Projet SOCRATE - (8) Structure d'un système d'exploitation adapté à la base de données

Guy Mazaré

▶ **To cite this version:**

Guy Mazaré. Projet SOCRATE - (8) Structure d'un système d'exploitation adapté à la base de données. Réseaux et télécommunications [cs.NI]. Université Joseph-Fourier - Grenoble I, 1973. Français. NNT : . tel-00010505

HAL Id: tel-00010505
https://theses.hal.science/tel-00010505

Submitted on 11 Oct 2005

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

N° d'ordre

TU 561

THESE

présentée à

L'UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

pour obtenir

LE GRADE DE DOCTEUR INGENIEUR

par

Guy MAZARÉ
Ingénieur E.C.P.

PROJET SOCRATE

(8) Structure d'un Système d'exploitation adapté à la Banque de Donnée

Thèse soutenue le 18 Mai 1973 devant la commission d'examen

Monsieur	L. BOLLIET	Président
Monsieur	J.R. ABRIAL	Rapporteur
Monsieur	L. POUZIN	Examinateurs
Monsieur	G. VEILLON	

UNIVERSITE SCIENTIFIQUE
ET MEDICALE DE GRENOBLE

LISTE DES PROFESSEURS

Président : Monsieur Michel SOUTIF
Vice-Président : Monsieur Gabriel CAU

PROFESSEURS TITULAIRES

MM.	ANGLES D'AURIAC Paul	Mécanique des fluides
	ARNAUD Georges	Clinique des maladies infectieuses
	ARNAUD Paul	Chimie
	AUBERT Guy	Physique
	AYANT Yves	Physique approfondie
Mme	BARBIER Marie-Jeanne	Electrochimie
MM.	BARBIER Jean-Claude	Physique expérimentale
	BARBIER Reynold	Géologie appliquée
	BARJON Robert	Physique nucléaire
	BARNOUD Fernand	Biosynthèse de la cellulose
	BARRA Jean-René	Statistiques
	BARRIE Joseph	Clinique chirurgicale
	BENOIT Jean	Radioélectricité
	BERNARD Alain	Mathématiques Pures
	BESSON Jean	Electrochimie
	BEZES Henri	Chirurgie générale
	BLAMBERT Maurice	Mathématiques Pures
	BOLLIET Louis	Informatique (IUT B)
	BONNET Georges	Electrotechnique
	BONNET Jean-Louis	Clinique ophtalmologique
	BONNET-EYMARD Joseph	Pathologie médicale
	BONNIER Etienne	Electrochimie Electrométallurgie
	BOUCHERLE André	Chimie et Toxicologie
	BOUCHEZ Robert	Physique nucléaire
	BOUSSARD Jean-Claude	Mathématiques Appliquées
	BRAVARD Yves	Géographie
	BRISSONNEAU Pierre	Physique du solide
	BUYLE-BODIN Maurice	Electronique
	CABANAC Jean	Pathologie chirurgicale
	CABANEL Jean	Clinique rhumatologique et hydrologie
	CALAS François	Anatomie
	CARRAZ Gilbert	Biologie animale et pharmacodynamie
	CAU Gabriel	Médecine légale et Toxicologie
	CAUQUIS Georges	Chimie organique
	CHABAUTY Claude	Mathématiques Pures
	CHARACHON Robert	Oto-Rhino-Laryngologie
	CHATEAU Robert	Thérapeutique
	CHENE Marcel	Chimie papetière
	COEUR André	Pharmacie chimique
	CONTAMIN Robert	Clinique gynécologique
	COUDERC Pierre	Anatomie Pathologique
	CRAYA Antoine	Mécanique

Mme	DEBELMAS Anne-Marie	Matière médicale
MM.	DEBELMAS Jacques	Géologie générale
	DEGRANGE Charles	Zoologie
	DESRE Pierre	Métallurgie
	DESSAUX Georges	Physiologie animale
	DODU Jacques	Mécanique appliquée
	DOLIQUE Jean-Michel	Physique des plasmas
	DREYFUS Bernard	Thermodynamique
	DUCROS Pierre	Cristallographie
	DUGOIS Pierre	Clinique de Dermatologie et Syphiligraphie
	FAU René	Clinique neuro-psychiatrique
	FELICI Noël	Electrostatique
	GAGNAIRE Didier	Chimie physique
	GALLISSOT François	Mathématiques Pures
	GALVANI Octave	Mathématiques Pures
	GASTINEL Noël	Analyse numérique
	GEINDRE Michel	Electroradiologie
	GERBER Robert	Mathématiques Pures
	GIRAUD Pierre	Géologie
	KLEIN Joseph	Mathématiques Pures
Mme	KOFLER Lucie	Botanique et Physilogie végétale
MM.	KOSZUL Jean-Louis	Mathématiques Pures
	KRAVTCHENKO Julien	Mécanique
	KUNTZMANN Jean	Mathématiques appliquées
	LACAZE Albert	Thermodynamique
	LACHARME Jean	Biologie végétale
	LAJZEROWICZ Joseph	Physique
	LATREILLE René	Chirurgie générale
	LATURAZE Jean	Biochimie pharmaceutique
	LAURENT Pierre-Jean	Mathématiques appliquées
	LEDRU Jean	Clinique médicale B
	LLIBOUTRY Louis	Géophysique
	LOUP Jean	Géographie
Mle	LUTZ Elisabeth	Mathématiques Pures
MM.	MALGRANGE Bernard	Mathématiques Pures
	MALINAS Yves	Clinique obstétricale
	MARTIN-NOEL Pierre	Seméiologie médicale
	MASSEPORT Jean	Géographie
	MAZARE Yves	Clinique médicale A
	MICHEL Robert	Minéralogie et Pétrographie
	MOURIQUAND Claude	Histologie
	MOUSSA André	Chimie nucléaire
	NEEL Louis	Physique du solide
	OZENDA Paul	Botanique
	PAUTHENET René	Electrotechnique
	PAYAN Jean-Jacques	Mathématiques Pures
	PEBAY-PEYROULA Jean-Claude	Physique
	PERRET René	Servomécanismes
	PILLET Emile	Physique industrielle
	RASSAT André	Chimie systématique
	RENARD Michel	Thermodynamique
	REULOS René	Physique industrielle
	RINALDI Renaud	Physique
	ROGET Jean	Clinique de pédiatrie et de puériculture
	SANTON Lucien	Mécanique
	SEIGNEURIN Raymond	Microbiologie et Hygiène
	SENGEL Philippe	Zoologie
	SILBERT Robert	Mécanique des fluides
	SOUTIF Michel	Physique générale

MM.	TANCHE Maurice	Physiologie
	TRAYNARD Philippe	Chimie générale
	VAILLAND François	Zoologie
	VALENTIN Jacques	Physique nucléaire
	VAUQUOIS Bernard	Calcul électronique
Mme	VERAIN Alice	Pharmacie galénique
M.	VERAIN André	Physique
Mme	VEYRET Germaine	Géographie
MM.	VEYRET Paul	Géographie
	VIGNAIS Pierre	Biochimie médicale
	YOCCOZ Jean	Physique nucléaire théorique

PROFESSEURS ASSOCIES

MM.	BULLEMER Bernhard	Physique
	HANO JUN-ICHI	Mathématiques Pures
	STEPHENS Michaël	Mathématiques appliquées

PROFESSEURS SANS CHAIRE

MM.	BEAUDOING André	Pédiatrie
Mme	BERTRANDIAS Françoise	Mathématiques Pures
MM.	BERTRANDIAS Jean-Paul	Mathématiques appliquées
	BIAREZ Jean-Pierre	Mécanique
	BONNETAIN Lucien	Chimie minérale
Mme	BONNIER Jane	Chimie générale
MM.	CARLIER Georges	Biologie végétale
	COHEN Joseph	Electrotechnique
	COUMES André	Radioélectricité
	DEPASSEL Roger	Mécanique des fluides
	DEPORTES Charles	Chimie minérale
	GAUTHIER Yves	Sciences biologiques
	GAVEND Michel	Pharmacologie
	GERMAIN Jean-Pierre	Mécanique
	GIDON Paul	Géologie et Minéralogie
	GLENAT René	Chimie organique
	HACQUES Gérard	Calcul numérique
	JANIN Bernard	Géographie
Mme	KAHANE Josette	Physique
MM.	MULLER Jean-Michel	Thérapeutique
	PERRIAUX Jean-Jacques	Géologie et Minéralogie
	POULOUJADOFF Michel	Electrotechnique
	REBECQ Jacques	Biologie (CUS)
	REVOL Michel	Urologie
	REYMOND Jean-Charles	Chirurgie générale
	ROBERT André	Chimie papetière
	DE ROUGEMONT Jacques	Neurochirurgie
	SARRAZIN Roger	Anatomie et chirurgie
	SARROT-REYNAULD Jean	Géologie
	SIBILLE Robert	Construction mécanique
	SIROT Louis	Chirurgie générale
Mme	SOUTIF Jeanne	Physique générale

MAITRES DE CONFERENCES ET MAITRES DE CONFERENCES AGREGES

Mle	AGNIUS-DELORD Claudine	Physique pharmaceutique
	ALARY Josette	Chimie analytique
MM.	AMBLARD Pierre	Dermatologie
	AMBROISE-THOMAS Pierre	Parasitologie
	ARMAND Yves	Chimie
	BEGUIN Claude	Chimie organique
	BELORIZKY Elie	Physique
	BENZAKEN Claude	Mathématiques appliquées
	BILLET Jean	Géographie
	BLIMAN Samuel	Electronique (EIE)
	BLOCH Daniel	Electrotechnique
Mme	BOUCHE Liane	Mathématiques (CUS)
MM.	BOUCHET Yves	Anatomie
	BOUVARD Maurice	Mécanique des fluides
	BRODEAU François	Mathématiques (IUT B)
	BRUGEL Lucien	Energétique
	BUISSON Roger	Physique
	BUTEL Jean	Orthopédie
	CHAMBAZ Edmond	Biochimie médicale
	CHAMPETIER Jean	Anatomie et organogénèse
	CHIAVERINA Jean	Biologie appliquée (EFP)
	CHIBON Pierre	Biologie animale
	COHEN-ADDAD Jean-Pierre	Spectrométrie physique
	COLOMB Maurice	Biochimie médicale
	CONTE René	Physique
	COULOMB Max	Radiologie
	CROUZET Guy	Radiologie
	DURAND Francis	Métallurgie
	DUSSAUD René	Mathématiques (CUS)
Mme	ETERRADOSSI Jacqueline	Physiologie
MM.	FAURE Jacques	Médecine légale
	GENSAC Pierre	Botanique
	GIDON Maurice	Géologie
	GRIFFITHS Michaël	Mathématiques appliquées
	GROULADE Joseph	Biochimie médicale
	HOLLARD Daniel	Hématologie
	HUGONOT Robert	Hygiène et Médecine préventive
	IDELMAN Simon	Physiologie animale
	IVANES Marcel	Electricité
	JALBERT Pierre	Histologie
	JOLY Jean-René	Mathématiques Pures
	JOUBERT Jean-Claude	Physique du solide
	JULLIEN Pierre	Mathématiques Pures
	KAHANE André	Physique générale
	KUHN Gérard	Physique
	LACOUME Jean-Louis	Physique
Mme	LAJZEROWICZ Jeannine	Physique
MM.	LANCIA Roland	Physique atomique
	LE JUNTER Noël	Electronique
	LEROY Philippe	Mathématiques
	LOISEAUX Jean-Marie	Physique nucléaire
	LONGEQUEUE Jean-Pierre	Physique nucléaire
	LUU DUC Cuong	Chimie organique
	MACHE Régis	Physiologie végétale
	MAGNIN Robert	Hygiène et Médecine préventive
	MARECHAL Jean	Mécanique
	MARTIN-BOUYER Michel	Chimie (CUS)

MM.	MAYNARD Roger	Physique du solide
	MICHOULIER Jean	Physique (IUT A)
	MICOUD Max	Maladies infectieuses
	MOREAU René	Hydraulique (INP)
	NEGRE Robert	Mécanique
	PARAMELLE Bernard	Pneumologie
	PECCOUD François	Analyse (IUT B)
	PEFFEN René	Métallurgie
	PELMONT Jean	Physiologie animale
	PERRET Jean	Neurologie
	PERRIN Louis	Pathologie expérimentale
	PFISTER Jean-Claude	Physique du solide
	PHELIP Xavier	Rhumatologie
Mlle	RIERY Yvette	Biologie animale
MM.	RACHAIL Michel	Médecine interne
	RACINET Claude	Gynécologie et obstétrique
	RENAUD Maurice	Chimie
	RICHARD Lucien	Botanique
Mme	RINAUDO Marquerite	Chimie macromoléculaire
MM.	ROMIER Guy	Mathématiques (IUT B)
	SHOM Jean-Claude	Chimie générale
	STIEGLITZ Paul	Anesthésiologie
	STOEBNER Pierre	Anatomie pathologique
	VAN CUTSEM Bernard	Mathématiques appliquées
	VEILLON Gérard	Mathématiques appliquées (INP)
	VIALON Pierre	Géologie
	VOOG Robert	Médecine interne
	VROUSSOS Constantin	Radiologie
	ZADWORNY François	Electronique

MAITRES DE CONFERENCES ASSOCIES

MM.	BOUDOURIS Georges	Radioélectricité
	CHEEKE John	Thermodynamique
	GOLDSCHMIDT Hubert	Mathématiques
	SIDNEY STUARD	Mathématiques Pures
	YACOUD Mahmoud	Médecine légale

CHARGES DE FONCTIONS DE MAITRES DE CONFERENCES

Mme	BERIEL Hélène	Physilogie
Mme	RENAUDET Jacqueline	Microbiologie

Fait le 30 mai 1972.

Mes remerciements vont en premier lieu à Jean Raymond ABRIAL, qui, à la tête du projet SOCRATE, sut toujours donner à notre équipe souffle et dynamisme ; il est à l'origine de ces travaux et en demeure l'animateur infatigable.

Je tiens aussi à remercier :

Monsieur Louis BOLLIET, qui a toujours manifesté beaucoup d'intérêt pour nos travaux, et qui me fait l'honneur de présider le jury ;

Monsieur Louis POUZIN, dont la renommée dans le domaine des systèmes n'est plus à faire, qui s'est penché sur ce travail et a accepté de le juger ;

Monsieur Gérard VEILLON, qui a bien voulu faire partie du jury, et qui, par ses nombreuses critiques, a largement contribué à l'amélioration de cet ouvrage ;

Et toute l'équipe du projet SOCRATE :

D'abord Robert MORIN, dont la compétence et le souci d'efficacité nous furent précieux ;

Jean Paul CAHEN et Jean Claude FAVRE, avec lesquels la collaboration fut permanente, détendue et enrichissante ;

Rui BARBOSA, Dominique PORTAL et Andrée STIERS, qui complètèrent cette équipe si sympathique ; et enfin tous nos "anciens" qui surent donner un élan si généreux au projet.

Je n'oublie pas non plus tous ceux et celles qui ont contribués à la réalisation matérielle de cet ouvrage.

-:-:-:-:-:-:-:-:-:-:-

S O M M A I R E

INTRODUCTION ---------- 1

1. ARCHITECTURE GENERALE - NIVEAUX DE MACHINES ---------- 2

1.1. Contraintes et objectifs du système ---------- 2

1.2. 1.1.1. Objectifs ---------- 2

1.1.2. Matériel ---------- 5

1.2. Architecture générale du système en niveaux ---------- 8

1.2.1. Intérêt ---------- 8

1.2.2. Construction en 3 niveaux ---------- 8

1.3. Construction d'un système d'exploitation par machines virtuelles ---------- 10

1.3.1. Résolution séparée des problèmes "technologiques" 10

1.3.2. Construction de machines virtuelles simples et résolution des problèmes "technologiques" ------- 11

1.3.2.1. Taches

1.3.2.2. Gestion du hardware

1.3.2.3. Concept d'interruption

1.3.3. Univers des machines virtuelles ---------- 14

1.3.4. Hiérarchies de machines ---------- 15

1.3.5. ---------- 16

1.4. Présentation du système d'exploitation de SOCRATE ------ 17

1.4.1. Description d'une "machine de base" ---------- 17

1.4.2. Articulation des chapitres suivants ---------- 17

2. DESCRIPTION EXTERNE D'UNE "MACHINE DE BASE" ---------- 19

2.1. Mémoire virtuelle ---------- 19

2.1.1. Les mémoires virtuelles dans les systèmes ------- 19

2.1.2. Les espaces virtuels de SOCRATE ---------- 22

2.1.2.1. **Définition**

2.1.2.2. Réalisation

2.1.3. Emploi des espaces virtuels par les taches ----- 25

2.1.4. Mécanisme d'adressage ---------- 26
2.1.4.1. Registres spéciaux et "voies"
2.1.4.2. Défauts de page
2.1.4.3. Utilisation des primitives de couplage par le superviseur
2.1.5. Mécanismes intermédiaires ---------- 29

2.2. Gestion des ressources autres que la mémoire -------- 31
2.2.1. Les processeurs logiques ---------- 31
2.2.2. Coopération entre ces processeurs logiques ---- 32
2.2.3. Activité physique de ces processeurs ---------- 34
2.2.4. Parallèlisme et activité logique des "processeurs-logiques" ---------- 35
2.2.5. Configuration logique ---------- 39

2.3. Organisation des taches ---------- 40
2.3.1. Taches : rappels ---------- 40
2.3.2. Création dynamique de taches ---------- 40
2.3.3. Parallèlisme effectif ---------- 41
2.3.4. Coroutines ---------- 42
2.3.5. Bases, taches et contextes ---------- 44
2.3.6. Organisations possibles des taches dans le superviseur ---------- 45

2.4. Conclusion : taches et synchronisation ---------- 46

3. FONCTIONNEMENT DE LA BANQUE SUR CETTE MACHINE DE BASE ----- 48
3.1. Le superviseur ---------- 48
3.1.1. Base et Utilisateurs ---------- 48
3.1.2. Définition des processeurs, des taches et des contextes ---------- 49
3.1.2.1. Taches
3.1.2.2. Processeurs
3.1.2.3. Contextes
3.1.3. Algorithmes déroulés pour un utilisateur ------ 52
3.1.4. "Appels superviseur" par des programmes compilés 53

3.2. Périphériques ---------- 54
3.2.1. Travail des périphériques dans l'espace virtuel 54
3.2.2. Attachements ---------- 57
3.3. Problèmes de protection d'accès ---------- 59
3.3.1. Droits d'accès aux objets partageables ------ 59
3.3.2. Protections physiques fournies par les systèmes de segmentation et pagination ---------- 60
3.3.3. Partage des bases de données ---------- 61
3.3.4. Problèmes posés par les langages de bas niveau 62
3.4. Problèmes de dead-lock ---------- 64

4. DESCRIPTION INTERNE DE LA MACHINE DE BASE ---------- 67
4.1. Présentation ---------- 67
4.1.1. Rappels et terminologie ---------- 67
4.1.2. Régulateurs ---------- 68

4.2. Régulation des processeurs ---------- 70

4.3. Cadres logiques et régulation mémoire ---------- 72
4.3.1. Définitions ---------- 72
4.3.2. Utilisation des cadres logiques ---------- 73
4.3.3. Régulateur de mémoire ---------- 73
4.3.4. Suite du travail de préparation d'un cadre logique ---------- 74

4.4. Travail sur l'espace virtuel ---------- 75

4.5. Couplage des deux régulations ---------- 77
4.5.1. Schéma général ---------- 77
4.5.2. Mécanisme du couplage ---------- 80
4.5.3. "Interruptions" ---------- 81

4.6. Politiques d'allocation ---------- 82

CONCLUSION ---------- 86

ANNEXE A : Protection d'accès aux bases ---------- 93

Le projet SOCRATE a pour objectif l'étude et la réalisation d'une Banque de Données, qui permette à un grand nombre d'utilisateurs de travailler à la fois sur plusieurs Bases de Données structurées, et connectées ("on line"). Un système d'accès multiple doit donc assurer le partage des ressources physiques et des informations.

Initialement, nous pensions construire un superviseur simple, et le faire fonctionner comme un programme, sous un système d'exploitation classique. Mais dans le cadre d'un projet de recherche, il a paru plus enrichissant de concevoir et de réaliser un système d'exploitation complet, capable de travailler seul, et comportant d'autres composants (compilateurs, assembleurs, ...) que le seul compilateur du langage SOCRATE. Nous pensons ainsi montrer que certaines idées déjà développées dans ce projet sont générales, et peuvent servir à un système plus complet. D'autre part, les concepts dégagés dans une telle construction peuvent être plus "purs", n'étant pas tributaires d'un autre système. Un développement envisagé de SOCRATE concernant la définition de structures de machines plus appropriées, il est intéressant de travailler directement sur celles qui existent, et de les remettre en cause. Enfin, des contraintes de réalisation pratique du prototype nous ont poussés dans ce sens.

Un important document [8] a été publié récemment par l'équipe SOCRATE, et fait le point des recherches actuelles. On y trouvera détaillés les mécanismes et les algorithmes de ce système d'exploitation. La présentationqui va suivre veut en dégager seulement les idées et concepts principaux, et les comparer à ce qui existe dans de récentes réalisations de même type. Elle est volontairement incomplète, et aborde trop vite bon nombre de points que le lecteur retrouvera en [8]. Elle se veut une étude critique autant qu'une présentation de notre travail.

Dans un premier chapitre introductif, nous décrivons les caractéristiques externes du système réalisé, puis nous détaillons les notions de "niveau de machine" et de "machines virtuelles", qui paraissent fondamentales dans la construction d'un système. Nous pourrons alors expliciter la conception de celui-ci à l'aide de ces notions, puis en détailler les différents "niveaux" dans les chapitres suivants.

CHAPITRE 1

1. ARCHITECTURE GENERALE - NIVEAUX DE MACHINE

1.1. Contraintes et Objectifs du système

1.1.1. Objectifs

La Banque de Données SOCRATE a pour but de permettre à un grand nombre d'utilisateurs de travailler simultanément sur plusieurs Bases de Données. Ces bases recouvrent chacune une application distincte, et n'ont a priori que peu de rapports entre elles. On peut imaginer une gestion du personnel, une gestion de stocks, un système de réservation automatique, un système de documentation, etc... Un certain nombre de ces bases peuvent être montées "on line" et utilisées à la fois.

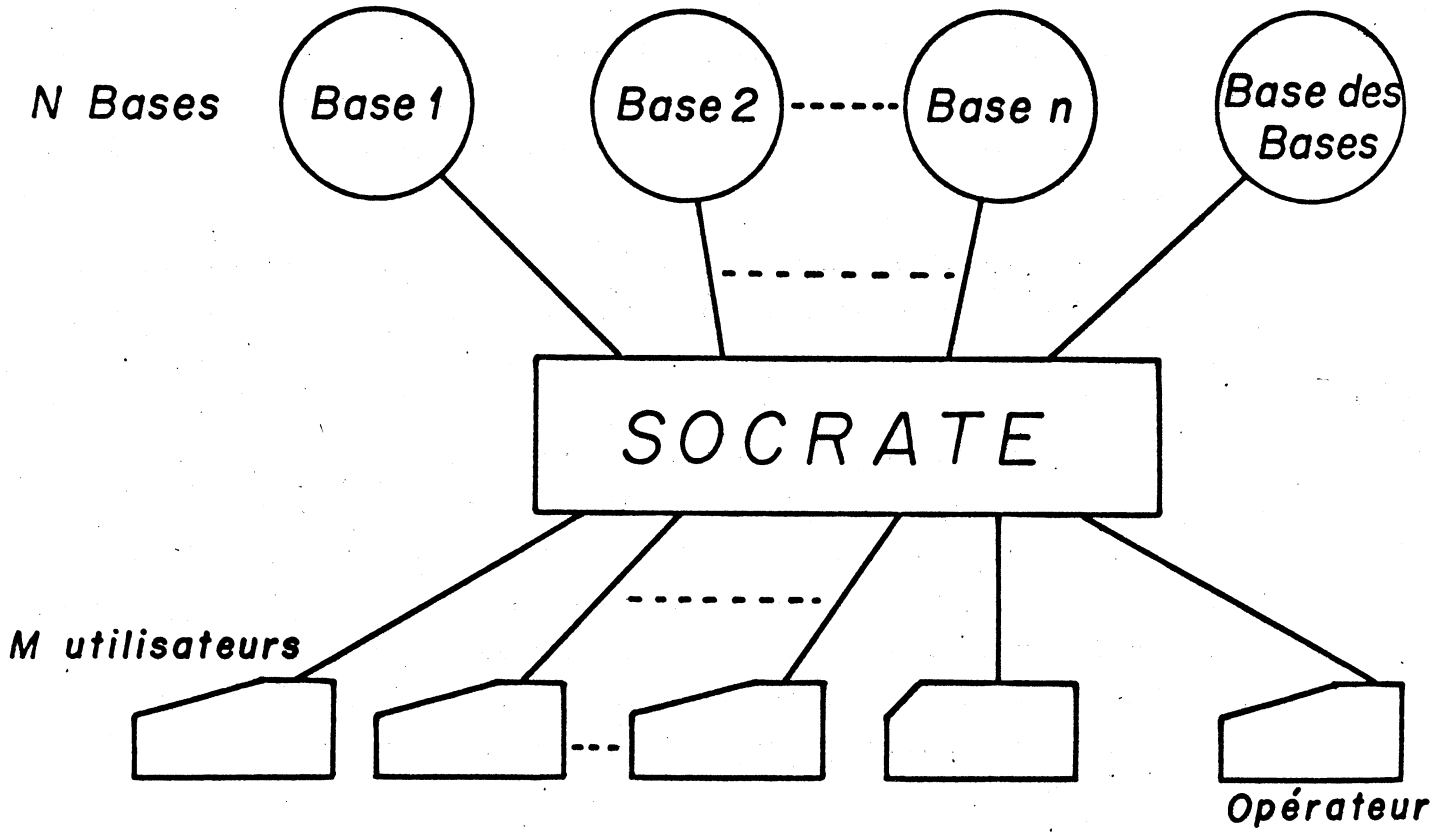


Fig : 1.1 - Partage de N Bases par M utilisateurs

Une "Base des Bases", montée de façon permanente, contient toutes les informations permettant de gérer la Banque : renseignements sur les Bases, sur les utilisateurs, éléments de facturation, etc...

Notre système doit donc assurer deux types de partage :

- Partage des ressources de l'ordinateur : mémoires centrale et auxiliaire unités centrales, périphériques.
- Partage des informations contenues dans les bases : nous voulons permettre des manipulations simultanées de tous les objets qu'elles contiennent; il nous faut donc assurer une cohérence élémentaire et certaines protections.

Les bases de données sont structurées et gérées par les techniques de SOCRATE; un compilateur transforme les requêtes ou programmes de requêtes de l'utilisateur en code capable d'y manipuler des informations.

On y trouve principalement :

- une masse d'informations (de type "fichier"), accessibles à l'ensemble des utilisateurs de la Base (sous d'éventuelles restrictions);
- des zones libres, soit générales, soit propres à l'utilisateur, dans lesquelles on pourra créer de nouveaux objets SOCRATE, puis qu'on pourra libérer (par exemple : création d'une chaîne d'inverses pour une durée limitée) ;
- des programmes soit partagés, soit propres à l'utilisateur; celui-ci peut donc compiler et cataloguer des programmes, considérés par la Banque comme des objets semblables aux autres ("fichier").

Ces bases résident sur des unités à accès aléatoire : essentiellement des disques amovibles. Pendant une session J, N de ces bases sont "on line" et utilisables. Elles sont alors considérées comme des extensions de la mémoire centrale, grâce à un dispositif de pagination; celui-ci assure la présence des "pages disques" dans les "cadres" de la mémoire centrale, à la demande. Le code généré par les compilateurs est lui aussi rangé dans de tels espaces virtuels. Dans le cadre de notre

implémentation, la pagination est entièrement software et les compilateurs en tiennent compte : ainsi ils génèrent une instruction BRANCH pour un branchement dans la même page, mais un "appel-noyau" pour un branchement dans une autre.

Le compilateur du langage SOCRATE est un des composants essentielsdu système. Il permet à l'utilisateur d'accèder aux données structurées, de faire certains calculs, des éditions, etc ... :du travail "de gestion" essentiellement. Un compilateur FORTRAN est aussi en cours de réalisation, de façon à permettre des calculs plus importants. Il est intégré au langage SOCRATE et peut manipuler les objets contenus dans les bases. A un niveau plus "système", un sous-assembleur permet d'écrire certaines procédures réellement optimisées, ainsi que les composants du système; son utilisation pose certains problèmes de protection (qui seront examinés au § 3.3.4.).

Ces trois outils génèrent des programmes "paginés", c'est-à-dire stockés sur disques, ayant connaissance de la pagination, et amenés en mémoire centrale par page, à la demande.

Notre but est donc la réalisation d'un système d'exploitation "stand alone", chargeable directement sur machine nue. Il se compose essentiellement de :

- un noyau résident, réalisant certains mécanismes fondamentaux : élection des utilisateurs, partage des Unités Centrales (UC) et des cadres de mémoire centrale, pagination software;
- des composants non résidents : compilateurs, utilitaires, programmes de gestion de la Base Des Bases (BDB) ;
- des programmes "utilisateurs", par application, rangés dans l'espace virtuel et catalogués;
- des routines d'entrée-sortie sur périphériques; elles pourraient être paginées, on a préféré les rendre résidentes.

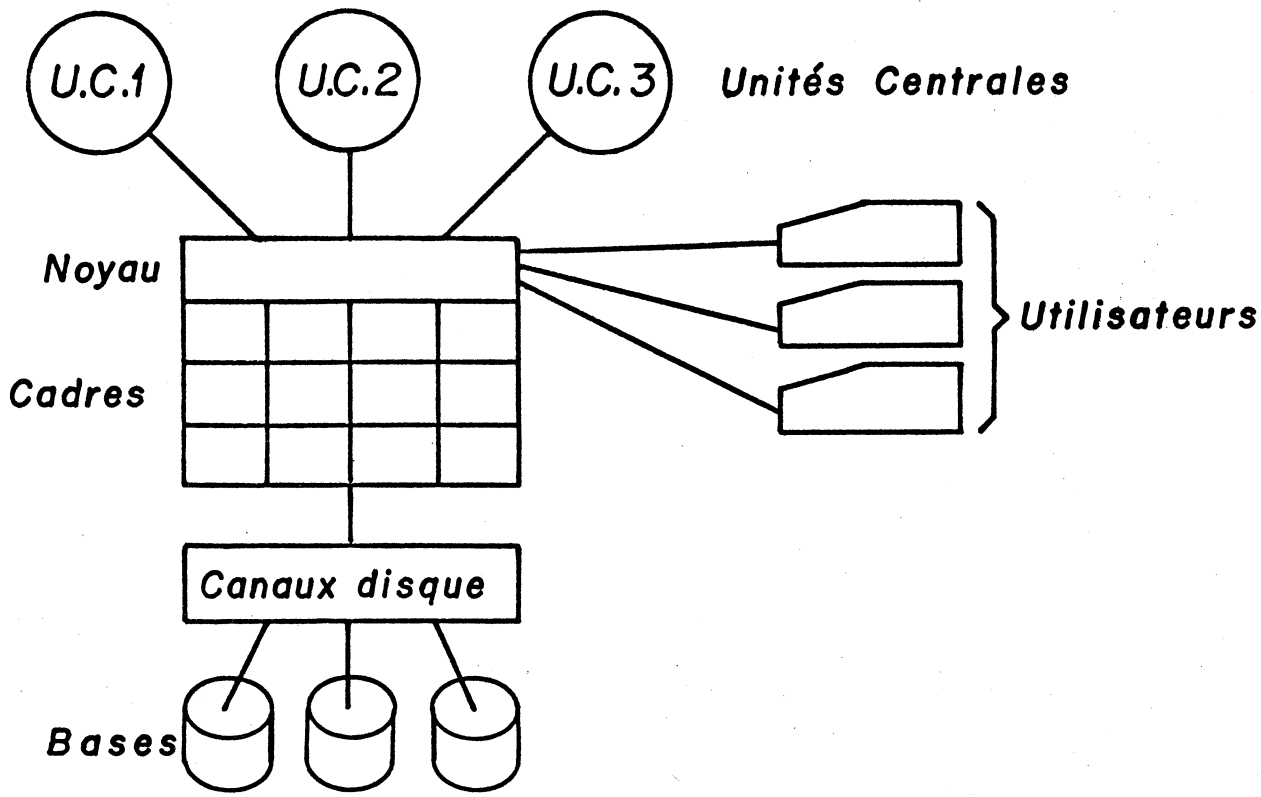


Fig : 1.2 - Organisation générale des ressources physiques

Ce type d'organisation et la taille réduite du noyau résident (de l'ordre de 30 K-octets pour 15 consoles) permettent d'envisager le fonctionnement d'un tel système sur une machine relativement petite.

1.1.2. Matériel

La réalisation effective du prototype a été faite sur un IBM 360 standard, monoprocesseur. En fait, il s'agit d'une machine virtuelle générée par le système CP qui fonctionne sur le 360/67 de l'IMAG. Dans un premier temps nous avons utilisé le sous-système conversationnel CMS avant de l'exclure totalement de notre réalisation.

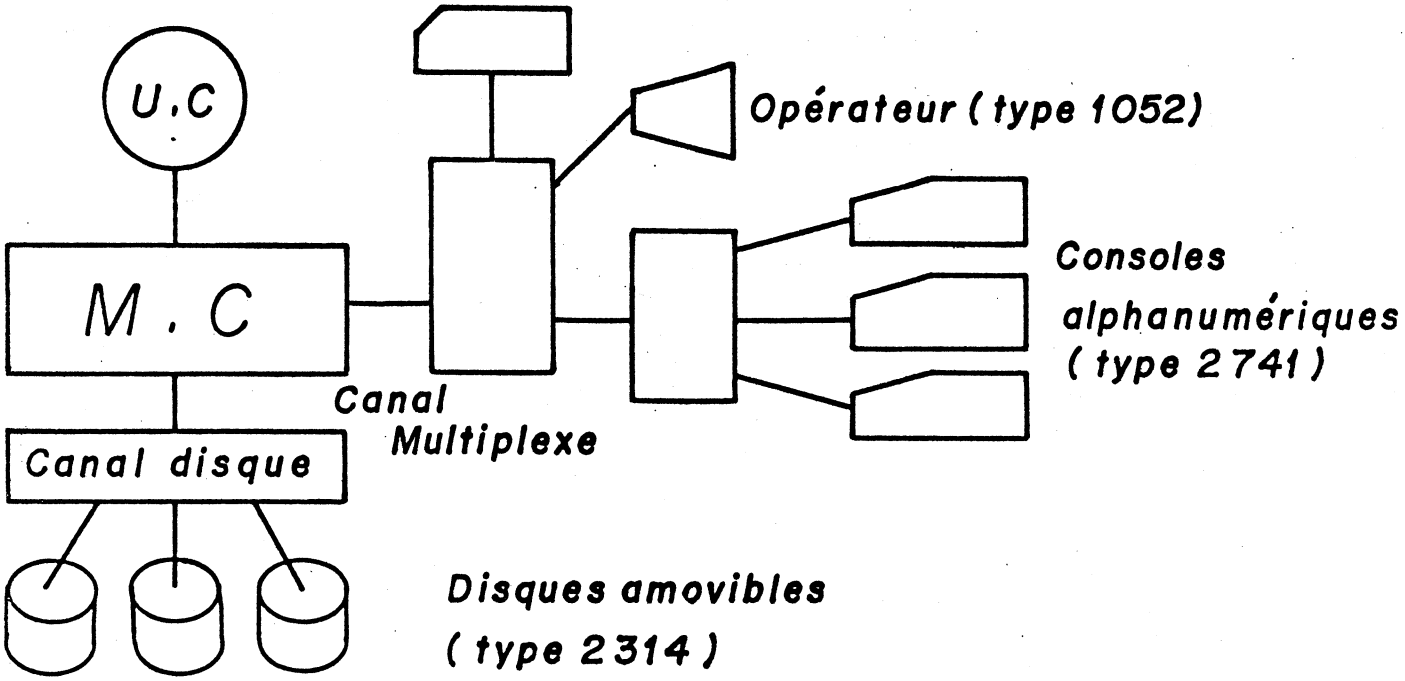


Fig: 1.3 - Configuration réelle

S'il est vrai que nous avons écrit ce prototype en code IBM 360, il reste que les algorithmes et les concepts sont applicables à toute une gamme de machines :

- mono ou multi-processeur
- ayant un mécanisme d'adressage du type Base + Déplacement, la base étant contenue dans un registre général ou spécial (nous en utilisons quatre), et le déplacement maximum supérieur ou égal à 1K.

D'autre part, nous n'avons pas utilisé une série de dispositifs tels que les interruptions internes (SVC), ou la protection mémoire. Nous n'avons pas non plus utilisé l'horloge ("Timer"), le partage en tranche de temps n'étant pas encore implémenté (mais il le sera facilement, la désactivation d'un utilisateur sur interruption étant déjà pratiquée couramment).

Seule une très petite partie du noyau (niveau 0) tient compte des particularités de la machine : système d'interruption et fonctionnement des canaux. Ce niveau construit une machine théorique qui doit être facilement réalisable sur la plupart des ordinateurs actuels. C'est la description d'une telle architecture par "niveaux" que nous allons présenter maintenant.

1.2. Architecture générale du système en niveaux

1.2.1. Intérêt

Dans la programmation de ses applications, l'utilisateur résoud un certain nombre de problèmes qui **lui** sont propres; il utilise le système qui lui est fourni, sans même savoir comment celui-ci est implémenté : il n'en a qu'une connaissance externe.

Lors de la réalisation de ce système, on peut encore distinguer deux types de problèmes, déjà présentés dans sa thèse par SALTZER [48] :

- les problèmes technologiques, dus au fait que les ressources physiques sont limitées et qu'il faut les partager ;
- les problèmes intrinsèques : essentiellement des problèmes de synchronisation et de communication entre les objets qui le composent.

Il est tentant d'essayer de construire un tel système en deux étapes, en deux "niveaux" : les problèmes technologiques étant résolus au niveau 0, le reste du superviseur peut être construit en ignorant les premiers problèmes. Si de plus toutes les particularités de la machine physique sont traitées à ce niveau, tout le reste peut en être complètement indépendant; toutefois cela est peut être de moindre intérêt dans le cas où un grand nombre de composants sont écrits dans le code de cette machine.

1.2.2. Construction en 3 niveaux

En rajoutant un niveau "utilisateur", on distinguera trois couches :

(a) Le noyau (niveau 0);
(b) Le superviseur et les composants ;
(c) Les utilisateurs qui peuvent bâtir des "sous-systèmes" concernant leur application.

L'interface entre (a) et (b) veut être une machine homogène, cohérente, en introduisant au besoin des concepts qui n'existent pas sur la machine physique. Ensuite (b) gère (c) à l'aide des primitives de (a), et des composants de (b) (toujours activés à l'aide de (a)).

Vu de (b) et (c), (a) apparaît alors comme une machine particulière, théorique, et plus simple à manier que la machine physique. C'est une "machine virtuelle", concept qui a été utilisé et travaillé par la plupart des systèmes de multiprogrammation, comme on va le voir. Les intérêts en sont nombreux : logique de la construction, simplicité et clarté des mécanismes, séparation possible du travail, "portabilité", etc ...

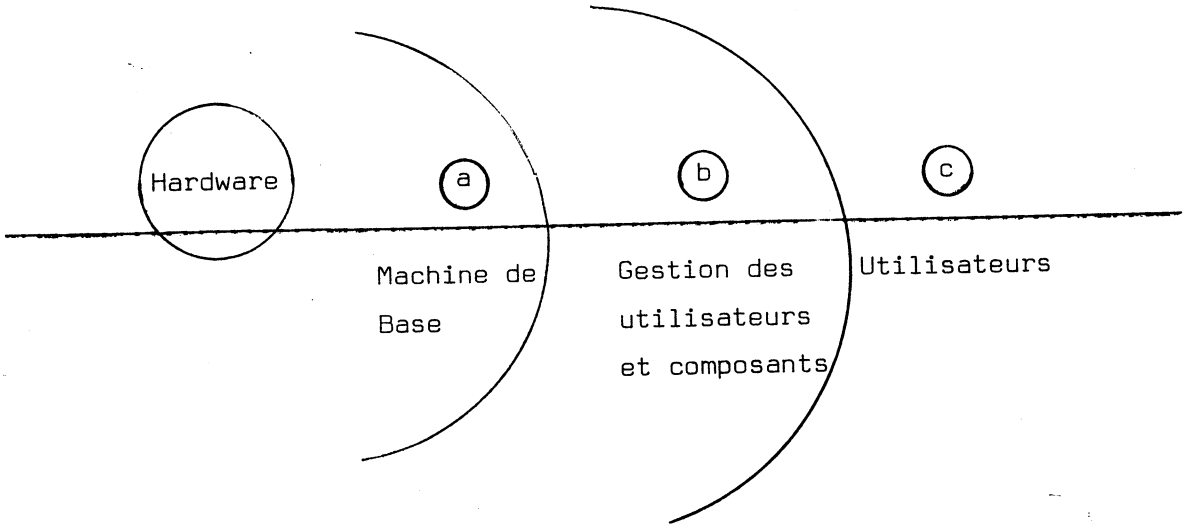


Fig : 1.4 - Trois niveaux de machine.

1.3. Construction d'un système d'exploitation par machines virtuelles

1.3.1. Résolution séparée des problèmes "technologiques"

Ces problèmes sont résolus par un mécanisme que nous appellerons "dispatcher" ou "régulateur", et dont le rôle est de créer des "processeurs logiques" (LAMPSON [41]) ou "pseudo-processeur" (SALTZER [49]) sur la base du ou des processeurs physiques.

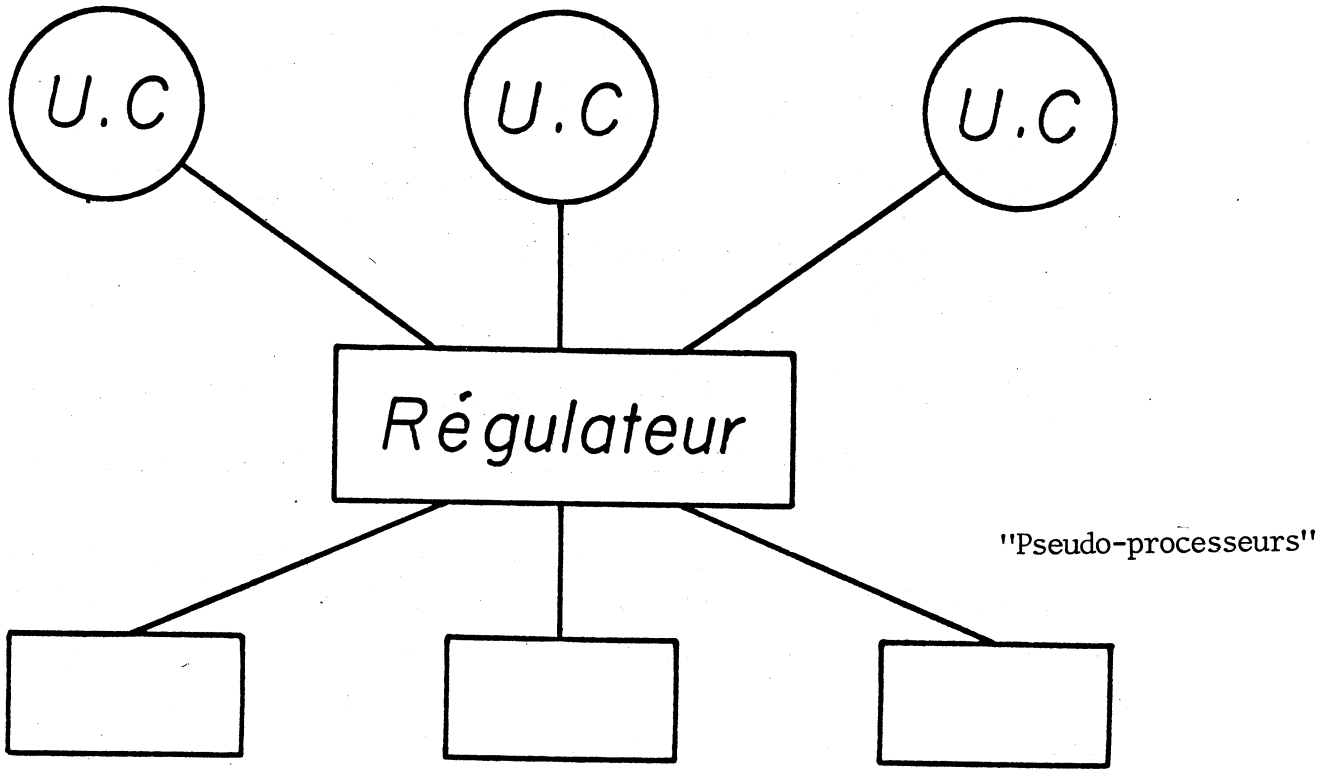


Fig : 1.5 - Création de "pseudo-processeurs" sur la base des processeurs physiques.

Dans ces premières approches le "pseudo-processeur" généré n'est pas tout de suite assimilé aux "tâches" ou "processus"*, qui sont les entités de base du niveau voisin. Cependant, cette distinction a disparu par la suite ([26], [33], [36], [51]). On verra en § 2 comment dans notre construction nous avons ré-introduit ce niveau intermédiaire, pour en faire un concept essentiel de notre machine.

* *Nota : Précisons le sens que nous donnons à ces termes :*
Un "processus" est l'activité d'un programme pour le compte d'un utilisateur.
Une "tache" représente souvent l'ensemble des activités mises en jeu par un utilisateur ; c'est alors une famille de processus coopérants, travaillant sur des objets communs.
Nous avons choisi de réaliser seulement des taches à un seul processus ; par la suite, nous confondrons donc les deux termes.

1.3.2. Construction de machines virtuelles simples et résolution des problèmes "technologiques"

1.3.2.1. La solution la plus classique consiste à activer un certain nombre de taches ou processus sur la ou les unités centrales.

Ces tâches peuvent être strictement séquentielles, "monoprogrammées" : c'est le cas de SIRIS 8 [14]. Elles sont alors très simples à utiliser; le bon emploi des ressources du système (parallèlisme) est garanti par le fait qu'on travaille en multiprogrammation. C'est aussi la solution que nous avons adoptée.

Dans des systèmes plus complexes on autorise les taches à programmer un certain parallélisme : le programmeur peut initialiser, démarrer, stopper, relancer et tuer des taches; il peut utiliser des primitives qui permettent à ces sous-taches de se synchroniser entre elles.

C'est le cas des systèmes de WIRTH ([51]), HANSEN ([33],[34]) entre autres. On peut alors reporter certains problèmes de "Scheduling" au niveau de la famille des sous-taches, le programmeur décidant d'activer une tâche et de bloquer les autres. On construit ainsi des systèmes dans lesquels les fonctions de scheduling sont décentralisées (HANSEN [33]). C'est une première approche des systèmes d'exploitation extensibles.

1.3.2.2. Gestion du hardware

Dans ces systèmes, la gestion des interruptions externes, en provenance des périphériques, est transparente, cachée au programmeur qui manipule des objets plus logiques (§ 1.3.2.3.).

Le noyau ("nucleus") apparaît alors comme une extension de la structure hardware de la machine ; cependant, le désir de conserver une certaine souplesse dans les algorithmes de dispatching conduit à préférer une réalisation software, ou mixte (HANSEN [33], [34]).

Les taches sont alors distribuées de façon quelconque sur les processeurs hardware, leur fonctionnement est en particulier indépendant du nombre de ces processeurs, a tel point que celui-ci peut varier dynamiquement. Cela permet de modifier la configuration physique dans le cours de la session, rien qu'en déclarant au dispatcher que ce processeur n'est plus disponible (BOULLE [14]). Nous avons ainsi réalisé un "régulateur" qui règle ces problèmes de gestion du hardware au niveau 0, indépendamment des niveaux supérieurs.

On va voir que ce "hardware management" est un mécanisme très voisin du système d'interruption sur une machine classique (du type 360).

1.3.2.3. Concept d'interruption

On distingue habituellement les interruptions externes, mécanisme de synchronisation des Unités Centrales, avec les périphériques et l'extérieur, des interruptions internes qui ne servent qu'à intercaler du code à la place d'un "appel superviseur". Les premières peuvent remettre en cause l'affectation des taches aux processeurs, ce qui n'est pas le cas des secondes ([16], [48], [14]). Nous nous intéressons ici seulement aux premières. (On verra que dans notre système, interruptions internes et externes sont toutes traitées de la même façon).

Le concept d'interruption, généralisé sur les machines de la 3ème génération pour permettre un parallèlisme entre unité centrale et périphérique, s'est révélé d'un emploi pénible et délicat pour le programmeur. Dans les systèmes modernes ce concept, s'il existe toujours à la base, disparaît dès que l'on travaille au dessus du niveau 0, pour être remplacé par d'autres mécanismes.

CONWAY [16], WIRTH [51] considèrent une entrée-sortie comme une instruction spéciale (DOIO de WIRTH), qui a la caractéristique d'être très longue. Le programmeur qui a par ailleurs la possibilité de faire du parallèlisme en créant des sous-taches, doit le programmer effectivement s'il veut travailler à autre chose en attendant la fin de l'entrée/sortie.

Une autre solution proposée par HANSEN [33] consiste, en même temps qu'en construisant le niveau 0, à livrer avec le système un certain nombre de processus "externes". Le programmeur les démarrera et se synchronisera avec eux en utilisant des primitives qu'il a à sa disposition. Ce sont ces processus qui réalisent les entrée/sortie (au contraire des processus internes qui, eux, "supportent" les utilisateurs).

Cette solution, met en jeu deux "niveaux" de programmation dans la réalisation du "noyau" livré aux utilisateurs; mais elle semble intéressante parce qu'elle permet d'utiliser les mêmes mécanismes de synchronisation que le reste du système.

Pour nous, nous avons choisi de générer au niveau 0 des "processeurs logiques" associés aux périphériques; ces processeurs peuvent s'endormir après lancement de l'entrée-sortie et sont réveillés par l'interruption. Les algorithmes réalisant l'exécution de telles entrée-sortie sont donc programmés facilement avec les primitives du niveau 0, mais peuvent être livrés avec le noyau, comme les processus externes de HANSEN.

Tout cela est détaillé au § 2.2.

Plusieurs auteurs reconnaissent le système d'interruption comme un "scheduler hardware" ; c'est en effet un mécanisme qui remet en cause l'affectation des processeurs aux taches. Il est cependant regrettable que ce scheduler soit insuffisant et doive être complété par un scheduler software, forcément dissocié. Une approche plus satisfaisante serait la construction d'un seul scheduler, cohérent, totalement hardware. Le concept d'interruption serait alors remplacé par les mécanismes de synchronisation fournis au niveau 0. Tout cela est développé par LAMPSON [41]; des systèmes comme SIRIS 8 vont déjà dans ce sens [14].

1.3.3. Univers de ces machines virtuelles

Les taches travaillent sur des objets, programmes ou données, qui leur sont propres, ou qu'elles partagent avec les autres.

Certains systèmes allouent un espace virtuel à chaque tache, on a chaque famille de tache. Cette notion d'espace virtuel qui recouvre beaucoup de chose sera reprise au § 2.1.

DENNIS et VAN HORN [23] associent à une famille de tache une "C-list" qui décrit ses droits sur les données et les programmes du système : c'est une définition de l' "univers" de la tache.

D'autres systèmes utilisant un mécanisme de segmentation et pagination associent une table de segment à chaque tache (ou machine virtuelle). Cette table décrit aussi tout l'univers de la machine ; les restrictions d'accès y sont indiquées et sont vérifiées par hardware.

Ces taches ou machines virtuelles apparaissent donc :

- d'une part, comme des machines capables de fonctionner (elles sont "supportées" par les processeurs réels), et de dérouler un jeu d'instructions comprenant des instructions machines et des primitives fournies par le noyau ;
- d'autre part, comme des entités vivant dans un certain univers, cohérent, mais qui n'est qu'une partie du système dans lequel elles se trouvent.

1.3.4. Hiérarchies de machines

Le terme fréquemment utilisé regroupe plusieurs idées différentes, mais non indépendantes. Soulignons certains de ces aspects :

1.3.4.1. Hiérarchie de réalisation d'un système en couches concentriques (DIJKSTRA [26]) : à partir de la machine physique on bâtit une machine de niveau 0; une fois celle-ci testée, on l'utilise pour bâtir une machine de niveau 1 plus complète; etc... L'intérêt de cette approche réside dans les facilités de réalisation apportées : test systématiques, mise au point plus rapide, ... (C'est plutôt une méthodologie).

1.3.4.2. Systèmes d'exploitation hiérarchique : une décentralisation des fonctions de Scheduling étant autorisée, on peut alors construire des "operating systems" extensibles (HANSEN [33]). Cela permet de réaliser un deuxième système sous le premier, etc....

1.3.4.3. Processeurs et Processus sont deux aspects de la même chose :

Si l'on parle de la "machine de base" réalisée au niveau 0, et qu'on la considère du point de vue du programmeur de la tache, c'est un processeur qui se définit par un jeu d'instructions.

Par contre, le concepteur du niveau 0 voit n processus ou machines virtuelles entre lesquelles il doit partager les ressources physiques.

Si maintenant, le programmeur au niveau 1 veut activer des sous-taches et les gérer, le "processus" de niveau 1 devient "processeur" aux yeux des utilisateurs de niveau 2, et ainsi de suite.

Cette conception des systèmes, qui ne s'arrête pas à la frontière classique hardware/software, a été developpée par ANCEAU [10]: c'est la notion de système hiérarchisé.

1.3.5.

Cette présentation rapide de quelques tendances des systèmes d'exploitations modernes était destinée à préciser certaines idées introduites ou modifiées par ces constructions de système en "niveaux de machine". Elle doit aider à situer notre travail dans ce domaine où les concepts comme les terminologies ne sont pas encore bien précis, et recouvrent souvent plusieurs choses.

1.4. Présentation du système d'exploitation de SOCRATE

1.4.1. Description d'une "machine de base"

Nous allons essentiellement nous attacher à présenter une machine de base, cohérente, d'où aura disparu le concept d'interruption : il s'agit donc d'une machine au "niveau 0", semblable à celles dont nous venons de parler. Elle se situe donc à un seul niveau, et le mécanisme de Scheduling est unique. Tout aspect hiérarchique au sens des § 1.3.4.2 et § 1.3.4.3 en est donc exclu.

Cependant, dans l'utilisation de cette machine, les taches peuvent se mettre en attente sur des "réservations" de certaines ressources. Par exemple, si toutes les taches commencent par réserver un certain nombre de cadres* de mémoire centrale, ne seront actives qu'un nombre de taches limité par le nombre total des cadres. On voit que même si le mécanisme du scheduler est unique, et réalisé au niveau 0, la politique de scheduling n'est pas fixée pour autant, et peut être modifiée par la construction des taches elles-mêmes aux niveaux supérieurs.

* *Nota : Le cadre est l'élément de mémoire centrale qui sera capable de recevoir une page de mémoire secondaire (cf. p. 24).*

1.4.2. Articulation des chapitres suivants

Dans le chapitre 2 nous présentons cette machine "vue de l'extérieur" c'est-à-dire du programmeur du niveau supérieur. Ses caractéristiques essentielles sont :

- l'accès à des mémoires virtuelles partagées ou non, et d'un type particulier ;
- elle fournit un jeu de "processeurs logiques" cohérents, qui savent travailler ensemble. Ces processeurs restent à définir (à "déclarer") au niveau supérieur. On peut aussi écrire certaines fonctions d'entrée-sortie élaborées et les associer à des processeurs logiques de type "périphériques".

- elle permet à N taches d'être actives en pseudo-simultanéité. Ce sont ces taches qui dérouleront les algorithmes écrits aux niveaux supérieurs.

La construction du superviseur de SOCRATE à l'aide de cette machine, et son architecture complète seront alors détaillées au § 3.

Enfin la description interne de la réalisation et du fonctionnement seront donnés au § 4.

CHAPITRE 2

2. DESCRIPTION EXTERNE D'UNE "MACHINE DE BASE"

Le rôle de cette machine est d'activer un certain nombre de taches (§ 2.3) sur des processeurs-logiques (§ 2.2); ces taches manipulent des données et des programmes situés dans des espaces virtuels que nous allons maintenant décrire.

2.1. Mémoire virtuelle

Après avoir rapidement revu ce que le terme de mémoire virtuelle signifie dans les systèmes actuels, nous dégagerons les particularités des "espaces virtuels" de SOCRATE; puis nous définirons ce que les taches peuvent en faire, et comment elles le font, en détaillant le mécanisme d'adressage.

2.1.1. Les mémoires virtuelles dans les systèmes

On distingue fonctionnellement deux types de mémoire :

- les mémoires de manoeuvre, dont le contenu est perdu d'une session à l'autre (mémoire centrale, éventuellement tambour de pagination, ...)
- les mémoires permanentes, contenant par exemple des fichiers, des programmes, ...; elles sont généralement amovibles, et beaucoup plus grandes que les mémoires de manoeuvre.

La notion de mémoire virtuelle a été introduite pour apporter un certain nombre de facilités dans la manipulation de ces mémoires :

- continuité de la mémoire centrale ;
- homogénéisation des différents niveaux de mémoire de manoeuvre ;
- création d'une mémoire virtuelle par processus (ou machine virtuelle) ;
- partage de certains objets (données, programmes), ce qui peut résoudre aussi des problèmes de communication entre processus ;
- nomination et partage d'objets permanents, ce qui permet alors de résoudre les problèmes de liens entre ces objets.

Certains de ces problèmes sont résolus de façon hardware par le mécanisme d'adressage de certaines machines ; d'autres le sont par des systèmes d'exploitations utilisant au mieux ces mécanismes, même si ceux-ci n'ont pas été conçus dans ce but.

La continuité de la mémoire centrale peut être simulée par des dispositifs du type mémoire topographique (Ex : 10070)

L'homogénéisation de la mémoire de manoeuvre (mémoire centrale et disques à tête fixe, par exemple) est résolue par un mécanisme d'adressage du type :

[Table des pages, Adresse page, Adresse mot] (IRIS 80 Mode B [15])

ou du type :

[Table des segments, Table des pages, Adresse page, Adresse mot]

(IBM 360/67, 370 [40] ; IRIS 80 Mode C) ;

ce mécanisme est complété par des routines de "defaut de page" capables de transférer une page manquante, et des algorithmes de gestion de l'espace mémoire-centrale et mémoire-secondaire. Le tout constitue un dispositif de pagination ou de segmentation et pagination. Le système (software) qui supporte un certain nombre de taches les désactive pendant la durée du transfert.

Une étape suivante peut être d'assurer une mémoire virtuelle par tache (de taille soit égale, soit éventuellement supérieure à la mémoire centrale). Pour cela, quand on élit une autre tache, on change le pointeur sur la table des pages ou des segments (SIRIS 8, CP/67), ou bien l'on recharge la mémoire topographique (Esope[13]).

Dans ce contexte, le partage d'un certain nombre de pages, sans recopie, entre des processus,peut être réalisé en faisant correspondre certaines pages de différentes mémoires virtuelles à des mêmes pages physiques (Notion de "couplage" dans ESOPE [29], partage de certaines pages du système CMS par les machines virtuelles de CP/67 [42]). Cela est facilité par les dispositifs de protection (hardware) offerts par ces mécanismes d'adressage. Mais le partage de certains objets "logiques" a plus naturellement lieu au niveau du segment : par définition, celui-ci est une entité logique que l'on peut nommer et partager. Ce partage peut être envisagé de façon statique : c'est le cas de systèmes comme CP+ [42]. Mais il trouve sa pleine signification en devenant dynamique : on trouve alors des fonctions du type "rendre un segment connu à un processus" (MULTICS [12]) : le processus n'est plus limité à "sa" mémoire virtuelle.

De tels systèmes offrent en plus un mécanisme de "nomination" permettant de retrouver des segments par leur nom, et aussi de les ranger dans l'espace permanent en leur attribuant un nom. La correspondance entre le nom et la (les) adresse(s) est réalisée au moment où l'on rend le segment connu.

Si le nombre des segments est limité (cas de CP/67), par exemple à une petite table par machine virtuelle, on peut implémenter un tel mécanisme en rendant résidentes toutes les tables de segments et de pages ; mais dès que ce nombre de segments devient dynamique, il faut paginer les tables des pages, et donc introduire la notion de "faute de segment" analogue à la "faute de page", en cas d'accès à un segment dont la table de page est absente de la mémoire centrale (MULTICS [12]). On verra que, en SOCRATE, dès que l'on voudra multiplier le nombre des segments, il sera possible d'introduire un tel mécanisme à deux niveaux (cf § 4.4.).

(Cette présentation des "mémoires virtuelles" dans certains systèmes est très rapide; pour une étude plus complète nous reportons le lecteur à l'article de DENNING [20]).

2.1.2. Les "espaces virtuels" de SOCRATE

2.1.2.1. Définition

Tous les processus de SOCRATE connaissent un nombre réduit d'espaces virtuels que l'on peut assimiler à des segments; cependant :

- ces espaces virtuels sont connus et utilisés chacun par de nombreux processus, sans restriction d'accès a priori (c'est-à-dire au niveau de l'accès hardware);
- ils sont très grands, quasi infinis, et les objets y sont rangés de façon statique; leur réalisation physique est l'oeuvre des algorithmes de "projection" : **espace virtuel → disques;**
- le système de nomination permet d'accèder aux objets à l'intérieur d'un segment, donc à un niveau beaucoup plus fin que le seul niveau de segment; de plus ce système de nomination est universel, c'est-à-dire que tous les processus nomment les objets de la même façon : l'existence et le nom de ceux-ci sont indépendants du processus qui les nomme (sauf évidemment pour des objets "privés", propres au processus).

La correspondance **nom → adresse virtuelle** étant statique, fixe durant toute la vie de l'objet, cela résoud du même coup tous les problèmes de liens entre ces objets (références, adresses des sous-programmes).

Les "espaces virtuels" sont donc des "espaces de noms", assez différents du concept de mémoire virtuelle habituel.

Remarquons encore que le mécanisme d'accès aux objets étant celui du langage SOCRATE, la dénomination de ceux-ci est toujours compilée, et les protections d'accès peuvent toujours être traitées ou mises en place durant cette compilation.

2.1.2.2. Réalisation

Le fonctionnement de notre "pagination" a été complètement décrit en [1] et repris en [8] § 2.3.1. Nous ne rappellerons ici que les idées directrices, très succintement.

L'accès à une information dans l'espace virtuel met en jeu deux mécanismes qui se complètent.

a - La façon dont cet espace virtuel immense (2^{32} mots) est réalisé, "replié" sur un nombre de cylindres restreint : l' "espace disque".

Pour cela, une technique de "hash-code" s'impose :

soient av l'adresse virtuelle d'un atome d'information,
ad l'adresse dans l'espace disque,
treelle la taille de celui-ci.

A un atome d'adresse virtuelle av on sait faire correspondre une adresse disque :

ad = hash(av)

Cette fonction n'est évidemment pas injective, l'espace d'arrivée étant beaucoup plus petit que l'espace de départ ; elle détermine donc des classes d'équivalence sur l'ensemble des adresses virtuelles.

Si plusieurs éléments de la même classe doivent être projetés sur l'espace réel (en ad), ils le seront par nécessité à des adresses disques différentes, mais que l'on saura associer à l'adresse ad de la classe, en les chaînant ensemble dans un "dictionnaire de hash-code" qui comporte une entrée par classe. Ce dictionnaire doit aussi contenir l'information significative permettant de retrouver l'adresse av de chaque élément de la classe : soit advirt(ad).

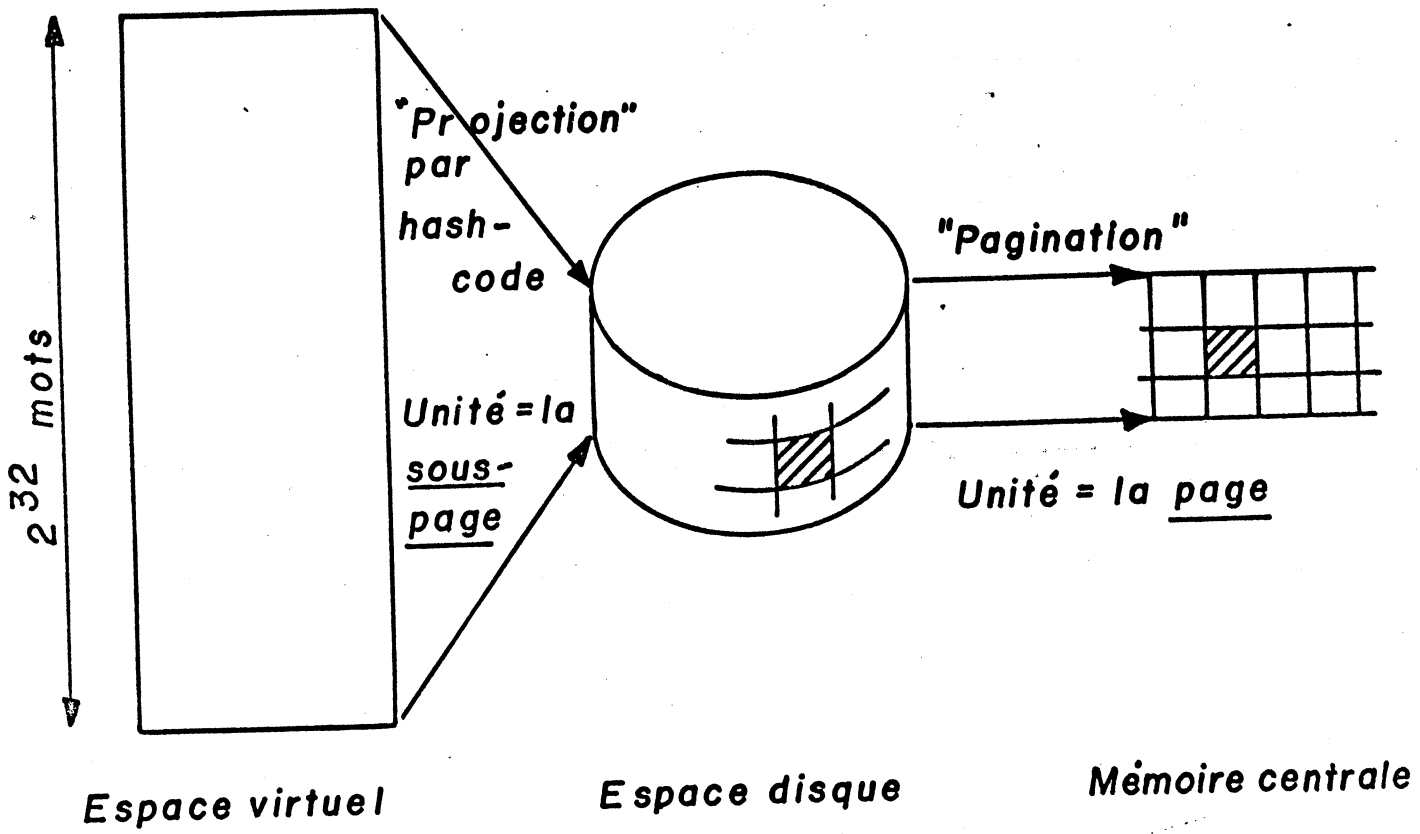


Figure 2.1 : REALISATION DE L'ESPACE VIRTUEL -

Pour trouver l'adresse disque ad de l'élément av, il faut donc :

- calculer ad = hash(av)
- comparer advirt(ad) à av ; s'ils sont différents, suivre le chaînage pour rechercher à d'autres adresses ad' si advirt(ad') = av.

Il est possible de ne rien trouver : l'élément d'adresse av n'a pas de représentation réelle, il n'existe pas (il est nul).

Nous avons choisi comme fonction hash(av) = av mod(treelle), et c'est pourquoi nous parlons d'un "repliement" du virtuel sur le réel. La correspondance est réalisée pour des éléments, des "atomes" de 7 ou 15 mots consécutifs : la sous-page. Et nous avons réalisé le dictionnaire de hash-code en mémoire secondaire, de façon "éparpillé", chaque élément du dictionnaire ou "mot de contrôle" étant rangé avec la sous-page correspondante (cela afin de permettre l'accès en une seule fois aux deux).

b - L'espace disque, considéré comme une extension de la mémoire centrale, est divisé en pages (1 K-octet) ; celles-ci sont amenées à la demande dans les cadres de la mémoire centrale ; s'il n'y a pas de cadre libre on en libère un, en recopiant auparavant son contenu sur disque si celui-ci avait été modifié. Nous employons actuellement LRU comme algorithme de choix du cadre à vider. Il s'agit donc d'une pagination classique, mais software (consultation d'une table des pages présentes dans les cadres).

Actuellement les espaces virtuels sont en nombre très réduit et l'espace disque leur est alloué statiquement; cela signifie que la taille de l'espace réel supportant un espace virtuel n'est pas modifiable dynamiquement. Par contre, les pages en sont consécutives sur disque, ce qui réduit énormément la taille des tables de correspondance :

page de l'espace réel ↔ adresse disque

Ces tables peuvent donc être résidentes, ce qui simplifie le travail de la pagination, et diminue le nombre d'accès disque nécessaires.

2.1.3. Emploi des espaces virtuels par les taches

Considérons une tache, active, en train d'exécuter (par exemple) un programme généré par le compilateur SOCRATE; pour cela elle a besoin d'accèder à un certain nombre de choses en mémoire centrale :

- des instructions du programme ;
- des variables de travail, et des zones de sauvegarde du contexte en cas d'interruption externe ou d'appel de sous-programme ;
- de temps en temps, d'éléments du fichier (base) présents en mémoire centrale ;
- éventuellement de buffers, le temps de certaines entrée-sortie (cela sera revu en § 2.1.5 et § 3.2).

Chacun de ces éléments se trouve dans un espace virtuel :

- les programmes, dans un espace virtuel "programme" attaché à la base ; on y trouvera des programmes privés, et des programmes généraux de l'application. De plus il existe un espace virtuel "programmes généraux" où l'on trouvera les programmes de la Base des Bases et tous les composants du système qui sont paginés.

- les variables de travail et les contextes : ceux-ci s'empilent lors de chaque appel-noyau ou appel de sous-programme; cette "pile des contextes" est rangée dans un espace virtuel système, qui est découpé par utilisateur. (De même qu'en MULTICS [48], où la pile des contextes "call stack" est un segment privé à chaque tache). En fonctionnement normal, on n'utilise que le sommetde cette pile, qui sera toujours contenu dans une page.

- les "fichiers" (la base de données) résident évidemment dans un espace virtuel, dont on amène une page en mémoire pour y prendre l'information.

Pour fonctionner, la tache devra s'assurer que ces pages sont bien présentes en mémoire centrale. Si, à cause de la "préemption", l'une d'elle disparaît, la tache devient "non éligible" jusqu'à ce que cette page soit ramenée en mémoire.

2.1.4. Mécanisme d'adressage en mémoire centrale

2.1.4.1. Registres spéciaux et "voies"

Une tache a besoin pour s'exécuter de 3 pages présentes en mémoire; le système se charge d'amener ces pages avant d'élire la tache, et de faire pointer un registre de base sur chacune d'elle. Le programme pourra alors adresser le contenu de ces pages en spécifiant seulement le déplacement.

Dans notre implémentation 360, par exemple, R9, R10 et R11 sont des bases des pages de travail, programme et fichier; le programme lit la variable TOTO dans la page de travail par L R2, TOTO(R9). L'existence d'un certain nombre d'instructions indexées permet de plus une certaine souplesse dans le maniement de ces variables de travail.

La "voie" est le mécanisme et l'ensemble des informations qui permettent au système de savoir si la page est prête, donc si la tache est éligible (toutes ses pages sont prêtes), et de ré-ajuster le registre de base ; une tache comprend donc 3 voies de "page".

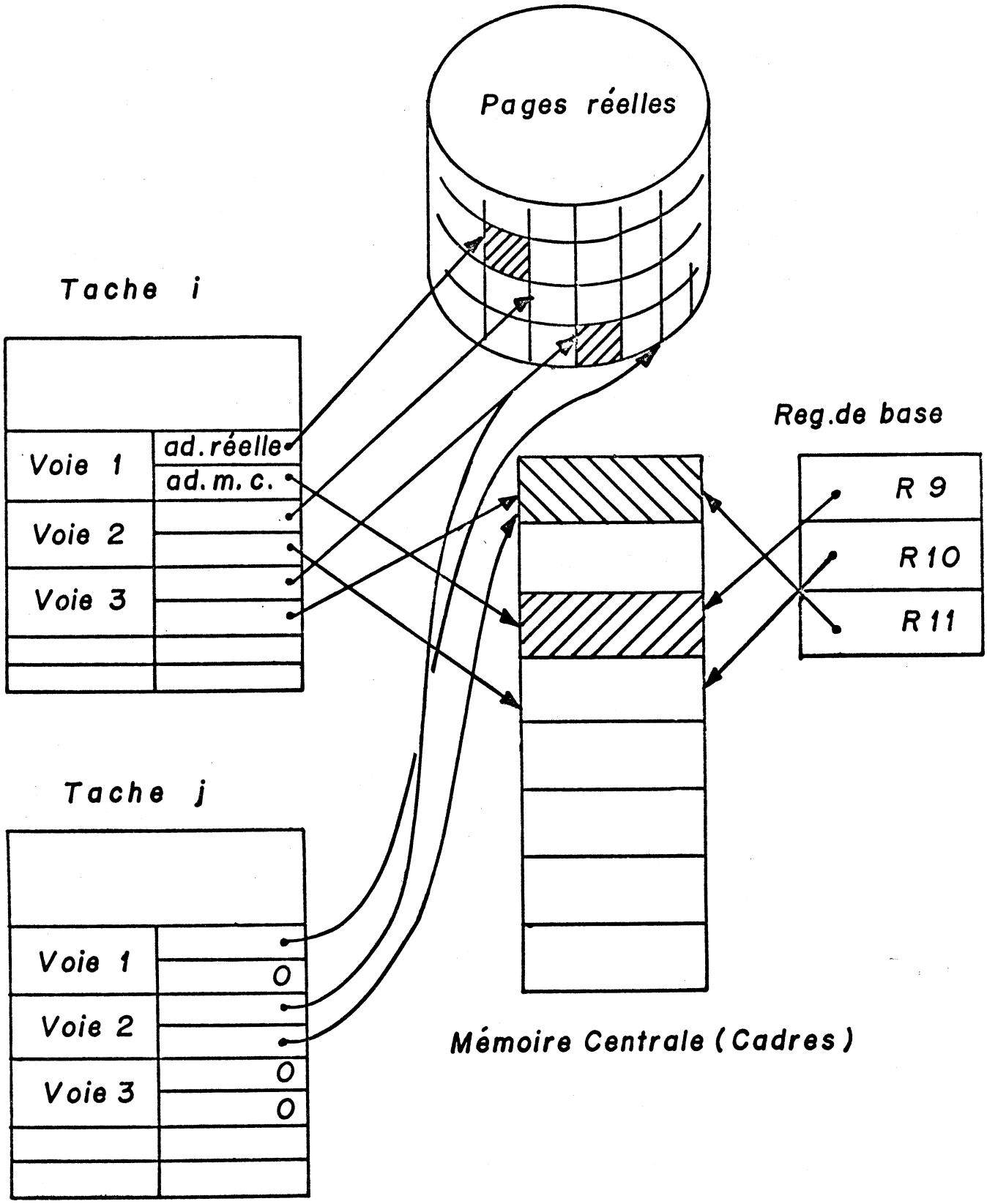


Fig : 2.2 - Taches, Voies,et Pages

(Cas où la tache i est active)

2.1.4.2. Défauts de pages

En l'absence de toute interruption "défaut de page" émise par hardware, la demande d'une nouvelle page doit être explicitée. Ainsi les programmes compilés, quand ils veulent brancher en dehors de la page courante, génèrent un appel au superviseur; celui-ci libère la page courante, et demande la page qu'il faut. De même un appel de sous-programme qui a besoin d'une nouvelle page de variable de travail lache l'ancienne, et demande la page suivante.

Pour cela, le superviseur utilise deux primitives de la machine de base : Amener et Lacher.

- Amener **(adresse virtuelle)** signifie : amener dans un cadre de mémoire centrale la page contenant l'adresse virtuelle demandée, et réajuster une variable de la voie, qui est le déplacement dans la page du mot situé à cette adresse virtuelle (dans le cas particulier où la taille de sous-page est égale à la taille de page, ce déplacement est toujours égal à 1 mot).

- Lacher (n° de voie) signifie que la tache n'a plus besoin de la page qu'elle avait demandé par cette voie. Elle peut donc être élue même si aucune page n'est présente par cette voie.

La tache apparaît alors comme disposant d'une mémoire virtuel le de trois pages; elle doit expliciter le couplage de ces pages avec des pages des espaces virtuels, et le faire évoluer dynamiquement selon ses besoins. (Analogie avec le "couplage" d'ESOPE [13])

2.1.4.3. Utilisation des primitives de couplage par le superviseur

Le superviseur écrit sur cette machine de base réalise des fonctions plus évoluées à l'aide de ces mécanismes. Ainsi :

Appel (appel de sous-programme) lache l'ancien contexte, et fait Amener (contexte suivant); lache la page programme et demande le nouveau programme.

Sortie (fin de sous-programme et retour) lache le contexte et le programme et réclame le contexte et le programme précédents.

Remarque : Les contextes et les programmes seront rangés dans des espaces virtuels dont la taille de sous-page est égale à la taille de page. Ainsi, après Amener, on peut adresser directement les 255 mots de la sous-page, tous présents dans la page.

Lire : (lecture dans l'espace virtuel "fichier") fait :

- Amener (adresse virtuelle)
- Recopier l'information
- Lacher cette page.

Interprétation de branchement : lache la page programme et en demande une nouvelle.

etc...

On voit ainsi comment l'utilisateur de cette "machine de base" peut travailler avec des "espaces virtuels", mais page par page.

2.1.5. Mécanismes intermédiaires

On verra au § 4 que cette primitive Amener réalise en fait plusieurs travaux :

- elle obtient un cadre de mémoire centrale,
- elle utilise une primitive de plus bas niveau :
  Demander (page disque) qui amène une page de l'espace secondaire dans un cadre. Elle peut alors aller voir le mot d'en-tête, et éventuellement lacher cette page pour en demander une autre.

Dans la mesure où l'on veut travailler non avec des espaces virtuels, mais directement sur l'espace disque, il est intéressant de pouvoir utiliser cette primitive Demander, c'est pourquoi nous l'avons aussi

implémentée comme primitive de la machine de base (ainsi les espaces "variables de travail et contextes" (TRAV) sont très limités : une pile ou un arbre par utilisateur actif ; ils pourront être gérés directement sur disque, sans repasser par les mécanismes des espaces virtuels, qui sont forcément plus lourds, parce que complètement software. Mais c'est là un problème d'optimisation).

De même il pourra être intéressant d'obtenir des buffers en mémoire centrale : cela signifie obtenir un certain nombre de cadres consécutifs. Les primitives "Prendbuf" et "Rendbuf" permettent cela, en rentrant dans le jeu de la gestion générale des cadres. On en verra l'emploi au § 3.2.

2.2. Gestion des ressources autres que la mémoire

2.2.1. Les processeurs logiques

Pour traiter le problème du partage des ressources du type Unité Centrale, Périphériques, ou Ressources "logiques", nous introduisons le concept de Processeur Logique, qui contient deux aspects :

- l'aspect "processeur" : c'est une capacité de fonctionnement, c'est lui qui sera "supporté", émulé par le hardware;
- c'est une ressource logique, un objet du système dont l'utilisation simultanée peut être limitée.

Ces "processeurs logiques" sont tout à fait distincts des "processus" ou "taches" (même si ceux-ci sont considérés comme des processeurs pour le niveau du dessus); nous décrivons ici une machine de niveau 0, dans laquelle les taches sont traitées, servies par ces "processeurs logiques". Taches et processeurs-logiques sont deux types d'objets distincts et complémentaires.

Nous entendons par processeur logique une "capacité", la possibilité de travailler d'une certaine façon. Il correspond à une ressource **limitée** : il peut donc provoquer l'attente des taches qui veulent travailler sur lui. Par exemple, on pourra considérer comme des processeurs-logiques :

- un pool de buffer, en nombre limité;
- un programme non réentrant, utilisable par une seule tache à la fois ;
- un périphérique qui ne peut réaliser qu'une entrée/sortie à la fois ;
- des "sections critiques" : le processeur associé étant le seul capable de faire traverser de telles sections.

On considèrera en plus un processeur "banal", nommé CPU, qui n'est jamais limité et ne peut provoquer aucune attente : c'est le fonctionnement qui ne requiert pas d'autre ressource que l'unité centrale.

Les ressources logiques associées à de tels processeurs sont des ressources non "préemptibles" : une fois que la tache est active sur un processeur logique, celui-ci ne peut lui être enlevé ; par contre il peut ne pas être toujours actif sur le hardware.

Ces ressources sont caractérisées par un certain "parallèlisme possible" ou "degré de multiprogrammation" DM : c'est le nombre de taches qui peuvent être simultanément actives sur un tel processeur. Ainsi, un programme non réentrant, ou une section critique auront DM = 1; un pool de 5 buffers aura DM = 5; le processeur "CPU" aura DM infini.

2.2.2. Coopération entre ces processeurs logiques

Notre machine de base fonctionne comme un système multiprocesseur, formé d'un certain nombre d'unités centrales accèdant à une mémoire centrale commune :

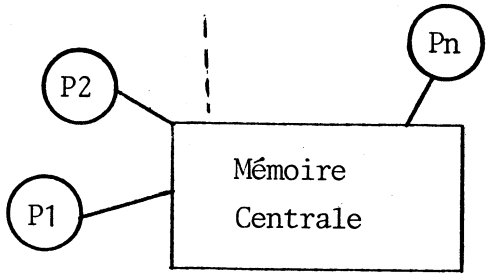


Figure 2.3. : Système multiprocesseur "théorique"

Chaque processeur dispose de files d'attente de taches à exécuter, en mémoire centrale. Ils possèdent un jeu d'instructions comportant des primitives qui leur permettent :

- d'envoyer une tache dans la file d'attente d'un autre processeur ;
- de retourner une tache au processeur qui la leur a envoyée ;
- d'explorer leur file d'attente et de démarrer une tache qui est "éligible", c'est-à-dire dont les demandes de pages ont été satisfaites (ce qui existe déjà en [48])

Cette exploration n'est faite que si le processeur en est "logiquement capable", donc si le nombre de taches qu'il active déjà est inférieur à son degré de multiprogrammation.

Ces actions sont déclenchées lorsque le processeur exécute les primitives Appel et Sortie pour le compte d'une tache.

Appel : soit la tache T1, active sur un processeur P1, qui déroule pour elle un certain programme PG1 ; en exécutant l'instruction :

Appel (processeur P2, programme PG2)

le processeur P1 :

- empile le contexte interrompu sur les variables de travail de T1, éventuellement change de page de contexte,
- met T1 dans la file de P2, et le lui signale (de façon que P2 puisse explorer cette file s'il le peut).

Sortie : assure de façon inverse le retour de la tache de P2 sur P1, la reprise de l'ancien contexte, et signale cela à P1.

En fait, chaque processeur dispose de deux files :

FILRET est la file des taches actives sur le processeur ;

FILTRAV est celle des taches qui demandent le processeur ;

Une tache ne passe de FILTRAV à FILRET que si le processeur est disponible logiquement.

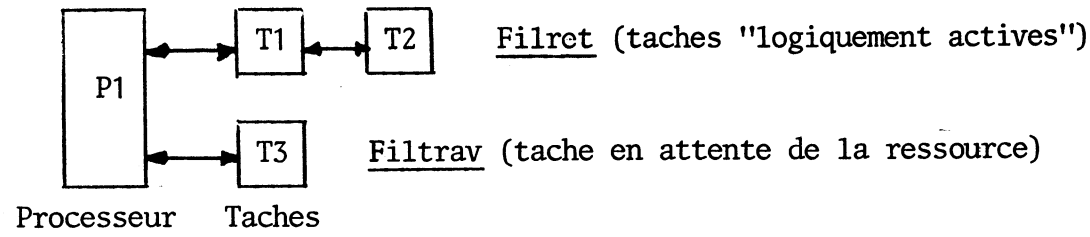


FIGURE 2.4. : Files d'attente d'un processeur-logique

On appellera séquence les programmes ou suites d'instructions écrites à ce niveau, et que l'on demande à un processeur d'exécuter pour le compte d'une tache. Elles peuvent être écrites de différentes façons : paginées, résidentes ou microprogrammées; ainsi les séquences de lecture/écriture associées à tel type de périphériques seront utilisées par des programmeurs

ne les connaissent pas, et faisant :

Appel (Processeur-console, Ecrire 2741, paramètres)

dans le cas, par exemple, d'une écriture sur un terminal type IBM 2741. Ainsi, il a la même facilité que s'il écrivait une instruction telle que DOIO [51], mais il reste conscient du fait que sa tache va attendre que le périphérique soit libre, puis s'exécutera dessus ; que ce travail requiert une ressource supplémentaire du système.

2.2.3. Activité physique des processeurs

Le fonctionnement des processeurs logiques est assuré par le partage des ressources physiques (Unités Centrales et périphériques) de la machine.

Le partage des unités centrales est assuré par le Régulateur qui sera détaillé au § 4.1.; les processeurs-logiques sont candidats ou non à ce partage, ceux qui n'ont rien à faire ne sont pas demandeurs. Les primitives Appel et Sortie peuvent avoir pour effet de "réveiller" un processeur, c'est-à-dire de le rendre demandeur d'unité centrale, pour qu'il puisse explorer sa file d'attente.

Le passage d'une unité centrale à un canal ou sous-canal est réalisé, sur les ordinateurs classiques, par le mécanisme de "changement de hardware"":

- exécution de l'instruction d'entrée-sortie (SIO) et mise en attente.
- interruption et "réveil" du processeur.

Il est évident que dans un système physique où les canaux sauraient exécuter les fonctions d'exploration de file d'attente, et sortie ainsi que certains petits travaux, ce mécanisme serait superflu.

Notre "machine de base" (niveau 0) est conçue dans cet esprit les périphériques étant des processeurs comme les autres.

Cependant, dans la réalisation de cette machine sur un ordinateur actuel, nous sommes obligés de mobiliser de l'unité centrale pour faire ces travaux. Le processeur-logique associé au canal est donc candidat à l'unité centrale; celle-ci initialise le travail, puis, par l'exécution de l'instruction : SIO-HASLP

(SIO, et primitive HASLP qui "endort" le processeur)

assure le "changement de hardware" en activant le canal et en "endormant" le processeur logique, qui n'est alors plus candidat à l'unité centrale. L'interruption le "réveille", et il la réclamera pour assurer le reste du travail, en particulier Sortie.

C'est ainsi, à l'aide de SIO, HASLP et Sortie que sont programmées les séquences d'entrée/sortie telles que Ecrire 2741 ; une fois celles-ci réalisées, tout ce mécanisme peut être complètement ignoré du programmeur, ainsi que le concept d'interruption ([16], [51]).

Notons aussi pour terminer que le fait de réveiller ou d'endormir un processeur modifie l'aspect des demandes d'unité centrale, et doit donc être suivie d'une réaffectation des unités aux demandeurs (Régulateurs) : c'est là le vrai sens de l'interruption; et elle peut alors être déclenchée par appel ou sortie ("interruption interne" ou "software" : [14], [48])

2.2.4. Parallélisme et activité logiques des "processeurs-logiques"

Un processeur qui a un degré de multiprogrammation DM supérieur à 1 peut "faire du parallélisme", c'est-à-dire activer simultanément plusieurs taches.

Dans le cas de plusieurs unités centrales, un même processeur logique peut donc être actif simultanément sur plusieurs unités : cela est réglé au niveau du régulateur.

Inversement, une tache peut avoir besoin de plusieurs processeurs logiques pour s'exécuter : une tache active sur un processeur logique "programme non réentrant" peut avoir besoin, en plus, d'un autre processeur associé à des buffers, par exemple. Pour cela elle a fait d'abord un appel au premier, puis, sur celui-ci, un appel au second.

Suivant ce mécanisme, une tache reste "logiquement active" sur le premier, tout en attendant ou en utilisant le second. On peut dire qu'elle se divise en deux sous-taches, la première partant travailler avec le processeur demandé, la seconde restant active sur celui d'origine, mais ne faisant qu'attendre que la première revienne (Primitives fork et join [16], [23]; **cela** a été montré en [9]). Cependant, notre implémentation se contente de déplacer la tache de file, en ayant incrémenté le compteur D "nombre de taches logiquement actives" sur le processeur d'origine.

Une tache peut alors demander successivement plusieurs ressources ; mais ce mécanisme la contraint à les rendre dans l'ordre inverse des demandes : une tache ne peut qu'empiler des ressources :

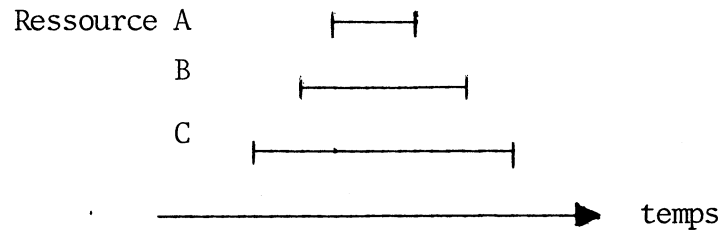


Figure 2.5. : Diagramme d'utilisation des ressources par une tache suivant Appel/sortie

Autrement dit, selon l'exemple précédent, la tache est forcée de libérer A avant B, et B avant C. Ceci s'est avéré trop restrictif pour nous permettre de construire le superviseur que nous voulions ; un exemple précis dans lequel il nous fallait libérer C avant de libérer B a été détaillé en [8], § 2.2.9.

Nous avons donc introduit deux autres primitives de demande et restitution de ressources, dont l'emploi est plus souple :

"prendre une ressource i"

"rendre la ressource i"

nommées P(i) et V(i), car elles fonctionnent comme les opérations de DIJKSTRA sur les sémaphores [25]. Si l'on considère les processeurs-logiques comme des sémaphores, ces opérations permettent à plusieurs taches de se synchroniser. Notons cependant que ces processeurs sont en nombre fixe et connus de toutes les taches : nous n'avons pas de sémaphores "locaux". Cela permet donc certaines synchronisations, comme celles d'un producteur et d'un consommateur ([25]), mais pas celles de sous-taches qui seraient créées dynamiquement (ce que notre système ne permet d'ailleurs pas).

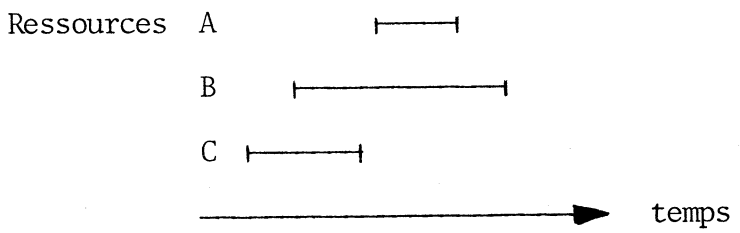


Figure 2.6. : Diagramme d'utilisation possible avec P et V

Ces opérations "P" et "V" sur les ressources sont plus souples, mais amènent à construire des systèmes dont le bon fonctionnement devient très difficile à prouver. Nous utiliserons donc Appel et Sortie dans toute la mesure du possible.

Certains systèmes ont été proposés avec comme seuls outils de synchronisation les sémaphores considérés comme des évènements ("wait", "signal") et des sections critiques ("critical v do S") [34, 35] ; si l'on considère les ressources comme des sections critiques, ce mécanisme permet seulement une allocation "imbriquée" semblable à celle de Appel et Sortie (figure 2.5).

(Si l'on veut permettre à une tache de créer, puis de stopper ou de tuer des taches-filles, il faut pouvoir les forcer à rendre les ressources qu'elles utilisent; ce problème difficile se simplifie si l'allocation des ressources est du type "imbriqué" (cf. HANSEN [34]))

Réalisation de P et V (cf. [8] 2.2.9)

Sur les processeurs logiques munis de file d'attente, le travail de P et V est :

- P (processeur) : Appel (processeur, sortie spéciale)
  la séquence Sortiespéciale faisant Sortie,
  mais sans décrémenter le compteur des taches actives sur le processeur.

- V (processeur) : Décrémente ce compteur, et réveille éventuellement le processeur.

Extension de Appel, P et V

Ces primitives opèrent sur des processeurs logiques qui tolèrent un parallélisme DM ; DM n'est pas forcément égal à 1, il peut être de N dans le cas d'une ressource qui tolère N utilisateurs à la fois (Ex : Table de N entrées).

Il est alors intéressant de permettre aux taches de moduler leurs demandes, c'est-à-dire de réclamer $n \leqslant DM$ parallélismes à la fois.

Les primitives Appel, P et V comportent alors un paramètre de plus : n.

De tels mécanismes sont connus pour risquer de provoquer des attentes illimitées ("Parallel P-Operations" [27]). Cependant, dans le cas d'une seule ressource quantifiée, et si la politique de gestion des files d'attente est FIFO (cf. § 4.2.) cette objection est levée.

On verra que cette extension permet de traiter très simplement l'exclusion lecture / mise à jour sur des éléments de fichiers partagés (cf § 3.3.3).

2.2.5. Configuration logique

Pour l'utilisateur de la machine de base, il n'existe plus qu'un jeu de processeurs logiques cohérents, capables de se donner mutuellement du travail.

La première étape de la construction du superviseur sera donc de définir la configuration logique, c'est-à-dire les processeurs logiques, leur DM, et de les initialiser. On associera en particulier un processeur de DM = 1 avec chaque périphérique (cf. § 3.2).

Cela permet de n'avoir qu'un seul mécanisme d'attente sur des ressources (autres que la mémoire), et d'ignorer les problèmes d'allocation de l'unité centrale. Ce mécanisme peut être utilisé de deux façons : Appel/sortie ou P/V - Appel/Sortie contraignent à une utilisation plus stricte, plus structurée des ressources, mais ne permettent pas de tout faire ; P/V sont d'un emploi plus souple, mais plus dangereux.

Cela permet aussi, en programmant de façon convenable les routines d'entrée/sortie associées aux périphériques, de ne voir l'extérieur du système que sous le jour de ces processeurs-logiques périphériques. Ainsi, on peut régler les problèmes de communication avec un réseau, ou avec une autre partition d'un système d'exploitation (dans le cas restreint où notre système tournerait dans ces conditions) : ces problèmes peuvent être réglés dans les séquences associées, SOCRATE ne connaissant toujours que son univers de "processeurs logiques" (cf. [8], § 7.2 et 7.3).

2.3. Organisation des taches

2.3.1. Taches : rappels

On a défini la tache comme représentant un programme en activité pour le compte d'un utilisateur ; elle peut donc être décrite par son vecteur d'état (registres et mot d'état), ainsi que par les espaces virtuels sur lesquels elle travaille.

Il est associé à la tache une "mémoire virtuelle" de 3 pages; à chacune de ces pages peut être "couplée", à la demande du programme, une page de l'espace virtuel. La tache peut alors adresser directement ces pages, des registres de base étant ajustés par le système.

Le "bloc de contrôle" d'une tache en mémoire centrale est donc quelque chose qui comprend des informations relatives aux espaces virtuels qu'elle peut utiliser, et un certain nombre de voies. Ces voies contiennent ce qui est nécessaire au système pour qu'il puisse assurer la présence des pages virtuelles demandées, et noter "non éligible" une tache dont une page disparaît. Ces informations sont donc forcément résidentes en mémoire centrale, puisqu'elles sont nécessaires à la présence de pages virtuelles en mémoire.

2.3.2. Création dynamique de taches

La plupart des systèmes fournissent des primitives qui permettent à une tache de créer, de démarrer, de se synchroniser avec d'autres taches ("taches filles"). ([16], [23], [26], [33], [51]). Au démarrage, un utilisateur est généralement associé à une tache; puis il peut générer toute une famille de taches. Cela présente deux avantages fondamentaux :

a - Cela permet un parallélisme réel pour le compte d'un utilisateur, pendant des entrées/sorties par exemple.

b - La possibilité de faire travailler plusieurs programmes en "coroutines" pour un seul utilisateur permet de faire des choses plus puissantes que le simple appel de sous-programme ; ainsi, en SOCRATE, on aimerait pouvoir travailler en coroutine avec une tache qui, par exemple, énumérerait toute une série de caractéristiques d'un fichier (cf. § 2.3.4).

Par contre, il apparait très dangereux de laisser les utilisateurs créer des taches de façon incontrôlable : ces taches nécessitent des blocs de contrôle résidents et assez importants ; si une saturation de la table de ces blocs intervient, que faut-il faire ? (Même si la demande est protégée par l'emploi d'un P, on risque un blocage ("deadlock")). La perspective d'être obligé de tuer un ou des utilisateurs qui sont en train de manipuler des espaces virtuels partagés avec d'autres n'est pas raisonnable, l'intégrité des bases de données risquant d'être compromise.

C'est pourquoi nous avons construit un système dans lequel aucune création dynamique de tache n'est possible. Toutes celles-ci sont définies à la génération du système, et on ne peut en rajouter d'autres dynamiquement. Cela paraît très restrictif ; en fait on va voir que les deux avantages cités plus haut peuvent être regagnés autrement, au moins en partie.

2.3.3. Parallélisme effectif

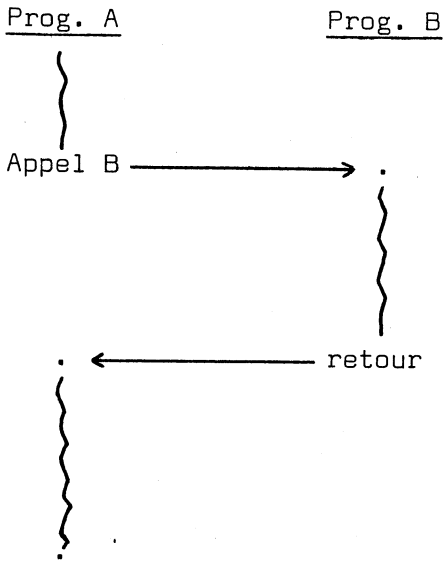
Dans un système de multiprogrammation tel que le nôtre, il n'apparaît pas indispensable de faire du parallélisme effectif pour le compte d'un utilisateur. De plus, pour le cas des entrées/sorties sur imprimantes, lecteur/perforateur de cartes, des techniques de spooling permettent en fait ce parallélisme. Restent les bandes, et la console de l'utilisateur, en sortie : il faut attendre la fin des tranferts ; des techniques de blocage peuvent alors réduire ces inconvénients. Si cela ne suffit pas, la définition d'une tache associée à chaque périphérique permettra toujours tout le parallélisme souhaité.

2.3.4. Coroutines

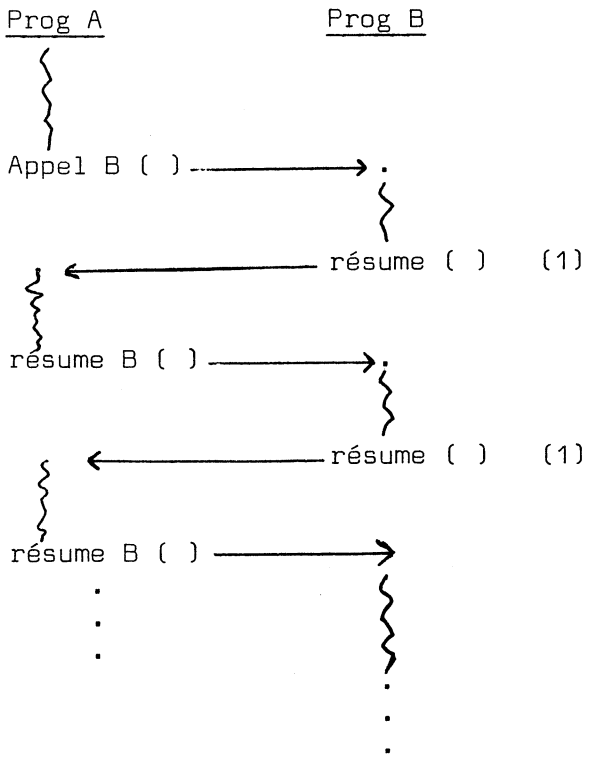
Le contexte d'un programme Socrate est l'ensemble des variables locales à celui-ci : il comprend la pile des variables allouées dans chaque bloc, les variables "globales" du programme, et le vecteur d'état à l'instant t.

Jusqu'ici nous avons considéré qu'une tache pouvait faire des appels de sous-programme : le contexte interrompu par cet appel est alors empilé dans la pile des contextes de la tache, située dans un espace virtuel (TRAV), et on initialise un nouveau contexte. Au retour du sous-programme, on récupère le contexte précédent : c'est l'appel procédural.

Dans ce cas, le transfert de contrôle est simple : du programme appelant A vers le sous-programme B, puis retour à A.



Mais nous voulons permettre des transferts de contrôle plus complexes, le programme A pouvant initialiser B, puis le ré-invoquer plusieurs fois pour lui demander une valeur.



Ainsi, si B est un programme capable d'énumérer un certain nombre d'objets qu'il répond en paramètre du résume, A pourra travailler successivement avec tous ces objets. On voit qu'en (1) le contexte de B ne doit pas être perdu, mais rangé pour être repris par la suite, pour que B donne la réponse suivante : A travaille avec B comme il travaillerait avec une bande (dont le contexte est réduit à sa position sur le dérouleur). Nous dirons que A et B travaillent "en coroutines".

Et de la même façon qu'un sous-programme peut à son tour lancer un autre sous-programme, B peut initialiser et travailler de cette façon avec C, D, etc.... Le mécanisme peut être récursif. De plus, un programme peut travailler avec plusieurs coroutines à la fois (comme il travaille avec plusieurs bandes).

On a vu qu'une tache faisant des appels procéduraux dispose d'une pile des contextes interrompus. Pour travailler "en coroutine", il faut que les contextes des participants soient rangés, et que l'on sache les retrouver quand on veut redémarrer B, par exemple. Cela revient à transformer la pile des contextes en un arbre, l'adresse des contextes "fils" étant notés dans le contexte "père".

Pour préciser cela, supposons que l'utilisateur ait à sa disposition, avec ses variables locales Xi, Yi, Zi, des variables locales CORTNi "coroutine i". Lorsqu'il programme un APPEL, il précise :

APPEL (Processeur, Séquence, CORTN1)

ce qui initialise un contexte, nommé CORTN1 par le programme initial. Le contrôle est alors passé à ce contexte exécutant la séquence appelée; celle ci peut :

- soit rendre définitivement le contrôle, en tuant son contexte, par SORTIE (comme précédemment),
- soit rendre le contrôle à celui qui l'a appelé, mais en conservant son contexte, par un

RESUME (Contexte père)

- soit initialiser et passer le contrôle à son tour à d'autres coroutines.

Lorsque le père a repris le contrôle sur un RESUME fait par son fils, il sait que CORTN1 existe toujours et peut lui rendre le contrôle par

RESUME (CORTN1).

Une séquence peut évidemment travailler ainsi en coroutine simultanément avec plusieurs autres. On peut ainsi avoir un arbre de contextes vivants pour une tache :

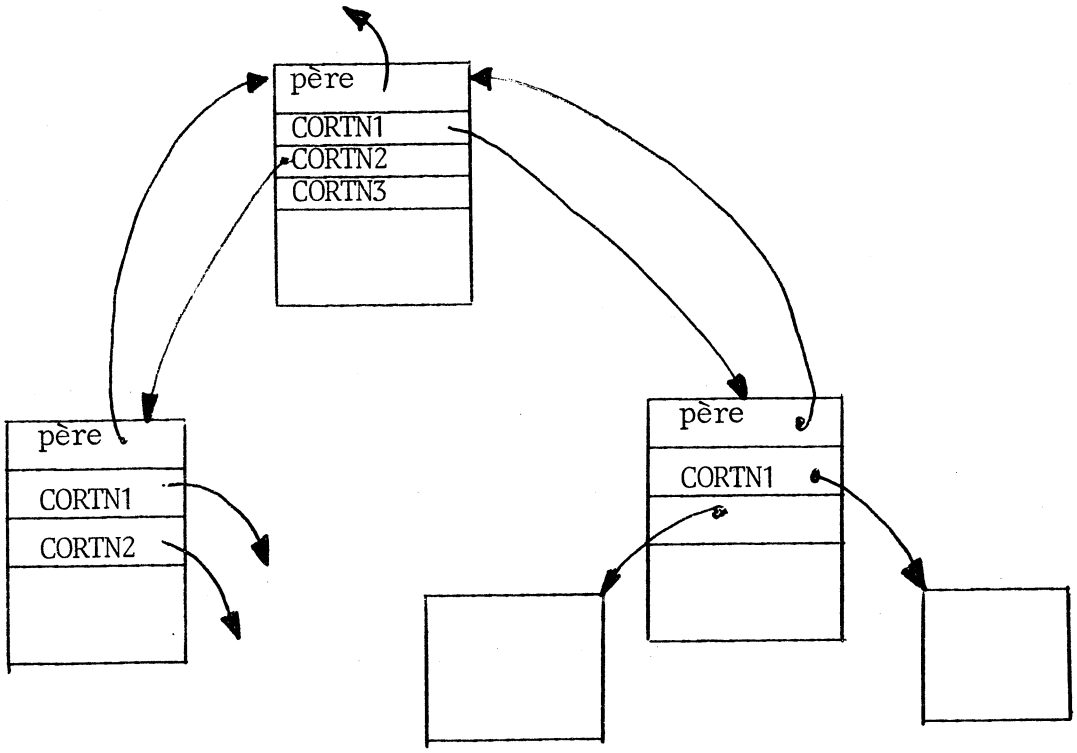


Figure 2.7. : Arbre des contextes d'une tâche

Le problème de l'allocation de ces contextes dans l'espace virtuel TRAV ou sur disque est facile à résoudre (utilisation d'une liste des pages libres, par exemple).

Il est intéressant de noter que, tels que les transferts de contrôle sont décrits, un seul contexte de l'arbre est actif à la fois, donc utilise les voies de la tache. En effet, au moment de APPEL, SORTIE ou RESUME, les pages demandées sur les voies 1 (TRAV = contexte) et 2 (Programme) sont changées par le système; et par convention rien n'est demandé sur la voie 3 lors de l'exécution d'une de ces primitives (celle-ci sert aussi à assurer la présence momentanée de deux pages de contexte pour recopie des paramètres).

En d'autres termes, nous réalisons un mécanisme de coroutines, mais **sans** aucun parallélisme (et nous n'avons donc pas besoin de primitives ni d'évènements de synchronisation).

Ce mécanisme nous est indispensable si l'on veut que certaines relations ne soient plus enregistrées mais calculées : par exemple, si sur un ensemble de PERSONNE la relation PERE est enregistrée, la relation FILS peut être calculée (donner lieu à un balayage du fichier); l'utilisation dans des citation de

... TOUT FILS DE X

donnera lieu à un travail en coroutine avec une séquence charger d'énumérer les fils de X. Et si l'on veut définir la relation DESCENDANT, il faut savoir initialiser des coroutines récursivement.

2.3.5. Bases, taches et contextes

DENNIS et VAN HORN [23] distinguent trois types d'objets actifs dans un système : ce sont les :

- "processes"
- "computation"
- "principal"

Une "computation" recouvre un ensemble de "processes" travaillant avec les mêmes privilèges (C-list) pour le compte d'un utilisateur ; cela correspond bien à nos taches, et les "processes" à nos contextes; toutefois chez nous un seul contexte par tache est actif simultanément (ce qui nous l'avons vu parait assez peu restrictif, et simplifie beaucoup la réalisation et l'utilisation de ces contextes).

Un "principal" est une famille d'utilisateurs travaillant dans un but commun; cela correspond à la définition d'une "Base de Données".

2.3.6. Organisations possibles des taches dans le superviseur

Les taches sont en nombre fixe, à prévoir à la génération du système, comme les processeurs logiques. Il reste cependant plusieurs façons de construire un système :

- soit en organisant "horizontalement" un grand nombre de taches spécialisées, qui déroulent des algorithmes précis; elles communiquent à l'aide de "boites aux lettres" situées dans un espace virtuel accessible par tous, et se synchronisent par des P et V sur des sémaphores prévus eux aussi statiquement.

- soit en lançant un petit nombre de taches, par exemple une par utilisateur; celles-ci font alors des appels et empilent des contextes, pour exécuter d'autres séquences.

Exemple :

Pour réaliser une Entrée/Sortie sur un terminal conversationnel, on peut employer l'une ou l'autre méthode :

- *Dans la première, on construit une tache associée au périphérique, qui déroule l'algorithme :*

```
(1) | α : P(sémaphore)     /* attente */
    |     Consulter boîte aux lettres
    |     Lecture ou Ecriture
    |           (attente de fin E/S)
    |     V(sémaphore1)
    |     allera α
```

En dehors de toute Entrée/Sortie, cette tache attend en α. Par ailleurs, on a défini une tache "utilisateur" qui demande une Entrée/Sortie par :

```
(2) |     Garnir boîte aux lettres
    |     V(sémaphore)    /* active1 */
    |     P(sémaphore1)   /* attente fin E/S */
```

Elle se contente de débloquer la tache (1) qui attendait. Si on a un périphérique par utilisateur, cette méthode nécessite deux taches pour chacun.

- *Dans la seconde, on définit une seule tache, qui fait :*

Appel (Processeur-périph., Lire/Ecrire)

Ce faisant, elle empile un contexte : c'est un appel procédural.

- toutes les organisations mixtes sont possibles; nous avons opté plutôt pour la seconde; d'autres systèmes, tel ESOPE [29] utilisent plutôt la première.

Remarquons que dans le premier cas, une ressource peut être implicitement associée à la tache qui déroule un algorithme : s'il existe une tache "writer", par exemple, qui déroule une séquence d'écriture qui est la seule à utiliser l'imprimante, on sait que l'imprimante ne sera utilisée que par une tache à la fois ! Tandis qu'en organisation verticale, il faut se protéger explicitement, par l'emploi d'un processeur logique par exemple. Dans le premier cas, on fera V (sémaphore) pour lancer la tâche qui va exécuter la séquence, donc pour utiliser la ressource; dans le second on fera

```
P(sémaphore)
séquence
V(sémaphore)
```

On ne peut donc pas associer une signification "prendre" ou "rendre" aux opérations P et V : cela est fonction du type d'organisation dans lequel on les utilise.

Remarquons encore que dans le premier cas (V), on n'attend pas, on signale seulement quelque chose; si on veut s'assurer de sa réalisation, on émet un P(sémaphore1).Mais entre temps, on peut avoir deux taches travaillant pour la même cause, ce qui n'est pas vrai dans l'autre organisation.

2.4. Conclusion : taches et synchronisations

La plupart des systèmes donnent à l'utilisateur la possibilité de programmer un certain parallèlisme quelquefois en langage évolué ([11], [16], [33], [35], [39], [51]). Au départ, cet utilisateur est séquentiel ; il peut alors créer des taches, et les synchroniser entre elles, puis les tuer : le parallèlisme passe alors nécessairement par la création dynamique de taches avec les inconvénients que l'on sait.

Une autre solution est de donner à chaque utilisateur un parallélisme statique, c'est-à-dire de lui attribuer un nombre fixe de taches. Dans de tels systèmes, on spécialise généralement ces taches (à des entrées/sorties, par exemple), et on ne fait pas de calcul en parallèle. En fait, on pourrait tolérer un certain parallèlisme dans l'exécution des programmes de l'utilisateur, **mais limité**.

Si l'on a création dynamique de taches, il faut savoir les synchroniser par des objets locaux : on conçoit alors les sémaphores ou évènements comme propres à la tache, et cela consolide la distinction entre objets de synchronisation (macros POST et WAIT) et ressources (ENQUEUE et DEQUEUE de OS 360 [39]).

Si le nombre de taches est fixe, ces objets de synchronisation peuvent eux aussi être statiques, et la différence entre ressources (au sens d'une organisation verticale), et sémaphores devient uniquement sémantique; on peut alors implémenter les opérations P et V [25] d'une seule façon. Un

seul mécanisme est nécessaire à la construction du système.

Mais des mécanismes plus élaborés ([33], [35]), ou plus restrictifs peuvent être utiles pour faciliter la construction du superviseur, et s'assurer plus aisément de son bon fonctionnement : c'est pourquoi nous conservons Appel et Sortie

CHAPITRE 3

3. FONCTIONNEMENT DE LA BANQUE SUR CETTE MACHINE DE BASE

3.1. Le superviseur

Pour préciser les idées, nous présentons ici très succintement la composition et le fonctionnement du superviseur; nous en détaillerons plus loin certains aspects particuliers.

3.1.1. Bases et utilisateurs

Rappelons que la Banque de Données se propose de gérer un certain nombre de Bases : chacune d'elle est un ensemble d'informations structurées, connectées ("on line"), et de programmes qui les traitent. Une base est utilisée simultanément par un certain nombre de personnes, en traitement par lot ("batch"), ou de façon conversationnelle.

Chaque utilisateur à droit à des structures privées, des données ou "fichiers" et des programmes privés. Ceux-ci sont rangés dans les espaces virtuels de la base (cf. [8], chapitre 4).

La "Base des Bases" regroupe l'ensemble des informations concernant les bases de données : implantation sur disque, noms des utilisateurs possibles, facturation, etc ... La gestion du système est ainsi faite par lui-même, un certains nombre de travaux pouvant être programmés en langage SOCRATE (par exemple la définition d'un utilisateur, la facturation, ...). On appellera "programmes de la BDB" toute cette catégorie de composants du système.

De plus, un "espace virtuel système" regroupe une autre catégorie de programmes du système non résidents : il s'agit des compilateurs, programmes de services, etc...

3.1.2. Définition des processeurs, des taches et des contextes

La première étape de la construction du superviseur consiste à définir les processeurs logiques et les taches.

3.1.2.1. Taches

On a défini essentiellement :

- une tache "opérateur" travaillant de façon privilégiée avec la BDB
- un certain nombre de taches "utilisateur", initialisées par l'opérateur, qui précise l'attachement du "standard d'entrée" de ces taches avec les terminaux conversationnels, le lecteur de cartes ou des bandes (cf. § 3.2.2.).
- quelques taches "de service" : videur de "spool" sur l'imprimante, lecteur de fichiers cartes, éventuellement "garbage collector" des références non nettoyées, etc...

Cette organisation de taches est essentiellement verticale, les utilisateurs vont dérouler des algorithmes séquentiels (§ 2.3.6). Les rares communications entre taches utilisent des processeurs définis pour cela, et des "boites à lettres" dans l'espace virtuel.

3.1.2.2. Processeurs

En conséquence, tout travail avec une ressource doit être protégé par un processeur logique; on a défini :

- un processeur par périphérique de la configuration physique (même pour ceux qui ne sont utilisés que par une seule tache, comme l'imprimante, pour faire disparaître la notion d'interruption) ;
- un processeur "CPU", capable de tout (DM infini) ;
- un processeur "PL1" pour activer certains composants encore écrits dans ce langage et non réentrants;
- des processeurs de synchronisation avec les "spools" ou le "garbage collector" ;
- un processeur "place de cadre" destiné à limiter le nombre de taches se partageant les cadres de la mémoire centrale ;
- des processeurs "verrou" qui assureront l'indivisibilité de certains travaux sur les bases (cf. § 3.3.2), et un processeur "place de verrou" ;
- etc ...

3.1.2.3. Contextes

Un programme compilé a, pour s'exécuter, besoin actuellement d'une page de contexte et de travail; celle-ci est donc réclamée lors d'un appel de sous-programme.

L'accès à de nouvelles pages, la création et la suppression de celles-ci sont donc fréquentes dans le système; si ces pages de TRAV se trouvent dans un espace virtuel (à grandes sous-pages), d'éventuelles collisions, et le déplacement de sous-pages occupant une place qu'il faut libérer ("squatters") prend beaucoup de temps et nécessite souvent plus d'un accès disque pour amener une page de TRAV.

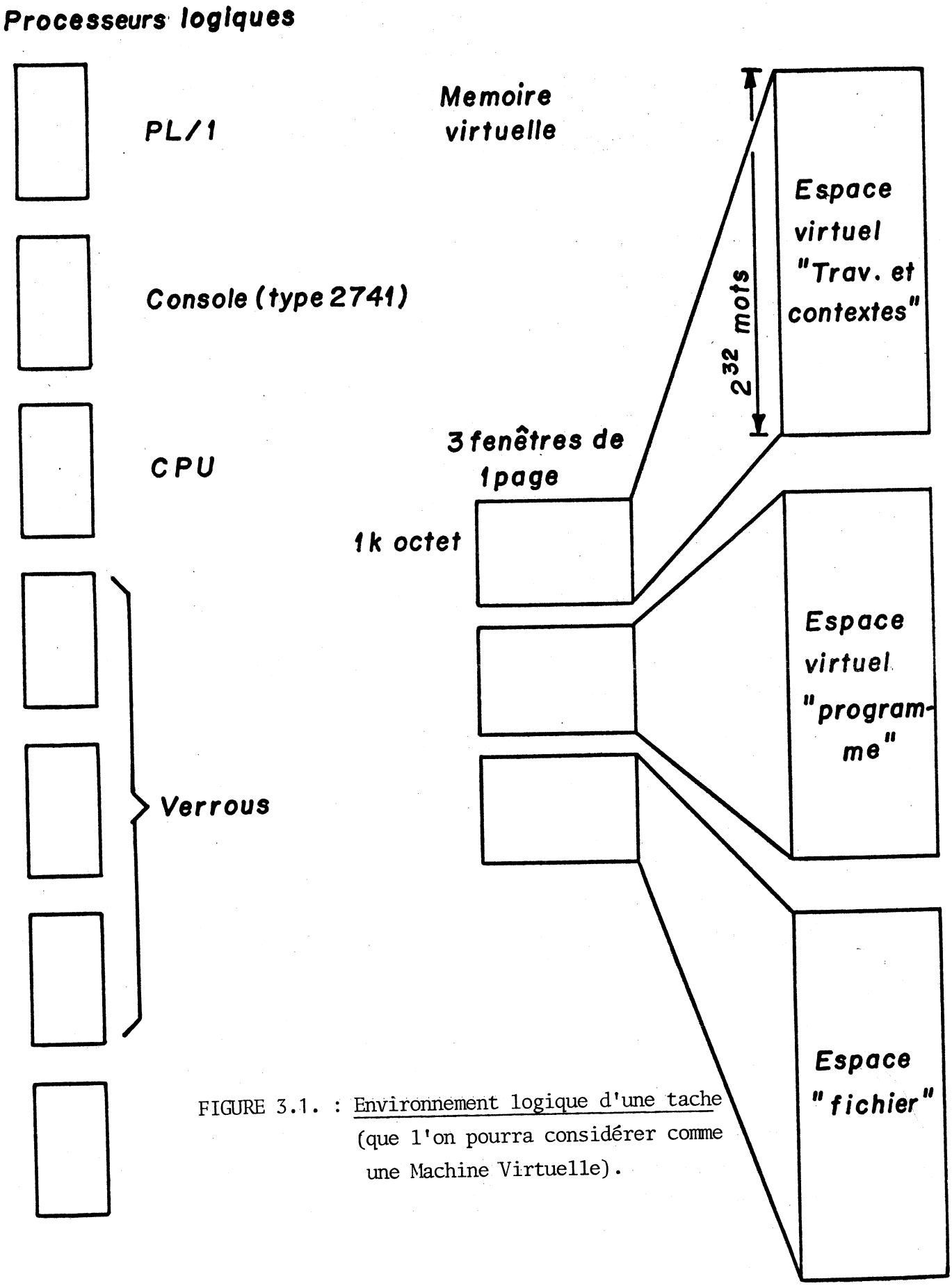


FIGURE 3.1. : Environnement logique d'une tache (que l'on pourra considérer comme une Machine Virtuelle).

On a donc préféré gérer directement ces pages sur l'espace disque, et utiliser la primitive DEMANDER (§ 2.1.5.); le nombre de ces pages est réduit (en moyenne une dizaine par tache), et on tient à jour en mémoire centrale une liste des pages-disque de TRAV libres.

De plus, on évite d'amener la nouvelle page depuis le disque lors d'un appel (le nouveau contexte n'a pas à être initialisé), et de l'y renvoyer lors de sortie, sauf si l'utilisateur travaille en mode "conservation des variables locales".

3.1.3. Algorithmes déroulés pour un utilisateur par la tache qui lui est associée.

Ils se composent schématiquement de :

- une "routine système" qui attend le "login" de l'utilisateur, le valide, met à jour la BDB, et lui permet de préciser ses "attachements"; cela est fait en majorité par des appels à des programmes de la BDB; puis elle lance la "routine utilisateur".
- la "routine utilisateur" demande l'introduction d'une requête SOCRATE (programme de requêtes terminé par '?', ou nom d'un programme à exécuter); elle lance soit la compilation du programme, et le catalogue, ou bien lance l'exécution; après quoi recommence, ou rend le contrôle à la "routine système".

L'emploi de la touche "attention" permet de forcer un appel à la routine système, avec préservation du contexte; on peut alors re-préciser des attachements, et continuer, ou bien relancer une autre série de programmes, avant de revenir terminer ceux qui s'éxécutaient.

3.1.4. "Appels superviseurs" par des programmes compilés

Le superviseur réalise de plus un certain nombre de fonctions qui pourront être demandées par les programmes; ce sont, par exemple :

- LIRE et ECRIRE (buffer, adresse virtuelle, application) ; réalisés à l'aide de AMENER et LACHER ;
- Appel de sous-programme paginé ;

  (Un tel programme est appelé par son adresse virtuelle; celle-ci est un nom, et ne change pas durant toute la vie du programme; les liens sont donc mis en place à la compilation, suivant la technique des références, et aucun "link édit" n'est donc nécessaire à l'exécution. Cela a déjà été l'un des objectifs de MULTICS [18]).

- Interprétation de branchement ;
- Entrée/Sortie sur un "standard" auquel est attaché un périphérique (cf. § 3.2.2) ;
- Verrouillage d'une réalisation d'entité (cf. § 3.3.3),
- etc...

3.2. Périphériques

Deux aspects de la programmation et de l'utilisation des périphériques sont intéressants à détailler ici : le travail de ceux-ci dans les espaces virtuels, et l'attachement des "standards" de l'utilisateur à des périphériques réels.

3.2.1. Travail des périphériques dans l'espace virtuel

Dans les systèmes qui génèrent des "mémoires virtuelles" pour des machines virtuelles, un des problèmes est de faire travailler les canaux sur ces mémoires virtuelles; cela peut être réalisé en amenant d'abord les pages intéressées en mémoire centrale, puis en interprétant le programme canal, ou bien en lui faisant adresser la mémoire centrale avec une indirection, en repassant par une table où sont notées les adresses en mémoire centrale des pages virtuelles intéressées ([40], [42]); en effet, le mécanisme d'adressage habituel ne peut être utilisé : un périphérique (par exemple un dérouleur) ne saurait être arrêté parce que le canal a fait une faute de page !

Dans l'écriture des séquences d'entrée-sortie associées aux périphériques, on peut imaginer de procéder d'une façon semblable : cette séquence devrait :

- assurer la présence de toutes les pages virtuelles intéressées dans des cadres (utilisation de AMENER, et blocage) ;
- construire un programme canal capable d'aller chercher des informations éparpillées dans les sous-pages et dans les cadres ;
- lancer l'entrée-sortie, en attendre la fin, et LACHER les pages.

Cette solution est techniquement possible; elle permet de travailler directement avec des buffers situés dans l'espace virtuel et de lancer les entrées-sorties directement sur celui-ci : la pagination amène les sous-pages virtuelles en mémoire, le canal les traite directement. Mais pendant ce traitement, il est interdit de déplacer les sous-pages ou les

pages : des mécanismes de blocages et d'attente doivent réalisér cela.

C'est la technique de virtualisation automatique des buffers (cf. [8], § 2.5.2) cf. figure 3.2.)

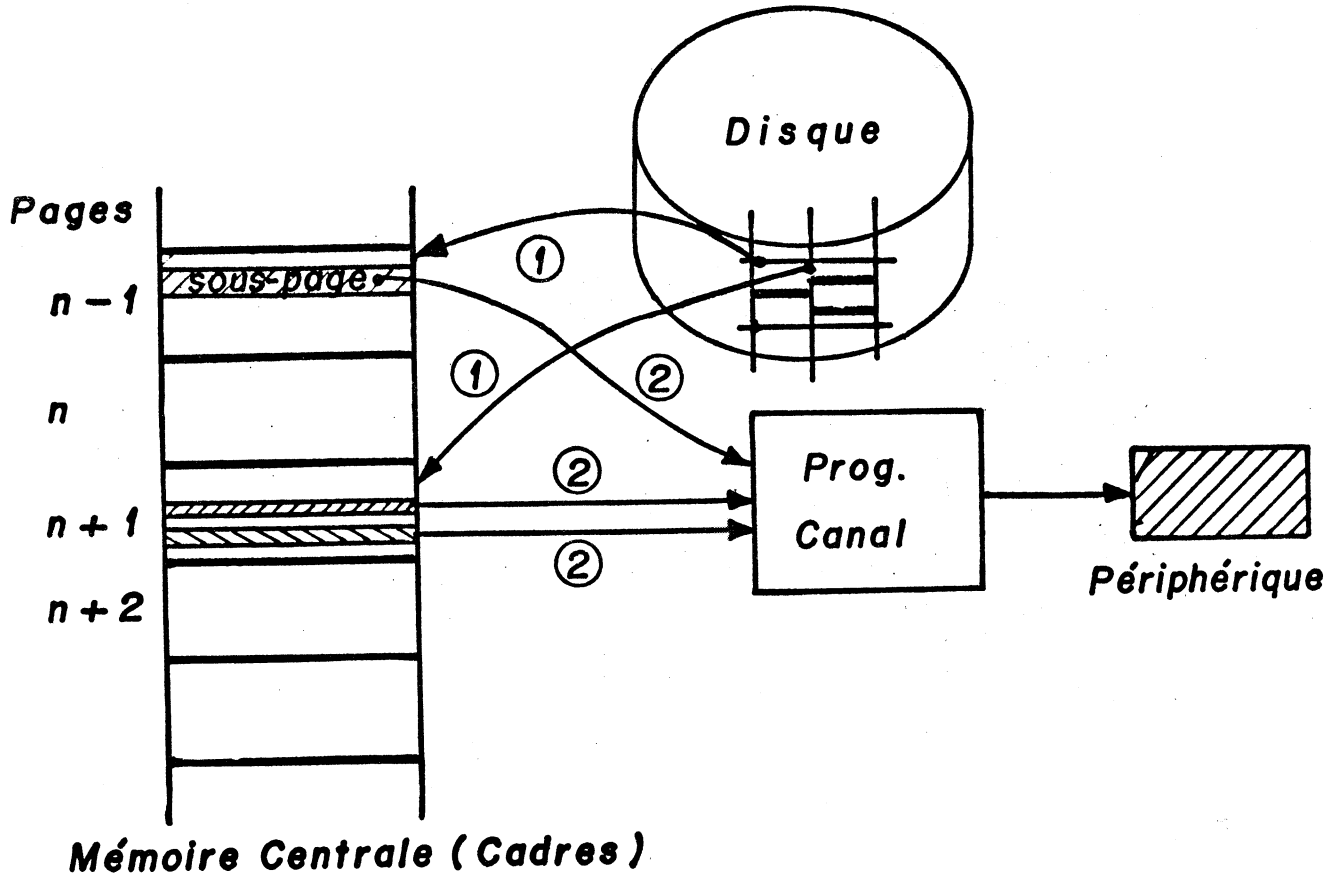


Fig. 3.2 - Virtualisation automatique :
① Travail de AMENER ② Travail du programme canal

L'autre solution consiste à utiliser des buffers en mémoire centrale, sur lesquels travaillent les canaux. Ainsi, une séquence qui réalise une entrée/sortie sur un buffer virtuel doit : (exemple d'une sortie)

- demander un buffer en mémoire centrale (primitive PRENDBUF qui donne n cadres consécutifs)
- Amener le buffer virtuel page par page dans des cadres et le recopier dans ce buffer (GLIRE)
- Construire le programme canal (simple)
- Lancer l'entrée/sortie, en attendre la fin, et rendre le buffer (primitive RENDBUF).

C'est la solution de la virtualisation programmée (cf. [8] , § 2.5.3 et figure 3.3). Elle utilise un programme canal simple, mais elle nécessite une recopie en mémoire centrale du buffer, et la réorganisation des pages dans les cadres lors de PRENDBUF. Elle est donc plus coûteuse, mais plus facile à réaliser, GLIRE et PRENDBUF existant déjà : nous l'avons adoptée dans un premier temps.

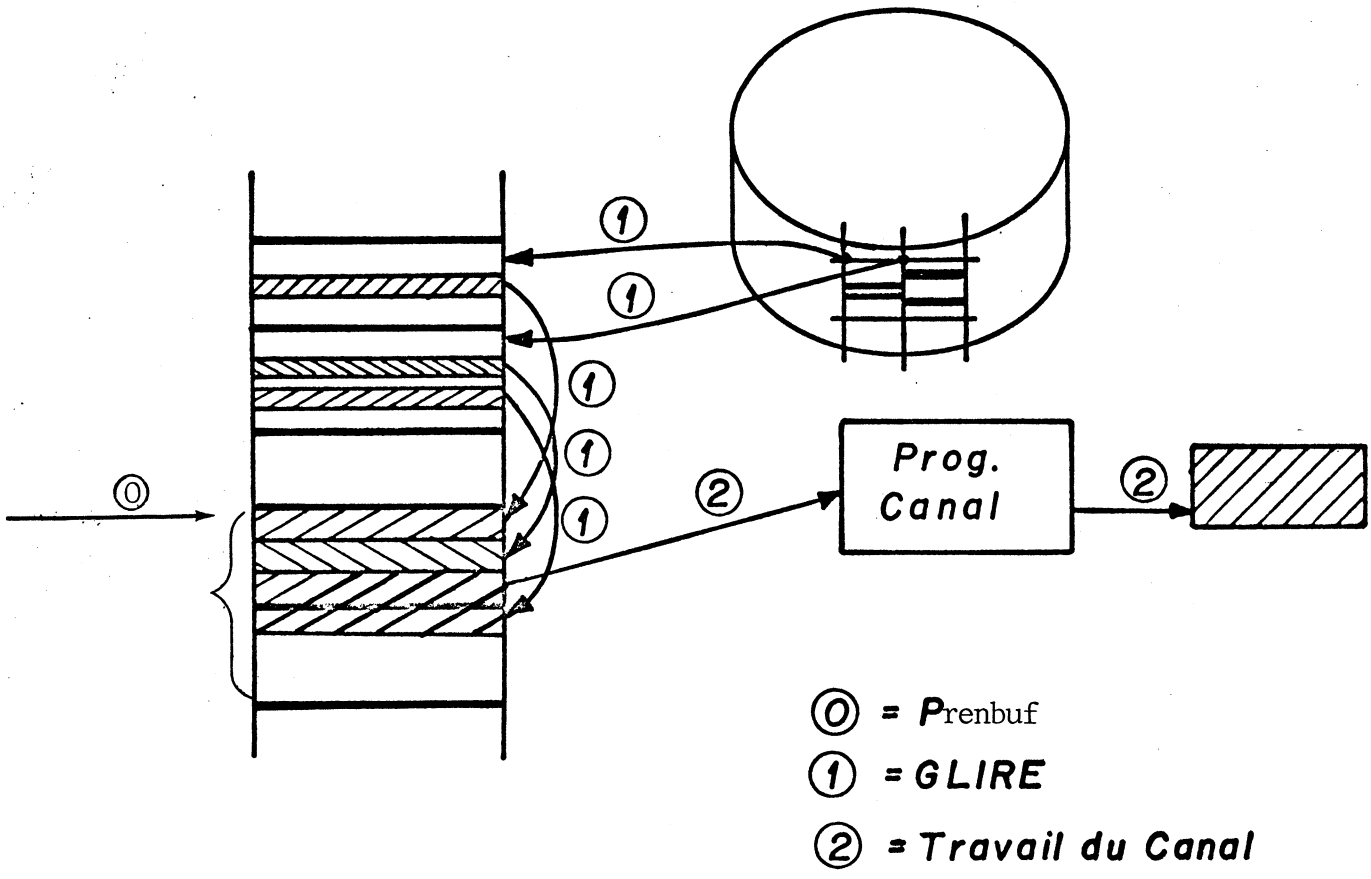


Fig : 3.3 Virtualisation programmée

Une fois ces séquences d'entrée/sortie écrites dans le langage de la machine de base, le programmeur utilise donc des périphériques qui travaillent directement dans l'espace virtuel, en particulier, quelqu'un peut accèder à une zone qu'un périphérique est en train de remplir ! (Le problème, qui se pose déjà en mémoire centrale, doit être résolu par ailleurs : buffers privés, protection par sémaphore, etc ...)

3.2.2. Attachements

L'utilisateur de SOCRATE ne sait pas avec quels périphériques il travaille; il ne connait que des"périphériques logiques" ou "standards" il a à sa disposition :

- un standard d'ordre,
- un standard d'entrée,
- un standard de sortie,
- des "périphériques banals" qu'il peut nommer
- etc ...

Quand un utilisateur conversationnel est initialisé, tous ses standards sont attachés à son terminal; pour un utilisateur "batch", entrée et ordre sont attachés au lecteur de cartes, sortie à l'imprimante.

L'opérateur et tout utilisateur conversationnel ont la possibilité de modifier dynamiquement ces attachements (touche "attention"); ainsi une sortie trop longue peut être transférée du terminal sur l'imprimante, par exemple.

Cela permet d'écrire des programmes indépendants des périphériques qu'ils utilisent; de plus, l'attachement à des textes SOCRATE permet de compiler des programmes préparés par avance à l'aide de l'éditeur, de ranger des sorties dans l'espace virtuel, etc ...

Pour cela, toute entrée-sortie repasse par un appel-superviseur qui consulte une table d'attachement de l'utilisateur, et appelle le processeur logique voulu pour exécuter la séquence associée. Cette table d'attachement fait partie de la tache, et définit l'environnement physique de celle-ci (comme la C-list [23]).

D'autre part, la BDB contient une table d'affectation de tous les périphériques aux taches; tout attachement active donc un programme de la BDB qui modifie cette table. Une gestion des ressources que sont ces périphériques peut alors être faite par le système. Certains sont alloués aux taches dès son démarrage (terminaux conversationnels, imprimante, ...) ; d'autres peuvent l'être dans le cours de la session, par intervention de l'opérateur (dérouleurs de bande); mais on peut toujours savoir qui utilise tel périphérique.

3.3. Problèmes de protection d'accès

Ce terme recouvre en fait trois problèmes distincts : d'une part les droits d'accès de l'utilisateur aux objets partageables, d'autre part les protections physiques qui garantissent que ces droits sont respectés; enfin les problèmes posés par des manipulations simultanées de ces objets.

3.3.1. Droits d'accès aux objets partageables

Dans la plupart des systèmes, l'objet partageable est le segment; plusieurs taches peuvent utiliser un segment, leurs "tables des segments" contiennent alors un pointeur sur la même table des pages. Il existe un grand nombre de segments dans l'espace permanent; pour qu'une tache puisse utiliser un segment donné il faut que cela ait été prévu à la génération du système ou de l'utilisateur (cf. [42]), ou bien qu'elle passe par une opération "rendre un segment connu à une tache " (cf. [12]). Cette opération consulte des dictionnaires et catalogues, vérifie les droits d'accès de cet utilisateur à ce segment, et indique, au niveau de la table des segments, le type d'accès autorisé (lecture seule, lecture et mise à jour, exécution, etc...)

Dans SOCRATE, chaque tache connait à priori un certain nombre d'espaces virtuels : espace fichier et espace programme de sa base, et de la BDB; espace système des zones de TRAV. Mais c'est le superviseur qui accède pour elle à ces espaces : un programme de requêtes qui s'exécute ne connait que les pages de sa mémoire virtuelle, auxquelles ont été couplées des pages programme ou fichier. Des vérifications sont faites par le superviseur au moment de ce couplage.

Les objets partageables ne sont pas les espaces virtuels, mais à un niveau plus fin les "entités" qui y sont contenues. Les droits d'accès à ces objets sont définis soit à la compilation, soit, s'il dépendent du contenu des données, sont vérifiés à l'exécution par des "actions spontanées" attachées structurellement aux objets eux-mêmes.

3.3.2. Protections physiques fournies par les systèmes de segmentation et pagination

3.3.2.1. Le type d'accès au segment est précisé (lecture et écriture, lecture seule, exécution, ...), et noté dans la table des segments. Il est vérifié à chaque accès [40].

Chez nous, il existe seulement deux types d'accès : lecture et mise à jour pour les TRAV et les fichiers (VOIES 1 et 3), exécution pour les programmes (VOIE 2); le superviseur amène toujours des programmes sur la VOIE 2, des TRAV ou des fichiers sur les autres.

3.3.2.2. Les accès en dehors du segment sont détectés comme des fautes, la longueur de chaque segment étant notée dans la table [40].

Chez nous, tout accès à un espace virtuel ne peut en dépasser la borne supérieure, celle-ci étant fixée par la limite d'adressage (32 bits). De plus, les accès aux objets logiques sont validés dans la mesure du possible à la compilation (cas des programmes, par exemple), ou à l'exécution par des routines mises en place à la compilation (du type "nombre maximum d'entités atteint"). Cela n'est possible que si les utilisateurs travaillent dans un langage de haut niveau; leur permettre un langage plus proche de la machine repose tous ces problèmes (cf § 3.3.4.).

3.3.2.3. Les accès en mémoire centrale en dehors du cadre sont impossibles : ils déclenchent une faute de page qui arrête la tache tant que la nouvelle page n'est pas là.

En l'absence de ce mécanisme, les compilateurs doivent s'assurer que les programmes n'adresseront jamais au delà de la limite de la page; si cela est nécessaire, ils génèrent des appels superviseurs qui demandent le chargement de la nouvelle page.

3.3.3. Partage des bases de données

Chaque base est accessible simultanément par plusieurs utilisateurs; il faut donc que des protections empèchent par exemple, que l'un supprime ou modifie ce que l'autre est en train de consulter. C'est là un problème qui se pose dès que plusieurs taches peuvent se partager l'accès à un fichier, et il existe deux types de solutions : la première consiste à associer des ressources aux objets du fichier, et à obliger les taches à demander ces ressources, donc éventuellement à se mettre en attente; la seconde, moins répandue, donne à chaque objet un "numéro de version"; chaque tache fait alors tout son travail en particulier, puis vérifie que personne n'a touché ce qu'elle a utilisé, et reporte toutes ses modifications sur la base, de façon ininterruptible; si le "numéro de version" a changé, elle recommence tout son travail. Cette solution a été proposée en [28]; elle peut être assez lourde, et ne s'applique pas facilement à des structures de fichiers où les objets à partager sont imbriqués les uns dans les autres.

Pour garantir un partage effectif des bases, le niveau de protection, l'objet sur lequel l'accès est contrôlé est la réalisation d'entité dans le cas d'une structure à entités imbriquées, un utilisateur travaillant avec une entité du niveau le plus bas permet aux autres de se servir de toutes les entités englobantes. Tous cela est présenté en détail en [8] chapitre 3.

Le mécanisme de base garantit seulement la cohérence des informations élémentaires contenues dans la base ("caractéristiques"), et le bon fonctionnement du système. Cela est obtenu en associant à chaque entité un processeur-logique "verrou", de DM = 1000, par exemple. Une lecture dans cette entité commencera par un P 1,verrou; une écriture ou une suppression, par un P 1000,verrou, réclamant tout le parallèlisme du verrou. Plusieurs lectures peuvent donc avoir lieu à la fois, mais l'écriture travaillera, toujours seule. Encore faut-il savoir associer dynamiquement un processeur à une entité. Tout cela a été vu en [8] § 3 et se trouve détaillé en annexe. Notons que le problème de l'exclusion lecture-écriture peut être réglé uniquement à l'aide de sémaphores classiques; c'est l'objet de [17].

Cependant, de telles protections apparaissent insuffisantes à l'utilisateur : la cohérence de deux informations n'est jamais garantie. Comme elle est coûteuse à obtenir, et pas toujours nécessaire, nous avons introduit dans le langage les requêtes BLOQUER ... LIBERER, qui assurent la cohérence de toutes les informations utilisées entre le BLOQUER et le LIBERER. Cela sera décrit dans une publication ultérieure ; le principe est de déterminer à la compilation toutes les entités qui seront ou pourront être accédées, et de les verrouiller toutes simultanément. Ainsi, on ne risque pas de blocage ("dead-lock") ; mais on effectue une "Parallel-P Operation" (cf. [27]), et le risque d'une attente illimitée subsiste.

3.3.4. Problèmes posés par des langages de bas niveau

Nous avons été amenés à implémenter de tels langages pour deux raisons : d'abord, pour pouvoir réécrire, et modifier dans le courant d'une session un certain nombre de composants du système (compilateurs, programmes de services, ...) ceux-ci doivent être optimisés au mieux, et pouvoir accéder les espaces virtuels directement par des adresses, et non au niveau des objets logiques. Ensuite, pour permettre à certains utilisateurs d'écrire de nouveaux programmes de services réellement optimisés, si le besoin s'en fait sentir. Ces programmes sont alors catalogués et utilisés comme s'ils avaient été écrits en langage SOCRATE.

Nous avons donc mis à la disposition de l'utilisateur un langage proche du langage d'assemblage. Cela soulève un grand nombre de problèmes, l'essentiel des protections du système étant réglées ou mises en places à la compilation.

Il faut d'abord éviter que le programme n'adresse des données ou des instructions en dehors des cadres de mémoire centrale qu'il utilise. Pour cela, une base implicite est fournie par l'assembleur et ré-alignée par le système en hautde chaque page de programme; des branchements en dehors de la page sont convertis en un appel superviseur qui sait restaurer le code condition. Toutes les données utiliséespar le programme sont définies symboliquement par le programmeur (DSECT), mais sont allouées par le système

dans la page de TRAV; elles sont donc limitées actuellement à 1 K (au-delà, elles sont à ranger dans l'espace virtuel); un mécanisme de pile est fourni par le système, permettant d'allouer ou de désallouer un morceau de la page TRAV, qui sera adressé symboliquement (Cela est possible par l'utilisation du registre d'index, ce qui limite les instructions autorisées à travailler sur une telle zone; de plus, un registre supplémentaire est interdit au programmeur). Et les accès-fichier ne sont faits que par le superviseur (LIRE, ECRIRE, ...).

Ensuite, ces programmes peuvent écrire n'importe où dans les espaces virtuels de l'utilisateur, et peuvent donc les détruire ; ce risque est accepté, de tels programmes devant rester l'oeuvre du responsable de la base de données ("Data Manager"), qui doit s'assurer de leur fiabilité avant de les livrer à ses utilisateurs.

Enfin, lorsque ces programmes veulent accèder à des objets partageables contenus dans une base, ils doivent le faire par des appels à des programmes écrits en SOCRATE, lesquels gèrent ces partages : le mécanisme d'accès aux objets logiques doit être le même pour tous les utilisateurs, c'est-à-dire celui de SOCRATE.

Cependant, assurer une sécurité totale conduit à n'autoriser qu'un sous-ensemble trop restreint du langage de la machine, ou à imposer des restrictions trop sévères. Nous avons préféré conserver une certaine puissance, quitte à limiter l'usage d'un tel langage à un petit nombre d'utilisateurs expérimentés, dans le cadre de sessions spéciales.

3.4. Problèmes de blocage ("dead-locks")

La revue des problèmes de blocage provoqués par le système que nous venons de présenter a été faite en 8 , § 3.7 ; depuis, une modification est intervenue : la primitive BLOQUER ne risque plus de provoquer de "dead-lock", et l'algorithme de détection de ce cas n'a plus de raison d'être. Nous rappellerons brièvement ici les idées qui se dégageaient de ce paragraphe.

Il existe deux catégories de méthodes pour régler ces problèmes éviter les blocages, ou bien les détecter et les traiter.

Détecter de telles situations n'est pas facile. Certains systèmes ontune tache spécialisée ("watchdog") qui consulte l'état d'avancement de toutes les autres (horloge interne), et peut s'apercevoir que certaines sont arrêtées depuis trop longtemps sur des demandes de ressources, ou des attentes d'interruption. Cette méthode est souvent approximative, mais permet de régler certains problèmes d' "attente indéfinie" ! D'autres, comme MURPHY [43] , proposent un algorithme pour détecter le dead-lock au moment où une nouvelle demande de ressource le provoque. En [8] , § 3.7., nous avons décrit un tel procédé pour les dead-locks introduits par la primitive BLOQUER(A) et les verrouillages sur les entités. Le problème est moins de détecter le dead-lock que de le résoudre : dans un système comme le nôtre, les tâches travaillent sur des bases de données communes, et il est déconseillé de stopper quelqu'un arbitrairement : il peut avoir déjà fait certaines modifications, qui rendent la base incohérente logiquement (la cohérence élémentaire étant toujours préservée).

On peut éviter un dead-lock de plusieurs façons; la plus élaborée est l'algorithme de HABERMAN [31] : il suppose que l'on connaît a priori toutes les ressources que vont demander chaque tache, et ne les alloue que si cela ne risque pas d'engager un dead-lock, compte-tenu des ressources que chacun va demander par la suite. Outre le coût d'une telle solution pour

un système comprenant beaucoup de ressources, l'hypothèse de la connaissance a priori des demandes ne nous permet pas de l'envisager. En effet, on ne peut pas savoir a priori quelles entités un programme va devoir accèder (par exemple).

Les méthodes les plus réalistes sont les moins ambitieuses ; leur idée est de construire un système libre de dead-lock par allocation des ressources quantifiées de façon globale (tout ce qui sera nécessaire est réservé au début), et construction d'algorithmes réclamant les ressources toujours dans le même ordre (HAVENDER [37]). Ces règles paraissent très contraignantes; en fait, nous sommes arrivés à construire pour SOCRATE un système qui les respecte toujours, et qui ne sous-emploie pas trop les ressources. Prenons comme exemple le cas de la mémoire centrale : pour exécuter un programme, une tache devra avoir réservé disons 5 cadres; mais elle n'en utilise que 1,5 en moyenne. Du fait de la pagination, les cadres sont considérés de façon globale, et la conséquence de ce sous-emploi est que les pages restent plus longtemps en mémoire - ce qui présente des avantages !

Cette première règle d'allocation en une seule fois des ressources quantifiées ne nous a donc jamais paru trop restrictive. Ainsi, une tache n'est lancée que une fois réservés tous les cadres nécessaires à son exécution, y compris les buffers ; dans le cas d'un appel de sous-programme, la réservation est lachée, puis refaite pour le sous-programme (car pour faire ce V(n,places-de-cadres) et P(n',places-de-cadres), aucun cadre n'est effectivement nécessaire !).

La seconde règle concerne la demande des ressources toujours dans le même ordre; elle est simplifiée par une architecture "en niveau", certains types d'allocation ayant toujours lieu avant d'autres. Le classement complet a été fait en [8], § 3.7.2.; il est facile à vérifier, du fait que toutes les taches utilisateurs fonctionnent de la même façon, et que les autres taches sont très peu nombreuses. Pour ce qui est des ressources non connues à priori, comme les entités à verrouiller, les programmes d'accès à ces objets logiques travaillent tous de la même façon, en verrouillant du plus externe vers le plus interne structurellement.

Malheureusement, même si ces deux règles sont strictement respectées, cela ne garantit pas l'absence de dead-lock. Un très bon contre-exemple est le Problème Des 3 Fumeurs (décrit en [46]). Ce n'est vrai que si les ressources, libres a priori, sont demandées puis lachées par une même tache; ça ne l'est plus forcément si l'on introduit des ressources de synchronisation (sémaphores).

Il devient alors très difficile de se convaincre du bon fonctionnement d'un système construit sous forme de taches indépendantes synchronisées de cette façon (c'est une des raisons pour lesquelles une organisation verticale nous a paru préférable : la plupart des objets de synchronisation sont des ressources à prendre et à libérer). Il existe pour cela des outils de représentation des systèmes : graphes de PETRI ([46], [44]), GERT présenté en [44]. Ils permettent une meilleure visualisation et compréhension d'architectures relativement simples; mais ils sont bien moins utiles si les algorithmes demandent des ressources non connues à priori. D'autres formalismes existent ([30], [32]), qui permettent de démontrer le bon fonctionnement de certains systèmes mais très simple. Dans notre cas, ces méthodes ne sont pas raisonnablement applicables, le nombre des ressources étant très grand (une par entité).

CHAPITRE 4

4. DESCRIPTION INTERNE DE LA MACHINE DE BASE

4.1. Présentation

4.1.1. Rappels et terminologie

Notre machine doit assurer le fonctionnement d'un certain nombre de taches (ou processus); celles-ci sont en nombre réduit : une tache par utilisateur et quelques taches systèmes. Pour fonctionner avec les ressources logiques du système, elles font des appels aux processeurs-logiques que l'on associe aux ressources. Le multiplexage des unités centrales et canaux pour "supporter" ces processeurs logiques est le travail du Régulateur de processeur. De la même façon, elles demandent la présence en mémoire de certaines pages de l'espace-disque ou de certains buffers : on appellera cadres-logiques ces demandes concernant la mémoire, et régulateur de mémoire le mécanisme qui alloue les cadres de mémoire centrale (ressource physique) à ces demandes. Une tache peut demander jusqu'à 3 pages-disques et 2 buffers : 5 voies sont prévues à cet effet, par tache. Le passage des demandes en termes d'espace virtuel aux demandes en termes de pages-disques est réalisé par la primitive Amener de la pagination (ce terme de "pagination" désigne toujours, chez nous, l'ensemble des mécanismes réalisant le passage de l'espace virtuel à la mémoire centrale).

Après description de ces deux régulations, nous verrons comment elles sont couplées, c'est-à-dire comment l'allocation d'un type de ressource tient compte des allocations déjà faites concernant l'autre type. Puis nous comparerons les stratégies d'allocation utilisées à d'autres déjà connues.

4.1.2. Régulateurs

Un régulateur est un mécanisme chargé d'allouer et de préparer les ressources physiques de la configuration à des représentations "logiques" de ces ressources, qui sont demandeuses. Cette allocation sera généralement faite sur la base des priorités des demandeurs.

Pour réaliser ces régulateurs, nous supposerons l'existence des instructions SOLO et TUTTI de [49], permettant à une unité centrale de s'isoler dans une section critique, sans risque d'interférence avec une autre, ni d'interruption. Tous les algorithmes de régulation comprenant la consultation des listes de demandeurs, l'élection de l'un d'eux, la désactivation d'un autre, etc... sont protégés par ces instructions. De même les manipulations des files d'attente des processeurs par Appel et Sortie, des files des cadres logiques par Demander et Lacher sont à protéger.

Dans un système monoprocesseur, ces primitives sont réalisées très simplement en se masquant à toutes les interruptions. Dans le cas d'un multiprocesseur, on peut les construire à l'aide d'une instruction du type Test and Set des 360 : cela a été décrit en [49] :

```
SOLO :  L : TS(flag) ; si ≠ alors allera L
TUTTI : MVI(0,flag)
```

D'autres solutions que Test and Set sont imaginables; LAMPSON [41] propose une instruction PRO qui assure l'exclusivité des accès, pendant quelques instants, à l'unité qui l'a émis : sous cette protection on peut alors réaliser le Test et le Set, ou bien effectuer un petit travail.

En l'absence de tout dispositif hardware spécial, il existe un algorithme, proposé par DIJKSTRA [24], qui assure cette exclusion élémentaire dans une section critique; malheureusement son coût est élevé.

Quelque soit la technique employée, nous supposerons que les régulateurs travaillent toujours sous cette protection; cela n'est pas restrictif car ils sont simples, et représentent assez peu d'instructions (il serait même souhaitable de les microprogrammer ou de les cabler).

4.2. Régulation des processeurs

Le rôle du régulateur de processeur est donc d'allouer la ou les Unités Centrales aux processeurs logiques qui en font la demande.

Ceux-ci sont rangés dans une "file d'attente du hardware", organisée par priorités : à chaque processeur logique est associé un niveau de priorité, et cette "file" se compose en fait d'une file par niveau. Sont présents dans celle-ci tous les processeurs qui sont éveillés, c'est-à-dire demandeurs d'unité centrale; les autres sont dits dormants. Chaque processeur dispose de deux files de taches : la file des taches réellement actives sur ce processeur, et la file des taches qui l'attendent. On a vu que, à la suite d'un appel, une tache pouvait rester logiquement active sur un processeur, mais se trouver en fait dans la file d'un autre; la primitive Sortie la ramènera dans la file des taches actives sur le premier.

Un processeur peut être dans l'état "dormant" pour plusieurs raisons :

- il peut être actif sur un canal ou sous-canal; l'interruption de fin d'entrée/sortie le réveillera ;
- il peut n'avoir rien à faire (ses deux files sont vides);
- il peut être "saturé" : des taches attendent dans sa seconde file, mais il ne peut les satisfaire, et les taches actives sur lui sont parties sur d'autres processeurs.

Dans ces trois cas, il ne réclame pas l'unité centrale, et est donc absent de la "file d'attente du hardware". Par contre, s'il y est présent, il a toujours quelque chose à faire : ses files ne sont pas vides.

Chaque fois qu'un processeur passe de l'état dormant à l'état éveillé, ou inversement, il y a modification de la demande générale de la ressource Unité Centrale, et le régulateur est activé : il remet en cause l'allocation de cette ressource aux demandeurs, sur la base des priorités. C'est d'ailleurs ce que réalise un mécanisme d'interruption classique.
Cf. fig. p. 78.

Ce passage d'un état dans l'autre peut avoir diverses causes :

- à la suite des instructions Appel et Sortie (et P et V), les files des taches attendant un processeur ont été modifiées, et certains peuvent être réveillés ou s'endormir ;
- une interruption externe, venue d'un canal ou sous-canal, peut signifier une fin d'entrée-sortie; le processeur qui travaillait sur ce canal abandonne le hardware-canal et redevient candidat à l'Unité Centrale : il est donc réveillé.
- un lancement d'entrée/sortie sur un canal (SIO-HASLP) représente le "changement de hardware" inverse : le processeur qui travaillait sur l'Unité Centrale devient actif sur le canal et abandonne la première : il s'endort.

Dans tous ces cas le régulateur est activé, et peut éventuellement arrêter un processeur pour en démarrer un autre (préemption). Ce dernier va alors pouvoir activer une tache qui se trouve dans une de ses files, en commençant par la première. Pour cela, il déroule l'algorithme "Scheduler" qui choisit la tache à activer; on a implémenté le même Scheduler pour tous les processeurs, mais il pourra être intéressant d'adapter cet algorithme de choix de la tache à la sémantique du processeur logique.

4.3. Cadres logiques et régulation-mémoire

Nous présentons ici rapidement ce qui a été complètement décrit en [8] , § 2.3.

4.3.1. Définitions

On appelle cadre ou cadre physique l'unité de mémoire centrale capable de recevoir des pages; ces pages sont des tranches de l'espace-disque (1 K-octet), ou plutôt l'information qu'elles contiennent. Dans une première présentation on négligera la projection :

Espace virtuel → Espace disque

qui sera reprise au § 4.4., pour s'intéresser uniquement au problème d'amener et de partager ces pages-disques en mémoire centrale.

Les cadres-logiques (C.L.) sont des utilisations de mémoire satisfaites sur la base des cadres physiques, mais pourvues d'un caractère logique. Ainsi, une page-disque chargée dans un cadre est un C.L., un buffer constitué de n cadres consécutifs en est un autre.

Les taches peuvent demander un cadre-logique par voie ; elles ne seront activées que lorsque ces demandes auront été satisfaites. Si plusieurs taches demandent un cadre-logique associé à une page-disque, celui-ci doit évidemment être partagé, être accessible à toutes ces taches : il n'est pas question d'avoir en mémoire deux copies d'une page de la base. Si un cadre logique vient à disparaître, plusieurs taches éventuellement devront être arrêtées. Par contre, dans notre implémentation, les buffers en mémoire centrale ne sont pas partagés, un tel cadre logique n'est utilisé que par une tâche; en effet, nous utilisons de tels buffers pour des entrées/sorties uniquement. Cependant, si l'on savait nommer de façon unique ces objets, on pourrait permettre aux taches de les partager : un cadre-logique est conçu de façon à être utilisé par plusieurs taches. (Cela serait utile, par exemple si l'on voulait construire une tache "bande", qui travaille en se synchronisant avec une tache utilisateur, en remplissant pour elle un buffer, ou plusieurs : méthodes de Flip-Flop).

4.3.2. Utilisation des cadres-logiques

Les cadres logiques sont soit demandeurs d'un cadre physique, soit nantis; dans ce dernier cas ils peuvent être soit prêts, c'est-à-dire que la page-disque y est chargée, soit en attente de transfert, que ce transfert consiste à renvoyer une ancienne page sur disque, ou à en amener la nouvelle.

Suivant l'utilisation qui est est faite, les C.L. ont une priorité différente : pour schématiser, on décrira 3 priorités :

1 : buffers utilisés
2 : pages-disques utilisées (une tache a fait DEMANDER)
3 : buffers ou pages-disques non utilisés.

Les taches font DEMANDER pour réclamer un cadre-logique; elles ne sont alors élues que si ce C.L. est prêt. Elles font LACHER quand elles n'en ont plus besoin. Une tache ne peut demander qu'un C.L. par voie. L'effet de Demander (page disque, n° de voie) est :

- consulter tous les C.L. "page-disque"
- si le C.L. cherché n'existe pas, le créer "demandeur" en priorité 2, et chaîner la voie dessus; puis lancer le régulateur de mémoire.
- si il existe, chaîner la voie dessus, et le mettre en priorité 2.

celui de Lacher (n° de voie) est :

- dechaîner la voie du C.L. de la voie,
- mettre ce C.L. en priorité 3 si aucune autre tache ne l'utilise; et lancer alors le régulateur de mémoire.

4.3.3. Régulateur de mémoire

Son rôle est d'affecter les cadres physiques aux C.L. demandeurs, suivant les priorités, et, au besoin, en privant un ou plusieurs C.L. "nantis"de leurs cadres. C'est donc un mécanisme tout à fait analogue au régulateur de processeur. (Cf. fig. p. 78).

Il opère en comparant la priorité du demandeur le plus prioritaire à celle du nanti le moins prioritaire; les demandeurs et les nantis étant organisés, dans chaque priorité, sous forme de liste doublement chaînée, le premier nanti de la file 3 est aussi le plus anciennement utilisé : algorithme de remplacement est donc, dans ce cas, L.R.U.

Si un "nanti" de priorité 3, c'est-à-dire non utilisé, est ainsi privé de son cadre, il est tué : son cadre logique disparaît, et si une tache le demande par la suite, il faudra le créer de nouveau; s'il était de priorité 2, il repasse "demandeur" dans la même priorité, et toutes les taches qui l'utilisent sont suspendues tant qu'il n'est pas de nouveau prêt. (On voit la nécessité du chaînage des voies sur le C.L. qu'elles utilisent).

4.3.4. Suite du travail de préparation d'un C.L.

Une fois "nanti", le C.L. n'est pas encore prêt : si il concerne une page-disque, il faut encore lancer la sortie éventuelle de l'ancienne page, et l'entrée de la nouvelle sur le canal disque, puis la validation des taches qui l'attendait; s'il concerne un buffer, il faut peut-être attendre la sortie de toutes les pages privées de leur cadre, et valider la tache.

Tout ce travail de "page control" est réalisé, dans notre implémentation, par l'unité centrale, au gré des interruptions venues du canal-disque; on verra comment au § 4.5.2.

4.4. Travail sur l'espace virtuel

Le système fournit une primitive AMENER (adresse virtuelle, n° de voie) dont le but est d'amener en mémoire la page-disque contenant la sous-page ayant cette adresse virtuelle, et de ré-ajuster le registre de base de la voie et la variable depla (Adresse dans la page du mot ayant cette adresse virtuelle). Cette primitive utilise DEMANDER (page réelle), consulte le mot d'en-tête, fait éventuellement LACHER puis DEMANDER d'une autre page, pour suivre une chaîne de sous-pages.

Pour passer de l'adresse virtuelle à l'adresse disque, il faut connaître la taille de sous-page, et la taille de l'espace-disque, puis les adresses des cylindres et pages de la Base. Chez nous, ceux-ci sont forcément consécutifs, et il suffit de connaître l'adresse de début de l'espace-disque. La "table des pages réelles" se réduit donc à cette adresse, la taille de l'espace réel, la taille de sous-page. Ces tables sont donc petites; de plus, le nombre de bases montées à la fois est limité. On a donc rendu ces tables résidentes.

Si l'on voulait travailler avec un très grand nombre de bases, il faudrait que AMENER commence par aller lire, dans la BDB, cette table, et en range les informations dans la voie; la suite du travail reste inchangée. On voit que cela est possible à condition que la table de l'espace virtuel BDB soit, elle, résidente. Cet accès à la "table des pages" n'est pas sans rappeler la "Faute de Segment" de MULTICS [12], que l'on a évoquée au § 2.1.1.

D'autre part, la correspondance **sous-page virtuelle → sous-page réelle** n'est pas permanente : une sous-page virtuelle "squatter" peut être déplacée à tout moment, c'est-à-dire changer de sous-page réelle (sur disque); elle peut aussi être recopiée à la place de la tête de chaîne, ou même être remise à zéro : la sous-page réelle est libérée.

Dans tous ces cas, il importe de signaler aux taches qui éventuellement utilisent déjà une telle sous-page que celle-ci change de place. Pour cela, on devrait associer à chaque sous-page présente en mémoire centrale une file des voies qui l'utilisent. Le nombre des sous-pages par page étant variable, cela serait coûteux en place, et en temps. Comme une telle situation se produit rarement, on se contente de consulter toutes les voies qui utilisent la page contenant cette sous-page, et on regarde si elles travaillent effectivement avec elle. Si oui, on force la tache à faire LACHER sur cette voie, puis à redemander AMENER (adresse virtuelle).

Un autre problème posé par l'utilisation des espaces virtuels est celui du cadre de manoeuvre : il arrive qu'une voie ait besoin de 2 pages disques en mémoire, pour recopier une sous-page dans une autre, ou pour mettre à jour un chaînage. Cela arrive rarement, et on le fait en amenant, sans perdre le contrôle, la seconde page dans un cadre de manoeuvre qui sera libéré immédiatement après. Cependant, une page ne doit jamais exister en deux exemplaires en mémoire centrale ; on commence donc par chercher la seconde page parmi les C.L. existants; suivant les cas, il faudra donc prêter puis reprendre leur cadre physique de manoeuvre, ou bien un cadre logique et un cadre physique, ou bien aucun des deux. Le problème se complique lorsque la sous-page qui intéresse la tache se trouve en définitive dans le cadre de manoeuvre : il faut alors échanger le C.L. "demandé" par la voie, et restituer celui qu'on utilisait précédemment; mais si d'autres l'utilisaient aussi, il faut leur faire refaire DEMANDER (C.L. de cette voie).

Le partage de ces espaces virtuels pose donc quelques problèmes; ils ont toutefois pu être résolus, au prix d'une certaine complexité parfois; mais de tels cas ne se produisent que très rarement, et cette complexité ne nuit pas aux performances générales du système. De plus, ces problèmes ne sont pas à négliger si l'on veut introduire des mécanismes de mémoires associatives (adresse virtuelle → adresse mémoire centrale).

4.5. Couplage des deux régulations

4.5.1. Schéma général

Les taches du système peuvent donc demander des processeurs-logiques ou des cadres logiques, et travailler avec; ces entités logiques sont réalisées physiquement grâce aux ressources physiques de la machine, qui leurs sont allouées par les régulateurs (cf. figure 4.1. page 78).

REMARQUES :

1 - *Au-dessus du niveau 0 de la machine de base, les taches ne connaissent que les processeurs-logiques et les cadres-logiques.*

2 - *Le schéma n'est pas vraiment symétrique, la commutation d'une unité centrale étant quasi instantanée alors que une page doit être chargée depuis le disque. Il le devient dans un contexte de microprogrammation dynamique : un processeur logique devient "nanti" quand une Unité Centrale lui est attribuée; il ne devient "prêt" que quand celle-ci a été chargée.*

Il y a une autre différence : un processeur logique actif sur une unité centrale ne peut servir qu'une tache à la fois, tandis qu'un cadre logique peut en servir plusieurs.

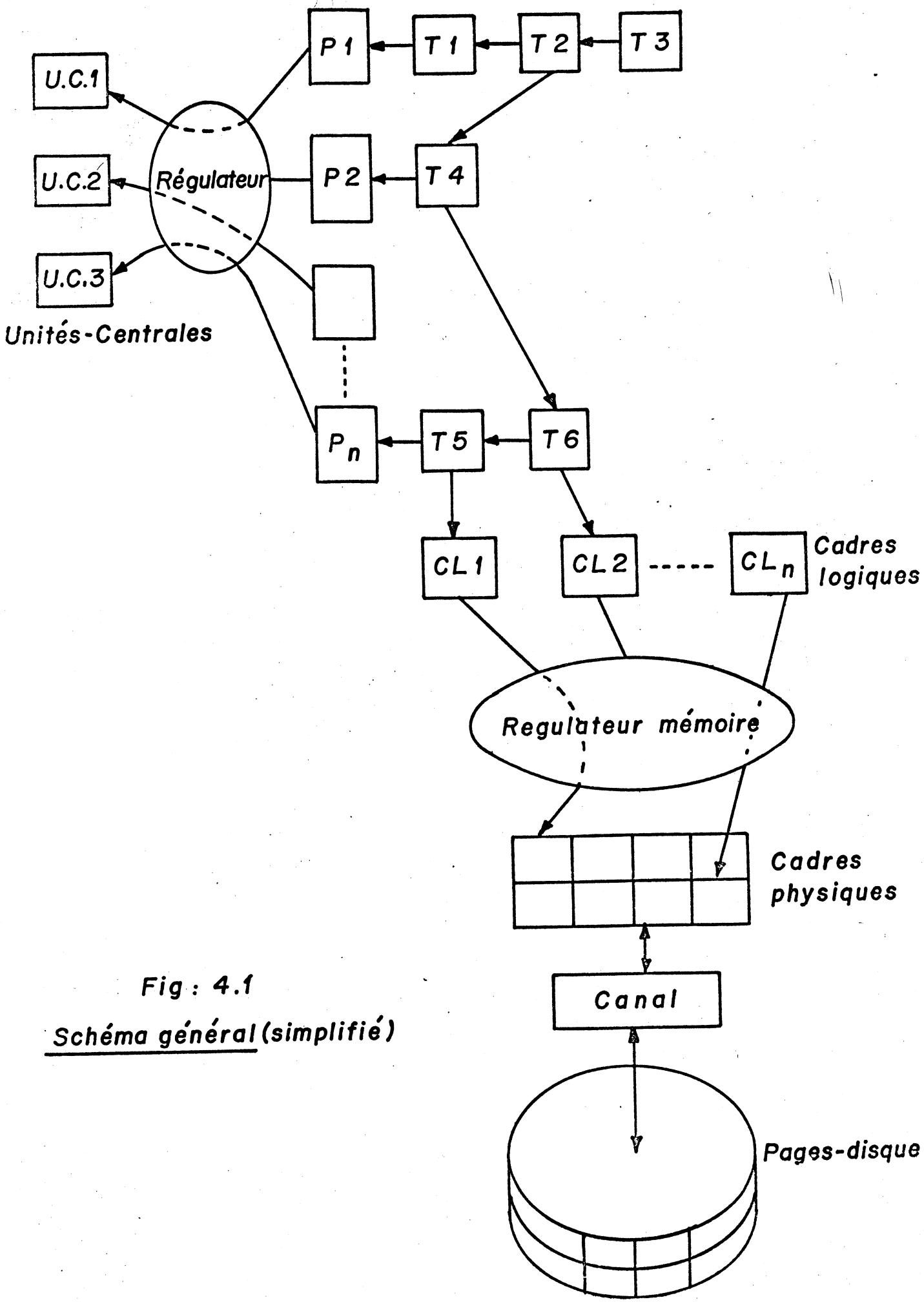
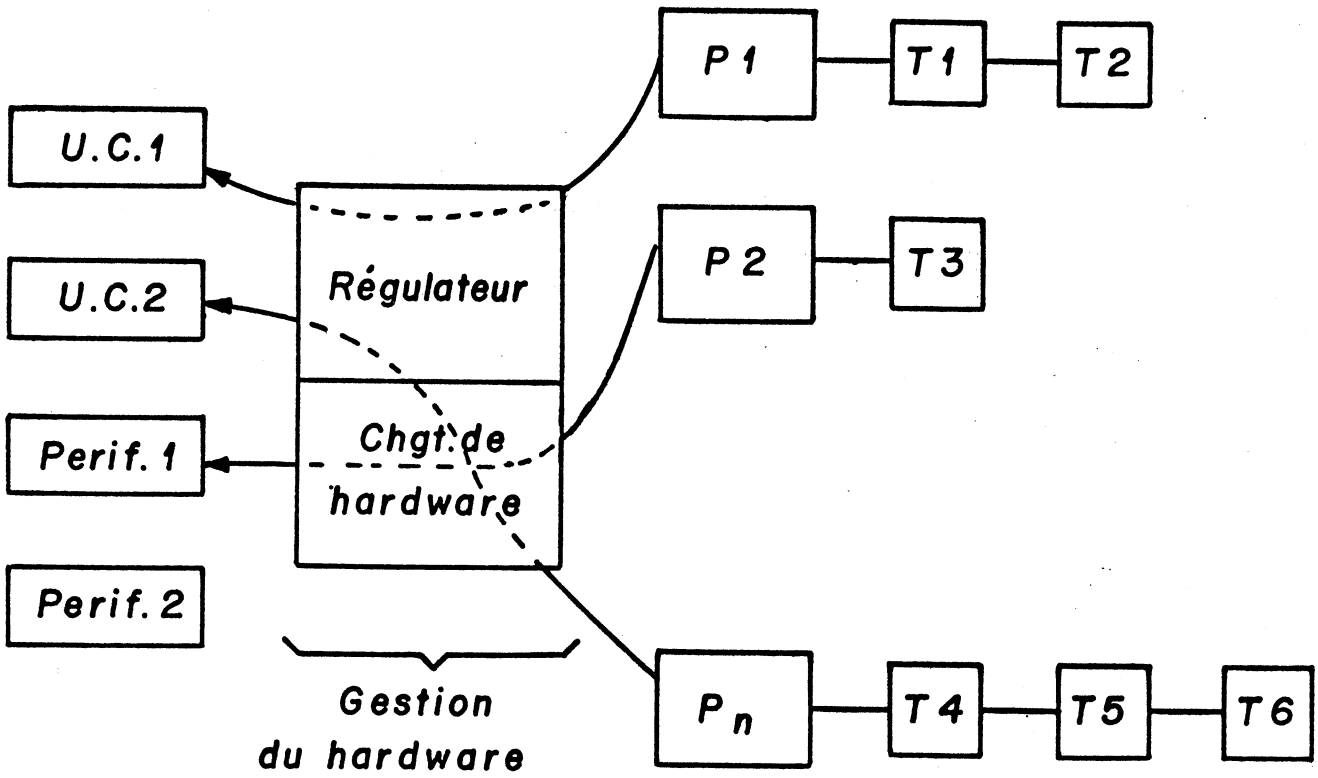


Fig : 4.1

Schéma général (simplifié)

3 - *On n'a pas porté sur le schéma le mécanisme du changement de hardware, et les canaux autres que le canal-disque.*

En fait le schéma serait :



4 - *De la même façon, toute la réalisation des cadres logiques sur les cadres physiques représente tout un travail de "page control" qui n'apparaît pas dans la figure 4.1. Cette fonction devra être remplie par une unité centrale, comme nous allons le décrire.*

4.5.2. Mécanisme du couplage des deux régulations

Ces deux mécanismes d'allocations ne travaillent évidemment pas de façon indépendante. La solution choisie consiste à n'allouer une unité centrale qu'à un processeur qui a des taches capables de travailler, c'est-à-dire dont les demandes de cadre logique sont satisfaites. Le temps de commutation d'une unité centrale étant très faible devant le temps de chargement d'une page, cela doit assurer la meilleure utilisation des ressources.

Pour cela, chaque tache dispose d'un bit d'éligibilité, et de un bit par voie signifiant "en attente de cadre logique". Quand un cadre logique devient prêt, un de ces derniers bit est basculé OFF; et s'ils sont tous OFF, la tache devient éligible. Quand un cadre logique est désactivé (privé de son cadre physique), la tache devient "non éligible" et "en attente de C.L.".

Le régulateur de processeur se trouve donc inclus dans un "super régulateur" capable de déterminer quel processeur a des taches éligibles, puis de lui allouer une unité centrale.

D'autre part, une bonne partie du travail de "page control" ne peut être faite directement au gré des interruptions : ainsi, si une sous-page disparaît, plusieurs taches peuvent avoir à faire AMENER; cela est dévolu au "super-régulateur" : chaque tache contient un bit "intervention requise", et un "prochaine adresse à exécuter", par voie. Dans le cas cité, "intervention requise" serait basculé ON, et la "prochaine adresse" serait AMENER. Le super-régulateur teste ces bits, et initialise le travail, jusqu'à ce qu'il soit arrêté par une demande de cadre physique, ou d'entrée/sortie disque. Il recommence alors pour d'autres taches; mais dès qu'il en trouve une éligible , il lui affecte une unité centrale.

Quand une unité centrale est affectée à un processeur, "Scheduler" va chercher la première tache éligible; celle-ci n'est pas forcément la première de la liste, et donc la politique d'allocation des ressources aux demandes n'est pas FIFO.

Cela repose le problème des attentes illimitées, en particulier dans le cas des P(1000, verrou) pour l'accès en écriture à une entité : une tache dont les pages sont absentes peut se faire doubler indéfiniment. Pour remédier à cela, la politique du Scheduler est de ne pas continuer à explorer la file d'attente logique quand il a rencontré une demande qui va provoquer une attente (telle que $Nb_{demandé} + D > DM$*).*

4.5.3. "Interruptions"

On peut imaginer que les fonctions du régulateur de mémoire et du régulateur de processeur soient réalisés par des dispositifs indépendants de l'unité centrale; de tels mécanismes doivent pouvoir être réalisés de façon cablée : ils ne seraient pas très différents des systèmes d'interruption actuels.

On peut aussi imaginer que tout le travail de "page control" soit fait de façon indépendante, par une unité associée au controleur de disque, et qui serait capable de travailler sur des files en mémoire centrale.

Une telle unité, après avoir basculé le bit d'éligibilité d'une tache, devrait être capable de signaler au régulateur de processeur que l'état d'une tache a changé, et donc qu'il faut remettre en cause l'allocation des Unités Centrales aux processeurs ; de même si une tache devient non éligible. Une communication directe entre les deux mécanismes reste donc nécessaire : si l'on considère tous les périphériques comme des unités capables de coopérer directement, le concept d'interruption disparaît. Il reste que le régulateur de processeur peut être activé sur une action d'une telle unité (Appel, Sortie), mais aussi sur un signal émanant du mécanisme de "page control".

4.6. Politiques d'allocation

Tous les systèmes actuels accordent une grande attention aux politiques d'allocation de mémoire et d'unité centrale ; il apparaît que celles-ci ont une très grosse importance pour éviter une dégradation rapide du fonctionnement particulièrement dans les systèmes utilisant un mécanisme de pagination. Dans ce cas, elles peuvent aussi être jumelées avec l'algorithme de choix de la page à vider.

En effet, si l'on alloue l'unité centrale à des taches qui n'ont pas leurs pages en mémoire, les défauts de pages seront fréquents ; la situation peut devenir catastrophique si le nombre de taches augmente, aucune d'elle ne parvenant à avoir en mémoire les pages qu'il lui faut, c'est le phénomène de "thrashing" : à partir d'un certain nombre de taches, on note une chute de performance très brutale de tout le système. Cependant, des politiques d'allocations évoluées peuvent éviter ce phénomène; et donc permettent au système de supporter des "pics" de charge sans cette chute presque totale des performances.

Ces politiques sont basées sur les concepts introduits par le modèle du **"working set"** de DENNING [19][21]: celui-ci suppose que les références que fait un programme à ses pages à un instant donné ne sont pas uniformément réparties sur l'ensemble des pages, mais obéissant à un "principe de localité" : pendant un certain temps, un programme référence plus particulièrement certaines de ses pages. On définit donc le working-set d'un programme à l'instant t

$$W(t,T)$$

comme l'ensemble des pages référencées entre t-T et T. Suivant le principe de localité, les éléments de ce working-set ont de grandes chances d'être prochainement référencés : le W.S. est alors une estimation des pages dont le programme va avoir besoin. (Soulignons aussi l'importance du paramètre T, taille de la fenêtre; Denning a donné des éléments permettant de choisir une valeur de T qui donne des résultats satisfaisants).

La connaissance du W(t,T) d'un programme nécessite un travail considérable, et bien souvent on se contentera, à un niveau moins fin, de la taille de ce working-set :

w(t,T).

Une première mesure à prendre pour éviter le phénomène de thrashing peut être de limiter le nombre de taches qui rentrent dans le jeu de la pagination (cette idée, très répandue, se trouve déjà chez SALTZER [48]; pour cela, on distingue réservation et allocation des cadres : les taches doivent d'abord réserver un certain nombre de cadres; puis l'allocation de ceux-ci sera faite au moment des défauts de page. Ainsi toute tache capable de travailler aura à sa disposition un nombre de cadres suffisant.

Encore faut-il que les taches qui peuvent ainsi travailler ne monopolisent pas la mémoire, et que cette réservation soit limitée dans le temps. Si l'on peut être sûr que les taches libèreront d'elles-mêmes leurs réservations assez tôt, le problème est réglé (cas des systèmes activant uniquement des programmes issus d'un compilateur). Sinon, il faut que le superviseur annulle périodiquement cette réservation, et fasse repasser les tâches dans un état "en demande de réservation" (c'est le cas dans ESOPE [13]).

On distingue alors les taches "prêtes" et les taches "actives en mémoire"; seules ces dernières occupent des cadres, et peuvent être élues sur l'unité-centrale. Elles restent dans cet état pendant un "quantum de temps mémoire" qui sera très grand devant le "quantum d'Unité Centrale". De tels Schedulers ont été présentés en [19], [13] et [41].

De plus, il faut pouvoir évaluer le nombre de pages dont la tache a besoin. Une première approche, très grossière, peut être de décider que toutes ont besoin d'un même nombre de pages. Il est toutefois plus satisfaisant d'associer à chaque programme la caractéristique "nombre de pages nécessaires à un bon fonctionnement". Le problème est alors de mesurer cette caractéristique. L'approche la plus complète consiste à réserver un nombre de pages égal à w(t,T), taille du working-set, cette taille étant évaluée de façon permanente pour chaque tache par le système; on peut encore s'assurer que le nombre de pages demandées effectivement (w(t+T,T)) ne dépasse pas la

réservation, et, si c'est le cas, annuler cette réservation pour forcer la tache à la redemander. (Malheureusement, en l'absence d'un mécanisme cablé facilitant cette mesure de w(t,T), cette méthode peut être très coûteuse).

On peut adjoindre à ces stratégies des algorithmes de remplacement (choix du cadre à vider) qui travaillent de façon locale : une faute de page d'un utilisateur ne provoque alors jamais la disparition d'une page d'une autre tache (DENNING [22]). Cela garanti alors l'absence de thrashing, (mais réalise une sorte de "partition", toute tache utilisant autant de pages qu'elle en a réservé ; cependant cette réservation est réajustée dynamiquement).

On peut travailler à un niveau plus fin : si l'on connaît la composition exacte du working-set W(t,T), on peut décider de n'allouer l'unité centrale qu'à un utilisateur qui a son working-set en mémoire. On opère alors un chargement a priori des pages de l'utilisateur, au moment de la réservation; une fois celles-ci prêtes, on peut activer l'utilisateur qui a une bonne chance de travailler quelque temps sans défaut de page.

Le dispatcher de CP [47] combine ces deux méthodes de réservation de place et de connaissance du Working-Set : les tâches élues en mémoire ont toujours droit à un nombre de cadres égal à la taille de leur W.S.; et celui-ci est connu, l'algorithme de remplacement ne vidant jamais des pages appartenant au W.S. d'une tache élue en mémoire. W(t,T) est déterminé à la fin de chaque "quantum d'Unité Centrale", par consultation des "used bits" qui sont positionnés par les références aux pages ; les pages qui n'ont pas servi sortent du working-set et pourront disparaître.

Les pages utilisées simultanément par plusieurs taches compliquent le problème, car on ne sait pas s'il faut ou non les inclure dans le workingset; si elles sont connues a priori, on peut les traiter à part. Sinon, si on les compte lors du calcul de w(t,T), on risque de fonctionner en dessous des capacités du système.

Pour notre part, nous avons réalisé un système qui tient compte dans une certaine mesure de ces stratégies :

- A chaque tache est associé un "working-set" minimal et fonctionnel de 3 pages, une par voie. Les taches ont connaissance de celui-ci, ce sont elles-mêmes qui en précisant la composition, par la primitive DEMANDER.

- Les trois pages sont toujours chargées par avance, et l'allocation de l'unité centrale n'est faite qu'aux taches qui ont ce "working-set" présent en mémoire. Une tache activée pourra donc travailler pendant un minimum de temps.
- Le nombre de taches utilisant des pages est limité : les taches font périodiquement un :

  P(n, place-de-verrou)

  et le V correspondant, de façon à "réserver" ces places. Une tache n'étant éligible que si ses 3 pages sont présentes, la condition $n \geqslant 3$ est nécessaire pour écarter la possibilité de **blocage** ($n \geqslant 3$ + nombre de cadres buffers si elle utilise des buffers). Mais n peut être choisi plus grand, constant ou fonction du programme déroulé, pour que chaque tache dispose d'un nombre de pages plus satisfaisant.

Nous pensons que ces trois points doivent assurer un fonctionnement correct du système SOCRATE; cependant, seules des mesures sur le fonctionnement du système en charge moyenne ou forte pourront confirmer ce point de vue.

CONCLUSION

Nous avons voulu présenter ici l'architecture d'un système d'exploitation, permettant à un grand nombre d'utilisateurs d'accèder à des bases de données connectées.Les idées dégagées au cours de l'élaboration de ce système nous ont paru suffisamment générales pour mériter une présentation un peu approfondie, et être comparée avec celles que contiennent d'autres systèmes. Certaines d'entre elles s'avèrant très générales, il a paru intéressant d'adjoindre au système des fonctions et des compilateurs qui en font un système d'exploitation plus universel. Citons la gestion de tous les espaces-disques sous forme d'espaces virtuels, et le système de nomination que cela recouvre; la construction d'une machine "logique" aux concepts simples permettant la réalisation facile et cohérente d'un superviseur; une gestion unifiée des attentes sur des ressources autres que la mémoire; l'élimination totale de tout dynamisme en mémoire centrale, qui évite tout danger de saturation.

Il est certain qu'en abordant de cette façon la conception d'un système d'exploitation, nous avons négligés certains problèmes, et que le résultat n'est pas aussi général que nous l'aurions souhaité. Entre autres, beaucoup de problèmes sont résolus à la compilation, ce qui allège certes le fonctionnement du système, mais introduit des restrictions d'utilisation.

Il reste maintenant à faire des mesures sur le système réalisé, dans des conditions de charge normales, afin de justifier les options prises, et d'ajuster certains paramètres qui restent à déterminer. Des optimisations nombreuses sont encore possibles, mais seule l'expérience nous apprendra si elles sont réellement efficaces. De même, nous pensons qu'il est possible de réaliser de façon cablée ou microprogrammée certains mécanismes et d'améliorer ainsi de façon considérable le fonctionnement du système; là encore, des mesures devraient nous permettre de déterminer quel gain on peut attendre de ces compléments hardware.

B I B L I O G R A P H I E

[1] J.R. ABRIAL, J. BAS, G. BEAUME, G. HENNERON, R. MORIN, G. VIGLIANO
Projet SOCRATE(1) : Spécifications Générales, Grenoble, Août 1970

[2] G. VIGLIANO - Projet SOCRATE (2.1) : Langage de Requête
Thèse, Grenoble, décembre 1970

[3] G. BEAUME - Projet SOCRATE (2.2) : Gestion des Mémoires
Thèse, Grenoble, décembre 1970

[4] R. MORIN - Projet SOCRATE (2.3) : Interprétation et Compilation
Thèse, Grenoble, décembre 1970

[5] Projet SOCRATE (3): Manuel d'utilisation du prototype

[6] J.R. ABRIAL, J. BAS, J.P. CAHEN, J.C. FAVRE, G. MAZARE, R. MORIN
Projet SOCRATE (4) : Données, Mémoires et Utilisateurs d'une Banque de Données, Journées AFCET (Banque de Données), Aix, juin, 1971

[7] J.P. CAHEN - Projet SOCRATE (5) : Liaisons bilatérales SOCRATE - Bandes magnétiques format standard, Grenoble, octobre 1971

[8] J.R. ABRIAL, J.P. CAHEN, J.C. FAVRE, D. PORTAL, G. MAZARE, R. MORIN
Projet SOCRATE (6) : Nouvelles spécifications (version 3), Grenobl Septembre 1972

[9] J.R. ABRIAL, J.P. CAHEN, J.C. FAVRE, D. PORTAL, G. MAZARE, R. MORIN
Projet SOCRATE (7) : L'architecture "système" du système SOCRATE Grenoble, Février 1972

[10] F. ANCEAU : Synchronisation dans les systèmes hiérarchisés de Processus-Processeurs, Congrès AFCET 1972, Grenoble, brochure 2

[11] J.P. ANDERSON : Program Structures for Parallel Processing
CACM 8, pp 786-788, décembre 1965

[12] A. BENSOUSSAN, C.T. CLINGEN, R.C. DALEY : The Multics Virtual Memory : Concepts and Design, CACM 15, mai 1972

[13] C. BETOURNE, J. BOULENGER, J. FERRIE, C. KAISER, S. KRAKOWIAK, J. MOSSIERE : Présentation Générale du système ESOPE
Notion d'espace virtuel dans le système ESOPE
Allocation de ressources dans le système ESOPE
AFCET, Congrès d'Informatique, Paris, septembre 1970, brochure 1.

[14] C. BOULLE : Certains aspects de réalisation de SIRIS 8
AFCET, Congrès d'Informatique, Paris, septembre 1970, brochure 1.

[15] C.I.I. - IRIS 80 : Manuel de présentation 3418 P1/Fr

[16] M.E. CONWAY : A Multiprocessor System Design
AFIPS Conf. Proc. 24, Spartan Books, Baltimore 1963, pp 139-146

[17] P.J. COURTOIS, F. HEYMANS, P.L. PARNAS : Concurrent control with "Readers" and "Writers" , CACM 14, octobre 1971, pp 667-668

[18] R.C. DALEY, J.B. DENNIS : Virtual Memory, Processes and Sharing in MULTICS, CACM 11, mai 1968, pp 306-312

[19] P.J. DENNING : The Working Set Model for Program Behavior
CACM 11, Mai 1968, pp 323-333

[20] P.J. DENNING : Virtual Memory - Computing Surveys, vol. 2, n° 3 (septembre 1970), pp 153-189

[21] P.J. DENNING, S.C. SCHWARTZ : Properties of the Working-Set Model
ACM Third Symposium on Operating Systems Principles, octobre 71,
Stanford University, pp 130-140

[22] P.J. DENNING : On Modeling Program Behavior, SJCC 1972, pp 937-944

[23] J.B. DENNIS, E.C. VAN HORN : Programming Semantics for Multiprogrammed
Computations, CACM 9, mars 1966, pp 143-155

[24] E.W. DIJKSTRA : Solution of a Problem in concurrent Programming Control
CACM 8, Septembre 1965

[25] E.W. DIJKSTRA : Cooperating Sequential Process - Programming Languages
F. Genuys, Ed. Academic Press, New York, 1968

[26] E.W. DIJKSTRA : The Structure of the "THE" - Multiprogramming System
CACM 11, mai 1968, pp 341-346

[27] E.W. DIJKSTRA : Hierarchical Ordering of Sequential Process
Acta Informatica, vol. 1, Fasc. 2, pp 115-138, 1971

[28] W.B. EASTON : Processes Synchronization without Long-Term Interlock
ACM Third Symposium On operating Systems Principle, octobre 71
Stanford University, pp 95-100

[29] J. FERRIE, J. MOSSIERE : ESOPE : Gestion des Processus et Partage des
Ressources, Cahiers de 1'IRIA, n° 8, janvier 1972

[30] P. GILBERT, W.J. CHANDLER : Interference Between Communicating Parallel
Processes, CACM 15, juin 1972, pp 427-437

[31] A.N. HABERMANN : Prevention of Systems Deadlocks
CACM 12, juillet 1969, pp 373-377+385

[32] A.N. HABERMANN : Synchronization of Communicating Processes
CACM 15, mars 1972, pp 171-176

[33] P.B. HANSEN : The Nucleus of a Multiprogramming System
CACM 15, juillet 1972, pp 574-578

[34] P.B. HANSEN : Short Term Scheduling in Multiprogramming Systems
ACM Third Symosium On Operating Systems Principle
octobre 1971, Stanford University, pp 101-105

[35] P.B. HANSEN : A Comparison of Two Synchronizing Concepts
ACTA Informatica 1 (1972), pp 190-199

[36] P.B. HANSEN : Structured Multiprogramming
CACM 15, juillet 1972, pp 574-578

[37] J.W. HAVENDER : Avoiding Dead-lock in Multitasking Systems
IBM System Journal, J-2, 7, 1968

[38] R.C. HOLT : Comments on Prevention of System Deadlocks,
CACM 14, janvier 1971, pp 36-38

[39] I.B.M. : S.R.L. O.S. 360 : Supervisor and Data Management Services
File S 360-36, Form C28-6646

[40] I.B.M. : IBM System/370 : Principle of Operation
GA 22-7000-2

[41] B.W. LAMPSON : A Scheduling Philosophy for Multiprocessing Systems
CACM 11, mai 1968, pp 347-360

[42] J.P. LE HEIGET : Généralisation de la notion d'espaces virtuels sous les systèmes CP-67/CMS, Thèse, Grenoble, Juillet 1972

[43] J.E. MURPHY : Ressource Allocation with Interlock Detection in a Multi-task System, FJCC, vol. 33, P^t 2, pp 1169-1176

[44] J.D. NOE : A Note on Representation of Operating Systems
Computer Science Group - University of Washington, Seattle
Technical Report, NO-70-09-03, septembre 1970

[45] D.L. PARNAS, A.N. HABERMANN : Comment on Deadlock Prevention Method
CACM 15, septembre 1972

[46] S.S. PATIL : Limitation and Capabilities of Dijkstra's Semaphore Primitive for coordination among Processes
M.I.T. Project MAC, C.S. Group Memo 57, février 1971

[47] J. RODRIGUEZ-ROSELL, J.P. DUPUY : The Design, implementation and evaluation of a Working-Set Dispatcher
IBM Centre Scientifique de Grenoble, Etude FF2.0134

[48] J. H. SALTZER : Traffic Control in a Multiplexed Computer System
MAC-TR-30 (Thesis), M.I.T. Cambridge, Mass., juillet 1966

[49] P. ULLMANN : Cooopération entre Taches et Dispatching
SOC-PG-PU-3, EMP Fontainebleau, janvier 1971

[50] R.W. WATSON : Timesharing System Design Concepts
Mac-Graw Hill, 1970

[51] N. WIRTH : On Multiprogramming, Machine Coding, and Computer Organization, CACM 12, Septembre 1969, pp. 489-498

ANNEXE

PROTECTIONS D'ACCES AUX BASES : MECANISMES AUTOMATIQUES

IMPLEMENTATION - Novembre 1972

1 - Les "programmes compilés" et les "programmes interprétatifs"
- 1.1. Primitives
- 1.2. Algorithmes
- 1.3. Références
- 1.4. Place de verrou

2 - Organisation interne
- 2.1. Table des verrous - verrous "libres"
- 2.2. Existence des entités
- 2.3. Algorithmes des divers point d'entrée
  - 2.3.1. Verrou (Lect/maj)
  - 2.3.2. Noverrou (Nolect/Nomaj)
  - 2.3.3. Verif
  - 2.3.4. Prplace, Rdplace
  - 2.3.5. Pchain, Vchain

3 - Suppression d'un programme, ou mise-à-jour de son "corps"

4 - Problème des références et des X en suppression

NOTE TECHNIQUE N° 4 - GUY MAZARE - 10 NOVEMBRE 1972

Ces mécanismes de protection sont proches de ceux décrits dans SOCRATE(6) Nouvelles Spécifications - Chapitre III. Ils ne garantissent que la cohérence élémentaire des informations (avec la restriction qui sera détaillée en § 4). Ils assurent un fonctionnement sans "casse" dans tous les cas.

1 - LES "PROGRAMMES COMPILES" ET LES "PROGRAMMES INTERPRETATIFS"

1.1. Primitives

Ces programmes utilisent des primitives fournies par le noyau et décrites au paragraphe 2. Tous les verrouillages portent sur une réalisation d'entités, et laissent libres d'éventuelles entités internes à la réalisation d'entités verrouillée .

Ce sont :

LECT(adresse virtuelle d'entité) :
passer en lecture sur cette entité, interdire toute mise-à-jour

NOLECT(adresse virtuelle) :
libérer cette entité

MAJ (adresse virtuelle) :
passer en mise-à-jour, interdisant toute lecture ou toute autre mise-à-jour

NOMAJ (adresse virtuelle) : libérer

VERIF (adresse virtuelle, lect ou maj) : vérifier l'existence d'une entité et faire Lect ou Maj dessus

PRPLACE(n) : prendre n places de verrou

RDPLACE(n) : en rendre n

PCHAIN : prendre la permission de parcourir un anneau de référence

VCHAIN : la rendre

Remarques :

- Toutes les primitives qui demandent des ressources provoqueront une attente de la tâche si la ressource est occupée.

- Les primitives LECT/MAJ(adresse virtuelle) protègent toute caractéristique de l'entité adresse virtuelle, ainsi que compteur et chaîne de bits des entités internes à adresse virtuelle.

Ainsi, dans l'exemple d'entités imbriquées :

```
ENTITE PERSONNE
       NOM
       ENTITE VOITURE
              NUMERO
```

Quand une tâche a fait Lect (personne P1), personne ne peut supprimer ou ajouter des voitures à P1 (pour cela il faut d'abord faire MAJ(P1)). Par contre on peut changer le numéro d'une voiture V_{11} de P1 en faisant MAJ(V_{11}).

1.2. Algorithmes

1.2.1. Programmes compilés

Quand on évalue une citation, en descendant d'entité en entité, on ne s'assure de rien : seule la lecture de bits non nuls dans les chaînes d'existence des entités nous donne à penser qu'on va chercher une entité qui existe.

Au moment où l'on arrive sur la caractéristique citée, que ce soit par une commande i ou m, ou dans un filtre, on fait :

```
{ Vérif (entité contenant la caractéristique, lect/maj)
  Si elle n'existe pas sortir
  Lire la caractéristique
  Nolect/nomaj
```

Si on consulte séquentiellement une classe d'entités (POUR TOUT, Entité filtrée, ...), on opère ainsi en relisant la chaîne de bits à chaque fois, mais sans relire le compteur. On s'arrête soit quand on a trouvé un nombre d'entités existantes égal au compteur, soit quand on a consulté un nombre de bits égal au nombre maximum de réalisations d'entités.

1.2.2. Suppression : le "killer"

Un programme compilé appelle le killer en lui passant :

avatuer : adresse virtuelle de l'entité à tuer
avpère : adresse virtuelle de l'entité l'englobant

Le killer fait :

- MAJ (avpère)
- Lire la chaîne de bits; si bit nul
  - NOMAJ(avpère)
  - Sortie
- bit ← 0
- décrémenter le compteur
- MAJ(avatuer)
- supflag et existflag (avatuer) ← ON
- NOMAJ (avpère)
- call supanneau (suppression de l'anneau des références)
- call supréf (suppression des références)
- call supent (suppression des entités internes)
- NOMAJ (avatuer)
- Sortie

On détaillera supref et supanneau en 1.3.

La suppression d'entités internes fait :

SUPENT

(on a trouvé un bit à 1)
effacer le bit
décrémenter le compteur
MAJ (adresse virtuelle de cette sous-entité)
Existflag et Supflag ← ON
call supanneau, supref, supent
.....
NOMAJ (cette sous entité)

1.2.3. Le créateur

Il fonctionne exactement comme le killer, et comme décrit en 3.4.4. de SOCRATE (6).

1.3. Références

Le mécanisme implémenté est des plus simples : une seule tâche à la fois peut manipuler une référence. Cela est assuré par les primitives Pchain et Vchain. Il n'y a pas de dead-lock à craindre puisque, quand on a pris cette ressource, on ne demande plus rien. (En particulier, quand on parcourt l'anneau pour remettre les références à zéro, on n'avertit pas les entités contenant ces références).

Lire une référence :

Pchain
Lire la référence
Vchain

Mise à jour de référence

/* on est en mise à jour sur l'entité englobante */
Call Supref
Vérif (nouvelle référence)
Pchain
Modification de l'anneau de l'entité nouvellement référenciée
Vchain

Supanneau (A)

Pchain
Parcourrir l'anneau, remettre référence et chaînage circulaire à zéro
Vchain

Supref :

Si la référence était ≠ 0 (soit pointant B)
Pchain
Modification de l'anneau de B
Référence ← 0
Vchain

1.4. Place de verrou

Le nombre des places de verrou étant limité par la taille de la table, tout programme compilé doit faire les demandes de places avant exécution de toute commande :

Exemple :

Programme SOCRATE Demande des places à l'exécution
Pour toute personne
i nom ---------- prplace(2) --- i --- rdplace (2)
m age = 20 --------- prplace(2) --- m --- rdplace (2)
t une voiture 1 --- prplace(6) --- call killer --- rdplace (6)
fin

Six places suffisent pour les programmes interprétatifs, puisque la profondeur de la structure est limitée à 6.

2. ORGANISATION INTERNE

2.1. Table des verrous

Il existe une table de 20 (par exemple) processeurs-verrous dans le système. Ces verrous sont occupés si quelqu'un a fait LECT ou MAJ dessus (D ≠ 0), libres sinon :

+ créer un verrou signifie initialiser une place de verrou libre avec l'adresse virtuelle demandée.
+ lâcher un verrou signifie le rendre libre; mais on conserve son adresse virtuelle; on se contente de le remettre dans la file des verrous libre et de basculer Libflg ON.

Ainsi, on conserve dans la table les verrous le plus longtemps possible; quand on récupère une place de verrou, on le fait en ôtant une place de la file de verrous libres, en écrasant son adresse virtuelle, et en mettant Libflag OFF. On choisit pour cela le plus ancien verrou libre (LRU).

Au démarrage, la table est initialisée avec des verrous tous libres et d'adresse virtuelle nulle.

2.2. Existence des entités

Quand on a fait LECT ou MAJ sur une entité, son existence n'est pas garantie. Par contre VERIF la garantie. Il bascule alors exisflg ON.

De même la suppression garantit que l'entité n'existe plus; elle bascule existflg ON et Supflg ON.

Quand on trouve un verrou dans la table, qu'il soit libre ou non :

- si existflg est OFF : on ne sait rien de son existence
- si existflg ON { et supflg OFF : elle existe
  { et supflg ON : elle n'existe plus

Si le verrou n'est pas dans la table, ou si existflg est OFF, il faut aller voir la chaîne de bits, en se protégeant par l'entité supérieure.

(Remarquons que toute manipulation de ces flags est faite alors qu'on est protégé par l'entité supérieure). Cela permet d'accélérer considérablement le travail de VERIF.

2.3. Algorithmes des différents points d'entrée

2.3.1. Verrou (av, nb) : LECT/MAJ

```
/* si nb = 1 : LECT ; si nb = 1000 = MAJ */
Solo
Chercher le verrou adresse virtuelle
S'il n'existe pas [ le créer (av; existflg, Supflg, libflg OFF)
                  | l'occuper (D ← nb)
                  | Tutti
                  | Sortie
Si libflg ON [ libflg OFF
             | déchaîner des verrous libres
             | l'occuper (D ← nb)
             | Tutti
Sinon /* il peut être déjà occupé */
             [ P nb, ce verrou
Sortie
```

2.3.2. Noverrou (av,nb) : NOLECT/NOMAJ

```
/* nb = 1 : NOLECT  nb = 1000 : NOMAJ */
Solo
Chercher le verrou av (il existe !)
V nb,(verrou av)
Solo
Si N=D=0 et libflg OFF alors [ libflg ON
                               chaîner dans les verrous libres
Tutti
```

Nota : Libflg évite que deux tâches faisant Noverrou ne chaînent toutes 2 le verrou dans la liste libre.

2.3.3. Vérif (av, nb)

```
Si av = 0 alors verrou (0,nb) /* av = 0 = fichier : existe
                                 toujours */
Chercher le verrou dans la table
Si il n'y est pas, ou si existflg OFF
        [ call avadv /* analyse l'av et répond avpère,
                        avchbit, etc... */
          verrou (avpère,1)
          Lire chaîne bits
          Si bit ON  [ verrou (av,nb)
                       existflg ← ON
                       noverrou (avpère,1)
                       sortie (av=1)
          Si bit OFF [ Noverrou (avpère,1)
                       Sortie (av=0)
Si supflg ON alors sortie (av=0)
Sinon /* l'entité existe */
        si libflg ON  [ libflg OFF
                        déchaîner
                        occuper (D ← nb)
                        tutti
                        sortie (av = 1)
```

si libflg OFF P nb, ce verrou
sortie (av = 1)

Au retour, av = 0 signifie l'entité n'existe pas, av ≠ 0 l'entité existe et a été verrouillée.

2.3.4. Prplace (av,n)

Si n > nb-de-places alors n = nb-de-places
P n,Placedeverrou

Rdplace (av,n)

Si n > nb-de-places alors n = nb-de-places
V n,Placedeverrou

Le processeur Place-de-verrou a un DM = nb-de-places.

2.3.5. Pchain

P 1,Procchaine

Vchain

V 1,Procchaine

Le processeur Procchaine est de DM = 1

3. SUPPRESSION D'UN PROGRAMME, OU MISE-A-JOUR DE SON "CORPS" (le code)

Cela n'a pas encore été implémenté au 10 Novembre 1972.

On ne veut pas mobiliser une place de verrou pour tout le temps de l'exécution d'un programme : on prévoira donc une sorte de verrou, statique, dans l'entité programme (un compteur).

D'autre part, on ne veut pas provoquer l'attente de quelqu'un qui tue un programme utilisé par ailleurs : on va refuser sa transaction.

Ce que nous appelons "entité programme" est, dans l'espace "fichier", le bloc de contrôle du programme; il comporte donc un compteur CPTR.

Exécution d'un programme (avprog = adresse de l'entité programme)

```
Vérif (avprog, 1000)
Si il n'existe plus        sortir ('programme inconnu')
Cptr ← cptr+1    (increm)
NOVerrou (avprog,1000)
    }
exécution
    }
Verrou(avprog,1000)  /* il existe ! */
cptr ← cptr-1    (increm)
NOVerrou (avprog,1000)
```

Remarque : Pour décrémenter le compteur CPTR il faut se protéger - sauf si on garantit que INCREM est ininterruptible.

```
Killer(avprog)
  Si on veut tuer un programme :
  Vérif (avprog,1000)
  Si cptr ≠ 0      Noverrou (avprog,1000)
                   Sortir ('refusé')
  sinon
    {
    tuer   idem tuer une entité qu'on a déjà verrouillée
    }
  Noverrou (avprog,1000)
```

Remarques : I - Cette méthode s'apparente de très près à la nouvelle méthode de notation des références par compteur et "nettoyeur" envisagée au paragraphe suivant.

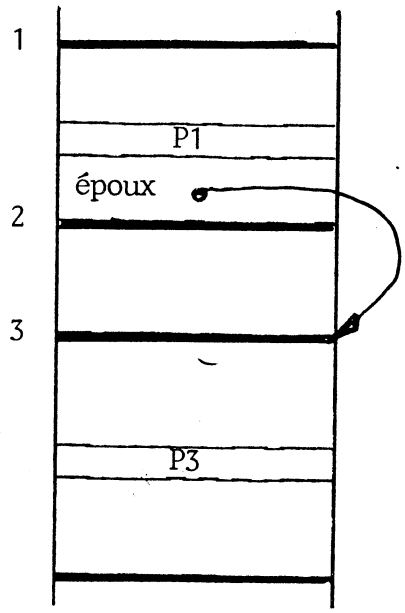
II - La mise-à-jour du corps d'un programme, si elle est permise, devra opérer comme le killer.

4. PROBLEME DES REFERENCES ET DES Xi EN SUPPRESSION

Prenons l'exemple :

```
Entité Personne
       nom mot
       époux référence personne
fin
```

Et supposons qu'à l'instant T on ait :



et qu'un utilisateur pose :

I NOM DE EPOUX DE PERSONNE 1 ?

Il va :

```
Trouver personne 1
Lire la référence = personne 3
Faire Vérif (pers.3,1)          <-------  (1)
Lire le nom
Noverrou (pers.3,1)
```

Mais si en (1) quelqu'un tue la personne P3 et génère une autre personne P4 celle-ci peut revenir se créer à la place de personne 3; la référence époux sera remise à 0, mais trop tard. Et on va trouver une personne qui n'a rien à voir avec l'époux de P1.

Cela peut aussi se produire si l'on fait :

```
M X1 = une personne ayant nom =P1
EXT
I NOM DE X1
```

Si pendant l'EXT la suppression et la création a eu lieu, on va retrouver quelqu'un qui n'a plus rien à voir avec P1.

Ce sont là des choses possibles avec cette implémentation. Le remède passe par une nouvelle gestion des emplacements, qui est envisagée par ailleurs.

a - Méthode de la chaîne des tombes

Déjà exposée en SOCRATE(6). Elle résoud ce problème.

b - Méthode du compteur et du "nettoyeur".

Dans chaque entité existe un compteur des gens qui la référencent; quand on tue l'entité, on n'efface pas ce compteur; on ne crée que dans un emplacement où le compteur est nul.

Quand on met une référence sur une entité, on incrémente ce compteur; de même quand on charge une adresse virtuelle d'entité dans un X. On le décrémente quand on supprime la référence, ou qu'on écrase un X (ou à la fin du programme utilisant les X, ou au "Logout" si on conserve les "statiques").

Il existe par ailleurs dans le système une tâche qui est chargée uniquement de remettre à zéro toutes les références pointant sur une entité donnée. Si en suppression le compteur n'est pas nul, on dit à cette tâche de nettoyer les références sur cette entité. (Cette tâche communique avec les autres via une "boite-aux-lettres" située dans l'espace virtuel).

c - Méthode du "nom historique".

A chaque entité créée est associé un nom historique ou nom universel, qui lui est propre et ne sera jamais réutilisé ; il est noté avec l'entité.

Les références et les X sont formés par l'adresse virtuelle et le nom historique ; toute utilisation de référence ou d'un X est alors validée par la comparaison de noms historiques.