THESE

présentée à

Université Scientifique et Médicale de Grenoble

Institut National Polytechnique de Grenoble

pour obtenir le grade de

DOCTEUR de 3ème CYCLE

par

Martine LUCAS

Jean MONTUELLE

CONCEPTION MODULAIRE DES SYSTEMES D'EXPLOITATION. présentation du projet SESAME

Thèse soutenue le 27 juin 1977 devant la Commission d'Examen :

Monsieur	G. VEILLON	: Président
Monsieur	S. KRAKOWIAK	: Rapporteur
Messieurs	J. BELLINO C. BETOURNE J.P. VERJUS	: Examinateurs

Le projet SESAME est développé au sein de l'équipe de recherche "Outils et méthodes pour la conception et l'écriture de systèmes d'exploitation" du Laboratoire d'Informatique de Grenoble. Ce travail a été effectué en commun par Bernard CASSAGNE, Jean-Louis CHEVAL, Flaviu CRISTIAN, François GAUDUEL, Jean-Michel GUILLOT, François GUYOT, Jean-Marc HERODIN, Sacha KRAKOWIAK, Martine LUCAS, Jean MONTUELLE et Jacques MOSSIERE. Cette étude a été menée dans le cadre d'un contrat SESORI (N° 75101).

TABLE DES MATIERES

3. LE BIBLIOTHECAIRE

DEUXIEME PARTIE

PREMIERE

PARTIE

INTRODUCTION

Le but du projet SESAME (Système d'Ecriture de Systèmes par Assemblage de Modules Elémentaires) est de réaliser un ensemble d'outils de production de gros programmes, et plus particulièrement de systèmes d'exploitation.

Les outils proposés par SESAME sont organisés autour de la notion de module. Un module est une unité de construction de systèmes. Un module comprend :

- un ensemble de données,
- un ensemble de procédures ou fonctions travaillant sur ces données.

Les seuls objets accessibles de l'extérieur du module sont des procédures ou des fonctions définies comme externes.

On trouvera dans la thèse de Martine LUCAS une présentation détaillée de la notion de module et une introduction à un ensemble d'outils adaptés à la programmation modulaire. La présence de ces outils facilite l'application d'une méthode rationnelle de conception de gros programmes, pour aboutir à une décomposition dont chaque partie est spécifiée par son interface et peut être réalisée sous forme de module. On trouvera dans la thèse de Jean MONTUELLE une présentation de cette méthode et un exemple d'application.

Ce fascicule est plus technique, il présente les différents aspects du projet SESAME sous la forme des outils de production et d'exécution offerts aux utilisateurs.

Les systèmes qui sont produits le sont pour un miniordinateur (dans notre cas un T1600). Comme il serait difficile d'implanter nos outils de production sur un petit calculateur, nous avons décidé de les mettre en oeuvre sur un IRIS 80. Nous distinguerons de ce fait deux parties principales dans la présentation qui va suivre. Une partie consacrée à la machine de production (qui délivre un texte binaire représentant le système), une autre à la machine hôte (où s'exécute le système).

Dans la première partie nous décrirons les trois outils sur lesquels ont porté nos efforts principaux :

i) Un langage de haut niveau, inspiré de PASCAL, permet l'écriture de modèles de modules. Un modèle est un texte source comportant des paramètres (appelés métavariables) et permettant d'engendrer un module (texte objet) après affectation de valeurs aux métavariables. Ce paramétrage et la possibilité de reporter (après compilation) certaines liaisons avec d'autres modules facilitent le réutilisation de modèles déjà existants.

ii) Un langage distinct du précédent qui constitue en quelque sorte le langage de commande de la production. Il permet d'écrire des programmes de connexion dont l'interprétation commande la création (ou la recherche) des différents constituant d'un système (ou d'un sous-système) et leurs connexions. Il définit l'interface que présente le système ainsi construit.

iii) Le bibliothécaire chargé de la conservation des différents textes source ou objet composant les versions partielles ou complètes des systèmes produits. Il apporte une aide matérielle importante dans la manipulation de textes (il est associé à un éditeur de textes) vis-à-vis de méthodes plus traditionnelles (cartes, "carnet de bord", ...). De plus, il connaît la nature des objets qu'il gère et des relations qui les lient. Ceci permet de contrôler les manipulations qui sont demandées et de fournir divers renseignements utiles à la recherche des textes.

Dans la seconde partie, nous nous intéressons au support offert par SESAME pour l'exécution des systèmes produits avec les outils précédents.

i) Les systèmes produits ont une structure d'exécution traduisant leur structure modulaire. Les mécanismes d'adressage utilisés doivent garantir si possible les règles d'accès et de protection que suppose la notion de module. Une telle structure d'exécution adapté au T1600 est proposée. Les mécanismes de changements de contexte lors des appels et retours de procédure ou fonctions externes y sont particulièrement développés puisqu'il sont directement liés à la modularité.

ii) L'utilisation de certains mécanismes propres au matériel ne peut être exprimé directement au niveau du langage d'écriture (interruptions, déroutements, lancement d'entrées/sorties, changement de mode d'exécution,...). D'autre part la traduction du parallélisme (exprimé dans le langage d'écriture des modèles) nécessite la présence, lors de l'exécution, de primitives pour la gestion de processus. Ces remarques entraînent l'existence d'un moniteur d'exécution constituant le noyau de tout système.

Pour pouvoir prendre en compte des particularités propres à certains systèmes, c'est en fait toute une famille de moniteurs d'exécution qui est proposée. En respectant les spécifications générales des primitives (interface) il est possible de traduire ces particularités par une réalisation particulière des modules constituant un moniteur. Une décomposition en modules est proposée pour la gestion de la multiprogrammation. Des exemples de réalisation sont fournis.

Un autre aspect du projet est de vérifier la validité de la méthode et des outils employés. Pour cela nous développons la construction, de toutes une chose de systèmes simples, à partir d'un ensemble de modules et en respectant dans son intégralité le mode de production que nous proposons. Cette partie intitulée PROSPER (PROduction de Systèmes pour l'Enseignement et la Recherche) n'est pas développée ici.

1. LE LANGAGE D'ECRITURE DES MODELES DE MODULES

1.1. Choix du langage

C'est dans le langage d'écriture que sont programmés les modules composant un système produit par SESAME. Les textes qui sont écrits sont en fait les modèles de ces modules. Un modèle est un programme paramétré qui décrit toute une famille de modules. Un représentant de cette famille peut être obtenu par compilation après affectation de valeurs aux métavariables (paramètres de génération) du modèle.

Le domaine d'application de ce langage est l'écriture de systèmes d'exploitation, plus généralement de tout logiciel possédant les particularités suivantes :

- composition par assemblage de parties construites indépendamment,
- possibilité d'activités parallèles,
- possibilité d'accès à des fonctions spécifiques du matériel (mécanismes d'interruption, d'entrée-sortie...).

Lors de la définition d'un langage de programmation, le choix des types de données utilisables est l'un des points les plus délicats : si ces types sont mal adaptés aux problèmes à traiter, le programmeur consacrera une part notable de son activité à la traduction des données de son problème en terme des structures disponibles. De plus une telle traduction ne peut que diminuer la lisibilité et les possibilités de modification de programmes.

La possibilité de définir des objets abstraits (sous forme de modèle) passe donc par un langage permettant la définition de nouveaux types. D'autre part la structuration des programmes est encouragée par :

- la possibilité de définir la portée des identificateurs à un ensemble restreint de procédure ou fonction (structure de bloc),
- l'utilisation, pour la représentation des algorithmes, de structures de contrôle basées sur des constructions "logiques" (séquence, répétition, alternative, choix).

Plutôt que de redéfinir un nouveau langage (et donc écrire son compilateur), il nous a semblé préférable d'adapter, à notre domaine d'application, un langage déjà existant et offrant les facilités énoncées ci-dessus. Notre choix s'est porté sur PASCAL [7] pour plusieurs raisons :

- ce langage tend actuellement à s'imposer comme langage de base pour ce genre d'application [2,11,6,1,5]
- le compilateur est écrit en PASCAL ce qui facilite sa modification
- une version est disponible sur l'IRIS 80 [10]

Dans les extensions nous avons porté notre effort sur l'expression de la modularité et du parallélisme (processus et synchronisation). Par contre un point n'est pas encore résolu par notre langage : permettre au programmeur

- de définir (simplement et jusqu'au niveau le plus fin) l'implémentation de ses structures de données;
- d'utiliser (à son gré) les mécanismes dépendant du matériel (lancement d'entrée-sortie; interruption, déroutement, appel au superviseur; changement de mode, de protection...)

Pour l'instant les modules qui sont nécessairement liés au matériel doivent être programmés dans un langage plus adapté (ici assembleur ou PL1600). Ils représentent un volume de code assez restreint (module de gestion d'un périphérique, de l'horloge...).

L'exécution des programmes produits par SESAME nécessite la présence de primitives sur la machine hôte (T1600) : allocation de l'espace mémoire, gestion du parallélisme... Ces primitives sont fournies par un moniteur d'exécution. Les modules composant ce moniteur seront eux aussi écrits dans un langage de "bas niveau". Ce moniteur d'exécution constitue le noyau du système produit, c'est une partie de ce système. Pour permettre une réalisation de ce noyau adaptée au système que l'on désire construire, les primitives choisies sont valables pour une famille assez large de telles réalisations (voir 5).

Enfin nous verrons que des opérations sur des représentations d'entiers permettent d'effectuer des manipulations au niveau de chaque élément binaire (mais en perdant la notion d'abstraction : en particulier la lisibilité).

Nous avons également apporté des restrictions ou des suppressions, lorsque la présence de certaines facilités de PASCAL nous semblaient contestables en regard du domaine d'application visé (suppression du type fichier par exemple) ou lorsque leur utilité ne nous semblait pas essentielle par rapport au travail de compilation ou à la rapidité d'exécution (suppression du qualificatif *packed* par exemple).

Pour présenter notre langage d'écriture des modèles nous ne signalerons que les points qui le différencient de PASCAL SFER [10] (nous reprenons l'ordre de présentation de ce manuel). Nous le ferons tout d'abord pour les différents éléments de programmation séquentielle, nous présenterons ensuite l'expression de la multiprogrammation.

Les cartes syntaxiques du langage figurent dans l'annexe 4.

1.2. Eléments du langage pour la programmation séquentielle

1.2.1. Vocabulaire

i) Identificateurs

Il est possible d'utiliser le caractère "souligné" (_) dans les identificateurs.

ii) Symboles réservés

Les symboles suivants ne sont plus réservés :

file, packed, program, value

- La notion de fichier est absente ici. La raison principale est que le langage est destiné à construire des systèmes de gestion de fichiers (comme partie de systèmes d'exploitation) plutôt que d'en proposer une réalisation particulière.
- Nous avons choisi de représenter les entiers et les caractères sous forme "tassée" (entier sur un mot, caractère sur un octet). Seules les variables logiques (*boolean*) ont une représentation

"élargie" (sur un octet). Cependant la présence d'opérations sur des représentations d'entiers permet de manipuler chaque élément binaire d'un mot. Ces différentes raisons expliquent la disparition du qualificatif *packed*.

- La notion de *program* est remplacée par celle de modèle (*pattern*).
- Il est possible de donner la valeur initiale de certaines variables dans la partie déclaration de ces variables. Le symbole *value* n'est donc plus utilisé.

Par contre de nouveaux symboles réservés ont été introduits. Il s'agit de :

dummy, env, ext, pattern, ref, result

Nous verrons leur emploi dans la suite de ce chapitre.

1.2.2. Définition de données

i) Déclaration de constantes

Pour une constante, une déclaration associe à un identificateur un type et une valeur constante. Le type peut être "entier" (*integer*), "caractère" (*char*) ou "chaîne de n caractères" (*array [1..n] of char*). Pour les entiers la valeur constante peut être exprimée sous forme d'une expression de constantes (calculable à la compilation).
Par exemple *const nb3 : integer = nb2 - 3 * nb1*

ii) Déclaration de types

Le mode de structuration *file* n'existe pas en SESAME.
Le type *real* n'est pas implémenté dans la première version du compilateur (peu d'intérêt pour les systèmes).

iii) Déclaration de variables

Une déclaration de variable associe à un identificateur un type (simple ou construit) et éventuellement une valeur initiale. Il est possible d'associer à plusieurs identificateurs le même type et la même valeur initiale.

Les types de variables pour lesquels une telle initialisation est possible sont :

- les types simples,
- les enregistrements (*record*) dont tous les champs sont de type simple,
- les tableaux dont les éléments sont de type simple.

Dans le cas d'un type simple (*integer, boolean, char*) on doit avoir une seule expression constante dans la partie initialisation de la déclaration. Dans le cas d'un enregistrement à n champs ou d'un tableau à n éléments, la partie initialisation doit comporter p expressions constantes (avec $p \leq n$). Les p premiers champs ou éléments sont initialisés dans l'ordre des expressions. Si dans la partie initialisation une même expression doit être répétée plusieurs fois à la suite, il est possible de ne l'écrire qu'une fois en la faisant précéder d'un facteur de répétition.
Par exemple :

```
var t : array [1..5] of integer := 0,0,0,0,0
```

peut être écrit plus simplement par :

```
var t : array [1..5] of integer := 5(0)
```

1.2.3. Expressions

i) Opération sur représentation d'entier

Dans les expressions entières il est possible d'employer certains opérateurs qui peuvent se définir comme des opérateurs de décalage ou des opérateurs logiques sur la représentation (complément à 2) d'un entier. La longueur (nombre de positions binaires) de cette représentation dépend de l'implémentation.

. Opérateurs de décalage

Les opérateurs de décalage ont la même priorité que les opérateurs de multiplication et de division. Ces opérateurs sont :

SRA (shift right arithmetic)
SLA (shift left arithmetic)
SRC (shift right circular)
SLC (shift left circular)
SRL (shift right logical)

SLL *(shift left logical)*

Ils doivent avoir des entiers comme opérandes. Le terme de gauche est celui sur lequel porte l'opération de décalage, le terme de droite indique le nombre de positions binaires de décalage. Il est pris modulo la longueur de la représentation.
Par exemple :

4 SLA 2 vaut *16*

. Opérateurs logiques
Les opérateurs *and*, *or* et *not* utilisés avec des opérandes booéens peuvent aussi s'employer avec des opérandes entiers.
Dans ce cas, les opérations *and*, *or* sont effectuées entre chaque élément binaire de la représentation des deux opérandes entiers; l'opération *not* est effectuée sur chaque élément binaire de la re-représentation de l'opérande entier.
Par exemple, lors d'une compilation, on désire produire une instruction dont l'octet de gauche représente un code opération et l'octet de droite un déplacement. Si 112 est la valeur (décimale) du code opération et 6 le déplacement de l'opérande, il est possible d'écrire la valeur de cette instruction sous la forme :

(112 SLL 8) or 6

ii) Restriction sur les manipulations d'ensembles
Les variables de types ensemble (*set*) peuvent prendre des valeurs écrites sous la forme d'intervalle :

[< expression1 >..< expression2 >]

Dans ce cas les deux expressions doivent pouvoir être calculées à la compilation.
Par exemple [*i-j .. i+j*] sera accepté si *i* et *j* sont des constantes.

1.2.4. Instructions

i) Instruction de sélection

L'instruction de sélection (*case*) définit une liste d'instructions dont une seule sera exécutée en fonction de la valeur d'une expression (celle qui suit le symbole *case*). Quand cette expression vaut *val*, c'est l'instruction "étiquetée" par *val* qui est exécutée. En PASCAL si *val* "n'étiquette" aucune instruction, le résultat n'est pas défini. En SESAME il est possible de rajouter une étiquette *else* devant l'instruction à exécuter en pareil cas.

Par exemple si *i* et *j* sont deux entiers, il est possible d'écrire

```
case i + j of
1 :   instruction1 ;
100 :  instruction2 ;
1000 :  instruction3 ;
else :  instruction4 ;  % cette instruction est exécuté
                        lorsque (i+j) est différent de
                        1, 100 ou 1000 %
```

ii) Appel de procédure

Il n'est pas possible de passer une procédure ou une fonction en paramètre. L'écriture de l'appel et les modes de passages autorisés pour les paramètres effectifs dépendent de la nature de la procédure ou fonction appelée. Cette procédure ou fonction a pu être déclarée (dans le même modèle) sans être précédée du qualificatif *ext*. Dans ce cas et à part la restriction introduite plus haut, l'appel se fait comme en PASCAL.

Si *ext* a été utilisé dans la déclaration (figurant le plus souvent dans un autre modèle), il s'agit d'une procédure ou fonction externe. Nous traiterons ce cas dans un paragraphe suivant.

iii) Fonctions standards

Comme les notions de fichiers, de réels, et de variables "tassées" sont absentes ici les fonctions standards qui subsistent sont :

ord(x)	rang de x dans l'ensemble des valeurs défini par le type de x.
char(x)	x est entier.Donne le caractère de rang x
succ(x)	Valeur du successeur de x (s'il existe) dans l'ensemble des valeurs de son type (scalaire ou intervale).
pred(x)	Valeur du prédécesseur de x (s'il existe)
new(p)	Alloue une nouvelle variable et affecte à la variable pointeur p son adresse.
dispose(p)	Libère la variable référencée par p (La gestion du tas dépend du moniteur d'exécution choisi)

D'autres fonctions standards ont été rajoutées (elles sont liées à l'implémentation).

oneof(e)	e est un ensemble. Fournit un élément de cet ensemble. L'implémentation choisie induit un certain ordre sur cet ensemble (celui de la déclaration du type). C'est le premier élément qui est choisi.
ad(v)	Fournit l'adresse de la variable v. Pour un tableau, donne l'adresse de son descripteur (origine réelle, origine virtuelle).
size(v)	Fournit la taille de la représentation de la variable v.

Ces deux dernières fonctions sont plus particulièrement destinées aux interfaces avec des modules de bas niveaux (réalisé en dehors de SESAME).

1.2.5. Textes source

i) Modèle

Un modèle de module est une unité de compilation. Il est constitué d'un ensemble de déclarations. Cet ensemble est délimité par *pattern* et *end*. L'en-tête du modèle contient l'identificateur du modèle, et éventuellement une liste de métavariables. Ces dernières sont des paramètres "formels" qui seront remplacés par des paramètres

"effectifs" lors d'une compilation permettant d'obtenir un module (binaire). Par exemple, avec un modèle ayant l'en-tête suivante :

pattern table (&taille:integer;&genre:type)

une compilation pourra être demandée par la commande suivante (adressée au connecteur) :

tab : module = table (50,record begin toto:integer;
jules:boolean end)

La compilation se fait comme si on avait écrit en PASCAL :

program tab;
constant &taille=50;
type &genre=record begin toto:integer;jules:boolean er

Le corps du modèle est constitué d'une liste de déclarations :

- les environnements qui sont utilisés par ce modèle
- les références à des procédures ou fonctions externes
- les déclarations usuelles d'étiquettes, de constantes, de type, de variables, de procédures et de fonctions
- les procédures ou fonctions externes qui sont les seuls objets pouvant être référencés de l'extérieur.

Ces objets définis au niveau du modèle sont dits propres au module (créé à partir de ce modèle par compilation). Ils sont accessibles depuis toutes les procédures ou fonctions de ce module. Si des variables propres au module ont besoin d'être initialisées par programme elles peuvent l'être par l'une des procédures externes.

ii) Environnement

Un environnement est un texte source possédant un nom connu du bibliothécaire. Ce texte contient des déclarations écrites dans le langage SESAME. La notion d'environnement a été introduite pour pouvoir "mettre en facteur" des déclarations entre plusieurs modèles. En effet plusieurs modèles peuvent avoir besoin des mêmes déclarations. Par exemple si un modèle présente une interface avec des paramètres de type construit, il est nécessaire que ces types soient aussi déclarés dans les modèles "appelants".Plutôt que d'écri

ces déclarations dans chacun des modèles (des erreurs sont toujours possibles), il est préférable (meilleure cohérence) de les écrire à un seul endroit et d'effectuer leurs recopies dans chacun des modèles de façon automatique. Ces déclarations portent sur des constantes et des types et éventuellement des procédures ou des fonctions ne faisant pas appel à des variables globales (procédure de conversion, de comparaison par exemple).
Nous avons choisi de n'effectuer aucune compilation préalable de tels textes. C'est lors de la compilation d'un modèle et à la rencontre d'une déclaration commençant par *env* que les textes des environnements désignés sont insérés dans le texte du modèle. Ainsi un modèle contenant :

env <*environ1*> , <*environ2*> ;

est rendu équivalent à un modèle identique où l'on aurait supprimé cette ligne et rajouté toutes les déclarations figurant dans <*environ1*> et <*environ2*> au niveau de déclaration des objets propres au modèle (niveau 0).

1.2.6. Procédure et fonctions externes

i) Déclaration

Parmi les objets d'un modèle, les seuls qui peuvent être référencés à l'intérieur d'autres modèles sont des procédures ou des fonctions déclarées au niveau 0 du modèle et précédées par le mot de base *ext*. Ces procédures ou fonctions sont dites externes.

ii) Référence

La compilation d'un modèle fournit un module avec un certain nom. Ce nom est donné dans la commande entrainant la création du module (langage de connexion). Tous les modules unis d'un même modèle conservent les identificateurs du modèle pour leurs procédures ou fonctions externes. Pour en référencer une dans un module particulier il est nécessaire de qualifier cet identificateur par le nom du module. On obtient ainsi un nom de la forme :

<nom du module>::<nom de la procédure ou fonction du modèle>

L'appel d'une telle procédure ou fonction pourra se faire avec ce nom.
Par exemple :

x:= catalogue::rechercher(y)

Si lors de l'écriture du modèle, ce nom n'est pas connu ou si l'on désire reporter la liaison à l'étape de connexion, un identificateur local peut être utilisé.
Par exemple :

x:= trouver(y)

Pour assurer la compilation de ces appels et vérifier la validité des liaisons qui seront établies, toutes les références externes doivent être déclarées sous forme d'un en-tête de procédure ou fonction. Chacune de ces déclarations est introduite par les mots de base *ref* (si le nom exact est connu) ou *dummy* (si un nom local est utilisé).
Par exemple :

ref fonction catalogue::rechercher(clé:integer)

ou *dummy fonction trouver(clef:integer)*

Passage de paramètres

Pour les fonctions ou procédures externes le passage de paramètres "par adresse" n'est pas autorisé. Le compilateur vérifie donc que l'en-tête des procédures ou fonctions introduites par *ext*, *ref* ou *dummy* ne contient pas de paramètres formels introduits par *var*. Pour permettre de rendre des résultats autrement que comme valeur d'une fonction (résultat qui pour des raisons d'implémentation doit tenir sur un mot) il est possible de passer des paramètres "par résultat".
Par exemple :

```
procédure choisir (result param1:record...end);
   begin
   ......% utilisation de param1 %...
   end
```

est équivalent à :

```
procédure choisir(var param1:record...end);
var localparam1:record...end
begin
... % utilisation de  localparam1 à la place de param1 %..
param1:=localparam1;
end;
```

Cette dernière écriture étant refusée pour une procédure externe.

1.3. Eléments du langage pour la multiprogrammation

1.3.1. Processus

Les processus ne sont pas déclarés : ils ne sont pas désignés par un identificateur. En fait les processus qu'offre le moniteur d'exécution sont utilisés par le compilateur pour servir de support à des activités parallèles qui s'expriment sous forme d'instructions.

Chaque activité est représentée par une instruction (composée, par exemple contenant des appels à des procédures ou fonctions d'autres modules). Lorsque plusieurs activités doivent* être conduites en parallèle, la liste des instructions les définissant est délimitée par *par begin* et *par end*.

* Pour les besoins du système (par exemple contrôle de divers processus externes). Comme les mécanismes mis en oeuvre à l'exécution sont importants, le parallélisme ainsi exprimé, ne doit être employé que lorsqu'il est nécessaire. Il ne s'agit pas ici d'instructions colatérales (ALGOL68) pouvant être exécutées dans un ordre indifférent.

Cette construction *par begin* <*suite d'instructions*> *par end* est elle-même une instruction qui peut être employée comme telle à l'intérieur d'un modèle. Cette instruction est considérée comme terminée lorsque toutes les instructions de la suite qu'elle contient sont elles-mêmes achevées.

1.3.2. Synchronisation

La synchronisation d'un processus se fait par l'intermédiaire des fonctions ou procédures qu'il est amené à activer. C'est à l'intérieur de moniteurs de synchronisation (qui sont des modules particuliers : partagés entre plusieurs processus) que ces contraintes s'expriment. L'en-tête du modèle s'écrit alors *monitor pattern...*

Les procédures ou fonctions des modules créés à partir d'un tel modèle sont en exclusion mutuelle : si une procédure ou fonction externe est en cours d'activation, un appel (déclenché par un autre processus) à la même(ou une autre procédure ou fonction du module sera retardée jusqu'à ce que le premier processus ait terminé l'activation ou bien s'il est obligé d'attendre (réalise un wait. Voir ci-dessous).

A l'intérieur d'un moniteur il est possible de déclarer des variables du type *condition*. Il y a seulement deux opérations qui sont possibles sur des variables de ce type. Elles s'expriment sous la forme des procédures standards :

wait(c,p) Cette fonction met en attente le processus "sur la condition *c*". *p* est une indication sur la priorité que le processus aura pour être "réveillé" (voir *signal*).

signal(c) Si aucun processus n'est en attente sur la condition *c*, cette procédure n'a aucun effet. Dans le cas contraire le processus qui a la plus forte priorité est réactivé. Le choix du processus ainsi "réveillé" dépend de l'implémentation (ce peut être, par exemple, parmi ceux qui

ont la plus forte valeur pour *p* , celui qui attend depuis le plus longtemps). Le processus qui a effectué *signal* est mis en attente.

Une description de ces deux fonctions et de celles qui réalisent l'exclusion mutuelle existe au chapitre 5.

Nous donnons ici l'exemple d'un tel moniteur. Il s'agit d'une boite aux lettres où peuvent être déposés des éléments du type *&elem*, la capacité de la boite étant de *&max*.

```
monitor pattern boite_lettres (&max:integer,&elem:type);

 var    n:integer:=0;
        in,out:integer:=1;
        non_vide,non_plein:condition:
        B:array[1..&max]of &elem;
 ext procedure deposer (e:&elem);
     begin if n=&max then wait (non_plein,0);
           n:=n+1;
           B[in]:=e;
           in:=(in mod &max)+1;
           signal(non_vide);
     end;
 ext procedure prendre (result e:&elem);
     begin if n=0 then wait (non_vide,0);
           n:=n-1;
           e:=B[out];
           out:=(out mod &max)+1;
           signal (non_plein);
     end;
 end.
```

2 - LE LANGAGE DE CONNEXION

2.1 - Introduction

Le langage décrit dans le paragraphe précédent permet l'écriture de modèles de modules. Par contre, la construction de systèmes par assemblage progressif de modules est exprimé à l'aide du langage de connexion distinct du précédent.

Ce langage permet à l'utilisateur de SESAME de définir :

- la liste des modules qu'il veut réunir pour former un système,
- la liste des connexions qui doivent être établies entre ces modules.

Nous appelons programme de connexion un texte source écrit dans ce langage. Les deux listes précédentes y sont représentées sous forme de déclarations et d'instructions. Son traitement par l'interpréteur du langage de connexion (ou connecteur) fournit un texte objet que nous appelerons système (ou sous-système). L'obtention de ce texte correspond à une étape intermédiaire ou finale dans la construction du système tout entier (celui que l'on se propose d'exécuter sur le T1600). Ce sont donc les programmes de connexion qui commandent la production, ils constituent une description statique de la structure d'un système.

2.2 - Définition d'un système

Extérieurement, sous forme de texte objet, un système et un module différent très peu. Les modules sont les composants de base de tout système. Après rassemblement de quelques modules dans un système il est nécessaire de définir les procédures ou fonctions externes de ces modules que l'on désire faire "ressortir" à l'extérieur du système. D'autre part il est possible que toutes les références ne peuvent être résolues soit parce que le module où figure la procédure ou fonction référencée ne fait

pas partie de ce système, soit parce que le nom conventionnel(*dummy*) utilisé n'a pas encore été remplacé par un nom réel. Tout comme un module, un système possède donc un certain nombre de procédures ou fonctions externes et des références (*ref* ou *dummy*) à d'autres modules ou systèmes. Ce système pourra entrer dans la composition d'un autre système. Comme nous le verrons, l'interface d'un système est exprimé dans le programme de connexion correspondant.

A la limite, on peut dire qu'un module est un système élémentaire (qui serait le résultat de l'interprétation d'un programme de connexion ne contenant que la demande de création de ce seul module).

2.3 - Rôle du connecteur

Le rôle du connecteur se ramène à deux fonctions principales :

- rassembler, en un seul texte, tous les textes constituants, déclarés dans le programme de connexion qu'il interprète. Si certains textes objets n'existent pas (recherche négative effectuée par le bibliothécaire), il se charge de les créer en interprétant les programmes de connexion correspondant à des sous-systèmes constituants et/ou en demandant au compilateur de créer les modules nécessaires.
- effectuer sur ce texte les connexions, c'est-à-dire effectuer une édition de liens (résolution des références externes pouvant être résolues dans cette étape après translation des adresses relativement au début de tout le texte).Il modifie en plus le dictionnaire des symboles externes et le dictionnaire des références fictives (qu'il a construit pour l'édition de liens)pour n'y faire apparaître que les symboles et les références qui sont demandés à être reportés au niveau du système. Il signale (pour éviter erreurs ou oublis) les références non encore résolues (*ref*) et celles qui ne le seront jamais (*dummy* restantes non"extériorisées"). Ces dernières sont néanmoins

reliées à une procédure *masse* qui permettra de détecter une erreur au cours de l'exécution.

Pour chacune des connexions établies, il y a vérification de l'interface. La liste des paramètres formels (et le type de la fonction) d'une définition d'externe et d'une référence externe est codée dans chaque texte objet (c'est la même procédure du compilateur qui est appliquée à chaque fois). Si les deux codes sont différents, la connexion est refusée*. Dans le cas contraire, nous ne pouvons affirmer que la connexion est toujours correcte. En effet, il est des cas où le codage est très difficile à réaliser. Nous avons choisi de représenter chaque type simple par un entier différent et les constructions (*array, record*) par d'autres entiers. Lorsque le type est, par exemple, défini de manière récursive, le codage choisi pour ce type est 0 (indication de "difficile"). La vérification n'est donc pas totale, mais elle permet de détecter la plupart des connexions incorrectes "par étourderie".

2.4 - Définition du langage de connexion

2.4.1 - Programme de connexion

Un programme de connexion est composé

- d'une suite de déclarations énumérant les constituants (modules et sous-systèmes) du système à construire,
- d'une suite de déclarations définissant les procédures et fonctions externes que le système fournit, les noms conventionnels des références à des procédures ou fonctions externes (d'autres systèmes) qui resteront à résoudre après construction de ce système,

---

* Nous n'avons pas cherché à établir une "équivalence de types". Les deux définitions doivent avoir été construites de la même manière pour que l'on soit assuré de leur compatibilité.

- de la définition éventuelle du "point d'activation". Elle porte sur la procédure principale c'est-à-dire celle qui doit être activée après chargement du système sur la machine hôte (T1600),
- d'une suite d'instructions dites de connexion permettant de résoudre les références fictives (remplacement par une référence réelle).

Un programme de connexion reçoit un nom. C'est sous ce nom qu'il sera conservé par le bibliothécaire. Ce nom permet aussi de retrouver le système (texte objet) correspondant.

<programme de connexion> := *system* *<nom du programme>*;
<déclaration des composants>
{ *<déclaration d'interface>* }
{ *<connexions>* }
{ *<point d'activation>* }
end.

<nom du programme> est un identificateur. Cet identificateur pourra figurer dans un modèle (référence externe du genre *ref*), dans un autre programe de connexion (où ce système est défini comme composant d'un nouveau système).

2.4.2 - Déclaration de composants

Chaque composant (module ou sous-système) est défini par une déclaration donnant en particulier le nom du texte source (modèle ou programme de connexion) à partir duquel ce composant peut être créé ou bien encore retrouvé (par le bibliothécaire) s'il existe déjà.

<déclaration des composants> ::= build from <liste des composants> ;
<liste des composants> ::= <composant> | <composant> ; <liste des composants> ;
<composant> ::= <nom de module> : module = <nom du modèle>
{(<liste de valeurs de métavariables>)}
| <nom de sous-système> : system

<nom de module> et *<nom de sous-système>* sont des identificateurs.Ainsi, un nom est donné à chaque composant mais seulement pendant la phase d'interprétation du programme de connexion (pour effectuer en particulier l'édition de liens que nous verrons en 2.4.4). Une fois la connexion réalisée, les composants ne sont plus "visibles de l'extérieur". *<nom de sous-système>* doit être le même identificateur que celui nommant le programme de connexion définissant ce sous-système.

Exemple -

```
system lister_table_entiers ;
   build from
     traitement : module = édition ;
     table_entiers : module = table (1000, integer) ;
     gestion_périphérique : system;
.......
```

définit un système constitué au moins de trois composants :

- un module compilé à partir du modèle *édition* qui ne comporte aucune métavariable,
- un module compilé à partir du modèle *table* avec pour valeurs affectées aux deux métavariables que celui-ci contient : l'entier *1000* et le type *integer*,
- un sous-système résultat d'une interprétation du programme de connexion *gestion-périphérique*.

2.4.3 - Déclaration d'interface

Cette partie définit, parmi les procédures et fonctions externes *(ext)* des composants, celles qui constitueront l'interface du système ainsi que les références fictives *(dummy)* dont la résolution est reportée à une étape ultérieure.

<déclaration d'interface> ::= interface <liste de morceaux d'interface> ;
<liste de morceaux d'interface> ::= <morceau d'interface>
| <morceau d'interface> ;
<liste de morceaux d'interface >

<morceau d'interface> ::= with <nom du composant> do <liste de prises> end
<liste de prises> ::= <prise> | <prise> ; <liste de prises>
<prise> ::= ext <identificateur> on <nom de procédure ou fonction externe>
| dummy <identificateur> on <nom conventionnel>

Par exemple :

```
system gestion_nombres ;
   build from
      table_entiers : module = table (1000,integer) ;
   .......
   interface
      with table_entiers do
      ext mettre on ajouter ;
      dummy erreur 1 on overflow ;
      end ;
   .......
```

indique que la procédure ou fonction qui s'appelait *table_entiers.ajouter* est reportée au niveau du système sous le nom de *mettre*. L'identificateur *gestion_nombres.mettre* pourra être utilisé désormais comme référence à cette procédure ou fonction. D'autre part, le nom conventionnel *overflow* qui figurait dans le module *table_entiers* ne sera pas lié à une procédure

ou fonction externe, à l'intérieur de ce programme de connexion. La liaison sera établie plus tard, mais en considérant le système tout entier et avec un nom conventionnel qui est désormais *erreur1*.

Les identificateurs des procédures ou fonctions externes figurant au niveau des composants sont perdus : il ne sera plus possible d'y faire référence par la suite. Seules les procédures et fonctions ayant reçu un nouvel identificateur au niveau du système pourront être appelées à l'extérieur de ce système. Le même genre de remarque s'applique aux identificateurs de références fictives.

2.4.4 - Instructions de connexion

Une instruction définit la procédure ou fonction externe d'un module (ou d'un sous-système) que l'on veut faire correspondre à une procédure ou fonction fictive (déclarée sous un nom conventionnel).

<connexion> ::= connect <liste de morceaux de connexion> ;
<liste de morceaux de connexion> ::= <morceau de connexion>
| <morceau de connexion> ;
<liste de morceaux de connexion>
<morceau de connexion> ::= with <nom de composant> do
<liaisons>
end
<liaisons> ::= <instruction de connexion>
| <instruction de connexion> ; <liaisons>
<instruction de connexion> ::= <nom conventionnel> :=
<nom de composant> . <nom de procédure
ou fonction>

L'identificateur suivant le *with* doit figurer dans la liste des composants *(build from)* du même programme de connexion. En partie droite d'une instruction de connexion doit figurer l'identificateur d'une procédure ou fonction externe déclarée au niveau d'un composant du système que l'on construit.

Par exemple :

```
.......
connect
   with gestion_nombres do
                erreur1 := opérateur.signaler ;
        end ;
.......
```

indique que le nom conventionnel *erreur1* déclarée au niveau de *gestion_nombres* fait référence à la procédure ou fonction externe *signaler* du comsant *opérateur*.

Normalement, tout nom conventionnel qui n'a pas été redéfini dans la partie *interface* doit figurer en partie gauche d'une instruction. S'il n'en est pas ainsi, le fait est signalé à l'utilisateur.

2.4.5 - Définition du point d'activation

Dans le programme de connexion définissant le système complet que l'on désire produire (pour exécution sur le T1600), il est nécessaire d'indiquer la procédure externe(d'un de ses composants) qui devra être activée après le chargement. Cette déclaration annule celles qui pou - vaient exister au niveau des composants.

<point d'activation >::= start <nom de composant> . <nom de procédure>

Dans un tel programme de connexion, la partie *interface* ne doit pas contenir de redéfinitions de références fictives (elles ne seraient jamais résolues). Par contre, des références réelles figurant dans les composants et qui seraient encore non résolues, peuvent subsister. De même, des définitions d'externe peuvent être proposées dans *interface*.

En effet, le système produit pourra encore subir une édition de liens (grâce à l'éditeur de liens standard de T1600) avec des modules écrits directement (en dehors de SESAME).

2.4.6 - Extensions envisagées

Un programme de connexion est un texte source conservé par le bibliothécaire. Il est possible d'envisager la réutilisation d'un même programme pour la construction de systèmes différents. Cependant tel quel, ses possibilités de réutilisation sont discutables car un programme de connexion ne comporte aucun paramètre. Lorsque nous aurons suffisamment de programmes et s'il apparaît que le manque de paramétrage est un obstacle à une réutilisation éventuelle, nous pourrons introduire des procédures de connexion qui différeraient de la notion de programme de connexion par la présence possible de paramètres pour définir les caractéristiques d'un système particulier à produire. Pour un système, une procédure de connexion est équivalente à un modèle pour un module.

Les systèmes comportent souvent des familles de composants remplissant des fonctions voisines (gestion de périphérique par exemple). Il serait donc commode de désigner par un seul identificateur tous les composants d'une telle famille et de référencer l'un d'eux par indexation. On pourrait introduire pour cela la notion de "tableau de modules".

Exemple de déclaration de composants :

```
tab : array [2..4] of module =
      2 : table (50,integer) ,
      3,4 : table (100, real) ;
```

Ces composants sont référencés ensuite par *tab [2], tab [3], tab [4]*

```
with tab [3] do
        overflow := operateur.signaler
```

Pour abréger, dans certains cas, l'écriture des connexions avec des éléments de tableaux de modules, on introduirait une instruction itérative.

Exemple :

```
with each tab do
        overflow := operateur.signaler
```

Sans aller si loin, la syntaxe du langage de connexion telle que nous l'avons préservée est certainement perfectible. Il s'agit ici de la définition d'une première version. Nous espérons que son utilisation permettra d'un améliorer les moyens d'expression .

3. LE BIBLIOTHECAIRE

Le bibliothécaire est la partie de SESAME chargée de la gestion (catalogage, conservation, recherche, édition) de tous les textes élaborés sur la machine de production pour servir à la construction de systèmes d'exploitation. L'accès et la manipulation de ces textes est assurés par le bibliothécaire par l'intermédiaire de commandes qui lui sont adressées soit directement par les membres de l'équipe de production, soit indirectement lors de l'utilisation des autres outils de SESAME (interprétation d'un programme de connexion).

3.1. Objets gérés par le bibliothécaire

Les objets gérés par le bibliothécaire sont des textes, il s'agit:

- des modèles de modules (textes sources écrits dans le langage d'écriture),
- des environnements (textes sources écrits dans le langage d'écriture),
- des programmes de connexion (textes sources écrits dans le langage de connexion),
- des modules (textes objets résultant de la compilation d'un modèle de module suivant certaines valeurs affectées aux métavariables),
- des systèmes (ou sous systèmes) résultant d'une connexion de plusieurs modules entre eux (textes objets).

3.2. Les fonctions assurées par le bibliothécaire

De par les objets qu'il gère, le bibliothécaire supporte et organise l'ensemble des informations utilisées tout au long du développement d'un système produit. Les opérations qui peuvent être faites sur ces informations dépendent de leur nature. Le bibliothécaire en assure le contrôle. D'autre part, des relations diverses existent entre ces informations. Le bibliothécaire permet de les expliciter. Il gère en quelque sorte une base de données spécialisée dans la production de systèmes.

Les fonctions qui peuvent être demandées au bibliothécaire sont les suivantes :

- introduction ou suppression de textes sources (modèles, environnement et programmes de connexion) dans l'espace mémoire géré par le bibliothécaire. Ces différents textes portent un nom qui peut être qualifié (par le nom d'un usager, le nom d'un projet, ...)
- modification de textes sources déjà existants, avec ou sans conservation des versions précédentes du même texte. Ces modifications sont réalisées par l'intermédiaire d'un éditeur de texte.
- recherche de textes sources par différents moyens : par leur nom, par des mots clefs pouvant être attachés à chacun de ces textes, par leurs relations avec d'autres textes (modèles référencés à l'intérieur d'un modèle ou d'un programme de connexion donné, programmes de connexion utilisant un modèle donné, version précédente du même texte, ...)
- introduction ou suppression de textes objets dans l'espace mémoire géré par le bibliothécaire. S'il s'agit d'un module, le nom qui lui est donné est composé du nom du modèle et de la liste des valeurs effectives des métavariables qui ont servi à sa génération. S'il s'agit d'un système le nom est composé à partir de celui du programme de connexion qui est à l'origine de sa création.
- extraction d'informations diverses : impression des textes sources (complète ou partielle : par exemple impression de la seule partie intitulée documentation), de catalogues, de la liste des composants d'un système, des versions différentes d'un même modèle, état actuel du développement du système ...

3.3. Relations entre les objets gérés

Pour que le bibliothécaire puisse effectuer des opérations telles que celles définies ci-dessus, les textes conservés doivent être organisés de façon à faire apparaître les relations qui les unissent.

3.3.1. Relations exprimant des transformations sur les objets

Parmi les relations possibles certaines expriment des transformations ayant permis de passer d'un objet à un autre.

i) Entre le texte objet représentant un module et le texte source d'un modèle il peut exister la relation suivante : "est le résultat d'une compilation de". Ainsi à chaque compilation il sera demandé au bibliothécaire de rajouter, dans une liste propre au modèle compilé, un lien vers le texte objet résultat de cette compilation (représentation de la relation inverse de celle annoncée). D'autre part, nous avons déjà signalé que le texte objet était identifié par le nom de son modèle et de la liste des valeurs des métavariables ayant servi à sa compilation. Ceci permet en particulier de réduire éventuellement le nombre de compilations nécessaires à la construction d'un système. En effet, l'interpréteur des programmes de connexion pourra s'assurer que le module qu'on lui demande de créer ne peut pas être copié simplement à partir d'un texte objet déjà existant, plutôt que de réclamer systématiquement sa compilation.

ii) Entre le texte objet représentant un système et un programme de connexion il peut exister la relation "est le résultat de l'interprétation de". Ici encore c'est le nom du texte objet qui permet de retrouver cette relation.

iii) Entre deux textes sources il peut exister la relation "a été modifé à partir de". Dans ce cas l'objet modifié est de même nature que l'objet initial et on peut donc avoir composition des modifications. La méthode habituelle consiste à définir des versions de travail d'un texte source par modifications successives d'un même texte "de référence" (texte initial ou texte auquel on désire apporter des "améliorations"). Cette méthode conduit à une structure arborescente entre les versions, avec la relation "a été modifié à partir de" (qui devient ici "est une nouvelle version de"). Dans notre cas, cette relation ne devrait être établie que si les modifications faites respectent les spécifications du modèle, chaque version étant une réalisation particulière (correcte ou non) du même objet abstrait.

3.3.2. Relations exprimant la structure d'un système

Les différents composants d'un système sont liés par des relations exprimant la structure de ce système.

i) Entre deux textes objets peut exister la relation "appelle". Cela signifie que dans le premier texte existe au moins une référence à une procédure ou fonction externe du second texte. Une telle relation sera utilisée lors d'une édition de liens pendant la construction du système.

ii) Il est possible de définir une relation analogue à la précédente mais au niveau des textes sources. On pourra dire par exemple que "T1 utilise T2", si dans le modèle T1 il existe une référence à une procédure ou fonction externe d'un module ou d'un système obtenu à partir de T2. Dans certains cas cette relation n'est pas précisée, il s'agit alors de relations correspondant à des références fictives *(dummy)*. Dans le même ordre d'idée, on pourra dire qu'un modèle "utilise" un environnement si une référence à cet environnement figure dans le texte du modèle.

iii) Enfin la donnée d'un programme de connexion introduit une nouvelle relation entre un texte objet (le système créé par ce programme de connexion) et une liste de textes objets (modules ou sous-systèmes composants). Nous pouvons appeler cette relation : "est composé de".

De même au niveau des textes sources, une relation analogue existe:

Un programme de connexion "utilise" certains modèles (et d'autres programmes de connexion). Il est nécessaire que les textes "utilisés" existent avant de pouvoir interpréter le programme de connexion.

3.3.3. Autres relations

D'autres relations peuvent être utiles dans la gestion des différents textes gérés par le bibliothécaire.

i) Les textes peuvent être liés à un utilisateur particulier, à une liste d'utilisateurs, à un projet particulier, à une liste de projets, ... Ces relations expriment une idée de protection et sont représentées au niveau des noms (noms qualifiés).

ii) Les textes peuvent être liés à des mots clefs particuliers (essayant de circonscrire le domaine d'application du modèle, du programme de connexion) ou encore à un pseudonyme (lorsqu'un texte est souvent utilisé il est possible de l'identifier par un nom plus simple que le nom complet).

iii) Diverses autres relations pourraient encore être introduites : relation chronologique (date de création, de dernière utilisation), relation entre un système et les programmes ou jeux de données ayant permis de les tester ... Ces facilités ne sont pas présentes dans le bilbiothécaire que nous avons réalisé. De telles relations devront être retrouvées "à la main" à partir de la documentation figurant dans chacun des textes.

3.4. Cohérence entre objets assurée par le bibliothécaire

Grâce aux relations appartenant aux deux premiers types cités plus haut (3.3.1 et 3.3.2) le bibliothécaire est capable de garantir une certaine cohérence dans les manipulations portant sur les objets qu'il gère.

3.4.1. Cohérence entre texte source et texte objet

Règle : Un texte objet ne peut exister seul : il doit lui correspondre un texte source. En d'autres termes si un texte objet est détruit (par mégarde) il doit être possible de le reconstruire à partir d'un texte source déjà existant (dans le bibliothécaire).

Moyens :

i) Le bibliothécaire ne permet pas la modification directe des textes objets. Il n'accepte des textes objets que s'ils proviennent d'une compilation (module) ou de l'interprétation d'un programme de connexion (système). Le compilateur et

l'interpréteur ne travaillent eux-mêmes que sur des textes sources fournis par le bibliothécaire.

ii) Le bibliothécaire maintient pour chaque texte source la liste des textes objets qui en sont issus (valeurs différentes des métavariables). Ainsi si un texte source est modifié (ou détruit) tous les textes objets qui en sont issus sont détruits.

iii) L'identification d'un texte objet par le nom du texte source dont il est issu et la liste des métavariables le caractérisant permet de :

- retrouver immédiatement le texte source (ou vérifier qu'il existe toujours)
- reconstruire automatiquement le texte objet (ou d'en construire automatiquement une nouvelle version conforme à une modification survenue dans le modèle)
- éviter les copies multiples d'un même texte objet.

3.4.2. Cohérence entre versions

Règle : Les modifications apportées à un texte source ne doivent pas être destructrices;il doit être possible de revenir à une version antérieure facilement s'il s'avère que les modifications introduites sont erronées. Autrement dit une version doit pouvoir être reconstruite à partir de la version dont elle est issue.

Moyens : Toutes les versions de travail intermédiaires issues d'un même texte de référence (par composition de transformations) pourraient être conservées. Mais tous ces textes occuperaient beaucoup de place. Comme la liste des modifications est souvent très courte par rapport au texte à modifier, ce sont plutôt ces listes qui sont conservées par le bibliothécaire. Il est ainsi possible de ne conserver que le texte de certaines versions de travail, les autres pouvant être retrouvées par recomposition des modifications à partir de l'un de ces textes existants (le texte de référence existe toujours).

Pour assurer cette recomposition lorsqu'une version est détruite (destruction du texte correspondant, s'il existe, et destruction de la liste des modifications l'ayant créé), tous ses descendants sont détruits par le bibliothécaire.

3.4.3. Cohérence entre programme de connexion et textes source

Règle : Un système doit pouvoir être reconstruit. Le programme de connexion permettant de créer ce système existe (cohérence de 3.4.1). Par contre, les textes objets le composant (sous systèmes et modules) n'existent plus forcément. Ils doivent cependant pouvoir être reconstruit à leur tour. Pour cela il est nécessaire que les textes sources (programmes de connexion et modèles) qui les avaient créés existent toujours.

Moyen : A chaque texte source (programme de connexion et modèle) le bibliothécaire associe la liste des programmes de connexion qui y font référence. Si un texte source est modifié (ou détruit), les programmes de connexion figurant dans sa liste sont considérés comme modifiés (ce qui entraîne la destruction du système issu de ce programme de connexion). Chacun de ces programmes de connexion peut posséder à son tour une telle liste : tous les systèmes composés à partir de textes objets ainsi détruits sont détruits à leur tour ...

3.5. Réalisation du blibothécaire

3.5.1. Gestion de mémoire et organisation des données

Un des buts du bibliothécaire est de gérer automatiquement et efficacement les moyens de stockage disponibles pour la conservation des textes produits par SESAME. Les solutions apportées par les systèmes de gestion de fichiers (par exemple le SGF de SIRIS 8) ne sont pas totalement satisfaisantes. Dans notre cas il s'agit de conserver un très grand nombre de textes la plupart du temps petits (quelques milliers de caractères), alors que les systèmes de gestion de fichiers sont plutôt destinés à la gestion d'un nombre relativement limité de très grands volumes d'information. Le bibliothécaire s'apparenterait plutôt aux systèmes de gestion de base de données à cause du nombre important de relations qui doivent être représentées. Mais ici encore, il est à remarquer que le type des objets et des relations est connu a priori. La réalisation du bibliothécaire peut donc s'appuyer sur certaines particularités que possèdent ces objets et ces relations (relations exprimées au niveau du nom des objets, relation sous forme de liste linéaire, d'arborescence).

L'espace disque géré par le bibliothécaire est divisé en blocs de taille fixe. Le bloc constitue la plus petite unité allouable. L'espace disque a comme support un fichier dont les accès sont du type "virtuel direct". L'allocation des blocs se fait par l'intermédiaire d'une chaîne de bits (chaque position binaire indiquant si le bloc est libre ou non). Ces blocs servent à la construction de segments. Un segment est caractérisé par un nom et un descripteur. Le nom désigne l'emplacement contenant le descripteur dans une table : la table des segments. Le descripteur de segment comprend :

- la tête de la liste des numéros de blocs de l'espace disque cons
- la longueur occupée par l'information dans le dernier bloc du se

Les emplacements non occupés dans la table des segments sont organisés dans une liste libre.

3.5.2. Gestion des noms

Le nom (identificateur) d'un objet géré par le bibliothécaire permet de retrouver l'index du descripteur de cet objet.

Le nom d'un texte objet ou d'une version de travail (texte source) est composé à partir du nom du texte source dont il est issu (pour une version de travail : le texte de référence). La recherche du descripteur d'un texte objet ou d'une version de travail passera par la recherche du descripteur du modèle (de référence) ou du programme de connexion associés.

Donc seuls les noms des textes de référence figurent dans un catalogue spécifique. Pour l'instant la recherche dans ce catalogue se fait de manière séquentielle. Lorsque le nombre d'utilisateurs à servir et le nombre d'objets à conserver le justifieront,une organisation en plusieurs niveaux et une recherche beaucoup plus rapide (adressage dispersé) seront fournies.

3.5.3. Descripteur des textes de référence

Le descripteur d'un texte de référence (modèle, environnement ou programme de connexion) est composé :

- de la tête de la liste des programmes de connexion où figure au moins une déclaration faisant référence à un module issu de ce modèle ou d'un système issu de ce programme de connexion (S'il s'agit d'un environnement ce sera la liste des modèles l'utilisant). Chaque élément de cette liste contient l'index du descripteur de chacun de ces objets.

- de la tête de la liste des objets composants. Cette liste n'est utilisée que pour les procédures de connexion : elle contient l'index du descripteur de chacun des modèles ou programmes de connexion (textes de référence) utilisés pour produire les modules et sous-systèmes qui y sont définis.

- de la tête de la liste des textes objets issus. Pour un environnement cette liste est vide. Pour un programme de connexion, elle contient au plus un élément : le système qu'il permet de produire. Pour un modèle, elle contient les différents modules compilés et conservés (valeurs différentes des métavariables)

- le nom du segment contenant une description de cet objet (documentation)

- un nom de version que nous appelons "contexte" et qui repère un noeud particulier dans l'arbre des versions. Dans les fonctions fournies par le bibliothécaire, la désignation d'une version peut se faire, en effet, par rapport au contexte plutôt que par rapport à la racine (ce qui réduit la longueur de l'identificateur

- un pointeur, vers le "fils aîné" dans l'arbre des versions.

3.5.4. Descripteur des versions de travail

Comme le nombre de relations existant entre les différents objets du bibliothécaire peut être très élevé, il a été décidé que certaines relations exprimées au niveau du texte de référence ne le seraient plus dans les versions de travail qui en seraient issues. Ainsi les deux premières listes reprérées dans le descripteur d'un texte de référence doivent être considérées comme communes à toutes les versions d'un même arbre.

Le descripteur d'une version de travail est composé de :

- la tête de la liste des textes objets issus,
- le nom du segment de documentation,
- le nom du segment contenant (éventuellement) le texte,
- le nom du segment contenant la liste des modifications permettant de le reconstruire à partir de son "père".
- un pointeur vers le "fils aîné" et un pointeur vers le "frère suivant" dans l'arbre des versions.

3.5.5. Descripteurs des textes objets

Le descripteur d'un texte objet est composé de :

- la chaîne de caractères lui servant de nom. Il s'agit du nom du texte de référence suivi du numéro de version (puis de la liste des valeurs effectives des métavariables de compilation il s'agit d'un module).
- du nom du segment contenant le texte binaire,
- d'un index (dans le catalogue des pseudonymes) permettant de retrouver, s'il existe, le pseudonyme associé à cet objet.

3.6. COMMUNICATION DU BIBLIOTHECAIRE AVEC L'EXTERIEUR

Le bibliothécaire peut être appelé directement par un utilisateur de SESAME ou bien par l'intermédiaire de l'interpréteur du langage de connexion, du compilateur ou de l'éditeur de texte. Ce dernier est celui déjà existant sur SIRIS 8. L'utilisation du bibliothécaire se fait suivant le mode conversationnel (SIRIS 8 temps partagé).

Les entrées de textes de référence peuvent se faire à partir d'un lecteur de cartes, de la console ou bien par remplacement du texte de référence actuel pour une des versions de travail qui en est issue . Les entrées de nouvelles versions se font par recopie d'un fichier de SIRIS 8 de type séquentiel (produit par l'éditeur de texte). Enfin les textes objets sont fournis par le compilateur (module) ou le connecteur (système). Dans ce cas le bibliothécaire fournit une fonction de la forme "lire une ligne dans un segment".

Les sorties de textes source (documentation) ou d'états des objets conservés et de leurs relations peuvent se faire sur l'imprimante ou sur le terminal. Un texte source est recopié dans un fichier séquentiel pour pouvoir être traité par l'éditeur de texte. Le compilateur et le connecteur ont à leur dispositi une fonction permettant de lire le contenu des objets du bibliothécaire (sans recopie).

Un texte objet représentant un système peut être produit sur une bande magnétique, un ruban de papier ou des cartes. Le format du binaire est celui qui est accepté par l'éditeur de liens ou le chargeur standard de T1600.

DEUXIEME PARTIE

4. LA STRUCTURE D'EXECUTION

4.1. Introduction

Un système d'exploitation est un programme constitué par un ensemble de modules binaires liés. Une exécution de ce système peut se représenter par un ensemble de processus évoluant simultanément parmi les procédures en fonction des données qu'ils manipulent et des appels qu'ils y rencontrent.

Nous observerons tout d'abord un processus à un moment donné de son évolution pour énumérer les différents types d'objets auxquels il peut accéder. Nous employons une terminologie inspirée de [4].

Ensuite, pour étudier, de façon générale, les chemins d'accès aux différents objets d'un système modulaire, nous nous placerons tout d'abord dans un cas idéalisé. Nous utiliserons la notion de segment (suite d'emplacements contigus) pour regrouper les objets de même durée de vie, de même domaine de protection (partageables par une même communauté d'usagers). Un segment est désigné par son nom. Une liste de noms de segments est un segment lui-même. Nous supposons avoir des possibilités d'adressage variées et étendues (indirection multiple, indexation) pour n'avoir pas à considérer dans cette première étude des problèmes de rapidité d'accès.

Enfin, nous proposerons un schéma d'adressage pour une réalisation sur le T1600. C'est alors que nous aborderons ces problèmes de rapidité.

4.2. L'espace adressable d'une procédure : les objets accessibles

Soit M un module du système S.

Soit P une procédure de M, de niveau de déclaration n. Cette procédure peut être externe (accessible à partir d'autres modules : $n=0$) ou interne ($n\geq 0$).

Il est possible de répartir en plusieurs classes les objets qu'un processus peut désigner au cours de l'exécution de P. C'est ce que nous nous proposons de faire.

Remarque : Par la suite, pour ne pas utiliser partout "fonction ou procédure" (délivre un résultat ou non) nous n'emploierons que le terme de procédure. Lorsque nous parlerons de représentation, nous emploierons le terme d'objet pour désigner aussi bien des données que des procédures (objet-procédure).

4.2.1. Les objets primitifs

Ce sont les objets préexistants à toute exécution et que toute procédure de *S* peut manipuler. Ils sont fournis par la "machine" sur laquelle s'exécute le système. Ils permettent de construire tous les objets de *S*.

Par exemple, s'il s'agit de la "machine SESAME", ce seront :

- des valeurs simples (valeurs entières, booléennes, alphanumériques; valeur indéfinie de pointeur; ensemble vide..),
- des procédures primitives (désignation de variables, opérations entre variables d'un certain type, appel et retour de procédure...).

4.2.2. Les objets extérieurs au module

Ce sont les objets non primitifs (durée de vie inférieure ou égale à *S*) qui peuvent être manipulés par les procédures d'autres modules.

Il s'agit :

- Des objets invariants correspondant à la définition d'un "environnement" partagé entre *M* et d'autres modules. On y trouve des constantes simples ou composées (tableau de caractères représentant un message par exemple, valeur initiale d'une structure..) et éventuellement des procédures qui travaillent sur les seules valeurs des paramètres effectifs sans utiliser de variables globales ou remanentes (procédures de conversion, de comparaison de deux valeurs d'un même type..).
- Des procédures externes d'autres modules (déclarées comme *ref* ou comme *dummy*).
- Des objets repérés par une variable de type pointeur accessible à *P*. Ces objets ont une durée de vie qui leur est propre. Leur manipulation et leur création-destruction sont contrôlées par les procédures primitives qui régissent leur type.

4.2.3. Les objets propres au module

Ce sont les objets qui sont accessibles à *P* mais aussi à toute autre procédure définie dans *M* (et seulement à celle-ci). Ils ont la durée de vie de *M*. Ils sont partagés par tous les processus.

Il s'agit :

- Des données rémanentes définissant l'état du module entre deux activations de procédures externes ou l'état initial avant le premier appel.
- Des objets invariants déclarés dans *M* au niveau 0 : constantes et procédures (externes ou non).

4.2.4. Les objets propres à la procédure

Ce sont les objets qui ne peuvent être désignés qu'à l'intérieur de la seule procédure *P*.

Il s'agit :

- Des données locales (variables déclarées dans *P*, variables de travail) qui ont une durée de vie égale à la durée d'activation de *P*. A chaque activation de *P*, le processus actif trouve un nouvel exemplaire de ces données à une valeur initiale fixée par la compilation.
- Des procédures déclarées dans *P* (niveau n+1).
- Des composants de la procédure *P* : notations d'appels à des procédures primitives, étiquettes, constantes. Ces objets ne sont pas modifiés par l'exécution. Ils peuvent exister en un seul exemplaire et avoir une durée de vie supérieure à l'activation.

4.2.5. Les objets globaux à la procédure

Il s'agit des objets propres aux procédures englobantes (niveau < n). Ils n'existe donc pas d'objets globaux pour les procédures déclarées au niveau 0 (en particulier les procédures externes).*

4.2.6. Les paramètres effectifs

Ce sont des objets qui sont créés et initialisés (paramètres d'entrée) avant l'appel de *P* par la procédure appelante. Ils sont détruits au plus tôt après retour dans l'appelant (paramètre de sortie). Ils ont donc une durée de vie supérieure aux objets propres de *P*.

---

* Pour nous, les objets rémanents n'entrent pas dans la définition de "global".

Si *P* est une procédure externe, le passage des paramètres se fait par valeur ou par résultat. C'est-à-dire que tout paramètre d'entrée est initialisé avec la valeur (au moment de l'appel) d'un objet accessible à l'appelant. Tandis que la valeur d'un paramètre de sortie est recopiée, au moment du retour de *P*, dans une variable accessible à l'appelant.

Par contre, si *P* est une procédure interne, les paramètres peuvent aussi être passés par adresse : le fait de passer un objet en paramètre le rend accessible à l'intérieur de *P*.

4.3. Représentation et mécanismes d'accès

4.3.1. Représentation d'une procédure

Une procédure est composée d'objets invariants qui peuvent donc être conservés bien au delà de leur durée de vie implicite et être partagés pour éviter leur duplication.

Tout objet-procédure est composé :

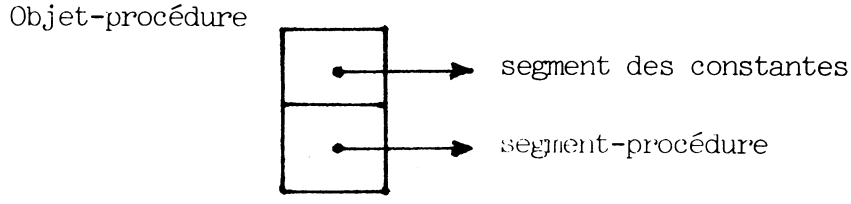
- d'un segment exécutable (segment-procédure),
- d'un segment en "lecture seule" (constantes, étiquettes).

S'il est évident qu'un seul segment-procédure existe pour une procédure déclarée dans le module, il est possible d'envisager son partage entre plusieurs modules (sans qu'elle ait été déclarée dans un environnement commun).

Par exemple : soit deux modules issus du même modèle avec les mêmes valeurs pour les metavariables, le produit de la compilation sera le même. La partie invariante n'a pas besoin d'être dupliquée lors de l'exécution.

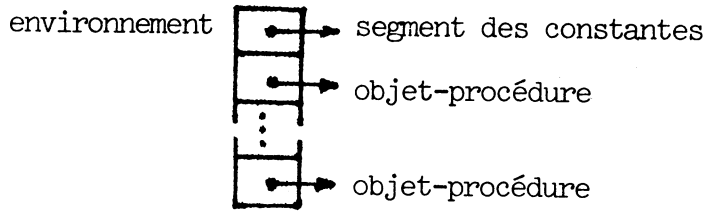
Il est possible d'envisager d'autres cas où un segment-procédure peut être réutilisé (avec éventuellement des segments de constantes différents). Mais la détection est cependant moins aisée.

Nous supposerons donc toute procédure comme potentiellement partageable entre plusieurs activations (récursivité), plusieurs processus (multiprogrammation) et plusieurs modules (optimisation).



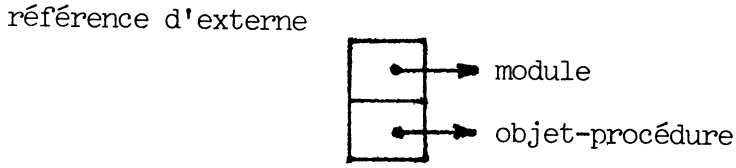
4.3.2. Représentation d'un environnement

Un environnement regroupe des objets invariants mais cette fois partagés entre plusieurs modules de manière explicite.



4.3.3. Représentation des références à des procédures externes

Puisque les segments-procédures peuvent être communs à plusieurs modules, toute référence à des procédures externes doit comporter une identification du module concerné en même temps que l'identification de l'objet-procédure.



C'est un tel élément de liaison qui correspond à une déclaration du type

ref procédure <*nom de module*>.<*nom de procédure*>

ou

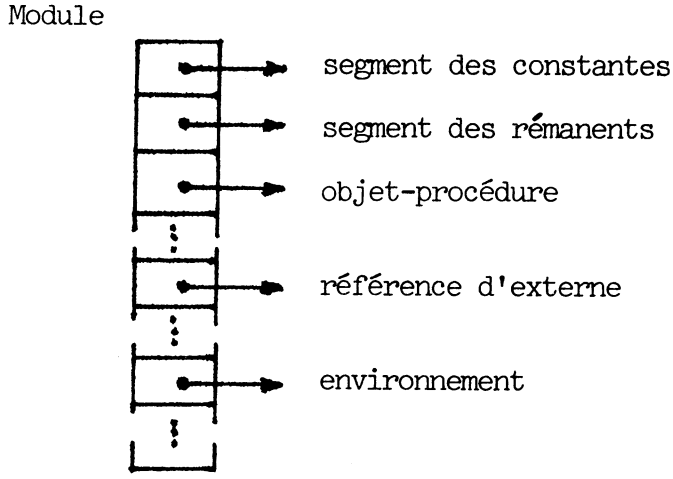
dummy procédure <*non conventionnel*>

Ces éléments sont remplis lors de l'édition de liens.

4.3.4. Représentation d'un module

Un module est une définition statique d'objets comprenant :

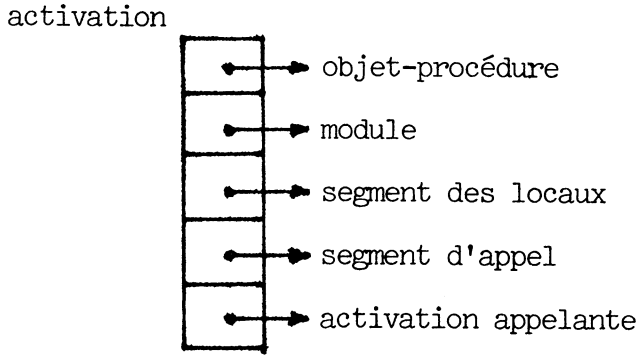
- les données qui y sont déclarées,
- l'ensemble des procédures qu'il fournit aux autres modules,
- l'ensemble des références aux environnements et aux externes d'autres modules qu'il connaît.



4.3.5. Représentation de l'activation d'une procédure

L'activation d'une procédure est représentée par un ensemble dynamique d'objets :

- le module dans lequel cette procédure est appelée,
- la procédure elle-même,
- les objets propres (locaux) associés à cette activation,
- l'identification d'un segment d'appel (contenant par exemple les paramètres effectifs d'une procédure appelée par cette activation),
- l'identification de l'activation appelante.



Les objets accessibles à la procédure le sont à partir de ce descripteur. Par exemples : . les objets remanents sont ceux du module indiqué,

. les paramètres effectifs de cette activation figurent dans le segment d'appel de l'activation appelante,

. les objets globaux figurent dans les locaux ou la zone d'appel d'une activation dans la chaîne des appelants.

A un instant donné les capacités d'adressage (contexte) d'un processus sont résumées dans le descripteur de la dernière activation. Il fait partie du vecteur d'état du processus. Pour un même processus les descripteurs sont gérés dans une pile représentant toutes ses activations en cours. Comme les processus sont liés dans une relation de hiérarchie, il existe, en fait, une arborescence de piles, représentant l'histoire des différents processus.

La figure 4.1 exprime les liaisons existant entre les différentes activations et les différents modules (schématisés par leur descripteur). Le plan inférieur (non représenté mais vers lequel sont dirigés les flèches verticales) représente l'espace des segments. Les modules définissent un plan statique qui structure cet espace en y distinguant les segments composant un même module. (Ici le module *M1* contient une référence à la procédure externe *P2* du module *M2*). Le plan supérieur (dynamique) contient l'arborescence des piles dont deux régions sont représentées. Le vecteur d'état du processus repère la dernière activation (il s'agit de l'exécution de *M2.P2* appelé par *M1.P1*).

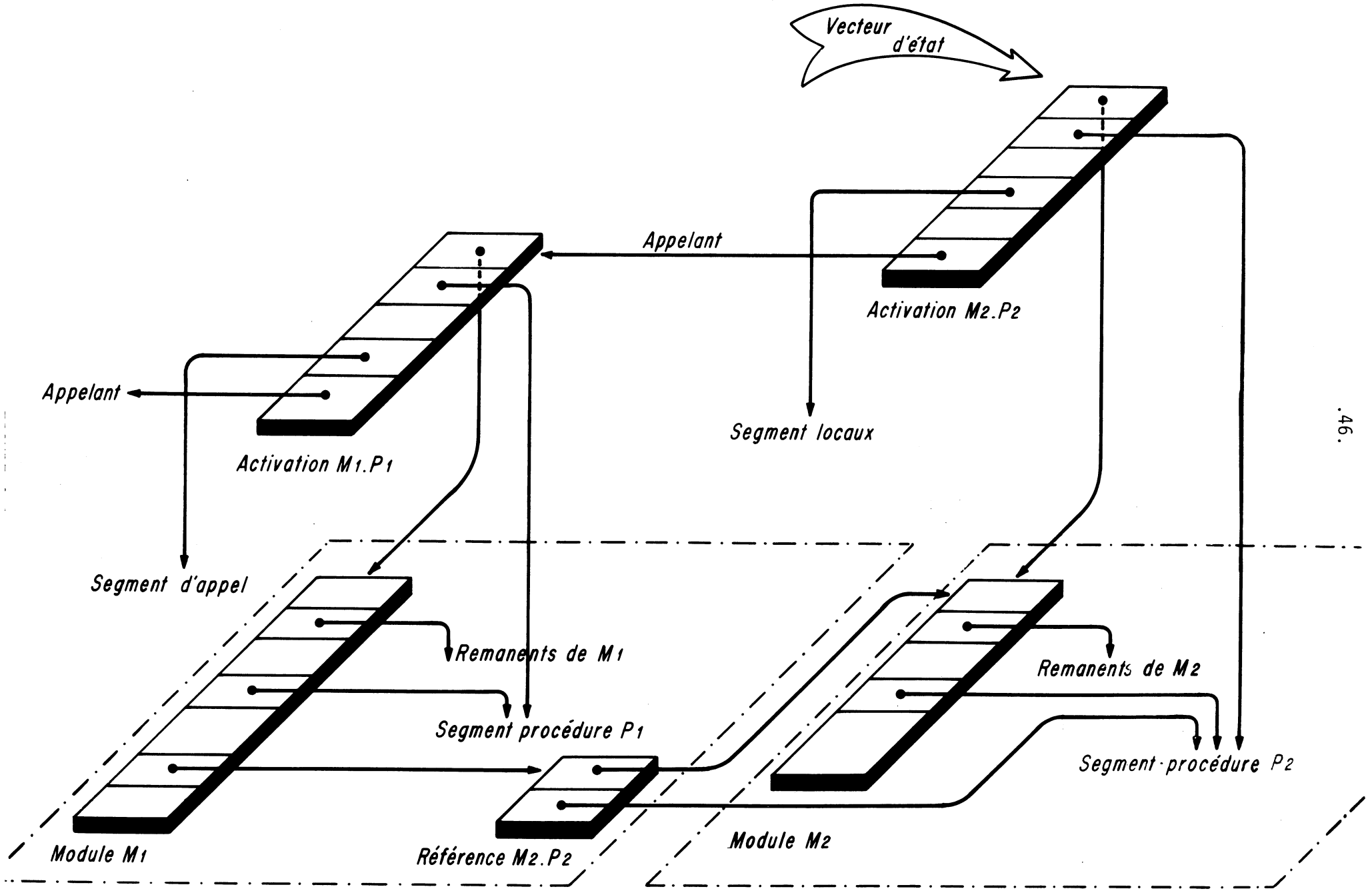
4.3.5.1 Appel de procédure

Lors d'un appel, la procédure appelante crée un segment d'appel dans lequel elle place les valeurs ou les repères des paramètres effectifs. L'appel proprement dit modifie le contexte du processus en créant un nouveau descripteur d'activation.

Si l'appelée est une procédure interne au module courant, l'identification du module reste la même que celle de l'appelant, l'identification de l'objet procédure à exécuter est prise dans le descripteur de ce module. Si l'appelée est une procédure externe d'un autre module, on utilise la référence externe correspondante (figurant dans le module courant) pour initialiser ces deux informations. Un segment est créé pour les locaux.

4.3.5.2 Retour de procédure

Le segment des locaux est détruit. Le contexte du processus est restauré avec le descripteur de l'activation appelante. Après prise des paramètres de retour (résultat) le segment d'appel peut être détruit.


Vecteur d'état
Appelant
Activation M2.P2
Appelant
Activation M1.P1
Segment locaux
Segment d'appel
Remanents de M1
Segment procédure P1
Remanents de M2
Segment procédure P2
Module M1
Référence M2.P2
Module M2

4.4. Exemple de structure d'exécution pour le T1600

4.4.1. Introduction

Comme nous ne voulons pas interpréter les systèmes produits mais les exécuter directement sur la machine hôte, il nous faut utiliser les mécanismes d'adressage du T1600 en adaptant la structure d'exécution étudiée précédemment*.

A un instant donné, les 3 registres de base (*C*, *L* et *W*) et le compteur ordinal *P* permettent au processus actif d'accéder directement à 4 zones distinctes. Par rapport à la notion de segment, ces zones présentent au moins les particularités suivantes :

- elles sont en nombre très limité,
- pour les 3 premières (que nous appellerons zones de données) leur taille est limitée à 256 mots,
- pour la quatrième (nous conserverons le terme : segment procédure), il n'est possible d'effectuer un saut direct que dans une zone de 256 mots centrés autour de l'instruction courante,
- aucun mécanisme de protection n'est prévu au niveau de ces zones.

Il est possible d'accéder à d'autres objets (en une seule instruction) en employant l'adressage indirect. Un relais vers cet objet est alors placé dans une des zones de données. Mais, dans ce cas, les objets doivent être organisés sous forme de tableau d'éléments de même type (adressage indirect post-indexé). Nous aurons un relais par tableau.

4.4.2. Remarques sur la protection

C'est le compilateur qui contrôle l'accessibilité aux objets. Mais, d'une part, son travail pourraît être facilité par l'existence de mécanismes câblés adaptés, et d'autre part, pour des raisons de fiabilité il peut être intéressant d'effectuer un contrôle supplémentaire à l'exécution (redondance).

Le seul mécanisme de protection prévu pour T1600 est inclus dans le mécanisme de "translation dynamique des adresses", qui garantit que toute adresse réelle utilisée par la tâche en cours figure bien entre les deux limites fixées par les registes limites : *SLO* et *SLE*.

Remarquons tout d'abord que ce mécanisme n'étant effectif qu'en mode esclave, aucune protection n'est possible en mode maître.

---

* En annexe 3 figure un résumé sur les particularités du T 1600.

L'utilisation de ce mécanisme passe donc par le regroupement, à l'intérieur de ces limites, de l'ensemble des zones dont on autorise l'accès à un moment donné.

Si l'on veut contrôler, au maximum, l'accès aux objets, la mise à jour de ces limites doit intervenir lors de tout appel ou retour de procédure, c'est-à-dire lorsque le contexte d'exécution change. Cette façon de faire conduit à associer une tâche (un couple *SLO*, *SLE*) à chaque activation de procédure*. Il est évident que le regroupement des zones accessibles à partir d'une procédure ne peut être défini statiquement. Force est donc de recopier à côté des objets créés pour cette activation (locaux...), la valeur des objets existant déjà par ailleurs (segment-procédure, remanents..). Le maintien de la cohérence entre les copies est un problème presque impossible à résoudre étant donné la multiprogrammation.

Même en supposant cette difficulté résolue, nous excluons une telle solution pour la lourdeur de fonctionnement qui serait imposée à tout système produit. Nous nous limiterons donc aux seules protections que peuvent garantir le compilateur et le connecteur.

Nous n'excluons pas pour autant l'utilisation du mécanisme de translation dynamique pour définir une protection portant sur un ensemble explicite de modules ayant une durée de vie propre (sous-système).

4.4.3. Utilisation des registres de base

Le nombre de registres de base étant restreint, il ne sera pas toujours possible de désigner rapidement toute donnée accessible. En pareil cas, il est intéressant

- De regrouper sous un même registre de base les données de durée de vie équivalente mais de nature différente (puisque nous ne pouvons y attacher de protection).
- D'utiliser, au mieux, les registres de bases sans avoir à les initialiser à chaque accès.

---

* C'est une solution de ce genre qui a été retenue à Nancy, pour la réalisation sur T1600 d'un compilateur PASCAL [9]. Mais, dans ce cas, cette solution a été retenue avant tout pour pouvoir exécuter des programmes dont la taille est supérieure à celle de la mémoire principale (par exemple le compilateur). Le mécanisme de translation dynamique est employé pour la gestion de la mémoire hiérarchisée.

On remarquera, tout d'abord, que les instructions portant sur deux opérandes ne font intervenir qu'un seul registre de base. Il est donc possible, avec un seul registre de base disponible, d'effectuer des opérations.

D'autre part, parmi les données les plus fréquemment accédées figureront certainement les variables rémanentes et les variables locales, ensuite les paramètres, enfin les variables globales et les constantes (beaucoup de ces dernières figurent déjà dans le champ "adressage immédiat" des instructions).

Nous affecterons donc la base *C* aux variables rémanentes et la base L aux variables locales. La base W sera utilisée comme base auxiliaire pour les accès où les bases précédentes ne conviennent pas.

4.4.4. Représentation d'un module

Comme un environnement ne contient que des objets invariants, il est possible d'en garder une copie au niveau de chaque module. Les constantes sont gardées dans le segment des constantes du module, les objets procédures rajoutés à la liste de ceux du module. Ceci est fait à la compilation. Au prix d'une certaine perte de place *, l'accès aux objets primitivement définis dans un environnement ne réclamant plus l'établissement d'une base spéciale. Vu la multiplicité des segments de constantes, leur même durée de vie et le peu d'informations qu'ils contiennent en général nous les regroupons dans la zone basée par *C*. Ceci limite le partage possible d'un segment procédure entre plusieurs modules. Mais les cas qui sont ainsi supprimés correspondraient à des optimisations de détection non immédiate.

Les données qui peuvent être structurées sous forme de tableaux seront représentées dans la zone basée par *C* par un relais d'indirection indexé (origine réelle ou virtuelle). Ceci permet de décharger cette zone des données les plus encombrantes.

Figureront en particulier sous forme d'un tableau :

- les références au procédures externes (table de liaison),
- les adresses des différents segments-procédure du module,
- les relais permettant d'effectuer dans ces segments-procédures des sauts en avant supérieurs à 127 mots.

---

* Nous avons introduit la notion d'environnement avant tout pour définir des types et des constantes communes à plusieurs modules (souci de cohérence des définitions et non d'optimisation). Dans la plupart des cas ces informations seront utilisées par le compilateur, très peu subsisteront à l'exécution.

Ce schéma d'adressage permet d'accéder le plus rapidement possible aux objets du module tout en conservant une base *C* de valeur fixe. La zone de 256 mots ainsi basé doit suffire pour contenir tous les objets qui ne sont pas des tableaux ou qui ne sont pas accédés en passant par des tableaux.

Nous dirons que la valeur de *C* identifie le module. La figure 4.2 illustre la représentation ainsi définie.

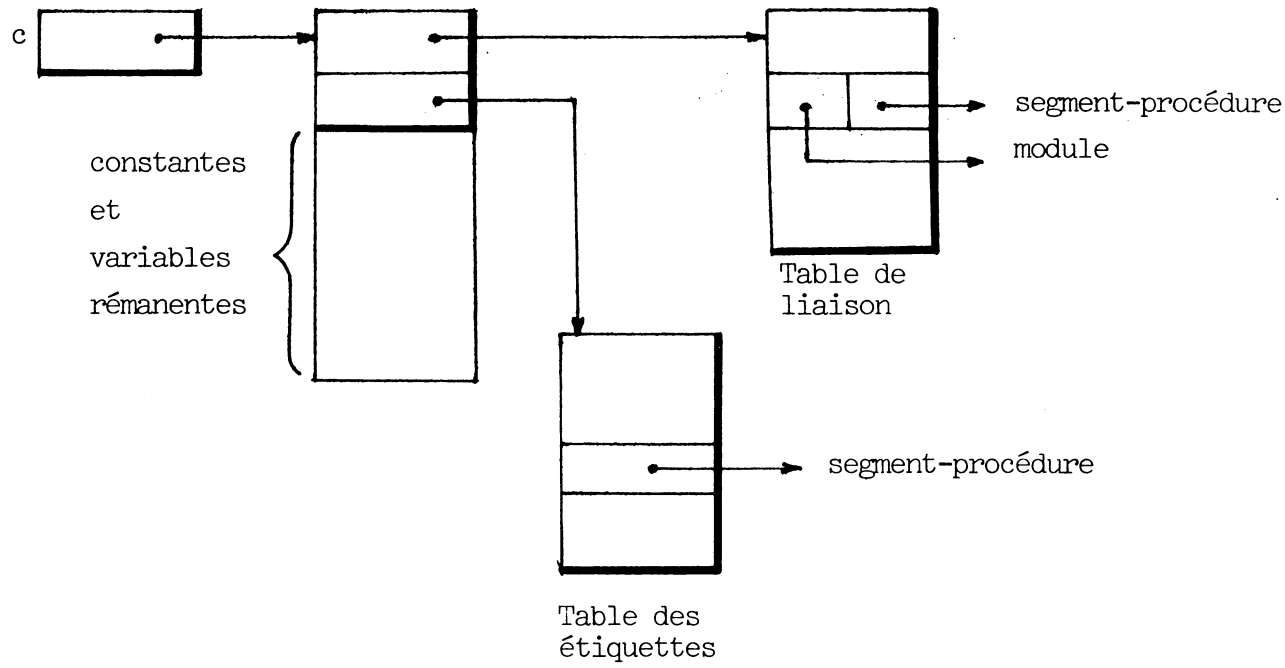


FIGURE 4.2

4.4.5. Représentation d'une activation de procédure

Pour chaque processus, l'activation en cours doit être identifiée dans son vecteur d'état (c'est à partir de lui que le contexte est retrouvé). Les registres *C* et *P* indentifient respectivement le module et le segment-procédure. Le registre *L* permet de baser la dernière région de la pile. C'est à partir de *L* que l'on retrouve l'activation appelante, la zone des locaux, la zone d'appel*.

---

* Nous rappelons que cette zone n'est utilisée que lors d'un appel réalisé par cette activation.

4.4.5.1 Influence de la multiprogrammation

En raison de la multiprogrammation, il existe une arborescence de piles. Les processus évoluent de façon distincte et sont en nombre variable. Un tel ensemble de piles réclamera en général une gestion dynamique et une allocation par blocs.

Une telle gestion oblige donc :

- à formuler une demande d'allocation ou de libération à chaque évolution de la pile (empilement ou enlèvement d'une région),
- à connaître la taille de chaque région (pour l'allocation),
- à utiliser deux registres de base si l'on désire accéder en même temps à deux régions différentes,
- à structurer les régions en tenant compte de la discontinuité toujours possible entre les régions (en particulier l'adresse de retour et l'identification du module, sauvegardées lors d'un appel, doivent figurer à des déplacements fixes pour pouvoir être retrouvées lors du retour).

4.4.5.2 L'organisation de la pile

Nous tenons compte des remarques précédentes dans l'organisation de la pile représentée par les figures 4.3 et 4.4. La première caractérise l'activation courante en dehors d'un appel (à une autre procédure), la seconde lors de la réalisation d'un tel appel ou juste après retour de l'appelée. Un appel équivalent est réalisé lors de la création d'un nouveau processus.

Nous illustrons son fonctionnement par les mécanismes d'appel et de retour qui sont décrits ci-dessous.

i) Appel de procédure

Soit *C1* et *P1* les valeurs identifiant respectivement le module et le segment-procédure appelés. S'il s'agit d'un autre module que le module courant, ces valeurs se trouvent dans la table de liaison du module courant. S'il s'agit du même module, *C1* est la valeur actuelle de la base *C* et *P1* est trouvé dans la table des procédures composant le module.

Un tel appel exécute les étapes suivantes (voir figure 4.4) :
Dans l'appelant,

- une zone d'appel est créée et chargée avec les paramètres effectifs,
- l'identification du module courant (*C*) et l'adresse de retour (*P*) sont sauvegardées (respectivement en *2, L* et *1, L*)

- le registre Y est chargé avec *C1**, enfin un branchement est effectué à *P1* (*P* est donc chargé avec *P1*),

Dans l'appelé,

- une nouvelle région est créée (*W* contient la base de cette région),
- la valeur de *L* (toujours celle de l'appelant) y est sauvegardée (en *0*,*W*),
- *C* est chargé avec le contenu de *Y*, *L* avec celui de *W*,

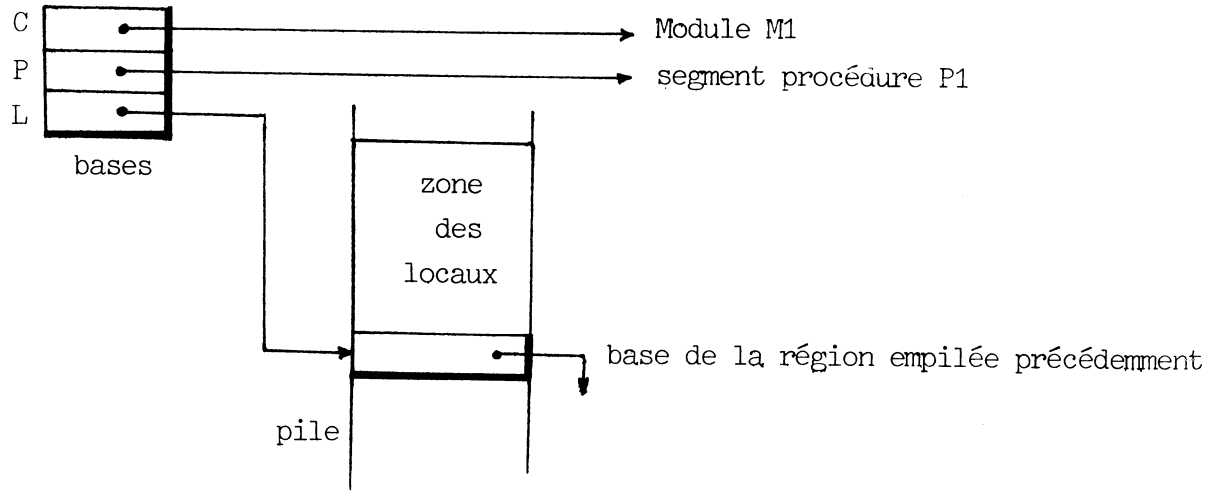


Figure 4.3

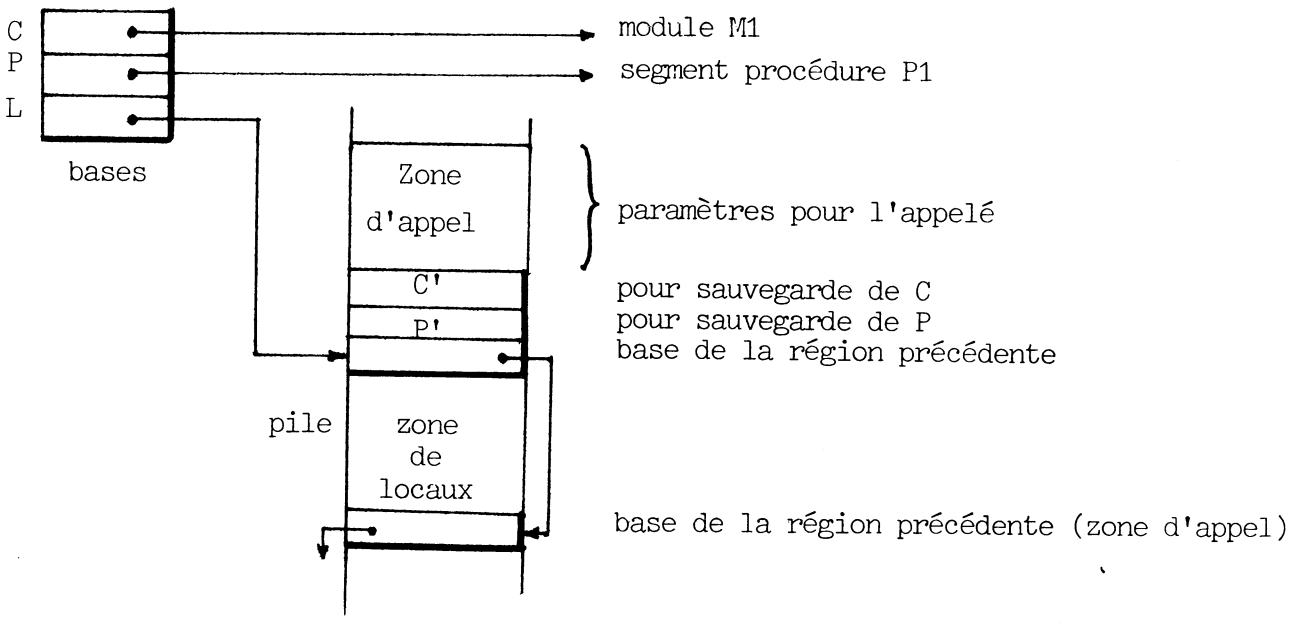


Figure 4.4

* *C* ne peut être modifié tout de suite car il sert à baser l'emplacement contenant *P1* .

ii) Retour de procédure
Les opérations effectuées à la fin de tout segment-procédure sont :

- restauration de L avec son ancienne valeur (en *0,L*),
- libération de la région courante,
- restauration de C avec son ancienne valeur (en *2,L*),
- branchement à l'adresse de retour dans l'appelant (en *1,L*),

4.4.5.3 Améliorations possibles

Nous nous plaçons dans le cas où la gestion dynamique des piles est nécessaire. Pour réduire le nombre de demandes d'allocation (et de libération) et pouvoir utiliser le même registre de base *L* il est intéressant de regrouper des régions voisines dans la pile.

Dans ce qui suit nous présentons 3 améliorations différentes qui vont dans ce sens. Elles peuvent, dans une certaine mesure, se combiner.

i) On peut envisager de réserver, en même temps que la région d'appel, la région des locaux de l'appelé. Pour cela le compilateur doit être capable de déterminer la taille de cette dernière région.

Ceci est impossible si l'appelé est une procédure externe d'un autre module (compilation séparée). Il en est de même pour une procédure externe appelée dans le même module puisque la compilation de la procédure est unique et ne dépend pas de l'endroit où elle est appelée : il faut continuer à réserver les deux régions de façon séparée et utiliser deux bases distinctes.

La compilation doit distinguer : procédure externe (préfixée par *ext*) et procédure interne lors de la traduction :

- d'un appel (réservation des locaux de l'appelé si l'appelé est interne),
- d'une entrée de procédure (réservation des locaux si la procédure est déclarée comme *ext*,
- de la procédure elle-même (déplacements et base).

La structure qui en résulte est décrite par la figure 4.5.

Dans l'exemple de cette figure *P1* est une procédure interne, *P2* une procédure externe.

Cette solution est donc intéressante pour les procédures internes.

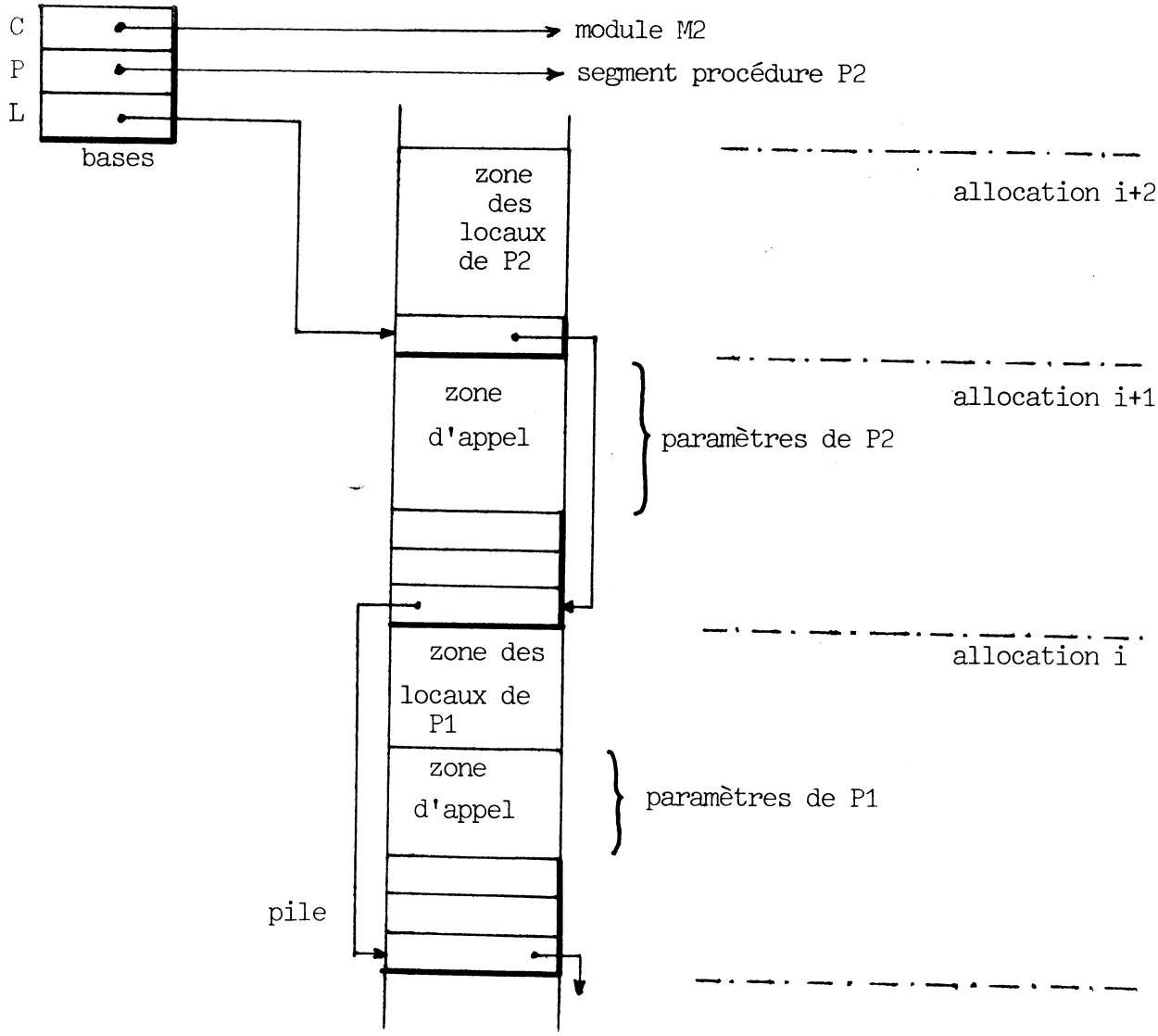


Figure 4.5

ii) En entrée de procédure il est possible de réserver, en même temps que les locaux, une zone destinée à recevoir une copie de la région précédente (zone d'appel). Ainsi la procédure pourra accéder à ses paramètres et fournir un résultat directement (par la base L) sans avoir à initialiser, pour chaque accès, le registre W*.

La figure 4.6 représente le haut de la pile au moment où la copie est réalisée.

---

* Celui-ci sert de base à toute sorte d'accès et ne peut être réservé à un usage bien particulier.

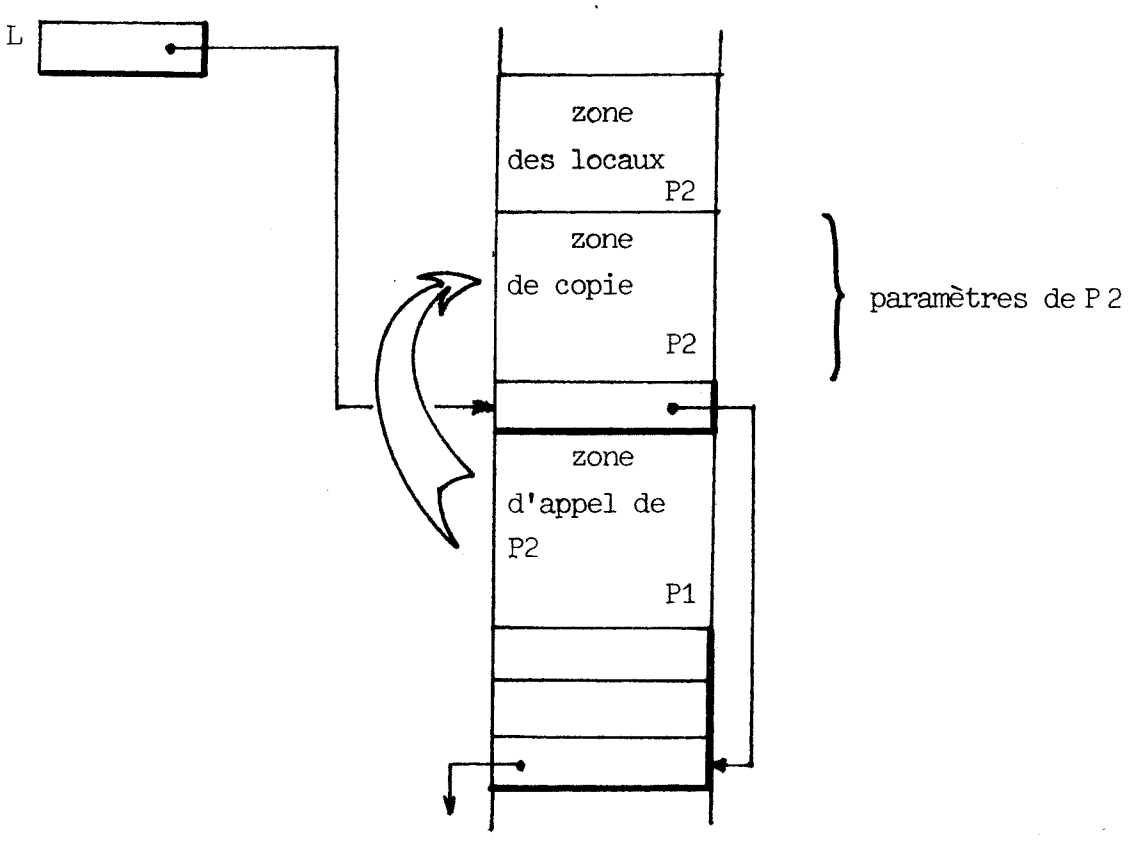


Figure 4,6

Cette solution s'applique à toute procédure (externe ou interne). Le nombre d'accès que l'on peut avoir à faire aux paramètres peut justifier une telle solution qui accélère ces accès après avoir perdu du temps lors de la copie. D'autre part *W* pourra être utilisé pour baser d'autres données conjointement aux accès réalisés dans la zone de copie. Ces gains peuvent ainsi compenser la place "perdue" dans chaque région.

iii) Lors de la compilation d'une procédure il est possible de déterminer toutes les procédures que celle-ci peut appeler et, pour chacune d'elles, le nombre de mots nécessaires à son appel (paramètres, résultat). Il est donc possible de réserver en même temps que les objets locaux une zone d'appel suffisamment grande pour pouvoir servir lors de n'importe quel appel réalisé par la procédure considérée.

Une activation est alors représentée par une seule région (que la procédure soit interne ou externe). Pour pouvoir baser par L la zone des locaux et la zone d'appel, il est nécessaire que :

- les emplacements pour la sauvegarde des registres *L*, *C* et *P* aient un déplacement fixe,
- le début de la zone d'appel soit lui aussi à un déplacement fixe. En effet, les procédures appelées devront pouvoir venir y chercher leurs paramètres.

Il est possible de répondre à ces contraintes: la figure 4.7 propose une solution.

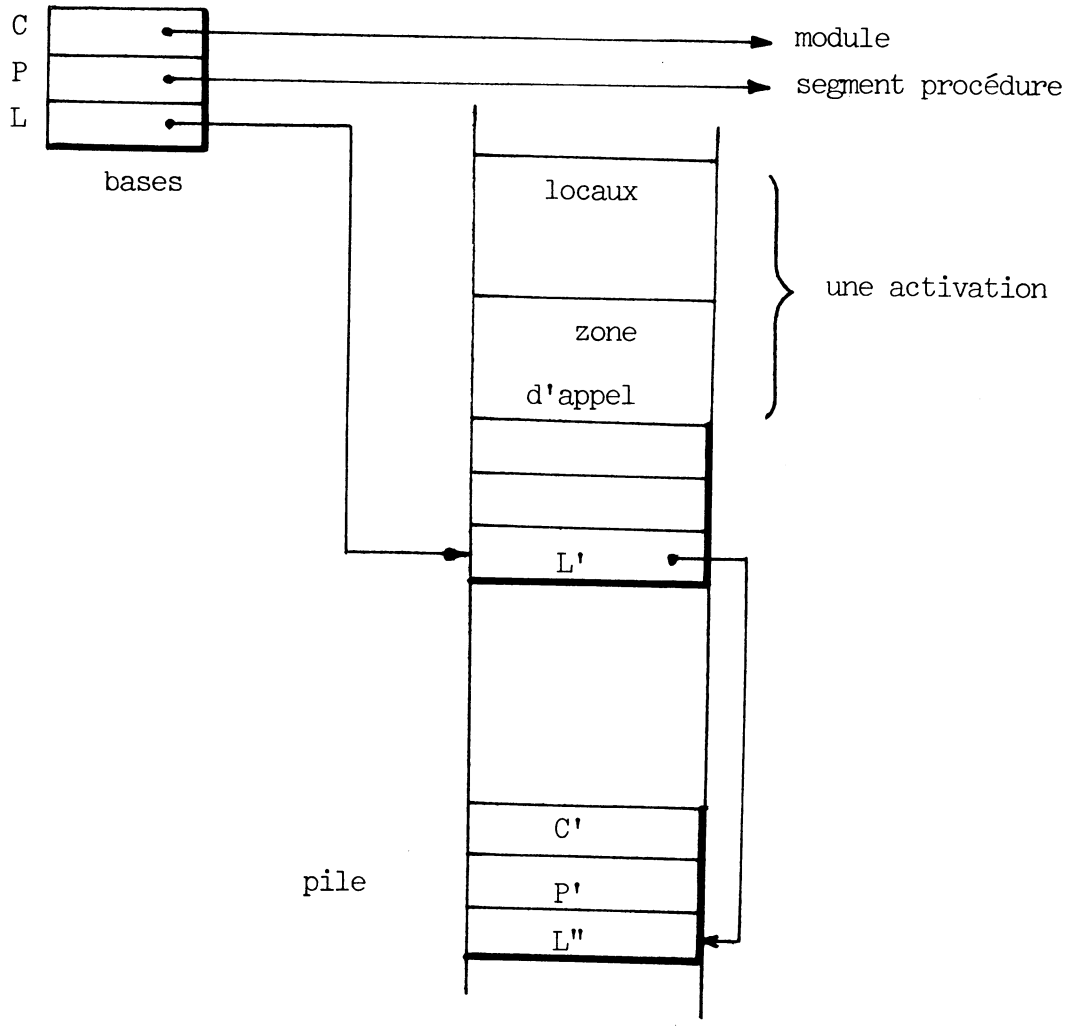


Figure 4.7

Bien que la zone d'appel soit située "en dessous" de la zone des locaux elle est néanmoins destinée à recevoir les paramètres de toute procédure appelée pour cette activation.

Dans le cas d'une compilation à un seul passage, il est intéressant d'avoir la zone des locaux à un emplacement fixe dans la région (pour le calcul des déplacements). Dans ce cas, la zone d'appel (qui n'est plus fixe) pourra être située "au dessus" de la zone des locaux et il sera nécessaire de passer la valeur de sa base lors de l'appel. Cette valeur sauvegardée par l'appelé lui permettra d'initialiser *W* pour accéder aux paramètres. Les variables globales seront accèdées en chargeant *W* avec la sauvegarde de *L*.

Nous pensons que la taille réservée pour la zone d'appel ne sera jamais très importante.

La solution iii) se caractérise par une certaine perte de place (fonctionnement interne) mais les demandes d'allocations sont moins fréquentes et les accès plus rapides.

5. MONITEUR D'EXECUTION

5.1. Introduction

Le moniteur d'exécution forme le système de base de la machine hôte (T1600). Il fournit des fonctions et des objets nécessaires au chargement, à l'exécution et à la mise au point de tout système produit. Il étend les possibilités de la machine hôte favorisant ainsi l'existence d'outils élaborés sur la machine de production. Son existence est, par exemple, connue du compilateur et du connecteur. C'est le noyau de toute exécution : il réalise dynamiquement ce qui n'a pu être fait lors des écritures et des compilations séparées, lors de la connexion. Il gère, en particulier, les processus, les interruptions, l'espace mémoire, il propose des mécanismes de synchronisation, de protection, de liaison, de détection d'erreurs.

Par la suite, nous ne nous intéresserons qu'à certaines de ces "fonctions d'exécution". Nous laisserons de côté en particulier, le mécanisme de détection des erreurs [3]. La part la plus importante sera consacrée à la multiprogrammation.

La conception du moniteur d'exécution est guidée par les principes suivants :

- l'ensemble des primitives choisies doit être adapté à la formulation qui en est faite dans le langage d'écriture ou de connexion (par exemple sous forme d'instructions parallèles, de moniteurs de synchronisation, de déroutements virtuels),
- les spécifications de ces primitives doivent être les moins contraignantes possibles. Elles doivent permettre la réalisation de toute une famille de moniteurs d'exécution présentant des caractéristiques différentes (portant sur la rapidité d'exécution, le gain de place, la fiabilité, la facilité de mise au point...) sans avoir à toucher au compilateur (donc à son interface),
- la structure du moniteur doit faciliter ces modifications (passage d'une réalisation à une autre) en les localisant à des endroits précis et en offrant le plus possible de parties réutilisables.

Pour satisfaire aux deux derniers critères, nous nous sommes donc servis de la notion de module et l'avons appliquée à toute une famille de moniteurs d'exécution. Ils présentent tous le même ensemble de primitives (syntaxiquement équivalent mais sémantiquement différent). Certaines de ces primitives

ont été choisies en fonction du langage d'écriture des modules (instructions parallèles, évolution de la pile, synchronisation...) et ne sont normalement connues que du compilateur ou du connecteur. Ce sont elles qui ont guidé notre étude.

Les paragraphes suivants représentent les différents composants d'un moniteur d'exécution. Pour chacun nous présenterons le modèle général choisi en essayant de le justifier (nous pouvons déjà faire remarquer qu'ils sont construits autour d'une seule notion : processus, file d'attente, pile, interruptions, synchronisation, liens de parenté...).

Enfin, pour que cette démonstration soit complète et pour valider notre décomposition (respect des principes énoncés précédemment) nous présentons, dans la plupart des cas, plusieurs versions possibles d'un même modèle. Ces versions sont, en général, décrites sous forme de programmes (modèles ou procédures) de manière à expliciter complètement (et nous espérons brièvement) ce que réalise chaque partie du moniteur d'exécution. La plupart de ces programmes figurent en annexe. Les sections des paragraphes suivants intitulés "réalisation" sont avant tout destinées à faciliter la compréhension de ces programmes. Nous reprenons les notations introduites dans le chapitre 3 de la thèse de Jean Montuelle.

Dans la décomposition que nous avons retenue, un moniteur d'exécution est composé :

- D'un *scheduler* assurant le multiplexage de l'unité centrale. Il offre pour cela la notion de processus.
- D'un ensemble de procédures permettant la réalisation de moniteurs de synchronisation.
- D'un module gérant les interruptions.
- D'un module conservant les files d'attente (de processus) pour le compte des modules ou procédures précédentes.
- D'un module allouant ou libérant de l'espace mémoire (piles) pour les processus.
- D'un module gérant les processus sous forme d'une hiérarchie et à qui sont adressées les demandes d'activation parallèles.
- D'un module recevant les "appels au superviseur" et les transformant en des appels à des fonctions ou procédures externes de modules particuliers (dont ceux du moniteur).

5.2. Le module *scheduler*

5.2.1. Introduction

Nous voulons décrire le multiplexage de l'unité centrale (partage de la ressource "temps calcul") à l'aide d'un module. *Scheduler* (Fig.5.1) présente donc un ensemble de fonctions externes qui doivent être considérées comme pouvant être appelées par tous les processus pour qui ce multiplexage est réalisé. En fait, à un moment donné, seul le processus en exécution réalisera de tels appels.

Le modèle proposé par T1600 (microprogramme) peut être, par certains côtés, contraignant. Nous pensons, en particulier, à l'identification d'une tâche* par sa priorité fixant une fois pour toutes une relation d'ordre stricte entre les tâches (au sein des files d'attente par exemple).

Scheduler est donc là pour cacher les détails de la multiprogrammation (instructions privilégiées telles que ACQ, ARM, QUIT, structure de la mémoire débanalisée) en présentant une interface modulaire moins restrictive.

Cependant dans le cas où les options retenues par le microprogramme conviennent, il doit être possible d'utiliser au maximum les possibilités offertes par celui-ci pour garantir une certaine rapidité. Ainsi, en toute logique, notre module devrait offrir un état supplémentaire : *différé*** puisque cette possibilité existe dans le microprogramme. Nous n'en parlerons pas (nous supposerons par exemple que tout processus activable possède les informations qui lui sont nécessaires pour s'exécuter immédiatement).

---

* Le terme de tâche sera utilisé lorsque nous ferons référence au scheduler microprogrammé, la notion de processus étant une abstraction présentée par le module *scheduler*.

** Lors de l'affectation de l'unité centrale, un processus de l'état *différé* peut être choisi, mais avant de recharger son contexte d'exécution, une certaine séquence de code sera exécutée (par exemple : va et vient avec la mémoire auxiliaire).

```
pattern scheduler;

ext procedure initialiser;
    % met le scheduler dans l'état initial. C'est obligatoirement la
    première procédure appelée dans ce module %

ext function créer (initial : descrp) : nom_processus;
    % prend en considération le contexte initial d'un nouveau processus.
    Ce processus est mis dans l'état bloqué. Le nom qui est rendu servira
    à le désigner pendant toute sa durée de vie %.

ext procédure débloquer (n : nom_processus; p : integer);
    % le processus identifié par n est mis dans l'état activable.
    p peut être pris comme une priorité %.

ext procédure bloquer (n : nom_processus);
    % le processus identifié par n est placé dans l'état bloqué %

ext procédure attendre;
    % le processus actif est placé dans l'état bloqué %.

ext procédure élire;
    % l'unité centrale est allouée au processus le plus prioritaire de
    l'état actif ou activable %.

ext function actif : nom_processus;
    % fournit le nom du processus qui possède l'unité centrale %.

ext function décrire (n : nom_processus) : contexte;
    % fournit la valeur courante du contexte du processus identifié par n
    (celui-ci doit figurer dans l'état bloqué ou activable) %.

ext function priorité (n : nom_processus) : integer;
    % fournit la priorité initiale du processus n s'il est défini %.

ext procédure tuer ( n : nom_processus);
    % le processus identifié par n est placé dans l'état indéfini.
    n pourra être réutilisé pour identifier un nouveau processus %.

ext procédure mourir;
    % le processus actif est placé dans l'état indéfini. Son nom peut
    être réutilisé %.

end.
```

Figure 5.1

5.2.2. Etat des processus

Tout processus peut se trouver dans l'un des états suivants : indéfini, bloqué, activable ou actif. C'est le scheduler qui effectue les transitions entre ces états en fonction des demandes qui lui sont adressées.

Les transitions possibles sont représentées sur le diagramme de la figure 5.2 .

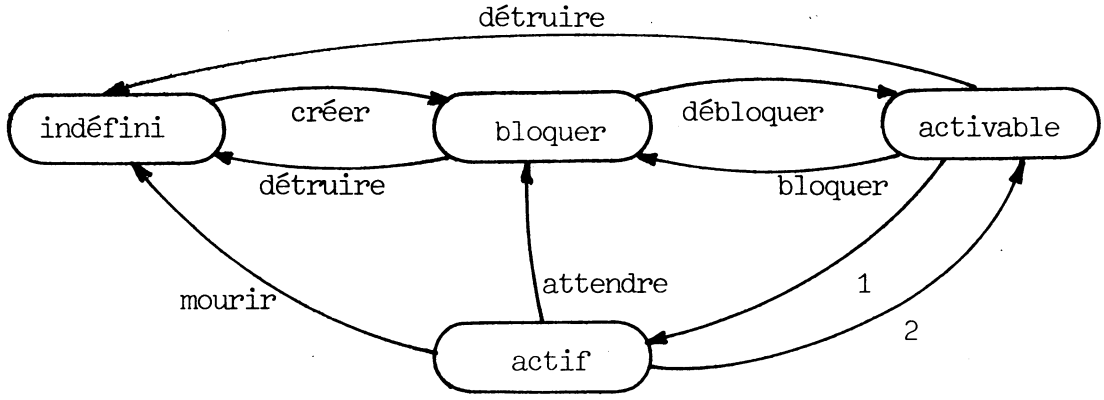


Figure 5.2

Nous avons fait figurer sur chacune des transitions le nom de la demande qui en est responsable. Les branches notées 1 et 2 représentent le multiplexage, ces transitions sont la conséquence d'une demande *élire*. Pour situer l'état initial ou final des processus nous avons présenté un état commun "indéfini" qui contient, pour ainsi dire, les processus inexistants.

Sur un monoprocesseur, au plus un processus peut se trouver dans l'état actif. S'il en existe un c'est forcément celui pour lequel les instructions sont exécutées. L'opération de multiplexage n'intervient qu'à l'occasion d'une demande *élire* . *Scheduler* ne considère l'état des processus qu'à ce moment là et effectue son choix parmi ceux de l'état actif ou activable. Si le processus choisi n'est pas celui qui possédait l'unité centrale, le contexte de ce dernier est sauvegardé avant de charger celui du processus choisi.

Un processus peut donc se voir retirer l'unité centrale à chaque activation de *élire* (et seulement à ces instants là),

- soit parce qu'il existe un processus activable plus prioritaire que lui le forçant à repasser à l'état activable,
- soit parce qu'il n'est déjà plus dans l'état actif (*attendre* et *mourir* ne retirent pas l'unité centrale).

Considérons maintenant les interruptions. L'effet d'une interruption est de faire exécuter, par le processus en exécution, une séquence d'instructions particulières qui s'intercalent entre deux instructions normalement prévues.

Cette séquence est en fait un appel à la fonction *signaler* du module *interruptions* que nous décrivons plus loin. Disons simplement que cette fonction appellera éventuellement *débloquer*, puis *élire*.

Une interruption peut donc s'accompagner d'une commutation de processus sur l'unité centrale. Sur un monoprocesseur, pour réaliser une section critique il suffit de :

- masquer toute interruption à l'entrée (une instruction),
- ne pas employer *élire* à l'intérieur,
- de rétablir le masque avant entrée (une instruction).

Pour délimiter une telle section critique nous emploierons les notations *entrex* et *sortix* (opérations de masquage et démasquage).

Nous avons, désormais, à notre disposition des instructions permettant de décrire facilement des opérations de synchronisation telles que P sur un sémaphore :

```
begin entrex;
      compteur := compteur-1;
      if compteur < 0 then begin
                                 file := file ∪ [scheduler.actif]
                           scheduler.attendre;
                           sortix;
                           scheduler.élire;
                           end;
       else sortix;
end;
```

On comprendra, sur cet exemple, pourquoi nous avons séparé *attendre* et *élire*. A ce niveau, on peut considérer *élire* comme une procédure consultant (en exclusion mutuelle) l'état et les priorités des processus et qui en fonction de cette consultation peut durer plus ou moins longtemps (temps pendant lequel l'unité centrale n'est pas allouée au processus. Il peut être infini si le processus en exécution est passé dans l'état indéfini).

Une partie de programme interruptible est caractérisée par le fait qu'un *élire* peut être intercalé à tout moment (multiplexage de l'unité centrale).

5.2.3. Descripteur de processus

A chaque processus est associé un nom unique. Nous prendrons :

```
type nom_processus = 1... &maxpr;
```

Pendant toute la durée de vie du processus (de l'état indéfini jusqu'au retour dans cet état), ce nom unique permet de désigner un descripteur caractérisant le processus par :

- son contexte d'exécution,
- une indication de priorité.

Le contexte d'exécution est chargé dans les registres de l'unité centrale lorsque celle-ci est allouée au processus pour le remettre dans sa situation d'activation antérieure (ou initiale la première fois). Il est sauvegardé lorsque le processeur lui est enlevé. Au cours de l'exécution c'est le contenu des registres internes qui définissent le contexte.

L'indication de priorité permet d'influencer le choix effectué par *scheduler*. L'utilisation qui en est faite dépend d'une réalisation particulière.

Nous écrirons pour T1600 :

```
type contexte = record
                    regp : address; % compteur-ordinal %
                    regc : address; % base des rémanents %
                    regl : address; % base des locaux %
                    end;
        descrp = record contex : contexte; prior : integer end;
```

Il s'agit ici d'un descripteur réduit mais suffisant au niveau de la description.

Il existe un processus prédéfini. Il porte un nom : *&ancêtre* fixé lors de l'écriture du moniteur d'exécution. C'est le seul existant au "lancement" de tout système. C'est lui qui initialise les différents modules du moniteur. C'est l'ancêtre commun de tous les processus.

5.2.4. Modèle pour *scheduler*

La figure 5.1 présente un modèle général pour *scheduler*. Il tient compte de toutes les transitions d'état possibles. Il propose, en outre, des fonctions donnant des renseignements sur le degré d'avancement du processus (*décrire*) ou sur leur priorité.

L'interface que nous proposons rend compatibles entre elles plusieurs versions possibles. Une réalisation particulière pourra fixer des contraintes supplémentaires : nombre de processus, priorités, possibilité de *tuer* et de *bloquer* un autre processus, allocation par tranches de temps, etc...
Ainsi *tuer*, *bloquer* et *décrire* peuvent être utilisés conjointement à un mécanisme de détection d'erreurs pour contrôler l'évolution de processus peu sûrs et limiter leur durée d'exécution (phase de mise au point, processus utilisateurs).

5.2.5. Contrôles effectués par *scheduler*

Les vérifications qui pourraient être faites au niveau du module *scheduler* quant à la validité des demandes sont ici absentes.

On demande surtout au *scheduler* d'agir rapidement : certaines des fonctions externes, qu'il offre, sont appelées depuis une séquence ininterruptible. En fait, l'existence du scheduler n'est connue qu'à l'intérieur du moniteur d'exécution. Seule la fonction externe *actif* sera offerte à l'extérieur pour fournir aux processus le moyen de s'identifier (par exemple lors de dialogues ou d'accès à des variables rémanentes qui leur sont propres). On suppose donc que les autres fonctions externes sont appelées correctement. Nous verrons leur utilisation dans le module *interruptions*, *gestion_processus* et les procédures implémentant les moniteurs de synchronisation.

5.2.6. Réalisations

Les fonctions externes des modules décrits sont restreintes à celles que nous aurons à appeler par la suite. Sont absentes ici *tuer*, *bloquer* et *décrire*.

Deux tâches sont utilisées pour la gestion interne de *scheduler*.

Celle qui a pour numéro *&maxpr+1* est toujours activable et de plus basse priorité. Elle boucle perpétuellement sur l'instruction "attente d'interruption" qui met l'unité centrale en état de veille. C'est l'adresse de ce segment procédure qu'*initialiser* charge dans *situation* [*&maxpr+1*].

Celle qui a pour numéro *&maxpr+2* est la tâche associée aux processus indéfinis. *Situation* [*&maxpr+2*] est utilisé pour recevoir le contexte d'un processus de nom x qui vient de se détruire. En effet, comme x peut déjà avoir été réalloué, *situation* [x] ne sera pas détruit à la suite d'un *élire*.

a) *Scheduler_1* (Fig. A2.1) est directement inspiré du moniteur microprogrammé en ce qui concerne la politique d'allocation de l'unité centrale.

La priorité d'un processus est déterminée par son nom : le processus qui sera élu est celui qui possède le nom le moins élevé (c'est ce que nous avons exprimé dans le commentaire de A2.1 en employant la notation de *first of** au lieu de *one of*).

Comme seule *créer* décide du nom, un tel modèle ne pourra être utilisé que lorsque la priorité attachée à un processus a peu d'importance. En effet, les priorités attribuées ne peuvent être contrôlées et pourront varier d'une exécution à l'autre. L'avantage de cette réalisation est d'être proche du microprogramme, donc rapide.

Remarque : Si *créer* utilise *first of libre*, lors de sa création un processus reçoit le nom libre le moins élevé. Ainsi, si un système n'emploie qu'un *par begin... par end* (pour lancer plusieurs processus "bouclant") nous pourrons, par exemple, affirmer** que les instructions parallèles reçoivent une priorité décroissante dans l'ordre de leur écriture.

---

* Nous supposons que la définition du type d'un ensemble induit un certain ordre sur les éléments de cet ensemble. *One of* ne fait pas référence à cet ordre, mais l'implémentation choisie peut le rendre équivalent à *first of*.

** Cette affirmation dépend aussi de l'algorithme choisi dans *gestion-processus* décrit plus loin.

b) Tout en gardant la même politique d'allocation de l'unité centrale, on désire maintenant influencer le choix qui est fait du nom. Le champ *prior* figurant dans le paramètre *initial* de *créer* détermine une certaine classe de priorité. Supposons *prior = 0.. 4* (soit 5 classes de priorité croissante). Nous limitons à 25 le nombre de processus pouvant exister en même temps dans une même classe : *classe = 1.. 25*.

L'état d'allocation des noms est représenté par :

```
libre : array [0..4] of classe
```

Le choix du nom est réalisé par la séquence :

```
i := 4-prior of initial;
if libre[i] = [ ] then trap ("classe saturée")
   else begin
        j := one of libre [i];
        libre := libre - [j];
        nom := 25 * i+j
        end;
```

Un processus est maintenant élu si aucun autre, actif ou activable, ne figure dans une classe de priorité supérieure.

Remarque : Il est peu probable que dans un système constitué de modules écrits et compilés séparément, on puisse avoir 125 classes ne contenant qu'un seul processus (ce qui équivaudrait à laisser le choix du nom entièrement à la charge du programmeur).

c) Si l'on désire adopter une autre politique d'allocation de l'unité centrale, il est nécessaire que le choix du processus à activer se fasse en dehors du microprogramme. Dans *scheduler_2* (Fig. A2.2) le processus est choisi par programme avant d'être présenté, seul, dans *actif*. C'est donc lui qui sera élu lors du prochain *ACQ*.

La gestion des processus activables et actifs en file d'attente est réalisée dans un module séparé (différents modèles *file* seront présentés ultérieurement).

Nous précisons ici que la fonction : *file.classer (x, y, p)* range le nouvel élément y dans la file d'attente dont le premier élément est x, en lui attribuant la priorité p. L'élément qui est désormais premier est retourné en résultat.

&maxpr+1 est le dernier élément de toute file, par convention il a la priorité la plus basse.

débloquer, *attendre* et *mourir* sont moins rapides que dans *scheduler_1*.

Mais la priorité des processus n'est plus liée à leur nom (ni au numéro de la tâche qui leur est associée) : elle peut évoluer dynamiquement en fonction :

- du paramètre passé lors de chaque *débloquer* (par exemple, pour rendre prioritaire le traitement d'une interruption),
- de l'instant du passage à l'état activable ("premier entré, premier sorti"),
- d'une stratégie interne au *scheduler* (basée, par exemple, sur le temps d'occupation de l'unité centrale).

d) La fonction qui associe à un nom de processus un numéro de tâche peut être différente de la fonction "identité" :

- si avec un mécanisme d'élection semblable à celui de *scheduler_1* nous désirons affecter à un nouveau processus un numéro de tâche déjà pris,
- si le nombre de processus nécessaires est supérieur à 125. Dans ce cas les tâches (contexte et éventuellement priorité) sont associées aux processus qui ont les plus grandes chances d'être activés par la suite.

5.3. Procédures pour la synchronisation

5.3.1. Choix des moniteurs de synchronisation

La gestion centralisée de toutes les variables de synchronisation d'un système peut se concevoir à l'intérieur d'un même module : *gestion-ressources* par exemple. Celui-ci pourrait faire partie du moniteur d'exécution en offrant des procédures externes de la forme :

```
function créer_ressource (nbre_pts_entrée : integer) : nom_sémaphore;
procédure détruire_ressource (n : nom_sémaphore);
procédure réclamer (n : nom_sémaphore);
procédure relacher (n : nom_sémaphore);
```

Une telle solution :

- pose des problèmes d'allocation dynamique,

- s'accorde mal avec une écriture séparée des modules puisque un des "avantages" est la possibilité de *réclamer* et *relâcher* le même sémaphore depuis différents modules.

Nous pensons qu'il est préférable de réserver statiquement les variables de synchronisation lorsque l'on en a besoin. Elles participent au même titre que les autres variables à la définition du module que l'on décrit. Elles ne sont consultées et modifiées qu'à travers les procédures externes qu'offrent le module. Nous supposons ici que le langage d'écriture propose la notion de moniteur [5] pour exprimer les contraintes de synchronisation. En effet, ceux-ci s'accordent bien avec la programmation modulaire : un moniteur est un module synchronisé.

5.3.2. Procédures pour les moniteurs de synchronisation

Les moniteurs se distinguent des autres modules par l'utilisation supplémentaire des 4 procédures suivantes : *enter, exit, wait* et *signal*.

Ces procédures sont contenues dans le moniteur d'exécution pour être mises en commun pour tous les moniteurs (c'est un nouvel exemple de segments-procédure partagés entre différents modules).

D'autre part, l'appel de ces procédures (par "appel au superviseur") permettra de faire passer le processus actif en mode maître, mode nécessaire pour effectuer des opérations pouvant entraîner une commutation de processus.

L'exclusion mutuelle des fonctions d'un moniteur est assurée par les fonctions *enter* et *exit* appelées recpectivement au début et à la fin de toute fonction externe de ce moniteur. Les variables de synchronisation correspondantes sont :

```
m_libre : boolean init true
m_premier : 1 .. &maxpr+1 init &maxpr+1
                % contient &maxpr+1 lorsque le moniteur est libre,
            sinon le nom du premier processus en attente %
```

Dans la description que nous donnons (Fig. 5.3) ces deux variables sont considérées comme globales.

A chaque condition (de synchronisation) correspond une variable :

```
c_premier :1 .. &maxpr+1 init &maxpr+1
                % contient le nom du premier processus en attente sur
                  cette condition ou &maxpr+1 si aucun n'est en attente %
```

```
procédure enter;
    begin entrex;
        if m_libre then begin m_libre := false;
                              sortix;
                        end
          else begin
               m_premier := classer1 (m_premier, scheduler.actif, 0);
               scheduler.attendre;
               sortix;
               scheduler.élire;
               end;
    end;

procédure exit;
    begin entrex;
        if m_premier = &maxpr+1 then begin m_libre := true;
                                           sortix;
                                     end
          else begin
               scheduler.débloquer (m_premier, 0);
               m_premier := successeur1 (m_premier);
               sortix;
               scheduler.élire;
               end;
    end;
```

Figure 5.3 (début)

```
procédure wait (var c_premier : 1..&maxpr+1; p : integer);
    begin entrex;
        if m_premier = &maxpr+1 then m_libre := true
          else begin
               scheduler.debloquer (m_premier, 0);
               m_premier := successeur1 (m_premier);
               end;
        c_premier := classer2 (c_premier, scheduler.actif, p);
        scheduler.attendre;
        sortix ;
        scheduler.élire;
    end;

procédure signal (var c_premier : 1 ..&maxpr+1);
    begin entrex;
        if c_premier = &maxpr+1 then sortix
          else begin
               scheduler.débloquer (c_premier, 0);
               c_premier := successeur2 (c_premier);
               m_premier := classer1 (m_premier, scheduler.actif, 0);
               scheduler.attendre;
               sortix;
               scheduler.élire;
               end;
    end;
```

Figure 5.3 (fin)

Comme il peut y avoir plusieurs conditions par moniteur, nous supposons que les expressions :

<condition> . wait (p) et *<condition> . signal*

sont traduites en :

wait (<condition>_premier, p) et *signal (<condition>_premier)*

avec passage par adresse de *<condition>_premier*

Le paramètre *p* figurant dans *wait* est une indication sur la place que devra occuper le processus actif dans la file d'attente associée à la condition. Une opération *signal* portant sur la même condition débloquera un processus parmi ceux ayant eu la plus forte valeur pour *p*. C'est un module séparé qui assure la gestion de ces files (pour toutes les conditions), *Classer2* et *successeur2* sont des références fictives aux procédures *classer* et *successeur* de ce module (modèle *file* fig. 5.5).

Si au niveau du langage d'écriture, le paramètre *p* n'est pas prévu, il suffit de considérer que *<condition>. wait* est transformé en *wait (<condition>_premier, 0)*.

Comme la gestion de la file d'attente associée à la variable d'exclusion du moniteur peut être différente, *classer1* et *successeur1* peuvent faire référence à un module *file* autre que celui associé aux conditions.

5.4. Module pour la gestion des interruptions

Nous voulons décrire, sous forme de module, les interactions que peuvent avoir les processus avec le mécanisme des interruptions. Il s'agit toujours de synchronisation mais cette fois-ci avec des processus externes* (tels qu'horloge, unités d'entrée/sortie, lignes de comptage). La figure 5.4 présente le modèle *interruptions* que nous avons retenu (une description plus détaillée existe à la figure A2.3).

---

* Les déroutements et appels au superviseur sont gérés respectivement par le mécanisme de traitement d'erreurs et par le module *tronc-commun* (ce dernier sera décrit par la suite).

```
pattern interruptions;

ext procédure initialiser;
    % c'est la première procédure à exécuter pour ce module %.

ext procédure signaler (n : niveau; c : cause);
    % c'est la procédure qui est appelée lors de toute interruption.
    n et c permettent d'identifier l'origine de l'interruption.
    c est mémorisé jusqu'à un nouvel appel avec la même cause.
    Le premier processus (s'il existe) qui attend une interruption
    de ce niveau est débloqué %.

ext function recevoir (n : niveau; p : integer) : cause;
    % le processus actif indique qu'il est prêt à effectuer le traitement
    d'une nouvelle interruption. Si aucune interruption n'est arrivée
    depuis le dernier recevoir sur le même niveau, il est mis en file
    d'attente avec la priorité p %.

end.
```

Figure 5.4

Toute interruption se traduit par l'insertion, dans le déroulement "normal" du processus actif, d'un appel à la procédure externe *signaler*. Une certaine priorité est donnée au processus éventuellement débloqué.

Le module *interruptions* cache le mécanisme proposé par T1600 sous forme de tâches *hard*. Celles-ci resteront la propriété du moniteur d'exécution pour en garantir une bonne utilisation et proposer des niveaux d'interruptions éventuellement différents des niveaux réels* (en particulier certains niveaux directement traités par le moniteur d'exécution auront disparu).

---

* Il s'agit, en fait, des sous-niveaux de T1600.

La réalisation d'un tel module doit donc

- définir les différents niveaux d'interruptions qui seront proposés (en particulier le type de *cause* et ses valeurs possibles)
- affecter, éventuellement, une priorité à chacun d'eux
- déterminer les tâches *hard* qui doivent réaliser la gestion de ces interruptions idéalisées. Celles-ci auront à déterminer, lors d'une interruption réelle, le niveau virtuel et la cause avant d'effectuer *signaler*.

5.5. Module pour la gestion des files d'attente

5.5.1. Modèle

La gestion des files d'attente fait l'objet de modèles particuliers.

En effet, nous avons vu qu'il était possible de définir *scheduler*, *interruptions* et les procédures des moniteurs de synchronisation sans faire de suppositions sur l'organisation des files d'attente*. Il est donc normal que le choix d'une organisation particulière puisse se faire à l'extérieur de tels modules ou procédures.

D'autre part, un processus ne peut se trouver que dans une seule file d'attente à la fois. Il est donc possible de regrouper dans une structure de données unique toutes les files d'attente (de même organisation) plutôt que d'en prévoir une pour chaque niveau d'interruption, chaque condition ou moniteur de synchronisation. Ce regroupement autorise, en outre, des vérifivations simples de cohérence (ex : tout processus dans une seule file).

La figure 5.5 présente un modèle général pour une telle gestion. Les informations à ordonner dans une des files sont des entiers compris entre 1 et *&maxelem+1*. Une file est identifiée par le nom de son premier élément (tout élément d'une file identifie une sous-file). *&maxelem+1* identifie la file vide. C'est donc le seul élément qui pourra figurer dans plusieurs files.

---

* Même dans le cas d'un *wait* parametré, si deux processus recoivent le même paramètre il peuvent être ordonnés suivant : leur nom, l'instant où il sont mis en attente, la priorité de leur création...

```
pattern file;

ext procédure initialiser;
    % ce doit être la première procédure appelée dans ce module.
    Toutes les files sont vides %.

ext function classer (x, y : élément; p : integer) : élément;
    % y est placé dans la file d'attente commençant par x (et se terminant
    par &maxelem+1). p pourra être pris comme une priorité attachée à y.
    Enfin le premier élément de la pile constitue le résultat %.

ext function successeur (x : élément) : élément;
    % rend l'élément suivant x dans la file identifiée par x %.

ext procédure retirer (x,y : élément);
    % retire l'élément y de la file identifiée par x %.

end.
```

Figure 5.5

5.5.2. Réalisations

Dans les files d'attente utilisées pour la gestion des processus (par *scheduler* et les mécanismes de synchronisation) les éléments sont des noms de processus ou *&maxpr+1*.

Ici encore nous supposons que les modules sont utilisés correctement. En particulier que :

- un processus est rajouté à une file vide par :
  classer (&maxpr+1, y, p),
- aucune demande n'est de la forme : *successeur (&maxpr+1)* ou *retirer (&maxpr+1)*

a) Dans *file_1* (Fig. A2.4) les processus seront rangés suivant l'ordre de leur nom. L'indication de priorité n'est donc pas utilisée. Cette politique est la plus proche de T1600. C'est de cette façon que sont rangés les processus activables de *scheduler_1*. Mais ce dernier n'utilise pas les services de *file_1* puisque l'ensemble *activable* nécessaire au microprogramme suffit à assurer cette gestion. Mais dans ce cas, *file_1* peut être utilisé par la synchronisation pour garder la même politique au niveau de toute file d'attente.*

b) Un modèle qui applique la politique "premier entré, premier sorti" est décrit par *file_2* (Fig. A2.5). Ici encore l'indication de priorité n'est pas utilisée. Remarquons que la rapidité de *classer* peut être élevée dans tous les cas si l'on prend soin, dans le module appelant, de tenir à jour, en même temps que l'élément qui identifie la file, la valeur du dernier élément (autre que *&maxelem+1*).

L'adjonction d'un nouveau processus se fait par :

classer (dernier, y, 0)

c) Enfin *file_3* (Fig. A2.6) tient compte de l'indication de priorité passée en paramètre dans *classer*. Les processus sont rangés dans l'ordre des valeurs décroissantes de p. A *&maxelem+1* est associée, en permanence, la valeur *-1*. Lorsque la même valeur de p est associée à deux processus, la politique "premier entré, premier sorti" est appliquée. On suppose $p \geq o$. Si $p=o$ c'est la priorité donnée par *scheduler.priorité* qui est utilisée (ce sera, par exemple, la priorité donnée lors de la création du processus).

C'est un tel modèle qui sera utilisé lorsque le langage d'écriture offre un *wait* paramétré ($p \neq o$).

Il y a, en général, deux manières d'utiliser l'indication de priorité :

- p prend peu de valeurs distinctes : dans ce cas il s'agit plutôt de classes de priorité, la politique prise pour résoudre le cas des p égaux est alors un choix qui peut être important,

---

* Dans ce cas, il est aussi possible d'employer les opérations sur sémaphores fournis par T1600 pour réaliser la synchronisation (on trouvera en [5] une réalisation des moniteurs au moyen de sémaphores). Si l'exécution peut s'avérer plus rapide, les files sont alors multipliées au niveau de chacun des sémaphores.

- p prend des valeurs réparties uniformément sur un intervalle très grand : dans ce cas la politique choisie peut être indifférente (où influencée par un critère de rapidité).

Dans les trois exemples présentés *retirer* pourrait être amélioré si l'on disposait d'un tableau *précédent* permettant d'établir des liens inverses dans une file d'attente. Mais le supplément de travail à faire lors de *classer* et *successeur* peut contrebalancer le gain réalisé lors de *retirer* (les files sont en général assez courtes). De plus, dans le cas où l'on s'interdit de *tuer* un processus, la fonction *retirer* n'est utile que pour la file des processus activables.

5.6. Module allocateur de piles

5.6.1. Modèle

C'est le module qui gère l'espace nécessaire à l'exécution des processus. Chaque processus possède son espace propre : lorsqu'il est créé, une pile lui est préparée. A chaque évolution de cette pile, il doit avertir ce module pour que la gestion s'effectue correctement. En particulier, il peut ainsi vérifier qu'il ne "dépasse" pas l'espace qui lui est attribué (nous ne disposons pas, en effet, de moyen de protection câblé) ou bien faire en sorte que son espace évolue en fonction de ses besoins.

Cette évolution se fait par blocs (un certain nombre d'emplacements contigüs) correspondant aux activations de procédure (région). L'empilement ou le dépilement d'un bloc sont signalés au moyen des fonctions *fournir* et *rendre*.

Les piles évoluant de façon distincte en fonction de l'avancement des processus, il n'est pas toujours possible de garantir la contiguité des blocs d'une même pile. C'est pourquoi la gestion de la base permettant d'accéder aux emplacements du dernier bloc (bloc courant) est assurée par ce module. Nous dirons que la fonction *fournir* renvoit un nom de bloc et que les accès se font par {nom de bloc, déplacement dans ce bloc}. C'est au processus de conserver dans sa pile sous forme de liens dynamiques, entre les différents blocs, les valeurs successives des bases.

Si le processus est détruit par un autre, *enlever* permet de détruire sa pile. Cette procédure remplace donc tous les *rendre* qui auraient dû être faits si le processus avait atteint sa fin normale.

Les allocateurs de piles (il n'en existe qu'un seul exemplaire par moniteur d'exécution) seront construits suivant le modèle de la figure 5.6.

```
pattern allocateur_pile;
ext procédure initialiser
    % initialise l'état du module. Ce doit être la première procédure
    appelée %.
ext procédure préparer (p : nom_processus);
    % prépare une pile pour le processus identifié par p %.
ext function fournir (n : integer) : nom_bloc;
    % un bloc de taille n est réservé au sommet de la pile du processus
    actif. Si cette opération est impossible un déroutement virtuel
    ("dépassement")est déclenché. Si elle n'est pas possible, pour
    l'instant, un déroutement ("saturé")est déclenché %.
ext procédure rendre ( b : nom_bloc);
    % un bloc est enlevé du sommet de la pile du processus actif. Si
    le module détecte que la valeur de b n'est pas compatible avec sa
    propre gestion, un déroutement virtuel ("incompatible")est déclenché %.
ext procédure enlever (p : nom_processus);
    % détruit la pile associée au processus identifié par p.
    Est utilisée lorsque le processus ne se termine pas normalement %.
end.
```

Figure 5.6

Les vérifications qui sont faites au niveau de ce module ne sont pas forcément complètes. On peut, par exemple, faire confiance au compilateur pour :

- limiter l'adressage d'objets de la pile aux seuls blocs fournis et non rendus (il serait difficile de faire mieux en l'absence de mécanisme de protection câblé),
- formuler des demandes correctes (*fournir* et *rendre*).

D'autre part, seuls certains modules du moniteur d'exécution feront référence à *préparer* et *enlever*, ce qui garanti un certain "domaine de validité". Par exemple *préparer* n'est appelé que par *scheduler* et seulement lors de la création d'un nouveau processus.

On laisse donc à la spécification d'une version particulière du modèle, le soin de fixer les contrôles qui sont effectués (la redondance des tests est utile pour augmenter la fiabilité d'un système). Dans nos exemples nous n'écrirons de tels tests que s'ils n'alourdissent pas la description

5.6.2. Réalisations

5.6.2.1. Allocation statique

La réalisation la plus simple consiste à partitionner l'espace réservé pour l'exécution des processus en autant de piles qu'il peut y avoir de noms de processus différents. La demande *préparer* initialise le sommet de la pile. Les fonctions *fournir* et *rendre* se contentent de vérifier que le sommet de pile ne dépasse pas les deux extémités de la zone impartie. Un exemple d'une telle réalisation est décrit à la figure A2.7.

Si cette solution est simple, elle est assez peu efficace car elle fait perdre beaucoup de place :

- à un instant donné le nombre de processus créés peut être très inférieur au nombre maximum de processus possible,
- la taille allouée pour un processus peut être insuffisante alors que d'autres processus n'utilisent que très peu leur espace.

C'est évidemment cette solution que l'on retiendrait si l'on connaissait, a priori, tous les processus et la taille nécessaire de chacune de leur pile.

5.6.2.2. Allocation pour morceaux de taille fixe

Pour pallier à ces deux objections, il suffit de partitionner l'espace réservé en morceaux de taille fixe, non liés à un processus particulier. Une pile pourra être composée d'un ou plusieurs de ces morceaux.

La demande *préparer* associe le morceau indéfini à la pile (c'est un morceau où il n'est pas possible d'ajouter un nouveau bloc. La fonction *fournir* alloue un nouveau morceau lorsque le morceau courant s'avère insuffisant. Lorsque, lors d'un *rendre*, le dernier bloc d'un morceau est enlevé de la pile, ce morceau est désalloué. Un exemple d'une telle réalisation est décrit à la figure A2.8 .

Par rapport à la solution précédente, il n'existe plus qu'un problème de fractionnement interne dans chacun des morceaux. Par contre, la taille de l'unité d'allocation ayant diminuée, elle limite d'autant plus la taille maximum autorisée pour un bloc.

Il faut cependant remarquer qu'en programmation modulaire les objets de tailles importantes (tableaux par exemple) décrivent, en général, l'état d'un module particulier qui en contrôle et structure les accès. Ce sont des variables rémanentes qui ne figurent donc pas dans la pile.

D'autre part, dans le cas du T1600, une base permet d'adresser au plus 256 mots. Le compilateur limitera donc, en général, la taille du bloc à ce nombre. En supposant, par exemple, la taille moyenne d'un bloc égale à 20 mots, une taille de morceaux de 256 mots permet l'activation simultanée d'une douzaine de procédures pour un même processus (ce qui est déjà beaucoup). L'existence de piles composées de plusieurs morceaux seraient donc assez rare. De ce fait, les opérations *fournir* et *rendre* ne sont que peu ralenties par rapport à la solution précédente : un ralentissement n'intervenant que lorsque l'on doit allouer ou désallouer un nouveau morceau.

5.6.2.3. Allocation par blocs d'activation

Une solution différente est d'allouer dynamiquement un bloc de la taille demandée à chaque *fournir*.

Par exemple, l'espace réservé est géré sous forme de tas ou bien les blocs sont puisés dans le tas unique du moniteur d'exécution (*fournir* et *rendre* sont typiquement les opérations d'un tas). Toutes les techniques propres à ce genre de gestion peuvent être appliquées.

Remarquons cependant que :

- si la fragmentation interne disparaît, apparaît une fragmentation externe,
- il est impensable d'effectuer des translations : des repères sur les blocs pouvant exister un peu partout (dans des registres : W ou L, dans des emplacements mémoire : liens statiques et dynamiques, paramètres passés par adresse),
- dans tous les cas (que la pile soit importante ou non) des algorithmes assez longs sont mis en jeu (recherche d'un bloc convenable, fusion..).

C'est pourquoi nous proposons de gérer l'espace réservé sous forme de pile unique. Chaque bloc est alloué au sommet courant de la pile (algorithme très simple pour *fournir*). Pour restituer un bloc, on le marque libre. Si ce bloc est situé au sommet courant de pile, tous les blocs marqués libres sont enlevés jusqu'au premier bloc non libre (algorithme simple pour *rendre*).

Cette solution est cependant très dépendante du comportement des processus. Elle est, par exemple, intéressante si tous les processus évoluant en parallèle se comportent de la même façon, c'est-à-dire si les procédures qu'ils exécutent à un moment donné ont tous une durée d'activation du même ordre de grandeur.

Dans le cas contraire, la situation suivante peut arriver : il n'est plus possible de fournir un bloc alors que la plupart ont été libérés mais non enlevés (par exemple il existe un processus exécutant une procédure récursive où il attend longtemps à chaque fois alors que les autres processus évoluent très vite).

Un exemple d'un tel module est décrit à la figure A2.9.

A chaque bloc de la pile est associé un descripteur qui contient la base allouée pour ce bloc et un lien vers le descripteur du bloc précédent (pour le même processus). Les blocs libres mais non encore enlevés contiennent *-1* dans leur champ *précédent*.

Tous ces descripteurs sont gérés en pile dans *région*. Nous avons utilisé un tableau (limité par *&maxblocs*) pour les besoins de la description. Pour la réalisation, on pourra faire figurer ces descripteurs (modifiés) dans la pile d'allocation elle-même. Si l'on désire une réalisation plus proche de la description, on utilisera deux piles évoluant en sens contraire (entre *&haut* et *&bas*). Cette dernière solution a l'avantage de séparer variables de gestion et variables gérées (région), donc de conserver la notion de module (protection).

5.7. Module pour la gestion des processus

Ce module structure l'ensemble des processus, sous forme d'une hiérarchie, par la relation "créateur (père) de". L'état du module représente donc à tout moment le graphe de cette relation.

Cette hiérarchie est le reflet de l'expression du parallélisme dans le langage d'écriture. L'instruction :

par begin
{ *<p1>* } *<instruction 1>;*
{ *<p2>* } *<instruction 2>;*
⋮
{ *<pn>* } *<instruction n>;*
par end

sera transformée par la compilation en :

```
      gestion_processus.créer (@instruction 1, p1);
      goto L1;
      <instruction 1>;
      gestion_processus.terminer;
L1 :  gestion_processus.créer (@instruction 2, p2);
      goto L2;
        .
        .
        .
Ln :  gestion_processus.place_aux_jeunes;
```

Le module *gestion_processus* (Fig. 5.7) est donc directement inspiré de cette traduction : il offre les trois procédures externes citées dans ce dernier programme. Si aucune indication de priorité ne figure elle sera traduite par 0.

```
pattern gestion_processus;

ext procédure initialiser;
    % met le module dans l'état initial : il n'existe qu'un seul processus
    prédéfini : l'ancêtre de tous les autres %.

ext procédure créer (début : address; p : integer);
    % crée un nouveau fils pour le processus actif. Début repère la
    première instruction à exécuter par ce fils et p est une indication
    de priorité %.

ext procédure place_aux_jeunes
    % cette procédure ne prendra fin que lorsque tous les fils (actuels)
    du processus actif auront terminé %.

ext procédure terminer;
    % l'appel de cette procédure est la dernière instruction qu'exécute
    un processus. Elle signale qu'il a terminé son travail et qu'il
    peut être détruit %.

end.
```

Figure 5.7

Remarque : Le père aurait pu exécuter lui-même une des instructions parallèles. Mais nous avons choisi de suspendre son activité pendant que ses fils travaillent. Ceci pour des raisons de protection. En effet, certaines parties du système peuvent être peu sûres. L'exécution d'un *par begin... par end* (même avec une seule instruction "parallèle") garantit alors que :

- l'état d'allocation de la pile du père ne sera pas modifiée,
- le père pourra reprendre son activité, lorsqu'une erreur survenant dans une des branches parallèles entraîne une destruction du processus fautif,
- l'identité du père ne pourra être utilisée pour réclamer des services que l'on ne désirerait pas fournir (voir par exemple le paragraphe suivant).

Elle permet aussi éventuellement de redéfinir des priorités pour l'ensemble des instructions parallèles.

La figure A2.10 donne une réalisation possible d'un tel module. Une telle description est indépendante des choix qui peuvent être faits au niveau de *scheduler*.

Cependant, dans le cas où les options prises par le microprogramme conviennent (c'est-à-dire lorsque l'on peut employer *scheduler_1*) il est possible d'écrire ce module en employant les instructions :

- ARM équivalent à *scheduler.débloquer; scheduler.élire*
- QUIT équivalent à *scheduler.attendre; scheduler.élire*
- ACT et WAIT (opération *P* et *V* sur sémaphore privé)

La rapidité d'exécution peut s'en trouver améliorée.

5.8. Module pour la gestion des appels au superviseur

5.8.1. Introduction

Les processus sont créés en mode esclave. Il leur est, cependant, nécessaire de passer, de temps à autre, en mode maître pour pouvoir exécuter des instructions privilégiées. Par exemple c'est le cas, lorsqu'ils activent certaines fonctions du moniteur d'exécution (où l'on masque des interruptions, commute des processus etc...). Ce changement de mode est réalisé grâce aux instructions "appel au superviseur" (*SVC*) et "retour du superviseur" (*RSV*).

D'autre part, dans l'ensemble des modules supportés par la machine à un moment donné peuvent cohabiter des parties de durées de vie différentes (sous-systèmes). Par exemple :

- le moniteur d'exécution (durée de vie la plus longue) et un système d'exploitation particulier,
- un système d'exploitation et le travail à réaliser pour un usager (par exemple une compilation, un tri...).

Il est donc nécessaire d'établir des liaisons lors de l'exécution pour que la partie de plus courte durée de vie puisse demander des services à l'autre. Le mécanisme des SVC est capable d'assurer une telle liaison en associant à un numéro (paramètre du *SVC*) le couple d'adresse repérant une certaine fonction d'un certain module serveur. Lorsque les différentes parties sont protégées il permet aussi de franchir la "barrière de protection" (en un point obligé).

5.8.2. Modèle

La gestion de ces liaisons est assurée par un module indépendant : *tronc_commun* (Fig. 5.8). Toute exécution de l'instruction *SVC n* active la procédure *appeler* de ce module. La fonction externe associée au numéro *n* est appelée ensuite. La figure 5.9 illustre ce mécanisme.

Il est nécessaire que le contenu de la table de liaison de *tronc_commun* puisse évoluer en fonction des différentes parties qui se succèdent lors de l'exécution. Par exemple, lors d'un changement de système d'exploitation les services qui sont rendus aux utilisateurs sont alors modifiés. C'est pourquoi le module *tronc_commun* possède deux fonctions externes : *ajouter* et *retrancher* qui permettent de modifier les associations {numéro de SVC, fonction externe d'un module}.

Ces modifications doivent être contrôlées pour éviter des incorrections toujours possibles. Elles interviennent donc lors de l'initialisation d'un sous-système et lors de son abandon, opérations qui peuvent être réalisées par le même processus (celui qui charge et "décharge" cette partie).

Ce processus ajoute les associations correspondant à :
- des services que peut rendre ce nouveau sous-système,
- des fonctions devant être exécutées en mode maître.

```
pattern tronc_commun;

ext procédure initialiser;

    % met le module dans l'état initial : les seules associations alors connues
    sont celles correspondant à rajouter, retrancher et transmettre. Un seul
    processus (&ancêtre) a le droit de rajouter et de transmettre %

ext procédure rajouter (n : integer; pe : point_entrée; maître : boolean);

    % prend en compte une nouvelle association si le numéro n n'est pas déjà
    associé et si le processus actif a ce droit %

ext procédure retrancher;

    % supprime toutes les associations rajoutées par le processus actif qui
    perd le droit de rajouter et de transmettre %

ext procédure transmettre (p : nom_processus; t : boolean; maître : boolean);

    % si le processus actif a ce droit il peut le transmettre à un autre
    processus (t = trues) en même temps que le droit de rajouter (avec mode
    maître possible ou non) %

ext procédure appeler (n : integer);

    % reçoit tous les appels superviseur et les traduit en l'appel de la
    fonction externe associée à n (si elle existe) %

end.
```

Figure 5.8

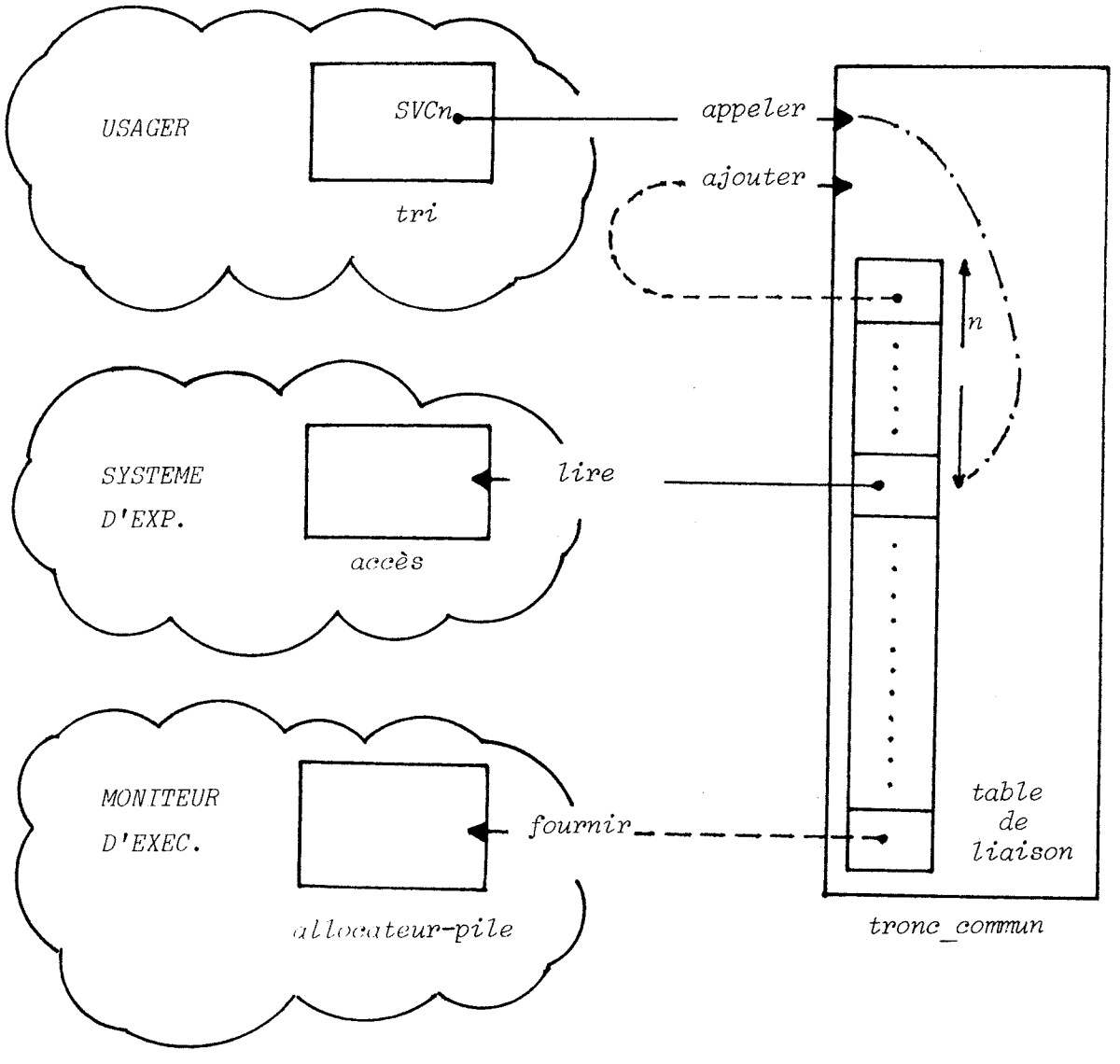

USAGER
SVCn
tri
appeler
ajouter
n
SYSTEME
D'EXP.
accès
lire
MONITEUR
D'EXEC.
allocateur-pile
fournir
table
de
liaison
tronc_commun

Figure 5.9

Ensuite ce processus peut transmettre à un ou plusieurs de ses fils les possibilités :

- d'ajouter de nouvelles associations (lors du chargement d'une nouvelle partie),
- de transmettre à son tour ses possibilités.

Par exemple : *transmettre (n, false, false)* autorise le processus n à ajouter de nouvelles associations (mais sans passage en mode maître) et lui supprime le droit de transmettre à son tour.

L'opération *retrancher* supprime toutes les associations ajoutées par le processus actif. En particulier un processus qui n'a pu en ajouter ne pourra pas en détruire.

Les associations correspondant à *ajouter*, *retrancher* et *transmettre* sont permanentes et seul le processus *&ancêtre* possède initialement les possibilités de rajouter des associations et de transmettre ses droits.

5.8.3. Insertion des SVC dans les segments procédures

Considérons tout d'abord le cas des appels au moniteur d'exécution. Certaines des primitives qu'il propose (gestion des processus, gestion des piles) sont connues du compilateur.

Parmi celles-ci, certaines réclament le mode maître (gestion de processus), donc un appel sous forme de SVC. Par contre, les autres (gestion des piles) pourraient être appelées comme toute fonction externe si :

- aucune protection câblée n'est envisagée pour protéger le moniteur,
- le module compilé et le moniteur sont connectés ensemble (statiquement).

Dans le cas contraire, un appel sous forme de SVC peut s'avérer nécessaire. Une compilation pourrait ainsi s'accompagner d'une option spéciale pour signaler quel cas est à considérer (appel par SVC ou normal).

En conséquence, à chacune des primitives du moniteur correspond un numéro de SVC prédéfini et connu du compilateur.

Considérons, maintenant, une partie autre que le moniteur qui désirerait se réserver un numéro de SVC pour réaliser un appel, soit :

- au *tronc-commun* pour, par exemple, rajouter une association,
- à une fonction externe d'un de ses modules pour passer en mode maître (lancement d'une entrée/sortie par exemple),
- à une fonction externe d'un des modules d'une autre partie (de durée de vie plus longue).

Une telle demande peut être formulée de manière explicite dans le langage d'écriture qui propose :

- une indication telle que *SVC(n)* lors des déclarations de références externes,
- une notion de point d'entrée ayant pour valeur : le couple d'adresses repérant une fonction externe d'un module.

Par exemple, l'appel faisant référence à :

SVC(1) dummy procédure rajouter (n:integer; pe : point_entrée;
maître : boolean);

permet, lors de la phase d'initialisation d'un système d'exploitation, de rajouter une (ou plusieurs) association entre le numéro *n* et le point d'entrée *pe*. Ainsi à *rajouter (10, SGF.lire, false);* pourrait correspondre :

SVC(10) dummy function lire (<paramètres>) : <résultat> ;

déclaré dans un système chargé par la suite (par exemple pour traiter le travail d'un usager). Mais ceci limite la réutilisation que l'on peut faire du modèle (qui contient une telle déclaration) en le liant à un numéro particulier.

Une solution, plus convenable, est de n'effectuer la liaison au numéro que lors d'une étape de connexion.

Dans le cadre de la programmation modulaire, le mieux est d'ignorer ce problème lors de l'écriture (et de la compilation) et de ne le résoudre entièrement qu'à la connexion. Ainsi une référence externe déclarée par

dummy function lire (<paramètre>) : <résultat>;

ne fait pas de supposition quant au type d'appel utilisé.

Seule une instruction de la forme :

with <*nom module*> *do lire := svc(10) end*

apparaissant dans un programme de connexion fait la différence. La référence fictive *lire* est remplacée en une référence à une procédure rajoutée par le connecteur dans le binaire produit. Cette procédure effectuera l'appel proprement dit par *svc10* et assurera le retour de telle sorte que l'on ait pas à modifier ce qu'a produit le compilateur.

Cette procédure peut même effectuer une liaison définitive de telle sorte que les appels ultérieurs avec *lire* se fassent directement sur la fonction externe associée à *10*. Dans ce cas le mécanisme de SVC (qui est plus lent qu'un appel "normal") ne serait utilisé que la première fois. Ceci n'est possible que lorsque le SVC n'était pas destiné à :

- faire passer en mode maître,
- franchir une "barrière de protection".

5.9. Interface présentée par un moniteur d'exécution

Pour ce qui concerne les parties que nous venons de décrire, un moniteur d'exécution se compose de :

- un module *schéduler*,
- un module *interruptions*,
- quatre procédures *enter*, *exit*, *wait* et *signal*,
- un (ou plusieurs) module *file*,
- un module *allocateur_pile*,
- un module *gestion_processus*,
- un module *tronc_commun*.

et il présente à l'extérieur (au moyen de SVC) les fonctions externes suivantes :

function scheduler.actif : integer % qui donne un entier caractérisant le processus %

function interruptions.recevoir (n, p : integer) : ? % qui permet d'attendre une interruption de niveau n avec la priorité p et d'en déterminer la cause %

function allocateur_pile.fournir (n : integer) : address % qui permet à un processus de réclamer un nouveau bloc de taille n et de récupérer la valeur de sa base %

function allocateur_pile.rendre (base : address) % qui permet à un processus d'enlever le dernier bloc de sa pile %

procédure gestion_processus.créer (d : address; p : integer) % qui permet de créer un nouveau processus sur l'instruction d'adresse d avec la priorité p %

procédure gestion_processus.place_aux_jeunes; % qui permet d'attendre que tous les fils aient terminé %

procédure tronc_commun.rajouter (n : integer; c, p : address; m:boolean) % qui signale que la fonction externe repérée par (c, p) pourra être appelée par la suite par SVC n (en mode maître si m = vrai) %

procédure tronc_commun.retrancher % qui supprime toutes les associations entre numéro de SVC et fonctions externes rajoutées par ce processus %

procédure transmettre (p : integer; t, m : boolean) % qui transmet au processus identifié par p la possibilité
- *de rajouter en mode esclave*
- *de rajouter en mode maître si m = vrai*
- *de transmettre si t = vrai %*

et enfin, les quatre procédures implémentant les moniteurs.

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

[1] BETOURNE C., FERAUD L., JOULIA J., RIGAUD J.M.
Le langage d'écriture de système 'LEST'
Congrès AFCET - Paris (1976)

[2] BRINCH HANSEN P.
The programming language CONCURRENT PASCAL
I.E.E.E. Trans. on software engineering, vol.1, num. 2 (1975)

[3] CRISTIAN F., GUILLOT J.M.
Rapport D.E.A. - Grenoble (1977)

[4] CROCUS
Systèmes d'exploitation des ordinateurs
Dunod, Paris (1975)

[5] HOARE C.A.R.
Monitors : an operating system structuring concept
CACM, vol. 17, num. 10 (1974)

[6] ICHBIAH J.D., HELIARD J.C., RISSEN J.P., COUSOT P.
The System implementation language LIS
Reference manual, CII (1975)

[7] JENSEN K., WIRTH N.
PASCAL - User manual and report
Springer-Verlag (1974)

[8] LISKOV B.H.
An introduction to CLU
Computation structures, Memo n° 133 - MIT (1976)

[9] SCHWAAB M., TISSERANT A.
Implémentation de PASCAL sur miniordinateur T1600
Journées PASCAL, Nice (1975)

[10] Service de synthèse et d'orientation de la recherche en informatique
SFER-PASCAL : Le langage de programmation PASCAL. Système d'exploitation SIRIS 7-8

[11] WIRTH N.
MODULA : a language for modular multiprogramming .
Software-practice and experience, vol. 7 (1977)