le service conserve la propriété de la page mais ne la donne à aucune de ses sous-tâches. Notons que ces trois étapes peuvent être réalisées efficacement grâce au multicall.

Un dernier point intéressant est le maintien d'une table de translation « physique → pseudo-physique ». Cette table permet à une tâche de retrouver, à partir de l'adresse physique d'une page, le moyen de désigner la page. Cette table est utile pour la virtualisation [BDF+03, § 3.3.4]. Notre implémentation permet de déléguer la mise à jour de cette éventuelle table au service de politique : cela rend cette mise à jour plus efficace et retire du code au noyau.

### C.3.2 Gestion des estampilles

Une estampille permet de « dater » la création d'un objet dans le temps ; un pointeur ne peut pointer sur un objet que si il a été créé « après » l'objet vers lequel il pointe. Lors de la réutilisation d'une page mémoire pour y mettre un nouvel objet, tous les

```
{(KTCB), (KDCB)} → (DF→0,4096 restants) :
(Pagedirectory) → (PD→0,1024 restants) :
p.timestamp.up = 0xffffffff ;

(Zéro) → {(KTCB), (KDCB), (Pagedirectory)} :
p.timestamp = fetch_timestamp() ;

(Zéro) → {(Dataframe),(Pagetable),(UTCB)} :
assert( p.timestamp == 0xffffffffffffffff) ; // La page reste invalide

{(Dataframe),(Pagetable),(UTCB)} → (Zéro)  :
assert( p.timestamp == 0xffffffffffffffff) ; // La page reste invalide
```

Listing C.4 – Gestion des estampilles

pointeurs sur des objets occupant précédemment la même page mémoire ne peuvent plus fonctionner, car ils ont été créés avant (i.e. ont une estampille inférieure) à celle du nouvel objet.

Celon ce schéma, on pourrait ne modifier les estampilles qu'au moment de la création d'objets ; la création d'un nouvel objet créerait une nouvelle estampille, qui remplacerait l'ancienne et invaliderait tous les pointeurs universels sur les anciens objets.

Mais il est également possible qu'un objet devienne inutilisable sans pour autant qu'un nouvel objet vienne le remplacer. C'est le cas lorsqu'un objet est révoqué explicitement, ou même lorsqu'une page est dans l'état zéro ou un état de nettoyage (à ce moment, la page ne contient aucun objet).

Il y a donc deux instants où les estampilles sont modifiées : lors de la création d'un objet, et lors de la destruction de cet objet. Ces instants sont représentés par des flèches de couleurs différentes sur la Figure C.1 page 257.

Le code pour écrire ces estampilles a déjà été donné dans l'annexe A sur l'implémentation du système de capacité : il s'agit des écritures sur les objets dans les fonctions `revoke` et `bind` du Listing A.2.

Nous reproduisons ici (Listing C.4) une version de ce code concentrée uniquement sur les modifications d'estampille, et qui prend en compte la remarque suivante : certains objets systèmes n'ont jamais de pointeurs vers eux. C'est le cas des types `Dataframe` et `Pagetable` par exemple. Dans ce cas, il est inutile de mettre à jour leurs estampilles, ni lors de la création, ni lors de la destruction ; optimisation d'autant plus intéressante que les objets `Dataframe` et `Pagetable` sont de loin les objets les plus fréquemment créés.

Dans l'implémentation actuelle, les objets qui peuvent être pointés sont les KTCBs, dont la création crée une capacité ; les KDCBs, qui en plus d'être pointés par une capacité sont également pointés par tous les threads qui sont dans le domaine représenté par le KDCB ; et les répertoires de pages, auxquels ne sont pas associés de capacités, mais qui sont pointés par les domaines qui sont dans l'espace d'adressage représenté par le répertoire de pages.

L'UTCB est pointée par le KTCB, mais pas avec une estampille, car la destruction de l'UTCB entraîne celle du KTCB.

### C.3.3 Consistance du TLB assurée et exportée

Un aspect dont nous n'avons pas parlé est la consistance du TLB. En effet, lorsque les tables des pages sont modifiées, le TLB qui est un cache de ces tables des pages, n'est généralement pas mis à jour ; c'est le cas sur l'architecture IA32. C'est la raison pour laquelle il est nécessaire d'invalider explicitement certaines entrées de ce cache, afin de le forcer à charger les dernières modifications.

#### C.3.3.1 Consistance et optimisations des flush TLB en monoprocesseur

L'IA32 est une architecture où le TLB n'est pas taggué, i.e. les entrées TLB ne contiennent que celles de l'espace d'adressage courant. Le TLB peut être invalidé entrée par entrée, en spécifiant l'adresse virtuelle à invalider ; ou « flushé » dans son intégralité, à chaque fois que l'espace d'adressage courant est changé[18]

Toutes ces opérations d'invalidation du TLB sont très coûteuses en temps CPU, et il est important de chercher à les minimiser. Par contre, une invalidation incorrecte des entrées du TLB peut conduire à des problèmes de sécurité. Par exemple, si une page de type `Dataframe` est devenue de type `Pagetable`, une entrée TLB non mise à jour permettrait à l'espace utilisateur d'écrire directement dans une table des pages.

Pour cette raison, nous avons décidé que le noyau n'effectuerai des invalidations de TLB que pour des raisons de sécurité ; les autres invalidation doivent être faites sur demande par les applications. Ceci suit le principe d'exportation de la gestion de la consistance (§ 5.3.2), appliqué ici à la gestion de la consistance du TLB. Ainsi, si une tâche de l'espace utilisateur sait qu'elle n'a pas besoin d'invalider une entrée (parce qu'elle sait qu'elle n'est pas présente dans le TLB, ou qu'elle ne va pas s'en servir tout de suite, ou qu'elle préfère grouper les invalidations), elle n'a pas besoin de le faire.

Les seules invalidations nécessaires pour raison de sécurité sont lorsqu'une page accessible en écriture cesse de l'être pour être réutilisée (soit pour un autre processus, soit pour un objet système). Il doit y avoir une invalidation de tous les mappings[19] vers cette page avant sa réutilisation.

On pourrait penser pouvoir invalider individuellement les entrées à chaque fois que l'entrée d'une table de pages d'une page de donnée accessible en écriture. Le problème principal est que l'instruction d'invalidation d'entrée du TLB prend en argument une adresse virtuelle, et non une adresse physique. Or, quand on supprime une entrée d'une table des pages, le noyau ne connait pas à quelle(s) adresse(s) physique(s) cela correspond.

Une autre solution consiste à flusher tout le TLB à chaque fois qu'une entrée est modifiée, mais cela aurait un coût prohibitif. Nous proposons deux moyens pour le

[18] il y a également des entrées « globales », qui peuvent rester lors d'un changement d'espace d'adressage. Ces entrées posent des problèmes de confidentialité, et peuvent permettre à une tâche d'empêcher à une autre l'accès à ses mappings. Dans Anaxagoros, seul les mappings du noyaux peuvent être globaux, et le noyau veille à ce que toutes les autres entrées ne soient pas globales

[19] Il faut faire attention au fait qu'une page peut être mappée plusieurs fois.

diminuer :

- Si une entrée n'est présente que dans une table des pages, qui n'est mappée que dans une page, on peut retrouver l'adresse virtuelle du mapping[20]. Dans ce cas, on peut utiliser l'instruction d'invalidation sélective des entrées.
- Si il y a eu un flush entre le moment où le nombre de mappings de la page est tombé à 0, et le moment où la page a commencé son nettoyage (i.e. a changé de type pour `(DF→0,4096 restants)`), il n'y a pas besoin d'invalidations supplémentaires. Pour cela, nous estampillons l'instant où le nombre de mappings de la page tombe à 0, et l'instant du dernier flush de TLB, et comparons les dates au moment de la destruction de la page ; le flush n'a lieu que si nécessaire. L'espace utilisateur peut en particulier tirer partie de cette connaissance pour grouper des flushs de TLB.

Les cas couverts par ces deux techniques sont relativements fréquents et permettent d'éliminer drastiquement le nombre de flushs de sécurité à exécuter, surtout si la tâche exécutée connait ces optimisations.

Certains autres systèmes demandent également aux OS virtualisés de faire en sorte d'optimiser les flush de TLB dans leur interface avec l'hyperviseur ; on peut ainsi citer Xen [BDF+03, § 3.3.3].

#### C.3.3.2 Contrôle des TLB shootdowns en multiprocesseur

**Présentation** En multiprocesseur, la suppression d'un mapping ne concerne pas seulement le processeur qui supprime ce mapping, mais tous les processeurs du système. Lorsqu'un mapping supprimé est réutilisé, il faut s'assurer qu'il n'est pas accessible par un autre processeur, i.e. qu'il ne soit pas resté dans le TLB de l'autre processeur. La solution classique à ce problème s'appelle le TLB shootdown [BRGH89], et consiste à envoyer des interruptions inter-processeurs à tous les processeurs qui peuvent utiliser l'entrée pour qu'ils la suppriment de leur TLB.

**Première approche et problématique** En première approche, on peut envoyer les requêtes de shootdown et simplement étendre les optimisations décrites ci-dessus au multiprocesseur. Si une entrée n'est présente que dans une table des pages, qui n'est mappée que dans un seul répertoire des pages, seuls les processeurs qui utilisent ce répertoire des pages doivent invalider l'entrée. De plus, tous les processeurs qui ont connu un flush de leur TLB entre le moment où le nombre de mappings est tombé à 0, et le moment où la page a commencé son nettoyage, n'ont pas besoin de recevoir un IPI.

Mais ces optimisations, bien qu'utiles, ne préviennent pas contre certaines attaques : une tâche peut tout a fait mapper plusieurs pages, puis les nettoyer, afin de provoquer un shootdown sur tous les processeurs[21] ; et faire cette opération en

[20] On peut faire cela en inscrivant dans le tableau des pages physique l'adresse physique de l'entrée de la table des pages qui pointe vers la page, et l'adresse physique de l'entrée du répertoire de pages qui pointe vers la table des pages.

[21] sauf si ils ont flushé leur espace d'adressage en même temps

boucle. Le traitement de ces shootdowns en continu peut ralentir l'exécution des tâches exécutées sur les autres processeurs, et leur faire manquer leurs échéances.

Il faut donc à la fois assurer que toutes les entrées de tous les processeurs vers une page physique soient bien supprimées lors du nettoyage de la page ; et en même temps qu'on n'envoie pas de demande de shootdown aux processeurs non-concernés. Nous avons envisagés des solutions pour ce problème.

**Élément de solution 1 : shootdown volontaire** Un premier élément de solution consiste à interdire la réutilisation de pages tant que tous les processeurs n'ont pas évincés leur entrée. C'est à dire que ce n'est pas au noyau de décider d'envoyer des interruptions de shootdown, mais à l'espace utilisateur ; le noyau ne fait que vérifier que le shootdown a bien été effectué. Il s'agit encore d'une exportation de la gestion de la consistance.

Notons que cela ne résoud pas le problème qu'il faille envoyer des shootdowns à des processeurs qui ne sont pas concernés, mais donne à l'espace utilisateur le moyen de contrôler quand ces demandes sont effectuées. Dans les faits, une tâche pourra supprimer une page, puis attendre un certain temps qu'il y ait eu changement de contexte (donc flush sur IA32) sur tous les processeurs, avant de pouvoir la réutiliser, et cela sans qu'il y ait de shootdown. Cette solution suppose seulement de faire confiance à tous les ordonnanceurs pour changer de tâches « régulièrement ». Mais cette solution semble difficile à mettre en œuvre.

**Élément de solution 2 : partitionnement** La solution précédente n'est pas entièrement satisfaisante, car on peut avoir envie de réutiliser des pages « promptement ». Pour envoyer des shootdowns seulement aux processeurs concernés, il faudrait comptabiliser pour chaque page la liste des processeurs en train d'utiliser ces pages.

Il est difficile et très coûteux de faire cela ; essentiellement cela demanderait, à chaque changement de contexte, de parcourir le répertoire de page afin de recenser les tables de pages et les pages utilisées, et de marquer chacune comme étant utilisée par le processeur. Il faudrait également les marquer comme n'étant plus utilisées au changement de contexte suivant.

Au lieu de cela, on peut restreindre chaque page à ne pouvoir être utilisée que par certains processeurs. À chaque page physique `p`, on associe un ensemble `p.usable_cpus` des processeurs qui peuvent utiliser la page, de la manière suivante :

1. `p` ne peut être dans une entrée d'une page `p'` de type `Pagetable` que si `p'.usable_cpus` est inclus dans `p.usable_cpus` (i.e. un processeur non-utilisable par $p$ ne doit pas l'être par $p'$) ;
2. `p'` ne peut être dans une entrée d'une page `p''` de type `Pagedirectory` que si `p''.usable_cpus` est inclus dans `p'.usable_cpus` (i.e., un processeur non-utilisable par $p'$ ne doit pas l'être par $p''$) ;
3. si `p''` est de type `Pagedirectory`, seuls les processeurs appartenant à `p.usable_cpus` peuvent utiliser `p` comme espace d'adressage.

**Propriété C.13.** *Si un CPU $c$ accède à une page `p`, alors $c \in$ `p.usable_cpus`*

*Démonstration.* $c$ accède à `p` implique qu'il existe une table des pages `p'` et un répertoire des pages `p''` tels que $c$ utilise `p''` qui pointe vers `p'` qui pointe vers `p`.

Or, $c$ utilise `p''` implique que $c$ est dans `p''.usable_cpus` (item 3), ce qui implique que $c$ est dans `p'.usable_cpus` (item 2), ce qui implique que $c$ est dans `p.usable_cpus` (item 1). □

Par contrapposée, si $c$ n'est pas dans `p.usable_cpus`, alors $c$ ne peut pas accéder à `p`. Il est donc inutile de lui envoyer une requête de TLB shootdown.

En mettant `usable_cpus` à des valeurs adéquates, on peut ainsi éviter d'envoyer des requêtes de shootdown à certains CPUs, ce qui résoud le problème. On peut par exemple réserver des processeurs à des codes critiques, pour garantir que les tâches non-critiques ne peuvent pas impacter les processeurs pour le code critique.

**Notes**

- Ce mécanisme est à la fois une garantie de sécurité, mais également une optimisations pour certains shootdowns. En contrepartie, la « migration » d'une tâche d'un CPU à un autre CPU demande de modifier les indicateurs de toutes les pages, la rendant beaucoup plus lente. En pratique, ce mécanisme sépare donc pour chaque tâche les CPUs en groupes, et interdit la migration de la tâche d'un groupe de CPU à un autre.
- Ce mécanisme peut être facilement implémenté avec des bitfields, les opérations d'inclusion d'ensemble se traduisant par des opérations simples sur les masques.
- Ce mécanisme permet le partage certaines de pages par tous les processeurs (par exemple pour des librairies) sans problème. Dans ce cas, une modification de ces pages requiérerait un TLB shootdown de tous les processeurs qui utilisent la page.
- En réalité, ce mécanisme doit être couplé avec la gestion de la propriété des pages, en particulier en ce qui concerne la modification du champs `usable_cpus`.

**Solutions pour d'autres matériels** Différentes modification peuvent être apportées pour d'autres matériels.

Par exemple, certaines architectures permettent de « taguer » le TLB, ce qui permettrait de supprimer selectivement certaines entrées. Il serait intéressant de voir comment le tag peut être intégré à notre mécanisme de mémoire virtuelle pour optimiser les flushs et shootdowns de TLBs.

Enfin, une bonne solution serait d'avoir un mécanisme hardware dédié, qui permette de flusher une page *physique* de tous les TLBs de tous les processeurs. Le coût hardware pour un tel mécanisme est peut-être important (il faut comparer l'adresse physique à toutes les entrées présentes), mais faciliterait énormément toute la gestion du TLB.

### C.3.4 Autres points de conception du noyau

#### C.3.4.1 Points de préemption explicites et `pending_preemption`

Nous avons vu que certaines parties du code de mémoire virtuelle nécessitent d'être exécutées de manière non-préemptible (en particulier, cela permet d'assurer que le compteur de mappings reste « proche » du nombre réel de mappings). En général, l'implémentation de notre noyau suit le modèle d'exécution « par interrupt » [FHL+99], i.e. avec une pile par processeur, et l'exécution du noyau ne peut donc pas s'arrêter n'importe où.

Comme certaines opérations peuvent être longues, on permet néanmoins des préemptions dans le noyau à des endroits contrôlés ; ce sont les points de préemption explicites.

La manière la plus évidente d'implémenter ce mécanisme consiste à démasquer puis remasquer les interruptions (c'est ainsi que c'est réalisé dans L4/Fiasco [STT+09, p. 2]). Ainsi si une interruption est en attente, elle va pouvoir s'exécuter à ce moment.

Le problème de cette implémentation est son coût caché : quand on pose un point de préemption explicite, il faut être prêt à être interrompu. Cela signifie avoir libéré tous les verrous, fait tout le nettoyage nécessaire etc ; pour les reprendre ensuite dans le cas commun où il n'y a pas eu de préemption.

Plutôt que de faire cela, nous avons mis en place un drapeau `pending_preemption`. Le noyau s'exécute avec les interruptions non masquées et les interruptions sont traitées normalement. Lorsqu'une interruption conduit à une préemption (e.g. le timer) et que le processeur est dans le noyau, le flag `pending_preemption` est mis à 1 pour le processeur concerné. La préemption explicite se fait par simple lecture de ce flag, et tout le nettoyage peut se faire une fois qu'on sait qu'il y a une préemption en attente. Dans l'implémentation actuelle, le noyau vérifie ce flag pour certaines opérations longues, et entre chaque appels de multicall.

Outre l'économie des libérations/reprise de verrous , comme les interruptions sont non masquées, elles peuvent être traitées plus vites. On peut également programmer un watchdog pour contrôler qu'il n'y a pas de blocage dans le noyau, ce qui est utile pour la sûreté de fonctionnement. Le code est généralement plus simple dans le noyau. Il y a donc différents avantages à cette implémentation des points de préemption explicite.

#### C.3.4.2 Fonction `check`

**Présentation et utilisation** Dans le modèle que nous avons présenté, nous avons utilisé les `check`s comme des gardes, c'est à dire que la condition en argument est toujours vérifiée. Ce n'est pas toujours le cas en réalité, et on utilise les `check`s pour s'assurer que des propriétés sont vraies. Nous présentons ici cette fonction, utilisée non seulement dans le service mémoire mais dans tout le système.

Il faut différencier l'utilisation de `check` de celle de fonctions comme `assert`. Une condition fausse dans `assert` indique une défaillance, qui peut provenir du programme, du système d'exploitation ou du matériel ; on utilise `assert` à des fins de débugage (par exemple pour implémenter de la programmation par contrat), ou pour la programmation défensive.

On utilise `check` pour vérifier des conditions qui peuvent être fausses en fonctionnement normal, et qui proviennent d'une entrée du programme qui est incorrecte. Des exemples où `check` est utilisé dans Anaxagoros sont

- lors de la vérification des arguments d'appels systèmes (dans le noyau) ou d'appels de services (par les services) ;
- lors du parsing de paquets réseaux[22], fichiers tars ou ELF[23] ;
- lors de la vérification que le temps entre interruptions successives n'est pas trop court pour des tâches sporadiques.

En résumé, *dès qu'on se repose sur une propriété dépendant d'entrées extérieures au programme, il faut vérifier que cette propriété est vraie à l'aide de* `check`. L'intérêt de cette fonction unique simple `check` est qu'on identifie facilement les vérifications de sécurité faites, et son emploi facile encourage le programmeur à faire toutes les vérifications nécessaires. De plus, l'utilisation de cette fonction simple réduit le code, et donc réduit le nombre de bugs (Bernstein [Ber07, § 4.2] propose également de factoriser ces codes de vérifications dans le but de réduire la taille du code).

Sur les 12500 lignes de code d'Anaxagoros (dont 8000 pour le noyau et les include et 1350 pour les librairies partagées), il y a 280 appels à `assert` et environ 150 appels à `check` (souvent situés dans des fonctions couramment appellée). L'utilisation de ces fonctions sont donc fortement utile dans du code sécurisé.

**Usage et implémentation** La majorité des vérifications concernent la vérification d'entrées de l'espace utilisateur, le plus souvent les valeurs des arguments passés (e.g. vérifier qu'une adresse physique est bien alignée sur une page) ou le séquencement des appels (c'est ce que font les `check`s dans les listings présentés précedemment).

En général, la seule action requise lorsqu'un `check` échoue est de retourner à l'utilisateur avec un code d'erreur « argument incorrect ». Le noyau pourra se charger de prévenir le module de surveillance du programme utilisateur, si il y en a un (voir section 3.1.3.1).

Il arrive parfois que des actions de « nettoyage » soient requises en cas d'erreur. Cela ne peut arriver que dans le noyau, car les autres services ne contrôlent pas les instants où le temps CPU peut être révoqué, et la ressource peut donc ne jamais être nettoyée. Ces autres services doivent s'arranger pour que l'absence de nettoyage laisse la ressource dans un état défini, afin qu'un autre thread puisse faire ce nettoyage ; il n'y a donc pas de nettoyages autre que le retour à l'utilisateur.

Dans le noyau, et en particulier dans le service mémoire, il peut donc arriver qu'il y aie des actions de nettoyage à effectuer. Il y a deux conditions de génie logiciel à prendre en considération :

- Les actions de nettoyage à entreprendre sont différentes aux différents endroits du code ;

[22] [Rip03, p. 40] parle du « ping of death », qui est un exemple classique d'attaque à la suite d'une vérification d'entrée non effectuée

[23] le bug CVE-CAN-2004-1234 permettait l'arrêt du système lorsqu'un fichier ELF était malicieusement formé

- Un même code peut devoir entreprendre plusieurs actions, par exemple lorsqu'il y a un `check` dans une fonction utilisée à plusieurs endroits différents.

Le mécanisme le plus adéquat pour résoudre ce problème est le mécanisme d'exception, mais il n'existe pas en language C. Nous avons implémenté un mécanisme de remplacement, qui consiste à empiler des « closures » sur une pile d'exception. Le bas de la pile est occupé par la fonction qui retourne à l'utilisateur. Ce code est présenté Listing C.5

**Notes**
- Des vérifications de programmation défensive, non présentées, vérifient que la pile d'exception est bien vidée à chaque sortie du noyau.
- Le `EXPECT_FALSE` est une macro qui, si le compilateur le permet (c'est le cas de gcc), oriente le prédicteur de branchement pour indiquer que le saut a peu de chance d'être pris.

### C.3.4.3 Mapping temporaire des pages

Notre modèle de service mémoire permet d'utiliser n'importe quelle page pour n'importe quel usage, ce qui est très utile et pratique. Mais cela pose quelques problèmes d'implémentation sur l'IA32 concernant l'accès aux adresses physiques.

En effet, pour accéder à la mémoire il faut passer par le mécanisme de translation d'adresse[24], ce qui implique qu'il faille installer des mappings mémoires qui permettent l'accès à la mémoire physique. De plus, l'IA32 n'a pas d'espace mémoire propre au noyau : le noyau a seulement une partie réservée dans chaque espace d'adressage qui est relativement petit. En conséquence, l'espace d'adressage du noyau est très petit, et on ne peut pas mapper toute la mémoire physique dans l'espace d'adressage du noyau : pour accéder à la mémoire physique, il faut installer des mappings temporaires.

Par exemple, pour pouvoir écrire dans une table des pages, le service mémoire doit d'abord installer un mapping qui lui permet d'écrire dans la table des pages.

L'implémentation actuelle de ce mécanisme est relativement simple : le noyau fournit différents « slots » (2 dans l'implémentation actuelle) dans laquelle installer les mappings. Le code qui installe un mapping est responsable de choisir le slot, d'installer la page dedans, et d'éventuellement la libérer quand elle a fini d'écrire dans la page[25]. Elmeleegy propose une implémentation plus performante en moyenne (et plus complexe) [ECCZ05], mais dont l'intérêt est moindre pour des applications temps-réel.

**Notes**
- Ces mappings temporaires ne demandent pas de modifier le compteur du nombre de mappings installés. En multiprocesseur, le décompte est fait par le use/destroy lock : le code est écrit de manière à respecter la propriété « mapping temporaire installé ⇒ on est dans une section use pour la page ». Cette propriété est importante, car c'est elle qui garantit qu'il est impossible de modifier une page en cours de destruction.

[24] on peut en fait désactiver le mécanisme de translation de manière temporaire, mais cela occasionne un flush complet du TLB, et est donc très coûteux.

[25] cette libération sert en fait uniquement en cas programmation défensive, pour vérifier que tous les slots sont bien libérés

```
struct undo_closure
{
  void (*function)(void *arg) ;
  void *arg ;
} ;

void return_to_user( error_t code) ;

#define UNDO_CLOSURE_MAX_HEIGHT 3
static struct undo_closure undo_closure_stack[UNDO_CLOSURE_MAX_HEIGHT]
  = {{return_to_user, (void *) ERROR_BAD_ARGUMENT}} ;

static unsigned int undo_closure_stack_height = 0 ;

void check_push_undo( void (*undo_fun)(void *), void *arg)
{
  assert( undo_closure_stack_height < UNDO_CLOSURE_MAX_HEIGHT) ;
  undo_closure_stack[undo_closure_stack_height++]
    = (struct undo_closure){undo_fun, arg} ;
}

// retire les nb premieres closures de la liste
void check_remove_undo( void)
{
  assert(undo_closure_stack_height > 0) ;
  undo_closure_stack_height-- ;
}

#define check( expression)                          \
  do { if( EXPECT_FALSE( (expression) == 0))        \
      check_fails() ;                               \
  } while(0)

void check_fails(line,file)
{
  do {
      int i = undo_closure_stack_height ;
      undo_closure_stack[i].function( undo_closure_stack[i].arg) ;
  } while(--undo_closure_stack_height) ;
}
```

Listing C.5 – Implémentation des `check`

- Pour certaines opérations (dont l'appel de service), on peut économiser des mappings temporaires si certaines informations sont stockées dans le tableau des pages physiques, au lieu de les stocker dans la page. Ces optimisations contribuent énormément à la réduction du temps d'appel de service.
- Tout ce mécanisme peut être économisé si la mémoire physique peut être entièrement incluse dans l'espace d'adressage du noyau. Cela conduit à des gains de performances mesurables, et ce malgré nos optimisations. On peut en particulier faire cela sur des espaces d'adressages en 64 bits.

#### C.3.4.4 Multicall

Le multicall est le traitement des appels de services par lot, i.e. la possibilité d'invoquer plusieurs méthodes sur plusieurs capacités différentes en un seul appel système.

**Intérêts** Il y a plusieurs avantages à faire cela. Le premier est que cela permet d'économiser tous les coûts de mise en place d'une communication : changement d'espace d'adressage, vérifications de capacités, etc. qui peuvent être coûteux. Nous allons plus loin dans notre implémentation « hiérarchique » du multicall que nous allons décrire, et nous permettons de factoriser au maximum tous les coûts de mise en place de plusieurs opérations (prise d'un use lock, mappings temporaires, etc). Le multicall permet donc un système plus efficace. Ceci est particulièrement important pour le service mémoire, où le traitement par lot des modifications des entrées dans les tables de pages est important pour la performance [BDF+03].

Les autres avantages sont sur la structuration. Le multicall encourage l'utilisation de l'interface RISC pour les services (§ 5.3.3), i.e. la décomposition en méthodes qui effectuent des actions simples. Cela simplifie l'écriture du service partagé, et le système devient globalement plus flexible : le multicall permet de « composer » des appels systèmes de haut niveau à la carte, en enchaînant les appels à plusieurs services. Cela est rendu possible parce que le multicall est utilisé dans tout le système.

Notons que la présence d'une erreur durant l'exécution d'un multicall est problématique, car il est difficile de retrouver où s'est arrêtée l'exécution. Heureusement, Anaxagoros est conçu de sorte que toutes les erreurs dans les appels systèmes sont imputables au programme qui les utilise (i.e. il en a fait une mauvaise utilisation). Ainsi, un programme multicall bien écrit est garantit de s'exécuter sans encombre jusqu'à terminaison.

**Implémentation** L'implémentation se base sur plusieurs idées. Pour un service donné, on fait correspondre à chaque méthode un numéro, appellé *opcode*. Le multicall consiste à placer dans un buffer le numéro de méthode suivi du nombre et des arguments de la méthode ; l'ensemble opcode plus arguments est appellé *instruction* (en effet, ce schéma ressemble au format des instructions machines).

Les instructions machines sont généralement exécutées séquentiellement (sauf instruction de saut), mais ce n'est pas le cas pour notre jeu d'instruction. Dans chaque

instruction est écrite l'adresse de l'instruction suivante ; ainsi chaque instruction est une instruction de saut. La Figure C.4 représente le format d'une instruction.

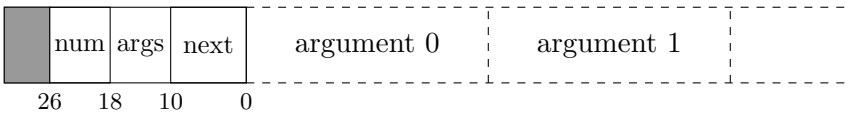



FIG. C.4 – Format d'une instruction multicall. `next` est l'adresse de la prochaine instruction, `args` le nombre d'arguments, et `num` le numéro de méthode. Certains bits ne sont pas encore utilisé.

Il y a plusieurs raisons pour ne pas exécuter les instructions séquentiellement. La première est que cela permet de laisser des trous pour les données : les « programmes » multicall peuvent ainsi facilement mélanger instructions et données dans le même buffer.

La deuxième raison est la composition des programmes multicall. Il est important de minimiser le temps d'écriture du programme multicall et de faciliter cette écriture. Faire un saut à chaque instruction permet d'écrire le programme dans un ordre différent de celui dans lequel il est exécuté, ce qui facilite son écriture (voir l'exemple plus bas).

Certains numéros de méthode sont réservés, et exécutent les mêmes actions pour tous les services du système. Pour l'instant, ces méthodes réservées sont :

- `return` demande au service de retourner vers l'appelant (par invocation de la capacité de retour). C'est l'instruction de fin d'un programme multicall.

- `transfert` demande au service d'invoquer une autre capacité en « tail call » (similairement à [SFS96]). Cela permet de faire des invocations sur plusieurs capacités, voir plusieurs services, successivement.

- `remove_cap` demande au service d'invalider une capacité passée en argument. On fournit généralement à un premier service les capacités nécessaires pour son invocation, et avant que ce premier service transfère l'exécution à un deuxième, on lui demande de supprimer ces capacités pour éviter que le deuxième service ne puisse les utiliser. Notons qu'on pourrait demander le nettoyage de données comme on le demande pour les capacités ; notons également que le premier service a accès aux capacités du deuxième, et qu'il faut donc lui faire confiance.

Enfin, le multicall peut être implémenté hiérarchiquement. C'est à dire que certains opcodes peuvent démarrer l'exécution d'un autre ensemble d'instructions. Par exemple, plutôt que d'appeller de manière répetée la méthode « crée entrée dans la table des pages », on fournit à cette méthode une liste des entrées à créer. Cela permet de factoriser les coûts de mise en place d'une méthode au maximum. De même, les fonctions de lecture et écriture ne prennent pas en argument des caractères, mais des chaînes de caractères.

**Exemple : fonction `writev`** Nous allons partir d'un exemple qui montre comment implémenter l'appel système `writev` avec le multicall. `writev` permet d'écrire dans un descripteur la concaténation de plusieurs buffers de données en un seul appel, et est donc équivalent à l'appel de plusieurs `write`. L'appel est donné Listing C.6 ; le programme multicall résultant est donné Figure C.5.

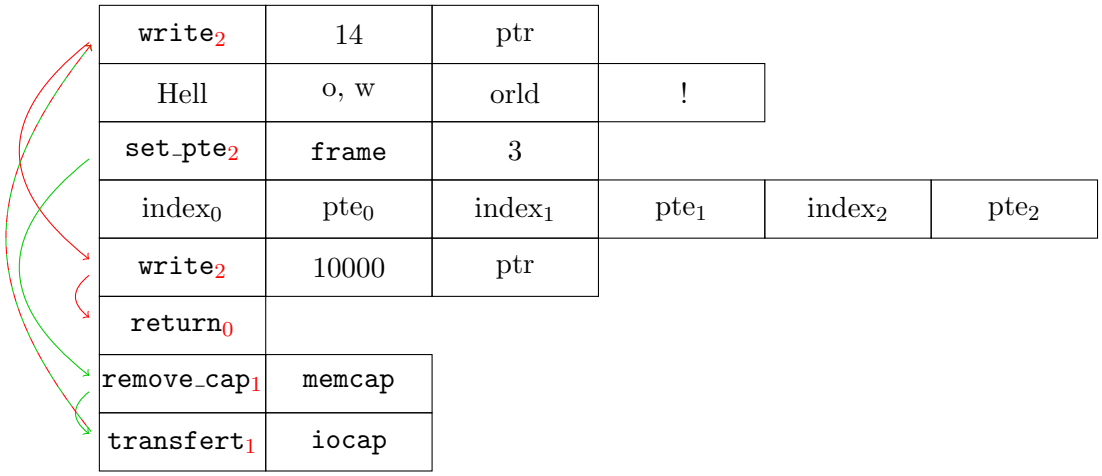



FIG. C.5 – Programme multicall pour exécuter un `writev`.

```
struct iovec iov[2] = {{"Hello, world !\n", 14}, {buffer, 10000}} ;
writev(fd, iov, 2) ;
```

Listing C.6 – Programme UNIX correspondant

La Figure C.5 donne la suite d'octets présents dans le programme multicall. Les flèches à gauche indiquent la prochaine instruction (champs `next`) ; l'exécution commence à la méthode `set_pte` dans le service mémoire, installe 3 entrées, retire la capacité mémoire, puis l'exécution est transférée au service d'entrée sortie, qui fait un appel à write avec les données simplement copiées en argument, et un avec les données transférées par les mappings mis en place par le service mémoire. Enfin, l'exécution termine par l'appel à return.

Le programme multicall est écrit au fur et à mesure des besoins du programme. À chaque appel à `write` nécessaire peut correspondre des mappings à installer, et les programmes pour les services IO et mémoires sont écris en parallèle. L'écriture de ces programmes est encore laborieuse malgré la présence de macros les facilitant, et bénéficierait de l'usage d'un IDL.

## C.4 CONCLUSION

Le service mémoire est un service central du noyau et de l'ensemble du système d'exploitation. C'est lui qui permet les garanties d'intégrité et de confidentialité du système ; son implémentation doit également garantir l'absence de déni de ressources,

et l'exécution en temps borné constant, y compris pour la récupération de mémoire. Enfin, il doit être prêt pour le passage au multiprocesseur.

Nous avons présenté dans ce chapitre une conception qui répond à tous ces problèmes, fruit d'une réflexion menée tout au long de cette thèse. Le système résultant a une interface simple, est sécurisé de manière prouvable (et la preuve est fournie dans ce chapitre), et peut être utilisé sur du multiprocesseur à mémoire partagée avec de très bonnes performances attendues.

Plusieurs choses restent cependant à faire. Il faudrait comparer ce service mémoire à d'autres systèmes similaires, comme par exemple à des hyperviseurs comme Xen. On peut également comparer l'ensemble « service mémoire, service politique et applications » à l'allocation de mémoire sur d'autres systèmes comme Linux.

L'implémentation sur multiprocesseur, même si elle est largement « préparée » et implémentée dans une large part, reste à être terminée et évaluée en terme de passage à l'échelle.

Enfin, nous n'avons pas abordé la question de la pagination. Il serait intéressant de montrer qu'on peut implémenter la pagination entièrement dans un service de politique, par exemple.

# Glossaire

**A**

**allocation** partage et affectation des ressources aux différentes tâches. 21
  **allocation déterministe** Propriété du système qui consiste à ce que l'allocation effective soit égale à l'ordonnancement prévue. 15, 17, 20, 28, 29, 37, 40, 52, 64, 73, 75, 78, *voir* politique (module de)

**appel de service** mécanisme pour l'envoi d'une requête à un service partagé. 53, 54, 78, 81–83, 103, 117, 128

**B**

**bloquant** Se dit d'un mécanisme pour lequel l'arrêt d'un thread peut indéfiniment empêcher l'exécution d'un autre. Anglais : blocking. 148, 149, 151, 160, *voir* non-bloquant

**C**

**canal caché** canal de communication non autorisé (anglais : covert channel). 43, 113, 198

**capacité** jeton d'autorisation utilisé par une tâche pour accéder à un objet (anglais : capability). 16

**C-list** liste des capacités possédées ; dans Anaxagoros se présente sous la forme d'un tableau. 93, 104, 105, 109, 112–114, 123, 125, 136

**cohérence (problème de)** nécessité de respecter des invariants pour un ensemble de variables à un moment donné. 145, 195

**concurrence (problèmes de)** Ensemble des problèmes liés au multithreading ; se décompose en problèmes de séquencement et de cohérence. 145, 195

**D**

**déni de ressource** Type de déni de service consistant à monopoliser un type de ressource afin d'empêcher un ou plusieurs programmes qui en dépendent de pouvoir s'exécuter.. 20, 23, 24, 39, 53, 58, 119, 123, 152, 153

**déni de service** Situation où un attaquant a réussi à empêcher l'exécution d'une autre tâche (anglais : denial of service, DoS).. 16, 19, 23, 36, 39, 40, 55, 72, 76, 80, 88, 117, 128

**déterminisme** Propriété d'un système pour lequel des entrées similaires produisent le même comportement. *voir* indterminisme

**domaine (de protection)** unité de séparation des différents privilèges. Les processus UNIX sont des exemples de domaine.. 51, 91–93, 99, 111

**donnée globale** donnée d'un service qui peut être utilisée ou modifiée par n'importe quel requête. 130, 201

**donnée locale** données dans un service accessible par un seul client, i.e. pile et arguments. 134

**E**

**espace d'adressage** Entité assurant la correspondance entre adresse virtuelle et physique, accompagnée de droits de lecture, d'écriture ou d'exécution sur ces adresses. 91–93

**exportation des données** principe consistant à minimiser les données stockés dans le service en les plaçant dans les clients. 196

**F**

**FCFS** Premier arrivé, premier servi. Politique d'allocation consistant à attribuer les ressources à la première tâche qui en fait la demande. Efficace (pas de ressource gâchée), mais fondamentalement insécurisée.. 23, 36

**fragmentation interne** Ressources perdues car inutilisées à l'intérieur d'un pool de ressource. 55

**framebuffer** Espace mémoire contenant la couleur de chaque pixel de l'écran. 137

**I**

**indéterminisme** Propriété d'un système qui n'est pas déterministe. *voir* dterminisme

**invariant** Propriété sur les données d'un service. 144, 175, 183, 212

**invariant de sécurité** Propriété que doit assurer le service pour remplir son rôle. 212

**IPI** Interruption inter-processeur, mécanisme permettant à un processeur de déclencher une interruption à un autre processeur. 159, 194, 195

**K**

**KDCB** Kernel thread control block, objet système stockant les informations relatives à un domaine.. 93, 258

**KTCB** Kernel thread control block, objet système stockant les informations relatives à un thread.. 93, 107, 258

**M**

**majorant d'exécution** Temps d'exécution maximum garantit pour un bout de code. 139, *voir* WCET

**masquage des interruptions** Mécanisme fourni par le processeur permettant de ne temporairement plus recevoir d'interruptions. Permet d'implémenter

des sections non préemptibles. 130, 139, 149, 159, 167, 171, 177, 178, *voir* WCET

**multicall** Mécanisme permettant d'exécuter d'un seul coup plusieurs appels de service successifs. 206

**N**

**non-bloquant** Le contraire de bloquant. Anglais : non-blocking. 148, 149, 151, 153, *voir* bloquant

**notification** mécanisme permettant de prévenir un autre domaine d'un évènement en un temps borné. 97, 118

**O**

**ordonnancement** affectation d'une ressource à une tâche en fonction du temps. 21

**P**

**politique** Choix que l'OS impose à tout le système. 65

**politique (module de)** Composant(s) logiciel seul responsable de l'allocation pour un type de ressource. 30, 31, 37, 76–79, *voir* politique d'allocation

**politique d'allocation** Politique régissant la quantité de ressource allouée à chaque tâche. 21, 24, 76, 77, *voir* allocation

**politique d'allocation déterministe** politique d'allocation qui permet de prédire la quantité de ressource allouées à une tâche non allocation deterministe, pas politique d'allocation deterministe. 56, 150

**préemption programmable** mécanisme permettant d'avertir le service qu'une préemption a un lieu. 144, 176, 177, 180, *voir* rollforward lock

**prêt de ressource** mécanisme permettant à un service d'exécuter des requêtes d'un client en se servant de ses ressources. 17, 62, 64, 73, 75, 78, 117, 129

**prêt de ressource semi-permanent** prêt où les ressources peuvent rester prêtées entre deux appels. 62, 119, 123, 125, 135, 136

**prêt de ressource transitoire** prêt de ressource limité à la durée de l'appel de service. 62, 119, 123, 128

**prêt de thread** mécanisme implémentant le prêt de ressources. 136, 139

**principal** détenteur de droits. 91, 92

**program counter** registre processeur indiquant la prochaine addresse d'exécution. 93, 117, 118, 134, 159, 163–165, 173, 177

**R**

**race condition** bug indéterministe, qui n'apparait que pour certaines séquences d'évènement. 105, 134, 162, 202, 246, 247, 273, *voir* indterminisme

**recouvrement** Dans cette thèse, action exécutée pour débloquer un verrou par ailleurs bloquant. 160–162

**ressource propre** ressources qui n'ont pas été prêtées par un client. 129

**S**

**section critique** Zone de code d'un programme dans laquelle les threads s'exécutent en exclusion mutuelle. 148–150

**section non préemptible** Zone de code d'un programme qui est garantie d'être exécutée jusqu'au bout, sans se faire préempter. 149–151, 157–159, 193, 194

**séquencement (problèmes de)** Nécessité de respecter un ordre pour faire certaines actions, souvent spécifié par un automate.. 145

**service** Domaine de protection séparé qui doit être contacté pour effectuer certaines actions. Il permet de protéger l'accès à une ressource. 16, 23, 53

  **service de politique** Service qui implémente une politique d'allocation. *voir* politique d'allocation

**stack pointer** registre indiquant le sommet de la pile. 164

**stateless** se dit d'un service qui exécute les différentes requêtes séparément, en isolation. 130, 181, 184, 196, 204

**T**

**table des pages** Tableau associant une adresse de page physique à une adresse virtuelle dans un espace d'adressage. 92, *voir* espace d'adressage

  **table des pages de premier niveau** Tableau racine d'une table des pages sur plusieurs niveaux. 92

**tableau des ressources** tableau regroupant des informations par ressource. 89, 132, 224

**TCB** Trusted computing base : le TCB d'une tâche est l'ensemble du code pour lequel une malfonction peut empêcher l'exécution normale de cette tâche. 26, 43

**thread** Fil d'exécution indépendant, exécutant un code séquentiel. Le thread est l'unité de dispatch du temps CPU : un ordonnanceur donne du temps CPU à un thread. L'ordonnancement peut impliquer le noyau (thread noyau) ou non (thread utilisateur) ; quand non précisé on parle de thread noyau. see. 52, 91, 93, 123, 138

  **thread noyau** Thread visible par le noyau, ordonnancé à l'aide de l'appel à la méthode `schedule` ; dans Anaxagoros, est aussi un principal. 163, 179, 258, *voir* principal

  **thread propre** Thread du service qui n'est pas été prêté par un client. 135, 137

  **thread utilisateur** Thread invisible pour le noyau, obtenu simplement en changeant de pile et de registres en espace utilisateur (anglais : user-level thread). 161, 163, 166, 169, 172, 177, 212

**U**

**upcall de reprise** Lorsque le thread est réordonnancé, le program counter repart de cette adresse.. 173

**UTCB** User thread control block, zone mémoire attachée à un thread accessible uniquement au domaine courant lorsque le thread correspondant est en train d'être exécuté.. 93, 121, 162–164, 259

**V**

**verrou** Variable utilisée pour implémenter une synchronisation (français : verrou). Quand on ne précise pas, un lock assure une exclusion mutuelle. 15, 26, 29, 30, 148, 162, 175

- **mutex lock** lock s'assurant qu'au plus un seul thread ne possède le lock. 148, 151, 185
- **recoverable lock** Verrou très général permettant d'implémenter les autres types de verrous non-bloquants.. 144, 172, 174, 175, 178, 180, 190, *voir* recouvrement
- **revocable lock** type de verrou dont les threads qui l'utilisent ne sont pas garanti de terminer leur section critique.. 154, 156, 172–176, 178, 183, 187, 203, 250
- **rollback lock** section critique non bloquante permettant de sérialiser des traitements. 144, 175, 187, 188, 190, 203
- **rollforward lock** section critique non bloquante permettant de sérialiser des traitements. 139, 144, 157, 166, 168, 169, 171, 173–175, 178, 179, 183, 187–190, 203
- **sleeplock** mécanisme de synchronisation qui peut arrêter de donner du temps CPU au thread qui l'exécute, i.e. modifier la décision d'ordonnancement de manière imprévisible. 17, 23, 29, 30, 36, 37, 139, 150, 161, 178, 209, 211
- **spinlock** mécanisme de synchronisation qui attend en boucle avant de rentrer dans une section critique. 139, 148–150, 158, 160, 167–169, 171, 178, 186, 270

## W

**WCET** Temps d'exécution minimal dans le pire cas, i.e plus petit majorant d'exécution. *voir* majorant d'excution

# Bibliographie


[AB86] James Archibald and Jean-Loup Baer. Cache coherence protocols : evaluation using a multiprocessor simulation model. *ACM Trans. Comput. Syst.*, 4(4) :273–298, 1986. pages 147

[ABG+86] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach : A new kernel foundation for unix development. Technical report, Carnegie Mellon University, August 1986. pages 45, 50, 55, 56, 68, 69, 83

[ABLL92] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations : effective kernel support for the user-level management of parallelism. *ACM Trans. Comput. Syst.*, 10(1) :53–79, 1992. pages 161, 164, 166, 176, 212

[AD94] Rajeev Alur and David Dill. A theory of timed automata. *Journal of theoretical computer science*, 126(2) :183–235, 1994. pages 33

[AD98] Christophe Aussaguès and Vincent David. A method and a technique to model and ensure timeliness in safety critical real-time systems. In *Fourth IEEE ICECCS'98*, pages 2–13, Los Alamitos, CA, USA, 1998. pages 14, 28, 33

[Aer] Aeronautical Radio, Incorporated. ARINC 653 standard : Avionics application software standard interface. pages 13, 28, 55, 65, 71

[AFHOT06] Jim Alves-Foss, W. Scott Harrison, Paul Oman, and Carol Taylor. The mils architecture for high-assurance embedded systems. *International journal of embedded systems ISSN 1741-1068*, 2(3-4) :239–247, 2006. pages 49, 84

[And03] Bjorn Andersson. *Static-priority scheduling on multiprocessors*. PhD thesis, Department of Computer Engineering, Chalmers University of Technology, 2003. pages 27, 31

[Bak91] T. P. Baker. Stack-based scheduling for realtime processes. *Real-Time Syst.*, 3(1) :67–99, 1991. pages 26, 29

[BALL89] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. In *SOSP '89 : Proceedings of the twelfth ACM symposium on Operating systems principles*, pages 102–113, New York, NY, USA, 1989. ACM Press. pages 53, 59

[BBLB03] Scott A. Brandt, Scott Banachowski, Caixue Lin, and Timothy Bisson. Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In *RTSS '03 : Proceedings of the 24th IEEE International Real-Time Systems Symposium*, page 396, Washington, DC, USA, 2003. IEEE Computer Society. pages 27, 29, 35

[BC05] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel - 3rd edition*. O'reilly, 2005. pages 22, 23, 36, 45, 67, 108, 132, 153, 158, 164

[BCGM99] Sanjoy Baruah, Deji Chen, Sergey Gorinsky, and Aloysius Mok. Generalized multiframe tasks. *Real-Time Systems*, 17 :5–22, 1999. pages 33

[BCPV96] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress : a notion of fairness in resource allocation. *Algorithmica*, 6 :600–625, 1996. pages 31

[BD96] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing systems*, 2(2) :131–152, 1996. pages 41, 146

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of ACM SOSP '03 :*, pages 164–177. ACM Press, 2003. pages 43, 46, 55, 64, 68, 70, 72, 77, 98, 126, 256, 260, 263, 269, 280, 283, 286, 293

[BDM99] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers : A new facility for resource management in server systems. In *Proceedings of OSDI '99*, pages 45–58. USENIX, 1999. pages 52, 63, 70

[BDRS08] Carolyn Boettcher, Rance DeLong, John Rushby, and Wilmar Sifre. The mils component integration approach to secure information sharing. In *27th IEEE/AIAA Digital Avionics Systems Conference (DASC), St Paul MN*, October 2008. pages 15, 40, 48, 49, 55, 84, 87, 97, 102

[BDV03] Alan Burns, Brian Dobbing, and Tullio Vardanega. Guide for the use of the ada ravenscar profile in high integrity systems. Technical Report YCS-2003-348, University of York, January 2003. pages 13

[Ber93] B.N. Bershad. Practical considerations for non-blocking concurrent objects. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 264–273, May 1993. pages 161, 167, 172, 178, 189

[Ber07] Daniel J. Bernstein. Some thoughts on security after ten years of qmail 1.0. In *CSAW '07 : Proceedings of the 2007 ACM workshop on Computer security architecture*, pages 1–10, New York, NY, USA, 2007. ACM. pages 38, 42, 43, 183, 198, 218, 290

[BF08] Sanjoy Baruah and Nathan Fisher. Hybrid-priority scheduling of resource-sharing sporadic task systems. *Real-Time and Embedded Technology and Applications Symposium, IEEE*, 0 :248–257, 2008. pages 26, 27

[BFF+92] Alan C. Bomberger, A. Peri Frantz, William S. Frantz, Ann C. Hardy, Norman R. Hardy, Charles Landau, and Jonathan Shapiro. The keykos nanokernel architecture. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 95–112, April 1992. pages 25, 51, 62, 69, 99

[BRE92] Brian N. Bershad, David D. Redell, and John R. Ellis. Fast mutual exclusion for uniprocessors. In *ASPLOS-V : Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pages 223–233, New York, NY, USA, 1992. ACM. pages 173, 177

[BRGH89] D. L. Black, R. F. Rashid, D. B. Golub, and C. R. Hill. Translation lookaside buffer consistency : a software approach. *SIGARCH Comput. Archit. News*, 17(2) :113–122, 1989. pages 286

[BSP+95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *15th Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Colorado, 1995. pages 46, 67, 72, 84

[But05] Giorgio C. Buttazzo. Rate monotonic vs. edf : Judgment day. *Real-Time Systems*, 29(1) :5–26, January 2005. pages 27, 33

[Can96] Bryan M. Cantrill. Runtime performance analysis of the m-to-n scheduling model. Technical report, Brown University, Providence, RI, USA, 1996. pages 177

[CD94] David R. Cheriton and Kenneth J. Duda. A caching model of operating system kernel functionality. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI)*, pages 179–193. USENIX Association, November 1994. pages 25, 92

[CDA+05] D. Chabrol, V. David, C. Aussaguès, S. Louise, and F. Daumas. Deterministic distributed safety-critical real-time systems within the OASIS approach. In *17th IASTED PDCS'05*, November 2005. pages 33, 55, 56, 212

[Che84] David R. Cheriton. An experiment using registers for fast message-based interprocess communication. *SIGOPS Oper. Syst. Rev.*, 18(4) :12–20, 1984. pages 53

[CMDD62] F.J. Corbato, M. Merwin-Daggett, and R.C. Daley. An experimental time-sharing system. In *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*, pages 335–344. ACM New York, NY, USA, 1962. pages 78

[CV65] F. J. Corbató and V. A. Vyssotsky. Introduction and overview of the Multics system. In *AFIPS '65 (Fall, part I) : Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*, pages 185–196, New York, NY, USA, 1965. ACM. pages 44

[DAL+04] V. David, C. Aussaguès, S. Louise, Ph. Hilsenkopf, B. Ortolo, and C. Hessler. The OASIS based qualified display system. In *Proceedings ANS (NPIC&HMIT 2004)*, 2004. pages 33, 65

[Den76] Peter J. Denning. Fault tolerant operating systems. *ACM Computing Survey*, 8(4) :359–389, 1976. pages 38, 39, 40, 41, 42, 47, 50, 81, 82, 102, 183

[Der74] Michael L. Dertouzos. Control robotics : The procedural control of physical processes. In *IFIP Congress*, pages 807–813, 1974. pages 27

[DH66] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. Technical Report MIT/LCS/TR-23, Massachusetts Institute of Technology, 1966. pages 21, 50, 100

[Dij68] Edsger W. Dijkstra. The structure of the "the"-multiprogramming system. *Commun. ACM*, 11(5) :341–346, 1968. pages 151, 158

[DM89] Michael. L. Dertouzos and Aloysius K. Mok. Multiprocessor online scheduling of hard-real-time tasks. *IEEE Transactions on Software Engineering*, 15(12) :1497–1506, 1989. pages 34, 37

[DM96] L. L. Peterson D. Mosberger, P. Druschel. Implementing atomic sequences on uniprocessors using rollforward. *Software : Practice and Experience*, 26(1) :1–23, 1996. pages 178

[ECCZ05] Khaled Elmeleegy, Anupam Chanda, Alan L. Cox, and Willy Zwaenepoel. A portable kernel abstraction for low-overhead ephemeral mapping management. In *ATEC '05 : Proceedings of the USENIX Annual Technical Conference*, pages 28–28, Berkeley, CA, USA, 2005. USENIX Association. pages 108, 269, 291

[EDE08] Dhammika Elkaduwe, Philip Derrin, and Kevin Elphinstone. Kernel design for isolation and assurance of physical memory. In *1st Workshop on Isolation and Integration in Embedded Systems (IIES'08), Glasgow, UK*, April 2008. pages 25, 62, 63, 64, 92, 104, 106, 115, 116, 126, 223, 256, 279

[EK96] Dawson R. Engler and M. Frans Kaashoek. Dpf : fast, flexible message demultiplexing using dynamic code generation. *SIGCOMM Comput. Commun. Rev.*, 26(4) :53–59, 1996. pages 68

[EKJO95] D. R. Engler, M. F. Kaashoek, and Jr. J. O'Toole. Exokernel : an operating system architecture for application-level resource management. In *Proceedings of SOSP '95*, pages 251–266. ACM Press, 1995. pages 15, 43, 46, 59, 65, 67, 68, 70, 77, 82, 89, 134, 149, 161, 177, 198, 200, 204

[Elp] Kevin Elphinstone. Future directions in the evolution of the l4 microkernel. pages 25, 58

[ELS88] Jan Edler, Jim Lipkis, and Edith Schonberg. Process management for highly parallel unix systems. In *USENIX Workshop on Unix and Supercomputers*, pages 1–17, 1988. Symunix. pages 52, 149, 177

[Eng95] Dawson R. Engler. The design and implementation of a prototype exokernel system. Master's thesis, Massachussets Institute of Technology, 1995. pages 101, 102

[FH05] Norman Feske and Christian Helmuth. A Nitpicker's guide to a minimal-complexity secure GUI. In *ACSAC '05 : Proceedings of the 21st Annual Computer Security Applications Conference*, pages 85–94, Washington, DC, USA, 2005. IEEE Computer Society. pages 39, 217

[FH06] Norman Feske and Christian Helmuth. Design of the bastei os architecture. Technical Report TUD-FI06-07, Technische Universität Dresden, December 2006. pages 62, 70, 84, 128

[FHL+96] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *In Proceedings of the Second Symposium on Operating Systems Design and Implementation*, 1996. pages 70, 96

[FHL+99] Bryan Ford, Mike Hibler, Jay Lepreau, Roland McGrath, and Patrick Tullmann. Interface and execution models in the fluke kernel. In *OSDI '99 : Proceedings of the third symposium on Operating systems design and implementation*, pages 101–115, Berkeley, CA, USA, 1999. USENIX Association. pages 61, 150, 159, 207, 256, 289

[FL94] Bryan Ford and Jay Lepreau. Evolving Mach 3.0 to a migrating thread model. In *Usenix Winter Conference*, pages 97–114, 1994. pages 45, 53, 55, 59, 64, 203

[FR82] M. P. Fle and G. Roucairol. On serializability of iterated transactions. In *PODC '82 : Proceedings of the first ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 194–200, New York, NY, USA, 1982. ACM. pages 146, 200

[FR85] Marie-Paule Flé and Gérard Roucairol. A language theoretic approach to serialization problem in concurrent systems. In *FCT '85 : Fundamentals of Computation Theory*, pages 128–145, London, UK, 1985. Springer-Verlag. pages 146, 200

[Fra04] Keir Fraser. Practical lock-freedom. Technical report, University of Cambridge, February 2004. pages 43, 148, 152, 153, 274

[FRK02] H. Franke, R. Russell, and M. Kirkwood. Fuss, futexes and furwocks : Fast userlevel locking in Linux. In *Ottawa Linux Symposium*, pages 479–489, 2002. pages 158, 178

[FS96] Bryan Ford and Sai Susarla. CPU Inheritance Scheduling. In *Usenix OSDI'96*, pages 91–105, 1996. pages 57, 72, 91, 96, 97, 150, 178, 179, 282

[Gab91] Richard P. Gabriel. The rise of worse is better, 1991. `www.jwz.org/doc/worse-is-better.html`. pages 184

[Gas88] Morrie Gasser. *Building a Secure Computer System*. Van Nostrand Reinhold, May 1988. pages 16, 39, 44, 47, 48

[GC96] Michael Greenwald and David R. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Operating Systems Design and Implementation*, pages 123–136, 1996. pages 30, 133, 153, 157, 175, 176, 179, 192, 199, 274

[GGV96] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI)*, pages 107–121, 1996. pages 28, 72

[GSB+99] Eran Gabber, Christopher Small, John Bruno, José Brustoloni, and Avi Silberschatz. The Pebble component-based operating system. In *Proceedings of the 1999 USENIX Technical Conference*, pages 267–282, June 1999. pages 59, 61, 63

[HAF+07] Galen Hunt, Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, James Larus, Steven Levi, Bjarne Steensgaard, David Tarditi, and Ted Wobber. Sealing os processes to improve dependability and safety. *SIGOPS Oper. Syst. Rev.*, 41(3) :341–354, 2007. pages 46, 47

[Han70] Per Brinch Hansen. The nucleus of a multiprogramming system. *Commun. ACM*, 13(4) :238–241, 1970. pages 57, 69, 70, 96

[Han99] Steven M. Hand. Self-paging in the nemesis operating system. In *Operating Systems Design and Implementation*, pages 73–86, 1999. pages 134

[Har85] Norman Hardy. The KeyKOS architecture. *ACM SIGOPS Operating Systems Review*, September 1985. pages 50, 51, 83

[Har88] Norm Hardy. The confused deputy : (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22(4) :36–38, 1988. pages 102

[Har01] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. *Lecture Notes in Computer Science*, 2180 :300–? ?, 2001. pages 153

[HDF+07] P.E. Hladik, A.M. Deplanche, S. Faucou, Y. Trinquet, and N. LINA. Adequacy between AUTOSAR OS specification and real-time scheduling theory. In *Industrial Embedded Systems, 2007. SIES'07. International Symposium on*, pages 225–233, 2007. pages 27, 28

[HE03] Andreas Haeberlen and Kevin Elphinstone. User-level management of kernel memory. In *Proceedings of the 8th Asia-Pacific Computer Systems Architecture Conference*, Aizu-Wakamatsu City, Japan, September 24–26 2003. pages 25

[Her90] M. Herlihy. A methodology for implementing highly concurrent data structures. In *PPOPP '90 : Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 197–206, New York, NY, USA, 1990. ACM. pages 153, 172, 182, 232, 236

[Her93] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5) :745–770, November 1993. pages 153

[HF05] Tim Harris and Keir Fraser. Revocable locks for non-blocking programming. In *PPoPP '05 : Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 72–82, New York, NY, USA, 2005. ACM. pages 147, 154, 161, 173, 174, 175, 176, 178

[HHL+97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of $\mu$-kernel-based systems. In *SOSP '97 : Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 66–77, New York, NY, USA, 1997. ACM Press. pages 23, 45, 60, 68, 98, 198

[Hil92] Dan Hildebrand. An architectural overview of qnx. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126, Berkeley, CA, USA, 1992. USENIX Association. pages 57

[HK93] Graham Hamilton and Panos Kougiouris. The Spring nucleus : A microkernel for objects. Technical Report TR-93-14, Sun Microsystems Laboratories, Inc, April 1993. pages 58, 59, 60

[hKW00] Poul henning Kamp and Robert N. M. Watson. Jails : Confining the omnipotent root. In *In Proc. 2nd Intl. SANE Conference*, 2000. pages 199

[HL92] K.S. Hong and J.Y.-T. Leung. On-line scheduling of real-time tasks. *IEEE Transactions on Computers*, 41(10) :1326–1331, 1992. pages 37

[HLM03] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization : Double-ended queues as an example. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems*, May 2003. pages 176

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory : architectural support for lock-free data structures. In *ISCA '93 : Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM. pages 208

[HPHS04] Michael Hohmuth, Michael Peter, Hermann Härtig, and Jonathan S. Shapiro. Reducing TCB size by using untrusted components : small kernels versus virtual-machine monitors. In *EW11 : Proceedings of the 11th workshop on ACM SIGOPS European workshop*, page 22, New York, NY, USA, 2004. ACM. pages 43, 84

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability : a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3) :463–492, 1990. pages 146, 232, 234, 242, 274

[Int09] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3 : System Programming Guide*, 2009. pages 59, 148, 163, 165, 260

[JLDJB95] M.B. Jones, P.J. Leach, R.P. Draves, and III J.S. Barrera. Modular real-time resource management in the rialto operating system. *hotos*, 00 :12, 1995. pages 52, 63

[JRR97] Michael B. Jones, Daniela Rosu, and Marcel-Catalin Rosu. CPU reservations and time constraints : efficient, predictable scheduling of independent activities. *SIGOPS Oper. Syst. Rev.*, 31(5) :198–211, 1997. pages 26, 29

[KDK+89] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems : the mars approach. *Micro, IEEE*, 9(1) :25–40, Feb 1989. pages 28

[KEG+97] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceno, Russell Hunt, David Mazieres, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *SOSP '97 : Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages

52–65, New York, NY, USA, 1997. ACM Press. pages 46, 69, 70, 77, 87, 177, 198, 204

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4 : Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, USA, Oct 2009. ACM. pages 45, 47, 102, 223, 240, 260

[Key88] KeyLogic. Introduction to keykos concepts. Technical Report KL004-07, Key Logic Inc, 1988. `http://www.cis.upenn.edu/~KeyKOS/agorics/KeyKos/Concepts/welcome.html`. pages 20, 24, 53

[Key89] KeyLogic. Keysafe. Technical Report SEC009-01, Key Logic Inc, 1989. `http://www.cis.upenn.edu/~KeyKOS/agorics/KeyKos/keysafe/Keysafe.html`. pages 50, 102

[KW07] Robert Kaiser and Stephan Wagner. Evolution of the PikeOS microkernel. In *MIKES 2007 : 1st International Workshop on Microkernels for Embedded Systems*, pages 50 – 58. NICTA, 2007. pages 45, 56, 58, 71

[Lam71] Butler Lampson. Protection. *ACM Operating System Review*, 1 :18–24, January 1971. pages 41, 99, 100, 101, 223

[Lam73] Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10) :613–615, 1973. pages 240

[Lam74] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8) :453–455, August 1974. pages 149

[Lam77] Leslie Lamport. On concurrent reading and writing. *Communications of the ACM*, 20(11) :806–811, November 1977. pages 151, 232, 234

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9) :690–691, September 1979. pages 148

[Lam90] Leslie Lamport. The concurrent reading and writing of clocks. *ACM Transactions on Computer systems*, 8(4) :305–310, November 1990. pages 147, 153

[LCC+75] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in Hydra. In *Proceedings of SOSP '75*, pages 132–140, New York, NY, USA, 1975. ACM. pages 69

[LDAVN06] Matthieu Lemerre, Vincent David, Christophe Aussaguès, and Guy Vidal-Naquet. A new representation for the scheduling problem and

its applications. In *Proceedings WiP IEEE RTSS'06*, pages 89–92, December 2006. pages 15, 31, 33, 37

[LDAVN08] Matthieu Lemerre, Vincent David, Christophe Aussaguès, and Guy Vidal-Naquet. Equivalence between schedule representations : Theory and applications. *RTAS*, 0 :237–247, 2008. pages 15, 31, 33, 37

[Lem06] Matthieu Lemerre. Extension d'un système temps-réel sur multiprocesseur. Master's thesis, CEA LIST/Supélec/Université Paris XI, 2006. pages 15

[Lev84] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984. pages 25, 44, 50, 51, 95, 104, 105, 115, 116, 125, 203, 280

[LHH97] Jochen Liedtke, Hermann Härtig, and Michael Hohmuth. Os-controlled cache predictability. In *Proceedings of the 3rs IEEE Real-time Technology and Applications Symposium (RTAS)*, Montreal, Canada, June 1997. pages 81, 220

[Lie93] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of SOSP'93*, Asheville, NC, December 1993. pages 45, 53, 56, 57, 60, 117, 128

[Lie95a] J. Liedtke. On micro-kernel construction. In *SOSP '95 : Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 237–250, New York, NY, USA, 1995. ACM Press. pages 45, 57, 65, 68, 108, 128

[Lie95b] Jochen Liedtke. Improved address-space switching on Pentium processors by transparently multiplexing user address spaces. Arbeitspapiere der GMD No. 933, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1995. pages 99, 109

[LIJ97] Jochen Liedtke, Nayeem Islam, and Trent Jaeger. Preventing denial-of-service attacks on a microkernel for weboses. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, Cape Cod, MA, May 5–6 1997. pages 25, 40, 53, 58, 65

[Lin76] Theodore A. Linden. Operating system structures to support security and reliable software. *ACM Comput. Surv.*, 8(4) :409–445, 1976. pages 39, 50, 102, 112, 125

[Lip82] M. Lipow. Number of faults per line of code. *Software Engineering, IEEE Transactions on*, SE-8(4) :437–439, July 1982. pages 42

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20 :46–61, 1973. pages 27, 34

[LMR00] S. Loureiro, R. Molva, and Y. Roudier. Mobile code security. *Proceedings of ISYPAR*, 2000. pages 46

[Loe92] Keith Loepere, editor. *Mach 3 Kernel Principles*. Open Software Foundation and Carnegie Mellon University, July 1992. pages 62, 119, 136, 203, 260

[LS76] Butler W. Lampson and Howard E. Sturgis. Reflections on an operating system design. *Commun. ACM*, 19(5) :251–265, 1976. pages 24, 25, 51, 56, 115, 116

[Mas92] Henry Massalin. *Synthesis : An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University, 1992. pages 30, 68, 153, 179, 274

[MP96] David Mosberger and Larry L. Peterson. Making paths explicit in the scout operating system. In *Proceeding of the USENIX 2nd Symposium on OS Design and Implementation (OSDI '96)*, pages 153–167, 1996. pages 52, 59, 91

[MR97] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3) :217–252, 1997. pages 22, 98

[MS03] M. Miller and J. Shapiro. Paradigm regained : Abstraction mechanism for access control, 2003. pages 42, 50, 51, 69, 84, 102

[MSLM91] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. In *Proceedings of the13th ACM Symposium on Operating Systems Principle*, pages 110–121, Pacific Grove, CA, 1991. Psyche. pages 149, 161, 164, 165, 177, 212

[MST93] Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda. Processor capacity reserves for multimedia operating systems. Technical Report CS-93-157, Carnegie Mellon University, 1993. pages 23, 26, 28, 64

[MYS03] Mark S. Miller, Ka-Ping Yee, and Jonathan Shapiro. Capability myths demolished. Technical Report SRL2003-02, Systems Research Laboratory, Department of Computer Science, Johns Hopkins University, March 2003. pages 50, 100, 101, 102, 105, 114, 125

[Neu04] Peter G. Neumann. Principled assuredly trustworthy composable architectures. Technical report, SRI International, December 2004. pages 43, 76

[NW77] R. M. Needham and R. D.H. Walker. The cambridge cap computer and its protection system. In *SOSP '77 : Proceedings of the sixth ACM symposium on Operating systems principles*, pages 1–10, New York, NY, USA, 1977. ACM Press. pages 51, 70, 279

[Ope] Open Group. Single unix specification, version 3. pages 24, 65

[Orm07] T. Ormandy. An empirical study into the security exposure to host of hostile virtualized environments. *CanSecWest 2007*, 2007. pages 46

[Pet81] G.L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3) :115–116, 1981. pages 149

[RAA+91] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the chorus distributed operating systems. Technical Report CS/TR-90-25.1, Chorus systèmes, 1991. pages 45, 56

[RCK+09] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. Automatic device driver synthesis with termite. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)*, Big Sky, MT, USA, October 2009. pages 145, 191, 219

[RCKH09] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo : Taming device drivers. In *Proceedings of the 4th EuroSys Conference*, Nuremberg, Germany, Apr 2009. pages 145, 182, 183, 185

[Red74] D. Redell. *Naming and protection in extendable operating systems*. PhD thesis, MIT, 1974. pages 50, 52, 100, 105, 125

[Rip03] Christophe Rippert. *Protection dans les architectures de systèmes flexibles*. PhD thesis, Université Joseph-Fourier - Grenoble I, 2003. pages 46, 47, 65, 67, 72, 290

[RJMO98] Raj Rajkumar, Kanaka Juvva, Anastasio Molano, and Shuichi Oikawa. Resource kernels : a resource-centric approach to real-time and multimedia systems. In *SPIE/ACM Conference on Multimedia Computing and Networking*, 1998. pages 29

[Ros95] Timothy Roscoe. *The Structure of a Multi-Service Operating System*. PhD thesis, University of Cambridge, April 1995. pages 26, 36, 46, 54, 56, 58, 60, 70, 72, 87, 200, 204, 212

[RR83] John Rushby and Brian Randell. A distributed secure system. *IEEE Computer*, 16(7) :55–67, July 1983. pages 48

[RT74] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Commun. ACM*, 17(7) :365–375, 1974. pages 44

[Rus81] John Rushby. The design and verification of secure systems. In *Eighth ACM Symposium on Operating System Principles (SOSP)*, pages 12–21, Asilomar, CA, December 1981. (ACM *Operating Systems Review*, Vol. 15, No. 5). pages 48, 55, 69, 84

[Rus84] John Rushby. A trusted computing base for embedded systems. In *7th DoD/NBS Computer Security Initiative Conference*, volume 10, pages 294–311, Gaithersburg, MD, September 1984. pages 42, 44, 47, 48, 87, 200

[Rus89] John Rushby. Kernels for safety ? In T. Anderson, editor, *Safe and Secure Computing Systems*, pages 210–220. Blackwell Scientific Publications, 1989. (Proceedings of a Symposium held in Glasgow, October 1986). pages 16, 40, 42, 53, 86, 144, 182

[Rus92] John Rushby. Noninterference, transitivity, and channel-control security policies. Technical report, SRI International, dec 1992. pages 48, 83

[Rus98] J. Rushby. Partitioning in avionics architectures : Requirements, 1998. pages 12, 15, 28, 38, 39, 55, 71, 88, 94, 97

[SA02] Jonathan S. Shapiro and Jonathan Adams. Design evolution of the eros single-level store. In *ATEC '02 : Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 59–72, Berkeley, CA, USA, 2002. USENIX Association. pages 63

[SDD+04] Jonathan Shapiro, Ph. D, Michael Scott Doerrie, Eric Northup, and Mark Miller. Towards a verified, general-purpose operating system kernel. In *NICTA Formal Methods Program Workshop on Operating Systems Verification*, pages 1–19, 2004. pages 45

[SESS96] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster : surviving misbehaved kernel extensions. *SIGOPS Oper. Syst. Rev.*, 30(SI) :213–227, 1996. pages 67

[SESS97] M. Seltzer, Y. Endo, C. Small, and K. A. Smith. Issues in extensible operating systems. Technical Report Computer Science Technical Report TR-18-97, Harvard University, November 1997. pages 65, 66, 67

[SFS96] J. S. Shapiro, D. J. Farber, and J. M. Smith. The measured performance of a fast local IPC. In *IWOOOS '96 : Proceedings of the 5th International Workshop on Object Orientation in Operating Systems (IWOOOS '96)*, page 89, Washington, DC, USA, 1996. IEEE Computer Society. pages 53, 57, 62, 119, 123, 128, 294

[SH02] Jonathan S. Shapiro and Norman Hardy. Eros : A principle-driven operating system from the ground up. *IEEE Software*, pages 26–33, January/February 2002. pages 25, 52, 76

[Sha99] Jonathan Shapiro. *EROS : A capability system.* PhD thesis, University of Pennsylvania, 1999. pages 25, 44, 45, 50, 51, 102, 106, 115, 116, 118

[Sha03] Jonathan S. Shapiro. Vulnerabilities in synchronous IPC designs. In *IEEE Symposium on Security and Privacy, Oakland, CA*, 2003. pages 53, 55, 57, 58, 84, 108, 118

[SP99] Oliver Spatscheck and Larry L. Petersen. Defending against denial of service attacks in Scout. In *Proceedings of USENIX OSDI'99*, New Orleans, Louisiana, February 1999. pages 53, 81

[SR89] J. A. Stankovic and K. Ramamritham. The spring kernel : a new paradigm for real-time operating systems. *SIGOPS Oper. Syst. Rev.*, 23(3) :54–71, 1989. pages 33

[SR04] John A. Stankovic and R. Rajkumar. Real-time operating systems. *Real-Time Syst.*, 28(2-3) :237–253, 2004. pages 13, 26

[SRL90] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols : An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9) :1175–1185, 1990. pages 26

[SS72] Michael D. Schroeder and Jerome H. Saltzer. A hardware architecture for implementing protection rings. *Commun. ACM*, 15(3) :157–170, 1972. pages 59

[SS74] Saltzer and Schroeder. The protection of information in computer systems. *Communication of the ACM*, 7, 1974. pages 16, 41, 50, 100, 215

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. Eros : a fast capability system. In *ACM Symposium on Operating Systems Principles (SOSP'99)*, volume 34, pages 170–185, December 1999. pages 45, 47, 50, 51, 62, 92, 96, 98, 106, 115, 279, 280

[SSS04] Anshumal Sinha, Sandeep Sarat, and Jonathan S. Shapiro. Network subsystems reloaded : a high-performance, defensible network subsystem. In *Proceedings of the USENIX Annual Technical Conference 2004*, pages 19–19. USENIX Association, 2004. pages 62, 63, 65, 119, 126, 128, 217

[STT+09] Erik Schierboom, Alejandro Tamalet, Hendrik Tews, Marko van Eekelen, and Sjaak Smetsers. Preemption abstraction : A lightweight approach to modelling concurrency. In *LNCS / Proceeding of FMICS 2009*. Springer Berlin, 2009. pages 289

[SVNC04] Jonathan S. Shapiro, John Vanderburgh, Eric Northup, and David Chizmadia. Design of the eros trusted window system. In *SSYM'04 : Proceedings of the 13th conference on USENIX Security Symposium*, pages 165–178, Berkeley, CA, USA, 2004. USENIX Association. pages 39, 134

[SWH05] Udo Steinberg, Jean Wolter, and Hermann Hartig. Fast component interaction for real-time systems. In *Proceedings of ECRTS'05*, pages 89–97, July 2005. pages 23, 57, 64, 150, 179

[Tan01] Andrew S. Tanenbaum. *Modern Operating Systems (Second Edition)*. Prentice Hall, 2001. pages 44

[TEF07] Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. Secretly monopolizing the cpu without superuser privileges. In *USENIX Security Symposium*, August 2007. pages 32

[Tho78] K. Thompson. UNIX implementation. *The Bell System Technical Journal*, 57(6 (part 2)) :1931–1940, 1978. pages 21, 44, 59

[VH96] Alistair C. Veitch and Norman C. Hutchinson. Kea - a dynamically extensible and configurable operating system kernel. *iccds*, 00 :236, 1996. pages 59, 60

[WB07] Neal H. Walfield and Marcus Brinkmann. A critique of the GNU Hurd multi-server operating system. *SIGOPS Oper. Syst. Rev.*, 41(4) :30–39, 2007. pages 45

[WCC+74] William A. Wulf, Ellis S. Cohen, William M. Corwin, Anita K. Jones, Roy Levin, C. Pierson, and Fred J. Pollack. Hydra : The kernel of a multiprocessor operating system. *Commun. ACM*, 17(6) :337–345, 1974. pages 50, 51, 59, 69, 71, 84, 95, 112

[WN79] M. V. Wilkes and R. M. Needham. *The Campridge CAP Computer and Its Operating System*. Elsevier North Holland, Inc, 1979. pages 51, 59

[WS73] W. Wulf and Mary Shaw. Global variable considered harmful. *SIGPLAN Not.*, 8(2) :28–34, 1973. pages 198

[WSG02] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the denali isolation kernel. *SIGOPS Oper. Syst. Rev.*, 36(SI) :195–209, 2002. pages 46, 68

[WTUH03] Adam Wiggins, Harvey Tuch, Volkmar Uhlig, and Gernot Heiser. Implementation of fast address-space switching and TLB sharing on the StrongARM processor. In *Proceedings of the 8th Asia-Pacific Computer Systems Architecture Conference*, Aizu-Wakamatsu City, Japan, Sep 2003. Springer Verlag. pages 99, 122