Processus cyclique et appel procédural

Jacky Estublier

THESE

présentée à

Institut National Polytechnique de Grenoble

pour obtenir le grade de
DOCTEUR de 3ème cycle Informatique

par

Jacky ESTUBLIER

PROCESSUS CYCLIQUE ET APPEL PROCEDURAL

Thèse soutenue le 3 avril 1978 devant la Commission d'Examen

Président : S. KRAKOWIAK
J. BRIAT
P. DARONDEAU
Examinateurs : C. KAISER
J. MOSSIERE
J.P. VERJUS

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Monsieur Philippe TRAYNARD : Président

Monsieur Pierre-Jean LAURENT : Vice Président

PROFESSEURS TITULAIRES

MM.	BENOIT Jean	Radioélectricité
	BESSON Jean	Electrochimie
	BLOCH Daniel	Physique du solide
	BONNETAIN Lucien	Chimie minérale
	BONNIER Etienne	Electrochimie et électrométallurgie
	BOUDOURIS Georges	Radioélectricité
	BRISSONNEAU Pierre	Physique du solide
	BUYLE-BODIN Maurice	Electronique
	COUMES André	Radioélectricité
	DURAND Francis	Métallurgie
	FELICI Noël	Electrostatique
	FOULARD Claude	Automatique
	LESPINARD Georges	Mécanique
	MOREAU René	Mécanique
	PARIAUD Jean-Charles	Chimie-Physique
	PAUTHENET René	Physique du solide
	PERRET René	Servomécanismes
	POLOUJADOFF Michel	Electrotechnique
	SILBER Robert	Mécanique des fluides

PROFESSEUR ASSOCIE

M.	ROUXEL Roland	Automatique

PROFESSEURS SANS CHAIRE

MM.	BLIMAN Samuel	Electronique
	BOUVARD Maurice	Génie mécanique
	COHEN Joseph	Electrotechnique
	LACOUME Jean-Louis	Géophysique
	LANCIA Roland	Electronique
	ROBERT François	Analyse Numérique
	VEILLON Gérard	Informatique fondamentale et appliquée
	ZADWORNY François	Electronique

MAITRES DE CONFERENCES

MM.	ANCEAU François	Mathématiques appliquées
	CHARTIER Germain	Electronique
	GUYOT Pierre	Chimie minérale
	IVANES Marcel	Electrotechnique
	JOUBERT Jean-Claude	Physique du solide
	MORET Roger	Electrotechnique nucléaire
	PIERRARD Jean-Marie	Mécanique
	SABONNADIERE Jean-Claude	Informatique fondamentale et appliquée
Mme.	SAUCIER Gabrièle	Informatique fondamentale et appliquée

MAITRE DE CONFERENCES ASSOCIE

M.	LANDAU Ioan	Automatique

CHERCHEURS DU C.N.R.S. (Directeur et Maîtres de Recherche)

MM.	FRUCHART Robert	Directeur de Recherche
	ANSARA Ibrahim	Maître de Recherche
	CARRE René	Maître de Recherche
	DRIOLE Jean	Maître de Recherche
	MATHIEU Jean-Claude	Maître de Recherche
	MUNIER Jacques	Maître de Recherche

Je remercie S. KRAKOWIAK d'avoir accepté de présider mon jury et J.P. VERJUS de m'avoir autorisé à présenter cette thèse.

Je tiens à remercier P. DARONDEAU qui m'a dirigé, conseillé et corrigé durant toute la période de rédaction, ainsi que tous les membres de l'équipe OURS puisque j'expose ici une partie d'un travail collectif qui s'étend sur plusieurs années, et tout particulièrement J. BRIAT, notre responsable scientifique, auquel l'équipe doit la plupart des idées de base du projet OURS et, entre autres, celles exposées ici.*

Je remercie J. RAYMOND pour la relecture systématique et la mise en forme de cette thèse et pour son amitié, P. LAFFORGUE et X. ROUSSET de PINA pour les conseils, les discussions et les corrections apportées au manuscrit, les personnes extérieures à l'équipe qui m'ont relu et aidé : J. MOSSIERE et C. KAISER et qui m'ont fait l'honneur de faire partie de ce jury.

Je n'oublierai surtout pas, bien qu'ils n'aient pas pris une part active dans cet ouvrage, tous les membres de l'équipe CII de l'IMAG, mes amis et amies dont la bonne humeur, l'amitié et les discussions (montagne entre autres) m'ont permis de conserver un bon moral durant la période pénible qu'est la rédaction d'une thèse.

Je remercie enfin M.J. DOREL et le Service Reprographie de l'IMAG pour la réalisation matérielle de cet ouvrage.

*Equipe OURS : J. BRIAT, J. ESTUBLIER, B. MAILLOT, J. RAYMOND, X. ROUSSET de PINA, S. ROUVEYROL, A. TARABOUT, I. VATTON.

SOMMAIRE

Chapitre IV - ASPECTS QUALITATIFS

CHAPITRE I

PRÉSENTATION

Les méthodes de conception et de réalisation de noyaux de système ont été l'objet de très nombreuses études dont nous retenons une structuration des programmes et des données inspirée des langages à structure de bloc : notion de procédures, de données gérées dynamiquement, de modules.

Si on suppose que chaque module est le siège d'activités concurrentes ou parallèles, quel mécanisme doit-on adopter pour réaliser et contrôler les communications entre les modules ? Dans la plupart des systèmes et logiciels "classiques", le mécanisme utilisé est celui de l'appel procédural. Une autre possibilité, la communication par message, a également été utilisée, mais très partiellement, jusqu'à l'apparition des réseaux et systèmes répartis qui l'ont employée massivement et quasi exclusivement.

Il nous a semblé intéressant de réaliser une étude comparative de ces deux mécanismes : l'appel procédural et la communication par message.

En nous appuyant sur l'expérience acquise par la conception et la réalisation du projet OURS, qui consiste à écrire un noyau de système où les modules communiquent uniquement par message, nous avons essayé de préciser les différences fondamentales et pratiques entre ces deux mécanismes.

Dans le premier chapitre, nous précisons le contexte de cette étude ; au chapitre II, nous effectuons une comparaison entre les quantités d'espace mémoire et au chapitre III, le temps d'exécution nécessaire à chacun de ces deux mécanismes.

Au Chapitre IV, nous analysons les différences pratiques telles qu'elles peuvent être ressenties par un concepteur ou un programmeur.

I.1. Module

Toute cette étude traite des communications entre des modules. Cette notion est en cours d'évolution (introduction des types abstraits, des langages spécialisés avec notion de module, langages de connexion de module).

Nous utiliserons tout au long de ce travail la définition suivante de module.

Un module est un programme dont seuls certains objets (dits externes) sont accessibles de l'extérieur. Ces objets externes peuvent a priori être exécutables (procédure) ou non (données). Partant de l'idée qu'une donnée ne doit être définie (pour des raisons de sécurité) qu'en même temps que les opérations autorisées sur elle, nous restreindrons par convention les objets externes d'un module à être des procédures (pouvant être des fonctions, c'est-à-dire fournir une valeur d'un type spécifié). Cette convention revient à interdire la modification d'une donnée d'un module autrement que par l'appel d'une procédure externe, la lecture étant également assurée par un appel de fonction [MO 7

Un module comprend donc :

- un ensemble de procédures externes, accessible depuis l'extérieur du module
- un ensemble de procédures } internes, inaccessible directement
- un ensemble de données } depuis l'extérieur du module.

Les avantages de la notion de module sont :
aide à la décomposition d'un logiciel, indépendance de réalisation des modules, expression homogène des modules matériels et logiciels, protection des données ... [PA 72, PA 72a].

I.2. Relations entre module

Un module peut appeler des fonctions d'autres modules pour demander leur exécution.

L'exécution de ces fonctions peut être :

- synchrone. La fonction appelée s'exécute comme un sous-programme
- asynchrone. La fonction appelée s'exécute comme un processus fils.

I.2.1. L'appel procédural

L'appel procédural entre module est le mécanisme actuellement le plus utilisé. Un seul et même processus exécute les diverses fonctions et procédures à appeler. La création de processus avec le même type d'appel (comme dans la machine B6500 [HA 68]) et la création de pile cactus n'est pas considéré ici comme appel procédural.

I.2.2. Communication par message

Un message est une suite d'informations contiguës précédé d'informations de contrôle. Ce bloc est échangé entre des entités (processus, système ...) qui désirent communiquer.

Le mécanisme de communication par message a trouvé son application principale dans la communication à distance et les réseaux. Il est par contre peu utilisé pour le passage de paramètres entre fonctions [BR 69]. On peut se demander s'il ne serait pas intéressant d'appliquer ce type de communication pour les échanges intermodule. Pour cela, une solution possible consiste à associer un processus à chaque module. Ce processus n'exécute que les procédures du module et communique avec les autres modules par messages. Nous allons montrer que ce mécanisme bénéficie de trois avantages importants :

- il est plus puissant et plus souple que l'appel procédural ;
- la taille des piles associées à un processus peut diminuer fortement ;
- il est plus efficace de lancer un processus préexistant que de changer l'espace d'exécution d'un processus.

Le projet OURS en utilisant un tel mécanisme nous a permis d'étudier dans quelle mesure on peut tirer parti de ces avantages.

Le mécanisme de communication par message peut-il concurrencer l'appel procédural ? Dans quelle mesure ? Tel est l'objet de ce travail.

I.2.3. Le projet OURS

Il n'est pas question de présenter en détail le projet OURS mais les grandes lignes et les choix principaux qui ont été effectués à sa conception.

Ce noyau de système reprend les idées maintenant classiques de décomposition modulaire et de niveaux d'astraction, avec, cependant, des diffé-

rences sensibles quant à l'implémentation qui en est faite.

Le projet consiste à écrire les deux premiers niveaux :

Hardext : (hardware extension), système de communication par message, de gestion des processus, des modules, des niveaux d'abstraction et de la machine réelle (Iris 80 bi-processeur) [IRIS]

UGR : Unités de Gestion des Ressources. Gestion de la mémoire centrale paginée [IV 76], gestion de la mémoire secondaire (disques [TR 76]), répartition des ressources, volumes ...

La couche UGR a pour rôle de fournir à l'utilisateur de cette couche une machine disposant d'espace virtuel (128 * 128 K mots) dans lequel celui-ci peut amener (lier) des segments (bloc d'information de 512 à 128 K mots) désignés par un nom universel. Cette couche fournit également des volumes physiques connus par un nom universel sur lequel l'utilisateur peut lire (lier dans son espace virtuel) ou ranger (sauvegarder depuis l'EV) des segments. UGR peut également gérer des volumes "privés", lesquels sont accédés explicitement au moyen d'instructions symboliques d'accès aux supports externes. Des instructions permettent d'effectuer des allocations de ressources, de garantir la cohérence des fichiers, de permettre la communication et la protection réciproque des processus. Les segments sont partageables.

La couche UGR est elle-même un réseau de modules réalisé en utilisant les outils fournis par la couche hardext.

La couche hardext a pour rôle de fournir :

- un mécanisme permettant de construire un système comme un réseau de modules et de fournir aux processus s'exécutant dans ces modules un moyen de communication ;
- un certain nombre de service (gestion d'horloge, échéancier, comptabilité, reprises après erreur ...) ;
- une image plus agréable du matériel : indépendance du nombre de processeurs, transparence de la structure et de la mémoire, transparence de commutation du contexte de la machine, ...

Ces mécanismes reposent donc sur un petit nombre d'éléments primitifs. Leur mise en oeuvre requiert la réalisation de deux fonctions :

- le système de communication,
- la machine IRIS.

Cette thèse présente quelques aspects de la couche hardext que nous avons contribué à concevoir et réaliser en LIS [LIS].

Hardext réécrit en LP80 [LP 80] est actuellement opérationnel.

Les caractéristiques que nous allons essentiellement analyser sont celles-ci :

- toutes les communications entre les modules se font par message,
- à chaque module, sont associés des processus chargés d'en exécuter les procédures.

I.3. Contexte de travail

I.3.1. Synchronisme

L'appel procédural est par définition synchrone. Un seul et même processus exécute le code des fonctions et procédures appelées.

Une tâche donnée est réalisée par un processus. Ce processus exécute lui-même toutes les instructions nécessaires à l'accomplissement de cette tâche. L'appel procédural ne permet donc pas l'exécution en parallèle de plusieurs parties d'une même tâche.

Remarque : Il serait possible d'offrir la même possibilité grâce à un mécanisme de communication par message. On pourrait écrire :

```
ENVOI (message X, destinataire) ;
ATTEND (message X) ;
```

Nous aurions l'équivalent d'un appel procédural, à la différence près que, dans ce cas, le processus qui exécute l'instruction ENVOI serait suspendu jusqu'au retour du message X.

Nous obtenons alors un certain nombre de processus en attente du retour d'un message.

Ces processus pourraient souvent être utilisés pour une autre activité, pendant que le message envoyé est traité par le module destinataire. Pour cela, nous pouvons désynchroniser l'envoi d'un message et la réception du retour du même message. L'instruction

ENVOI ne suspend pas le processus qui l'exécute, l'instruction ATTEND fournit un message porteur d'une nouvelle requête au module ou constituant le retour d'un message envoyé à un autre module. Le processus effectue donc un multiplexage de ses exécutions.

Ce mécanisme permet d'une part de respecter le synchronisme dans le sens où on ne permet pas l'exécution parallèle de plusieurs parties d'une même tâche et d'autre part elle permet de désynchroniser l'exécution des différents processus serveurs qui coopèrent à la réalisation d'une tâche. Le terme asynchronisme utilisé dans ce travail devra être pris dans le sens indépendance mutuelle des processus serveurs.

Les deux mécanismes que nous comparons sont donc utilisés dans un même domaine, celui de l'exécution synchrone d'une tâche.

I.3.2. Appel par valeur

Les paramètres d'une fonction peuvent être passés à la fonction appelée par valeur (recopie des paramètres dans des variables propres à la fonction appelée), par nom ou par pointeur (désignation des paramètres par leur adresse).

Si on reprend la définition d'un module donné précédemment, on constate que l'absence de variable accessible de l'extérieur ne permet pas le passage de paramètres intermodules par nom ou par pointeur.

Indépendamment de cette raison, le déplacement des paramètres du module appelant au module appelé permet une meilleure "localité" des références.

La plupart des auteurs actuellement préconisent l'appel par valeur pour les relations intermodule ; la même tendance se retrouve dans les langages pour les appels de procédure. Cependant, dans la plupart des langages, les tableaux sont passés par référence pour des raisons évidentes d'encombrement et de coût de recopie. Il nous semblerait intéressant qu'une étude soit entreprise pour voir dans quelle mesure le passage entre module des références des variables locales de grande taille (tableaux) pourrait aussi être utilisé.

I.4. Critères de comparaison

Nous avons distingué les critères quantitatifs (quantité de place mémoire nécessaire, coût en temps de calcul) des critères qualitatifs tels que l'aisance d'utilisation, la puissance, la généralité ....

Nous comparons dans les deux prochains chapitres l'efficacité (en terme de coût mémoire et de coût en temps de calcul) des deux mécanismes de communication intermodule : l'appel procédural et l'appel par message.

Dans le chapitre II, nous essaierons de quantifier la différence d'espace mémoire nécessaire au fonctionnement de ces mécanismes. Sur l'exemple précis du système OURS, nous donnerons des valeurs comparatives en fonction de plusieurs contextes d'utilisation.

Dans le chapitre III, nous analysons les différences éventuelles en temps d'exécution entre les deux mécanismes. Nous ne donnons pas de chiffres mais nous dégageons les critères et nous rappelons les expériences [Multics, ...] qui permettent de reconnaître les domaines où une méthode est plus rapide que l'autre.

Au chapitre IV, nous faisons le tour des diverses caractéristiques pour lesquelles les deux mécanismes se comportent différemment. Nous nous aidons beaucoup dans ce chapitre de l'expérience pratique acquise en programmant OURS d'abord dans un langage évolué, puis dans un langage de bas niveau.

Nous pensons que les divers renseignements que nous donnons peuvent être utiles à quiconque désirerait choisir une de ces deux méthodes ou désirerait programmer des modules avec des processus cycliques.

Chapitre II

Comparaison au niveau de l'espace mémoire consommé

Nous essayons dans ce chapitre de comparer et d'évaluer l'espace mémoire consommé par les deux mécanismes qui nous intéressent, à savoir :

- le contrôle procédural où l'appel d'un module se traduit par l'appel d'une procédure point d'entrée du module,
- et le contrôle par processus cyclique dans lequel l'appel d'un module se traduit par l'envoi d'un message à un processus s'exécutant exclusivement au sein de ce module. Nous rappelons que pour se ramener à un domaine d'utilisation identique, la deuxième méthode doit être limitée au cas de la communication synchrone ; i.e. la tâche appelante est suspendue pendant l'exécution du module appelé.

La réentrance des modules pose des problèmes particuliers d'allocation de mémoire. En conséquence, nous envisageons dans un premier temps une analyse comparative en excluant le multiplexage (c'est-à-dire que nous nous plaçons dans l'hypothèse où un module est lien d'exécution d'un seul processus).

Par ailleurs, aucune hypothèse n'est faite sur la façon dont les modules sont codés (langage employé, synchronisation interne, ...).

Dans ce chapitre, nous nous bornerons essentiellement à l'évaluation du coût de mémoire au niveau de la pile (mécanisme généralement employé pour le contrôle procédural) et au niveau des messages échangés entre processus cyclique.

Les diverses méthodes employées pour gérer la pile de contexte des appels procéduraux sont voisines. Il n'en est pas de même pour la gestion des messages et des processus cycliques. Nous n'allons pas comparer ici toutes les méthodes existantes, mais seulement d'une part la méthode classique de gestion de la pile et d'autre part quelques méthodes de communication par message et en particulier celle utilisée dans OURS.

II.1. Modules non réentrants

Un module est non réentrant s'il admet une seule exécution à la fois.

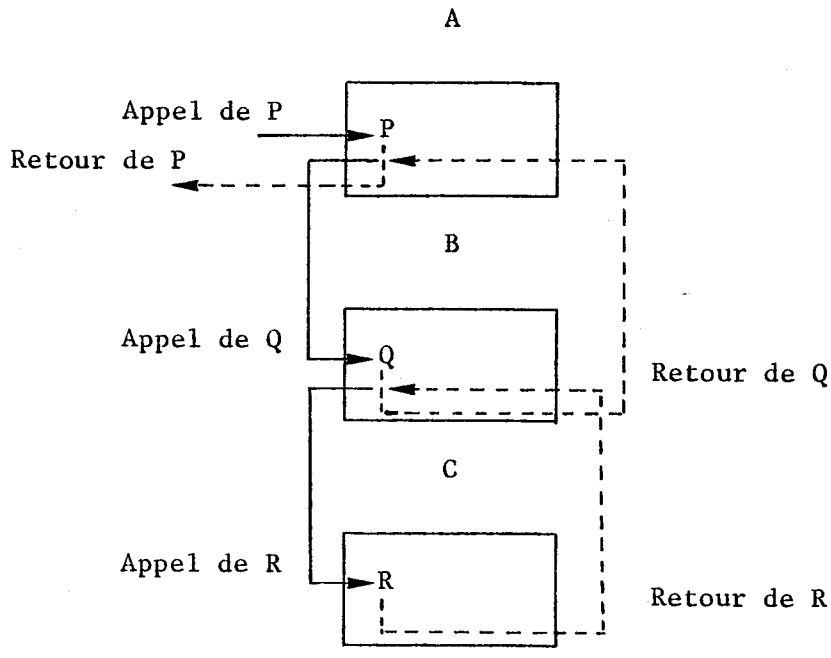
Généralement, les modules dans les systèmes sont réentrants.

Nous ne prétendons pas dans ce paragraphe décrire un cas réaliste mais seulement montrer dans un contexte simple les différences fondamentales entre les deux mécanismes à comparer. Nous allons supposer l'existence d'un seul processus et sur un exemple simple, montrer comment, par modifications successives, on peut transformer un contrôle procédural en un contrôle par processus cyclique.

II.1.1. Fonction et contenu de la pile associée au contrôle procédural

Soient trois modules A, B, C contenant respectivement les points d'entrée P, Q, R. L'exécution de P entraîne un appel de Q, lequel effectue un appel de R.

La cascade des appels et retours peut se schématiser comme suit :



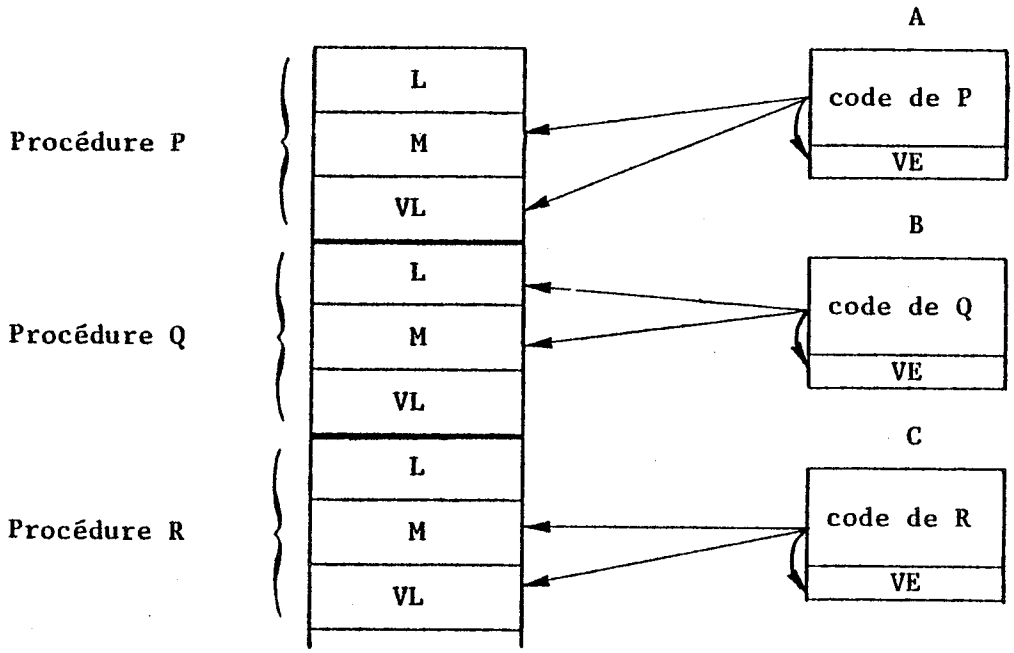
Lors de l'appel de Q, la procédure P doit sauvegarder ses variables locales (VL) et les paramètres qui lui ont été passés (M) afin de reprendre son exécution au retour de Q.

P doit, enfin, sauvegarder les liens de retour L vers la procédure qui l'a appelé.

A chaque appel, la procédure appelante doit donc sauvegarder le triplet (VL, M, L).

Les triplets (VL, M, L) sont traditionnellement conservés dans une pile.

Lors de l'exécution de R, la pile contient :



Exemple A

VE : Variables d'état

VL : Variables locales

M : Paramètres d'appel

L : Liens de retour vers la procédure appelante

⟶ : Variable que le code peut accéder.

Le code ne contient pas de variables mais il peut éventuellement contenir des constantes.

II.1.2. Gestion de la pile

Le degré d'imbrication maximal (i.e. le nombre maximal de triplets dans la pile) est difficile à estimer. Il faudrait pour cela analyser le graphe des appels intermodules. Du fait de la compilation séparée des modules, cette analyse n'est généralement pas réalisable par le compilateur, mais elle peut être effectuée à l'intérieur de chaque module par l'optimiseur global du compilateur. Elle doit donc être faite à la main ou par l'éditeur de liens.

Dans OURS (comme dans beaucoup d'autres noyaux), la structure en niveaux d'abstraction interdisant l'appel récursif des modules, le niveau d'imbrication est fixe et borné (il est au plus égal au nombre de modules). De plus, la taille de VL et de M varie pour chaque point d'entrée, la détermination statique de la taille maximum de la pile est délicate. Un langage comme LIS [LIS] qui permet de compiler globalement tout un noyau de système, pourrait effectuer ce calcul ; ce n'est pas le cas.

Pour cette raison, on est tenté d'allouer dynamiquement de l'espace pour la pile (la pile est alors constituée par un ensemble de triplets (VL, M, L) chaînés entre eux).

Cette méthode présente l'avantage de ne pas avoir à connaître par avance la taille maximale de la pile et de n'utiliser, à un instant donné, que l'espace mémoire nécessaire. En fait, ce deuxième point ne constitue pas un avantage important. En effet, l'exécution de certaines instructions s'effectue de façon asynchrone (par exemple, appel d'un canal), ce qui brise l'enchaînement procédural des appels et conduit à la suspension du processus.

Ces instructions sont liées à la nature asynchrone de certaines parties du matériel ; elles se trouvent donc dans la couche logicielle la plus "basse" et sont de ce fait exécutées au bout d'une cascade d'appels procéduraux, donc lorsque la pile a atteint un maximum relatif.

L'expérience montre que le temps d'exécution d'une entrée-sortie est relativement important (délais de transfert) et que la fréquence de telles demandes est assez élevée. A titre d'exemple, rappelons que,

sur CP-67, seuls deux à trois processus parmi trente à quarante sont réellement actifs à un moment donné [LE 77, DU 74].

La quasi-totalité des autres processus sont suspendus sur entrée-sortie-en-cours, ou sur attente-de-message en provenance de la console utilisateur.

Pour tous ces cas, les processus sont suspendus dans les couches basses du noyau et leur pile possède ou approche la taille maximale.

L'allocation dynamique ne permet donc pas toujours de gagner beaucoup de place. De fait, nous constatons que de nombreux systèmes allouent statiquement l'espace nécessaire au "contexte" d'un processus (pile et renseignements sur ce processus) :
512 mots pour SIRIS8, 512 mots pour GEMO.

L'encombrement mémoire de la pile n'est pas toujours négligeable. Il ne présente pas un inconvénient sérieux pour les systèmes de traitement par lots où le nombre de processus existant simultanément est faible. Inversement, pour les systèmes à temps partagé dont le but est de servir le plus grand nombre possible d'usagers (donc de processus), l'espace mémoire occupé par les diverses piles est assez important.

Il est donc intéressant de voir si la taille des piles peut être réduite.

II.1.3. Utilisation des variables locales et des paramètres

On distingue généralement deux catégories de variables dans un module :

- les variables rémanentes : ce sont des variables ayant la durée de vie du module ;
- les variables locales : ce sont des variables dont la durée de vie est celle d'une exécution du module (entre deux exécutions, leur valeur est non significative).

On définit par ailleurs l'état d'un module : à tout instant (en déhors de l'exécution d'une procédure externe), les variables d'un module ont des valeurs dont l'ensemble définit l'état du module. Une autre façon de définir l'état qui ne fait pas intervenir les

variables consiste à donner la suite d'appels de procédures (avec les valeurs des paramètres) entre l'instant initial (ou tout instant où l'état est défini) et l'instant actuel. L'état d'un module résume donc en ce qui concerne son comportement futur toute l'histoire antérieure du module [MO 77].

Les variables qui définissent l'état sont appelées variables d'état ; il s'agit d'une part des variables rémanentes et d'autre part d'une catégorie de variables dont il est rarement fait mention. Ce sont des variables dont la valeur est significative entre deux exécutions du module (ce ne sont donc pas des variables locales et elles jouissent d'une certaine rémanence), mais leur durée de vie est inférieure à celle du module.

Nous citerons l'exemple d'une table des pages associée à un processus pour un module gérant la pagination. Cette table des pages est créée puis détruite au cours de la vie du module. Il en est de même pour des descripteurs de segment au niveau du module de segmentation ... Le plus souvent, c'est le type de ces variables (type "table de page", type "descripteur de segment") qui est rémanent au module et non les diverses variables de ce type.

Ces variables, que l'on pourrait qualifier de variables-d'état-non-rémanentes, imposent aux modules une gestion de mémoire libre (pour leur création et leur destruction), tandis que les variables rémanentes sont statiquement réservées et que les variables locales sont allouées dynamiquement dans la pile des processus.

Cette classe de variables revêt une grande importance pour un module puisque leur gestion est entièrement à la charge du module (création, accès, emplacement ...), alors que les autres catégories de variables sont généralement implicitement gérées par les compilateurs.

Toutefois, certains langages fournissent des outils qui facilitent l'utilisation de ces variables (variables de type record, pointeurs, domaines, structures basées, ...).

Nous avons débordé les variables d'état d'un module ; nous nous intéressons maintenant aux variables locales.

Les variables locales (registres et variables de travail) ont pour rôle, à partir des paramètres fournis en entrée, et à l'aide des variables d'état, de calculer les nouvelles valeurs des variables d'état,celles des paramètres des procédures à appeler et celle des résultats à retourner (s'il y a lieu).

A chaque appel de procédure, l'ensemble variables locales, paramètres est sauvegardé. Il est permis de se demander si cet empilement massif est toujours justifié. En effet, les variables locales ayant servi à calculer au constituant les paramètres d'une procédure n'ont généralement pas à être empilées. Il en est de même pour les variables locales ayant servi à calculer une nouvelle valeur des variables d'état avant l'appel d'une procédure. Pour illustrer ceci, prenons l'exemple du programme P suivant :

```
     T:array (1...N) ;                & variable d'état &
Procédure P ;                         & cette procédure ne fait rien ! &
     begin
          A,B,I,J,K : integer ;        & variable locale &
          Read (I,J,K) ;
          A := T(I) + T(J) ;
1)        if A < N then T(A) := K ;
          B := Q(J,K) ;                & appel de la fonction Q d'un autre
          A := B+T(B)                    module &
          if A = N then return A + B ;
     end ;
```

Les variables A et I ont servi uniquement à élaborer de nouvelles valeurs de la variable globale T ; leur valeur est non significative au moment de l'appel de Q. Les variables J et K servent uniquement de paramètres à la fonction Q ; leur valeur au retour de la procédure Q est sans intérêt ; de même, B n'a pas de valeur significative à l'appel de Q.

Les cinq variables A, B, I, J, K sont inutilement empilées durant l'exécution de Q.

En utilisant au mieux les propriétés des langages à structure de blocs, nous pouvons écrire :

```
     T : array (1...N) ;
Procédure P ;
     begin B, J, K : integer ;
        begin
             A, I : integer ;
             Read (I,J,K) ;
             A := T(I) + T(J) ;
             if A < N then T(A) := K ;
        end ;
        B := Q(J,K) ;            & Appel de Q &
        begin
             A : integer ;
             A := B+T(B)
             if A = N then return A+B ;
        end ;
     end ;
```

De cette manière, les variables A et I n'ont pas été empilées, alors que les paramètres donnés à Q : B, J, K ont été inutilement stockés ; mais c'est là une contrainte apportée par les langages à structure de blocs.

Il faut noter que ce devrait être la tâche d'un compilateur évolué que de reconnaître dans l'exemple 1 que A et I n'ont pas à être empilés, et même s'apercevoir de l'inutilité de l'empilement de B, J et K. Mais c'est là beaucoup demander !

Nous constatons dans l'exemple précédent qu'il y a des situations où nous pouvons nous dispenser de l'empilement des variables locales et des paramètres. Le problème est de savoir si on peut toujours se ramener au cas précédent : existe-t-il des "variables locales" dont la val reste significative entre deux exécutions d'une procédure externe ? Nous avons déjà relevé l'existence de variables d'état non rémanentes. Ces variables permettent en effet de définir l'état d'un module ; Elles font partie

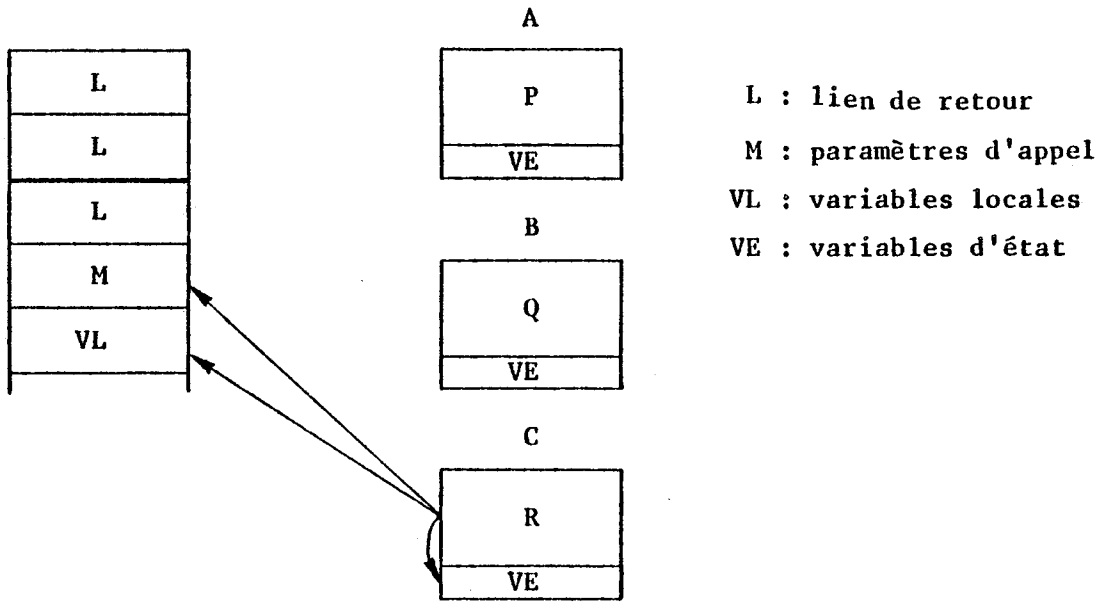
des variables d'état ; elles sont donc conservées à l'intérieur du module. L'expérience acquise lors de la programmation de OURS a montré que ces variables sont généralement en petit nombre (jamais plus de deux dans OURS).

L'utilisation de processus cycliques impose de bien faire la distinction entre les variables locales et les variables d'état non rémanentes. Quand leur rémanence est précisément le temps d'exécution de la procédure appelée, ces dernières dans un mécanisme de pile sont réservées dans la pile. Placer ces variables dans la pile permettait de résoudre le problème de rémanence, mais elle avait le défaut de donner la même rémanence aux variables non significatives que sont les variables locales.

Le palliatif généralement utilisé pour résoudre ce problème consiste à recopier lors de l'appel d'une procédure les variables d'état non rémanentes dans des zones prévues pour ces variables. Ce procédé permet de considérer leur ancien emplacement comme non significatif et donc de leur appliquer un traitement identique à celui des variables locales.

Nous considérons désormais que nous pouvons toujours nous ramener au cas où les variables locales de la pile sont non significatives et que nous pouvons nous en débarrasser.

Dans l'exemple a) du début du paragraphe, la pile devient :



Exemple B

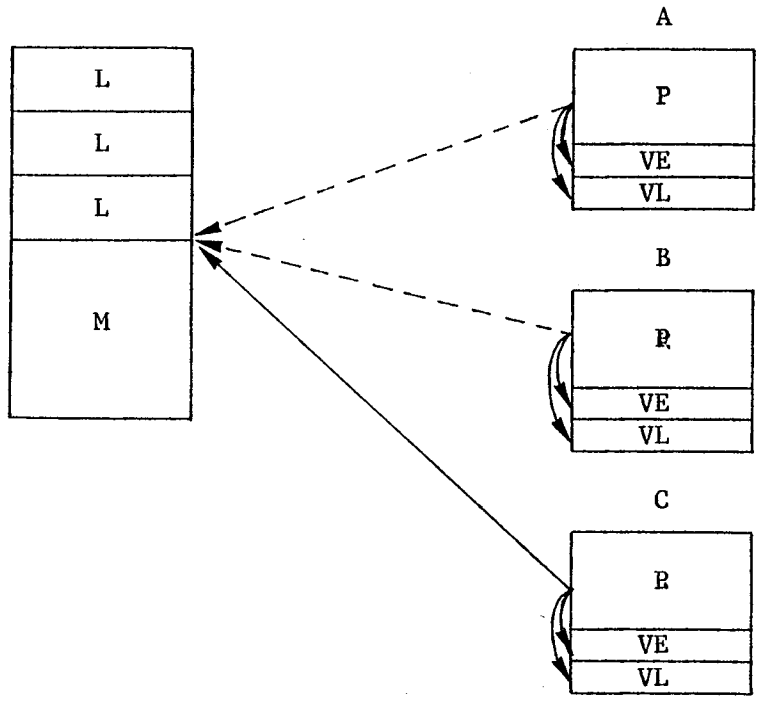
La pile est réduite alors à un ensemble de liens de retour et à un triplet (VL, M, L).

II.1.4. Communication par message

La raison pour laquelle les variables locales sont habituellement dans la pile est que l'empilement est réalisé par un simple déplacement du pointeur du haut de pile.

Nous avons vu que nous pouvions nous dispenser d'empiler les variables locales. Dès lors, il n'y a pas de raison de les réserver dans la pile. Chaque module connaissant la taille de ses variables locales, celles-ci peuvent lui être associées de façon similaire aux variables d'état.

Notre exemple devient :



Exemple C

Dans ces conditions, la pile ne comporte plus que deux types d'informations :

- les liens de retour ({L}) afin d'effectuer les retours procéduraux ;
- et un emplacement pour les paramètres de la fonction appelée.

En s'imposant une profondeur d'appel maximale (taille de la pile des liens de retour) et en connaissant la taille maximum des paramètres (max de M), nous pouvons déterminer une taille optimale pour la "pile" qui apparaît alors comme un message.

Celui-ci se comporte comme un "conteneur" ayant un en-tête fixe qui indique le chemin parcouru et qui transmet les paramètres entre les modules. En fin d'exécution, le module remplit le même message que le message reçu en entrée (ou un autre possédant le même en-tête) avec de nouveaux paramètres, puis le message transite vers un autre module.

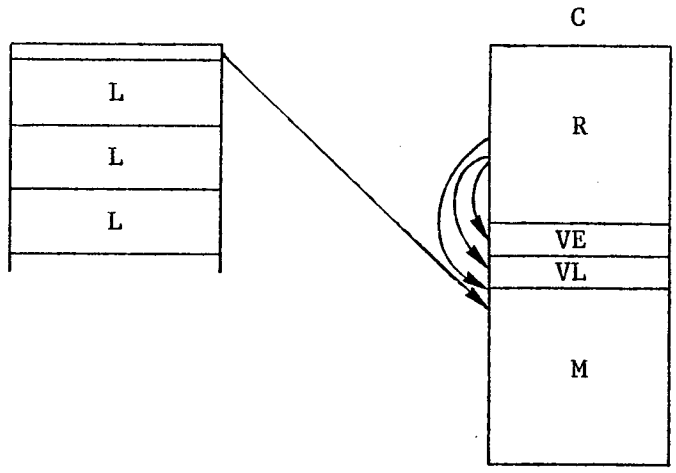
C'est la solution retenue pour hydra [WL 75]. La taille des liens de retour est difficile à estimer (cf. Ch. II.1.1) ; dans hydra elle constitue un paramètre de la création des messages. Dans OURS, nous avons pu nous limiter à un en-tête fixe de quatre mots.

Note : Une zone de variables locales allouées statiquement à un module empêche l'utilisation récursive des fonctions du module. Cette restriction apparamment contraignante est toutefois acceptable, sachant que beaucoup de langages n'admettent pas la récursivité.

La taille des paramètres à transmettre aux fonctions d'un module est très variable (de 0 à 256 mots dans OURS). Il peut donc être coûteux de réserver statiquement, pour toute la durée de vie d'un processus, une zone mémoire de taille maximum (256 mots pour OURS).

L'espace mémoire pour les paramètres peut être optimisé, dans la mesure où chaque module dispose d'un emplacement pour ses paramètres. Le message, en-tête compris, est recopié dans cet emplacement. Cette méthode évite de réserver inutilement un "containeur" de grande taille.

Pointeur vers le message courant



Exemple D

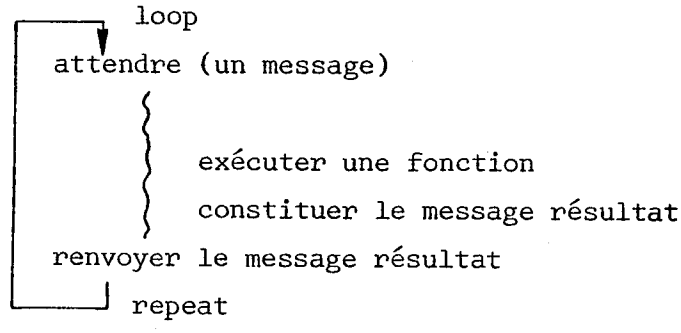
II.1.5. Processus cyclique

Dans l'exemple que nous venons d'introduire, on constate que l'exécution d'une fonction d'un module est similaire à l'exécution d'un processus asynchrone, c'est-à-dire :

- soumission d'une requête (réception d'un message) ;
- exécution de la fonction spécifiée dans la requête ;
- acquittement (émission d'un message résultat).

Au lieu de considérer un module A comme un ensemble de points d'entrée accessibles par un processus X, on peut estimer que le processus X demande à un processus serveur, s'exécutant dans le module A, de réaliser pour lui une fonction en lui fournissant dans un message l'identification et les paramètres de la fonction et en recevant en retour un message contenant les résultats.

La structure d'un module devient alors celle-ci :



Ainsi le processus qui s'exécute dans un tel module effectue une boucle infinie : attente d'un message (le message contient l'identification et les paramètres d'une fonction), exécution de la fonction demandée, constitution et émission d'un message résultat. Ce processus que nous appelons désormais "processus cyclique" n'effectue pas d'appel intermodule de type procédural. Il construit un message destiné au module à appeler, et le lui envoie après y avoir placé l'identification et les paramètres de la fonction à exécuter.

Du fait que les variables locales n'ont pas à être empilées sur un appel intermodule, les variables locales du processus sont non significatives et celui-ci est donc disponible pour une autre activité.

Puisqu'il n'effectue pas d'appel procédural intermodule, le processus cyclique n'a pas besoin d'une pile de contexte.

Si tous les modules sont exécutés par des processus cycliques, les piles des contextes associées aux processus n'existent plus. Par contre, chaque module comporte une partie "variables locales" et une partie "variables d'état".

NB : Nous étudions dans ce travail les relations entre les modules. Chaque processus cyclique peut appeler les fonctions internes au module de façon procédurale. Une pile interne au module peut donc exister pour ce processus cyclique ; mais ceci est en dehors de notre sujet d'étude. Il faut noter que si une telle pile interne existe, elle est gérée par le compilateur qui a compilé le module ; si une pile d'appel intermodule existe, elle est gérée par le système.

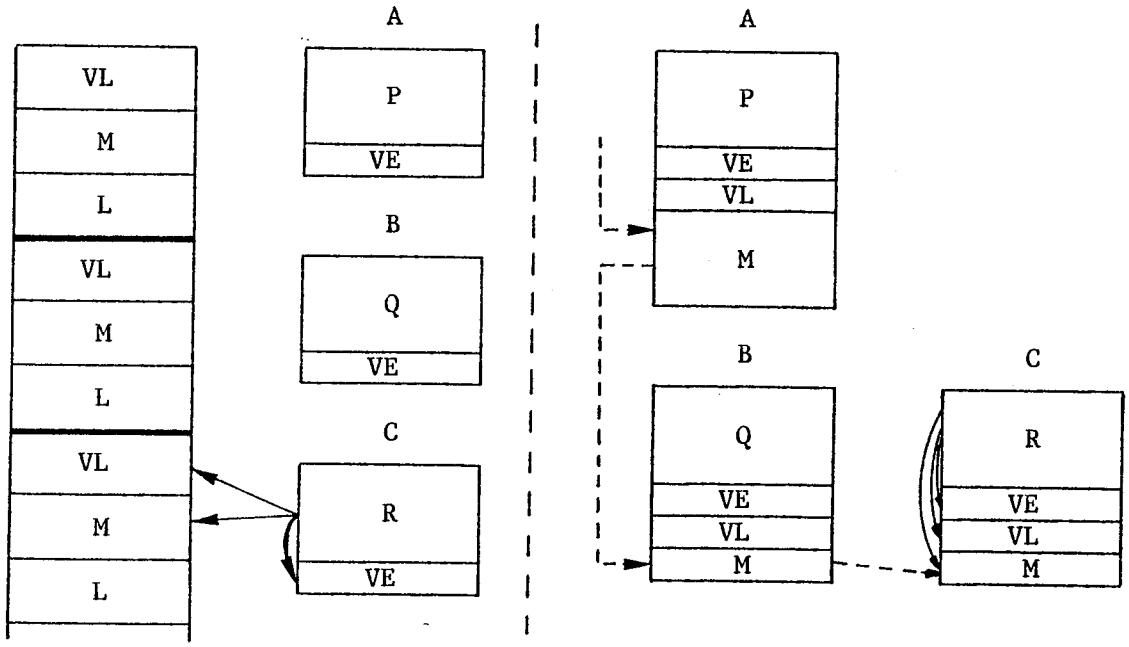
Nous sommes donc passés depuis l'exemple A jusqu'à l'exemple D d'une pile unique par processus (indépendamment du nombre de module) à un ensemble d'informations réparties dans tous les modules, indépendamment du nombre de processus. On ressent bien intuitivement que la différence d'espace mémoire consommé entre les deux exemples extrêmes (A et D) dépend du rapport nombre de processus / nombre de modules.

Le nombre de modules étant constant, cette éventuelle différence dépend donc uniquement du nombre de processus, donc de la charge.

II.1.6. Influence de la charge

Comme aux paragraphes précédents, nous appellerons VL les variables locales, M les paramètres d'une fonction point d'entrée du module et VE les variables d'état d'un module.

Nous avons vu quels sont les éléments de notre comparaison : d'une part le module soumis à l'appel procédural, et d'autre part le module, lieu d'exécution d'un processus échangeant des messages avec d'autres modules. Si nous reprenons le même exemple que précédemment, les deux alternatives se représentent comme suit :



Appel procédural

Transfert de messages à des processus cycliques

Supposons un système constitué de cinq modules tels que le degré d'imbrication des appels soit de 3. On se place dans l'hypothèse où il existe une seule tâche.

Dans ces conditions :

- si les modules sont exécutés par un processus cyclique, une zone (VL+M) étant réservée par module, le coût mémoire pour servir ce processus est 5*(VL+M) ;

- si les modules sont appelés de façon procédurale, une zone (VL+M) étant réservée par degrés d'imbrication, le coût mémoire devient 3*(VL+M).

Puisqu'il n'y a qu'un processus actif, l'allocation systématique des zones (VL+M) pour chacun des modules s'avère onéreuse puisque quatre parmi elles se trouvent être inemployées.

Supposons maintenant n tâches actives. Chacune de ces tâches dispose d'une pile de taille égale au mieux à 3*(VL+M).

Sachant que les modules sont non réentrants, on supposera que n ne sera pas très grand, surtout quand on exclut le multiplexage. Dans le cas des processus cycliques, cinq processus au plus sont en cours de traitement à un instant donné ; les autres restant en attente à l'entrée d'un module.

Avec n = 5, l'espace mémoire occupé est

5*(3*(VL+M)) dans le cas de l'appel procédural
5*(VL+M) pour les processus cycliques.

De ces chiffres (hâtifs il est vrai ; cf. paragraphe suivant); il ressort nettement que plus un système à processus cycliques est chargé en nombre de processus utilisateurs, plus la méthode des processus cycliques se révèle économique.

II.1.7. Conclusion

Le contexte de ce paragraphe est volontairement limité (module non réentrant et non multiplexé), nous ne cherchons pas ici à tirer des conclusions générales mais plutôt à relever les différences essentielles entre ces deux méthodes et à voir comment elles se combinent entre elles.

Reprenons les résultats tirés de l'exemple précédent (le système comprend 5 modules, avec un degré d'imbrication des appels de 3).

Chaque pile associée à l'appel procédural a pour taille au minimum 3*(VL+M) ; chaque module, lieu d'exécution d'un processus cyclique, réserve une zone de taille égale à (VL+M).

S'il existe cinq processus, l'espace mémoire occupé est respectivement :

5*3(VL+M) pour les piles associées à l'appel procédural

5*(VL+M) pour l'espace réservé aux processus cyclique.

Nous allons essayer de déterminer dans quelle mesure ces résultats sont valides.

Pour cela, nous allons d'abord appliquer nos conclusions à l'exemple particulier du système OURS.

On peut en effet se demander si les zones VL et M sont de taille égale dans les deux mécanismes.

. En ce qui concerne M, zone des paramètres, nous avons déjà précisé que chaque message comporte un en-tête standard. Dans hydra, elle est constituée d'une pile de lien de retour et d'une chaîne de bit (qui indique si le retour procédural doit être effectué ou non). Dans OURS, l'en-tête est une zone fixe de 4 mots qui servent à préciser l'association tâche-message, à effectuer des opérations comptables et à indiquer où doit être ultérieurement envoyé le message. En tenant compte des fréquences relatives des divers types de message relevé sur l'IRIS 80 (message utilisateur (appel superviseur), déroutement (pagination et erreur), IT (entrée-sortie)), la taille moyenne d'un message est de l'ordre de 7 à 8 mots. En considérant le résultat de ces mesures, dans le système OURS, l'en-tête augmente la taille du message de moitié.

Si M' est la taille du message, en-tête comprise, on a :

$$M' \simeq \frac{3}{2} M.$$

Note : L'en-tête pour les messages du réseau Cyclades est de 4 mots pour une taille de message moyenne de 6 mots (en application TS). Pour le système hydra, nous ne possédons pas les tailles moyennes mais les en-têtes sont généralement plus grandes que celles de OURS. Dans beaucoup de systèmes à message, l'ordre de grandeur du rapport $\frac{M'}{M}$ *est 2.*

Toutefois, il est possible d'améliorer partiellement l'utilisation de l'espace mémoire par la création dynamique des processus cycliques en fonction de la charge du module et, dans tous les cas, par "l'utilisation optimale" des processus déjà existants.

On entend par "utilisation optimale" des processus déjà existants la minimisation des cas de suspension de ces processus. Un processus est logiquement suspendu sur :

- appel procédural intermodule
- appel d'instruction abstraite (cf. § suivant)
- blocage par synchronisation.

Les processus cycliques n'effectue pas d'appel procédural intermodule ; cette cause importante de suspension est donc éliminée.

Les autres causes de suspension méritent d'être examinées car leurs remèdes ont des conséquences directes et sensibles sur le coût en espace mémoire et sur la structure des modules.

II.2.1. Appel d'instruction abstraite

Nous avons retenu des nombreuses recherches sur la méthodologie d'écriture de gros programmes (programmes système notamment) non seulement la décomposition modulaire, mais également la structuration en niveau d'abstraction [DJ 68].

Contrairement à une machine virtuelle, qui offre les mêmes outils qu'une machine existante, un niveau d'abstraction fournit des outils élaborés de manipulation de l'information et de gestion des processus.

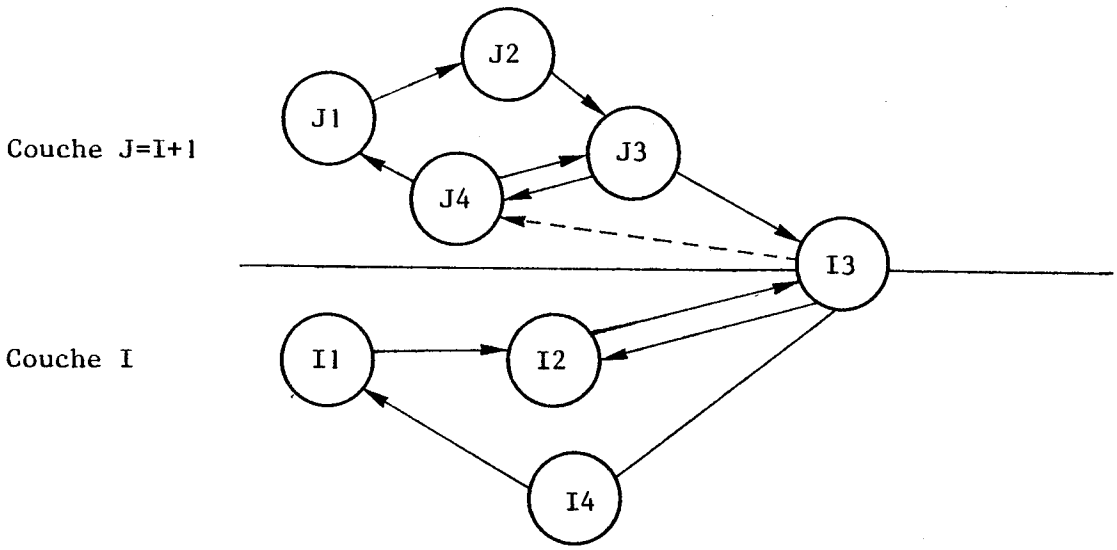
Un niveau d'abstraction a pour but de masquer tout ou partie des caractéristiques désagréables du matériel utilisé (déroutements, interruptions, instructions de base) pour fournir un interface mieux adapté à la résolution d'une classe donnée de problèmes.

Citons quelques réalisations expérimentales, bien que différentes dans leur but et leur ampleur : GEMAU [GR], HADRA [WL 74], SAR [BE 75], le système industriel Plessey [WI 72].

Un niveau d'abstraction définit une machine abstraite dont les instructions sont les fonctions réalisées par le niveau d'abstraction. Une instruction de cette machine abstraite est réalisée par l'exécution d'un programme qui s'exécute sur une machine abstraite du niveau "inférieur".

L'appel par un processus qui s'exécute dans le niveau i+1 d'une instruction abstraite provoque la suspension de ce processus et l'activation d'un processus du module du niveau i qui réalise l'instruction abstraite.

Une fois l'instruction abstraite effectuée, le processus demandeur du niveau i+1 est réactivé. L'exécution d'une instruction abstraite est donc rigoureusement synchrone ; elle provoque l'équivalent d'un appel procédural entre deux modules, chose que nous nous étions efforcés d'éliminer. Ainsi, par exemple, si le module i3 réalise des fonctions "longues" invoquées par le module J3, nous serons tentés de mettre le module I3 dans la couche I+1, permettant ainsi aux processus du module J3 d'invoquer le module I3 de façon asynchrone :



Cette solution se justifie sur le plan de l'efficacité pour des modules effectuant des fonctions longues ; en effet, il en découle un plus grand asynchronisme, donc une meilleure utilisation des processus cycliques. Inversement, ce procédé brise la simplicité des interfaces entre deux couches et, à la limite, revient à nier la notion de couche d'abstraction. En poussant ce procédé au maximum, on arriverait assez

rapidement à trouver tous les modules dans la même couche. Le découpage en niveaux d'abstraction est une opération délicate qui doit prendre en compte deux critères parfois incompatibles : une structure claire et une bonne efficacité. Par exemple, au cours de l'élaboration du projet OURS, le noyau a été structuré en 2, 3 ou 4 couches.

Il faut noter que ce problème est inexistant si tous les appels intermodules sont de type procédural : puisque l'appel d'une instruction abstraite et l'appel d'un module de la même couche sont de même nature.

Nous abordons maintenant la troisième source de suspension d'un module, à savoir la synchronisation interne.

II.2.2. Synchronisation interne

La réentrance implique que les variables d'état d'un module (en un seul exemplaire) puissent être accédées "simultanément" par plusieurs processus. Dans un tel module, une synchronisation d'accès aux variables critiques s'impose. Les divers problèmes de synchronisation ont fait l'objet de très nombreuses études, aussi nous ne rappellerons pas les méthodes proposées, mais nous allons analyser leurs conséquences sur la gestion des processus et sur l'utilisation de leur zone de réentrance.

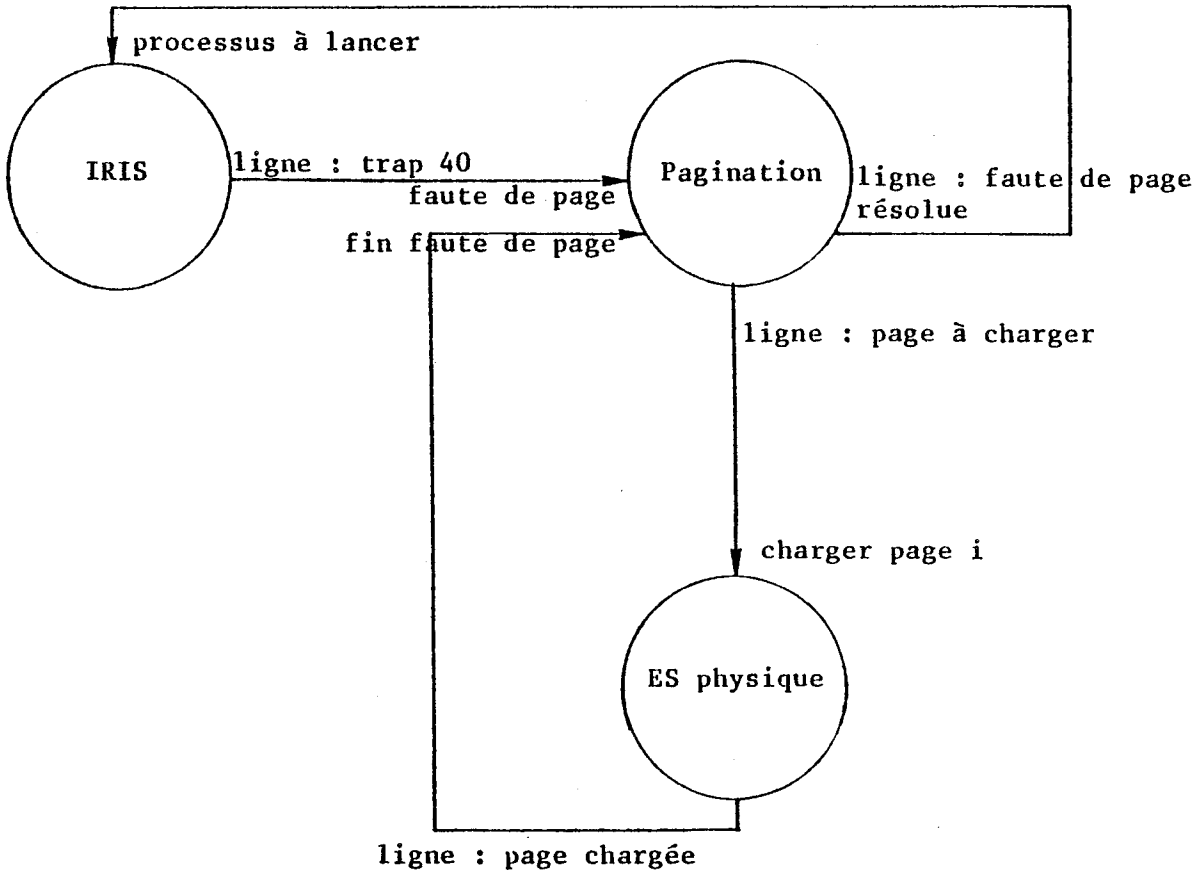
Hormis la méthode de l'attente active, heureusement peu employée sauf pour des séquences courtes exécutées par des processus ininterruptibles, toutes les méthodes visent à suspendre un processus s'il y a blocage sur l'accès à une fonction ou à une donnée critique, ou pour éviter l'exécution simultanée de deux fonctions incompatibles.

Dans le cas où le module est le lieu d'exécution de processus cycliques, la suspension éventuelle des processus impose, si on veut obtenir une réentrance effective de n, plus de n processus cycliques. Les zones de réentrance des processus suspendus sont alors inutilement réservées.

Prenons pour exemple, dans le module de pagination, la fonction de résolution des fautes de page. Quand certaines pages sont référencées par plusieurs processus, la même faute de page peut se produire

plusieurs fois avant que cette page soit effectivement chargée. Si la faute de page est en cours de résolution (page en transit), on doit attendre que la page soit chargée avant d'effectuer le retour procédural (renvoyer).

```
                    BEGIN
0                       ATTENDRE (message) ;
                        case CODE-FONCTION OF
                        <FAUTE-DE-PAGE> : Faute-de-page ;
                        <FIN-FAUTE-DE-PAGE> : Fin-faute-de-page ;
                    cycle ;
          Proc FAUTE-DE-PAGE ;
               BEGIN
1                   Si page i en transit alors    & une faute a déjà
2                       chaîner message file i ;  été faite sur cette
3                       RETURN ;                  page &
                        End ;
                    Sinon
4                       message.code-fonction := charger-page i ;
5                       renvoyer (message, ligne : page-à-charger) ;
6                       RETURN ;
               END ; END ;
          Proc FIN-FAUTE-PAGE ;
7              Tant-que file i non vide faire
8                       déchaîner message ;
9                       RENVOYER (message, ligne : faute de page résolu
               END ;
```



A la ligne 2, en utilisant un mécanisme de synchronisation classique, nous aurions écrit : SUSPENDRE (processus) ; de même, aux lignes 4 et 5, nous aurions écrit : page à charger (page i) pour effectuer un appel procédural au module d'entrée-sortie physique.

Les mesures effectuées [DU 74] ont montré que de nombreux processus étaient suspendus en attente de résolution de transfert de page. En conséquence, il faudrait envisager un nombre relativement important de processus cycliques dans le module de pagination de façon à limiter les suspensions. Il est facile de constater que cette solution est coûteuse. Nous avons vu comment il était possible d'effectuer un appel intermodule par transfert de message à des processus cycliques.

A la ligne 4 nous modifions le message pour indiquer la fonction demandée ainsi que ses paramètres ; à la ligne suivante, nous envoyons au module d'E/S physique le message adapté à la nouvelle requête. La destination du message est indiquée par la mention de l'identification d'une ligne de sortie du module (page-à-charger). Puis, à la ligne 6, le processus

cyclique devient disponible et revient attendre un nouveau message (ligne 0).

Parallèlement, un processus du module E/S physique réalise le transfert de la page et affecte au code fonction du message la valeur "fin faute de page" en sorte qu'il puisse transiter vers le module de pagination.

De façon à éviter la suspension d'un processus cyclique lorsque la page est en transit, on utilise un mécanisme similaire, à savoir le multiplexage : au lieu de suspendre le processus (c'est-à-dire chaîner ce processus sur une file), c'est le message qui est chaîné sur la même file i (ligne 2). Cette tâche étant arrêtée, le processus cyclique devient disponible et peut entreprendre une nouvelle exécution (lignes 3 et 0).

Quand l'évènement attendu arrive, ce qui correspond dans notre example à l'arrivée du message annonçant le chargement de la page i, les messages en attente sont traités aux lignes 7, 8, 9. Dans notre exemple, leur demande est déjà satisfaite puisque la faute de page est résolue. Il suffit alors de renvoyer le message vers l'IRIS (ligne : faute-de-page-résolue) pour continuer l'exécution qui avait provoqué la faute de page.

Dans le module de pagination, les processus ne sont jamais interrompus grâce aux mécanismes du multiplexage et de l'appel asynchrone. D'autre part, les processus de ce module étant ininterruptibles et s'exécutant sur un bi-processeur (IRIS 80), deux processus cycliques sont suffisants.

Le gain d'espace ainsi réalisé est important au point que, pour beaucoup de modules, l'efficacité impose d'utiliser le multiplexage.

Nous avons simplement voulu montrer l'importance du multiplexage ; nous traiterons plus loin le problème dans sa généralité (cf. Chap. III.2).

Noter que la suspension d'un processus impose le stockage dans la pile de l'ensemble des variables locales et des paramètres, alors que multiplexer revient à stocker le message seulement.

II.2.3. Influence de la charge

J. Mossière [MO 77] signale que, a priori, la création de processus cycliques dans un module "semble multiplier inutilement le nombre de processus". Cette opinion est sans doute discutable. En effet, le rapport entre le nombre de processus "habituels" et le nombre de processus cycliques, et par conséquent l'espace mémoire consommé, varie fortement en fonction de l'application envisagée.

Nous avons vu (Chap. II.2.1) que les relations existant entre les modules de niveau d'abstraction différents sont synchrones (interprétation d'une instruction abstraite). Notre comparaison entre l'appel synchrone (procédural) et l'appel asynchrone entre modules (processus cycliques) se limite donc aux relations entre les modules d'une même couche d'abstraction.

Nous allons étudier l'influence de la charge sur le système OURS puis nous verrons dans quelle mesure cet exemple est significatif.

En premier lieu, il nous faut préciser la structure du système OURS :

couche Hardext	non modulaire
couche UGR	5 modules
couche machine fichier	5 modules
couche P3S	5 modules
couche TS	5 modules

Le degré d'imbrication moyen des appels est de 3 ; le nombre moyen de processus cyclique est de 10 par couche.

Notre comparaison est réalisée, dans un premier temps, dans le cadre d'un système à forte puis à faible charge conversationnelle ; enfin, elle est effectuée sur un système non interactif.

II.2.3.1. Forte charge conversationnelle

Nous supposerons ici une charge de 80 utilisateurs conversationnels. Nous nous plaçons dans le cas - non réaliste - où tous les processus sont uniformément répartis dans les différentes couches ; soit par couche :

20 processus conversationnels
10 processus cycliques
5 modules.

Nous utiliserons désormais la notation du paragraphe II.1.7, notamment :

M : taille des paramètres envoyés à une procédure
M' : taille d'un message envoyé à un module
VL : taille des variables locales à empiler (appel procédural)
VL' : taille des variables locales associées à un processus cyclique.

En outre, nous supposerons que :

$M' \simeq \frac{3}{2} M$ et $VL' \simeq a\ VL$ avec $a \in \{1.3\}$

Nous utiliserons $a = \frac{3}{2}$. Nous insistons encore sur le fait que ces estimations sont en grande partie intuitives et mériteraient à elles seules une longue analyse.

En supposant que les piles de contexte associées à un processus sont allouées statiquement et sont de taille optimale, l'espace mémoire consommé si les modules sont soumis à l'appel procédural est de :
(nombre de processus dans la couche) × (niveau imbrication) × (VL + M)
soit 20 × 3 × (VL+M)
ou enfin 60 (VL+M).

Si les modules sont le lieu d'exécution de processus cycliques, il y a, à un instant donné, au plus 10 processeurs conversationnels en cours d'interprétation (par les 10 processus cycliques), les 10 autres processus utilisateurs étant suspendus par multiplexage ou à l'entrée du module.
L'espace mémoire consommé est alors :
10(VL+M) + 10 M' (messages multiplexés) = 15(VL+M) + 15M,
soit un gain de 45 VL et de 30 M en faveur des processus cycliques.

Nous pouvons remarquer que même en utilisant des hypothèses très pessimistes pour les rapports M'/M et VL'/VL, nos conclusions restent valides.

II.2.3.2. Charge conversationnelle faible

Envisageons maintenant cette analyse dans le cas où ce même système est caractérisé par une charge conversationnelle faible.

Notons que, quelle que soit la charge, les processus cycliques consomment toujours 15(VL+M), alors que la consommation dans le cas de l'appel procédural est directement proportionnelle à la charge.

En conséquence, pour une charge de 5 processus externes par couche (20 au total), les résultats deviennent :

5×3 (VL+M) pour les piles des appels procéduraux
15 (VL+M) pour les processus cycliques,

soit un résultat comparable.

Notons que, dans ce cas, selon les hypothèses sur les rapports VL'/VL et M'/M, les conclusions peuvent être opposées.

II.2.3.3. Charge non conversationnelle

Les mesures effectuées sur les systèmes à traitement par lot indiquent qu'en général, à un instant donné, 4 à 8 processus se trouvent être actifs dans le système, soit, pour notre exemple, 1 à 2 processus par couche.

Les mêmes résultats deviennent :

$1.5 * 3$ (VL+M) $\simeq$ 5 (VL+M) pour les piles des appels procéduraux
15 (VL+M) pour les processus cycliques,

soit un gain sensible pour l'appel procédural.

II.2.3.4. Répartition de la charge

Des mesures effectuées sur le comportement du système CP révèlent que parmi 40 à 50 processus conversationnels, seuls 2 à 5 processus sont réellement actifs.

On rappelle que les processus suspendus le sont en grande majorité en attente de transfert de page ou en attente de transfert de message en provenance de la console utilisateur ; ce qui correspond dans OURS à une demande de service auprès du module de pagination ou du module des entrées/sorties logiques de UGR. Dans les deux cas, ces

processus sont suspendus dans la couche UGR.

Pour cette couche particulière et pour la forte charge conversationnelle évoquée précédemment - 80 utilisateurs conversationnels - environ 60 à 70 d'entre eux sont suspendus dans la couche UGR. Nous obtenons approximativement :

70 processus conversationnels (5 processus actifs et 65 suspendus)

10 processus cycliques.

L'espace occupé par les piles d'appel procédural est

$$70 \times 3 \text{ (VL+M)} \simeq 200 \text{ (VL+M)}$$

et l'espace consommé par les processus cycliques vaut :

$$15(VL+M) + 65\ M' = 15(VL+M) + 100\ M.$$

Le gain d'espace effectué par les processus cycliques est très appréciable ; inversement, les autres couches se retrouvent dans le cas de la charge non conversationnelle pour laquelle l'appel procédural est avantageux.

II.3. Conclusion

Du point de vue de l'espace mémoire consommé, l'intérêt essentiel des processus cycliques est d'éviter lors d'un appel de procédure la conservation des variables locales inutiles et, par voie de conséquence, la gestion d'une pile. Par contre, le mécanisme des processus cycliques peut, parfois, multiplier le nombre de processus. Le choix entre l'un ou l'autre des deux mécanismes dépend avant tout du type d'application.

Les processus cycliques se révèlent avantageux pour les composants soumis à de fortes charges et multiplexés (interpréteurs, modules du noyau, E/S, unité centrale, compilateur, ...). Inversement, l'appel procédural se montre mieux adapté aux autres cas : modules non réentrants ou peu chargés, ou rarement utilisés.

A titre d'exemple, nous ferons remarquer que ce résultat a été intuitivement bien perçu par la plupart des concepteurs : l'unité centrale,qui est le composant le plus chargé et le plus multiplexé, a le comportement typique d'un processus cylique ; en entrée, elle

reçoit un message (Registres, PSD, ...) et elle émet en sortie un autre message (Registres, PSD, ...) ; elle peut aussi être multipliée (multiprocesseurs).

Chapitre III

Comparaison du coût d'exécution

Le fait qu'un module soit soumis à des appels procéduraux ou qu'il soit le lieu d'exécution d'un processus cyclique échangeant des messages avec d'autres modules, introduit-il une différence dans le temps d'exécution des services fournis par ces modules ?

Si une telle différence existe, elle peut a priori résulter de trois facteurs :

- différence de fréquence ou de coût des appels aux primitives de gestion de la mémoire, selon que les informations échangées entre module figurent dans des messages ou dans des piles de contexte ;
- différence de fréquence ou de complexité des opérations de création ou de modification des contextes d'exécution ;
- différence dans la rapidité de l'exécution des algorithmes programmés dans le module.

Nous allons successivement analyser ces trois points.

III.1. Coût de la gestion mémoire

Les opérations de création et de libération de mémoire se limitent uniquement aux appels et aux retours intermodules. Une éventuelle différence de coût de gestion de la mémoire ne peut provenir que d'une différence de complexité des opérations à effectuer pour gérer la pile des appels procéduraux ou pour gérer les messages à envoyer aux modules.

Les solutions existantes sont extrêmement variables d'un système à l'autre. Elles sont très dépendantes des caractéristiques du matériel employé. Nous nous sommes limités au passage de paramètre par valeur. En conséquence, la façon de faire transiter des paramètres d'un module vers l'espace d'exécution d'un autre module dépend étroitement des relations entre les divers supports de mémoire disponibles.

En effet, deux modules peuvent partager le même support de mémoire ; on trouve aussi le cas où ils disposent d'une mémoire privée et d'une mémoire commune, et le cas où ils ne possèdent aucune mémoire commune.

Nous allons dans ce qui suit analyser pour chacune de ces alternatives les différences de coût entre la gestion d'une pile d'appel procédural et la communication par message.

III.1.1. Modules à support de mémoire commun

Il est fondamental de distinguer le support de mémoire, i.e. la mémoire physique, de l'espace d'adressage ou d'exécution d'un processus.

Tout en ayant le même support de mémoire physique, des processus peuvent avoir des espaces d'exécution totalement ou partiellement disjoints. C'est généralement le cas dans les calculateurs classiques multiprogrammés où l'on utilise le mécanisme des espaces virtuels, des registres de base et des clefs de protection.

III.1.1.1. Gestion de la pile

Dans la mesure où les divers modules ont été écrits dans le même langage et compilés simultanément, l'appel d'un module se fait de façon similaire à un appel de procédure classique : les paramètres sont recopiés dans la pile et le pointeur de sommet de pile est mis à jour. Ce mécanisme est bien connu et ne nécessite donc pas d'être présenté plus en détail. Nous ferons seulement remarquer qu'il y a une recopie des paramètres.

Le compilateur réalise la protection des objets des différents modules. Il n'y a donc aucune contrainte sur les espaces d'exécution de ces modules, exception faite de la pile qui doit être commune.

Toutefois, les langages qui permettent de compiler plusieurs modules sont rares. La plupart des modules actuellement sont compilés indépendamment.

Dans ce cas, la protection des objets n'est plus effectuée par le compilateur. Elle est alors assurée par la création d'un espace d'exécution propre à chaque module. Dans ces conditions, le passage des paramètres s'effectue traditionnellement par registre, malgré les limitations et les risques d'erreur que cela comporte. De plus, la transmission des paramètres s'effectue en deux recopies :

recopie dans les registres, recopie des registres dans des variables locales.

III.1.1.2. Gestion des messages

Il existe de nombreuses façons de transférer un message entre deux modules.

Par exemple, si les processus associés aux modules disposent d'espaces virtuels adjoints, une première méthode consiste à modifier les espaces virtuels ou plus précisément les tables de pages en faisant apparaître une page de l'espace virtuel émetteur dans l'espace virtuel du module récepteur. Cette méthode permet de transférer uniquement un nombre entier de pages. Malgré son efficacité (2 mots à modifier), elle est peu utilisée en raison de la trop grande taille des pages (généralement 2 ou 4K octets).

Une autre façon de faire est d'utiliser un emplacement commun aux espaces virtuels pour y transférer les messages. Ce transfert, selon les choix d'implémentation, s'effectue en une ou deux recopies.

Si les modules utilisent un adressage réel, le transfert pourra toujours être effectué en une seule recopie.

III.1.1.3. Conclusion

Quand les modules ont un même support de mémoire, le coût de la gestion de la pile des appels procéduraux est très voisine de celui de la gestion des messages. Toutefois, quelques différences peuvent être relevées quant au volume d'information à transférer : un message est plus volumineux que les paramètres qu'il renferme (en raison de l'en-tête). Par contre, l'espace à réserver lors d'un appel procédural est constitué des variables temporaires des procédures du module appelé et des paramètres à transférer ; pour un message, seul l'espace nécessaire au message est à réserver.

En conséquence, la différence de coût de la gestion mémoire entre les deux mécanismes est faible. Toutefois, un compilateur optimisé qui compilerait tous les modules d'un système pourrait mettre en oeuvre une gestion de pile optimisée qui gagnerait autant sur le

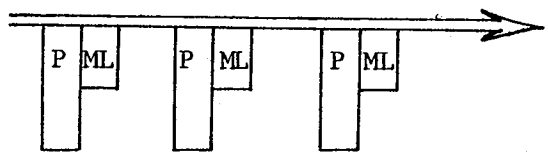
coût de l'allocation de mémoire (par la création de piles allouées statiquement) que sur le volume d'information à transférer (par la détermination des variables locales à ne pas empiler).

Pour remplir pleinement sa fonction, le compilateur associé devra en plus tenir compte du parallélisme en gérant dynamiquement autant de pile qu'il y a de processus.

Nous venons de traiter les modules qui disposent d'un support mémoire commun. Nous abordons maintenant l'analyse du coût de gestion de la mémoire pour l'appel procédural et pour la communication par message entre des modules à support de mémoire disjoint.

III.1.2. Modules à support mémoire disjoint

Chaque module dispose d'une mémoire locale d'accès rapide et peut éventuellement adresser les mémoires locales d'autres processeurs.



ML : mémoire locale
P : processeur
⟹ : bus d'accès à la mémoire

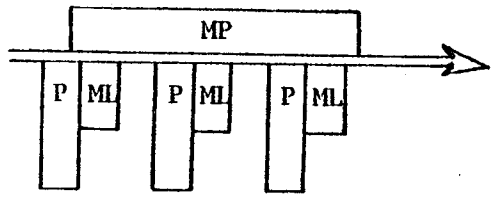
Même dans le cas où les processeurs peuvent adresser les mémoires locales d'autres processeurs, le coût des accès aux autres mémoires locales est si élevé que, en pratique, les messages doivent impérativement être recopiés dans la mémoire locale du processeur qui effectue la commande.

A titre d'exemple, dans le projet "Computer module" [FA 73] où tous les processeurs peuvent accéder à une hiérarchie de mémoire à trois niveaux, les temps d'accès sont respectivement

1,5 µs (mémoire locale du processeur)
7 µs ("cluster" local)
20 µs ("cluster" distant).

Le transfert des messages est réalisable en une seule recopie mais le coût sera plus important si les deux modules sont dans des "clusters" lointains. Multimicro [MA 76], SPS [CR 75] ou les réseaux présentent un cas similaire.

Dans certaines architectures, une mémoire est commune à tous les processeurs.



MP : mémoire partagée
P : processeur
ML : mémoire locale
⇒ : bus d'accès à la mémoire

Généralement le transfert s'effectue en deux recopies, d'abord dans la mémoire commune, ensuite dans la mémoire locale du module récepteur. Cette solution est appliquée, notamment dans Pluribus [OB 74], Micral M [REE 76].

Effectuer des appels procéduraux entre modules et donc en l'occurrence entre processeur n'est pas très économique. Il faudrait en effet transférer par message les paramètres ainsi que le bloc de contrôle du processus appelant et réaliser des liaisons entre les diverses parties de la pile dispersée dans les mémoires locales des processeurs.

Il est possible d'effectuer des appels procéduraux entre modules implantés sur des processeurs distincts, mais on se heurte à de nombreux problèmes techniques qui rendent cette solution impraticable ; par contre, le mécanisme de transfert de message entre processus cyclique est parfaitement adapté à ce genre d'architecture. En fait, cette méthode est universellement adoptée pour résoudre les problèmes de communication entre processeur.

Il suffit pour s'en convaincre de constater que la réalisation d'un réseau se limite quasiment à une gestion de transfert de message.

III.1.3. Conclusion

Les deux mécanismes (pile et message) ne peuvent être comparés que lorsque les modules ont un support mémoire commun (machines classiques) ; dans les autres cas, seul le mécanisme de transfert de message entre processus cyclique peut réellement convenir.

Dans le domaine où les deux mécanismes sont comparables (i.e. modules à support mémoire commun), on s'aperçoit que le coût de gestion de la mémoire est quasiment indépendant du mode de communication utilisé.

On peut ajouter que, quel que soit le mécanisme de gestion de la mémoire physique, notre conclusion reste vraie. En conséquence, nous ferons dans la suite abstraction des mécanismes sous-jacents de gestion de la mémoire et nous supposerons posséder une mémoire infinie.

Pour notre problème, la gestion de la mémoire n'introduit donc pas de différence de coût. Nous allons maintenant essayer de déterminer s'il en est de même pour les deux autres facteurs que nous avons dégagés : le coût de l'appel intermodule et le coût de l'exécution des modules.

III.2. Coût de l'appel intermodule

L'appel intermodule comporte plusieurs actions correspondant à :

- la constitution d'un espace d'exécution pour la procédure appelée,
- la recopie des paramètres dans cet espace d'exécution (passage des paramètres par valeur),
- les branchements dans le nouvel espace d'exécution.

Le coût de la gestion mémoire associée au passage des paramètres vient d'être traité (cf. III.1). Nous nous intéressons ici uniquement à la différence de coût concernant la constitution/destruction des espaces d'exécution et le branchement dans le nouvel espace selon qu'il s'agit d'un appel procédural ou du lancement d'un processus cyclique.

III.2.1. Appel procédural

Le coût d'un appel procédural dépend étroitement du fait que les modules concernés disposent d'espaces d'exécution communs ou disjoints.

III.2.1.1. Modules à espaces d'exécution communs

Les modules ayant un même espace d'exécution ont soit été compilés simultanément, soit ont été compilés indépendamment, mais alors on leur a volontairement affecté un même espace d'exécution (c'est généralement le cas des modules des noyaux de système).

Pour des modules compilés simultanément, l'appel intermodule se traduit par un appel de procédure classique. L'espace d'exécution, exception faite de la pile, reste inchangé et l'appel ou le retour intermodule sont effectués par deux instructions de branchement seulement.

La liaison entre des modules compilés séparément est réalisée par l'édition de liens. L'appel intermodule s'effectue également en deux instructions. Par contre, l'édition de liens du système est plus coûteuse.

D'après ce qui précède, l'appel procédural, pour les modules à espace d'exécution commun, est peu coûteux. Il est donc naturel que tous les langages de programmation actuels utilisent ce mécanisme. Nous allons voir que cette conclusion s'avère différente dans le cas des modules à espace d'exécution disjoints.

III.2.1.2. Modules à espaces d'exécution disjoints

Un des aspects fondamentaux de la notion de module est la protection des variables internes.

Si on ne dispose pas d'un compilateur capable de garantir cette protection, il est nécessaire d'utiliser un mécanisme pour la réaliser. Le procédé le plus communément utilisé consiste à doter chaque module d'un espace d'exécution différent, ce qui rend impossible l'accès direct aux variables d'un autre module.

Les machines classiques offrent plusieurs dispositifs qui permettent la création d'espace d'exécution indépendants : espaces virtuels, clefs de protection, bases d'adressage. A un niveau plus élaboré, les sous-systèmes offrent d'autres mécanismes : fichiers, liens, variables, annuaires, droits d'accès, ...

Chaque appel intermodule de type procédural impose la création d'un espace d'exécution propre au module référencé.

Bien souvent, cet espace d'exécution est constitué d'un espace virtuel, de droits d'accès, de prérogatives, de mécanisme de désignation des objets, ...

L'appel procédural intermodule consiste alors en une conservation de l'espace d'exécution associé au module appelant, puis en une constitution de l'espace d'exécution du module appelé (recherche de descripteur de segment, de table de page, de table de segment, de droits d'accès ou de catalogue, ...).

Ce mécanisme se rencontre entre autre dans des systèmes comme GEMAU [BR 74], HYDRA [W 74], MULTICS [B2 72] qui désirent fournir des espaces d'exécution plus complets et plus riches que ceux offerts par le hardware.

Ces opérations de changement d'espace d'exécution sont complexes et coûteuses. Pourtant elles sont couramment utilisés dans les systèmes actuels.

III.2.2. Processus cyclique

L'appel d'un module, lien d'exécution de processus cycliques, se traduit par un transfert de message et par l'activation de l'un des processus cycliques.

Les processus cycliques n'effectuent pas d'appel intermodule et ils ne peuvent adresser que les variables internes de leur module. Leur espace d'exécution est donc constant, exception faite du message à traiter.
A la création d'un processus cyclique, un espace d'exécution est constitué et lui reste définitivement associé.

L'appel d'un module se traduit alors simplement par l'activation d'un processus préexistant.

Activer un processus est une action mise en oeuvre à chaque tranche de temps, par l'allocateur d'unité centrale.

Bien souvent, l'allocateur d'unité centrale profite du fait que le noyau a le contrôle pour effectuer une opération de répartition (allocation

d'une tranche de temps à un nouveau processus).

L'activation d'un processus est équivalente

. à la terminaison de la tranche de temps du processus appelant (elle constitue donc une partie de l'overhead habituel du système)

. et au choix d'un processus à activer (chaînage de ce processus dans la file des processus en attente d'unité centrale). Le choix du processus, ou du message à associer à ce processus, peut introduire des pertes de temps si, par exemple, les messages sont triés par ordre de priorité. Dans OURS, le nombre de priorité des messages est fixe et à chaque priorité correspond une file d'attente. Cette opération est donc peut coûteuse.

Le lancement d'un processus cyclique introduit peu de coût supplémentaire (un déchaînage et un chaînage) du moins pour un intertemps entre deux appels intermodules grands par rapport à la durée d'une tranche de temps élémentaire.

III.2.3. Conclusion

Nous venons de voir que l'appel procédural intermodule est extrêmement efficace dans le cas où les modules disposent du même espace d'exécution, mais devient coûteux dans les autres cas.

L'activation d'un processus cyclique, par contre, est toujours assez peu coûteuse.

De cette étude, nous pouvons tirer les conclusions suivantes :

- les modules qui disposent d'espace d'exécution disjoint n'ont pas intérêt à être trop petits quant au volume d'exécution à exécuter, sinon le coût de l'appel pourrait dépasser le coût de la fonction à réaliser ;
- pour ces mêmes modules, le coût du lancement d'un processus cyclique est sensiblement inférieur au coût d'un appel procédural ;
- pour les modules ayant le même espace d'exécution, l'appel procédural est plus efficace.

Dans le projet OURS, les appels engendrés par les modules du noyau à un module contenant de brèves fonctions de service sont effectués de façon procédurale avec passage de paramètre par registre. Les

relations entre les autres modules sont réalisées par transfert de messages interprétés par des processus cycliques.

Nous venons de voir que le coût de l'appel intermodule dépend en partie du mécanisme d'appel intermodule utilisé. Nous allons maintenant essayer de déterminer s'il en est de même pour le coût de l'exécution d'un module.

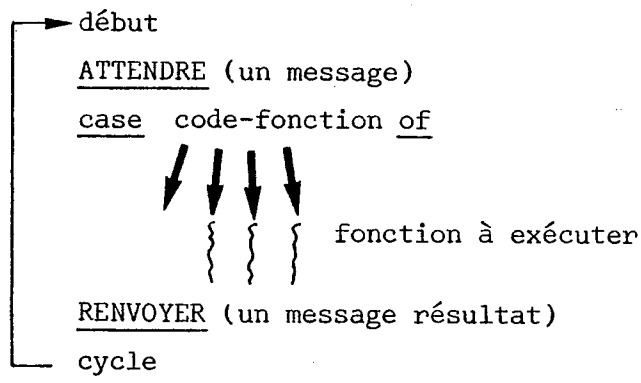
III.3. Coût de l'exécution des modules

L'aspect fonctionnel d'un module est indépendant du mécanisme d'appel intermodule utilisé. Par contre, sa structure interne y est sensible. Cette différence de structure interne peut influer sur la quantité de code à exécuter pour réaliser une fonction et donc intervenir sur le coût d'exécution de la fonction.

C'est cette différence de volume de code que nous cherchons à apprécier.

III.3.1. Structure générale

Les processus cycliques imposent à leur module-support une structure d'ensemble de la forme :



L'instruction RENVOYER provoque l'envoi d'un message résultat qui est analogue à la recopie des résultats d'une fonction appelée de façon procédurale. L'instruction ATTENDRE indique au système que le processus doit être suspendu jusqu'à l'arrivée d'un nouveau message.

Parrapport à l'appel procédural, au sein du module appelé, une instruction supplémentaire est exécutée à savoir :
"case code-fonction of" qui permet de choisir la fonction à exécuter en se servant du code-fonction trouvé dans le message.

Mais cela ne constitue pas un inconvénient majeur des processus cycliques. Des différences beaucoup plus significatives apparaissent au niveau de la synchronisation interne et des appels intermodules (cf. Annexe technique I).

III.3.2. Synchronisation interne

Une action de synchronisation doit être effectuée lorsque, à un moment donné, des actions entreprises par deux processus sont incompatibles : conflits d'accès à des données ou à des procédures, fonctions mutuellement exclusives, ... Cette action de synchronisation aboutit à la suspension de l'une des deux exécutions. Ceci est vrai qu'il s'agisse d'un processus "classique" ou d'un processus cyclique. La méthode de synchronisation ne dépend pas du type des processus manipulés ; nous ne parlerons donc pas des diverses méthodes de synchronisation existantes.

Nous avons vu au Chapitre II.2.1 que, pour des raisons d'efficacité, les processus cycliques ne doivent pas être suspendus sur une synchronisation. Pour éviter cela, les processus cycliques effectuent du multiplexage.

Suspension et multiplexage se traduisent par des actions très voisines. Dans le premier cas, on chaîne un processus sur la file d'attente associée à un évènement, dans l'autre on chaîne un message sur cette même file.

C'est l'exécution en cours qui est ainsi suspendue, et non pas le processus "responsable" de cette exécution.

Ce processus est alors disponible pour une autre activité et va pour cela exécuter l'instruction ATTENDRE.

Les fonctions traditionnelles de synchronisation peuvent être modifiées pour effectuer du multiplexage et éviter ainsi la suspension. Prenons par exemple une synchronisation classique par sémaphore : P et V.
A chaque point de synchronisation particulier est associé un sémaphore.

Le code habituel du P est :

si condition non satisfaite alors suspendre

devient : si condition non satisfaite alors chaîner le message
allera début.

Le code habituel de V est :

réactiver 1 processus

devient : déchaîner un message, modifier le code fonction, chaîner le message en entrée du module.

Le multiplexage est alors réalisé de façon efficace et quasiment transparente au programmeur et surtout le coût supplémentaire du multiplexage par rapport à une synchronisation est quasiment nul.

Si le multiplexage doit apparaître explicitement dans les algorithmes, il s'ensuit un accroissement du volume du code. Nous aurions un phénomène semblable si les actions de synchronisation et de suspension devaient apparaître explicitement dans le programme. Cet accroissement de volume apparent est éventuellement préjudiciable à la lisiblité des algorithmes mais n'a pas d'impact sur la vitesse d'exécution d'une fonction.

Ainsi la structure générale de contrôle et les mécanismes de synchronisation internes sont, en ce qui concerne le coût de l'exécution des fonctions d'un module, indépendantes du mécanisme d'appel inter-module utilisé. En est-il de même pour les mécanismes de contrôle de la réentrance.

III.3.3. Contrôle de la réentrance

Une opération de synchronisation, notamment quand elle aboutit à une suspension, est une opération coûteuse. Il convient de minimiser le nombre des cas de synchronisation qui conduisent à une suspension, ou encore de manière plus critique, à une attente active.

Un module réentrant appelé de façon procédurale peut être exécuté, à un moment donné, par un grand nombre de processus. La probabilité des conflits et donc les besoins de synchronisation augmentent alors de façon sensible. L'exécution des actions de synchronisation consomme alors une quantité croissante et non négligeable de temps de calcul.

En conséquence, globalement le temps moyen d'exécution d'une fonction augmente.

Le degré de réentrance, lieu d'exécution des processus cycliques, est égal au nombre de processus cycliques qui s'exécutent dans ce module. Ce degré de réentrance est choisi par le concepteur d'un module

afin d'être adapté au mieux à son problème.

Dans ces conditions, les actions de synchronisation qui aboutissent à une suspension devraient être peu fréquentes. On peut donc raisonnablement attendre de la part d'un module exécuté par des processus cycliques un "débit" égal ou supérieur à celui du même module appelé procéduralement.

Ce résultat a été démontré expérimentalement par l'équipe MULTICS en limitant la réentrance d'un compilateur et en constatant une augmentation de son débit.

III.3.4. Conclusion

Malgré quelques différences notables dans les structures de contrôle et de synchronisation, et parfois au niveau du volume du programme d'un module, la quantité d'instructions à dérouler pour exécuter une fonction est indépendante du mécanisme d'appel intermodule utilisé.

Par contre, le contrôle du degré de réentrance (utilisation de processus cyclique), en diminuant la probabilité des conflits entre processus, permet de minimiser le temps de calcul consommé par les actions de synchronisation.

Pour des modules fortement partagés, l'utilisation de processus cycliques permet d'éviter l'augmentation du temps d'exécution d'une fonction quand la charge du module augmente.

III.4. Conclusion générale

Le choix d'un mécanisme d'appel intermodule (appel procédural ou lancement d'un processus cyclique) dépend du type d'application envisagé.

L'appel procédural d'un module est moins coûteux qu'une activation de processus cyclique lorsque les modules sont dans un même espace d'exécution. Dans les autres cas, activer un processus cyclique est nettement moins coûteux. Pour les modules implantés sur des processus

distincts, le choix des processus cycliques s'impose.

La structure interne d'un module est très influencée par le mécanisme utilisé ; mais les différences relevées n'ont pas d'impact sur le coût de l'exécution des fonctions d'un module.

Toutefois, les modules réentrants et très partagés ont intérêt à contrôler leur degré de réentrance. Avec un appel procédural, ce contrôle ne peut se faire qu'avant ou après l'appel intermodule par des instructions supplémentaires. Les processus cycliques en permettent un contrôle efficace de la réentrance et donc dans certains cas une meilleure efficacité.

CHAPITRE IV

ASPECTS QUALITATIFS

Outre les questions de performance abordées au cours des deux chapitres précédents, il existe de nombreuses différences entre la communication intermodule par appels procéduraux et la communication intermodule par envoi de message à des processus cycliques.

Nous abordons dans ce chapitre l'analyse des différences les plus marquantes en essayant d'en faire ressortir l'impact sur le travail de conception et de réalisation d'un ensemble logiciel composé de modules.

IV.1. Généralités

Un mécanisme est plus puissant qu'un autre dans la mesure où les problèmes qu'il permet de résoudre constituent un sur-ensemble de ceux traités par le second mécanisme.

Il est toujours possible de convertir un appel procédural en un envoi de message à des processus cycliques. Cette conversion peut même être automatisée.

Par contre, le mécanisme de transfert de messages fournit diverses possibilités que ne procure pas l'appel procédural.

1) Asynchronisme

Un programme de la forme

```
        Proc T
          {
A         |
          }
        envoi (message 1, module X) ;
          {
B         |
          }
        envoi (message 2, module Y) ;
          {
C         |
          }
        attend (message 1 du module X) ;
          {
        attend (message 2 du module Y) ;
          }
```

ne peut être entièrement traduit en termes d'appels procéduraux.

Le langage PLITS [FE 77] fournit quelques extensions syntaxiques et sémantiques qui visent à résoudre ce genre de problème.

La réalisation effective de l'asynchronisme impose l'utilisation d'un mécanisme de communication par message ou d'une pile cactus.

Rappelons que, dans cette étude, on n'aborde pas les possibilités d'exécution asynchrones des diverses parties d'une même tâche, ceci afin de comparer les deux mécanismes dans le même domaine d'application. Le parallélisme que nous considérons est exprimé uniquement par la multiplicité des tâches en cours d'exécution (ou au cours de services que rendent les divers processus cycliques).

2) Modules matériels

Au chapitre III.1.2., on indique que l'appel procédural est très mal adapté aux communications dans un réseau d'ordinateur ou dans un système réparti et que, dans ces cas-là, l'appel par messages est couramment utilisé.

Il s'agit là d'un autre avantage essentiel de la communication par messages dont l'importance croît avec le développement rapide des réseaux d'ordinateur et des systèmes répartis.

3) En conclusion, sur le plan de la généralité d'application, le mécanisme de communication par messages rend de meilleurs services que le mécanisme de communication par appels procéduraux.

IV.2. Contrôle des processus

Nous allons présenter quelques-uns des problèmes liés au contrôle des processus, à savoir la malveillance, la comptabilisation et le contrôle des communications qui se trouvent être sensibles au mécanisme de communication utilisé.

IV.2.1. Contrôle des erreurs et anomalies

Nous avons vu que l'envoi d'un message entraîne la création d'une activité pour interpréter ce message. En raison de ce fonctionnement, un système de communication par messages peut connaître trois types d'anomalies que l'on ne rencontre pas dans un système de communication par appel procédural sans asynchronisme :

- un processus qui produit sans contrôle des messages crée autant d'activités et peut rapidement engorger le système de communication
- deux processus qui attendent tous deux un message en provenance de l'autre peuvent s'interbloquer
- un module qui oublie de renvoyer des messages fait disparaître des activités.

Dans ces conditions, il est utile de mettre en place les moyens nécessaires à la résolution de ces ennuis :

- la création de message doit être contrôlée. Dans HYDRA, la création est une primitive du noyau et dans OURS, on vérifie l'existence d'un processus associé au message
- dans OURS, un processus ne peut pas attendre un message particulier, ce qui évite la suspension définitive de ce prossus au cas où le message n'arrive pas
- les relations intermodules dans OURS sont contrôlées par un graphe sans boucle, ce qui assure l'absence des interblocages
- enfin, l'utilisation d'une horloge de garde permet de vérifier qu'un message n'est pas traité depuis un temps excessif.

Un dernier problème concerne l'effacement des données de contrôle d'un message. Sur un plan plus général, il est naturellement impossible d'assurer la protection de certains emplacements d'un message. Toutefois, un langage de programmation incluant la notion de message permettrait d'effectuer cette protection de façon statique. Si l'en-tête du message est détérioré, on considère qu'il s'agit d'une erreur de la part du module et elle reçoit le même traitement que les autres anomalies d'un module.

IV.2.2. Traitement d'anomalies

La communication par messages permet de générer très aisément des activités parallèles. Le noyau de communication peut donc se servir de cette facilité pour traiter les cas d'anomalie de fonctionnement d'un module.

A chaque module est associé un module "opérateur", chargé de traiter notamment les anomalies de ce module.

Les processus en activité dans le module en anomalie sont arrêtés et un message standard contenant les renseignements sur l'anomalie détectée est envoyé au module "opérateur".

Celui-ci prend les décisions adaptées aux circonstances ; il entreprend notamment le redémarrage ou la destruction des processus en cours d'exécution dans le module en faute (primitives du noyau). L'exécution du module opérateur est de type coroutine par rapport à l'activité du module surveillé.

De nombreuses variantes peuvent être apportées à ce schéma. La possibilité de faire aisément exécuter un module espion en parallèle avec le module espionné offre de grandes facilités dans le domaine de la récupération et de la détection des erreurs, mais aussi pour la surveillance et l'espionnage.

Le mécanisme des processus cycliques permet d'arrêter le processus en faute, donc de conserver intact l'état de ce processus au moment de l'anomalie et de lancer l'exécution d'un autre processus chargé d'analyser l'erreur.

Avec un appel procédural, on ne peut que rendre le contrôle à la procédure appelante en lui signalant l'anomalie. Ce faisant, l'état courant au moment de l'erreur est détruit.

IV.2.3. Comptabilisation des ressources

La comptabilisation consiste à recenser à tout moment les ressources consommées par un "utilisateur". Dans les systèmes "classiques", à chaque utilisateur est associé un processus. Avec l'appel procédural, le même processus s'exécute dans les divers modules. En conséquence,

la comptabilisation est généralement assez facile à mettre en oeuvre : à chaque processus est associé un bloc de contrôle dans lequel le système range les informations de consommation des ressources.

Dans un système à processus cycliques et à niveaux d'abstraction, la relation entre le processus serveur qui s'exécute et la tâche "responsable" de cette exécution est assez délicate.

En effet, soient deux niveaux d'abstractions : i et i+1.
Quand un processus P_{i+1} est interprété par un processus P_i, les ressources consommées par P_i sont aussi des ressources consommées indirectement par P_{i+1} et itérativement si P_{i+1} interprétait un processus P_{i+2} ....

Si nous désirons effectuer une comptabilisation qui permette, à la fois, de connaître les ressources consommées par les utilisateurs et les ressources consommées par chaque processus cyclique (et chaque module), nous pouvons utiliser le mécanisme suivant mis en oeuvre par le noyau :

A chaque processus sont associés deux blocs de renseignements comptables :

- un bloc "consommation propre" indiquant les ressources consommées par le processus quand il s'exécute
- un bloc "consommation indirecte" désignant les ressources consommées pour l'interprétation du processus. Lorsqu'un processus serveur (P_i) prend un message pour le traiter, ses deux blocs comptables associés sont nuls.

En fin de traitement du message (instruction Renvoyer), la somme des deux blocs comptables est ajoutée au bloc comptable "consommation indirecte" du processus (P_{i+1}) émetteur du message qui vient d'être traité.
Avant d'être reinitialisés à zéro, les deux blocs comptables du processus P_i sont ajoutés respectivement aux deux blocs comptables du module dans lequel P_i s'exécute.

Un tel mécanisme de comptabilisation permet la collecte de renseignements sur l'activité de chaque exécution d'un module et sur l'activité globale d'un module. Ce mécanisme pourrait être réalisé pour des appels procéduraux intermodules à condition que ceux-ci soient toujours interprétés par le noyau (GEMAU), ce qui n'est généralement pas le cas.

IV.2.4. Contrôle du retour des appels intermodules

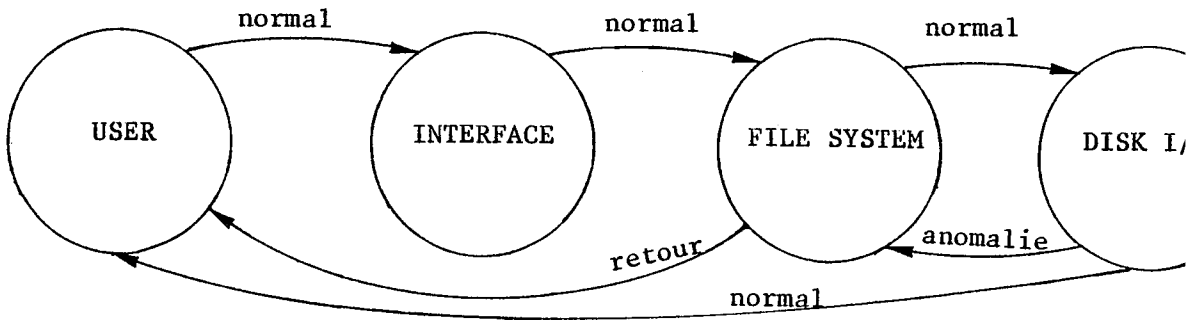
Dans un mécanisme classique, à chaque appel intermodule de type procédural correspond, au bout d'un temps borné, un retour de procédure.

Ce mécanisme trouve sa justification pour l'appel de procédure à l'intérieur d'un programme, le retour systématique limitant les risques d'erreur et étant de toute façon peu coûteux.

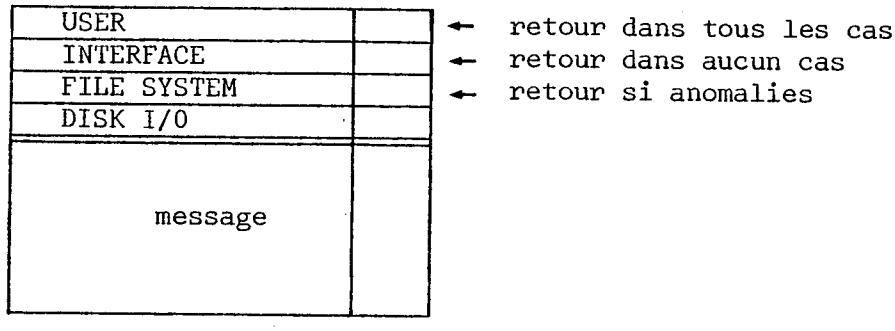
Il n'en va pas de même pour les appels intermodules. Parfois, lorsque son travail est terminé, un module en appelle un autre en ne désirant reprendre le contrôle que si des anomalies sont détectées plus loin dans la chaîne des appels intermodules. De plus, le retour d'un appel procédural intermodule est souvent coûteux et il convient d'éliminer ceux qui sont inutiles.

Le mécanisme consistant à transférer un message de module en module permet de trouver des solutions satisfaisantes à ce problème.

Reprenons un exemple tiré de HYDRA :



L'état du message en cours de traitement dans DISK I/O est celui-ci :



En fait, HYDRA dispose d'un mécanisme de retour conditionnel semblable aux "ON CONDITION" de PL1 ou aux ENABLE de BLISS. Ce mécanisme permet à un module qui effectue un appel procédural d'indiquer dans quelles conditions le retour correspondant doit être réalisé (16 conditions possibles). Tout module, en fin de traitement, émet la condition produite au cours de son exécution.

Dans ce mécanisme, le trajet des appels est figé, alors que le trajet des retours est dynamique et varie en fonction des occurrences des évènements qui sont apparus pendant l'exécution des modules de la chaîne. Par contre, les trajets des appels et des retours ne dépendent pas du processus interprété.

Dans OURS, nous avons voulu généraliser ce mécanisme et rendre le trajet du message dépendant à la fois du processus interprété et des conditions d'exécution émises par le module (comme pour les retours de Hydra).

Pour ce faire, nous considérons que chaque module dispose d'un certain nombre de sorties, chacune d'entre elles correspondant à une condition d'exécution du module (normal, anomalie 1, anomalie 2, ...).

Dans ces conditions, tout processus indique au noyau, grâce à un automate d'états finis, à quel module doit être connectée chacune des sorties du module courant.

Dans l'exemple précédent, l'automate de OURS pourrait être :

sortie / module	normal 0	anormal 1
USER	interface	
INTERFACE	file system	
FILE SYSTEM	disk I/O	user
DISK I/O	user	file system

Si nous reprenons l'exemple donné au § II.2.2

module \ sortie	0	1
IRIS	Pagination	
Pagination	IRIS	E/S physique
E/S physique	Pagination	

IRIS : sortie 0 ⇒ trap 40

Pagination : sortie 0 ⇒ faute de page résolue

sortie 1 ⇒ page à charger

E/S physique : sortie 0 ⇒ page chargée.

En conséquence, l'identité du processus auquel le message est associé, celle du module courant ainsi que la sortie utilisée déterminent sans ambiguïté le prochain module récepteur de ce message. La pile des appels telle qu'elle est utilisée dans HYDRA ne se justifie donc plus.

L'automate de contrôle est un objet du système OURS et peut être utilisé, sans duplication, par tous les processus qui désirent, pour leur message, le même comportement [BR 76b].

IV.3. Répartition des ressources

Il est clair que l'efficacité globale d'un système passe par la répartition optimale des ressources disponibles. Ce problème se heurte à deux difficultés : la détermination des ressources nécessaires à un processus et la faculté de prédire les besoins futurs.

Pourtant il existe de nombreux renseignements sur l'utilisation des ressources qui sont peu ou mal exploitées. Lorsqu'un processus effectue un appel intermodule, les besoins en ressources vont probablement changer ; les ressources nécessaires au module appelant risquent de ne plus être utilisées ; enfin, il est probable que le module appelé va exploiter d'autres ressources. Si l'appel intermodule n'utilise pas les services du noyau (liaison figée à l'édition de lien ou à la compilation), l'allocateur de ressources n'est pas prévenu et, de ce fait, des renseignements intéressants sont perdus.

L'appel procédural entre des modules statiquement connectés ne permet pas de réaliser aisément un bon contrôle de l'activité des processus, sauf si un compilateur très très évolué a calculé statiquement les ressources consommées et signale au noyau que des modifications vont intervenir.

Un processus cyclique exécute toujours le même code. Son comportement est donc très semblable d'une exécution à l'autre, même si les données en entrée diffèrent notablement. Ce processus aura donc un comportement constant et fortement localisé [BU 77]. Pour les modules fréquemment utilisés, il est facile de connaître le comportement de ses processus cycliques et de leur attacher définitivement un comportement type (classe de répartition, quantum, ressources maximales ou minimales).

Un appel intermodule, dans un système à communication par messages, se traduit par le lancement d'un processus qui possède les renseignements utiles à l'affectation précise des ressources qui lui sont nécessaires. Dans ces conditions, la tâche d'un allocateur de ressources se trouve grandement facilitée. On peut conclure en relevant le fait que les systèmes de communication par messages à des processus cycliques offrent, au niveau de la répartition des ressources, des facilités que ne permet pas toujours un mécanisme d'appel procédural.

IV.4. Contrôle de la charge

Nous savons que le degré de réentrance d'un module est égal au nombre de processus cycliques qui s'y exécutent. Nous avons vu que pour des raisons de coût de consommation d'espace en mémoire, le nombre de processus cycliques par module est borné. On conçoit alors que, pour un ensemble de modules donnés,de façon à réduire les goulots d'étranglement, il est nécessaire d'équilibrer les vitesses relatives d'exécution des divers modules.

Nous appellerons débit d'un module le nombre d'exécutions de ce module par unité de temps.

Comment le programmeur peut-il contrôler le débit d'un module ? Si ce module est soumis à l'appel procédural, la réentrance ne peut être

limitée que par programme car il n'existe pas d'outils particuliers de régulation du débit d'un module. Les modules les plus référencés sont susceptibles de constituer des goulots d'étranglement. En effet, pour un trop grand degré de réentrance, le débit d'un module décroît en raison notamment du coût des synchronisations d'accès aux variables d'état (cf. Chap. III.5). Les processus sont alors "freinés" dans les modules surchargés alors qu'il aurait été plus judicieux de les arrêter avant leur entrée dans le module, donc notamment avant que les ressources nécessaires à leur exécution dans ce module soient allouées.

Pour les modules exécutés par des processus cycliques, le débit peut être régulé en tenant compte de deux critères :

- le nombre de processus cycliques du module
- et les caractéristiques des processus du module.

Pour un faible nombre de processus, on a constaté que le débit croît avec le nombre de processus existant dans un module. Pour un nombre plus grand de processus, le débit finit par passer par un maximum, puis par décroître.

Chaque groupe de processus possède des caractéristiques propres : mode d'exécution (maître/esclave, réel/virtuel, B/C, ...), priorité, interruptibilité, classe de répartition, quantum, temps maximum d'exécution, prérogatives, ...

Ces caractéristiques influent directement sur la performance d'un module.

En résumé, on peut dire que l'appel procédural ne fournit pas directement d'outils adaptés à la régulation de la charge d'un module. Inversement, pour les processus cycliques, la détermination a priori du débit optimal d'un module est délicate. Seuls les essais en vraie grandeur doivent permettre d'ajuster au mieux les débits respectifs des divers modules.

IV.5. Systèmes répartis

Dans un système à communication par messages, tous les modules communiquent au moyen d'un interface unique qui leur permet de se désigner indépendamment de leur localisation.

Ainsi, que les modules soient dans une mémoire commune, implantés au câble sur des mini-ordinateurs distincts, ne modifie en rien la logique et le code des modules.

Il nous est donc possible de transformer graduellement un système "logiquement réparti" (tous les modules sont dans une mémoire commune, les messages transitent par recopie) en un système physiquement réparti (les modules sont sur des matériels distincts, les messages transitent sur des bus ou sur un réseau de transport).

Pendant la phase où le système est logiquement réparti, la logique des divers modules peut être testée ; ensuite, en soumettant une charge synthétique au système à tester, nous pouvons obtenir la répartition de la charge sur les divers modules.

En faisant varier la charge (en nombre de travaux à effectuer) et le type de charge (type des travaux à effectuer), on peut obtenir des abaques qui indiquent quel doit être le débit de chaque module pour que la charge globale à supporter soit convenablement répartie sur tous les composants du système (pas de goulot d'étranglement).

Le débit d'un module dépend des caractéristiques technologiques du processeur qui doit le supporter. Les abaques permettent donc de déterminer le débit qu'un module doit assurer pour que le système puisse supporter une charge maximale donnée. En fonction de ce débit, le concepteur peut alors choisir une technologie dans laquelle il réalisera ses processeurs pour que les modules aient l'efficacité voulue.

La connexion des différents modules matériels peut se faire au fur et à mesure de leur réalisation.

IV.6. Aisance d'écriture

De même que la comparaison entre deux langages de programmation s'effectu par rapport à un type de problème donné, de même la comparaison entre la facilité d'écriture de l'appel procédural et celle offerte par le mécanisme des processus cycliques est réellement significative dans la mesure où elle est effectuée par rapport à un même domaine d'applicati

Nous allons donc essayer de déterminer le terrain sur lequel les mécanismes d'appel intermodule sont comparables, ainsi que pour chacune des méthodes, la classe de problèmes qui sont le mieux traités.

Deux résultats semblent s'imposer : l'appel procédural est mal adapté aux systèmes répartis, tandis que les processus cycliques présentent peu d'intérêt pour les logiciels simples peu utilisés et de petit volume.

Le domaine de comparaison se limite alors aux problèmes du système et aux gros logiciels implémentés sur les machines classiques. Au sein même de ce domaine, nous allons déterminer les aspects qu'un mécanisme permet de traiter plus simplement que l'autre.

IV.6.1. Facilités permises par les processus cycliques

Le mécanisme des processus cycliques permet une écriture simple et efficace pour les modules nécessitant une exécution en exclusion mutuelle : par exemple, processus pilotant un périphérique, résolution de l'accès à une table, répartiteur de ressources.

Le module étant exécuté par un seul processus, l'exclusion mutuelle est réalisée de façon naturelle ; de plus, le fonctionnement du module est assez bien représentatif du fonctionnement du dispositif matériel (module pilotant un périphérique par exemple).

Si le module n'effectue pas de multiplexage, aucune instruction de contrôle ou de synchronisation n'est nécessaire et le module est réduit à son aspect fonctionnel, aucun problème parasite ne venant l'affecter.

Il est assez fréquent de rencontrer de telles situations dans les systèmes ; aussi la programmation aisée de ces modules avec le mécanisme des processus cycliques est-elle un avantage appréciable. Après les composants très multiplexés tels que UC, canaux, les modules de ce type ont été codés à l'aide des processus cycliques.

La communication par message présente l'avantage d'effectuer facilement du parallélisme : générer une tâche parallèle revient à créer un message et à l'envoyer au système de communication. Nous dirons même que dans un système à processus cycliques, c'est la limitation du parallélisme sauvage qui constitue un problème.

Nous venons de voir (cf. Chap. IV.2.1) les facilités relatives à la reprise des erreurs et des anomalies ; nous n'y reviendrons pas, mais cet avantage joint à la possibilité d'effectuer aisément de l'"espionnage" est assez intéressant.

IV.6.2. Difficultés dues aux processus cycliques

Un module ne peut être efficace que s'il est conçu en fonction de la méthode de contrôle utilisée. En particulier, écrire un module pour des processus cycliques impose, par rapport au même module soumis à l'appel procédural, la réalisation supplémentaire des actions suivantes :

- déterminer l'emplacement des appels intermodules et des synchronisations afin que le nombre des variables locales à conserver soit minimum et si possible nul. Des acrobaties déplaisantes peuvent être utilisées pour atteindre cet objectif ;
- déterminer et gérer les variables locales à conserver lors d'un appel intermodule ou lors d'une synchronisation ;
- gérer à la main le multiplexage dans la mesure où celui-ci n'a pas été intégré dans les primitives de synchronisation ;
- tenir compte, dans la structure générale du module, du retour ou de la reprise des messages, ce qui constitue un aspect délicat du contrôle et de la synchronisation ;
- décider du nombre de processus à utiliser dans un module ainsi que leurs caractéristiques.

La programmation à l'aide de processus cycliques présente un certain nombre de contraintes d'écriture dues en grande partie à l'inexistance d'outils de programmation adaptés à la mise en oeuvre de ce mécanisme.

IV.6.3. Facilités et difficultés de l'appel procédural

L'appel procédural est bien connu avec l'avantage essentiel de la simplicité.

Il faut toutefois reconnaître que cette simplicité est due en partie à l'aspect systématique du mécanisme avec notamment sur appel de procédure, empilement de toutes les variables locales.

Une optimisation réalisable, en particulier par la recherche des variables d'état non rémanentes (cf. Chap. II.1.2) compliquerait sensiblement ce mécanisme et l'amènerait probablement à un degré de complexité comparable à celui du mécanisme des processus cycliques.

Enfin et surtout, l'appel procédural est un mécanisme bien connu des programmeurs et donc utilisé correctement et sans perte de temps.

IV.6.4. Conclusion

Le mécanisme des processus cycliques impose en pratique l'utilisation du principe des niveaux d'abstraction. L'effort supplémentaire qui en découle devrait en contrepartie

- limiter les difficultes de mise au point
- apporter des facilités au niveau du contrôle et de l'équilibrage de la charge
- enfin, fournir des spécifications plus complètes et plus claires.

Les processus cycliques se prêtent bien à la mise en oeuvre de quelques types d'applications. Toutefois, pour les autres applications, ce mécanisme présente une certaine lourdeur que l'on ne rencontre pas dans l'appel procédural. Ceci constitue, à mon sens, le reproche essentiel que l'on puisse faire aux processus cycliques et il est vrai qu'il s'agit là d'un défaut d'importance.

Cette caractéristique implique que la programmation utilisant le mécanisme des processus cycliques est actuellement réservé aux logiciels où l'efficacité est un objectif essentiel et aux logiciels demandant le contrôle du parallélisme ou de l'asynchronisme.

Cette conclusion peut être nuancée en remarquant que beaucoup de contraintes d'ecriture spécifiques aux processus cycliques peuvent être traitées par un compilateur spécialisé, ce qui devrait rendre, une fois la phase d'adaptation effectuée, l'utilisation des processus cycliques à peu près aussi "agréable" que celle de l'appel procédural.

Conclusion

Pour une décomposition d'un logiciel faisant apparaître des modules, nous pensons avoir montré que le mécanisme de communication inter-module par message permet, dans certains cas, un gain de place mémoire appréciable (modules chargés ou multiplexés) et parfois un gain de temps à l'exécution.

L'absence d'outils adaptés explique les difficultés d'écriture rencontrées dans l'utilisation de la communication par message et sa relative rareté dans les logiciels classiques.

Nous sommes convaincus que les recherches entreprises sur ce thème apporteront les outils nécessaires à une utilisation commode de la communication par message, débouchant entre autre sur une meilleure utilisation des possibilités de l'asynchronisme et du parallélisme. La communication par message pourra alors être légitimement utilisée dans les cas où elle est plus efficace et ceux où elle permet plus de facilités.

Nous sommes également convaincus que dans bien des cas, la simplicité de l'appel procédural, son efficacité dans les logiciels simples font que ce mécanisme reste parfaitement adapté à la plupart des logiciels ; seul le quasi "monopole" de ce mécanisme est mis en cause par la communication par message.

De plus, comme nous l'avons vu (Chap. II.2.3) à l'intérieur d'une même application, un mécanisme peut se révéler avantageux à un "niveau" et pas à un autre.

Il faudra peut-être envisager des systèmes "mixtes" où, aux niveaux supérieurs, les appels sont procéduraux et sont traduits plus "bas" (près de la machine) par une communication par message.

C'est une voie qui semble être déjà intuitivement utilisée dans des applications actuelles (systèmes répartis, systèmes connectés à des réseaux, ...).

BIBLIOGRAPHIE

[BE 72] A. BENSOUSSAN, C.T. CLINGEN, R.C. DALEY
The multics virtual memory : concept and design.
C. ACM 15.5, May 1972.

[BE 75] Y. BECKERS, HERMAN, RAYNAL
Environnement fonctionnel pour une machine langage de haut niveau.
Thèse de 3ème Cycle, Rennes, Septembre 1975.

[BR 69] P. BRINCH-HANSEN
RC 4000 software multiprogramming system.
A/S Regencentrallen-capenhagen, 1969.

[BR 74] J. BRIAT, S. GUIBOUD-RIBAUD
Espace d'adressage et espace d'exécution dans le système GEMAU.
International symposium on operating systems theory and practice. Paris, April 1974.

[BR 76a] J. BRIAT, X. ROUSSET de PINA, J.P. VERJUS
Le projet OURS ; ses principes.
Congrès AFCET, Paris 1976.

[BR 76b] J. BRIAT, J. ESTUBLIER
Manuel technique hardext. OURS 2001.

[BR 76c] J. BRIAT, J. ESTUBLIER
Manuel langage hardext. OURS 3000.

[BU 77] BURJEVIN
Mesures sur le comportement des programmes en vue de la modélisation des systèmes informatiques.
Thèse Docteur-Ingénieur, Université de Rennes, Mai 1977.

[CR 75] CROWLEY, MILLING
The control system for the SPS.
CERN LAB 11-CO/A5-3, Mai 1975.

[DJ 68] E.W. DIJKSTRA
The multiprogramming system.
C. ACM 9.3, March 1968.

[DU 74] J.P. DUPUIS
Amélioration fonctionnelle dans un système d'opération à trois niveaux de hiérarchie.
Mémoire CNAM, Grenoble, Mars 1974.

[ES 76] J. ESTUBLIER, A. TARABOUT, I. VATTON
Manuel langage MAS-UGR, OURS 3001.

[FA 73] FABER, FELDMAN, HEINRICH, HOPWOOD, LARSEN, LOOMIS, ROWE
The distributed computing system.
Proc. 7 IEEE computer soc. international conference, 197

[GR 75] S. GUIBOUD-RIBAUD
Mécanismes d'adressage et de protection dans les systèmes informatiques. Application au noyau GEMAU.
Thèse d'Etat, Juin 1975, Grenoble.

[HA 68] E.A. HAUCK, B.A. DENT
Burroughs B6500/B7500 stock mécanisme proceeding AFIPS SJCC 1968.

[IRIS] IRIS 80
Manuel d'utilisation. Réf. 4152 B/FR.

[IV 76] I. VATTON
Segmentation pagination dans OURS. OURS 2000.

[LE 77] J. LEROUDIER
Systèmes adaptatifs à mémoire virtuelle.
Thèse d'Etat, Grenoble, Mai 1977.

[LIS] Manuel de références.

[LP 80] M. PEQUIGNOT
LP80, metteur au point.
Manuel d'utilisation. 1977. Centre Interuniversitaire de Calcul de Grenoble.

[MA 76] G. MAZARE
MCB asymmetric multi-processor system.
Proceedings of the 2nd euromicro symposium. Venice, 1976.

[MO 77] J. MOSSIERE
Méthode pour l'écriture des systèmes d'exploitation.
Thèse d'Etat. Septembre 1977, Grenoble.

[OR 74] ORNSTEIN, BARKER, BRESSLER, CROWTHER
The BNN multiprocessor.
7th annual Hawaï international conference on system science. 1974.

[PA 72a] D.L. PARNAS
Information distribution aspects of design methodology.

[PA 72b] D.L. PARNAS
A technique for software module specification with example.
CACM 15.2, 1972.

[REE 76] R.E.E. MICRAL M
Manuel de références.
Réf. G.09.F. Janvier 1976.

[TR 76] A. TARABOUT
Gestion des E/S disque et tambour. OURS 2002.
ESPHISIQ ESLOGIR réalisation des E/S. OURS 2003.

[WI 72] R.K. WILLIAMS
System 250 : basic concepts.
Proceeding conference on computer system and technology.
IERE, London, October 1972.

[WL 75] W. WULF
The hydra operating system, pp. 93-118.

[WL 74] W. WULF, F. COHEN, W. CORWIN, A. JONES, R. LEVIN, C. PIERSON, F. POLLAK
Hydra : the kernel of a multiprogramming system.
C. ACM 17.6, June 1974.

```
COMPILATEUR LP60 - VERSION 2.2 -  CENTRE INTERUNIVERSITAIRE DE CALCUL DE GRENOBLE -  DECEMBRE  1977

  CARTE      @DON   @INS   PF
  01    1362 0000   0FA7   01                &
  01    1363 0000   0FA7   01 C              *    AIGUILLAGE VERS LES FONCTIONS DU SUPERVISEUR
  01    1364 0000   0FA7   01 C              *    ------------------------------------------
  01    1365 0000   0FA7   01 C              &
  01    1366 0000   0FA7   01
  01    1367 0000   0FA7   01      &
  01    1368 0000   0FA7   01 C    * LE REGISTRE DE BASE B4 POINTE SUR LE MESSAGE D'ENTREE DECRIVANT LA COMMANDE.
  01    1369 0000   0FA7   01 C    * L'AIGUILLAGE VERS LES DIVERSES FONCTIONS DE ESPHYSIQ SE FAIT DANS UNE
  01    1370 0000   0FA7   01 C    * INSTRUCTION "CASE OF" DONT LE SELECTEUR EST LE CODE FONCTION CONTENU
  01    1371 0000   0FA7   01 C    * DANS L'ENTETE DU MESSAGE.
  01    1372 0000   0FA7   01 C    &
  01    1373 0000   0FA7   01
  01    1374 0000   0FA7   01
  01    1375 0000   0FA7   01       POINT+U'ENTREE:          INITIALISATION DU PROCESSUS CYCLIQUE                    ALLOCATION STATIQUE
  01    1376 0000   0FA9   01      &--------------&                                                                  D'UNE ZONE DE
  01    1377 0000   0FA9   01                                                                                       MEMOIRE POUR LES
  01    1378 0000   0FA9   01           B0:= RF;                                                                    VARIABLES LOCALES
  01    1379 0000   0FAA   01           R1:= L'LOCAL+ESPH;
  01    1380 0000   0FAB   01           MEM+ALLOC;                          & ALLOCATION DONNEES LOCALES  &
  01    1381 0000   0FAE   01           B2:= R2;
  01    1382 0000   0FAF   01           INIT+LOCAL;
  01    1383 0000   0FB0   01           RF:= B0;
  01    1384 0000   0FB1   01           PUSH(ADRETOUR,RF);
  01    1385 0000   0FB2   01           LOOP BEGIN       BOUCLE DU PROCESSUS CYCLIQUE
  01    1386 0000   0FB2   02                R1:= 0;
  01    1387 0000   0FB3   02                    +ATTEND;                  & ATTENTE MESSAGE             &
  01    1388 0000   0FB4   02                R1:= MES+CF;                  & CODE FONCTION               &
  01    1389 0000   0FB5   02                R1:= R1-BASE+CF+ESPH;
  01    1390 0000   0FB6   02                CASE R1/15 OF                 & AIGUILLAGE                  & R1 contient le code fonction
  01    1391 0000   0FC7   02                     ALLOUERPERIPH                                            trouvé dans le message
  01    1392 0000   0FC7   02                OR   LIBERERPERIPH                                            ramené à l'intervalle 1 à 15
  01    1393 0000   0FC9   02                OR   RECONFIGURER
  01    1394 0000   0FCB   02                OR   MONTERSURPERIPH
  01    1395 0000   0FCD   02                OR   DEMONTERPERIPH
  01    1396 0000   0FCF   02                OR   TRANSFERTCONSOLE
  01    1397 0000   0FD1   02                OR   TRANSFERTNORMAL
  01    1398 0000   0FD3   02                OR   TRANSFERT+ST
  01    1399 0000   0FD5   02                OR   TRANSFERT+ECHOPLEX
  01    1400 0000   0FD7   02                OR   ARRET2
  01    1401 0000   0FD9   02                OR   ARRET3
  01    1402 0000   0FDB   02                OR   TEST1
  01    1403 0000   0FDD   02                OR   TEST2
  01    1404 0000   0FDF   02                OR   TRANSFERTOPTIMISE
  01    1405 0000   0FE1   02                OR   NULL;
  01    1406 0000   0FE3   02                END;
 AUCUNE ERREUR
JOB STEP 01 TERMINATED AT  11*28*11* AFTER 0000.59 MIN
.       CORE USED  0099      DISC USED  0000    WAIT TIME 0000.17
*******************************************************************************
```

Extrait du module ESPHYSIQ de A. TARABOUT

```
COMPILATEUR LP80 - VERSION 2.2 -  CENTRE INTERUNIVERSITAIRE DE CALCUL DE GRENOBLE -  DECEMBRE  1977

  CARTE     @DON   @INS   PF
  01   0142 0000  0BF5  01              &*****************************************************************
  01   0143 0000  0BF5  01 C            *                                                               *
  01   0144 0000  0BF5  01 C            *                  L I B E R E R P E R I P H                     *
  01   0145 0000  0BF5  01 C            *                                                               *
  01   0146 0000  0BF5  01 C            *      FONCTION: TRAITEMENT DE LA PRIMITIVE LIBERERPERIPH       *
  01   0147 0000  0BF5  01 C            *      ---------                                                *
  01   0148 0000  0BF5  01 C            *                                                               *
  01   0149 0000  0BF5  01 C            *****************************************************************&
  01   0150 0000  0BF5  01
  01   0151 0000  0BF5  01
  01   0152 0000  0BF5  01
  01   0153 0000  0BF5  01
  01   0154 0000  0BF5  01
  01   0155 0000  0BF5  01      PROCEDURE LIBERERPERIPH;
  01   0156 0000  0BF6  01      BEGIN
  01   0157 0000  0BF6  02      LABEL FIN+LIBERER;
  01   0158 0000  0BF6  02           OUT(CHAR,TXT+LIBERER);                                  &* TEST *&
  01   0159 0000  0BF7  02           PUSH(ADRETOUR,RF);
  01   0160 0000  0BF8  02           VERIFIERNU(,LIB+NOMPER);
  01   0161 0000  0BFA  02           IF R0+=0 THEN GOTO FIN+LIBERER;
  01   0162 0000  0BFD  02           LIBERERNU;
  01   0163 0000  0BFE  02           DOP+M+ETATPER:= RE:= DOP+M+ETATPER AND M'LIBERE;
  01   0164 0000  0C01  02           IF (RF:=DOP+ABCES)↑=PTVIDE THEN BEGIN        & TRANSFERTS EN COURS        &
  01   0165 0000  0C04  03                R0:= ERR+TRF+EN+COURS;
  01   0166 0000  0C05  03                ARRETPERIPH;
  01   0167 0000  0C06  03                END;
  01   0168 0000  0C06  02      FIN+LIBERER:
  01   0169 0000  0C06  02           SB'MES(R2:=IB'MES+CERR):= R0;
  01   0170 0000  0C08  02               +RENVOI(,0);
  01   0171 0000  0C0A  02           PULL(ADRETOUR,RF);
  01   0172 0000  0C0B  02           END;
```

dernière page de la thèse

AUTORISATION DE SOUTENANCE

VU les dispositions de l'article 3 de l'arrêté du 16 Avril 1974,

VU le rapport de présentation de :

- Monsieur J.P. VERJUS, Professeur à l'Université de RENNES

Monsieur Jacky E S T U B L I E R

est autorisé à présenter une thèse en soutenance pour l'obtention du titre de DOCTEUR de TROISIEME CYCLE, spécialité "Génie Informatique".

Grenoble, le 14 Mars 1978

Ph. TRAYNARD
Président
de l'Institut National Polytechnique