Un noyau pour la communication et la synchronisation de processus répartis

Victor German Sanchez Arias

THESE

présentée à

l' Institut National Polytechnique de Grenoble

pour obtenir le grade de
DOCTEUR INGENIEUR

par

Victor German SANCHEZ ARIAS

UN NOYAU POUR LA COMMUNICATION ET LA SYNCHRONISATION

DE PROCESSUS REPARTIS

Thèse soutenue le 30 janvier 1985 devant la commission d'examen

J. MOSSIERE	Président
G. GUILLEMONT	
M. MAZARÉ	Examinateurs
T. MUNTEAN	

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Année universitaire 1982-1983

Président de l'Université : D. BLOCH

Vice-Président : René CARRE
Hervé CHERADAME
Marcel IVANES

PROFESSEURS DES UNIVERSITES :

ANCEAU François	E.N.S.I.M.A.G.
BARRAUD Alain	E.N.S.I.E.G.
BAUDELET Bernard	E.N.S.I.E.G.
BESSON Jean	E.N.S.E.E.G.
BLIMAN Samuel	E.N.S.E.R.G.
BLOCH Daniel	E.N.S.I.E.G.
BOIS Philippe	E.N.S.H.G.
BONNETAIN Lucien	E.N.S.E.E.G.
BONNIER Etienne	E.N.S.E.E.G.
BOUVARD Maurice	E.N.S.H.G.
BRISSONNEAU Pierre	E.N.S.I.E.G.
BUYLE BODIN Maurice	E.N.S.E.R.G.
CAVAIGNAC Jean-François	E.N.S.I.E.G.
CHARTIER Germain	E.N.S.I.E.G.
CHENEVIER Pierre	E.N.S.E.R.G.
CHERADAME Hervé	U.E.R.M.C.P.P.
CHERUY Arlette	E.N.S.I.E.G.
CHIAVERINA Jean	U.E.R.M.C.P.P.
COHEN Joseph	E.N.S.E.R.G.
COUMES André	E.N.S.E.R.G.
DURAND Francis	E.N.S.E.E.G.
DURAND Jean-Louis	E.N.S.I.E.G.
FELICI Noël	E.N.S.I.E.G.
FOULARD Claude	E.N.S.I.E.G.
GENTIL Pierre	E.N.S.E.R.G.
GUERIN Bernard	E.N.S.E.R.G.
GUYOT Pierre	E.N.S.E.E.G.
IVANES Marcel	E.N.S.I.E.G.
JAUSSAUD Pierre	E.N.S.I.E.G.
JOUBERT Jean-Claude	E.N.S.I.E.G.
JOURDAIN Geneviève	E.N.S.I.E.G.
LACOUME Jean-Louis	E.N.S.I.E.G.
LATOMBE Jean-Claude	E.N.S.I.M.A.G.

.../...

LESSIEUR Marcel	E.N.S.H.G.
LESPINARD Georges	E.N.S.H.G.
LONGEQUEUE Jean-Pierre	E.N.S.I.E.G.
MAZARE Guy	E.N.S.I.M.A.G.
MOREAU René	E.N.S.H.G.
MORET Roger	E.N.S.I.E.G.
MOSSIERE Jacques	E.N.S.I.M.A.G.
PARIAUD Jean-Charles	E.N.S.E.E.G.
PAUTHENET René	E.N.S.I.E.G.
PERRET René	E.N.S.I.E.G.
PERRET Robert	E.N.S.I.E.G.
PIAU Jean-Michel	E.N.S.H.G.
POLOUJADOFF Michel	E.N.S.I.E.G.
POUPOT Christian	E.N.S.E.R.G.
RAMEAU Jean-Jacques	E.N.S.E.E.G.
RENAUD Maurice	U.E.R.M.C.P.P.
ROBERT André	U.E.R.M.C.P.P.
ROBERT François	E.N.S.I.M.A.G.
SABONNADIERE Jean-Claude	E.N.S.I.E.G.
SAUCIER Gabrielle	E.N.S.I.M.A.G.
SCHLENKER Claire	E.N.S.I.E.G.
SCHLENKER Michel	E.N.S.I.E.G.
SERMET Pierre	E.N.S.E.R.G.
SILVY Jacques	U.E.R.M.C.P.P.
SOHM Jean-Claude	E.N.S.E.E.G.
SOUQUET Jean-Louis	E.N.S.E.E.G.
VEILLON Gérard	E.N.S.I.M.A.G.
ZADWORNY François	E.N.S.E.R.G.

PROFESSEURS ASSOCIES

BASTIN Georges	E.N.S.H.G.
BERRIL John	E.N.S.H.G.
CARREAU Pierre	E.N.S.H.G.
GANDINI Alessandro	U.E.R.M.C.P.P.
HAYASHI Hirashi	E.N.S.I.E.G.

PROFESSEURS UNIVERSITE DES SCIENCES SOCIALES (Grenoble II)

BOLLIET Louis
Chatelin Françoise

PROFESSEURS E.N.S. Mines de Saint-Etienne

RIEU Jean
SOUSTELLE Michel

CHERCHEURS DU C.N.R.S.

FRUCHART Robert	Directeur de Recherche
VACHAUD Georges	Directeur de Recherche

.../...

ALLIBERT Michel	Maître de Recherche
ANSARA Ibrahim	Maître de Recherche
ARMAND Michel	Maître de Recherche
BINDER Gilbert	
CARRE René	Maître de Recherche
DAVID René	Maître de Recherche
DEPORTES Jacques	
DRIOLE Jean	Maître de Recherche
GIGNOUX Damien	
GIVORD Dominique	
GUELIN Pierre	
HOPFINGER Emil	Maître de Recherche
JOUD Jean-Charles	Maître de Recherche
KAMARINOS Georges	Maître de Recherche
KLEITZ Michel	Maître de Recherche
LANDAU Ioan-Dore	Maître de Recherche
LASJAUNIAS J.C.	
MERMET Jean	Maître de Recherche
MUNIER Jacques	Maître de Recherche
PIAU Monique	
PORTESEIL Jean-Louis	
THOLENCE Jean-Louis	
VERDILLON André	

CHERCHEURS du MINISTERE de la RECHERCHE et de la TECHNOLOGIRE (Directeurs et Maîtres de Recherches, ENS Mines de St. Etienne)

LESBATS Pierre	Directeur de Recherche
BISCONDI Michel	Maître de Recherche
KOBYLANSKI André	Maître de Recherche
LE COZE Jean	Maître de Recherche
LALAUZE René	Maître de Recherche
LANCELOT Francis	Maître de Recherche
THEVENOT François	Maître de Recherche
TRAN MINH Canh	Maître de Recherche

PERSONNALITES HABILITEES à DIRIGER des TRAVAUX de RECHERCHE (Décision du Conseil Scientifique)

ALLIBERT Colette	E.N.S.E.E.G.
BERNARD Claude	E.N.S.E.E.G.
BONNET Rolland	E.N.S.E.E.G.
CAILLET Marcel	E.N.S.E.E.G.
CHATILLON Catherine	E.N.S.E.E.G.
CHATILLON Christian	E.N.S.E.E.G.
COULON Michel	E.N.S.E.E.G.
DIARD Jean-Paul	E.N.S.E.E.G.
EUSTAPOPOULOS Nicolas	E.N.S.E.E.G.
FOSTER Panayotis	E.N.S.E.E.G.

.../...

GALERIE Alain	E.N.S.E.E.G.
HAMMOU Abdelkader	E.N.S.E.E.G.
MALMEJAC Yves	E.N.S.E.E.G. (CENG)
MARTIN GARIN Régina	E.N.S.E.E.G.
NGUYEN TRUONG Bernadette	E.N.S.E.E.G.
RAVAINE Denis	E.N.S.E.E.G.
SAINFORT	E.N.S.E.E.G. (CENG)
SARRAZIN Pierre	E.N.S.E.E.G.
SIMON Jean-Paul	E.N.S.E.E.G.
TOUZAIN Philippe	E.N.S.E.E.G.
URBAIN Georges	E.N.S.E.E.G. (Laboratoire des ultra-réfractaires ODEILLON)
GUILHOT Bernard	E.N.S. Mines Saint Etienne
THOMAS Gérard	E.N.S. Mines Saint Etienne
DRIVER Julien	E.N.S. Mines Saint Etienne
BARIBAUD Michel	E.N.S.E.R.G.
BOREL Joseph	E.N.S.E.R.G.
CHOVET Alain	E.N.S.E.R.G.
CHEHIKIAN Alain	E.N.S.E.R.G.
DOLMAZON Jean-Marc	E.N.S.E.R.G.
HERAULT Jeanny	E.N.S.E.R.G.
MONLLOR Christian	E.N.S.E.R.G.
BORNARD Guy	E.N.S.I.E.G.
DESCHIZEAU Pierre	E.N.S.I.E.G.
GLANGEAUD François	E.N.S.I.E.G.
KOFMAN Walter	E.N.S.I.E.G.
LEJEUNE Gérard	E.N.S.I.E.G.
MAZUER Jean	E.N.S.I.E.G.
PERARD Jacques	E.N.S.I.E.G.
REINISCH Raymond	E.N.S.I.E.G.
ALEMANY Antoine	E.N.S.H.G.
BOIS Daniel	E.N.S.H.G.
DARVE Félix	E.N.S.H.G.
MICHEL Jean-Marie	E.N.S.H.G.
OBLED Charles	E.N.S.H.G.
ROWE Alain	E.N.S.H.G.
VAUCLIN Michel	E.N.S.H.G.
WACK Bernard	E.N.S.H.G.
BERT Didier	E.N.S.I.M.A.G.
CALMET Jacques	E.N.S.I.M.A.G.
COURTIN Jacques	E.N.S.I.M.A.G.
COURTOIS Bernard	E.N.S.I.M.A.G.
DELLA DORA Jean	E.N.S.I.M.A.G.
FONLUPT Jean	E.N.S.I.M.A.G.
SIFAKIS Joseph	E.N.S.I.M.A.G.
CHARUEL Robert	U.E.R.M.C.P.P.
CADET Jean	C.E.N.G.
COEURE Philippe	C.E.N.G. (LETI)

.../...

DELHAYE Jean-Marc	C.E.N.G. (STT)
DUPUY Michel	C.E.N.G. (LETI)
JOUVE Hubert	C.E.N.G. (LETI)
NICOLAU Yvan	C.E.N.G. (LETI)
NIFENECKER Hervé	C.E.N.G.
PERROUD Paul	C.E.N.G.
PEUZIN Jean-Claude	C.E.N.G. (LETI)
TAIEB Maurice	C.E.N.G.
VINCENDON Marc	C.E.N.G.

LABORATOIRES EXTERIEURS

DEMOULIN Eric	C.N.E.T.
DEVINE	C.N.E.T. (R.A.B.)
GERBER Roland	C.N.E.T.
MERCKEL Gérard	C.N.E.T.
PAULEAU Yves	C.N.E.T.
GAUBERT C.	I.N.S.A. Lyon

Je tiens à remercier

Monsieur Jacques MOSSIERE, Directeur du Laboratoire de Génie Informatique, qui m'a fait l'honneur de présider le jury de cette thèse.

Monsieur Marc GUILLEMONT, Responsable du projet CHORUS à l'INRIA, qui a bien voulu accepter de juger ce travail et a contribué par ses remarques à son amélioration.

Monsieur Guy MAZARE, Professeur à l'Institut National Polytechnique de Grenoble, pour la confiance qu'il m'a témoignée en m'accueillant dans son équipe et l'intérêt qu'il a apporté au développement de ce travail.

Monsieur Traian MUNTEAN, Maître-Assistant à l'Université Scientifique et Médicale de Grenoble, pour les idées et les encouragements qu'il m'a prodigués lors de la réalisation de ce travail.

Tous les membres de l'équipe avec qui j'ai pu travailler et échanger des idées, particulièrement Mademoiselle Cécile ROISIN et Monsieur Michel RIVEIL pour leurs lectures et critiques de ce travail.

Tous les membres de l'équipe BULL, pour m'avoir permis d'effectuer la frappe de cette thèse sur leur machine de traitement de textes.

Tous les membres du service de reprographie de l'IMAG qui ont assuré la réalisation matérielle de ce document.

CONACYT pour le soutien financier qui a permis la réalisation de cette thèse.

SOMMAIRE

INTRODUCTION

Grâce à l'évolution de la technologie dans les domaines de la microélectronique et des systèmes de communication (avec l'apparition des microordinateurs de plus en plus puissants et l'apparition des réseaux locaux), l'implémentation des applications réparties est devenue, ces dix derniers années, un problème en soi. Il convient d'y apporter des solutions de plus en plus performantes.

Parallélement à l'évolution du matériel, nous constatons aussi des avances au niveau du logiciel. Il existe une quantité relativement importante (et qui ne cesse d'augmenter) de logiciels pour l'expression de la répartition. On trouve des modèles ou des mécanismes pour l'expression du parallélisme et de la communication à différents niveaux d'implémentation et d'utilisation : des primitives spécifiques des systèmes d'exploitation, des langages séquentiels adaptés à l'expression des applications réparties, des langages parallèles, des langages de spécification et des modèles mathématiques du parallélisme, etc.

En résumé, nous avons, d'une part, des applications dites réparties. -Celles-ci sont d'une grande variété et n'ont pas forcément les mêmes caractéristiques- Et. d'autre part, pour la programmation et la spécification des applications réparties, nous avons des modèles et des mécanismes qui ne sont pas necessairements basés sur les mêmes principes.

Devant cette variété d'applications et d'outils, deux questions se sont posées :

- existe-t-il un type d'application répartie qui englobe toutes les autres?

- existe-t-il un modèle général pour l'expression de la répartition?

Nous considérons que le type d'application a une grande influence sur le choix du modèle à utiliser pour son implémentation.

Nous proposons dans ce travail, un modèle de noyau de communication pour la programmation des applications réparties. Dans ce cadre, nous avons suivi la démarche suivante · à partir d'un modèle théorique de base (CSP de HOARE), et un type d'application, nous avons défini un noyau de communication qui servira de base pour l'expression et la programmation du système réparti pour cette classe d'application.

Quel type d'applications réparties?

Sous le terme répartition nous pouvons classer une grande variété d'applications dites réparties, par exemple :

-Des applications qui demandent seulement l'extension à un environnement réparti du service de partage des ressources centralisé. Pour ce type d'application, il existe des mécanismes qui offrent des solutions relativement simples pour le problème de l'accès à distance (PANZ 82).

-Des applications qui ont besoin d'outils pour exprimer la coopération entre activités qui réalisent une tâche commune. Pour ce type d'application, il faut des mécanismes permettant d'exprimer des dialogues entre les processus et leur synchronisation.

-Des applications où le temps n'est pas un facteur essentiel, et d'autres où le temps de réponse du système peut être critique (GERT 75).

Dans toute cette gamme d'applications, nous nous sommes intéressé plus particulièrement aux applications réparties "temps réel" sur réseau local de type industriel. De manière plus générale, entre dans cette classe tout système réparti pour lequel il est impossible de dissocier traitement de l'information et contraintes de temps critiques.

Quel modèle de base?

Dans la littérature, nous trouvons une grande variété de modèles pour l'expression de la répartition. Parmi tous ces modèles, CSP (HOAR 78) offre la structuration de base claire d'un système sous forme de processus séquentiels communicants par messages et rendez-vous. Ce mécanisme primitif est le plus adéquat pour l'expression de mécanismes plus com-

plexes nécessaires dans les systèmes répartis. CSP est un modèle qui introduit des primitves essentielles pour l'expression du parallélisme, la communication et la synchronisation des processus séquentiels. Basé sur une sémantique mathématique precise, CSP permet de prouver des propriétés des programmes parallèles.

Dans le langage OCCAM (MAY 83, MUNT 83) - une implémentation de CSP réalisée par INMOS (WILS 83a) - l'objet canal a été introduit pour l'expression de la communication et la synchronisation .

Associé à la notion de processus séquentiel, un objet type de communication et synchronisation est, dans notre approche, essentiel pour l'expression de la répartition. Une application répartie sera spécifiée à l'aide de deux objets : l'un pour l'expression des traitements locaux : les processus, et l'autre pour l'expression de leurs interactions : les canaux.

Avec comme objectif d'étude l'application visée, il nous faut distinguer ses caractéristiques propres des caractérisitiques générales au niveau de l'expression de la communication et de la synchronisation. Nous allons alors en analyser les besoins spécifiques. Ensuite, à partir de la notion de canal et d'une généralisation du service de communication qu'offrent quelques modèles existants, nous allons proposer un noyau de communication pour des systèmes répartis temps réel.

Nous présentons dans le premier chapitre, les caractéristiques et les besoins généraux, au niveau de l'expression de la communication et la synchronisation, d'un système temps réel réparti sur réseau local.

Dans le deuxième, à partir de l'analyse de quelques modèles, nous décrirons un service général de communication.

Dans un troisième temps, à partir des besoins du type d'application visé et de notre mécanisme de communication de base, nous définirons le noyau de communication par canaux.

Dans la dernière partie de ce travail, nous présentons une réalisation de ce modèle que nous avons intégré partiellement au système UNIX

(BSD 4.2). Ce choix a été délibéré, et dû à l'absence de caractéristiques "temps réel" d'UNIX, pour montrer le caractère non exclusif du modèle. Dans l'équipe d'autres études et réalisations sont en cours aussi bien sur une famille plus large de systèmes hôtes (iRMX, RTEA, ..) que sur des réseaux locaux spécifiques (applications robotique, voir Thèse DE M. Riveill à paraître).

CHAPITRE I

LES SYSTEMES TEMPS REEL

INTRODUCTION

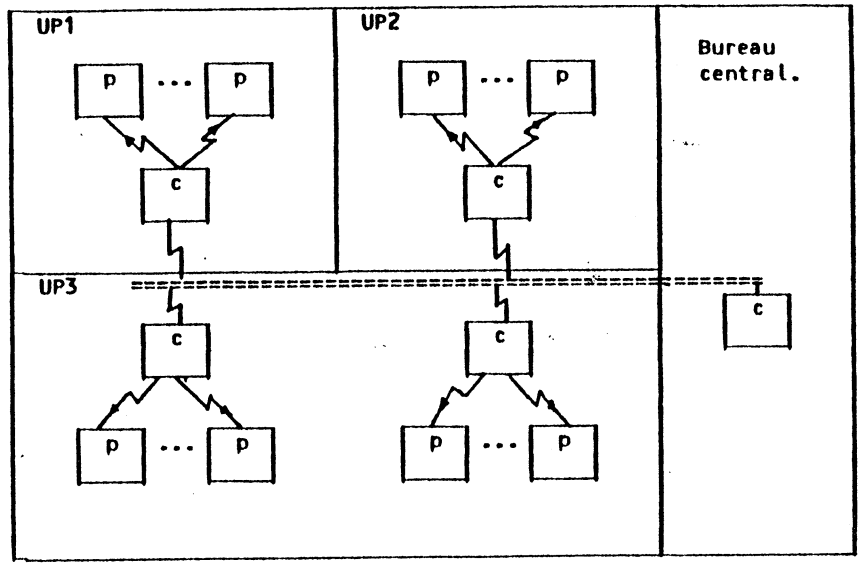
Le terme "temps réel", est une notion qui n'a pas de définition précise dans les systèmes informatiques. Pourtant, il existe des applications où la prise en compte de la notion de temps est un facteur très important.

Dans ce chapitre, nous allons d'abord présenter les caractéristiques générales d'une application "temps réel" et les outils classiques pour leur programmation. Ensuite, nous présenterons les caractéristiques générales d'un système temps réel réparti et nous terminerons par la caractérisation du type d'application visé dans ce travail.

1.1 Les applications temps réel.

Pour une première caractérisation, nous sommes parti de la définition générale donnée dans (YOUN 82) : un système temps réel est défini comme le système qui est capable de répondre à des événements externes dans des délais de temps prédéfinis.

A partir de cette "définition", un grand nombre d'applications peuvent être considérées du type temps réel. Dans cette étude, nous considérons les applications où le temps de réponse est un facteur critique. Une réponse en dehors des délais prédéfinis peut être catastrophique pour le système. On trouve des exemples de ce type d'applications dans les télécommunications, l'industrie, la robotique, etc. Un exemple typique en est le système de contrôle de procédés. Dans la figure I.1.1, on montre schématiquement ce type d'application.



UP unités de production.
p procedé.
c contrôleur.

Fig. I.1.1 Un système temps réel pour le contrôle des procédés.

Le contrôle des procédés est réalisé à travers des interfaces analogiques/digitales. Une horloge permet de tester régulierement les états des procédés. Pour le contrôle manuel, un interface avec l'opérateur

permet des opérations du type initialisation, arrêt, modification etc. Pour le contrôle, le système a besoin de la gestion des informations enregistrées. On peut avoir accès à ces informations à travers des écrans. Pour la nature de l'application, le système est divisé en tâches qui s'exécutent en parallèle et qui échangent continuellement des informations.

A partir de cet exemple, nous pouvons distinguer quatre caractéristiques importantes à considérer dans la conception d'un système temps réel :

- Un système complexe composé de tâches parallèles communicantes ;

- Un environnement externe qui se manifeste par des événements souvent non prévisibles et donc pas toujours "programmables" ;

- Des temps de réponse finis et critiques pour certains types d'événements ;

- L'importance du respect des contraintes de temps associées aux traitements internes du système.

L'ensemble de ces caractéristiques différencient les systèmes temps réel des autres systèmes.

Certes tout système informatique obéit à des contraintes de temps, mais on n'a pas dans tous les systèmes des exigences sévères sur le temps de réponse. Or une réponse en dehors d'un délai prédéfini peut être catastrophique pour une application temps réel. Pour garantir les réponses dans les délais predéfinis, les systèmes temps réel doivent donc offrir des outils pour le contrôle du temps et pour l'ordonnancement des événements.

Un système temps réel est directement lié à des phénomènes physiques, qui sont extrêmement variés et qui ne sont pas des processus programmables. Pour l'interaction avec cet environnement particulier, un système temps réel doit offrir des outils permettant la manipulation d'interfaces analogiques/digitales non standard.

A cause de la nature du type d'application, le système doit être sûr dans son fonctionnement. Une panne dans la communication entre différents

processus de l'application, peut, en effet, mettre en danger tout le système, ansi que son environnement. Pour garantir un bon fonctionnement, un tel système doit en consequence, posséder des mécanismes pour la prévention et la tolérance aux pannes.

I.2 Les outils classiques pour la programmation temps réel.

Pour maîtriser, d'une part, le temps, et d'autre part un environnement composé de processus physiques, il existe des outils spécifiques pour la programmation des applications temps réel (PYLE 80, BARN 80, YOUN 82).

Parmi les plus élémentaires nous trouvons, l'assembleur avec des instructions de contrôle spécifiques d'une machine et les primitives système avec l'utilisation des langages de plus haut niveau. Ces outils, qui ont été les seuls disponibles au début de la programmation temps réel, sont actuellement insuffisants pour l'expression des applications complexes. Le manque de clarté et l'étroite dépendance avec des matériaux spécifiques sont les principaux inconvenients de ce type d'outils.

A un plus haut niveau, nous trouvons des langages adaptés à la programmation temps réel. FORTRAN-Temps Réel est l'exemple classique de ce type d'approche. L'utilisation d'un langage de haut niveau a facilité la programmation et le transport des applications, mais son pouvoir d'expression reste toujours limité pour ce qui est de la programmation des applications complexes.

A un niveau encore plus élevé, nous trouvons des langages conçus pour la programmation temps réel. RTL/2, MODULA et ADA sont des exemples de cette autre génération de langages. Ces langages on profité des constructions des langages structurés (ALGOL, SIMULA, PASCAL) et introduit des concepts plus cohérents pour la programmation temps réel. En général la programmation avec ces langages, est plus simple et plus claire mais par contre ceux-ci ne sont pas toujours bien adaptés ni pour l'expression des concepts spécifiques à une application ni pour l'optimisation d'une implémentation.

Au même niveau que les langages précédents, nous trouvons des langages spécialisés. Ces langages rendent de grands services au niveau de la programmation et de l'optimisation d'une implémentation pour le type d'application vers lequel ils sont orientés. Mais, évidemment, ces langages ne peuvent pas être utilisés pour d'autres types d'applications.

Pour l'expression des applications temps réel complexes (applications composées d'un grand nombre d'activités parallèles avec nombreux échanges d'informations) et pour faciliter les preuves dans la spécification et la programmation de l'application, il faut que dans les langages temps réel les concepts de parallèlisme, communication et synchronisation soient clairs et indépendants des configurations de matériels utilisés.

I.3 La répartition dans les systèmes temps réel.

Les derniers progrés de la technologie, dans la microélectronique et les réseaux locaux, ont élargi le domaine d'applications des systèmes temps réel (ANCE 80).

Avec les coûts actuels, on peut tout à fait envisager l'utilisation d'un ensemble de machines reliées à un réseau local, comme support matériel pour une application répartie temps réel (WOOD 80). Evidemment, l'introduction de ces nouveaux matériels a une conséquence dans la conception d'un système temps réel. On devra en effet, mettre en oeuvre un autre facteur, celui d'un conept de répartition à maîtrisé dans la nouvelle génération de systèmes. La définition de ces nouveaux systèmes, à cause des contraintes sévères pesant sur le temps de réponse, ne pourra être conçue sans une architecture matérielle-logicielle claire et sans une vision globale des applications dans toutes les phases de leur implémentation : spécification, programmation et mise au point.

Doivent être considérés dans la définition d'un système réparti temps réel les facteur suivantes :

- -L'architecture du système de communication (type de réseau, station de transport,..) ;
- -L'architecture de la machine cible (monoprocesseur, multiprocesseur, etc.) ;
- -L'architecture du système d'exploitation ;
- -Les concepts généraux des langages de programmation et de spécification (parallélisme, communication, synchronisation) ; et
- -Les caractéristiques propres à chaque application.

I.4 Les caractérisitiques d'un système temps réel réparti sur réseau local.

Les domaines des applications temps réel sont de plus en plus nombreux et variés. Nous pouvons citer quelques domaines d'applications essentiels : le contrôle de procédés industriels, la télétransmission, la gestion, les systèmes embarqués, etc. Dans cette grande variété d'applications, on peut retrouver les caractéristiques générales d'une application temps réel (temps de réponse, environnement et sûreté), mais chacune a des caractéristiques et des exigences particulières qui ne peuvent past être négiglées lors de sa conception.

A cause de la grande varieté d'applications, il est difficile d'envisager des systèmes généraux. Chaque type d'application a des besoins spécifiques qui doivent être considerés dans la définition d'un système (YOUN 82, LANN 83). Par contre, il faut tenir en compte que le parallélisme, la communication, la synchronisation et le contrôle de temps sont des concepts fondamentaux et généraux pour la spécification et la programmation de toute application temps réel.

Pour la conception d'un système, il faut donc considérer, d'une part, l'aspect spécifique du type d'application et d'autre part, l'aspect général des concepts communication et synchronisation des processus parallèles.

Dans la suite, nous allons préciser les caractéristiques générales du type d'application qui nous servira de base pour la définition de notre modèle.

Un système complexe pour le contrôle de procédés industriels (WOOD 80) est composé d'un ensemble d'unités de production, qui sont réparties géographiquement. Chaque unité de production est composée à la fois d'un ensemble de processus physiques qui sont continuement surveillés et contrôlés par un ensemble de contrôleurs.

La figure I.4 dessine un système complexe pour le contrôle de procedés.

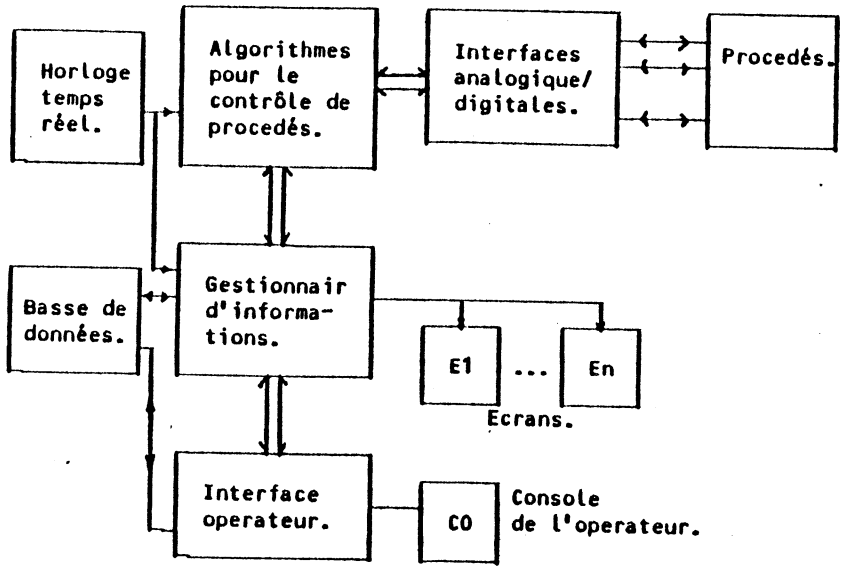


Fig. I.4 Un système pour le contrôle de procédés industriels.

Pour assurer le contrôle sur une unité de production, ces contrôleurs sont connectés à un système de communication. Et pour le contrôle de tout le système, toutes les unités de production sont reliées à un système de communication.

Pendant le fonctionement du système, on a continuellement des transferts d'informations (messures, signaux, informations de contrôle, etc.).

Ces informations peuvent être classées selnos différents niveaux :

- Au niveau le plus bas, on trouve les informations qui permettent le contrôle local d'un ensemble de processus physiques surveillés par un contrôleur,
- Au deuxième niveau, les informations qui permettent de coordonner tous les contrôleurs d'une unité de production,
- Enfin, au troisième niveau, les informations qui permettent la coordination et la gestion de tout le système.

La nature de l'information et le temps de réponse dépendent du niveau de communication.

Au niveau le plus bas, on a surtout des informations courtes (signaux, événements, informations de contrôle, etc.) et des temps de réponse courts et critiques (le contrôle des procédés physiques est ici

réalisé). A ce niveau on a des interfaces d'entrée-sortie non standard pour la surveillance et le contrôle des processus physiques (capteurs, interfaces analogique/digitales, etc).

Par contre aux niveaux plus élevés, où les tâches sont principalement la coordination et la gestion, les informations sont plus longues et le temps de réponse moins critiques. On a ici, pour la gestion et la coordination, des interfaces d'entrée-sortie classiques (disques, terminaux, imprimentes, etc).

Même si la description précédente est très générale, nous pouvons déjà souligner quelques points importants à considérer dans la définition d'une architecture.

Nous classerons ces points dans trois niveaux : système de communication, système d'exploitation et langage.

I.4.1 Niveau système de communication.

Vu la nature répartie de l'application, la solution qui nous semble plus adaptée au type d'application est l'utilisation d'un réseau de microordinateurs et miniordinateurs réliés à un réseau local comme support de communication Les microordinateurs avec des interfaces spécialisés permettront les contrôles locaux et le réseau le contrôle global. Pour assurer le temps de réponse fixé par l'application, le réseau doit, d'une part, permettre l'émission de messages prioritaires et d'autre part, garantir un temps de transmission borné. Pour donner une flexibilité au niveau de l'implémentation, nous considérons qu'une telle application nécessite en général un réseau hétérogène d'ordinateurs.

I.4.2 Niveau système d'exploitation.

Il faut préciser, tout d'abord, que dans ce cadre, la communication locale (pour les contrôles locaux) et la communication externe (pour le contrôle global) doivent être des activités fondamentales du système d'exploitation.

Le système devra offrir les caractérisitiques suivantes pour répondre à certaines tâches :

-Pour ce qui est de l'interaction avec les processus physiques, compter des outils puissants pour la programmation d'interfaces non standard et compter aussi des outils pour la programmation d'interfaces standard pour la manipulation des terminaux, imprimantes, disques, etc.
-Afin d'assurer les temps de réponse imposés par l'application, se prévaloir des outils de base pour la manipulation du temps et pour l'exécution "immédiate" des activités critiques.
-Pour l'optimisation de la mémoire central, à cause du faible espace mémoire des microordinateurs, le système ne doit pas être trop complexe.
-Pour la programmation de strucutres de communication complexes et pour faciliter l'implémentation, les outils de communication devront être simples.
-Etant donné le caractère hétérogène du réseau, le système d'exploitation, doit être transportable et modulaire.

I.4.3. Niveau langage.

Pour la spécification et la programmation d'une application répartie temps réel, deux niveaux d'expression sont nécessaires : un pour l'expression général de la répartition (parallélisme, communication et synchronisation) et un autre pour exprimer les besoins spécifiques du type d'application visé.

Pour la programmation et la spécification générale d'une application répartie les concepts de parallélisme, communication et synchronisation doivent être clairs et indépendants du matériel.

Par contre, pour la optimisation d'une application particulière, il faut pouvoir compter avec des outils proches aux matériaux.

Nous considérons que la programmation tant au niveau général (répartition des activités) et qu'au niveau spécifique (programmation d'interfaces non standard, de différents protocoles, etc) doivent être basées sur les mêmes principes de communication et de synchronisation. Cette unité dans la communication est fondamentale pour la spécification correcte et la preuve des applications.

I.5 Les caractéristiques d'un noyau de communcation pour systèmes temps réel réparti sur réseau local.

Dans le paragraphe précédent, nous avons donné les caractéristiques générales d'une architecture correspondant à un type particulier d'application. Dans la suite de ce travail, nous allons présenter uniquement la partie concernant la définition d'un noyau du système d'exploitation.

Du niveau inférieur, le système réseau de communication, nous retenons les éléments suivants : un réseau local de sites hétérogènes et de faible capacité de mémoire. La modularité et la transportabilité seront donc des points importante à considérer dans la définition de notre modèle.

Du niveau supérieur, c'est à dire le langage de programmation des applications, les facteurs importants à considérer sont: a) un mécanisme de communication et de synchronisation de base, simple, clair et puissant pour une expression propre de la répartition et b) en même temps un mécanisme souple pour son adaptation aux besoins de communication spécifiques d'une application.

Nous résumons ici les points que nous considérons importants pour la conception d'un noyau d'un système temps réel réparti sur réseau local :

- Répartition géographique de l'application.
- Haut degré de parallélisme dans l'exécution des activités.
- Haut degré de communication. Messages de petite et moyenne taille.
- Réponses "immédiates" pour certains types de messages (événements).
- Manipulation du temps.
- Manipulation d'interfaces non standard.
- Comportement dynamique du système (création et destruction dynamique des tâches).
- Sûreté dans le fonctionnement.
- Réseau local hétérogène avec des sites avec faible capacité en mémoire.

CHAPITRE II

CARACTERISATION DES SERVICES DE COMMUNICATION A PARTIR DE DIFFERENTS MODELES EXISTANTS

INTRODUCTION.

Dans le cas de l'implémentation des applications réparties, on trouve une grande variété d'outils pour l'expression du parallélisme, la communication et la synchronisation des processus. Ces outils vont des primitives élémentaires et spécifiques d'une machine aux outils langage de haut niveau orientés vers la programmation parallèle.

Avec l'objectif de définir un service général pour la communication et la synchronisation des processus parallèles, dans ce chapitre, nous allons d'abord faire une analyse de quelques modèles et mécanismes existants. Nous définirons ensuite un modèle abstrait de communication de façon à rendre ce service indépendant des implémentations.

II.1 Les services de communication dans différents modèles.

Dans le but d'avoir une caractérisation la plus générale possible des mécanismes de communication/synchronisation, nous avons fait abstraction de leurs niveaux d'implémentation et d'utilisation (niveau système, niveau langage, niveau spécification). Nous avons choisi, parmi la grande variété des mécanismes existants ceux qui nous ont semblés le plus représentatifs. Sans vouloir être exhaustif dans notre sélection, nous avons considéré dès mécanismes très élémentaires et spécifiques (mais en même temps très utilisés) jusqu'à des modèles théoriques de haut niveau.

Il faut remarquer que cette analyse a été centrée sur les mécanismes de communication/synchronisation basés sur l'échange de messages et que l'on n'a pas pris en compte toutes les autres facilités offertes par les modèles.

Les différents modèles choisis sont décrits dans les paragraphes suivants. La présentation des modèles et des mécanismes suit l'ordre indiqué : Nous débutons par la description des mécanismes de quelques systèmes commercialisés et d'utilisation courante (RSX-11M, iRMX et UNIX). Nous analysons ensuité différentes propositions de langages pour la programmation des systèmes répartis basés sur le protocole de rendez-vous de CSP (Pascal-CSP et Pascal-M). Nous présentons après un système d'exploitation réparti (CHORUS) et nous terminons par un modèle de communication d'un langage de spécification proposé dans (BOCH 83).

II.1.1 RSX-11M (DIGI 79).

RSX-11M est un système d'exploitation, qui a été développé par la DIGITAL EQUIPMENT CORPORATION pour les machines PDP11.

Le système RSX-11M offre pour la programmation temps réel : un répartiteur basé sur des priorités disponibles aux utilisateurs ; des primitives pour la création et la destruction des tâches ; des primitives pour le traitement des événements (le "system traps") : des primitives pour la manipulation du temps : des primitives pour la manipulation de l'espace d'adressage virtuel et logique : des primitives pour la communication par messages basées sur des boîtes à lettres : etc.

RSX-11M offre trois mécanismes de communication et synchronisation : un système de communication par messages, un système pour le traitement des événements et un système pour le partage de la mémoire.

Le système de communication permet le transfert de messages (de 13 mots) entre deux processus à travers des boîtes à lettres. Une boîte à lettres est un buffer où les messages sont enregistrés par ordre d'arrivée. La création de boîtes à lettres est réalisée implicitement par le système lors de la création du processus (une seule boîte à lettres est associée automatiquement à tout processus lors de sa création). L'accès à une boîte à lettres est réalisé à travers le nom du processus propiétaire. L'envoi d'un message est réalisé par la primitive SEND DATA. Une fois le message copie dans la boîte à lettres du processus récepteur, le processus émetteur continue son exécution. Un processus lit sa boîte à lettres avec la primitive RCVE DATA. Le processus récepteur aura en plus du message émis, l'identification du processus émetteur (2 mots). Un processus peut demander l'interruption de son exécution dès le chaînage d'un message dans sa boîte à lettres (primitive SPECIFY RCVE DATA AST). Le système garantit l'interruptibilité dans les séquences d'interruption (prévues pour cet événement) jusqu'à l'appel de la primitive AST SERVICE EXIT.

Les événements permettent la synchronisation soit entre processus, soit à l'intérieur d'un procesus. Le système offre 32 événements locaux et 32 globaux partagés entre tous les processus. Les primitives systèmes pour la manipulation des événements sont : SET EVENT FLAG pour poster un événement, CLEAR EVENT FLAG pour nettoyer un événement, WAIT pour attendre un ou plusieurs événements. Pour le traitement des interruptions, le système offre, associe aux événements, deux types de services · les SSTs ("synchronous systèm traps") pour le traitement des erreurs de programmation ou du système et les ASTs ("asynchronous système traps") pour le traitement des événements asynchrones (interruptions externes, événements logiques provoqués par le programmeur). Le système de "traps" permet au programmeur d'établir des points d'entrée pour des routines (écrites par l'utilisateur) pour le traitement d'interruptions.

Nous pouvons remarquer que ce mécanisme de communication est très élémentaire, mais qu'il a l'intérêt d'être un système simple au niveau de l'implémentation (absence d'un gestionnaire de boîtes à lettres). Par contre, la programmation de structures de synchronisation complexes à partir

de ses boîtes à lettres est pénible, alors que pour des applications peu complexes, ces boîtes à lettres suffisents comme mécanismes.

II.1.2 iRMX86 (INTE).

iRMX86 est un système d'exploitation pour les microrordinateurs de INTEL.

Le noyau d'iRMX86 a comme fonctions : le cadencement des processus (le répartiteur est basé sur des priorités disponibles aux utilisateurs), le contrôle de l'accès aux ressources du système, le contrôle de la communication des processus et le contrôle des événements. Pour la programmation, il offre aux utilisateurs les objets suivants : des tâches ("tasks") pour la programmation des processus séquentiels : des "boîtes à lettres ("mailbox") pour la communication par messages · des sémaphores pour la synchronisation des processus · des segments pour la gestion de la mémoire: des régions pour la communication par mémoire commune et les "jobs" pour le regroupement des objets (tâches, boîtes à lettres, sémaphores, etc.).

iRMX offre des primitives pour la création et la destruction dynamique de ses objets (tâches, boîtes à lettres, etc.) à travers des primitives CREATE$ objet,.. et DELET$ objet,.. Lors de la création de l'objet le système associé à celui-ci un identificateur unique ("token"). Grâce au "token" les tâches peuvent y acceder.

Une boîte à lettres permet l'émission et la réception des identificateurs des objets d'un processus (par exemple l'identificateur d'un segment de données pour l'émission et la réception d'un message). Chaque mailbox a deux queues associées, une, pour les tâches en attente de réception d'un objet et une deuxième pour les objets reçus mais pas encore lus. La queue d'objets est gérés en FIFO tandis que la queue de tâches peut être gérée soit en FIFO soit par priorités (le choix est fait au moment de la création). La queue d'objets est implémentée par deux types de buffers, l'un rapide et paramètrisable par l'utilisateur (entre 4 et 60 objets) l'autre plus lent seulement limité par la taille de mémoire disponible.

Une tâche envoie le "token" d'un objet avec la primitive SEND$MSG. S'il n'y a pas de tâches en attente, le "token" est copié dans la queue d'objets, sinon, le message est remis à la tâche réceptrice (la première

dans la queue ou la prioritaire selon la politique choisie). Un tâche reçoit un token d'une boîte à lettres avec la primitive RECV$MSG. Si la queue d'objets n'est pas vide la tâche reçoit "immediatement" le token, dans le cas contraire, la tâche réceptrice a l'option d'attendre l'arrivée d'un token (jusqu'à la fin d'un délai predéfini) ou de recevoir immediatement un avis de queue vide.

Une tâche émettrice a la possibilité par l'envoi d'un message, de demander un accusé de réception à la tâche réceptrice. La tâche réceptrice doit vérifier dans le message reçu la demande de ce service.

Indépendament des boîtes à lettres, iRMX offre un objet pour la synchronisation des tâches, les sémaphores.

Un sémaphore est composé d'une seule queue, la queue de tâches, qui peut être géré en FIFO ou par priorités (choix fait à la création). Un sémaphore peut être créé et détruit comme tout objet du système. Pour la manipulation d'un semaphore, le système offre deux primitives, SEND$UNITS pour l'addition d'un nombre d'unités et la primitive RECV$UNITS pour la demande d'un nombre d'unités. La lecture d'un sémaphore peut être temporisée.

Cette approche est intéressante par la simplicité de son noyau, et par le service de son système de communication plus élaboré que celui de RSX-11M. La création et la destruction dynamique des boîtes à lettres et sémaphores et la paramètrisation de leurs comportements donnent une flexibilité pour la programmation de structures plus complexes. Par contre, il est évident que la gestion des objets réalisée par le noyau iRMX est plus complexe que la gestion des boîtes à lettres de RSX-11M.

La programmation des applications temps réel est possible grâce aux services offerts par le noyau (mécanisme de priorités, création dynamiques des tâches, primitives pour la manipulation du temps, système pour le traitement des événements, etc).

La spécificité et la simplicité des mécanismes du système iRMX86 (d'ailleurs comme dans RSX-11M), représentent à la fois ses vantages et ses inconvénients. L'optimisation d'une programmation est possible grâce aux mécanismes de très bas niveau, mais par contre au niveau du transport et de la clarté dans la programmation, ces mécanismes présentent des limitations.

II.1.3 UNIX (RITC74).

UNIX est un système d 'exploitation développé par les labarotoires de la Bell.

UNIX est un système d'exploitation qui a été conçu pour des applications du type interactif temps partagé. Son répartiteur à été conçu pour être le plus équitable possible dans l'utilisation du CPU. Pour la multiprogrammation, il offre des primitives pour la création/destruction de processus, pour la communication par messages (les "pipes") et pour le traitement des événements (les signaux).

Un "pipe" est un espace de mémoire où des processus écrivants peuvent enregistrer des chaînes d'octets et des processus lecteurs peuvent lire des chaines d'octets (le nombre d'octets lus peut être différent du nombre d'octets écrits). Les chaînes dans un "pipe" sont enregistrés par ordre d'arrivée. Un "pipe" a un identificateur unique qui permet d'y acceder. La création est dynamique (primitive PIPE) ; par contre la destruction est implicite au moment de la termination du processus. On transmete l'identificateur d'un "pipe" d'un processus à un autre uniquement à travers la primitive de création de processus (FORK), un processus transmet implicitiment tous les identificateurs des "pipes" créés avant de l'exécution du FORK au processus crée, la communication est donc restrainte aux processus qui ont la même origine.

Les fonctions d'accès sont les mêmes que celles utilisées pour les fichiers ("read" et "write"). La lecture d'un "pipe" est bloquant s'il est vide et l'écriture est bloquante s'il est plein.

Pour le traitement des événements, UNIX offre les primitives systèmes suivantes : SIGNAL pour associer un des événements asynchrones prédéfinis (interruption, instruction illégale, erreur dans le bus, violation de segment, etc.) à une procédure de traitement ; KILL pour provoquer un événement logique par un processus. Le nombre d'événements logiques est réduit. Un nouveau événement efface le dernier s'il n'a pas été traité. Les mécanismes correspondants des deux autres systèmes (RSX-11M et iRMX) sont plus puissants.

Le système de communication est simple, la manipulation des "pipes" ne demandent que l'utilisation de trois primitives : PIPE pour la création, READ pour la lecture et WRITE pour l'écriture. Par contre son accès hiérarchisé et 'absence de la notion de message ne facilitent pas la programmation de structures plus complexes.

UNIX est un système conçu pour les applications temps partagé et il n'est manifettement pas adapté aux applications temps réel (un processus ne peut pas bloquer indéfiniment un autre processus en mémoire et obtenir une priorité garantissant une attribution rapide du processur). Mais il faut remarquer que depuis la définition du sytème original, plusieurs versions sont apparues et elles ont amélioré et introduit de nouveaux mécanismes (Berkeley, V7, Sytsème III, etc.). Il faut dire aussi que les pipes ont exercé une grande influence dans la définition de nouveaux mécanismes de communication.

II.1.4 UNIX-Distributed PULSE(WELL81).

UNIX-Distributed est un système d'exploitation réparti, qui a été développé à l'Université de York en Angleterre.

Ce système est un exemple des nombreuses extensions qui ont été proposées pour l'amélioration du système de communication d'UNIX. Avec l'objectif d'adapter UNIX aux applications réparties, le noyau a été rédéfini et un nouvel objet de communication avec ses primitives d'accès a été introduit.

Le noyau de ce système (le PULSE kernel), est basé sur la notion de tâche du type ADA et d'un mécanisme de communication par rendez-vous. Les objets de communication dénommés "mediums" sont gérés par le noyau et sont impléments comme une tâche-buffer unidirectionel qui mémorise dans une queue les messages provenant de plusieurs processus écrivains et qui transmet à un seul processus lecteur. Un "medium" est une tâche qui a quatre points d'entrée : pour l'émission d'un message (SEND_MESSAGE), pour la réception d'un message (RECEIVE_MESSAGE), pour l'émission d'un message prioritaire (EMERGENCY_SEND) et pour le status du medium (STATUS). Les

"mediums" sont protégés par des capacités. La gestion des capacités (propiétaire, lecture et écriture) est la responsabilité du programmeur. Trois types de messages sont gérés par un medium : les messages de contrôle pour modifier le status du medium, les messages de type rejet pour le traitement des exceptions réseau (exceptions du type ADA) et les messages de données (registres typés). Tout message a un champ dénommé "medium de réponse" qui permet aux processus émetteurs de signaler aux processus récepteurs le "medium" pour une réponse éventuelle.

Les processus peuvent demander la création, la destruction et la transmission des capacités des mediums. Les opérations de lecture et d'écriture sont réalisés par un mécanisme de rendez-vous du type ADA.

Pour les événements asynchrones, PULSE offre un mécanisme basé sur le traitement d'exceptions d'ADA. Un événement réseau est consideré comme un message de rejet sur un medium (interruption externe, mort du medium, mort du lecteur, message de rejet d'un processus écrivain, etc).

L'intérêt principal de cette approche provient de l'unification des deux concepts (événements et communication) dans un même objet unique, ainsi que de l'enrichissement de l'objet de communication (il n'est plus un objet passif, il a le comportement d'un processus). Il provient aussi de l'utilisation d'un système existant et connu, UNIX. Par contre, les applications temps réel ne sont pas prévues et la répartition des mediums est en cours de définition. Notre modèle est voisin dans sa structure de l'approche proposée dans PULSE.

II.1.5 CSP (HOAR78).

CSP est un langage pour la programmation parallèle qui a été développé à l'université The Queen's University de Belfast Northen Ireland.

A partir d'un petit ensemble de commandes, CSP offre un langage puissant pour l'expression du parallélisme, de la communication et de la synchronisation des processus séquentiels. Ses caractéristiques principales sont : l'utilisation des commandes gardées de Dijkstra (DIJK 76) pour l'expression du non-déterminisme ; l'introduction de la commande parallèle pour l'exécution "simultanée" des processus séquentiels ; la définition des opérations d'entrée et de sortie synchrones comme seul moyen de commu-

nication et de synchronisation des processus.

Une communication entre deux processus a lieu quand "simultanement" le premier exécute une commande de sortie avec le nom du deuxième comme processus destinataire et que le deuxième exécute une commande d'entrée avec le nom du premier comme processus source.

Il s'agit d'un mécanisme de communication simple : une communication symétrique, synchrone et unidirectional (point à point) avec une transmission sans bufferisation du message.

La notion du non-déterminisme a été introduite par les commandes gardées (DIJK 75). Avec l'utilisation des commandes d'entrée dans les commandes gardées en CSP, une nouvelle notion de non-détermisme a été introduite. Avec l'utilisation de ces commandes dans une commande d'alternative, on peut modeliser les interactions des processus avec des environnements non maitrisés ou non contrôlables au moment de l'écriture des programmes.

CSP, en faisant abstraction de toute implémentation, permet une expression claire du parallélisme, et par sa définition sémantique formelle, permet des preuves de programmes parallèles.

Comme CSP est un langage, le transport du modèle est possible. Le seul problème à résoudre se situe au niveau de son implementation.

Comme la notion de temps n'a pas été prévue dans sa version originale, CSP ne peut pas être utilisé pour des applications temps réel.

II.1.6 OCCAM (MAY 83).

OCCAM est un langage issu de CSP et développé par INMOS en Angleterre.

Pour l'implémentation du langage parallèle CSP, OCCAM a introduit la notion canal comme élément de synchronisation de la fonction de communication. Deux processus qui désirent communiquer entre eux ne se désignent pas par leurs noms mais utilisent un même nom canal. La fonction de communication, comme dans CSP, est "simultanée" et aucune "bufferisation" est réalisée dans le canal.

La création et la liaison des canaux aux processus s'effectuent au moment de la compilation du programme. Avant l'exécution d'un ensemble de

processus parallèles (par la commande parallèle), on doit déclarer quels canaux on va à utiliser. Le compilateur se charge de la validation de l'utilisation et des connexions des canaux (un canal permet la communication unidirectionnel entre deux processus ; et il ne peut être re-utilisé qu'à la fin de l'exécution de la commande parallèle).

L'intérêt d'OCCAM par rapport à la syntaxe de CSP est, à travers l'introduction de la notion canal, de rendre plus modulaire la programmation des processus avec une gestion de noms qui reste locale.

Etant donné la présence de la notion de temps dans OCCAM, on peut envisager son utilisation pour la programmation temps réel.

OCCAM est actuellement comercialisé et tourne sur plusieurs machines.

II.1.7 Pascal-CSP (ADAM81).

Cette extension du langage Pascal a été proposée à l'université Claude Bernard (MIAG) de Villeurbanne, France.

Cette approche est un exemple de l'influence de CSP dans la définition ou l'extension du concept de répartition sur un langage séquentiel existant.

Pour la défintion du mécanisme de communication de ce modèle, les notions suivantes, inspirées des commandes de CSP, ont été introduites : un objet de communication appelé "channel" et le rendez-vous comme protocole de communication.

La communication entre deux processus est réalisée par l'intermédiaire d'un "channel". Le synchronisation d'une communication est donc réalisée à l'aide de cet objet de communication. A la différence de la communication point à point de CSP, un "channel" permet la diffusion des messages (un processus émetteur sera bloqué jusqu'à ce que tous les processus récepteurs connectés au canal aient reçu le message). La liaison des "chanels" et des processus est réalisée statiquement au moment de la compilation du programme.

En plus des commandes de parallélisme et de non-déterminisme caractéristiques de CSP, on trouve des commandes pour le renommage des canaux (channel relabeling operator) et pour l'"encapsulation" des "channels" (hiding operator).

L'intérêt de ce type d'approche, nous le trouvons dans l'utilisation des concepts de haut niveau du parallélisme en profitant en même temps d'un langage séquentiel existant (Pascal). Au niveau de l'implémentation de ce modèle, il n'y a rien de prévu encore et la notion de temps n'est pas considerée dans sa conception originale.

II.1.8 Pascal-M (ABRA81).

Cette extension du langage Pascal a été proposé par le labarotoire Computer Systems Laboratory de l'université Queen Mary Colleg University of London.

Cette approche est un autre exemple de l'adaptation d'un langage séquentiel à un environnement réparti.

Dans le but de rendre réparti le langage séquentiel Pascal, trois types d'objets ont été introduits : les processus pour le traitement séquentiel ; les "mailbox" pour la communication par messages ; les modules pour la structuration d'un programme (interconnexion initiale de processus et de "mailbox"). De CSP, on a conservé les concepts de gardes pour l'expression du non-déterminisme et pour la communication, on utilise les "mailbox" pour la synchronisation des opérations d'entrée et de sortie. La communication est synchrone entre deux processus qui communiquent à travers un mailbox, mais à la différence des canaux d'OCCAM, un "mailbox" est un objet typé.

Un "mailbox" peut être connecté à plusieurs processus lecteurs et écrivains et il est caractérisé par le type de ses messages et par un identificateur unique qui peut être transmis dans un message. Cette dernière caractéristique permet aux processus des modifications dans les connexions initiales définies dans un module.

L'intérêt de cette approche comme dans la précédente, réside dans l'adaptation d'un langage séquentiel connu (PASCAL) à un environnement parallèle par l'introduction des notions de parallélisme d'un langage formel (CSP). A différence de l'approche précedente, Pascal-M offre une flexibilité dans la connexion de ses objets de communication (on peut changer les liaisons de façon dynamique).

La notion de temps n'est pas prévue dans ce modèle. Il existe une implémentation sur une machine monoprocesseur, INTEL 8086 et des versions pour des environnemnts répartis ont été prevues.

II.1.9 CHORUS (BANI80).

CHORUS est un système d'exploitation réparti qui a été défini à l'INRIA et qui a été implanté sur un INTEL 8086 sur le système Pascal UCSD (GUIL82).

CHORUS est un système qui a été conçu pour des applications réparties. Son noyau est basé sur l'échange de messages. Pour la programmation répartie, CHORUS offre aux programmeurs quatre types d'objets : les acteurs, pour le traitement séquentiel ; les messages, comme unité de synchronisation et d'échange ; les portes, comme objets de communication et les objets locaux gérés par les acteurs.

Un acteur est l'unité d'exécution locale et il est structuré en étapes de traitement qui sont déclenchées sur réception d'un message. Les portes permettent l'interface de communication entre les acteurs. Les portes sont créées et détruites dynamiquement et chaque porte a un identificateur unique. Pour le traitement des messages, une porte est associée à une étape de traitement (l'association peut être changée dynamiquement). Une communication entre deux processus a lieu, si un message est émis d'une des portes de l'acteur émetteur à une des portes de l'acteur récepteur. Les messages reçus par un acteur sont mis en file d'attente derrière leurs portes réceptrices respectives. Après la réception d'un message, une étape de traitement est declenchée si elle a été associée à la porte réceptrice correspondante et si elle a été selectionée explicitement par l'acteur. La selection des portes pour le traitement des messages (par les unités de traitement correspondantes), est réalisée à la fin du traitement d'une étape. Avec ce mécanisme, un acteur peut influencer dynamiqement l'exécution de ses étapes de traitement. Un acteur peut fixer une durée limite à l'attente d'un message sur une de ses portes. Une fois dépasée la limite, l'acteur est reactivé sur un message de contrôle.

Pour les événement externes (interruptions externes et opérations d'entrée et de sortie) CHORUS utilise le même mécanisme de base. Les ports externes PI d'un site sont considerés comme des portes CHORUS, une interruption esterne est transformée en une opération d'émission de message du port PI à la porte P de l'acteur qui traite l'interruption. Les périphériques sont aussi considérés comme des portes gérées par un acteur chargé de cette tâche.

Cette approche a l'intérêt d'être un système défini dès le début pour les applications réparties. Dans son noyau, le minimum d'activités ont été conservées (essentiellement le cadencement et la communication locale) et sa spécification a été faite indépendament des implémentations. La simplicité de son noyau et son independance par rapport aux implémentations font de CHORUS un système facilement transportable. Son mécanisme de communication est simple et puissant et avec son nombre réduit de primitives, on peut exprimer des structures de communication plus complexes.

CHORUS n'a pas été défini pour un type d'application particulier, c'est un système général. Actuallement, une version sur une machine multiprocesseur (SM90) est en cours de réalisation.

II.1.10 SSCS : Structured Specification of Communicating System (BOCH83).

SSCS est un modèle de communication pour la spécification de systèmes répartis.

A partir d'une méthode de spécification basée sur la définition de deux concepts, les processus et leurs interactions, deux objets ont été définis dans ce modèle : les processus pour le traitement séquentiel et les ports pour le traitement de la communication. Pour la spécification d'un système réparti, l'utilisateur doit spécifier les propriétés des processus et des ports. Pour la spécification d'un port, on doit énumérer les interactions possibles sur le port (par exemple lecture et écriture) et l'ordre d'exécution de ses interactions. La spécification d'un processus consiste en la spécification des propiétés des ses ports (comme elles ont été décrites auparavant) et des propriétés propres au processus (par exemple les relations entre les interactions et les ports). Une interconnexion entre processus est établie en connectant leurs ports. Une interaction a lieu quand des processus invoquent le même type d'interaction sur les ports interconnectés.

L'objet de communication de cette approche est un outil puissant pour la spécification de la communication et de la synchronisation des processus séquentiels. Un port n'est pas associé à des fonctions d'accès spécifiques et le modèle offre une grande flexibilité pour exprimer des contraintes d'accès par la définition d'un ordre dans l'exécution de ses

fonctions. Il faut dire que l'objectif de cette approche a été d'offrir un outil pour la spécification et non un outil pour l'implémentation des applications réparties.

Il n'existe pas d'implémentation de ce modèle et la notion de temps n'est pas prévue.

II.2 La définition d'un modèle général de communication.

A partir des mécanismes et modèles vus auparavant, on peut constater que pour l'utilisation partagée des réssources et pour le contrôle de la coopération et de la concurrence des activités parallèles d'un système réparti, on a besoin de deux concepts de base :

- L'un pour l'expression des traitement locaux des activités parallèles,

- L'autre pour exprimer les interactions entre les activités.

En prenant en compte ces deux notions, notre modèle a été basé sur deux types d'objets:

- Pour l'expression des activités, nous avons utilisé l'objet processus séquentiel ,

- Pour l'expression des interactions, nous avons utilisé un objet particulier, appelé l'objet de communication ou canal.

L'objet processus séquentiel nous permetra le traitement des informations.

L'objet de communication, représente le seul moyen que les processus auront pour communiquer et se synchroniser entre eux.

L'objet de communication a été défini indépendamment de son implémentation et son comportement est défini par ses contraintes d'accès et par les fonctions qu'il réalise.

Les contraintes d'accès sont des propriétés qui caractérisent l'objet et déterminent son interaction avec les processus (exemples : pour la lecture d'un buffer, on la contrainte d'accès : "buffer non vide", et pour son écriture, on a la contrainte : "buffer non plein" ; pour le protocole de rendez-vous, on a la contrainte "lecture et écriture simultanée", etc).

Les fonctions sur un objet de communication sont les opérations

suivantes telles que : création/destruction, liaison/déliaison et accès pour des consultations ou des modifications.

Les deux types d'objets de ce modèle, processus séquentiels et objets de communication, seront gérés et protégés par ce que nous appellerons le système de communication.

Dans les paragraphes qui suivent, nous allons décrire le comportement général de l'objet de communication à partir de ses propriétés et ses fonctions d'accès.

II.2.1 Type d'interaction directe ou indirecte.

L'interaction entre deux processus peut être caractérise comme directe ou indirecte. Avec le premier type, un processus communique avec un autre, sans nommer l'objet de communication. Par exemple dans CSP (HOAR78), un prcessus communique avec un autre en donnant le nom du processus appelé. Par contre, avec une interaction indirecte, la communication se fait toujours par l'intermédiaire de l'objet de communication. Par exemple, pour la communication par messages d'UNIX, les processus communiquent à travers l'objet de communication PIPE.

Nous pouvons remarquer que dans le type d'interaction indirecte, les notions de communication et de synchronisation sont complètement banalisées (l'objet de communication est transparent au programmeur). Par contre dans l'autre cas, la tâche communication et synchronisation devient présente dans la programmation de l'application.

Il faut remarquer, qu'au niveau de l'implémentation, nous aurons toujours besoin d'un objet pour la communication et la synchronisation des processus.

II.2.2 Création dynamique ou statique des objets de communication.

La création des objets de communication peut être réalisée par demandes explicites des processus (création dynamique) ou être fixée dès la conception du système (création statique).

On trouve un exemple de création statique dans les boîtes à lettres de RSX-11M. Avec la création du processus, le système associe automatiquement une boîte à lettres pour la commmunication par messages. Un processus garde sa boîte à lettres jusqu'à la fin de son exécution et il ne peut pas demander la création d'autres boîtes à lettres. Les messages sont de taille fixe (13 mots).

Cette approche simple a l'avantage de ne pas avoir besoin d'un gestionnaire de boîtes à lettres. Mais évidemment, pour la programmation de structures de synchronisation plus complexes, ces boîtes à lettres présentent des difficultés d'utilisation.

Avec la création dynamique un processus peut, pendant son exécution, créer et detruire dynamiquement les objets de communication qui lui son nécessaires. Pour offrir ce service, le système a besoin d'un gestionnaire des objets de communication. Un exemple de ce type de création se trouve dans les boîtes à lettres d'iRMX.

Il faut remarquer que les fonctions de création et de destruction ne sont pas toujours explicites dans tous les modèles. Par exemple, dans OCCAM, la création est associée à la déclaration de l'objet et la destruction est associée à la fin de l'exécution de la commande parallèle.

Mais au niveau de l'implémentation d'un modèle, les fonctions de création et de destruction sont toujours des primitives élémentaires.

II.2.3 Liaison statique ou liaison dynamique.

Les opérations de liaison et de déliaison des processus aux objets de communication peuvent être caractérisées comme statiques ou dynamiques.

La liaison est statique lorsque les fonctions de liaison et de déliaison sont associées aux fonctions de création et de destruction. On trouve un exemple de ce type de liaison dans les canaux d'OCCAM : un canal reste actif pendant toute la vie de ses processus communicants.

Avec cette approche, à cause de la relation entre les fonctions création-liaison et destruction-déliaison, l'objet de communication ne peut être réutilisé par les processus qu'après la destruction de l'objet.

Avec une liaison dynamique, un processus a la possibilité de gérer ses connexions avec les objets de communication. on trouve un exemplede ce type de gestion dans le système CHORUS : les ports sont créés dynamiquement. Evidemment, pour avoir cette facilité, il faut aussi avoir un sous-système pour la gestion des liaisons.

II.2.4 Comportement fixe ou modifiable.

Le comportement d'un objet de communication dépend de ses contraintes et de ses fonctions d'accès. Ce comportement peut être fixé par le modèle dès le début ou, par contre, on peut donner la possibilité à l'utilisateur de le modifier.

Nous trouvons un exemple de comportement fixe dans les "PIPES" d'UNIX. Un PIPE a le comportement d'un file (FIFO) avec les fonctions d'accès : lecture et écriture, et avec les contraintes d'accès : lecture si PIPE non vide et écriture si PIPE non plein. Ce comportement ne peut pas être changé par l'utilisateur.

Une approche de cette nature est simple au niveau de son implémentation et si l'objet est suffisamment puissant, la programmation de structures plus complexes à partir de cet objet ne doit pas représenter une grande difficulté.

Avec la deuxième approche, le programmeur a la facilité de pouvoir définir plusieurs comportements (modes de communication) en utilisant le même type d'objet.

Le comportement peut être modifié : a) Un changeant les contraintes d'accès, par exemple en iRMX, le programmeur peut choisir entre deux politiques de lecture d'une boîte à lettres : par ordre d'arrivée des demandes ou par ordre de priorité des tâches demandantes ; b) En définisant de nouvelles fonctions, par exemple CSC offre à l'utilisateur la facilité de définir les fonctions d'accès de l'objet port, ainsi que de spécifier les contraintes sur l'ordre d'exécution des fonctions.

Cette approche avec ensemble de comportements possibles, facilite à l'utilisateur la programmation d'une application répartie (il n'aura pas besoin de programmer tous les modes de commnication). Par contre, l'optimisation d'un tel service est un problème qui n'est pas toujours facile à résoudre.

II.2.5 Comportement statique ou dynamique.

L'objet de communication peut répondre seulement aux demandes des processus (objet statique) ; il peut de plus prendre des initiatives sur les processus (objet dynamique).

Les PIPES d'UNIX ou les canaux d'OCCAM sont des objets statiques qui répondent seulement aux demandes de communication (lecture et écriture) des processus.

Les "mediums" du modèle du projet UNIX réparti (WELLI81) sont des objets dynamiques qui peuvent signaler l'arrivée des événements inattendus (mort du "medium", arrivée d'un message de contrôle, etc).

Ce type d'approche est intéressante pour les applications qui ont besoin de réagir "immédiatement" sur des événements externes imprévisibles (exemple : les applications temps réel).

Le premier type d'objet ets plus simple à gérer que le deuxième, mais par contre, le deuxième offre la facilité du traitement asynchrone des événements.

II.2.6. Fonctions d'accès basées sur une référence unique ou sur l'échange de messages.

Jusqu'ici, notre modèle a été défini indépendamment de toute implémentation. Mais, en tenant compte du support physique d'implémentation, nous pouvons faire certaines considérations sur le principe de base utilisé par le mécanisme de synchronisation du modèle : mémoire commune ou échange de messages.

L'utilisation de la mémoire commune comme une répresentation unique de l'état d'un objet de communication, offre les avantages suivants :

- Un ordre élémentaire d'accès est inhérent, on ne peut pas avoir accès à la mémoire que par un seul processus à la fois ;

- L'accès à la mémoire peut être considéré comme "instantané" et les pertes pendant cette opération ne sont pas significatives ;

Par contre, les systèmes qui n'ont pas la facilité de la mémoire commune doivent résoudre des problèmes dûs au manque d'une référence unique :

- Un état d'incohérence ;

- Une mise à jour de l'état de l'objet de communication non "instantanée" (les délais de transmission peuvent alors être critiques pour certains types d'application) ;

- Enfin, une possibilité de pertes d'information non negligable.

Pour l'implémentation de ce type de système, à cause de l'absence de mémoire commune, on doit utiliser pour la synchronisation des fonctions d'accès, des mécanismes basés sur l'échange de messages.

II.2.7 Les fonctions élémentaires d'accès.

Les fonctions d'accès d'un modèle basé sur l'échange de messages, peuvent être définis par les primitives suivantes :

- ENVOYER (idoc,msg); pour l'envoi d'un message : msg
  à l'objet de communication : idoc

- RECEVOIR (idoc,msg); pour la réception d'un message : msg
  de l'objet de communication : idoc

II.2.8 La synchronisation des fonctions d'accès.

II.2.8.1 Communication asynchrone ou synchrone.

Pour l'interaction entre deux processus qui communiquent à travers un objet de communication, nous pouvons observer deux modes de synchronisation :

- Le mode synchrone. Les processus communicants réalisent leur interaction "simultanément". C'est le mode de communication par rendez-vous, utilisé dans les canaux d'OCCAM. Il faut remarquer, que pour ce type d'interaction, l'objet de communication est uniquement un élément de synchronisation et aucune mémorisation du message n'est nécessaire.

- Le mode asynchrone. Dans le cas où un processus demandant une opération sur un de ses objets de communication n'attend pas l'opération correspondante d'un des autres processus communicants (un processus écrivain n'attend pas la lecture de son message par un processus lecteur). Avec cette approche à cause de l'asynchronie dans la communication, l'objet doit être alors capable de mémoriser des messages. Pour ce mode, nous avons deux types de synchronisation pour l'interaction entre le processus et l'objet de communication :

  - Des opérations bloquantes. Le processus demandeur attend jusqu'à ce que la fonction soit réalisée. La fonction ENVOYER reste dans un état d'attente si la mémoire de l'objet est pleine. La fonction RECEVOIR reste dans un état d'attente si la mémoire est vide.

  - Des opérations non bloquantes. Le processus demandeur a toujours une réponse à ses demandes d'interaction. Si l'opération n'est pas effectuée, l'objet rend la valeur de son état courant (mémoire vide, memoire pleine).

  Les fonctions "read" et "write" d'un PIPE d'UNIX sont des exemples de fonctions asynchrones bloquantes.

Le mode de communication asynchrone est encore le plus utilisé dans les systèmes commercialisés. Il faut dire que toute structure de communication et de synchronisation complexe peut être bâtie soit sur un mécanisme asynchrone soit sur un mécanisme synchrone. L'avantage du mode de communication synchrone, la base du modèle CSP, réside dans la définition

d'une sémantique du parallélisme et de la communication.

II.2.8.2 Les modes d'interaction.

Pour la communication entre processus à travers un objet de communication, nous trouvons cinq modes d'interaction d'utilisation courante dans la programmation paralèlle :

- Le point à point. Le schéma le plus élémentaire pour l'interaction entre deux processus.

- Le serveur (un processus serveur et un ensemble de processus clients). Le schéma classique pour la programmation des lecteurs-redacteur.

- Le multiserveur (un processus client et un ensemble de processus serveurs) . Le schéma de base pour le multiplexage sur un ensemble de serveurs.

- Le multiclients-multiserveurs (un ensemble de processus clients et un ensemble de processus serveurs). Le schéma le plus généneral pour la relation client-serveur.

- La diffusion (un émetteur, plusieurs récepteurs). Un schema intéressant pour certaines applications.

Chaque mode d'interaction répond à différents besoins de communication. Les modèles existants offrent soit un seul mode de base soit plusieurs modes. En principe, on n'a pas besoin de tous les modes : on pourrait, à partir du plus élémentaire, définir tous les autres (le langage OCCAM permet la programmation de tous les modes à partir de l'interaction élémentaire point à point synchrone). Mais il faut dire que du point de vue de la programmation d'une application, il est plus agréable d'utiliser des objets de communication prédéfinis.

II.2.9 Les événements et la communication.

L'envoi et la réception ont été définis comme les fonctions élémentaires d'accès des objets de communication. Pourtant, dans beaucoup de mécanismes existants, on trouve un deuxième type de fonction pour la synchronisation des processus : les événements. Par exemple UNIX offre en plus du mécanisme de communication par messages (les PIPES), un deuxième service indépendant pour le traitement des événements (les signaux).

Pourtant, la notion d'événement pourrait apparaître inutile, parce qu'un événement peut être associé à la notion de message et leurs fonctions de traitement peuvent être associées aux fonctions de lecture et écriture. Par exemple, dans CSP, avec les commandes d'entrée et de sortie (où la communication et la synchronisation sont regroupées dans une même notion) et la commande alternative, la notion d'événement peut être exprimée sans mécanisme supplémentaire.

Au niveau de l'implémentation de l'approche, le première, basée sur l'unification des concepts communication et synchronisation, est plus simple : un seul ensemble de fonctions et un seul mécanisme sont nécessaires pour sa programmation. Par contre, pour la deuxième, il faut la gestion des deux modes de synchronisation (par message et par événement).

Au niveau de la programmation d'une application, il faut dire que la notion d'événement comme élément de synchronisation pure, est très importante dans certains types d'application (applications temps réel, RUEB 78, DIX 83).

II.2.10 Les services de communication des modèles de communication analysés.

A partir du service général de communication défini sur la base d'un objet de communication, nous présentons dans la figure III.2.1 un tableau avec les service offerts par les modèles analysés dans II.1.

Table comparative des mécanismes de communications presentés.

Caractéristiques : Modèle	Objet de communication.	Création : dynamique ou statique. (D ou S)	Liaison : dynamique ou statique. (D ou S)	Comportement : fixe ou modifiable. (F ou M)	Comportement : statique ou dynamique. (S ou D)	Communication : asynchrone ou synchrone. (A ou S)	Modes de communication. n lecteurs/ m écrivains	Traitement d'événements integré.	Implementation
1. RSX-11M	Boîtes à lettres.	S	S	S	S	A	1/m	non	oui
2. iRMX	Boîte à lettres.	D	D	M	S	A	1/m	non	oui
3. UNIX	"Pipe".	D	S	F	S	A	n/m	non	oui
4. UNIX-D	Mediums.	D	D	M	D	S	n/m	oui	oui
5. CSP	-	-	-	F	-	S	1/1	oui	-
6. OCCAM	Canal.	D	S	F	S	S	1/1	oui	oui
7. PASCAL-CSP	"Channel".	D	S	F	S	S	diffusion	oui	non
8. PASCAL-M	"Mailbox"	D	D	F	S	S	n/m	oui	oui
9. CHORUS	Ports	D	D	F	S	S	n/m	oui	oui
10. PORTS	Ports	D	D	M	D	S	n/m	oui	non

II.3 Conclusion : Une architecture pour la définition d'un noyau réparti temps réel.

II.3.1 Remarques générales sur l'aspect communication/synchronisation des modèles présentés.

En observant les caractéristiques des différents modèles et de leurs mécanismes de communication décrits dans les paragraphes précédents, nous pouvons faire les remarques suivantes :

- Il existe une grande variété de mécanismes pour l'expression de la communication et la synchronisation des processus. On trouve ces mécanismes à différents niveaux d'utilisation, des primitives spécifiques aux modèles de haut niveau.

- Dans les mécanismes les plus récents, l'activité de communication est séparée de l'activité de traitement.

Nous avons constaté que les primitives de plus bas niveau, spécifiques d'une machine, ne sont pas une bonne solution pour les problèmes de transport et de clarté dans la programmation de grandes applications. Mais pourtant et grâce du fait qu'elles sont proches du matériel, ce type de primitives est très utilisé pour des applications pas trop complexes et qui demandent des optimisations au niveau de l'utilisation de la mémoire et du temps de réponse (PILE 80). Les mécanismes de haut niveau, qu'on trouve dans les langages définis pour la programmation parallèle, font abstraction des matériels utilisés pour leurs implémentations. Ces mécanismes sont des outils plus souples et puissants pour la programmation des applications parallèles. Le transport de ce type de modèles est possible, si l'on réalise des compilateurs dans plusieurs machines cibles. Par contre, au niveau de l'implémentation, l'optimisation n'est pas assurée ; elle dépendra du programmeur du système et des facilités offertes par le matériel utilisé. Un autre facteur qui pourrait être déterminant pour certains types d'applications, est la taille de mémoire disponible. Si une application n'a comme support matériel que des microordinateurs avec de

grandes restrictions en espace mémoire, les modèles généraux et volumineux ne seront pas facilement implémentables.

Nous considérons que chaque modèle de communication répond aux besoins spécifiques d'un type d'application donnée et qu'on ne peut donc pas dire de façon générale qu'un modèle est meilleur qu'un autre, sans préciser son type d'application et son environnement. Par exemple dans un environnement réparti où l'absence de mémoire commune est une contrainte réelle, il est certain que les modèles les plus adaptés sont les modèles basés sur la notion de message comme élément de synchronisation.

A partir des remarques précédentes, nous considérons que pour la définition d'un modèle pour des applications réparties, les tâches de communication et de synchronisation doivent faire partie des services fondamentaux du système d'exploitation.

Un premier problème à résoudre pour l'intégration du service de communication dans le système d'exploitation, est le niveau de l'architecture où l'on doit placer ce service. On peut envisager trois types de solutions : a) définir le service au-dessus d'un système d'exploitation existant ; b) introduire le service dans le noyau d'un système et c) redéfinir un noyau basé sur un mécanisme de communication.

II.3.2 Exemple de l'évolution des services de communication dans les systèmes répartis.

Dans l'évolution des UNIX répartis, on peut trouver les différents niveaux d'integration du service de communication (RASH80, JOY81, BLAI83, LUDE81, etc). Dans les paragraphes suivants, nous présentons trois de ces approches.

II.3.2.1 Un exemple de service de communication au-dessus d'un système existant : The Newcastle Connection (BROW 82).

La "Newcastle connection" est un logiciel qui a été développé à l'université de Newcastle en Angleterre. Le support de communication utilisé est un réseau local de type anneau, le "Cambridge Ring". Les sites reliés au réseau sont des PDPs avec UNIX comme système d'exploitation.

L'objectif de cette approche a été d'offrir aux utilisateurs un service d'accès des objets distants sans changer ni les primitives ni le "kernel" UNIX.

Dans cette approche, un sevice de communication du type "appel à distance" a été intégré sans aucune modification du noyau UNIX ("kernel"). Le "newcastle connection" est un logiciel qui s'exécute comme un processus système. Pour tout appel d'un objet (fichier) non local, un protocole de communication est établi entre le site qui fait l'appel et le site qui possède l'objet (pour l'accès aux fichier distants, un identificateur global détermine la localité du fichier). Pour l'accès à un objet local, l'appel est passé au "kernel" local. Dans la figure II.3.2.1 on montre schématiquement la communication entre deux sites à travers le "Newcastle connection".

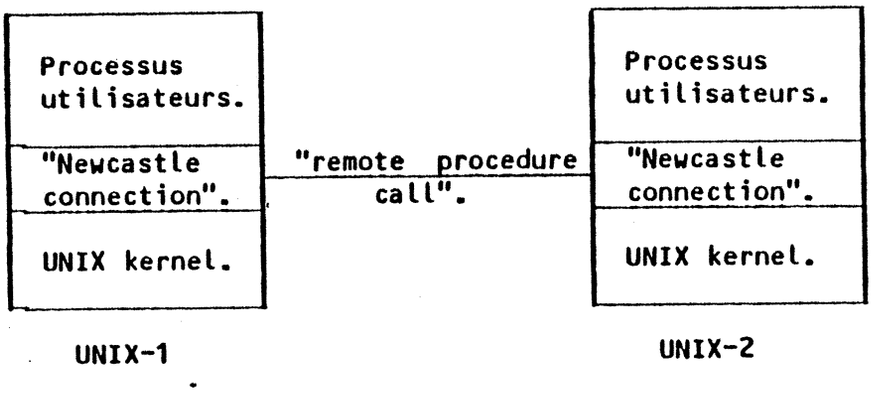


fig. II.3.2.1 "Newcastle connection".

Cette approche a les caractéristiques suivantes :

- Un logiciel simple grâce au réseau homogène de systèmes UNIX.
- Une implémentation qui ne demande pas de modification du noyau.
- Un service de communication bien adapté pour l'implémentation des serveurs à distance.

II.3.2.2 Un exemple d'introduction d'un service de communication dans le noyau : (ROWE 82).

Ce logiciel a été développé à l'université de Berkeley. Le support de communication utilisé a été un réseau local de type anneau : le réseau

COCANET (développé aussi dans le même laboratoire). Les sites du réseau sont des PDPs avec UNIX comme système d'exploitation.

Un des objectifs recherchés dès la définition de ce projet a été d'offrir un service de communication efficace pour le partage des ressources du réseau et un service de support pour les recherches en base de données de l'université.

Pour assurer un service efficace, le noyau d'UNIX a été modifié. Pour offrir un service de communication plus adapté à ses besoins, trois modes de communication on été définis : un service de datagramme ; un service de circuit virtuel ; et un service à diffusion "multicast".

Les caractéristiques de cette approche sont :

- Un réseau homogène de systèmes UNIX qui facilite l'implémentation du modèle.

- L'introduction de plusieurs services de communication dans le système UNIX adaptés à un type d'application (base de données).

II.3.2.3 Un exemple de système basé sur un mécanisme de communication : MERT (LICK 78).

MERT est un système d'exploitation temps réel qui a été développé aux laboratoires Bell. MERT est un système monoprocesseur qui a été developpé pour des machines PDPs.

Pour offrir un système adapté aux applications temps réel, les laboratoires Bell ont rédéfini un nouveau système qui a les caractéristiques suivantes :

Il s'agit d'un système hiérarchisé en quatre niveaux :
-Au niveau le plus bas, on trouve le noyau (le "kernel") qui a comme fonctions la communication et la synchronisation des processus (le traitement des interruptions et la communication par messages).
-Au deuxième niveau (le "mode kernel"), on trouve des processus pour la gestion des "drivers", des fichiers, et de la mémoire.

-Au troisième (le "mode superviseur"), on trouve les superviseurs du système (l'administrateur des processus utilisateurs, le superviseur du temps partagé, etc).
-Et au dernier niveau (le "mode user"), on trouve les processus de l'utilisateur.

Dans cette approche, à cause des contraintes de l'application visée, le noyau a été rédefini. Dans ce système, UNIX est exécuté comme un processus du système au niveau superviseur. Il existe une extension proposée pour un biprocesseur, D-MERT.

II.3.3 Architecture proposée.

En regardant ces différentes approches, nous constatons que la définition d'une architecture n'est pas indépendante du type d'applications envisagées.

En considérant d'une part la nature des applications visées par notre étude (haut degré de parallélisme et de communication et systèmes hétérogènes avec des restrictions en espace mémoire), et d'autre part toutes les remarques précédentes, nous pouvons déjà préciser les bases de l'architecture de notre modèle :

- Le haut degré de parallélisme et de communication avec des temps de réponse courts, nous a ammené à définir un noyau basé sur un mécanisme de communication.

- A cause des restrictions en espace de mémoire, notre modèle sera basé sur un mécanisme unique. Les processus pourront communiquer directement, en utilisant le mécanisme élémentaire, ou indirectement, en programmant des mécanismes plus complexes à partir du mécanisme de base.

Fig. II.3 L'architecture de communication de notre modèle.

CHAPITRE III

LES CANAUX : UN MODELE DE COMMUNICATION POUR UN NOYAU REPARTI TEMPS REEL

INTRODUCTION.

A partir de la notion canal, objet pour l'expression de la communication et de la synchronisation des processus parallèles, nous avons défini un modèle de communication pour un noyau réparti. Pour sa conception, nous avons pris en compte, la caractérisation de l'application type présentée dans le premier chapitre et les services de communication analysés dans le deuxième.

Nous avons considéré comme un des points importants dans la définition de notre modèle, l'établisement d'un compromis entre les besoins spécifiques d'un type d'application et les besoins généraux au niveau de l'expression de la communication et la synchronisation.

Nous estimons que pour matérialiser ce compromis, le modèle doit être :

- -A travers de l'utilisation de concepts de haut niveau, simple et clair dans l'expression de la communication et la synchronisation.
- -Conçu pour un type d'application.
- -Proche des matériaux utilisés.
- -Transportable.

Nous nous sommes fixé comme objectif l'utilisation de concepts de haut niveau pour la définition de notre noyau afin de rendre la programmation claire même au niveau le plus bas. Et en même temps, nous avons défini un modèle le plus proche possible du matériel pour une réponse aux exigences des applications temps réel : temps de réponse courts et facilités pour la programmation de "handlers" non standard.

La transportabilité du modèle a été aussi retenue comme un des objectifs principaux, à cause du type de réseau auquel nous nous intéressons : les réseaux hétérogènes.

Dans ce chapitre, nous présentons d'abord l'architecture que nous avons utilisée pour l'implémentation de notre modèle et ensuite la spécification de chacun des objets qui composent notre modèle.

III.1 L'architecture du modèle.

Les concepts message et canal ont été à la base de la définition d'un modèle de communication pour un noyau réparti.

Les caractéristiques de l'application type.

Dans le premier chapitre, nous avons fait une caractérisation générale du type d'application visé. Pour sa prise en considération dans la définition de notre architecture, nous la rappelons ici :

-C0. Répartition géographique des processus de l'application.
-C1. Haut degré du parallélisme dans l'exécution des activités.
-C2. Nombreux échanges de messages de petite et moyenne taille.
-C3. Réponse "immédiate" pour certains types de messages critiques
-C4. Manipulation du temps.
-C5. Manipulation d'interfaces non standard.
-C6. Comportement dynamique du système (création et destruction dynamique des tâches).
-C7. Sûreté dans le fonctionnement .
-C8. Réseau hétérogène avec des sites à capacité faible en mémoire.

Les principes de base de notre modèle : le message et le canal.

Pour la communication des processus parallèles, nous avons basé notre modèle sur les deux notions suivantes :

-Le message comme l'élément permettant l'expression des interactions (communications et synchronisations) des processus parallèles.

-Le canal comme l'objet de synchronisation des interactions des processus parallèles.

Nous estimons que ces deux notions s'adaptent bien à notre type d'application.

Un mécanisme basé sur le message est le plus approprié aux applications où l'absence de mémoire globale est une contrainte réelle (C0. Ré-

partition géographique des processus.).

Un objet orienté vers la communication et la synchronisation, nous facilitera la programmation des applications où l'activite communication est importante et complexe (C1. Haut degré du parallélisme.., C2. Nombreux échanges de messages..)

L'architecture du modèle.

Il s'agit de proposer un modèle qui soit à la fois proche des concepts de haut niveau (pour faciliter la programmation générale des structures de communication et synchronisation complexes) et proche de la machine (pour faciliter l'optimisation du type d'application visé (C3. Réponses "immediates" pour certains événements..)).

Nous proposons donc une architecture à trois niveaux :

- Au niveau le plus élevé, le niveau application, on a des processus séquentiels (P) qui communiquent à travers des processus canaux (C).

- Au niveau intermédaire, le niveau système, on a le processus qui se charge de la gestion des processus P et C, le processus noyau (NC). Nous trouvons aussi des processus P et C propres au noyau (processus système).

- Au niveau le plus bas, le niveau interface d'implémentation, on a un mécanisme de communication élémentaire qui nous permet d'une part, de programmer les interactions entre le trois différents processus du modèle (P, C et NC) et, d'autre part, d'implémenter le modèle sur une machine cible.

III.1.1 Le niveau application : les processu P et C.

Au niveau de l'application, notre objectif a été la définition d'un modèle : a) simple et clair pour l'expression de la communication et de la synchronisation des processus parallèles et b) adaptable aux besoins de communication des applications réparties temps réel.

La définition d'outils simples et puissants pour la réalisation, l'analyse et les preuves de programmes parallèles, est un des principaux axes de recherche dans le domaine de la programmation répartie. Au fur et à mesure de ces recherches, on peut constater les points suivants :

- Les communications et les synchronisations élémentaires au niveau de l'implémentation sont banalisées au niveau de l'application. Exemple : le rendez-vous CSP est indépendant du protocole utilisé pour son implémentation.

- Les mécanismes de communication et de synchronisation sont de plus en plus proches des processus communicants. Exemples : le moniteur est un modèle adapté pour exprimer le mode de communication clients-serveur. Le rendez-vous de CSP est orienté vers la communication par messages.

- Les activités correspondantes à la communication et à la synchronisation des processus sont séparées du traitement. Exemple : Le moniteur permet la programmation d'une communication du type serveur indépendamment des processus clients.

Un application temps réel répartie est caractérisée par la varieté et la complexité des structures de communication et de synchronisation utilisées pour l'expression des interactions de ses processus parallèles.

Communication synchrone, communication asynchrone, communication point à point, relation clients-serveur, sont des exemples courants de modes de communication qu'on trouve dans les applications répartis.

Pour l'implémentation de différents modes de communication, on peut suivre trois approches différentes à partir de :

-La définition d'un seul mécanisme de base,
-La définition de plusieurs mécanismes indépendants,
-D'un objet de base programmable.

Nous trouvons un très bon exemple de la première approche dans CSP : à partir du mode de communication par rendez-vous et d'un petit ensemble de commandes (les commandes d'entrée et de sortie, l'alternative, etc), on peut programmer des structures plus complexes. Cette approche est très intéressante par le nombre réduit de commandes. Mais il faut remarquer que si les commandes sont simples et puissantes, elles sont cependant complexes au niveau de leur implémentation. La commande alternative, par exemple, utilise implicitement des événements et un protocole qui demande l'utilisation de primitives plus élementaires. On peut faire une deuxième remarque : les structures plus complexes et courantes de la programmation répartie, comme une file, doivent être programmées et gérées par l'utilisateur.

La deuxième approche peut être intéressante par rapport à l'optimisation qu'on pourrait dans chaque mode de communication en ce qui concerne l'implémentation. Mais ella a l'inconvénient d'utiliser un nombre considérable de commandes.

Pour notre modèle, nous avons choisi la troisième approche. A partir d'une structure de base et d'un ensemble relativement limité de commandes, nous avons défini l'objet canal comme un processus.

Le modèle de communication par canaux.

Une application répartie sera exprimé par deux types de processus : les processus séquentiels appelés processus P et les processus canaux appelés processus C.

Un processus C est un objet qui a les caractéristiques suivantes :

- Objet unique pour l'expression de la communication/synchronisation des processus séquentiels P.

- Objet paramétrisable pour la définition de différents modes de communication de base (rendez-vous, FIFO, etc.).

- Objet programmable pour l'adaptation de l'objet à différents environnements.

A partir de ce modèle, les processus d'une application répartie permettront les fonctions suivantes :

-les processus P (programmés par l'utilisateur), les traitements séquentiels, et
-les processus C (disponibles à l'utilisateur) : a) à travers des paramétres, la définition de modes de comunication courantes et b) à travers leur programmation, la rédéfinition de modes de communication adaptés aux besoins spécifiques d'une application.

Le comportement d'un canal.

Le comportement d'un canal, est défini par :

- Le mode de communication (point à point, client-serveur, etc).

- Les contraintes d'accès (la longueur de la queue, le durée de vie des messages, etc.).

- Les fonctions d'accès au canal (lecture, écriture, traitement d'événements, etc).

Le choix du type de canal (son mode de comunication et ses contraintes d'accès) dépend des besoins de l'application en communication et synchronisation. Par contre, ses fonctionts d'accès seront liées à l'implémentation des protocoles internes et au système d'exploitation utilisé pour leur implémentation (enregistrement, transfert de messages ..). Le changement de système, l'optimisation d'un protocole ou la définition de nouveaux modes de communication, entraîneront seulement des modifications minimales dans les processus C, tandis que les processus P resteront inchangés.

Les service de communication.

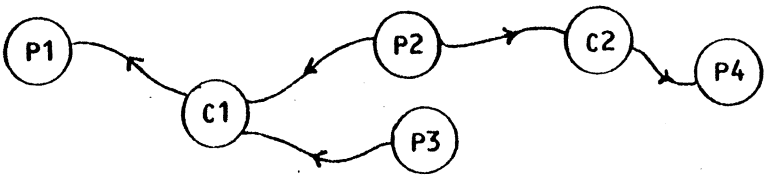
Les canaux seront accessibles aux processu P, à travers un ensemble de primitives offertes par le noyau du système. Cet ensemble de primitives permettra la gestion des canaux (création/destruction dynamiqe et liaison/déliaison dynamique), les accès aux canaux (lecture/écriture) et le traitement des événements.

Le comportement du processus P.

Un processus P représente l'unité d'exécution séquentielle d'un programme parallèle. Les états d'exécution interne d'un processus n'ont pas d'influence sur le reste des processus (ils ne partagent pas leur mémoires) et le seul moyen d'interaction entre eux sont les canaux.

Pour la modelisation du processus P, nous avons utilisé la notion d'acteur (HEWI 77). Un processus sera composé d'un ensemble d'activités qui sont déclenchés par l'émission des messages. La justification de notre choix est présenté dans le paragraphe correspondant à la spécification de l'interface d'implémentation (III.2.1).

Dans la figure III.1, on montre le niveau application. A ce niveau nous n'avons que deux types de processus, les processus P et C.



P1,..,P4 processus séquentiels.
C1, C2 processus canaux.

fig.III.1 Le niveau application.

Remarques.

Nous considérons que l'objet canal de notre modèle se place dans le cadre des recherches actuelles, par les aspects suivants :

- Les communications internes des canaux sont banalisées au niveau de l'application.

- Les canaux sont orientés vers l'échange de messages, donc vers la communication entre processus.

- Avec les canaux, on sépare le traitement de l'information de celui de la communication/synchronisation.

Avec la notion de canal, nous donnons aussi une réponse aux besoins des applications temps réel : proximité de la machine (conséquences des points : C3. Réponses "immédiates" pour certains types d'événements, C5. Manipu lation d'interfaces non standard). Le canal est un objet du type processus avec un comportement qui peut être reprogrammé pour s'adapter à un nouvel environnement.

Le canal est un processus général pour les communications. Son utilisation pour certains types de protocoles de communication simples et très courants, peut être onéreuse en espace mémoire et en temps d'exécution. Dans ce cas, l'utilisateur a la possibilité de programmer des canaux moins généraux mais plus optimaux : par exemple, des canaux accessibles à tout le monde sans qu'il soit besoin besoin de demander leur liaison, des canaux qui sont exécutés comme des procédures dans le noyau, des canaux spéciaux pour des "handlers" non standard, etc. La seule restriction à respecter dans la programmation de ces canaux est de maintenir le message comme unité de communication et de synchronisation entre les processus.

III.1.2 Le niveau système : le processus noyau.

Au niveau système, nous avons cherché la définition d'un noyau minimum, à cause des contraintes de l'espace de mémoire de notre configuration réseau visée par notre étude (C8. Réseau hétérogène de sites avec capacite faible en mémoire).

Le processus noyau (NC) a comme tâche principale la gestion et le contrôle des deux autres processus du modèle : les processus P et C.

Les tâches du noyau

Un des premiers problèmes qui se posent lors de la définition du noyau d'un système d'exploitation se résume à la question : quelles sont les tâches système qui doivent être intégrées dans le noyau?. Dans la définition du noyau CHORUS (GUIL 82), on trouve, par exemple, une analyse des différentes tâches système qui y peuvent être intégrées ainsi qu'une étude comparative des choix faits dans le cas de différents noyaux existants.

-Création et destruction des processus.
-Cadencement.
-Création et destruction des objets de communication.
-Communications locales.
-Routage.
-Réalisation d'entrées et de sortiés.
-Gestion des fichiers.
-Gestion de la mémoire.
-Traitement des interruptions.
-Gestion du temps.
-Traitement des erreurs.
-..

Pour la définition des tâches à intégrer à notre noyau, nous avons pros en compte différentes caractéristiques de notre type d'application.

C8. Réseau hétérogène et capacité faible en mémoire,
C3. Réponses "immédiates" pour certains événements critiques,
C5. Manipulation d'interfaces non standard..

A cause des limitations de l'espace mémoire, notre premier objectif a été de définir un noyau minimun. Pour le choix des tâches à y intégrer, nous avons faissons les obsevations suivantes :

-Tout sys-tème d'exploitation qui gère un ensemble de processus doit tout d'abord assurer le service de cadencement de ses processus et

-Tout sytème réparti a pour tâche fondamentale la communication.

Pour notre noyau, nous n'avons retenu que ces deux fonctions, et de plus, nous les avons liées dans une même opération : toute communication implique un changement du contrôle de l'exécution (voir mécanisme d'appel III.1.3). Toutes les autres tâches système (création/destruction de processus, traitement des entrées/sorties, etc) seront programmées à l'aide de processus P et C spécifiques que l'on appellera processus système.

Le comportement du processus noyau.

Toutes les demandes d'interaction des processus P ou C seront adressées au processus noyau qui pour le contrôle des interactiones, réalisera les fonctions suivantes :

-Validation de la demande'interaction.
-Mise à jour des états des processus impliqués dans la demande.
-Selection du prochain processus à exécuter.
-Exécution du processus selectioné.

Politique de selection ("scheduling").

Dans l'objectif de donner une priorité aux événements externes et aux opérations de communication, les processus noyau activera les processus qui attendent leur exécution, selon la politique de service suivante :

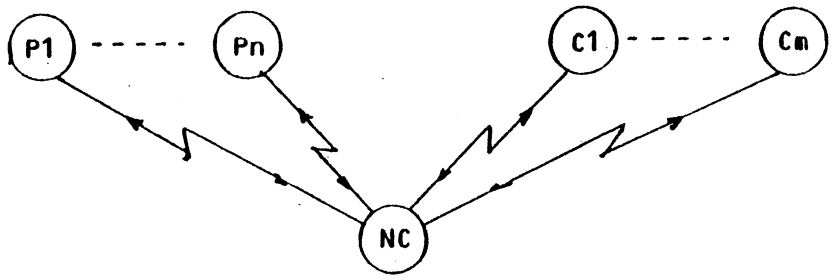
-Pour donner la priorité aux opérations de communication sur les opérations séquentiels, les processus C seront donc prioritaires sur les processus P.

-Pour la prise en compte des événements externes critiques (les "handlers" sont les processus système qui se chargent de la capta-

tion des événements externes et critiques) les processus système seront prioritaires sur les processus utilisateurs.

-Pour donner aux programmeurs un contrôle pour l'exécution de ses objets, les processus P et C, lors de leur création auront associé une priorité.

La figure III.1.2 présente le niveau système. A ce niveau toutes les interactions entre les processus P et C sont contrôlées par le processus noyau.



P1,....Pn processus séquentiels.
C1,....Cm processus canaux.
NC processus noyau.

fig. III.1.2 Niveau système.

Remarques.

La définition d'un noyau minimum dans l'environnement d'un réseau hétérogène, nous permettra, en fonction de la mémoire disponible, de programmer des systèmes sur mesure et de répartir les tâches système. Chaque site aura au moins son noyau, et le reste des tâches pourront résider dans le site local ou dans des sites distants.

Pour le temps de réponse (élément critique pour notre type d'application) avec la définition d'un noyau minimun et le traitement prioritaire sur les "handlers", les processus "critiques" seront commutés dans des délais courts. Il est certain que la rapidité de la commutation des processus n'est pas suffisant pour les applications temps réel. Pour

cette raison nous avons aussis défini un mécanisme de priorités disponible pour l'utilisateur.

Il faut remarquer que la définition de la politique de sélection n'est pas fixe, elle peut être rédefini en reprogrammant l'algorithme de "scheduling" du noyau.

La définition du noyau est indépendante d'une architecture particulière (monoprocesseur ou multiprocesseur). Le passage du contrôle de l'exécution ("scheduler") à un plusieurs processus, dépendra de l'architecture de la machine utilisée pour l'implémentation. Le noyau est aussi independante d'une implémentation en particulière.

III.1.3 Le niveau interface d'implémentation.

Au niveau de l'interface d'implémentation, notre premier objectif a été de définir des outils de base simple pour une programmation simple de notre modèle et et des outils simples pour une adaption facile à différentes machines (C8. Réseau hétérogène de sites ..).

Dans la figure III.1.3, nous montrons les deux sous-niveaux d'interface : l'interface du modèle et l'interface machine.

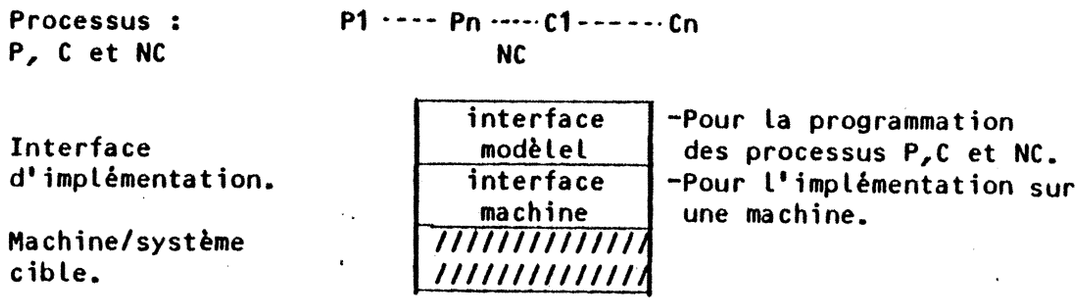


fig. III.1.3 L'interface d'implémentation.

Au niveau de l'interface avec le modèle, la programmation des processus P, C et NC est indépendante de la machine cible, on utilise une strucutre type de processus et un mécanisme de contrôle élémentaire pour la programmation des processus.

Au niveau de l'interface, par contre, la programmation de la structure type et du mécanisme de contrôle, dépend de la machine utilisée pour l'implémentation.

Le mécanisme d'appel.

Notre interface d'implémentation appelée le mécanisme d'appel, est composé de deux entités : un modèle de processus, pour la programmation des trois types de processus du système et un mécanisme de contrôle, pour

la programmation des différentes interactions entre les processus P, C et NC.

Dans l'objectif de définir une interface simple pour la définition de notre mécanisme d'appel, nous avons associé au mécanisme de contrôle plus élémentaire, le contrôle de l'exécution, le mécanisme de communication. Le contrôle d'exécution et la communication sont associés dans une même opération.

Pour la définiton du mécanisme d'appel, nous avons examiné les structures des mécanismes de contrôle suivants :

- Les interruptions, le mécanisme de niveau le plus bas et le plus proche de la machine.

- Le branchement inconditionnel : le JUMP ou GOTO, mécanismes de niveau plus élevé qui existent dans des langages du type assembleur, FORTRAN, PASCAL, etc.

- L'appel de procédure, mécanisme plus élaboré qu'on peut trouver dans les langages décrits ci-dessus.

- La coroutine (CONW 63a), mécanisme moins restrictif que l'appel de procédure et qui permet d'exprimer la multiprogrammation dans un environnement monoprocesseur.

- L'exécution asynchrone (ex. FORK UNIX(RITC 74)), primitive qui permet l'expression des exécution parallèles.

- L'exécution synchrone (ex. commande parallèle de CSP (HOAR 78), l'INITIATE d'ADA), primitive de haut niveau pour l'exécution et le contrôle du parallèlisme.

- Un mécanisme d'utilisation simple pour une programmation claire au niveau de l'application. A ce niveau, des mécanismes du type commande parallèle ou primitive fork sont plus clairs pour l'expression du parallélisme que les primitives élémentaires du type interruption ou GOTO.

- Un mécanisme souple et puissant pour la programmation de structures plus complexes. Comme outils puissants pour la programmation, les structures les mieux placées sont les primitives de haut niveau ; par contre, au niveau de l'optimisation de l'implémentation, les mécanismes les mieux placés sont les plus élémentaires.

- Un mécanisme optimum pour une programmation en "temps réel" (réponses "immédiates" pour certain type d'événements). Les mécanismes les plus optimus pour des réponses en "temps réel" sont les mécanismes élémentaires existants dans la machine utilisée pour l'implémentation. Par contre au niveau de la clarté d'expression et du transport ces mécanismes sont les moins adaptés.

- Un mécanisme souple au niveau de son implémentation, pour une adaptation simple à différents environnements. Pour avoir une implémentation optimale, surtout en terme de temps de réponse, le mécanisme sera le plus proche possible de la machine cible.

Modèle de processus.

La notion de processus que nous avons utilisé pour la définition de notre modèle est proche de la notion d'acteur (HEWI 77).

Un processus est défini comme un ensemble d'activités asociées à des points d'entrée. Une activité est exécutée par l'envoi d'un message sur son point d'entrée correspondant.

Mécanisme de contrôle élémentaire.

Comme le message a été choisi comme l'élément de base de notre modèle,nous l'avons associé au mécanisme de contrôle plus élémentaire, le contrôle de l'exécution. Le contrôle de l'exécution et de la communication sont associés dans une même opération.

A partir de ce principe, un processus lors de l'emission d'un message passe le contrôle d'exécution et lors de la réception prendre le contrôle d'exécution sur un de ses points d'entrée avec la réception d'un message.

Remarques (La transportabilité du modèle).

La définition de notre mécanisme d'appel basé sur une notion de processus général, facilitera l'implémentation de structures de contrôle plus complexes. Et en même temps, grâce à la simplicité de son fonctionnement et au fait d'être le seul mécanisme défini pour la programmation de notre modèle, le mécanisme d'appel sera facilement implémentable à partir des structures de contrôle disponibles dans la machine et le système utilisés pour une implémentation donnée.

Le mécanisme d'appel est la partie la plus proche du système et de la machine utilisée. Son implémentation dépend du niveau du langage utilisé (langage machine, langage de haut niveau) et des primitives élémentaires de communication et synchronisation offertes à ce niveau (événements, sémaphores, boîtes à lettres, pipes,..).

Il est évident que si on désire une implémentation optimale pour les applications temps réel, la programmation de l'interface doit être faite en utilisant une machine nue avec ses primitives élémentaires de contrôle (interruptions, masques, événements, etc.). Cependant, comme le mécanisme d'appel a été défini indépendamment d'une implémentation particulière, il est possible de programmer l'interface sur un système d'exploitation existant en utilisant ainsi des primitives plus élaborées que celles d'une machine nue.

Dans le dernier chapitre, nous présentons un exemple d'implémentation sur le système d'exploitation UNIX.

III.2 La spécification du modèle.

Après la définition de l'architecture, nous allons définir chacun des objets du modèle. Nous définirons, d'abord, la structure générale d'un processus et son mécanisme de contrôle élémentaire. A partir de ces outils, nous décrirons, ensuite, la structure et le comportement du processus noyau. Nous terminerons enfin par les structures et les comportements des processus P et C.

III.2.1 Le mécanisme d'appel.

Le mécanisme d'appel, comme il a été défini dans III.1.3, est composé d'un modèle de processus et d'un mécanisme élémentaire de contrôle. Le mécanisme d'appel nous permettra de définir, d'une part, le contrôle de l'exécution/communication des processus du système et, d'autre part, l'implémentation sur la machine cible.

III.2.1.1 La notion de processus.

Dans notre modèle, un processus est considéré comme un ensemble d'activités séquentielles qui sont déclenchées par l'arrivée de messages. A chaque instant il n'y a plus qu'une activité en exécution (notion d'acteur).

La structure d'un processus est composée d'un ensemble d'activités et d'une zone de mémoire commune et locale (non accessible aux autres acteurs). Chaque activité est associée à un point d'entrée qui permet son accès. La figure III.2.1.1 indique la structure d'un processus.

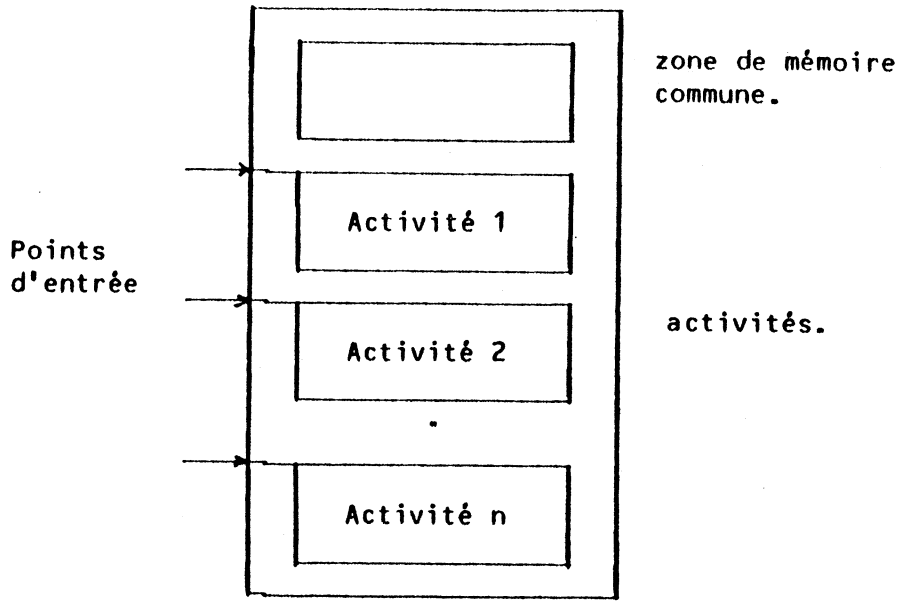


fig. III.2.1.1 Structure d'un processus.

Le comportement d'un processus.

- Un processus est composé d'une zone de mémoire commune et d'un ensemble d'activités. Les activités d'un processus sont programmées à l'aide d'un langage séquentiel (ex.: pascal, C, etc.).

- Un processus actif peut se trouver dans un de ces deux états :
  -Dans l'état EXEC, si le processus est en train de traiter ses variables internes ;
  -Dans l'état COMM, si le processus est en train d'effectuer une communication.
  Le traitement des variables internes du processus n'ont pas d'influence sur l'état de la communication globale du système. Par contre, quand un processus réalise une opération de communication, on a un changement de l'état de communication globale du système.

- Un processus communique avec un deuxième, seulement s'il envoie un message sur un des points d'entrée des activités du deuxième. Chaque processus doit contrôler le passage d'un de ses états à un autre (EXEC-COMM).

L'identification d'un processus.

Tout processus a un nom unique. Chacune de ses activités a un nom différent. Un processus ne communique avec un autre que s'il connait le nom du processus destinataire et le nom de l'activité à exécuter.

Par rapport à ce schéma de désignation, on peut dire qu'il s'agit d'une désignation du type global (les identificateurs des activités des processus doivent être connus par les processus qui peuvent éventuellement y accéder). On peut dire aussi que le type d'adressage est direct (si un processus veut communiquer avec un autre, il doit en connaître le nom).

Deux types d'erreur peuvent survenir au moment du passage du contrôle : la demande d'exécution d'une activité inexistante et la demande d'exécution d'un processus inexistant.

Le premier type d'erreur est contrôlé par le processus appelé, qui a la possibilité de le signaler au processus appelant ou d'exécuter une activité prévue par ce type de situation.

Le deuxième type d'erreur sort du contrôle au niveau du processus et une telle éventualité aura des conséquences graves pour le comportement normal du système.

Remarques :

Notre premier objectif, à ce niveau à, été la défintion d'un mécanisme élémentaire de synchronisation. Il faut noter que son mécanisme d'adressage et protection est très élémentaire et qu'il exige certaines précautions d'emploi de la part du programmeur. Cependant, cet aspect de protection et sécurité est pris en compte à un stade plus élevé par les processus noyau et canaux.

III.2.1.2 Le mécanisme de contrôle : le mécanisme d'appel.

Pour la définition de la synchronisation du passage de l'état de communication à l'état d'exécution, nous avions deux possibilités :

1) Le processus appelant après l'émission du message, revient immédiatement à son état EXEC (passage asynchrone).

2) Le processus appelant attend l'arrivée d'un signal (message) explicite pour retourner à l'état EXEC (passage synchrone).

La première approche est en principe plus simple à implémenter. Pour l'implémentation d'une communication synchrone (rendez-vous), le processus appelant doit se mettre en attente d'une réponse de la part du processus appelé.

La deuxième approche a l'avantage de faciliter l'implémentation de protocoles synchrones plus complexes (exemple CSP, ADA, etc.), et dans le cas où l'on a besoin d'une communication asynchrone, un processus devra débloquer tout de suite le processus appelant.

Pour la définition de notre mécanisme de contrôle, nous avons choisi la deuxième approche, parce qu'il existe un modèle formel qui est basé sur la communication synchrone (CSP). Avec l'objectif de le rendre le plus optimal possible, nous l'avons integré au niveau plus bas de notre architecture.

Le principe de fonctionnement.

Le principe de fonctionnement du mécanisme d'appel est le suivant :

- Un processus qui émette un message abandone en même temps le contrôle de sa propre exécution,
- il reprend le contrôle de son exécution à l'arrivée d'un message.

La figure III.2.1.2.c montre la structure du mécanisme d'appel, avec ses mécanismes élémentaires : MULTIPLEX pour la réception de messages et la reprise du contrôle d'exécution et SWUTCHP pour l'émission des messages et le passage du contrôle d'exécution à un autre processus.

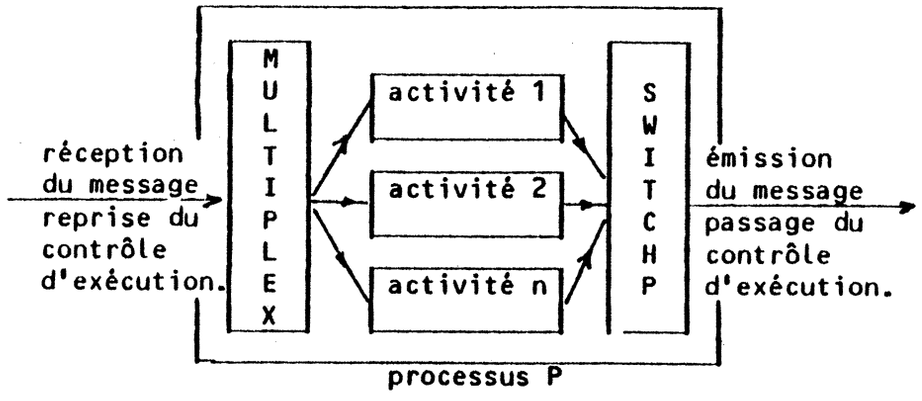


fig. III.2.1.2.c Le mécanisme de contrôle.

Le contrôle de l'exécution est réalisé à l'aide de deux primitives définies ci-dessous :

- MULTIPLEX permet à un processus de prendre le contrôle de l'exécution par la réception d'un message adressé à l'un quelconque de ses points d'entrée. Cette primitive est toujours associée au processus et elle est tranparente au niveau de la programmation.

- SWITCHP permet de passer le contrôle de l'exécution à un processus en lui envoyant un message. Cette primitive nous permet la communication entre les processus et elle sera la seule accessible au niveau de la programmation (toute demande de communciation au niveau application sera traduite au niveau de l'interface d'implémentation à une primitive SWITCHP).

III.2.1.3 La transportabilité du mécânisme d'appel.

Pour la programmation de notre modèle, nous avons défini un seul mécanisme de contrôle, qui est basé sur l'échange de messages.

La définition de ce mécanisme rendra plus simple le problemème du transport de notre système. Avec un seul type de mécanisme, le transport du système se réduit à la reprogrammation des primitives de contrôle (MULTIPLEX et SWITCHP). De plus, en ayant pris le message comme unité de synchronisation, le transport du système sur de systèmes répartis devient plus direct.

En résumé, notre modèle a été basé sur deux concepts de haut niveau, le message et le rendez-vous. Nous considérons que cette approche est bien adaptée, d'une part, aux applications réparties, où le message est l'élément de synchronisation et, d'autre part, aux langages qui utilisent des communications du type synchrone.

III.2.2 Le processus noyau.

Le noyau est le processus qui a pour tâche la gestion des deux autres types de processus du système (les processus P et C). Il faut signaler qu'en dehors de sa différence fonctionnelle, le processus noyau a la structure et utilise le mécanisme de contrôle qui ont été définis dans le paragraphe II.1.

La tâche fondamentale du processus noyau est : la gestion du cadencement et de la communication des processus P et C.

Dans ce paragraphe, nous présentons la spécification générale du processus noyau. L'annexe 3 : "Description du processus noyau" en donne une description plus détaillée.

III.2.2.1 La structure du noyau.

Le processus noyau est un exemple classique de processus serveur, où les clients sont les deux types de processus P et C et où le service à offrir est la communication/exécution des processus P et C selon la politique de service prédéfinie dans sa procédure de "scheduling" (SCHD).

De par ces fonctions et la structure définie dans II.1, le processus noyau inclue la structure suivante : dans sa zone de mémoire on trouve l'état de la communication du système (le table états des processus P et C), et dans les activités on trouve les services de communication offerts par le noyau.

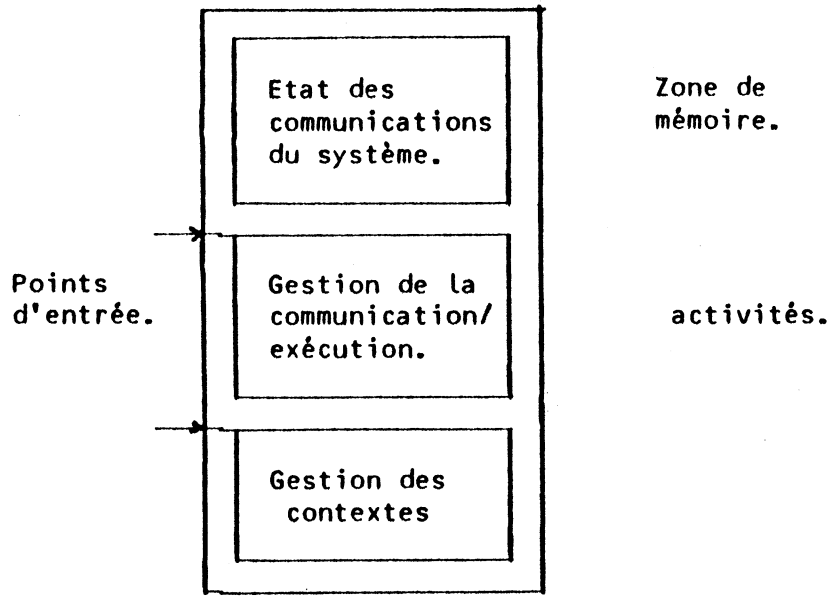


fig. III.2.2.1 La structure du noyau.

L'accès aux services de communication du noyau est associée implicitement à toute demande d'interaction des processus P et C. Toute primitive de communication au niveau de l'application (création des canaux, lecture de canaux, etc.) est traduite à une instruction SWITCHP adressée au point d'entrée correspondant à l'activité communication/ exécution du noyau. Il faut remarquer qu'au niveau de l'application, l'accès au noyau est complètement transparent à l'utilisateur (dans une application, les processus du type P communiquent seulement à travers des processus du type C).

Pour mantenir une cohérence dans le système, le noyau, en plus de la gestion de la communication/exécution, doit assurer un service de gestion des processus (création, destruction et modification des contextes des processus P et C). Les services du noyau sont donc regroupés en deux classes traitant chacun deux types de de demande :

- Pour la gestion de la communication/exécution (pour la communication entre les processus P et C) :

  P --> C pour une demande de communication de P à C.
  C --> P pour la terminaison de l'opération de C à P.

- Pour la gestion des contextes des processus (pour la communication avec le noyau) :

  P --> NC pour la mise à jour des contextes.
  C --> NC pour la mise à jour des contextes.

III.2.2.2 La gestion de la communication/exécution.

Le comportement du noyau est défini à partir du principe de fonctionnement de notre mécanisme de contrôle : tout processus réalisant une communication passe de l'état d'exécution à l'état de communication, et dans cette transition, le processus appelant donne le contrôle de l'exécution au processus appelé.

Un processus (P ou C) accède au noyau quand il exécute une demande d'interaction (ex. lecture d'une canal, termination d'une écriture, etc.) Cette demande se traduit par l'exécution de la primitve SWITCHP, en une émission de message adressée à l'activité communication/exécution du noyau. Une fois passé le contrôle d'exécution avec le message correspondant (la demande) au noyau, le processus appelant reste bloqué dans attente de la réalisation de sa demande.
Quand le noyau a le contrôle de l'exécution, il réalise les opérations suivantes : a) Validation de la demande ; b) Mis à jour des états des processus correspondants ; c) Sélection du prochain processus à exécuter (suivant la politique de service prédéfinie dans la procédure SCHD) et d) Transmisssion du contrôle de l'exécution au processus selectioné par la procedure SCHD avec le message correspondant.

A la fin de ces opérations, le noyau sera disponible à une nouvelle demande de service.

Le contrôle de la communication/exécution.

Pour la validation des communication/exécution des processus, le noyau utilise les tables d'états des processus P et C. Ces tables permettent au noyau la vérification des demandes des processus (détection de l'

exécution d'un processus inexistant, occurrence d'un événement invalide, etc.).

III.2.2.3 La gestion des contextes des processus P et C.

La tâche de création/destruction, surtout dans un environnement physiquement réparti, est une opération complexe qui peut demander plusieurs communications. Comme notre objectif est de construire un noyau minimum, cette tâche a été décomposée en deux parties : l'une réalisée par le processus système que nous appelerons Pcr et l'autre réalisée par le noyau lui même. Evidemment pour conserver un noyau minimum, celui-ci réalise uniquement la mise à jour de la table d'états des processus (introduction d'un nouveau processus, sortie d'un processus, modification de son contexte). Tout le reste, le protocole de création et de destruction est réalisé par le processus système Pcr. Il est évident, pour de raisons de sécurité, que la mise à jour des tables d'états sera un service accessible seulement au processus système Pcr.

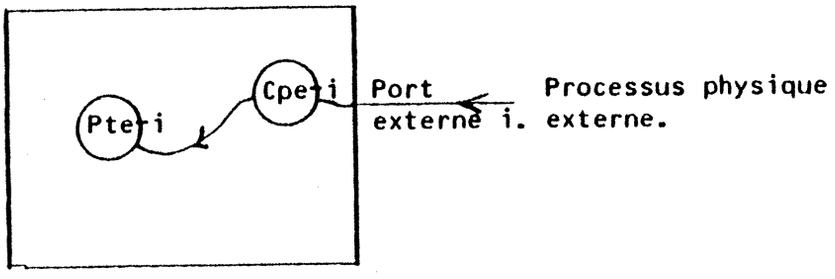
III.2.2.4 Le traitement des événements.

Tel qu'il a été défini, notre noyau ne répond qu'aux événement internes du système (demandes de communication ou changements de contextes). Mais si l'on veut que notre système soit adapté aux applications temps réel, le noyau doit aussi être capable de répondre aux événements provenant de ses ports externes (par exemple des interruptions externes ou opérations d'entrée/sortie provenants).

Pour une unification dans le traitement des événements (externes et internes), un événement est toujours associé aux opérations d'émission et de réception de messages.

Pour le traitement des événements externes, nous considérons les ports externes comme des processus canaux. L'écriture sur un de ces canaux est réalisée quand une information, provenant d'un processus physique extérieur au système, est envoyée au port externe auquel le processus physique est lié. La lecture de ces canaux est réalisée par les processus P qui communiquent avec ce type particulier de processus, les processus physi-

ques externes. Dans la figure III.2.2.2, nous montrons schématiquement le traitement des événements externes.



Pte-i...processus qui traite les messages provenants du port externe i.
Cpe-i...canal associé au port externe i.

Fig. III.2.2.2 Le traitement des événements externes.

Pour l'implémentation de ce schéma, nous associons à tout port externe i de la machine un "handler" Hi de bas niveau. Hi avec l'utilisation d'une primitive SWITCHP, transforme les informations provenant de l'ext2rieur en l'écriture d'un message au canal Cpe-i correspondant.

Hi est exécuté lors de l'arrivée d'une interruption au port externe i. Pour garantir un état cohérent dans les opérations de communication, les processus noyau et canaux ne peuvent pas être interrompus. Hi est exécuté directement sans contrôle du noyau.

Remarques.

Avec la transformation des interruptions en opérations sur des canaux, par les Hi correspondants, les événements externes sont traités par le noyau exactement comme les événements internes produits par les demandes d'interaction des processus P et C.

Avec cette démarche le noyau reste inchangé. L'introduction d'un nouveau port (par l'association de son "handler" spécifique et de son canal correspondante) ou la modification d'un port existant ne change en rien le comportement du noyau.

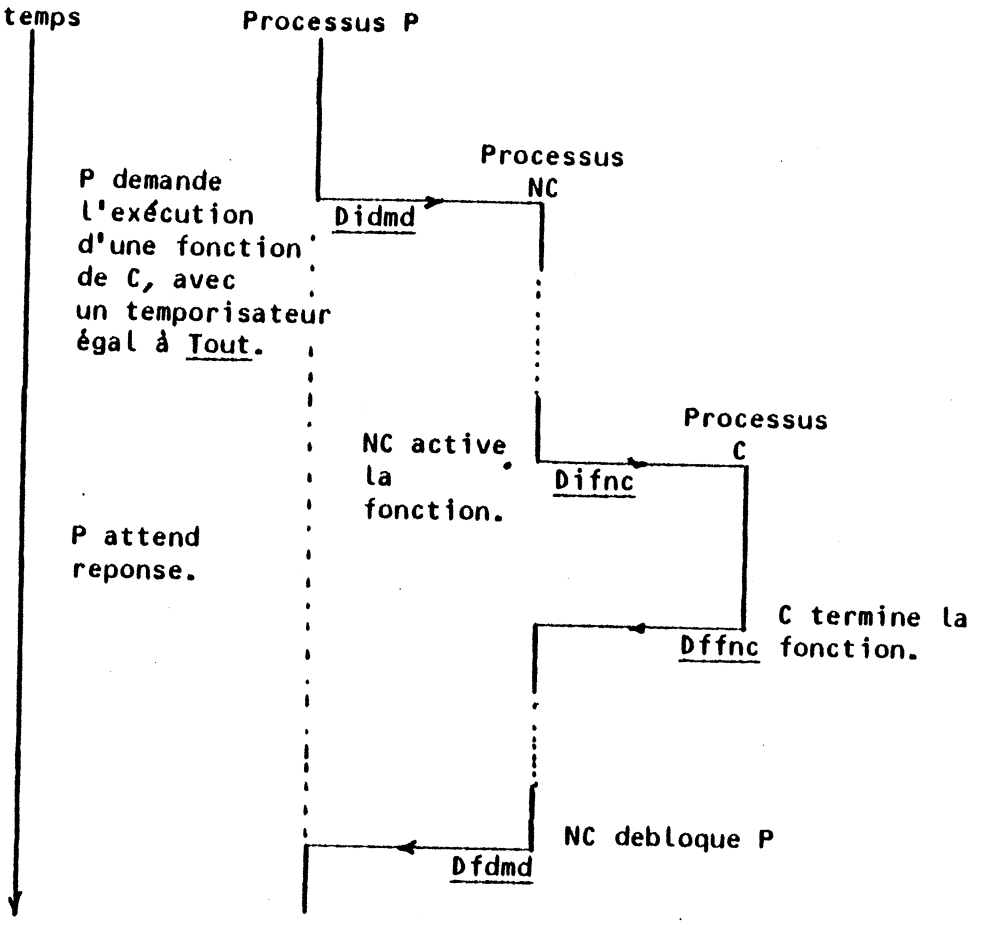
III.2.2.5 Le contrôle des temporisateurs.

Pour le contrôle des temporisateurs, associés aux fonctions d'accès aux canaux, nous utiliserons un canal particulier Ch.

L'accès à ce canal est direct (il n'est pas contrôlé par le noyau), Une lecture de Ch donne l'heure de l'horloge de la machine, et une écriture met à l'heure l'horloge.

Le canal Ch est à la disposition de tout processus qui est en train de s'exécuter.

Les primitives de communication du niveau application peuvent être affecté d'un temporisateur "Tout". Un temporisateur "Tout" sur une primitive, permet aux processus demandeur de préciser le temps maximal pour la réalisation de sa demande. Dans la figure III.2.2.4, nous montrons le déroulement d'une demande d'exécution d'une fonction avec temporisateur.



Evénements :
- Didmd...date : initialisation de la demande.
- Difnc...date : initialisation de la fonction.
- Dffnc...date : fin de la fonction.
- Dfdmd...date : fin de la demande.

Fig. 3.2.2.4 Le contrôle des temporisayeurs.

Pour le contrôle d'un temporisateur, on procède de la façon suivante :

- Le processus P avant de passer le contrôle au noyau, date sa demande (Didmd).

- Le processus NC, mémorise la date de la demande ainsi que le "Tout" de P et il continue ensuite son exécution normale.

- Le NC, avant l'activation de la fonction demandée, lit le canal Ch (Difnc) et fait le test suivant : si Dinfc-Didmd > Tout, la fonction n'est pas activée et le processus P est debloqué avec le code de retour "écoulement du temporisateur". Dans le cas contraire la fonction est activée normalement.

- A la fin de l'opération, avant le passage de contrôle au noyau, C datera l'événement de fin d'opération (Dffnc).

- Le processus NC fait le test suivant : si Dffnc-Didmd > Tout, le processus P est débloqué avec le code de retour "écoulement du temporisateur". Dans le cas contraire, le noyau continue son exécution normale.

- Avant l'exécution (continuation) du processus P, le noyau fait une lecture de Ch (Dfdm) et fait le contrôle suivant : si Dfdmd-Didmd > Tout, P est débloqué avec le code de retour "écoulement du temporisateur". Dans le cas contraire, P est débloqué avec le code "fin d'opération normale".

III.?.2.6 Le comportement du noyau.

En résumé, le noyau est le processus qui contrôle toute interaction entre les processus en offrant deux classes de services : une pour la communication/exécution disponible aux niveaux application et système et une deuxième pour la mise à jour des tables du noyau seulement disponible aux processus système.

Ces services, comme toute activité du modèle, sont accessibles aux processus, par l'envoi de messages sur leurs points d'entrées correspondants. Dans les lignes suivantes nous présentons la "programmation" du processus noyau.

```
Processus NOYAU (msg);

Begin

   COM : Begin   "Activité de communication/exécution"

          VALIDATION (msg) ; "Validation de l'interaction"
          MAJ (msg)        ; "Mise à jour des états des processus
                               correspondants"
          SCHD (msg)       ; "Passage du contrôle d'exécution"

         end;

   GES : Begin   "Activité de mise à jour des contextes des
                  processus"

          case (msg.tab) of
            "TABP" : MAJTBP (msg) ; "Mise à jour de la table P"
            "TABC" : MAJTBC (msg) ; "Mise à jour de la table C"
             .
             .

         end
end;
```

III.2.3 Les canaux et les service de communication.

Un processus canal a pour tâche le contrôle et la réalisation des opérations de communicatiion et de synchronisation effectuées par des processus P.

Le comportement d'un canal est déterminé, lors de sa création, à travres des paramètres. Le services de communication, la gestion des canaux et leur accès sont offerts aux processus P à travers un ensemble de primitives de communication par canaux (voir ANNEXE I).

La définition du comportement d'un canal.

Les paramètres de création d'un canal déterminent :
Son mode de communication :

- 1-1 mode point à point.
- n-1 mode serveur.
- 1-n mode client-multiserveur.
- n-n mode multiclient-multiserveur.
- dif diffusion.

Ses contraintes d'accès :

- nc nom du canal.
- ta type d'accès : public ou privé.
- lb longueur du buffer (de 0 à n).
- pr priorité.
- tv durée de vie du message dans le canal.

Les services de communication.

Les services de communication disponibles aux utilisateurs sont :

- La création/destruction dynamique des canaux.
- La liaison dynamique aux canaux.
- L'accès aux canaux (lecture et écriture).
- Le traitement des événements des canaux.

Les services de communication, de création/destruction et de liaison/déliaison sont réalisés par des processus système tandis que les services de lecture, d'écriture et de traitement des événements sont réalisés par les processus canaux.

III.2.3.1 La structure du processus canal.

Pour le contrôle de la communication et de la synchronisation, le canal a, dans sa zone de mémoire, les contraintes d'accès et l'état de ses processus communicants P. Ses activités réprésenten l'ensemble des services de communication offerts par le canal.

Le canal offre deux types de service, l'un pour la communication proprement dite (la lecture et l'écriture de messages) et l'autre pour le traitement des événements (un événement sera toujours associé à l'arrivée d'un message). La figure III.2.3.1 dessine la strucutre d'un canal.

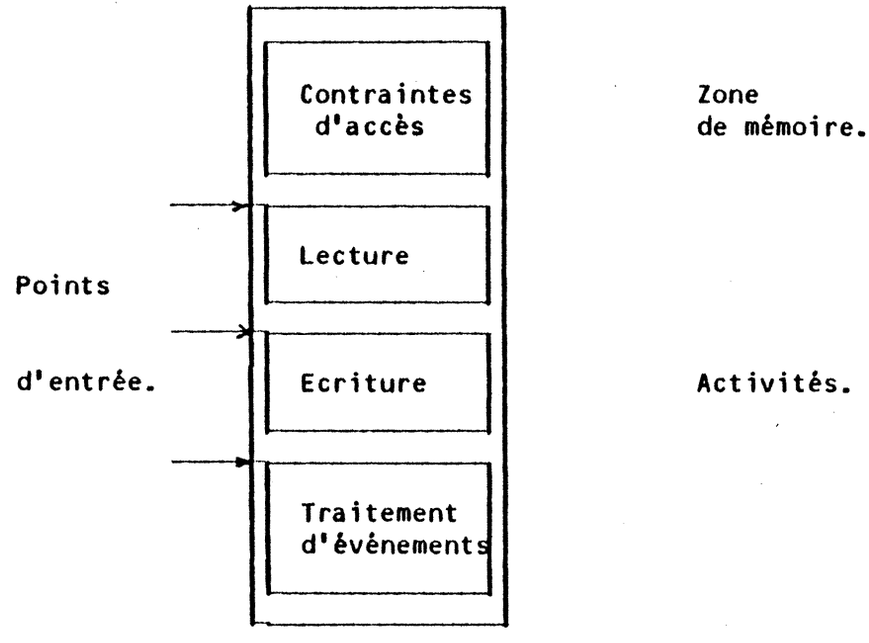


fig. III.2.3.1 La structure d'un canal.

Pour les opérations de lecture et d'écriture le canal a un comportement statique (il ne répond qu'aux demandes d'accès); par contre, pour le traitement d'événements inattendus, il a un comportement dynamique (il

peut signaler asynchronement l'arrivée d'un événement aux processus P). Ce dernier comportement nous semble important pour les applications qui ont besoin de la prise en compte "inmediate" des événements inattendus (par exemple l'écoulement d'une horloge, l'arrivée d'une signal critique, etc.)

Le fait d'avoir défini le canal comme un objet du type processus, facilitera aux utilisateurs la modification ou la redéfinition de nouvelles structures de communication. Plus concrétement, un processus canal permet :

- L'optimisation de l'implémentation des modes de communication existants sans modifier les processus communicants (exemple : l'optimisation d'un protocole).

- La définition de nouveaux modes de communication non prévus dans les modes prédéfinis.

- La redéfinition des modes de communication existants (exemple : la simplification d'un mode de communication).

III.2.3.1.1 Le mécanisme de désignation et de protection.

Au niveau de l'application, tout canal sera désigné par un nom logique choisi par le programmeur. Le système de communication associera au nom un identificateur interne unique, qui représentera le seul moyen d'accès au canal.

L'identificateur d'un canal est assigné par le système au moment de la création et le processus créateur devient son propriétaire.

Au moment de la création, un canal peut être défini comme public ou privé. Un canal public est accessible à tous les processus qui connaissent son nom logique. Un canal privé est accesible seulement par son identificateur, contrôlé par son créateur.

La gestion des noms des canaux sera réalisée par un des processus système, appelé Pcd.

III.2.3.2 Les services de communication.

Dans ce paragraphe, nous allons spécifier les services de communication. Pour une description plus détaillée des primitives voir l'annexe 1 : "Description des primitives d'accès".

III.2.3.2.1 La création et destruction dynamique des processus.

Nous avons introduit dans notre système de communication un service de création/destruction dynamique. Nous considérons que ce service est indispensable dans l'environnement dynamique des applications temps réel.

La gestion du service sera réalisée par un processus système appelé Pcd qui réalise les fonctions suivantes :

- La validation de l'opération.
- La mise en correspondance du nom logique avec son identificateur unique.
- La création du modèle de canal correspondant aux paramètres de création (mode de communication et contraintes d'accès).
- La mise à jour des tables du noyau (le nouveau processus sera mis dans un état initial neutre).

III.2.3.2.2 Les liens d'accès aux canaux.

Nous avons introduit dans notre système de communication un service de liaison dynamique. Pour le choix entre un service de liaison statique plus simple et un service de liaison dynamique plus puissant, nous avons considéré le même élément que pour la création des canaux : la nature dynamique de l'environnement des applications temps réel.

La gestion de ce service sera réalisée par le processus système appelé Pld qui remplira les fonctions suivantes :

- La validation de l'opération.
- L'établissement du lien d'accès.
- La mise à jour des tables du noyau (le canal passera de l'état neu-

tre à l'état d'attente, il sera prêt à répondre aux demandes de communication des processus P).

III.2.3.2.3 L'accès aux canaux.

Les fonctions fondamentales pour la communication et la synchronisation qu'un processus C offre, sont : la lecture et l'écriture de messages du canal. L'interaction entre les processus P et C a été définie comme synchrone, c'est à dire qu'un processus P demandant une lecture ou une écriture ne pourra poursuivre son exécution qu'après la réalisation de l'opération par C.

Avec les paramètres donnés au moment de la création du canal, on pourra définir plusieurs modes de communication de base.

Ainsi, avec le paramètre taille de buffer (lb), deux types de communication peuvent être définis :

- Sans buffer (lb=0), communication synchrone au niveau des processus P (exemple, le rendez-vous CSP).
- Avec buffer (lb 0), communication synchrone entre les processus P et C, mais asynchrone entre les processus P communicants (exemple, le pipe UNIX).

De même avec le paramètre "type d'interaction", on peut définir cinq modes de communication :

- 1-1 : point à point (1 lecteur - 1 écrivain).
- n-1 : serveur (n lecteurs - 1 écrivain).
- 1-m : client-multiserveur (1 lecteur - m écrivains).
- n-m : multiclient-multiserveur (n lecteurs - m écrivans).
- dif : diffusion.

III.2.3.2.4 Les événements des canaux.

Nous considérons que le non-déterminisme de la communication et l'occurrence aléatoire des messages sont des notions qui doivent être

considérees dans la définition d'un noyau de communication réparti temps réel.

Dans ce cadre, nous avons défini pour notre modèle un service de traitement d'événements.

Au niveau de l'implémentation, les messages et les événements sont toujours des messages qui sont échangés entre les différents processus du système. La différence entre ces deux types de message est dans l'information qu'ils contiennent.

- Le message représente l'unité d'information qui est échangée par les processus P à travers les canaux.

- L'événement représente le changement d'état du canal qui est produit par l'arrivée d'un message dans le canal.

Donc, tout événement sera toujours associé à l'arrivée d'un message sur un canal. Si un processus veut traiter un événement d'un canal, il doit d'abord se lier au canal. A l'arrivée de l'événement, le canal le signalera au processus demandeur.

Les événements qu'un canal peut recevoir sont :

-L'arrivée d'un message dans le canal.
-La sortie d'un message du canal.
-La liaison d'un processus au canal.
-La déliaison d'un processus du canal.
-Le canal vide.
-Le canal plein.
-Une erreur dans la communication.
-L'avortement du canal.
-L'avortement d'un processus lié au canal.
-L'arrivée d'un message de contrôle dans le canal.

Les services pour le traitement des événements sont :

-L'attente d'un événement sur un ou plusieurs canaux.
-Le traitement asynchrone des événements.

Avec le premier type de service, un processus qui demande la signalisation d'un événement reste bloqué jusqu'à l'arrivée de l'événement sur le canal correspondant.

Dans le deuxième cas, un processus peut demander la signalisation et le traitement asynchrone d'un événement sur un canal. Une fois acceptée la demande, si l'événement arrive le processus est interrumpu et la procédure correspondante est exécutée. A la fin de l'exécution, le processus continuera son exécution normale.

III.2.3.2.5 Le contrôle de la durée de vie des messages.

Nous avons associé au canal la notion de la durée de vie des messages. Un message ne pourra rester, dans le canal, plus du temps de vie predéfini au moment de la création du canal.

Pour le contrôle de la durée de vie d'un canal, on utilise le canal Ch pour la datation des événements. Les dates de tous les messages arrivés au canal (Damsg) sont mémorisés. Quand le canal est vide, avec l'arrivée du premier message, le canal mémorise cette date d'arrivée dans Ddmsg. Après, lors de l'arrivée de tout message, le canal fait le test suivant : si Damsg-Ddmsg > Temps-de-vie, alors, on efface tous les messages qui ont été dans le canal plus du temps de vie défini par le canal. Sinon, le canal mémorise la date d'arrivé du messsage et continue son exécution normale.

III.2.3.3 La programmation des canaux.

Dans les paragraphes précédents, nous avons décrit le comportement d'un canal général. Nous avons vu qu'un utilisateur peut exprimer plusieurs modes de communication à partir des paramètres donnés au moment de la création d'un canal. Nous avons décrit aussi les deux types de service qu'un canal offre : l'accès et le traitement d'événements. Avec tous ces mécanismes, l'utilisateur pourra exprimer différentes structures de communication classiques.

Mais il faut remarquer qu'un canal est un objet du type processus et l'utilisateur pourra toujours programmer des canaux spécifiques à ses besoins particuliers.

III.2.4 Les processus P.

Pour l'implémentation d'un système réparti (au niveau du système et au niveau de l'application), l'utilisateur programmera des processus P et utilisera, pour leurs interactions, les modèles de canal offerts par le système.

III.2.4.1 La structure d'un processus P et son initialisation.

Pour la programmation d'un processus, on utiliser la structure suivante :

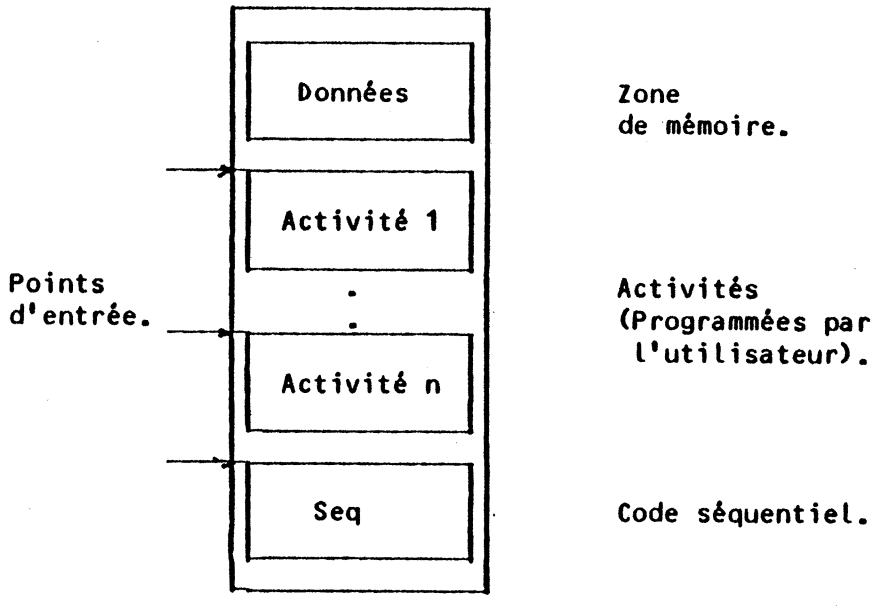


fig. III.2.4.1 Structure d'un processus P.

Un processus P est composé de un zone de mémoire commune et d'un ensemble d'activités pour le traitement asynchrone d'événements des canaux et d'une activité particulière appelée le code séquentiel qui est demarrée avec l'initialisation du processus.

La création et la terminaison d'un processus sont gerées par le processus système Pgen. Toutes les demandes seront envoyées à son canal correspondant.

Initialisation du processus.

L'initialisation d'un processus (P,C ou NC) est effectuée par l'intermédiaire d'un canal particulier appelé Co (canal d'origine). Co est automatiquement assigné par le système lors de la création du processus.

Un processus qui vient d'être créé se mettra automatiquement en attente du message d'initialisation sur son canal Co.

Pgen acitvera le processus en envoyant le message d'initialistion correspondant.

Après le démarrage du processus, le canal Co pourra être utilisé par par le programmeur pour d'autres communications.

Fin du processus.

Pour le contrôle de la fin d'exécution, le processus signalera l'événement "fin d'exécution" par l'entermediaire d'un message envoyé au canal du processus Pgen.

III.?.4.? La programmation de structures de communication à partir des canaux.

Dans ce paragraphe, nous allons donner des exemples pour la programmation de structures de contrôle plus complexes à partir des canaux. Les exemples montrés ici sont purement descriptifs, pour une information plus détaillée pour l'utilisation des canaux, voir l'annexe 1 (pour la description des primitives) et le chapitre IV (un exemple d'implémentation).

Au niveau de la programmation parallèle, nous trouvons deux modes de communication de base :

- Une communication "active" synchrone (rendez-vous CSP) ou asynchrone (PIPE d'UNIX).

- Une communication "par service" (appel à distance d'ADA).

Avec le premier, on n'a aucune relation hiérarchique entre les participants, chacun d'entre eux pouvant faire démarrer l'échange.

Par contre, avec le deuxième, nous trouvons une relation de client-serveur entre les processus communicants, et on a un processus qui active la communication.

Le premier mode de communication est directement implémentable avec les canaux. En plus, avec le paramètre longueur du buffer (lb) du canal on peut permeter les deux types de communication "active" : la communication synchrone sans buffer (lb=0) et la communication asynchrone avec buffer (lb>0).

```
Processus E;

(*idc identificatuer de canal.*)
(*msg pointeur de message.    *)

Begin
 .
 .
 ECRIRE(idc,msg,..);
 .
 .
end;
```

```
Processus R;

Begin
 .
 .
 LIRE(idc,msg,..);
 .
 .
end;
```

Par contre, le deuxième mode (appel à distance) est une opération plus complexe. Elle est implémentée de la façon suivante : le processus client passera au serveur dans le message de demande de service, l'identificateur du canal par lequel il désire la réponse.

```
Processus C;

(*idcd identificateur du canal de*)
(*     de demande.                *)
(*idcr identificateur de canal de*)
(*     de réponse.                *)
(*msg  pointeur du message.       *)

Begin
 .
 msg.IDCR=idcr;
 ECRIRE(idcd,msg,...);(*Demande de*)
                      ;(*service.  *)
 LIRE(idcr,msg,.....);(*Reponse.   *)
 .
 .
 .
 .
end;
```

```
Processus S;

Begin
  while true
  begin(*Service                  *)
   LIRE(idcd,msg,....);
   icdr=msg.IDCR;
   .
   .  "Service"

   .     (*Fin de service          *)
   ECRIRE(idcr,msg,..);
  end;
end;
```

fig. III.2.4.2.b Communication "par service".

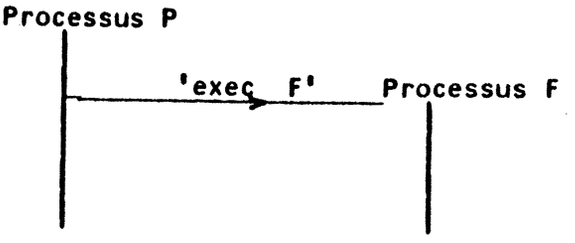
Avec cette approche, on peut aussi implémenter une demande de service asynchrone. Le processus client C, après sa demande peut continuer son exécution et demander la réponse plus tard.

III.2.4.2.1 La création/exécution parallèle des processus.

Une des commandes fondamentales de la programmation répartie est l'exécution en parallèle d'un processus.

A partir des mécanismes d'exécution parallèle qu'on trouve dans la littérature, nous pouvons classer l'exécution en deux modes :

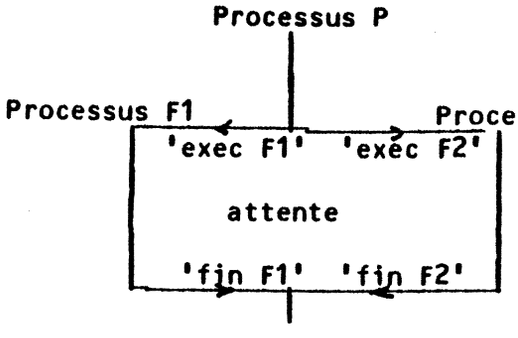
- Une exécution parallèle sans aucun contrôle explicite sur le déroulement du processus créé (exécution asynchrone du type fork (CONW 63b)).




```
Processus P;
Begin
  .
  fork (F);
  .
end;
```

'exec F' : signal qui déclanche l'exécution du processus F.

- Une exécution parallèle avec contrôle du processus créé (exécution synchrone du type Parbegin-parend (DIJK 68b) ou commande d'exécution parallèle de CSP (HOAR 78)).




```
Processus P;
Begin
  .
 parbegin
  F1;
  F2;
 parend;
end;
```

'exec P' : signal que déclanche l'exécution du processus P.
'fin P' : signal pour la synchronisation de la fin de P.

Nous pouvons constater que l'exécution synchrone est une opération composée. Pour sa mise en oeuvre, on doit d'abord générer "simultanément" des événements 'exec P' pour l'activation parallèle des processus et après attendre l'événement 'fin P' de tous les processus créés pour pouvoir continuer l'exécution.

La gestion de ces deux types de création sera pris en charge par le processus système Pgen.

Un processus P voulant exécuter des processus parallèles passera à Pgen la liste des processus à créer. Et si on a besoin de se synchroniser sur la fin des processus, un canal Cr de réponse, passé dans le message à Pgen, assurera le contrôle. P pourra donc se synchroniser avec la lecutr du message "fin de tous les processus" envoyé par Pgen.

"Exécution parallèle synchrone."

```
Processus P;

Begin
 (*idcr..identificateur du canal de réponse pour*)
 (*      l'exécution synchrone.                  *)
 .
 msg.cmd='créerp';
 msg.lp1=idf1;
 msg.lp2=idf2;
 msg.IDCR=idcr;
 .
 ECRIRE(idcgen,msg...): "Exécution synchrone"
 LIRE(idcr,msg,.....);
 .
 .
end;
```

III.2.4.3 Le traitement des événements.

Dans notre modèle, nous avons considéré l'événement comme un type particuier de mesage. Le traitement des événements nous permettra l'implémentation des opérations classiques de la programmation répatie, telles que l'attente d'un événement, le non-déterminisme et le traitement des événements externes.

III.2.4.3.1 L'attente sur un événement.

Cette opération est équivalente à une lecture, mais au lieu de lire un message, on aura un des états du canal.

Cette opération permet, par exemple, de savoir si un FIFO n'est pas vide sans avoir besoin de lire un des messages du canal. Ce type d'opération est très courant dans la probrammation réapartie.

III.2.4.3.2 L'attente sur un ensemble d'événements et le non-déterminisme.

L'expression du non-déterminisme est aussi une des notions fondamentales de la programmation parallèle. Elle nous permet d'établir une communication sur un ensemble de communications possibles. En regardant de plus près cette notion dans la commande alternative de CSP (HOAR 78), on peut remarquer, avec la figure suivante, la complexité de l'opération.

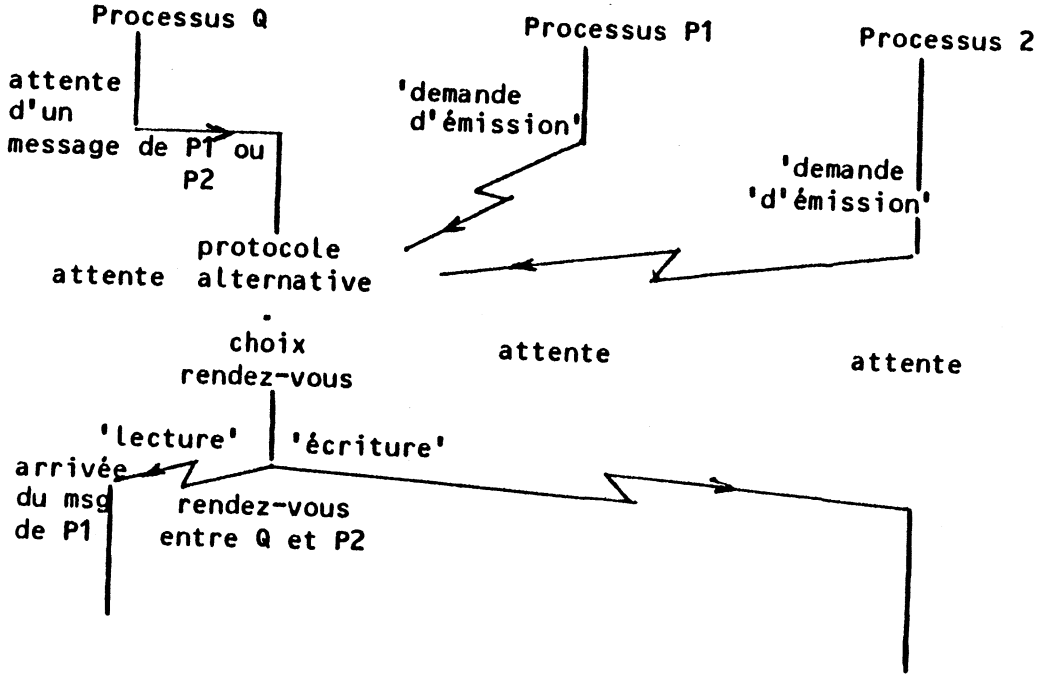


fig. III.2.4.3.2 La commande alternative avec des commandes d'entrée comme gardes.

Schématiquement, la suite des actions nécessaires à l'implémentation de cette commande est la suivante :

-Attente des demandes d'émission ;

-Choix de l'une d'entre elles si plusieurs sont arrivées en même

temps .

-Réalisation du rendez-vous entre les deux processus .

-Continuation des séquences des processus correspondants.

L'implémentation de cette commande devient encore plus complexe si on introduit les commandes de sortie dans les garde. Dans (SILV79, SILV81, SCHW78, BUCK83), on trouve des propositions d'impléméntation qui demandent l'utilisation des primitives plus élémentaires. Dans ce qui suit, nous ne prendons pas en compte ce cas là.

Pour l'implémentation de cette commande, nous pouvons utiliser l'attente d'un événement sur un ensemble de canaux.

Un processus pourra se mettre en attente de l'événement 'arrivée de message' sur l'ensemble des canaux (point à point). Le système débloquera le processus en lui donnant comme message de réponse, la liste des canaux qui ont reçu un message. Le choix et la continuation des séquences seront réalisés par un procédure prédefini.

"Programmation d'une alternative"

```
Processus P

Begin
 .
 .
 msg.EVE:='arrivée de message';
 msg.lst1:=idc1;
 .
 msg.lstn:=idcn;
 .
 ATTEVES(msg,...);      "Attente de l'événement 'arrivée"
                        "de message.                    "
 idc:=choix(msg.lst);   "Choix non-déterministe sur un  "
                        "des canaux.                    "
 LIRE(idc,msg,.....);   "Lecture du message.            "
 .
 .
end;
```

III.2.4.3.3 Le traitement asynchrone des événements.

Une des opérations fondamentales de la programmation temps réel est le traitement "immédiat" des événements externes inattendus et critiques.

Pour ce type d'opération, le noyau offre les services de traitement asynchrone des événements. Pour leur programmation, on procédera de la façon suivante ·

Un processus qui veut traiter de façon asynchrone l'événement d'un canal, devra d'abord se lier au canal. Il pourra, ensuite, demander le traitement asynchrone de l'événement en lui associant une activité. Si l'événement arrive, le processus sera interrompu par le système et l'activité correspondante exécutée. Si le processus ne veut plus prendre en compte cet événement, il pourra demander la désactivation du traitement.

```
"Traitement asynchrone d'un événement."

    Processus P;

      "Activités du processus :                                    "

      Activité1();      "Activité pour le traitement
                         asynchrone de l'événement 'eve X'
                         du canal idc.                            "
      begin
       .
       .
       .
      end;
      .
      .

      "Code séquentiel du processus :                              "

      Begin

        .
        LIERCANAL (idc,S,...);        "Liaison à la sortie (S)
        .                              du canal idc.
        .
       "Activation de la  procédure 'acitivité1' associée à
        l'événement 'eve X' du canal idc.                          "

        ACTIVE (idc,'eve X','Activité1',..);
        .                             "Le processus devient        "
        .                              interruptible.              "
        .
        .
        MASK (idt,..);                "L'interruption est
        .                              masquée.                    "
        .
        DMASK (idt,..);               "L'interruption est
        .                              demasquée.                  "
        .
        DACTIVE(idc,'activité1',...):"Désactivation de la          "
        .                              procédure asynchrone.       "
        .

       end
```

Une des opérations courantes de la programmation temps réel est le masquage et le démasquage du traitement asynchrone des événements. Avec ces deux notions, on peut rendre une sequence de code du processus non interruptible.

Nous avons intégré dans le service de traitement d'événement ce service à travers deux primitives MASK et DMASK. Dans l'exemple antérieur, on montre l'utilisation de ces primitives.

III.2.4.4 Le temps réel.

Pour la programmation temps réel, l'utilisateur aura à sa disposition des services spécifiques offerts par le noyau et que nous avons déjà décrits dans les paragraphes précédents et que nous résumons ici :

-La hiérarchisation dans l'exécution des processus P et C.
-Le mécanismes de priorités.
-Les temporisateurs dans les primitives d'accès au canaux.
-Le paramètre 'durée de vie d'un message'.
-L'exécution asynchrone d'activités associées à l'arrivée d'un événement.
-Le "masquage" et le "démasquage" des séquences de traitement.
-La programmation de processus réveils ou horloges logiques, programmées à partir de l'horloge physique de la machine cible.
-La programmation de canaux spécifiques pour les interfaces non standard d'une application.

III.2.4.5 La répartition.

La communication par canaux au niveau de la programmation de l'application est indépendante de l'implémentation et de la localisation des processus canaux.

Au niveau système, dans le cas d'un environnement composé d'un ensemble de sites reliés par un réseau local, les processus canaux seront plus élaborés que les canaux pour des communications locales.

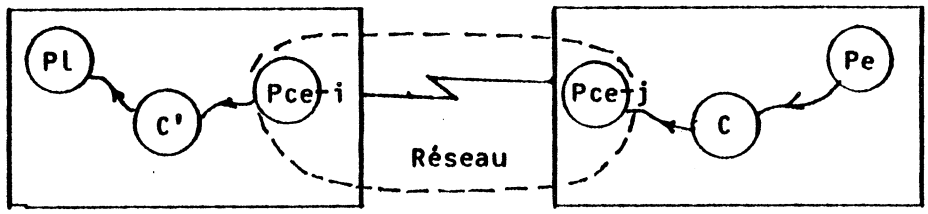
Dans la figure III.2.4.5.a, on montre une communication, au niveau application, entre un processus lecteur Pl et un processus écrivain Pe à

travers le processus C. La figure III.2.4.5.b montre l'implémentation du canal C, au niveau système, dans un environnement réparti.

Pl processus lecteur.
Pe processus écrivant.
C processus canal.

fig. III.2.4.5.a Une communication au niveau de l'application.

Après l'opération de création du canal C (dans cet exemple par Pe) et les opérations de liaison correspondantes (Pl en lecture et Pe en écriture), les processus Pl et Pe peuvent communiquer par l'intermédiaire du canal C.



canal réparti

Site i Site j

Pl : processus lecteur.
Pce-i: processus pour la communication externe du site i.
C' : processus canal.

Pe : processus écrivant.
Pce-j: processus pour la communication externe du site j.
C : processus canal.

fig. III.2.4.5.b Une implémentation du canal C dans un environnement réparti.

Dans un environnement un réparti, le canal C est décomposé par le système en un ensemble de processus.

Dans cet exemple le processus écrivain Pe et le canal C (crée par Pe)

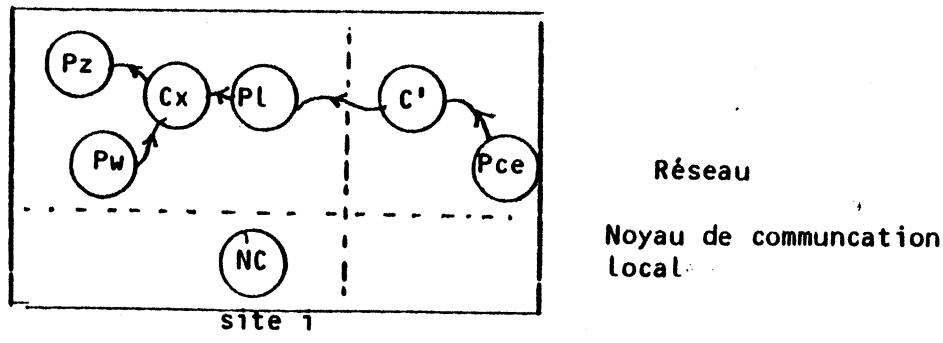
résident dans le site j, tandis que le processus lecteur Pl se trouve dans le site i. Pour la communication entre les processus Pe et Pl le système a crée le canal composé C'-Pcei-Réseau-Pcej-C.

Au niveau de l'application Pl et Pe communiquent toujours par le canal C. Le canal C' du site i, créé par le système, a le même nom que C.

III.2.4.5.1 Le système de communication externe.

Pour la répartition de la communication, transparente à l'utilisateur, chaque site a un noyau pour la gestion des communications locales et un système de communication externe (un processus système Pce et un ensemble de canaux C' crées et contrôlés par Pce) pour les communications externes avec les processus des autres sites.

Pce se charge de la répartition de la communication à travers ses processus C' et le canal réseau. La figure suivante dessine un site avec son processus noyau NC et son système de communication externe.



Système de communication externe.

Pce et C' : processus du système de communication externe.
Pz, Pw, Cx, Pl, C' et Pce : processus locaux au site i gérés par le processus noyau NC.

fig. III.2.4.5.1 Le système de communication externe.

III.2.4.5.2 La gestion de la communication externe.

Les canaux C' sont créés et contrôlés par le processus Pce lors de la liaison des processus P avec de canaux non locaux.

Toute demande de liaison sur un canal C non existant dans le site i, est ré-émis par le processus gestionaire des liaisons au processus Pce-i. Ce processus établit un dialogue avec le Pce-j du site i où le canal C a été crée. Une fois établies les liaisons correspondantes entre les processus C' et C, les communications entre les processus locaux et le processus Pce. seront gérées par le noyau comme n'importe quelle communication locale. Le processus Pce se charge de l'émission et de la réception des messages en suivant un protocole de communication défini pour le type de réseau utilisé.

Par exemple, en prenant la figure III.2.4.5.b, la répartition d'un canal se réalise de la façon suivante :

Le processus Pl du site i a fait une demande de liaison en lecture du canal C. Comme ce canal n'existe pas dans ce site, le processus Pce-i (aprés un protocole de récherche de C) crée C' et réalise les opérations suivantes :

-une liaison en écriture à C' :

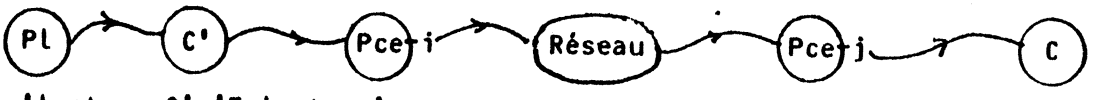
-une liaison en lecture pour Pl et

-une demande à C' de la signalisation de l'événement, demande de lecture 'E-lecture'.

Dans le site i, le processus Pce-i s'est lié en lecutre au canal C.

Après ce protocole de liaison les processus Pl et Pe seront reliés entre eux à travers le processus canal composé par les processus suivants : C'-Pcei-Réseau-Pcej-C.

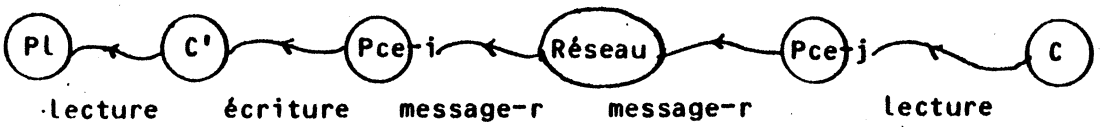
Une opération de lecture par Pl sur C' sera signalé à Pce-i, qui transmettra la demande de lecture au Pce-j (en suivant un protocole de communication réseau), qui demandera à sont tour la lecture de C. Cette opération est montrée dans la figure suivante :



site i :
  'lecture-C' demande de lecture sur C associée internement sur C'.
  'E-lecture' événement de C' signalé à Pce-i.
réseau :
  message-r message réseau émis par Pce-i et reçu par Pce-j.
site j :
  'lecture-C' demande de lecture de C.

fig. II.4.5.2.a Demande de lecture du canal C distant par le processus Pl local.

Une fois que le canal a été lu, Pce-j transmet le contenu au Pce-i qui à son tour, le transmettra à Pl par l'intermédiaire de C'. Dans la figure suivante on montre la conclusion de l'opération.



site j :
  lecture message lecture du message de C par Pce-j.
réseau :
  message message réseau émis par Pce-j et reçu par Pce-i.
site i :
  écriture message écriture du message de C par Pce-i.
  lecture message lecture du message de C par Pl.

fig. II.4.5.2.b Lecture du canal C distant par le processus Pl local.

Pour le contrôle des temporisateurs sur les fonctions d'accès aux canaux répartis, chaque Pce utilise l'horloge locale au site (Ch) pour la

datation des opérations. Un algorithme pour la synchronisation des horloges locales (RIVE 83, MINT 84) sera mis en route par les processus Pce de chaque site.

A chaque demande d'opération temporisée sur un canal distant, le processus Pce datera le message à émettre au Pce distant et mémorisera le temps limite prévu pour cette opération. Pce surveillera périodiquement l'état de ses temporisateurs. S'il y en a un qui est écoulé, le Pce débloquera le processus demandeur de la fonction avec le code de retour "temporisateur écoulé". Toute réponse arrivée aprés le temps prévu sera ignorée par Pce.

Il faut remarquer que la communication externe est transparente au noyau et aux processus application. La tâche du processus noyau reste toujours la gestion des ses processus locaux (P, C, Pce, C', etc.). Par contre la gestion de la communication inter-réseau est la tâche d'un processus spécialisé (Pce). Avec cette approche, des changements de protocole ou de réseau seront faits au niveau du processus Pce et ils n'auront pas d'influence sur la structure du noyau.

La spécification de la répartiton des canaux, avec la spécification des protocoles de création-destruction, de liaison-déliaison et d'accès, la gestion de noms et les algorithmes pour la répartiton du temps à partir de horloges locaux, etc. est l'objet d'un travail de recherche qui est en cours de réalisation. Dans ce travail, nous avons présenté seulement une première proposition pour la répartition des canaux.

CHAPITRE IV

UN EXEMPLE D'IMPLEMENTATION DU MODELE

INTRODUCTION.

Dans ce chapitre, nous allons présenter une implémentation de notre modèle de canaux.

Pour cette version, nous avons utilisé le système d'exploitation UNIX du VAX.

Les processus noyau et canaux ont été programmés dans le langage C tandis que la programmation de l'interface a demandé, en plus, l'utilisation des primitives du noyau UNIX.

Avant de décrire l'implémentation et de discuter les résultats obtenus, nous allons faire une présentation générale d'UNIX et de ses primitives de communication et de synchronisation.

IV.1 Le système d'exploitation UNIX (RITC 74).

UNIX (RITC 74) est un système d'exploitation multiutilisateurs interactif qui a été conçu pour des applications du type temps partagé. Ce système a été créé en 1969 pour la gamme des miniordinateurs PDP-9 et PDP-11.

Actuellement, il existe différentes versions d'UNIX (la version de Berkeley, la version 7, la version III, etc) ; on le trouve dans de microordinateurs (systèmes VENIX, ONIX, etc) et dans de grands systèmes (VAX-11/780, IBM/370 sur VM, etc.).

La popularité de ce système d'exploitation est due à sa simplicité et aux facilités qu'il offre pour la mise à point du logiciel. Le caractéristiques principales de ce système sont :

- Un noyau simple, les primitives offertes à ce niveau, sont peu nombreuses mais puissantes.
- Son système de fichiers, un système hiérarchique et d'utilisation simple.
- Son langage de commande, qui est aussi un langage de programmation.
- Son système d'entrée et des sortie, un système de manipulation simple et souple.
- Sa bibliothèque d'utilitaires avec une collection importante de logiciels (éditeurs, compilateurs, etc).

IV.1.1 Les primitives de communication et synchronisation.

IV.1.1.1 La notion de processus UNIX.

Un processus UNIX est défini comme l'exécution d'une image constituée des éléments suivants :

- L'image en mémoire d'un programme (les segments : code, pile et données) ,
- Ses registres ,
- Les états des fichiers ouverts et de son répertoire courant.

Le segment de données est local à chaque processus et ne peut pas être partagé. Un processus est toujours lié à un terminal et à un utilisateur et ce lien est indestructible.

IV.1.1.2 La gestion des processus.

Pour assurer l'exécution en temps partagé de ses processus, le système UNIX utilise le "swaper" pour la gestion de l'espace mémoire et du "scheduler" pour la gestion du CPU.

Le scheduler est basé sur une priorité dynamique associée internement à chaque processus. Cette priorité est fonction du rapport temps de calcul et temps réel consommé par le processus, et elle est réévaluée toutes les secondes. Avec ce mécanisme, un processus qui utilise une priorité élevée pour s'appropier du système, verra sa priorité décroire avec le temps. Par contre, un processus à basse priorité qui attend depuis longtemps vera sa priorité augmenter.

Le swapper gère l'allocation de la mémoire centrale avec des écchanges ("swapping") vers et depuis une zone de mémoire secondaire. L'algorithme pour charger un programme dans la mémoire centrale, est basé sur le temps d'attente en mémoire secondaire (le processus qui est dehors depuis longtemps sera le premier à être chargé). Pour la sortie d'un processus de la mémoire centrale, l'algorithme prend d'abord en compte ceux qui attendent des événements "lents", en fonction de leur âge en mémoire centrale, toujours avec des pénalités de taille.

Il faut remarquer que le "swapper" et le "scheduler" ont été conçus pour favoriser les processus du type interactif.

IV.1.1.3 La création dynamique des processus.

Mis à part les processus qui sont lancés automatiquement lors du démarrage du système, toute création d'un nouveau processus est faite à l'initiative d'un utilisateur sur un terminal.

Pour la création d'un processus, le noyau UNIX offre la primitve "fork". Lors de l'opération "fork", UNIX crée un nouveau processus qui est une copie conforme du processus créateur : les données, la pile et le code (s'il n'est pas partagé) sont copiés et tous les fichiers et les "pipes" (objets de communication) ouverts par le créateur sont hérités.
L'ensemble des processus descendant d'un même père forment une classe.

Pour l'exécution d'une image, le noyau UNIX offre la primitive "exec". Lors de l'opération "exec", l'image du processus appelant est remplacée par l'image du processus à exécuter.

IV.1.1.4 La communication entre processus.

Les processus ne partagent pas de zone de mémoire. Pour la communication et la synchronisation des processus, UNIX offre deux objets : les signaux et les pipes. Les fichiers peuvent être aussi utilisés comme objets de communication et de synchronisation, mais à cause de la lenteur des opérations sur les disques, ils ne seront pas considérés ici.

Les "pipes". Un "pipe" peut être considéré, comme un flux de données alimenté par un ou plusieurs processus écrivains et dans lequel des processus lecteurs peuvent extraire de l'information.
Un pipe permet la communication entre processus qui ont un même origine.
Pour créer un "pipe" un processus exécute la primitive PIPE et le système lui transmet en retour l'identificateur local du pipe. Le processus créateur d'un pipe devient son propriétaire. La seule façon de transmettre le nom du pipe à d'autres processus est à travers la primitive de création de processus (fork) : après l'exécution du fork, le processus fils hérite tous les pipes qui ont été ouverts par son père avant sa création. Pour l'accès aux pipes, UNIX offre les mêmes primitives d'accès aux fichiers : READ et WRITE, avec la différence que les primitives sur les pipes sont bloquantes. Si le pipe est vide, un processus lecteur reste bloqué tant qu'il n'a aucun caractère à lire. Si le pipe est plein, un processus écrivain réste bloqué tant qu'il n'y a pas suffisamment de place dans le pipe.

Il faut remarquer premièrement, qu'il n'existe pas de notion de nom global pour les pipes (la connaissance d'un pipe se fait par héritage) ; deuxièmement, qu'il n'existe pas de notion de message pour l'accès aux pipes, un processus lecteur peut lire un nombre d'octéts différent du nombre écrit par un processus écrivain ; et troisièmement, que la communication est restreinte aux processus formant une relation père-fils.

Les signaux. Un certain nombre d'événements, matériels et logiciels, peuvent être signalés par UNIX aux processus.
Si un processus ne demande pas le traitement de ces événements par la primitive SIGNAL, l'arrivée éventuelle d'un de ces événements provoquera sa terminaison. Par contre, un processus demandant le traitement d'un événement, pourra l'ignorer soit donner le nom d'une procédure à exécuter à chacune de ses occurences.
Le système ne mémorise pas les événements et l'information contenue dans l'événement est extrémmenent limitée : le numéro du signal.

Il faut remarquer que les événements d'UNIX ont été conçus essentiellement comme un moyen de traiter les événements anormaux et de contrôler l'exécution des processus en permettant de mettre fin à leur exécution en cas d'erreur.

IV.2 La programmation du modèle.

Pour l'implémentation du modèle, on a d'abord programmé le mécanisme d'appel et en suite, à partir de cet outil, on a programmé les processus noyau et canaux.

Pour l'implémentation du modèle, nous avons suivi les spécifications données dans les annexes I, II et III. Dans ce paragraphe, nous présentons les points les plus importants de l'implémentation.

Pour rendre les algorithmes les plus clairs possibles, nous avons utilisé un pseudo langage C. Mais, à part quelques détails sur les déclarations et initialisations de variables, les algorithmes introduits ici sont les mêmes que ceux que nous avons utilisés.

IV.2.1 La programmation de la structure de base.

L'interface comme elle a été définie dans III.2.1, se compose d'un modèle de processus et de son mécanisme élémentaire de contrôle : le mécanisme d'appel.

Comme le mécanisme d'appel est la structure de contrôle la plus élémentaire de notre modèle et en même temps il représente la couche la plus proche de la machine cible, on a été obligé d'utiliser des primitives de communication et de synchronisation spécifiques du système utilisé pour son implémentation.

Dans ce paragraphe, nous développons une implémentation de l'interface. Nous avons utilisé pour sa programmation le langage C et quelques primitives du noyau UNIX.

IV.2.1.1 La structure d'un processus.

Un processus a été défini comme un ensemble d'activités qui partagent une zone de mémoire commune. Cette zone de mémoire ne peut être accessible par aucun autre processus.

La notion du processus UNIX par rapport à sa zone de mémoire non partagée, s'adapte bien à la notion de processus de notre modèle, donc de

ce point de vue, les deux notions sont équivalentes.

Pour l'implémentation des activités, nous utilisons la notion classique de procédure et pour la zone de mémoire commune, nous utilisons des variables globales.

Un processus a la structure suivante :

```
int .....;
real ....;                    zone de variables globales.
.
.

act1() (.....);               act1, act2, .. , actn : noms logiques et
act2() (.....);                                       locaux des activités du processus.
.
.
actn() (.....);

pseq() (.....);               pseq : nom de l'activité séquentielle du
                                    processus.
```

IV.2.1.2 Le mécanisme d'appel.

Le mécanisme d'appel se compose de deux structures de contrôle élémentaires, une pour la reprise du contrôle de l'exécution (associée à la réception d'un message par le processus) et une autre pour le passage du contrôle de l'exécution (associée à l'émission d'un message).

En regardant les primitives de communication et de synchronisation d'UNIX, nous n'avons eu que deux possibilités pour la programmation de l'interface : les signaux et les pipes.

A cause du caractère très restreint des signaux (manque de mémorisation des signaux, impossibilité d'avoir plus d'information sur l'origine du signal, etc. pour plus de détail voir IV.1) nous ne les avons pas utilisés pour la programmation de l'interface.

Par contre, les pipes sont des outils plus évolués, ils se sont donc mieux adaptés à nos besoins.

Les primitives définies pour notre mécanisme de contrôle sont :

- SWITCHP : pour l'émission du message, le blocage de l'exécution du processus émetteur et le passage du contrôle d'exécution au processus source.

- MULTIPLEX : pour la réception du message, le déblocage du processus récepteur et la reprise du contrôle d'exécution sur un des points d'entrée.

A partir de ces comportements, nous avons programmé l'interface en utilisant des pipes UNIX de la façon suivante :

- Tout processus du système a un pipe associé appelé "pipe de contrôle"

- Un processus est initialisé avec une lecture sur son pipe de contrôle (le processus sera donc dans un état d'attente d'exécution).

- Le processus ne pourra s'exécuter que si un message est envoyé sur son pipe de contrôle.

- Lors de l'arrivée d'un message, le processus exécute l'activité signalée dans le message.

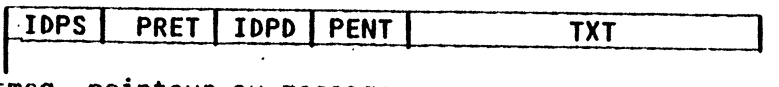
- Un processus qui a le contrôle de l'exécution ne le quittra, qu'avec l'envoi explicite d'un message, avec une écriture sur le pipe de contrôle du processus cible.

- Après l'émission du message, le processus restera bloqué en lecture sur son pipe de contrôle, dans l'attente de la réception d'un message pour retrouver le contrôle de l'exécution.

IV.2.1.2.1 La structure des données et du message du mécanisme d'appel.

Pour le contrôle de l'exécution, tout processus aura un ensemble de variables et des constantes dans sa zone de mémoire commune.

Tout message à échanger a la strucutre suivante :



ptmsg..pointeur au message.

IDPS...identificateur du processus source.
PRET...point de retour.
IDPD...identificateur du processus destinataire.
PENT...point d'entrée.
TXT....texte.
  TEXTE
    LGTXT...longueur du texte.

La zone de mémoire commune est composée des suivantes variables :

Constantes :

```
/*Structure du message.*/
   NBMAXMSG=m      /*Taille du message*/
   IDPS=0
   PRET=1
   IDPD=2
   PENT=4
   TXT =5
/*Points d'entrée.*/
   INITSEQ=0       /*Points d'entrée de contrôle.*/
   CONTSEQ=1
   ERREUR =2

   PENT1=10        /*Points d'entrée de l'utilisateur.*/
   PENT2=12
   .
   PENTn=n
```

Variables :

```
   idp       /*Identificateur du processus.*/
   ppctrl    /*Identificateur du pipe de contrôle du processus.*/
   lppctrl   /*Liste de pipes de contrôle des processus externes.*/
```

IV.2.1.2.2 Programmation de la primitive MULTIPLEX.

Au moment de la réception d'un message, cette primitive doit assurer le déblocage du processus et l'exécution de l'activité correspondante.

En profitant de la lecture bloquante des pipes, nous avons pu intégrer facilement les opérations de lecture du message et de déblocage du processus. Un processus après toute émission, est mis en attente par la primitive MULTIPLEX, avec une lecture sur son pipe de contrôle. Si le pipe est vide, le processus reste bloqué jusqu'à l'arrivée d'un nouveau message.

Pour l'exécution des activités, nous utiliserons la structure de contrôle 'case' du langage C.

Les points d'entrée d'un processus sont classés en deux types. Les premiers, appelés points de contrôle, permettent au mécanisme d'appel d'obtenir le contrôle de l'exécution, sur les points suivants :

- INITSEQ : pour l'initialisation du processus et l'exécution éventuelle de l'activité code séquentiel.

- CONTSEQ : pour assurer la continuation sur une séquence (point d'entrée variable, voir III.2.2).

- ERREUR : pour la récupération des erreurs.

- default : pour le multiplexage des acitivités définies par le processus.

Ces points d'entrée sont toujours associés à chaque processus et transparents à l'utilisateur.

La deuxième série de points d'entrée est définie pour les activités programmées par l'utilisateur.

Le paragraphe suivant indique les procédures correspondantes à la primitive MULTIPLEX (attdexec, init, cont, erreur et pents). Il faut remarquer que la procédure "attdexec" est la seule qui fait appel à la primitive READ d'UNIX.

```
/*Primitive MULTIPLEX : Pour l'attente-lecture et l'exécution d'une     */
/*                      activite.                                       */
/*ptmsg ..pointeur du message.                                          */

    multiplex(ptmsg)

    (switch (attdexec(ptmsg))                  /*Attente d'un message     */
      (
       case INITSEQ : init(ptmsg);break;       /*Pour l'initialisation du */
                                               /*processus.               */
       case CONTSEQ : cont(ptmsg);break;       /*Pour la continuation d'une*/
                                               /*séquence.                */
       case ERREUR  : erreur(ptmsg);break;     /*Pour la récupération     */
                                               /*d'erreurs.               */
       default      : pents(msg);              /*Pour l'initialisation des */
      )                                        /*activités de l'utilisateur.*/
    )



    /*Procédure spécifique d'UNIX. Lecture du pipe de contrôle (ppctrl)*/
    /*pour le déblocage du processus et la réception du message.  */  */
    attdexec(ptmsg)

    (ptmsg=msg;
     read(ppctrl,ptmsg,NBMXMSG);           /*lecture du pipe de contrôle  */
     return(msg(PENT));                    /*retur avec le point d'entrée */
    )                                      /*à exécuter (PENT)            */



   /*Procédure pour l'initialisation d'un processus :                     */
   /*Initialisation des variables de contrôle, et exécution eventuelle*/
   /*du code séquentiel du processus.                                     */
   init(ptmsg)
     (.."Récupération de : l'identificateur du processus (idp), de
         son pipe de contrôle (ppctrl) et de la liste de pipes de
         contrôle des processus externes (lppctrl)"...
     )

   /*Procédure pour la récupération du contrôle sur un point variable.*/
   pcont(ptmsg)
      (.."mémorisation de l'adresse de la dernière instruction
          exécutée (point d'entrée variable)"...
      )
```

```
/*Procédure pour le traitement des erreurs.                    */
erreur(ptmsg)
   (.."signalisation de l'erreur"..
   )



/*Les points d'entrée de l'utilisateur.                        */
pents(ptmsg)

(switch (msg(PENT))
   case PENT1   : act1();break;      /*Points d'entrée prédéfinis */
   case PENT2   : act2();break;      /*par l'utilisateur.         */
   .                                 /*act1,..,actn() activités   */
   .                                 /*programmées par l'utili-   */
   case PENTn   : actn();break;      /*sateur.                    */
)
```

IV.2.1.2.3 La primitive SWITCHP.

La primitive SWITCHP doit assurer l'émission du message et en même temps le blocage du processus émetteur.

L'émission d'un message est assurée par l'écriture du message sur le pipe de contrôle du processus cible. Le blocage du processus est réalisé par l'exécution d'une lecture sur son pipe de contrôle (exécution de la primitive MULTIPLEX). La primitive SWITCHP fait appel à la primitive READ d'UNIX.

Remarques.

Pour la programmation de l'interface nous n'avons utilisé que trois primitives du noyau UNIX : pipe pour la création des pipes, read pour la lecture des messages et le blocage des processus et write pour l'émission des messages.

De toute l'interface, nous n'avons que deux procédures qui dépendent des primitives d'UNIX, la procédure attdexec (avec un read) pour la lecture de messages et le blocage du processus et la procedure switchp (avec un write) pour l'émission des messages.

En résumé un processus a la structure suivante :

```
/*                    PROCESSUS                                    */
/*        Zone de mémoire propre au mécanisme d'appel              */
.
.
main ()
(..
  VIVANT=OK;/*Le processus est toujours prêt à la réception des    */
  while (VIVANT) multiplex(ptmsg)  /*            messages          */
)

/*                  Mécanisme d'appel                              */

multiplex (ptmsg) (...): /*Pour la réception d'un message.        */
switchp   (.....) (...); /*Pour l'émission d'un message.          */

/*       Activités de contrôle du mécanisme d'appel.               */

  init  (ptmsg) (...);   /*Pour initialiser le processus.         */
  cont  (ptmsg) (...);   /*Pour le contrôle de la continuation.   */
  err   (ptmsg) (...);   /*Pour le traitement des erreurs         */
  pents (ptmsg) (...);   /*Pour la commutation des activités      */
                         /*de l'utilisateur.                      */
/*                                                                */

/*     Les activités programmées par l'utilisateur.               */

integer msg(NBMAXMSG),*ptmsg;
........          ;  /*Zone de variables communes aux activités.  */

act1 (ptmsg) (...);
act2 (ptmsg) (...);
.
.
actn (ptmsg) (...);

pseq (ptmsg) (...);
```

IV.2.2 La programmation du processus noyau.

Une fois définie la structure de base, la programmation d'un processus consiste en la programmation de ses activités. Pour la communication avec les autres processus, l'utilisateur n'aura qu'une seule primitive à sa disposition : la primitive SWITCHP.

La tâche du processus noyau, comme elle a été définie, est la gestion des interations entre les processus P et C. Son comportement se résume à deux fonctions élémentaires :

- Prise en compte de la demande d'interaction, avec la mise à jour des processus impliqués.

- Réévaluation du prochain processus à activer (scheduling).

Pour la gestion des interactions, le noyau a besoin des contextes des processus P et C et des informations nécessaires pour le contrôle de la commutation et le cadencement des processus.

IV.2.2.1 La structure de données du processus noyau.

Dans la zone de mémoire local du processus noyau, on a tous les contextes ainsi que les informations de contrôle nécessaires à la gestion des processus.

Les contextes des processus.

table tabp(idp,pr,et,..) ;

tabp table des contextes des processus P.
  idp identificateur du processus P.
  pr priorité.
  et état du processus.
  .
  .

table tabc(idc,pr,et,..) ;

tabc table des contextes des processus C.
  idc identificateur du processus canal.
  pr priorité.
  et état du processus.
  .
  .

Tables de contrôle.

Pour la mémorisation des appels et des retours des processus P et C.

table laplp(idp,..)

laplp liste des appels aux processus P.
  idp identificateur du processus P.
  .
  .

table lretp(idp,...)

lretp liste des retours aux processus P.
  idp identificateur du processus P.
  .
  .

table laplc(idc,...)

laplc liste des appels aux processus C.
  idc identificateur du processus C.
  .
  .

table lretc(idc,..)

lretc liste des retours aux processus C.
  idc identificateur du processus C.
  .
  .

Les listes d'attente du scheduler.

table qp(idp,..)

qp liste des processus P en attente d'être exécutés.
idp identificateur du processus P.
.

table qc(idc,..)

qc liste des processus C en attente d'être exécutés.
idc identificateur du processus C.
.

Constantes internes du noyau.

PVERSC P demande une interaction à C.
CVERSP C demande une interaction à P.
PVERSNC P demande une interaction à NC (noyau).
CVERSNC C demande une interaction à NC.
.
.

IV.2.2.2 Les activités du processus noyau.

Pour la communication des processus P à travers les processus C, le noyau doit gérer deux types d'interaction :

- L'interaction d'un processus P vers un processus C, pour la demande d'exécution d'une fonction d'accès au canal.

- L'interaction d'un processus C vers un processus P, pour signaler la fin de la fonction d'accès.

Pour le traitement de ces interactions deux activités sont définies et ses points d'entrée correspondants sont :

PVERSC : ecmd(ptmsg) pour la demande d'exécution d'une fonction d'accès.

CVERSP : fcmd(ptmsg) pour la fin de l'opération d'accès.

Pour la mise à jour des contextes des processus P et C (pour la création et la destruction des processus) et la mise à jour de l'information de contrôle, nous avons défini deux autres activités :

PVERSNC : controlep(ptmsg) pour la modification des contextes et des informations de contrôle de P.

PVERSNC : controlec(ptmsg) pour la modification des contextes et des informations de contrôle de C.

A la fin du traitement d'une interaction, le prochain processus est activé par la procédure "schd".

Les points d'entrée du processus noyau sont ainsi programmés :

```
/*Les points d'entrée du noyau.                                    */
pents(ptmsg)

(..
 switch (msg(PENT))
   (              /*Interactions entre processus P et C            */
   case PVERSC  :  ecmd(ptmsg); schd(..): switchp (...); break;
   case CVERSP  :  fcmd(ptmsg); schd(..); switchp (...): break;
                  /*Interaction des P et C avec NC (noyau)         */
   case PVERSNC :  controlp(ptmsg); schd(..); switchp (...): break;
   case CVERSNC :  controlc(ptmsg); schd(..); switchp (...)· break;
   )
)
```

On peut remarquer que le processus noyau n'a pas d'activité séquentielle. Ce processus a le comportement d'un serveur classique, il répond toujours aux demandes d'interaction de ses clients, les processus P et C.

La programmation des ativités.

Le traitement d'une interaction consiste d'abord à vérifier de l'opération et ensuite, à mettre à jour les états des processus impliqués.

Dans la procédure de validation, nous vérifions l'existence du processus cible. En cas d'erreur, le processus demandeur est mis dans un état

d'erreur. Dans le cas contraire, on actualise les états des processus demandeur et du processus appelé. Pour la mise à jour des états et pour la réalisation des actions correspondantes, nous utiliserons une table de transitions (pour plus de détails sur cette table, voir l'anèxe II).

```
/*          Demande d'exécution d'une fonction de C.                    */
ecmd(ptmsg)

(..    /*Validation de l'interaction
                                                                         */
 if ( errecmd (ptmsg)) action ( majp (msg(IDPS),ERROP))
  else /*Mise à jour des processus
   (action ( majp (msg(IDPS),ECMD));                                     */
    action ( majc (msg(IDPDC),ECMD));
   )
)


/*          Fin d'exécution d'une fonction de C.                        */
fcmd(ptmsg)

(..   /*Validation de l'interaction
 if (errfcmd (ptmsg)) action (majc(IDPS),ERROP)                          */
  else/*Mise à jour des processus
   (action ( majc (IDPS),FCMD));                                         */
    action ( majp (IDPDP),FCMD));
   )
)
```

Les procédures :

```
errecmd(ptmsg) /*Fonction logique qui valide l'interaction ecmd.        */
(..."errecmd=FAUX si interaction invalide si non errecmd=VRAI"..);

majp(idp,eve)  /*Fonction qui à partir de l'arrivée de l'événement      */
               /*"eve" met à  jour l'état du processus "idp". Cette      */
(.......)      /*fonction donne comme résultat l'identificateur de       */
               /*l'action interne à exécuter.                            */

majc(idc,eve)  /*Fonction qui à partir de l'arrivée de l'événement      */
               /*"eve" met à  jour l'état du processus "idc". Cette      */
(.......)      /*fonction donne comme résultat l'identificateur de       */
               /*l'action interne à exécuter.

action(nact)   /*Procédure qui exécute l'action "nact"                  */
               /*Mémorisation de demandes, mise à jour des tables du*/
(..."......)   /*système, etc.                                           */
```

IV.2.2.3 L'algorithme pour le cadencement des processus.

La procédure "schd" se charge de l'activation des processus qui attendent d'être exécutés.

Pour la programmation de la politique de service défini dans III.2.2.2, nous avons utilisé deux listes d'attente, l' une pour les processus P : "qp", l'autre pour les processus C : "qc". Pour le choix du processus à activer, nous regardons d'abord la liste "qc" et seulement si elle est vide, nous regardons la liste "qp". Le processus à activer sera toujours le plus prioritaire.

```
/*Procédure qui détermine le prochain processus à exécuter          */
/*idpr...identificateur du processus.                               */
/*pent...point d'entrée de l'activité.                              */
/*pret...point de retour du processus.                              */
/*ptext..pointeur du texte du message.                              */
schd(idpr,pent,pret,ptxt)
..
(..
 if (non qcvide)
   choixc(idpr,pent,pret,ptext); /*Choix sur la liste des procs C  */
 else
   choixp(idpr,pent,pret,ptext); /*Choix sur la liste des procs P  */
)
```

Il faut remarquer que la politique de service n'est pas fixe : pour la changer, il suffit de modifier la procédure "schd" et éventuellement la structure de données des listes d'attente.

IV.2.3 La programmation des canaux et des primitives d'accès.

Le processus canal, comme il a été défini dans III.2.3, est un objet qui a comme tâche la gestion et le contrôle de la communication des processus P.

Un processus canal se caractérise par ses propriétés ou contraintes d'accès et par ses fonctions d'accès.

IV.2.3.1 La structure de données du processus canal.

Le format du message d'échange.

Pour la programmation des canaux (objets de communication de niveau plus haut que le mécanisme d'appel), on a introduit dans le message d'échange, certaines informations pour le contrôle de la communication :

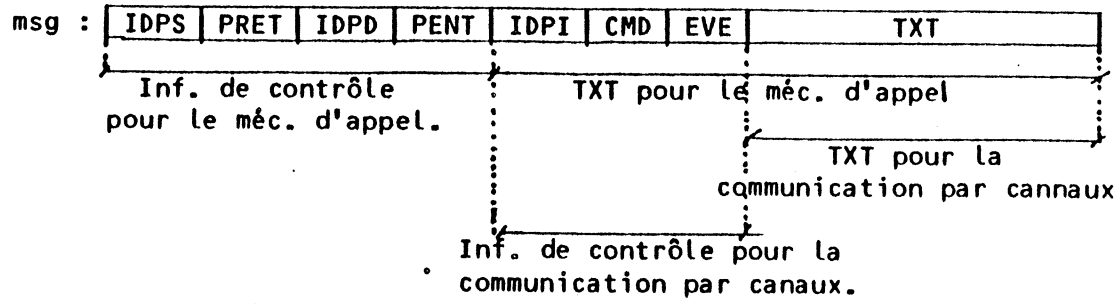
Adresse indirecte. Toutes les interactions entre les processus P et C sont contrôlées par le processus noyau. Par exemple, la demande d'exécution d'une fonction d'accès à un canal par un processus P doit être adressée d'abord au processus noyau. Le noyau mémorise la demande, et l'activation de la fonction du canal correspondant sera décidée par sa procédure "schd".

Les messages d'échange des processus P (ou C), ont toujours comme processus destinataire immédiat, le processus noyau et comme destinataire indirect, le processus C (ou P). Cette adresse indirecte (IDPI) est ajoutée dans le message.

Point d'entrée indirect. Pour l'exécution d'une activité de processus (P ou C), nous avons introduit le champ CMD comme point d'entrée correspondante.

Contrôle d'événements. Un processus qui demande une opération de communication sur un canal, peut être mise en attente pour l'une des raisons suivantes : l'opération est en train d'être réalisée par le canal ou l'opération est bloquée et le canal attand l'arrviée d'un événement pour son débloquage (par exemple dans le cas d'une lecture sur un canal vide, le processus lecteur sera débloqué lorsqu'un processus écrira sur le même canal).
Pour l'activation d'un processus bloqué, nous avons introduit le champ EVE.

Format du message d'échange au niveau système :



L'addition de ces nouvelles informations dans le format du message d'échange est nécessaire pour la programmation des canaux mais au niveau de l'interface d'implémentation elle n'a aucune influence. D'ailleurs, ces informations sont considérées par l'interface comme faisant partie du texte et aucun traitement n'y est effectué.

Les contraintes d'accès d'un canal.

La zone de mémoire locale du canal, est constituée tout d'abord des variables d'état, qui déterminent l'accès à ses fonctions, et (si la taille du canal a été déclarée différente de zéro) d'un espace pour la mémorisation des messages.

Les variables d'état du canal.

Les variables variables permettent le contrôle de l'accès aux fonctions du canal.

qvide	canal vide.
qplein	canal plein.
nents	nb. de liaisons d'entrées.
nsort	nb. de liaisons de sorties.

.

.

Espace de mémoire.

Dans cet espace nous mémoriserons les demandes d'opérations et les messages.

qapls table d'appels.
q table de messages.

IV.2.3.2 Les activités d'un canal.

Les fonctions d'accès fondamentales d'un canal sont : la lecture et l'écriture de messages, donc deux activités (avec leurs points d'entrée correspondants LECT et ECRT) sont associées à un processus canal

```
/*        Les points d'entrée d'un canal élémentaire.                  */
pents(ptmsg)

(..
 switch(ptmsg)
 (
  case LECT : dlect(ptmsg) ; break ; /*Demande de lecture.            */
  case ECRT : decrt(ptmsg) ; break ; /*Demande d'écriture.            */
 )
)
```

Comme les fonctions d'accès à un canal sont conditionnées par les contraintes d'accès, les états des variables sont testées avant l'exécution de la fonction. Si l'opération est possible, la fonction correspondante est exécutée et le processus demandeur pourra être activé par le noyau ; mais par contre, si l'opération n'est pas possible, la demande est mémorisée et le processus restera bloqué jusqu'à l'arrivée d'une autre demande qui change l'état qui le bloque.

```
/*                    Demande de lecture.                          */
dlect(ptmsg)

(..

   if (qvide)              /*Test de la condition de lecture.      */
    (dmdlec=VRAI;
     finop=NONLECT;        /*P sera bloqué.                        */
    )
   else
     if (dmdecr)           /*Test  si P' bloqué                    */
      (dmdecr=FAUX;
       finop=LECTECR;      /*Déblocage de P et de P'               */
      )
     else
      finop=LECT;
   execop(finop,ptmsg); /*Déblocage de P.                         */
)
```

```
/*                    Demande d'écriture.                          */
decr(ptmsg)

(..

  if (qplein)              /*Test de la condition d'écriture.      */
   (dmdecr=VRAI;
    finop=NONECRT;        /*P sera bloqué                         */
   )
  else
    if (dmdlect)           /*Test  si P' bloqué.                   */
     (dmdlect=FAUX
      finop=LECTECR;      /*Déblocage de P et P'.                 */
     )
    else
      finop=ECR;
  execop(finop,ptmsg); /*Déblocage de P.                         */
)
```

```
/*        Procédure d'exécution des fonctions du canal.          */

execop(op,ptmsg)

(...
  switch(op)
  (
   case LECT : /*Lecture et déblocage du processus.              */
           (lecture  (ptmsg);  ...  msg.cmd=flect;  ...);  break;
   case ECR  : /*Ecriture et déblocage du processus.             */
           (ecriture(ptmsg);  ...  msg.cmd=fecr;    ...);  break;
   case NONLECT : /*Blocage du processus en attente de lecture.  */
           (... msg.cmd=nlect; ...);                       break;
   case NONECR  : /*Blocage du processus en attente d'écriture. */
           (... msg.cmd=necr;  ...);                       break;
   case LECTECR : /*Lecture-écriture et déblocage des processus. */
           (copy  (ptmsg);        ...  msg.cmd=lececrt...); break;
  )
 /*Emission du message au processus au noyau.                    */
 SWITCHP (idn,SANSRET,CVERSP,ptxt);

)
```

Les événements.

Pour le contrôle des événements, un canal doit avoir dans sa zone de mémoire les listes suivantes :

LPi .... pour la mémorisation du nom du processus demandeur.
LEVEi .. pour la mémorisation de l'id. de l'événement attendu.
LPRCDi.. pour la mémorisation des procédures de traitement.
LMSKi... pour le contrôle du masquage des événements.

Le message de demande de traitement d'événement d'un processus doit contenir dans son texte, l'identificateur de l'événement (ideve), et si le demande est le traitement asynchrone d'un événement, le message doit contenir aussi l'identificateur de la procédure de traitement (idprc).

Pour le déblocage d'un processus, qui a demandé le traitement d'un événement, le canal teste avec tout changement de son état (associé à l'arrivée des messages) si l'événement attendu par le processus bloqué vient de se produire.

Les activités correspondantes du service de traitement d'événements sont :

Activité "attente" pour la demande d'attente d'un événement "eve" du processus "idp". Le canal mémorise la demande et laisse le processus dans un état d'attente de l'événement.

```
attente (ptmsg) /*Mémorisation de l'événement.                    */
(..
  LPi=msg.idp;
  LEVEi=msg.ideve;
)
```

Activité "active" pour la demande de traitement asynchrone de l'événement "eve" à travers l'activité "idprcdr" du processus "idp". Le canal mémorise la demande et et débloque le processus. Si l'événement arrive, le canal signalera (asynchronement) au processus l'arrivée de l'événement avec l'exécution de l'activité "idprcdr" correspondante.

```
active (ptmsg) /*Mémorisation d'une demande de traitement.        */
(..
  LPi=msg.idp;
  LEVEi=msg.ideve;
  LPCDRi=msg.idprcdr;
  LMASKi=NON;
)
```

Activité "dactive" pour l'annulation d'une demande de traitement asyncrhone d'événement. Le canal efface la demande et débloque le processus.

```
dactive (ptmsg) /*Effacement d'une activation.                    */
(..
  LPi=VIDE;
  LEVEi=VIDE;
  LPCDRi=VIDE;
  LMASKi=VIDE;
)
```

Activités pour le masquage et démasquage d'un traitement asynchrone. Le canal mettra à jour la variable LMASKi corrrespondante.

Pour le contrôle des événements, le canal teste avec la réception de tout message : a) s'il y a un processus bloqué à cause de sa demande de traitement d'événement et b) si l'événement attendu vient d'arriver. Si ces deux conditions sont satiasfaites, dans le cas d'une attente, le processus est débloqué et dans le cas d'une demande de traitement asynchrone,

l'activité "idprcdr" correspondante est exécutée.

IV.2.4 La programmation des primitives de communication.

Au niveau système, la programmation des processus noyau et canaux a consisté en l'écriture d'activités et en l'utilisation de la primitive SWITCHP pour les opérations de communication. Au niveau application, la programmation consiste dans l'écriture d'activites et l'utilisation d'un ensemble de primitives pour la communication par canaux. Cet ensemble de primitives est décrit dans l'ANNEXE I, et nous montrerons ici des exemples de leur implémentation.

Toutes les primitives de communication par canaux (créer/détruire, lier/délier, lecture/écriture et traitement d'événements) sont réduites à une primitive SWITCHP. L'adresse indirecte, en dépendant de la primitive de communication utilisée, sera l'adresse d'un processus système (pour les primitives de création/destruction et liaison/déliaison) ou l'adresse d'un processus canal (primitives lecture/écriture et traitement des événements).

Par exemple :

```
LIRECANAL (idc,ptmsg,to,ret)

(.."remplissage du msg avec tous les paramètres";
 .
 .
 msg.IDPI=idc;
 switchp (idp,AVECRET,idn,PVERSC,txt);
)
```

```
CEERCANAL (nomc,modc,typc,typl,lgc,pri,tvm,ret)

(.."remplissage du msg avec tous les paramètres";
 .
 .
 msg.IDPI=Ccr; /*Message adressé au canal du processus système CR  */
 switchp (idp,AVECRET,idn,PVERSC,txt);
)
```

IV.2.5 La programmation des processus système.

Un processus système est programmé comme un processus serveur. Les services (activités) sont accessibles à travers leur canal d'origine qui sera connu par tous les processus. Par exemple le processus gestionnaire de la création/destruction des canaux Pcr, sera accessible par son canal d'origine qu'on appelera Ccr.

```
/*                    Processus CR                              */

.              "Le mécanisme d'appel"
.

/*              Services-Activités offerts.                     */
creer (ptmsg)
      (...
        "vérifications";
        "activation"   ;
        .
        .
      )

detruir (ptmsg)
      (...
        "vérifications";
        "desactivation";
        .
        .
      )
```

IV.3 Conclusions.

A partir de cette première version nous pouvons faire quelques observations, d'abord sur l'implémentation du modèle canal et ensuite sur le système d'exploitation utilisé, UNIX.

IV.3.1 Commentaires sur la programmation du modèle.

De façon générale, le modèle canal doit être programmé en deux étapes :

- Au niveau del'interface avec la machine, le modèle de processus et le mécanisme de contrôle sont programmés à partir des primitives de contrôle de la machine cible.

- Au niveau de l'interface avec notre modèle de communication les processus noyau et les processus canaux sont programmés à partir du modèle de processus et du mécanisme d'appel.

Le niveau interface d'implémentation.

Pour l'implémentation du mécanisme de contrôle sur UNIX, nous avons utilisé le langage de haut niveau C et les primitives système pipe, read et write.

Les pipes nous ont permis d'associer facilement la notion de communication avec celle du contrôle d'exécution. Par contre, nous avons trouvé un problème dans l'utilisation d'une structure de contrôle d'un langage de haut niveau (l'appel de procédure) pour la programmation de notre mécanisme de contrôle.

En effet, un processus tel qu'il a été défini dans IV.2.1, après l'émission d'un message, doit être prêt, avec la réception d'un message, à prendre le contrôle de l'exécution sur n'importe quel point d'entrée (adresse) de ses activités. Eventuellement un de ces points d'entrée peut être l'adresse de l'instruction suivant l'émission (point d'entrée variable). Ce comportement ne peut pas être programmé directement avec un appel de procédure, parce qu'il rend toujours le contrôle à l'instruction suivante de l'appel. Pour la programmation du mécanisme d'appel, nous

avons fait des appels consécutifs.

Pour une programmation plus efficace du mécanisme d'appel, on devrait donc utiliser au moins pour la programmation du contrôle de l'exécution des structures plus élémentaires existant dans les langages assembleurs.

En général nous pouvons considér, que l'implémentation du mécanisme de contrôle dans des machines différentes ne doit pas être difficile et ne doit nécessiter que très peu de lignes de code. Il est souhaitable, par souci d'optimisation dans le temps de réponse, d'utiliser des langages et des primitives de contrôle qui soient les plus proches possibles de la machine cible.

Il est certain que l'optimisation du modèle (basé sur la notion de processus et de messages) sur une machine conventionelle sera toujours limitée. Il faudrait, pour améliorer les temps de réponses, disposer de machines dont l'architecture soit basée sur la notion de processus et de messages. Une telle machine, basée sur le langage OCCAM, est en cours de conception en Angleterre par la societé INMOS (WILS83a, WILS83b, CURR84, POUN84).

Le niveau interface modèle.

Nous avons constaté que la programmation du noyau a partir des outils de base (le modèle de processus et le mécanisme d'appel) et d'un langage de haut niveau est simple.

Le processus noyau est programmé comme un processus serveur par les processus P et C. Son code est complètement indepéndante de la machine cible. Grâce au fait que les activités du noyau ont été réduites au minimun (le cadencement et la communication locale), le nombre de lignes de code du noyau n'est pas importante.

Les processus canaux ont été programmés comme des serveurs des processus P, avec la particularité de pouvoir exécuter des actions sur les processus qui leur sont liés (exécution des activités avec l'arrivée asynchrone d'un événement).

Nous avons remarqué que la programmation d'un canal avec tous ses

possibilités, devient relativement complexe. Mais il faut dire que :
a) On n'est pas obligé de programmer tous les services définis pour un canal général. On peut définir des classes de canaux avec un sous-ensemble de services généraux, par exemple, des canaux qui n'ont pas besoin de liaisons, des canaux sans traitement des événements, etc.
b) Pour le contrôle communication-synchronisation, on doit toujours réaliser des fonctions de validation, de mémorisation des appels, de mise à jour des états, etc. Et dans ce sens, les canaux n'ajoutent pas de code supplémentaire. Mais par contre, il faut dire qu'à cause de la commutation fréquente des processus P et C, avec la sauvegarde nécessaire des contextes des processus, notre modèle aura des conséquence négatives sur le temps de réponse. Pour avoir une meilleure réponse, il faudrait disposer de machines conçues pour implémenter la notion de processus et de communication.

Les processus système. Pour rendre utilisable notre modèle, il faut au moins quelques processus système (les gestionnaires des processus, des canaux, des interruptions, des entrées-sorties, etc).

La programmation des gestionnaires des processus P et C est toujours simple et indépendante de la machine. Par contre pour les gestionnaires des interruptions et des entrées/sorties, on sera obligé d'utiliser des primitives élémentaires de la machine cible. La tâche de ces processus système est de convertir les signaux externes en demandes d'émission ou de réception de message pour le noyau. Ces processus n'ont pas été implémentés, mais nous pensons que leur réalisation doit être relativement simple.

Pour conclure les remarques sur la programmation de notre modèle, il faut noter, qu'il a été programmé sur un système d'exploitation existant, UNIX.

Les avantages d'une telle démarche ont été : l'utilisation de primitives de contrôle de haut niveau et l'emploi des utilitaires pour la mise au point du logiciel. Mais par contre, on a le grave inconvénient d'utiliser deux fois le contrôle sur les processus (le noyau UNIX fait aussi son scheduling). Pour avoir une version plus optimale, on devrait modifier le noyau UNIX.

IV.3.2 Commentaires sur UNIX.

UNIX avec ses primitives noyau, son langage C et ses utilitaires, nous a facilité l'implémentation du modèle.

Mais, par contre, au niveau de la communication-synchronisation, ses mécanismes ne sont pas les outils idéaux pour une programmation répartie temps réel. Les primitives UNIX souffrent de certains inconvénients :

Les pipes : absence de nom global, absence de la notion de message et le respect de la hiérarchie de création dans la communication.

Les signaux : non mémorisation des événements et son information réduite.

Mais il faut dire qu'UNIX a été conçu pour des applications interactives et non pour des applications réparties en temps réel. D'ailleurs les laboratoires Bell qui ont conçu UNIX, ont développé ensuite un système orienté vers les applications temps réel (MERT) et qui supporte UNIX comme un processus système.

Pour terminer, il faut remarquer que la version présentée ici, est une version simplifiée du modèle canal. Elle n'est pas complète et en plus elle n'est pas optimisée. Il faut remarquer aussi, qu'étant implémentée sur le système d'exploitation UNIX, elle ne pourra pas être utilisée pour des applications temps réel (UNIX ne garantit pas l'activation en "temps réel" d'un processus).

Notre intérêt principal a été l'évaluation de notre modèle par son implémentation sur un système et une machine donnés.

Dans un deuxième temps, pour mettre au point le modèle et l'étendre à la répartition, on utilisera un réseau local et une application type.

CONCLUSIONS

Nous considérons que le modèle canal utilisé comme base pour la programmation répartie temps réel, est un outil qui facilite, d'une part, l'expression du parallélisme et, d'autre part, la modularité et le transport de applications réparties.

La définition du canal comme un objet de type processus nous permettra d'avoir une programmation adaptée aux besoins spécifiques d'une application.

Plus concrétement, nous pouvons résumer les conclusions de notre travail dans chacun des niveaux de notre architecture.

Au niveau de la application, la définition d'un objet unique pour la programmation de la communication et de la synchronisation et d'un objet pour le traitement de l'information, permet une expression plus claire de la répartition et la rendre independante de l'implémentation grâce à la séparation de la communication du traitement. Par exemple, des changements sur les protocoles de communication utilisés n'auront pas d'effet sur la strucutre des applications et leur portabilité. En plus, avec la définition d'un seul objet pour l'expression des contraintes de communication, synchronisation et traitement des événements facilite la correction dans la construction de programmes.

Dans un environnement hétérogène de systèmes hôtes (par exemple un réseau local), notre modèle offre une interface de communication homogène pour les applications répaties.

Au niveau système, la définition d'un noyau minimal avec le cadencement et la communication locale comme tâches fondamentales, facilite la modularité du système. Avec le processus noyau et la répartition des processus système (gestion de mémoire, gestion de fichiers, etc), le modèle peut être programmé sur mesure, selon l'approche suivante : un site de faible capacité de mémoire aura son noyau et quelques processus système pour les traitements idispensables. Et pour les traitements moins importants, les processus locaux accéderont, de manière transparente, aux serveurs distants correspondants.

Au niveau de l'implémentation du modèle, la séparation des services de communication (avec la communication dans les processus canaux et la gestion de la communication dans le processus noyau) de l'implémentation (mécanisme d'appel) rend plus aisé le transport du modèle. Le seul code à reprogrammer pour l'implémentation du modèle sur une autre machine, est le mécanisme d'appel. Cet mécanisme est simple et composé seulement de deux primitives de contrôle élémentaire (primitives MULTIPLEX et SWITCHP). Le contrôle de l'exécution associée à l'émission et à la réception de messages a été choisi comme contrôle élémentaire de notre modèle.

La spécification des primitives d'accès aux canaux et l'implémentation présentées dans ce travail, constituent une première version du modèle. Des corrections ou des améliorations peuvent donc être proposées. Dans le futur, avec l'implémentation du modèle sur un réseau local et dans le cadre d'une application réelle, on pourra faire des évaluations de notre modèle. Actuellement, est en cours une implémentation du modèle sur un système iRMX-INTEL qui permettra l'implémentation d'une nouvelle version du langage LM pour la programmation de robots (Thèse M. Riveill à paraître). Ce travail nous a déjà apporté des expériences pour la définition de notre modèle. Des modifications dans les primitives d'accès ont été faites sur la version de départ. Enfin, le fait d'avoir développé deux versions du modèle, UNIX-VAX et iRMX-INTEL, nous a permis de faire une première évaluation de l'implémentation du modèle.

Il faut remarquer que la version présentée ici est en voie de devoloppement. Les canaux répartis n'ont pas encore été implémentés. Un deuxième travail de recherche sur la répartition est en cours. Il apportera sans aucun doute de nouvelles améliorations et extensions à ce modèle.

BIBLIOGRAPHIE

(ANCE 80) Anceau F.
Evolution des mocroprocessuers et son influence sur l'architecture de systèmes.
Real-time data handling and process contrôl.
North-Holland Publishing Company. Brussels 1980, p.227-233

(ANDR 82) Andrews G.R. and Schneider F.B.
Concepts and notations for concurrent programming.
Dept. Computer Science Raport TR 82-520.
Cornell University Ithaca, New York. Sep. 82.

(ABRA 82) Abramsky S. and Bornat R.
Pascal-m : un langage for distribuited systems.
Computer Systems Laboratory Queen Mary College. 1982.

(ADAM 83) Adamo J. M.
Pascal+CSP.
Université Claud Bernard. Villeurbanne France. 1983.

(BOCH 83) Bochman G. V. and Raynal M.
Structured Specification of Communicating Systems.
IEEE Transactions on Computers V.C-32 No.2. Feb. 83, p.120-133.

(BANI 80) Banino J.S., Caristan A., Guillemont M., Morisset G., Zimmermann H.
CHORUS : un architecture for distributed systems.
INRIA Research Report No. 42. Nov. 1983.

(BLAI 83) Blair G.S., Mariani J.A. and Shepherd W.D.
A practical extention to UNIX for Interprocess Communication.
Software Practice and Experience V.13 1983, p.45-48.

(BARN 80) Barnes J.G.P.
The developement of tasking primitives in high level languages.
Real-Time Data handling and process control. H. Meyer, Ed.
North-Holland Publishing Company. Brussels 1980, p.235-241

(BRIN 73a) Brinch Hansen.
Operating Systems Principles.
Pretince Hall. New Jersey 1973.

(BRIN 73b) Brinch Hansen.
Concurrent programming concepts.
ACM Computing Surveys 5.4 Dec 1973, p.223-245.

(BRIN 77) Brinch Hansen.
The arhitecture of concurrent programmes.
Pretince Hall. New Jersey 1970.

(BROW 82) Brownbrifge D.R., Marshall L.F., Rondell B.
The Newcastle connection.
Software-Practice and Experience Vol. 12. 1982, p.1147-1162.

(BUCK 83) Buckley N.G. and Silberschatz A.
An effective implementation for the generalized input-ouput construct of CSP.
ACM Transactions on programming languages and systems.
V. 5 No. 2, April 83, p.223-235.

(CONW 63a) Conway M. E.
Design of a separable transition diagram compiler.
Communications ACM 6,7. July 1963, p.396-408.

(CONW 63b) Conway M. E.
A multiprocesseur system design.
AFIPS Conf. Proc. Vol. 24. 1963, p.139-146.

(CURR 84) Curry B.J.
Language-based architecture eases system design-III.
Computer Design January 1984, p. 127-136;

(DIGI 79) Digital Equipement Corporation.
RSX-11M Reference Manual.
Digital Equipement Corporation. 1979.

(DIJK 68a) Dijksta E. W.
The structure of the 'THE' multiprogramming system.
Comm. ACM 11.5. May 1968, p.341-346.

(DIJK 68b) Dijkstra E. W.
Cooperating sequential Processes.
Programming Langages. F. Genuys (Ed). Academyc Press
New York. 1968.

(DIX 83) Dix T.I.
Exceptions and interrupts in CSP.
Science of computer programming. Mars 1983, p.189-204.

(FAY 84) Fay D.Q.M.
Experiences unsing INMOS PROTO-OCCAM (TM.
SIGPLAN Notices, V19 No. 9, September 1984

(FOST 81) Foster C. C.
Real time programming neglected topics.
Addison Wisley Pub. Co. 1981.

(GERT 75) Gertey J. and Sedlak J.
Software for Process control - A survey.
Automatica Vol. 11, Pergamon Press 1975, p.613-625.

(GUIL 82) Guillemont M.
Integration du système réparti CHORUS dans le langage de haut niveau Pascal.
Thèse de Docteur Ingénieur. Université Scientifique et Médicale de Grenoble. Sep. 1982.

(HEWI 77) Hewit C. B. H.
Laws for communicating parallel processes.
Proc. of the IFIP congres. North Holland 1977.

(HOAR 74) Hoare C. A. R.
Moniteurs : un operating system structuring concept.
Comm. ACM 17.10. Oct. 1974, p.549-557.

(HOAR 78) Hoare C. A. R.
Communicating sequential process.
Comm. ACM 21.8. Aug. 1978, p.666-677.

(INTE) INTEL.
iRMX 86 Nucleus reference manual.
Manual Number 9803122-03 INTEL.

(JOY 81) Joy W. and Fabry P.
An architecture for interprocess communication in UNIX.
Computer Systems Research Group.
Computer Science Division Berkely. CSRG TR/3 Draft of june 1981.

(KEEF 82) Keeffe D., Tomlison G., Wand I. and Wellings A.
PULSE project 1982.
Dep. of Computer Science Report No. 58. University of York 1982.

(LICK 77) Lycklama H., Bayer D.L.
The MERT operating system.
The Bell Systeme Technical Journal July-August 1978, p.2049-2086.

(LUDE 81) Luderer G.W.R., Che H., Haggerty J.P., Kirlis P.A. and
Marshall W.T.
A distributed UNIX system based on a virtual circuit switch.
ACM..1981, p.160-168.

(MAY 83) May D. & Taylor R.
"OCCAM"
SIGPLAN Notices V. 18 No. 4,1983

(MUNT 83) Muntean T.
Introduction à OCCAM langage parallèle issu de CSP pour la
programmation des systèmes de transinateurs.
Rapport de recherche No. 430. Laboratoire de Génie
Informatique IMAG, Dec. 1983.

(MUNT 84) Muntean T. et Riveill M.
Coincidence temporelle pour processus communicants un algorithme

Repport de recherche No. 448. Laboratoire de Génie Informatique IMAG, Juillet 1984.

(PANZ 82) Panzieri and Shrivastava S.K.
Reliable Remote Calls for Distributed UNIX :
An implmentation study.
Technical Raport No. 177. University of Newcastle 1982.

(POU 84) Pountain D.
The transputer and its special language, OCCAM.
BYTE August 1984, p.361-366.

(PYLE 79) Pyle I.C. and Wand I.C.
Real time programming language for industrial and scientific process control.
York Computer Science Rep. No. 21. England 1982.

(PYLE 80) Pyle I.C.
Review of standards in software for real-time systems.
Real-Time data handling and process control. H. Meyer Ed.
Nort-Holland Publishing Company. Brussels 1980.

(RASH 80) Rashid R.F.
An interprocess communication facility for UNIX.
Dep. Computer Science. Report CMU-CS-80-124.
Carneige Mellon University 1980.

(RITC 78) RITCHIE D.M. and Thompson K.
The UNIX Time Sharing System.
The Bell System Technical Journal. July-Aug 1979, p.1905-1930.

(RIVE 83) Riveill M.
Synchronisation d'horloges physiques.
Rapport de DEA ENSIMAG Grenoble, Sep. 1983.

(ROWE 82) Rowe L.A., Birman K.P.
A local Network based on the UNIX operating system.
IEEE Trans. on Soft. Engr. Vol SE-8 No.2 March 1982, 137-145.

(RUEB 78) Rueb W. and Schort G.
Nested Interrupts and controlled preemptions to satisfy priority realtime schedules.
"Institut für Informatik" of the Technical University Munich.

(SCHW 78) Schwarts J.
Distributed synchronisation of communicating sequential process.
DIA Reserch Report No. 51. University of Edinburgh 1978.

(SILB 79) Silberschatz A.
Communication and synchronisation in distributed systems.
IEEE Transactions of Software Engr. V.SE-5 No.6 Nov. 1979, p.542

(SILB 81) Silberschatz A.
Port directed communication.
The computer Journal V.24 No. 1, 1981, 78-82.

(WELL 81) Wellings A.J., Wand I.C., Tomlinson G.M.
Distributed UNIX Project 1981.
York Computer Science Report No. 47 University of York 1981.

(WIRT 77) Wirth N.
Towards a discipline of real-time programming.
Communication of ACM V.20 No. 8 1977,p 577-583.

(WOOD 80) Wood G.G.
Review of common practice and accepted hardware standards in process control.
Real-Time data handling process control. H. Meyer Ed.
North-Holland Publishing Company. Brussels 1980.

(WILS 83a) Wilson P.
OCCAM architecture eases system disign-part I.
Computer Design Nov. 1983, p.107-115.

(WILS 83b) Wilson P.
Language-based architecture eases system design-II.
Computer Design Dec. 1983, p.109-120.

(YOUN 82) Young S. J.
Real Time Langages design and development.
Ed. Ellis Horwood Limited. England 1982.

ANNEXE I : LES PRIMITIVES D'ACCES AUX CANAUX.

1. La création d'un canal.

2. La destruction d'un canal.

3. La liaison à un canal.

4. La déliaison d'un canal.

5. La lecture d'un canal.

6. L'écriture d'un canal.

7. Le traitement des événements.

   7.1 L'attente des événements.

   7.2 Le traitement asynchrone des événements.

      7.2.1 L'activation d'une procédure pour le traitement asynchrone d'un événement.

      7.2.2 La desactivation de la procédure de traitement asynchrone.

      7.2.3 Le masquage et demasquage des traitements asynchrones.

8. La modification des paramètres d'un canal.

ANNEXE 1 : LES PRIMITIVES D'ACCES AUX CANAUX.

La communication et la synchronisation entre les processus séquentiels (processus type P) d'un programme parallèle sont réalisées à travers l'objet canal (processus type C).

Les services de communication, accessibles par un ensemble de primitives, sont :

-Création et déstruction dynamique,
-Liaison et déliaison dynamique,
-Lecture et écriture,
-Traitement asynchrone des événements.

1. La création d'un canal.

Un processus P peut créer un canal en utilisant cette primitive.

Un canal se crée à partir d'un modèle déjà défini. Le système associe au canal un identificateur unique qui représentera le seul moyen d'accès. Un canal peut être déclaré public ou privé. Un canal public est accessible à tous les processus P connaissant son nom. L'accès à un canal privé est contrôlé par le processus propiétaire.

Les paramètres pour la création d'un canal sont :

-La priorité du canal,
-Le type du canal (publique ou privé),
-La longueur du canal (nb. maximum de messages dans le canal),
-Le temps de vie des message dans le canal,
-Le type de liaison.

Primitive :

CREERCANAL (NOMC,MODC,TYPEC,TYPEL,LGC,PRI,TVM,RET)

paramètres d'entrée :

-NOMC.....nom du canal.
-MODC.....modèle du canal.
-TYPEC.....type du canal : public ou privé.
-TYPEL.....type de liaison :
point-à-point (1-1).
serveur (n-1).
client-multiserveur (1-m).
multiclient-multiserveur (n-m).
diffusion.
-LGC.....longueur du canal.
-PRI.....priorité du canal.
-TVM.....temps de vie du messages.

paramètres de sortie :

-RET.....code de retour :
+0 identificateur du canal créé.
-1 modèle de canal inexistant.

2. La destruction d'un canal.

L'exécution avec succès de cette primitive implique la purge du canal et la fermeture de toutes les liaisons. Le canal signalera cet événement à tous les processus P qui l'ont demandé.

Primitive :

DETRUIRCANAL (IDC,RET)

paramètre d'entrée :

-IDC.....identificateur du canal.

paramètre de sortie :

-RET.....code de retour :
0 O.K.
-1 canal inexistant.

3. La liaison à un canal.

Si un processus veut accéder à un canal (pour une lecture ou une écriture) il doit d'abord se lier au canal. Si le canal est public, la liaison se fait à travers son nom ; par contre, si le canal est privé, elle se faira par son identificateur donné au moment de sa création. Le canal signalera l'événement "processus lié au canal" à tous les processus que l'ont demandé.

Primitive :

LIERCANAL (NOMC,IDC,MODL,RET)

paramètres d'entrée :

-NOMC.....nom du canal s'il est public.
-IDC.....identificateur du canal s'il est privé.
-MODL.....mode de liaison :
E liaison à l'entrée.
S liaison à la sortie.

paramètres de sortie :

-RET.....code de retour :
0 O.K.
-1 canal inexistant.
-2 liaison invalide.

4. La déliaison d'un canal.

Si un processus ne veut plus communiquer, il se délie du canal en utilisant cette primitive. L'événement "déliaison d'un processus" est signalé à tous les processus que l'ont demandé.

Primitive :

DELIERCANAL (IDC,RET)

paramètre d'entrée :

-IDC.....identificateur du canal.

paramètre de sortie :

-RET.....code de retour :

0 O.K.
-1 canal inexistant.
-2 déliaison invalide.

5. La lecture d'un message du canal.

Tout processus lié à la sortie d'un canal peut lire ses messages. A l'exception des canaux du type diffusion, la lecture implique l'effacement du message à l'interieur du canal. Pour les canaux du type diffusion,le message restera dans le canal, et il sera effacé si tous les processus liés à la sortie ont fait une lecture ou si le temps de vie des messages du canal est écoulé.

La lecture d'un message est temporisée avec le paramètre TO. En fonction de sa valeur, on a trois types de lecture :
1) Une lecture bloquante avec le temporisateur égal à 00. Si le canal est vide le processus restera bloqué jusqu'à l'arrivée d'un message ;
2) Une demande de lecture non bloquante avec le temporisateur égal à 0. Si le canal est vide le processus sera débloqué avec le code de reotur "canal vide".

3) Une lecture temporisé avec une valeur du temporisateur entre 0 et 00. Si le canal est vide, le processus attendra l'arrivée d'un message au canal ou l'écoulement de son temporisateur.

Primitive :

LECTURECANAL (IDC,PTMSG,TO,RET)

paramètres d'entrée :

-IDC.....identificateur du canal.
-PTMSG.....adresse pour la réception du message.
-TO.....temporisateur.

paramètre de sortie :

-RET.....code derretour :

1 lecture du message.
0 canal vide (écoulement du temporisateur).
-1 canal inexistant.
-2 erreur dans la communication.

6. L'écriture d'un mesage du canal.

Un processus P lié à l'entrée d'un canal peut écrire des messages à l'aide de cette primitive. S'il y a de la place, le message est enregistré dans le canal. Dans le cas contraire, le processus attendra la libération d'une place ou l'écoulement de son temporisateur. En fonction de la valeur du temporisateur TO, on aura trois types d'écriture :
1) Ecriture bloquante, avec TO égal à 00. Si le canal est plein, le processus attendra la fin d'une lecture. ;
2) Demande d'écriture non bloquante avec TO égal à 0. Si le canal est plein le processus sera débloqué avec le code de retour : "canal plein".
3) Ecriture temporisée avec 0 TO 00. Si le canal est plein, le processus attendra la libération d'une place ou l'écoulemnt de son temporisateur.

Primitive :

ECRITURECANAL (IDC,PTMSG,TO,RET)

paramètres d'entrée :

-IDC.....identificateur du canal.
-PTMSG.....adresse du message à envoyer.
-TO.....temporisateur.

paramètre de sortie :

-RET.....code de retour :
1 écriture du messsage.
0 canal plein (écoulement de TO).
-1 canal inexistant.
-2 erreur dans la communication.

7. Le traitement des événements.

Un processus P peut demander le traitemement des événements associés aux canaux auxquels il est lié. Un processus peut 1) se mettre en attente sur un ou plusieurs événements et 2) demander au canal la signalisation et le traitement asynchrone des événements.

Les événements associés à un canal sont :

-L'arrivée d'un message.
-La sortie d'un message.
-La liaison d'un processu au canal.
-La déliaison d'un processus du canal.
-Canal vide.
-Canal plein.
-Un erreur dans la communication.
-L'avortement du canal.
-L'avortement d'un processus lié au canal.

7.1 L'attente des événements.

Un processus qui est lié à un canal peut se mettre en attente sur un de ses événements. Si l'événement s'est déjà produit, le retour est immédiat. Dans le cas contraire , le processus attend jusqu'à l'écoulement de son temporisateur TO. Si TO égal à 0, cette primitive sera équivalente à un test sur l'événement, si TO est égal 00, le processus restera bloqué jusqu'à l'arrivée de l'événement.

Primitive :

ATTENTEVE (IDC,IDE,TO,RET)

paramètres d'entrée :

-IDC.....identificateur du canal.
-IDE.....identificateur de l'événement.
-TO.....temporisateur.

paramètre de sortie :

-RET.....code de retour :
1 arrivée de l'événement.
0 écoulement du temporisateur.
-1 canal inexistant.

Un processus lié à un canal peut aussi se mettre en attente d'un événement sur un ensemble de canaux. Le principe d'attente est le même, avec la différence que, dans une attente multiple, au lieu d'avoir un seul canal comme paramètre d'entrée, on aura un vecteur de canaux. Le retour se fera par l'arrivée du premier événement. S'ils sont plusieurs, on aura dans les paramètres de sortie, le sous-ensemble de canaux où les événement se sont produits ; dans le code de retour l'identificateur du canal qui a été le "premier" à le recevoir.

Primitive :

-ATTENTEVES (VEC,TO,RET)

paramètres de'entrée :

-VEC(IDC,IDE)
Vecteur de canaux.
IDC...identificateur du canal.
IDE...identificateur de l'événement.
-TO.....temporisateur.

paramètres de sortie:

-VEC(IDC,B)
Vecteur de canaux.
IDC...identificateur du canal.
B.....boolean (vrai si IDC a reçu l'événement
faux si IDC n'a pas reçu l'événement).

-RET.....code de retour :
+0 identificateur du canal qui a été le premier à recevoir l'événement.
0 écoulement du temporisateur.
-1 erreur dans les paramètres.
-2 erreur dans la communciation.

7.2 Le traitement asynchrone des événements.

Un processus peut demander au canal auquel il est lié, la signalisation et le traitement d'un de ses événements. Pour le traitement le processus associera à l'arrivée de l'événement une procédure pour son tratement. Une fois la demande aceptée par le canal et si le processus est en train de s'exécuter et l'événement arrive, le processus est interrompu et la procédure correspondante est exécutée. A la fin de l'exécution de la procédure, le processus continuera son exécution normale.

Un processus qui a demandé le traitement asynchrone d'un événement peut masquer et démasquer ce traitement avec les primitives MASK et DMASK.

7.2.1 Activation d'une procédure pour le traitement asynchrone d'un événement.

Un processus associera la procédure pour le traitement d'un événement à la primitive suivante :

Primitive :

ACTIVETRT (IDC,IDE,PTMSG,ADRSPRCD,RET)

paramètres d'entrée :

-IDC.....identificateur du canal.
-IDE.....identificateur de l'événement.
-PTMSG....adresse pour la réception du message associé à l'événement.
-ADRSPRCD.adresse de la procédure de traitement.

paramètre de sortie :

-RET.....code de retour :
+0 identificateur du traitement (IDT).
-1 canal inexistant.
-2 erreur dans les paramètres.

7.2.2 Desactivation de la procédure de traitement aynchrone.

Pour la désactivation d'une procédure de traitement d'un événement le processus utilisera la primitve suivante :

Primirtive :

DACTIVETRT (IDT,RET)

paramètre d'entrée :

-IDT.....identificateur du traitement.

paramètre de sortie :

-RET.....code de retour :

0 O.K.
-1 identificateur inexistant.

7.2.3 Le masquage et demasquage du traitement asynchrone.

Un processus qui a demande le traitement asynchrone d'un événement, peut faire que l'on ne puisse pas interompre une séquence de code avec les primitives de masquage et demasquage.

Un processus peut masquer le traitement d'un événement avec la primitive :

Primitive :

MASKE (IDT,RET)

paramètre d'entrée :

-IDT.....identificateur de traitement.

paramètre de sortie :

-RET.....code de retour :
0 O.K.
-1 identificateur inexistant.

Pour le démasquage du traitement, le processus utilisera la primitive suivante :

Primitive :

DMASKE (IDT,RET)

paramètre d'entrée :

-IDT.....identificateur de traitement.

paramètre de sortie :

-RET.....code de retour :
0 O.K.
-1 identificateur inexistant.

8. La modification des paramètres d'un canal.

Le processus propiétaire du canal peut demander la modification des paramètres du canal avec, la primitive suivante :

Primitive :

MODC (IDC,PRMT,VAL,RET)

paramètres d'entrée :

-IDC.....identificateur du canal.
-PRMT.....paramètre à modifier.
-VAL.....nouvelle valeur du paramètre.

paramètre de sortie :

-RET.....code de retour :
0 O.K.
-1 canal inexistant.
-2 paramètres invalides.
-3 valeur invalide.

7.2 Le traitement asynchrone des événements.

Un processus peut demander au canal auquel il est lié, la signalisation et le traitement d'un de ses événements. Pour le traitement le processus associera à l'arrivée de l'événement une procédure pour son tratement. Une fois la demande aceptée par le canal et si le processus est en train de s'exécuter et l'événement arrive, le processus est interrompu et la procédure correspondante est exécutée. A la fin de l'exécution de la procédure, le processus continuera son exécution normale.

Un processus qui a demandé le traitement asynchrone d'un événement peut masquer et démasquer ce traitement avec les primitives MASK et DMASK.

7.2.1 Activation d'une procédure pour le traitement asynchrone d'un événement.

Un processus associera la procédure pour le traitement d'un événement à la primitive suivante :

Primitive :

ACTIVETRT (IDC,IDE,PTMSG,ADRSPRCD,RET)

paramètres d'entrée :

-IDC.....identificateur du canal.
-IDE.....identificateur de l'événement.
-PTMSG....adresse pour la réception du message associé à l'événement.
-ADRSPRCD.adresse de la procédure de traitement.

paramètre de sortie :

-RET.....code de retour :
+0 identificateur du traitement (IDT).
-1 canal inexistant.
-2 erreur dans les paramètres.

7.2.2 Desactivation de la procédure de traitement aynchrone.

Pour la désactivation d'une procédure de traitement d'un événement le processus utilisera la primitve suivante :

Primirtive :

DACTIVETRT (IDT,RET)

paramètre d'entrée :

-IDT.....identificateur du traitement.

paramètre de sortie :

-RET.....code de retour :

0 O.K.
-1 identificateur inexistant.

7.2.3 Le masquage et demasquage du traitement asynchrone.

Un processus qui a demande le traitement asynchrone d'un événement, peut faire que l'on ne puisse pas interompre une séquence de code avec les primitives de masquage et demasquage.

Un processus peut masquer le traitement d'un événement avec la primitive :

Primitive :

MASKE (IDT,RET)

paramètre d'entrée :

-IDT.....identificateur de traitement.

paramètre de sortie :

-RET.....code de retour :
0 O.K.
-1 identificateur inexistant.

Pour le démasquage du traitement, le processus utilisera la primitive suivante :

Primitive :

DMASKE (IDT,RET)

paramètre d'entrée :

-IDT.....identificateur de traitement.

paramètre de sortie :

-RET.....code de retour :
0 O.K.
-1 identificateur inexistant.

8. La modification des paramètres d'un canal.

Le processus propriétaire du canal peut demander la modification des paramètres du canal avec, la primitive suivante :

Primitive :

MODC (IDC,PRMT,VAL,RET)

paramètres d'entrée :

-IDC.....identificateur du canal.
-PRMT.....paramètre à modifier.
-VAL.....nouvelle valeur du paramètre.

paramètre de sortie :

-RET.....code de retour :

0 O.K.

-1 canal inexistant.

-2 paramètres invalides.

-3 valeur invalide.

ANNEXE II : LE MECANISME D'APPEL.

1. La structure d'un processus.

2. La primitive MULTIPLEX.

3. La primitive SWITCHP.

4. Le format du message d'échange.

5. Considérations sur l'implémentation du mécanisme.

ANNEXE 2 : LE MECANISME D'APPEL.

1. La structure d'un processus.

Un processus est défini comme un ensemble d'activités qui partagent une zone de mémoire commune. Chaque activité est associée à un point d'entrée. Une activité est exécutée si un message est reçu sur son point d'entrée. Un processus communique avec un autre en envoyant un message sur un de ses points d'entrée. Avec l'envoi d'un message le processus passe le contrôle d'exécution au processus destinataire sur un de ses points d'entrée. Un processus reprend le contrôle d'execution avec la réception d'un message sur un de ses points d'entrée.

Un processus est composé de deux parties : le corps du processus (mémoire commune et ensemble d'activités) et le mécanisme d'appel pour le contrôle de l'exécution.

Le corps d'un processus contient l'ensemble des activités programmées par l'utilisateur. Une de ces activités peut être Pseq. A l'initialisation du processus, l'activité Pseq est exécutée si elle a été déclarée. Après son initialisation, le processus sera toujours prêt à exécuter ses activités avec la réception des messages adressés à ses points d'entrée correspondants.

Le mécanisme d'appel est composé de deux primitives :

La primitive MULTIPLEX qui permet en même temps la réception d'un message et le reprise du contrôle d'exécution sur un des points d'entrée du processus.

La primitive SWITCHP qui permet en même temps l'envoi d'un message et le passage du contrôle d'exécution à un autre processus sur un point d'entrée.

Il faut remarquer que la primitive MULTIPLEX est transparent au programmeur (par définition un processus est toujours prêt à exécuter une de ses activités). La seule primtive disponible au programmeur pour la commu-

nication entre processus est la primitive SWITCHP.

Tout processus a la structur indiquée ci-dessous.

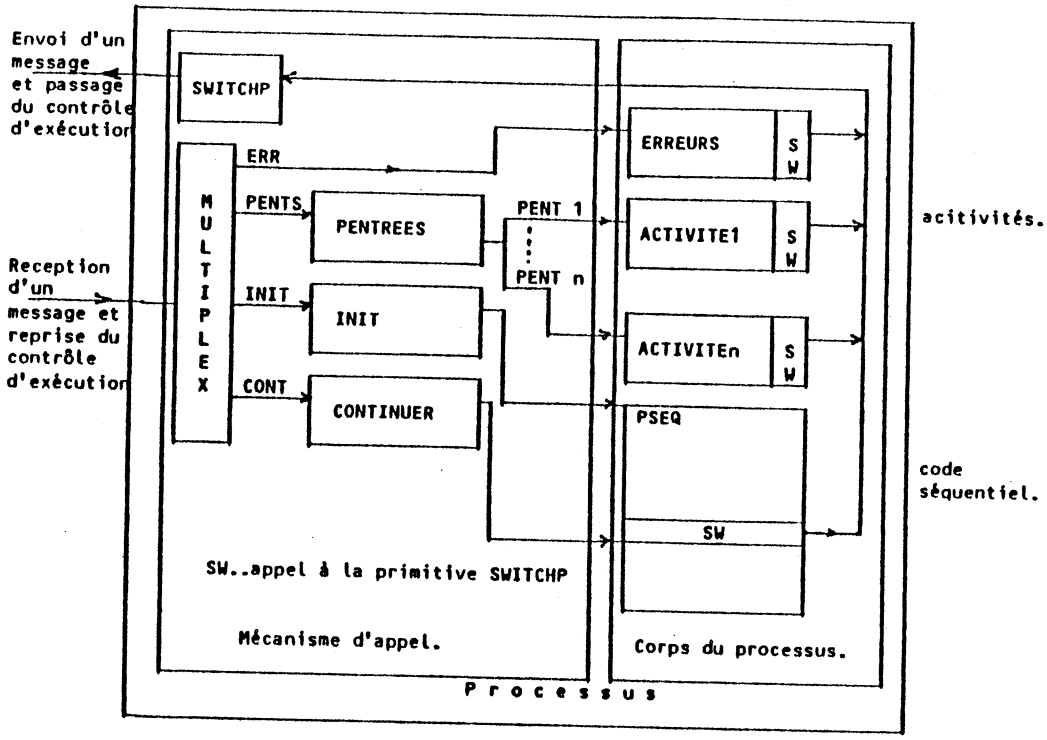


Fig. AII.1 La structure d'un processus.

2. La primitive MULTIPLEX.

Primitive qui permet en même temps, la réception d'un message et la repise du contrôle d'exécution sur un de ses quatre points d'entrée de contrôle d'un processus :

-CONT L'activité associée à ce point d'ntrée permet la continuation d'une exécution sur le dernier SWITCHP.

-INIT L'activité de ce point d'entrée intialise processus et l'exécution éventuelle de l'activité séquentielle Pseq

-ERR Ce point d'entrée déclenche l'exécution d'une activité

(programmée par l'utilisateur) pour le contrôle des erreurs dans la communication.

-PENTS Ce point d'entrée réalise le multiplexage sur les points d'entrée prédéfinis par le programmeur.

Primitive :

MULTIPLEX (ptmsg)

paramètres de sortie :

ptmsg.....adresse où se trouve le message arrivé.

Avec cette primitive un processus est toujours en attente de réception d'un message. Lors de la réception du message, cette primitive exécute l'activité correspondante au point de contrôle (CONT, INIT, ERR, PENTS) signalé dans le message.

Cette primitive est transparente au programmeur, elle est automatiquement exécutée après toute émission de message.

3. La primitive SWITCHP.

Cette primitive permet à un processus de passer le contrôle d'exécution à un autre avec l'envoi d'un message.

La primitive SWITCHP peut être exécutée sur deux modes :

Mode CONT, si on attend après l'exécution du SWICTHP un message pour pouvoir continuer l'exécution.

Mode SANSRET, si on n'attend pas de message de continuation et qu'on est prêt à exécuter une nouvelle activité.

Il faut remarquer qu'avec le mode CONT, on est aussi toujours prêt à reprendre le contrôle d'exécution sur un des autres points d'entrée, même si le message de continuation attendu n'est pas encore arrivé.

Il faut préciser que le contrôle d'exécution sur une continuation est fait sur une adresse variable (point d'entrée variable), tandis que tous les autres contrôles sont faits sur des points d'entrée fixes.

Primitive :

SWITCHP (pret,idpd,pent,ptxt)

paramètres d'entrée :

pret.....point de retour du processus appelant :
CONT.....si on attend un message de continuation.
SANSRET..si on n'attend pas de message de continuation
idpd.....identificateur du processus destinataire.
pent.....point d'entrée de l'activité du processus destinataire.
ptext....adresse du message à envoyer.

Cette primitive est la seule accessible à l'utilisateur et représente le seul moyen d'interaction entre processus. Avec ses deux modes d'exécution, l'utilisateur peut continuer son exécution (après l'arrivée du message de continuation correspondant) ou terminer la séquence d'une exécution. Dans les deux cas, après l'exécution de cette primtive, le processus sera toujours prêt a exécuter une des activités correspondante à ses points d'entrée.

4. Le format du message d'échange.

Le contrôle d'exécution est basé sur l'échange de messages des processus. L'exécution de la primitive SWITCHP entraîne l'envoi d'un message. La primitive MULTIPLEX est associée à la réception de ce message.

Le format du message d'échange est le suivant :

MSG(0)... identificateur du processus source.
MSG(1)... point de retour du processus source (CONT ou SANSRET).
MSG(2)... identificateur du processus destinataire.
MSG(3)... point d'entrée du processus destinataire.
MSG(4)... taille du texte (n).
MSG(5..n) texte.

Les informations du 0 au 3 sont indispensables pour le mécanisme d'appel. La taille du texte et la nature de son contenu dépendront des processus communicants.

5. Considérations sur l'implémentation du mécanisme.

Le mécanisme de communication défini auparavant est indépendant de l'implémentation. La programmation des primitves SWITCHP et MULTIPLEX dépendront des primitives de la machine et du système d'exploitation à utiliser.

Une implémentation peut se réealiser à différents niveaux : a) on peut utiliser une machine nue avec son assembleur et ses primitives élémentaires de contrôle (système d'interruptions, événements, mémoire commune, etc) ou b) une machine avec son système d'exploitation, un langage de haut niveau et des primitives système pour le contrôle d'execution (sémaphores, pipes, moniteurs, etc). Il faut remarquer que quelque soit le niveau, la programmation des primitives nécessite très peu de lignes de code.

Il faut remarquer aussi que le transfert "physique" d'un message d'un processus à un autre, n'est pas précisé. On peut envoyer tout le message (paramètres de contrôle et texte) puis passer le contrôle de l'exécution, ou envoyer seulement les paramètres de contrôle et un pointeur sur l'adresse du texte du message. Choisir l'une ou l'autre de ces possibilités dépendra des primitives élémentaires utilisées pour l'implémentation du mécanisme d'appel.

ANNEXE III : LE NOYAU DE COMMUNICATION.

1. Les objets du système.

2. La primitive élémentaire de communication.

3. Les événements du système.

    3.1 Les événements internes (l'appel au noyau).

    3.2 Les événements externes (les interruptions externes).

        3.2.1 Le traitement des interruptions externes.

4. Les états des processus.

    4.1 Les états des processus P.

    4.2 Les états des processs C.

5. Le scheduler.

6. La structure du processus noyau.

3. LE NOYAU DE COMMUNICATION.

1. Les objets du système.

Le noyau est le processus qui a comme tâche la gestion et le contrôle des deux autres types d'objets du modèle : les processus P et les processus C.

Au niveau de l'application, les processus P communiquent entre eux à travers les processus C, en utilisant les primitives de communication (création-destruction, liaison-déliaison, lecture-écriture) définies dans l'annexe I.

Au niveau du système, les processus P et C communiquent entre eux à travers le processus noyau NC.

Chaque fois que le noyau reçoit un message (appel), il réalise les trois fonctions suivantes :

-La mise à jour des états des processus communicants.

-La sélection du prochain processus, P ou C, à exécuter.

-Le passage du contrôle au processus sélectioné.

2. La primitive élémentaire de communication.

Pour la programmation des interactions entre les trois types de processus (P, C et NC), nous avons utilisé le mécanisme d'appel défini dans l'annexe II. Ce mécanisme permet aux processus, de prendre et de passer le contrôle d'exécution. Ce passage du contrôle est associé à l'envoi et la réception d'un message.

Au niveau de l'implémentation, nous avons utilisé pour toutes les interactions une seule primitive : SWITCHP. Cette primitive permet le passa-

ge du contrôle d'exécution avec l'envoi d'un message. Le processus source doit indiquer son nom, le point de retour attendu, le nom du processus destinataire et le point d'entrée de l'activité à exécuter.

3. Les événements du système.

Le contrôle d'exécution de ces objets se fait à travers deux types d'événements :

- Les événements internes associés aux demandes d'interaction des processus P et C au noyau.

- Les événements externes associés aux interruptions externes.

3.1 Les événements internes (l'appel au noyau).

L'exécution de la primitive SWITCHP, par P ou C vers NC génére un événement interne. Avec l'arrivée de cet événement, le noyau change l'états des processus correspondants et ensuite sélectione le prochain processus à exécuter (scheduler).

L'exécution de cette primitive par un processus P est associée à une des primitives de communication (créer/détruire, lier/délier, lire/écrire). Le processus P passe le contrôle au noyau et attend jusqu'à ce que la primitive de communication sur le canal soit terminée.

L'exécution de cette primitive par le canal, permet de signaler au noyau la fin du traitement d'une de ses primitives. Une fois passé le contrôle au noyau, le canal est prêt à exécuter une nouvelle primitive de communication.

L'exécution de cette primitive par le noyau vers un canal, déclenche l'initialisation d'une de ses primitives de communication demandée par le processus P avec le point d'entrée correspondant.

Une fois réalisée la primitive de communication, l'exécution de cette primitive par le noyau vers un processus P, permet à P de continuer son exécution sur son point de retour.

Dans la figure AII.3.1, on montre l'exécution d'une primitive de communication.

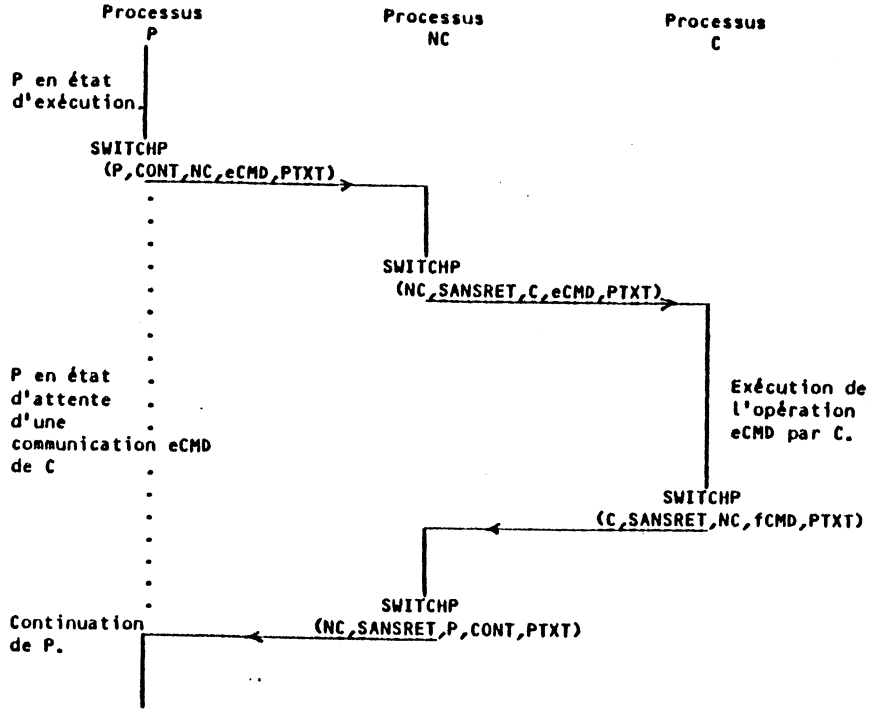


fig. AII.3.1 Exécution d'une primitive de communication.

3.2 Les événement externes (les interruptions externes).

Les événements exterens sont produits par les interruptions provenant des ports externes de la machine.

Un événement externe ne peut pas interrompre l'exécution d'un processus C parce qu'un processus C en train de s'exécuter est en train de changer l'état de la communication du système. Par contre, un processus P en train de s'exécuter, peut être interrompu parce qu'il réalise des opérations sur ses variables internes qui n'ont pas d'effet sur l'état de la communication du système (toute communication et synchronisation entre processus P est faite par l'intermédiaire des processus C).

3.2.1 Le traitement des interruptions externes.

A chaque port externe, on a associé un canal. L'arrivé des signaux et des informations sur un de ces ports se transforme en un message et en l'exécution d'une primitive d'écriture sur le canal correspondant.

A l'arrivée d'une interruption, si le processus en exécution est du type P, il est interrompu et la routine de traitement d'interruptions réalise les opérations suivantes :

- Génération d'un pseudo SWITCHP du processus P, en train de s'exécuter, vers le noyau. Le noyau mémorisera le point de retour de P.

- Génération d'un SWITCHP vers le canal correspondant, pour la prise en compte de l'information provenant du port externe. La primitive de communication sera une écriture sur le canal associé au port. A la fin de cette opération, l'information mémorisée par le canal sera disponible pour être traitée posterieurement par des processus P. Si un processus P avait demandé le traitement asynchrone de cet événement, le canal, à la fin de son opération, demandera au noyau l'exécution de l'activité prévue par le processus P (événement ePRCD).

Dans la figure AII.3.2.1, nous suivons le traitement d'une interruption externe.

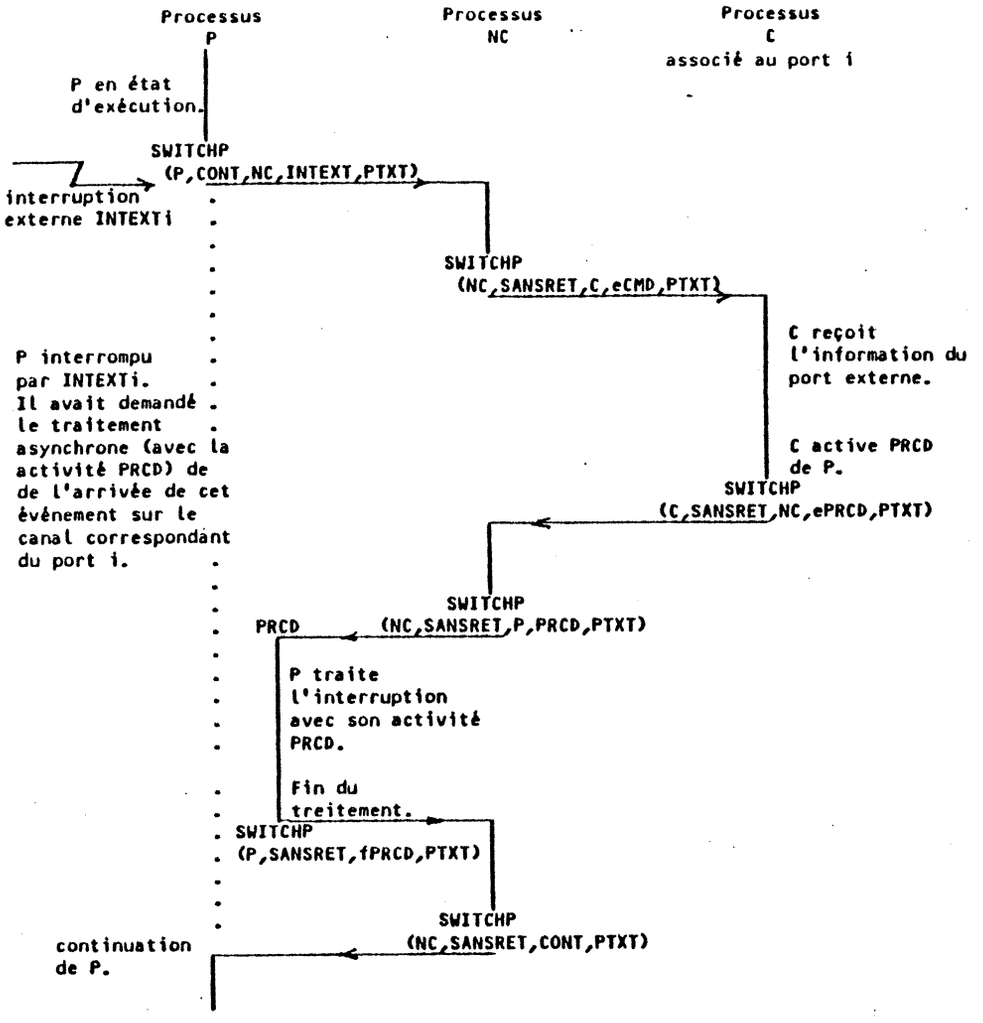


fig. AII.3.2.1 Traitement d'une interruption externe.

4. Les états des procesus.

Pour le contrôle d'exécution des processus, le noyau maintient une table d'états des processus P et C.

4.1 Les états des processus P.

Un processus peut se trouver dans un des trois états suivants :

-EX en exécution,
-ATTE en attente de la fin de l'opération initialisée par C.
-ACT en attente d'être exécuté par le processus noyau.

Un processus en train de s'exécuter (état EX), passe à l'état d'attente quand il demande l'exécution d'une primitive de communication sur un canal (événement eCMD). Une fois l'opération réalisée par le canal

correspondant (événement fCMD), le proessus deviendra activable (état ACT) et attendra être exécuté par le scheduler (événement SCH).

Un processus peut demander le traitement asynchrone d'un événement sur un canal avec une activité (PRCD) prédéfinie par lui (événement ePRCD). Une fois prise en compte cette demande par le canal, le processus P continuera son exécution normale. L'arrivée de cet événement sur le canal impliquera l'interruption de P et l'exécution de l'activité PRCD prévue par P (événement ePRCD). Une fois terminé le traitement, le processus P porsuivra son exécution normale.

La figure AII.4.1 présente le diagramme d'états d'un processus P.

Etats d'un processus P :

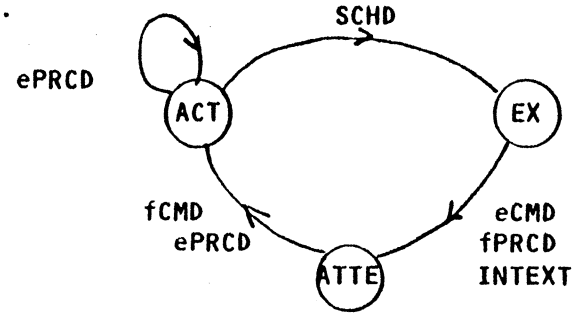
-EX.....le processus P est en train de s'exécuter et peut être interrompu.
-ACT.....le processus attand d'être exécuté par le scheduler.
-ATTE....le processus attend la terminaison d'une primitive de communication sur un canal.

Les événements internes d'un processus P :

-eCMD....P demande l'exécution d'un primitive de communication d'un canal.
-fCMD....C signale la terminaison d'une de ses primitives.
-ePRCD...C demande l'exécution d'une activité pour le traitement d'une interruption.
-fPRCD...P signale la fin du traitement asynchrone.
-SCHD....le scheduler passe le contrôle d'exécution à P.

L'événement externe d'un processus P :

-INTEXT..interruption externe.



événements : états	eCMD	fCMD	ePRCD	fPRCD	SCHD	INTEXT
EX	ATTE			ACT		ACT
ACT			ACT		EX	ACT
ATTE		ACT	ACT	ATTE		ATTE

fig. AII.4.1 diagramme d'éat d'un processus P.

4.2 Les états des processus C.

Un canal peut se trouver dans un des états suivants:

-EX en état d'exécution,
-INAC en état inactif donc disponible à une nouvelle opération,
-ACT en état d'attent d'être exécuté par le noyau.

Un canal passe à l'état activable (état ACT) par la demande d'exécution d'une de ses primitves (eCMD) de la part de P. Le canal devient passe à l'état d'exécution (état EX) une fois séléctioné par le scheduler (événement SCH).

Une fois la primitive de communication termninée, le canal le signalera au noyau (événement fCMD) et passera à un état d'inactivité (état INACT) où il attendra de nouvelles demandes d'exécution de ses primitives.

Dans le cas où un processus P avait demandé le traitement asynchrone d'un événement par une activité prédéfinie (PRCD) et après que l'événement

vienne de se produire avec la dernière opération, le canal demandera alors l'exécution de l'activité correspondante (événement ePCDR).

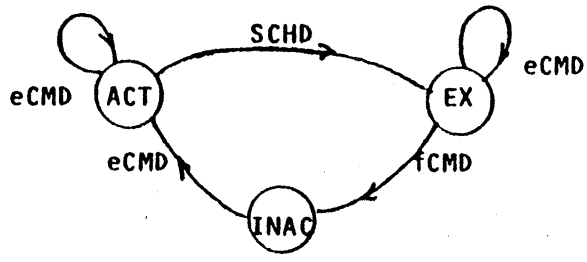
Dans la figure AII.4.2, on montre le diagramme d'états d'un processus C.

Les états d'un processus canal :

-EX.....le canal est en train de s'exécuter et on ne peut pas l'interrompre.
-ACT....le canal a une opération à réaliser et attend d'être exécuté par le scheduler.
-INAC...le canal a terminé son opération et il est prêt à recevoir de nouvelles demandes.

Les événements internes d'un processus canal :

-eCMD...P demande l'exécution d'une primitive de communication sur C.
-fCMD...C signale la terminaison d'une opération.
-SCHD...le scheduler passe le contrôle d'exécution au canal.



événements : états	eCMD	fCMD	SCHD
EX	EX	INAC	
ACT	ACT		EX
INAC	ACT		

fig. AII.4.2 Diagramme d'états d'un processus C.

5. Le scheduler.

Le multiplexage du processeur aux processus P et C suivra le principe selon lequel les processus C seront prioritaires sur les processus P. En plus de cet ordre d'exécution, auront des priorités assignées par le programmeur au moment de leur création.

6.La structure du noyau.

Le noyau se compose de deux procédures : la première, MAJETACOM, fait la mise à jour des états des processus correspondants ; la deuxième, SCHD, sélectione le prochain processus à exécuter selon la politique définie auparavant.

Le noyau est appelé par P, quand il attend un service d'un canal. Le processus C appelle le noyau quand il a terminé le service demandé.

AUTORISATION de SOUTENANCE

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

VU les rapports de présentation de Messieurs

. G. MAZARE, Professeur
. M. GUILLEMONT, Responsable projet
CHORUS à l'INRIA

Monsieur SANCHEZ-ARIAS Victor German

est autorisé à présenter une thèse en soutenance en vue de l'obtention du diplôme de DOCTEUR-INGENIEUR, spécialité "Informatique".

Fait à Grenoble, le 15 janvier 1985

Le Président de l'I.N.P.-G

D. BLOCH
Président
de l'Institut National Polytechnique
de Grenoble

P.O. le Vice-Président,