III. Communication Inter-Processus.

1. Introduction.

Les composants hardware d'une configuration (Unité centrale, mémoires, canal, unités d'entrées/sorties) rendent, dans les systèmes modernes, des services aux programmes applicatifs au travers de différents composants du système d'exploitation : méthode d'accès, gestionnaire de mémoire, scheduler,..

Ces composants sont constitués de un ou plusieurs programmes (séquence d'instructions machine) activés en fonction des demandes de service reçues par le système d'exploitation ; nous utiliserons le terme processus pour désigner une de ces activités.

Un processus est donc la suite d'actions obtenues par exécution d'une séquence d'instructions (un ou plusieurs programmes) dont le résultat peut consister en la réalisation d'une fonction du système d'exploitation. Le concept de processus peut être étendu aux actions obtenues suite à l'exécution d'une séquence d'instructions appartenant à un programme applicatif. On trouve dans la littérature différents termes qualifiant les processus: tasks, deamon, engine,..

Nous verrons qu'un système d'exploitation moderne supporte l'activité concurrente de plusieurs processus, cette activité concurrente est soit réelle (nous disposons alors de plusieurs unités centrales) soit simulées, on parle alors de pseudo-parallélisme.

Le parallélisme, qu'il soit simulé ou non, implique qu'un même programme (rappelons pour la dernière fois : une même séquence d'instructions machine) est exécuté simultanément dans le cadre de plusieurs processus. Il s'ensuit la nécessité de définir des mécanismes permettant à ces processus de communiquer entre eux, car ils sont appelés à collaborer (processus successifs) ou à entrer en concurrence l'un avec l'autre pour l'emploi de ressources uniques. L'emploi d'une ressource unique par plusieurs processus implique des mécanismes de sérialisation, et l'apparition éventuelle d'inter-blocages.

Nous allons développer dans ce chapitre les mécanismes de communication inter-processus et l'étude des inter-blocages (appelés deadlocks). Avant d'aborder le vif du sujet, nous allons nous attarder sur les notions de parallélisme et rappeler quelques éléments de description d'ordinateurs.

.

2. Nomenclature.

A. Systèmes multiprocesseurs.

Ces systèmes disposent de plusieurs unités centrales opérant simultanément. Les termes parallélisme et concurrence sont utilisés pour décrire les opérations simultanées de plusieurs processeurs, qu'ils exécutent ou non le même code.

Dans les années 70, la plupart des constructeurs optèrent pour le développement d'unités centrales rapides et puissantes basées sur le pipelining, ces processeurs basés sur la technologie Emitter-Coupled Logic (ECL) pouvaient traiter des opérations sur vecteurs ou scalaires. Certains tentèrent de développer des machines basées sur de multiple processeurs plus lents sans toutefois parvenir à atteindre les performances des premiers types de machines.

Cependant, les systèmes ECL durent adopter une architecture multiprocesseurs dans les années 80 afin de satisfaire la demande croissante en puissance de traitement. Il est acquis que l'utilisation de plusieurs processeurs est la seule voie permettant d'augmenter la vitesse de traitement de l'information alors que les softwares permettant de faire travailler plusieurs processeurs à la résolution d'un même problème sont encore perfectibles.

B. Classification.

Un système possédant une seule unité centrale exécute une seule séquence d'instructions sur un seul ensemble de données. La classification de Flynn baptise ces systèmes du nom de Single Instruction stream-Single Data stream (SISD).

Dans un système général à multiprocesseurs, chaque séquence d'instructions traite des données différentes, il s'ensuit que ces machines sont appelées MIMD avec M pour multiple.

Il est possible de concevoir un système au sein duquel une instruction est chargée par une unité centrale, diffusée aux autres unités centrales qui l'exécutent chacune sur des données différentes. Il est ainsi possible de traiter en une seule instruction tous les éléments d'un vecteur. Ces machines sont du type SIMD.

Enfin, la dernière combinaison MISD n'existe à proprement parler pas à moins de faire entrer dans cette catégorie les systèmes basés sur le pipelining ou les systèmes à tolérance de panne (Fault-tolerant systems).

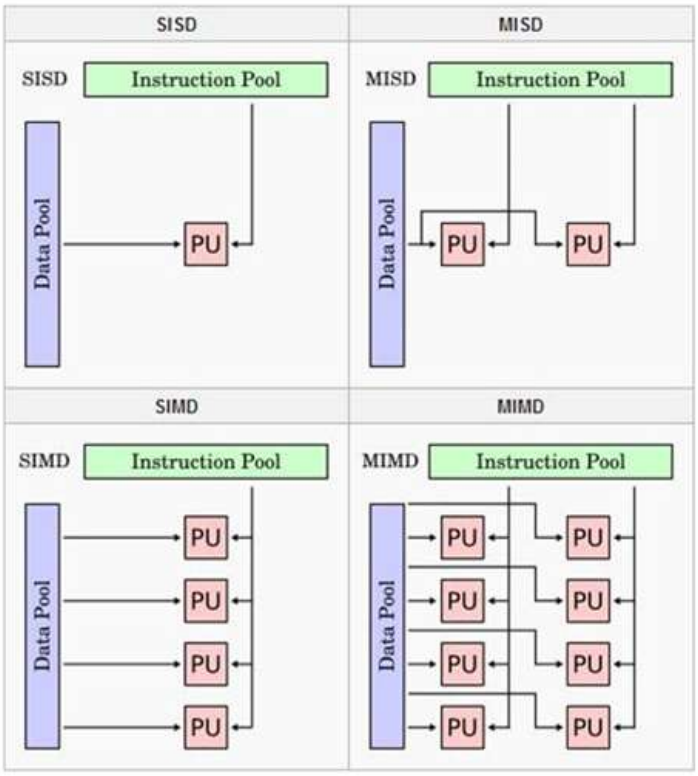

Fig. 17. Classification de Flynn.

C. Multiple Instruction Multiple Data.

Dans ce type de système, plusieurs processeurs indépendants traitent simultanément des données indépendantes. Chaque processeur soit dispose de sa propre mémoire soit accède une mémoire partagée. On distingue deux types d'architecture:

- les multiprocesseurs à mémoire partagée: shared memory
- les multiprocesseurs basés sur l'échange de message: message-passing.

La mémoire partagée est employée comme moyen de communication dans le premier type de système tandis qu'un réseau de communication (basé sur des switches) est employé pour transférer les informations dans le second.

a) Shared Memory Multi-processor.

Chaque unité centrale présente dans la configuration doit pouvoir accéder à la mémoire partagée, il existe différents schémas d'interconnexions entre ces éléments:

a) bus simple
b) bus local et bus système
c) bus multiples
d) cross-bar switch
e) Mémoire multi-porte
f) Réseau de switch à plusieurs niveaux.

Ces schémas sont illustrés à la figure ci-dessous:

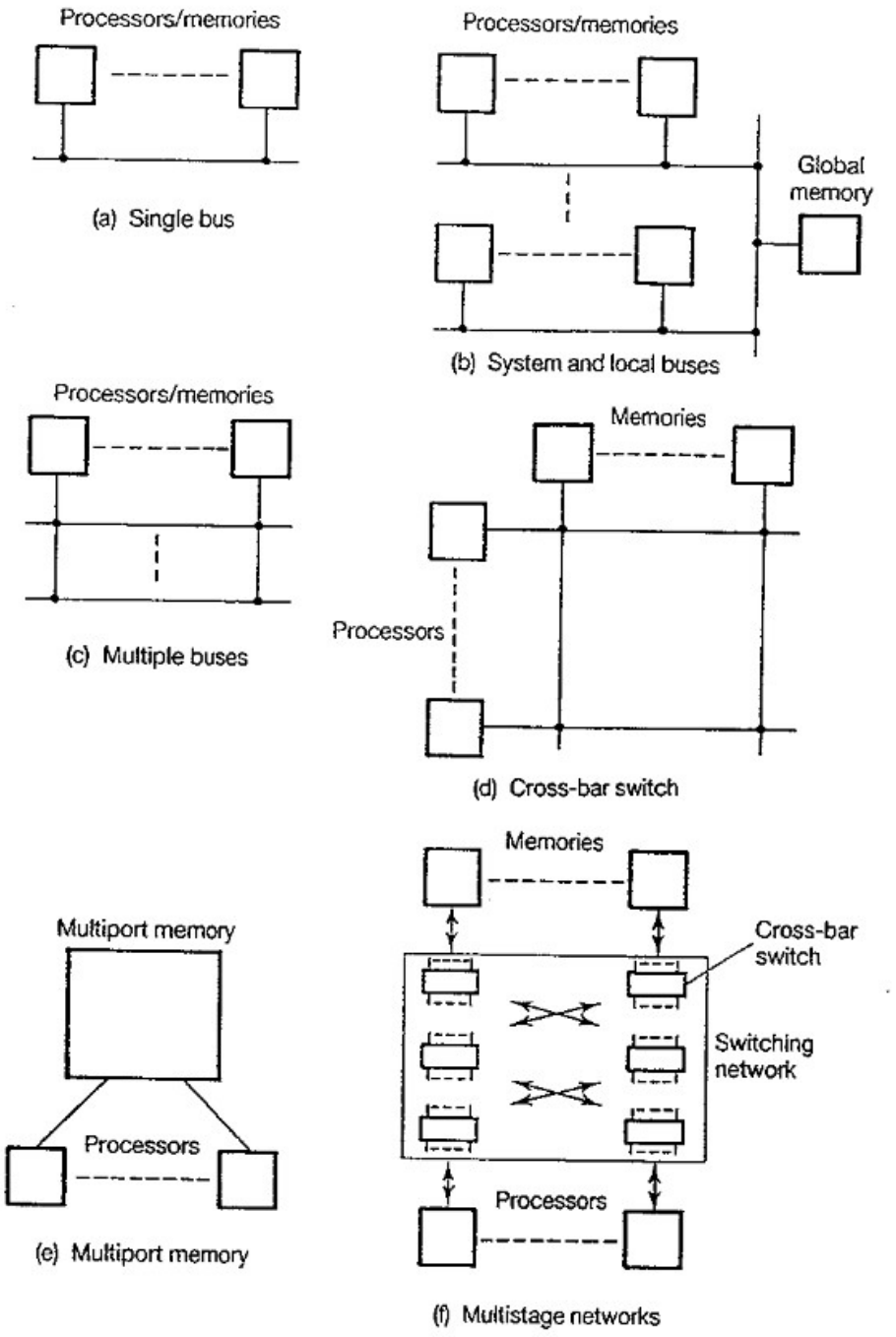

Fig. 18. Shared memory, schémas d'interconnexions.

Retenons que l'emploi d'un bus rend l'architecture simple mais inefficace pour un haut degré de parallélisme ; que le degré de complexité du cross-bar switch augmente rapidement avec le parallélisme ; qu'il en va de même pour le coût de réalisation d'une mémoire multi-portes qui supporte plusieurs accès concurrents. Le réseau de switches à plusieurs niveaux permet de réduire ce coût et cette complexité.

b) Message-passing.

On trouve différents types de liens entre les différents processeurs, certains sont illustrés à la figure ci-dessous.

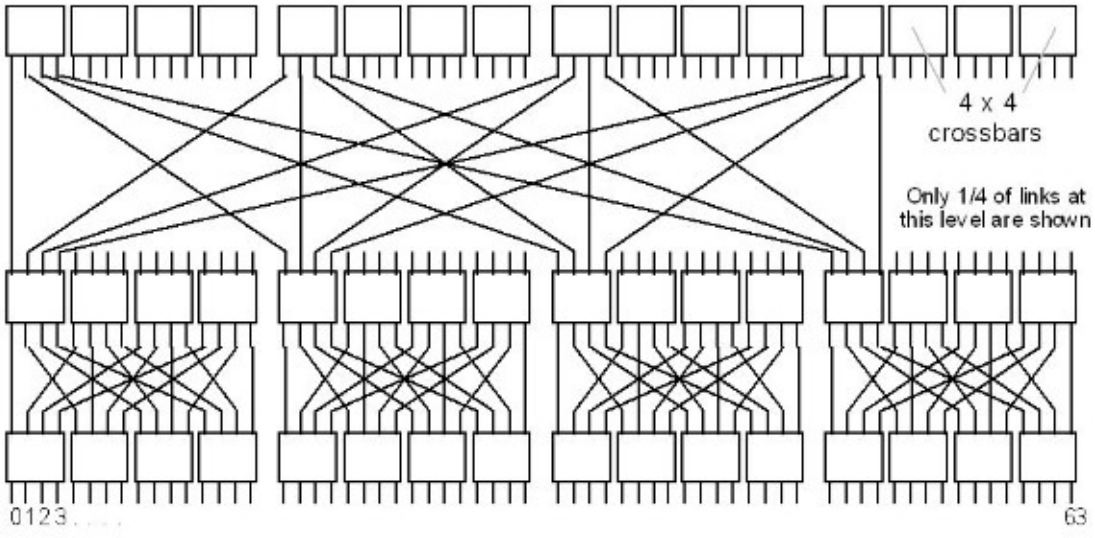

Fig. 19. Schéma d'interconnexion d'un système IBM SP2.

Un exemple de réalisation dans le système IBM SP2, les liens sont réalisés par un réseau de switches rapides, le réseau est à plusieurs niveaux suivant le nombre de processeurs.

D. Facteur d'Amdahl.

Afin de pouvoir exploiter les architectures parallèles, il faut pouvoir diviser le traitement de l'information en une série tâches ou processus qui pourront être exécutés simultanément. Nous pouvons exprimer le gain en vitesse au moyen du facteur S(n) (« Speedup factor »):

$$S(n) = \frac{t_{uniproc}}{t_{multiproc}}$$

avec $t_{uniproc}$ le temps d'exécution dans un système disposant d'un seul processeur et $t_{multiproc}$ le temps de traitement dans un système à « n » processeurs.

L'efficacité s'exprime comme

$$E(n) = \frac{S(n)}{n} * 100\%$$

Notons que l'efficacité maximale est obtenue lorsque le $S(n) = n$, ce qui est le cas lorsque le traitement demandé peut être décomposé en n processus de durée égale.

Il est cependant probable qu'une partie du traitement ne puisse être découpée en tâches parallèles, cette partie du traitement doit être sérialisée. Supposons donc qu'une fraction f du traitement doit être exécutée avant que n processus d'égale durée ne puissent continuer. Le facteur d'accélération s'exprime alors:

$$S(n) = \frac{t}{ft + (1-f)\,t/n} = \frac{n}{1 + (n-1)f}$$

Cette expression est appelée loi de Amdahl ; d'après cette loi, un nombre infini de processeurs nous permettrait d'atteindre un facteur $S(n)$ limité à $1/f$.

L'évolution de ce facteur est montrée à la figure ci-dessous:

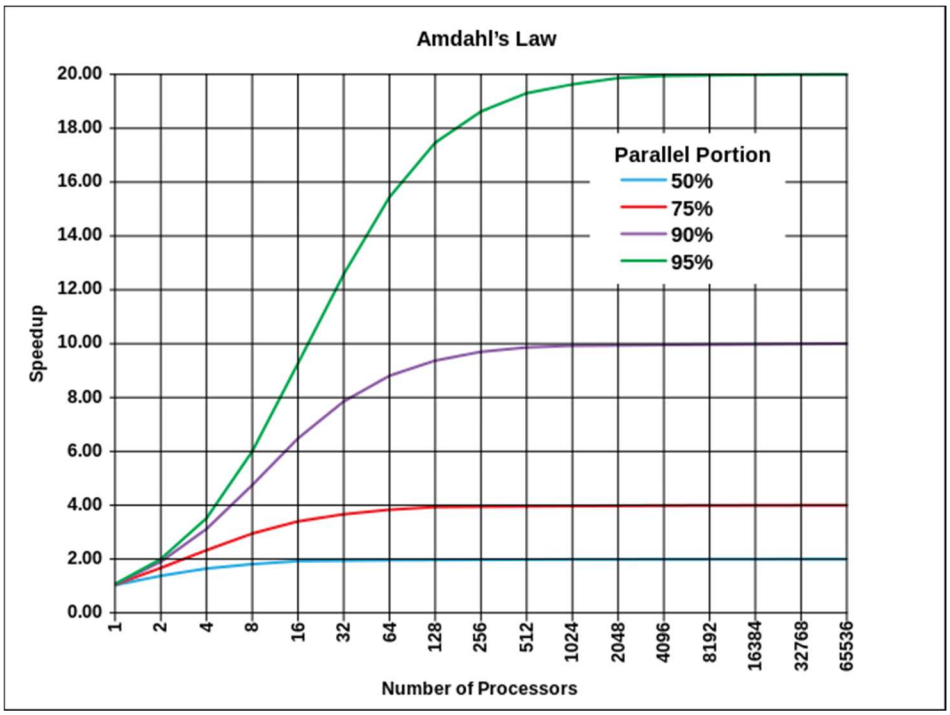

Fig. 20. Evolution du facteur d'accélération.

Des théories plus avancées permettent d'exprimer le gain en vitesse en fonction de l'overhead dû à la communication entre processus.

3. Programmation des systèmes à multiprocesseurs.

A. Processus

Avant d'aborder le corps du sujet, il est nécessaire de rappeler certaines notions fondamentales déjà expliquées dans le cours de première année.

Le processus se définit comme un programme actif (soit une séquence d'instructions) accompagné des valeurs contenues dans les registres généraux et spéciaux (PSW, Control Register, Access Register…) ainsi que des valeurs des variables manipulées par le programme. Pour rappel, il ne s'agit donc ni d'un processeur ni d'un programme (un même programme peut en effet être activé simultané au sein de différents processus).

Un processus ou tâche est un traitement effectué par une unité centrale auxquelles sont associés des input et des output. Un processus pourrait se résumer à une instruction machine.

B. Parallélisme des processus.

Certains traitements peuvent être directement parallélisés et peuvent profiter d'une architecture hautement parallèle (traitement d'images…), d'autres demandent une transformation plus fondamentale pour profiter du parallélisme.

Ces transformations peuvent être réalisées de deux manières:

- soit par le programmeur qui adopte une découpe d'un traitement lui permettant d'éclater ce traitement en processus parallèles, on parle de parallélisme explicite;
- soit par un compilateur reconnaissant les éléments potentiels pouvant être rendus parallèles, on parle de parallélisme implicite;

a) Parallélisme explicite.

Le premier exemple de structure parallèle a été appliqué dans le langage Fortran, et ensuite en Unix. En Fortran, une instruction FORK génère un processus qui entrera en concurrence avec les autres processus, les processus concurrents peuvent exécuter un JOIN à la fin pour se resynchroniser, à ce moment le traitement peut reprendre de manière séquentielle.

b) Parallélisme implicite.

Le parallélisme implicite est décidé par un compilateur lors de sa traduction d'une source en instructions machine ; il faut donc que ce compilateur possède une logique de détection des conditions permettant de paralléliser certains traitements.

Bernstein a établi en 1966 un ensemble de conditions qui sont suffisantes pour déterminer si plusieurs processus peuvent s'exécuter en parallèle. Ces conditions sont basées sur les variables employées par les processus réduits aux instructions.

Définissons donc les variables I (input) et O (Output) de telle sorte que

I_i représente l'ensemble des variables lues par le processus P_i
O_i représente l'ensemble des variables modifiées par le processus P_i

Pour que deux processus puissent s'éxécuter en parallèle, l'input du premier ne doit pas être l'output du second et inversément. Ces conditions s'expriment par

$$I_1 \bigcap O_2 = \varnothing$$

$$I_2 \bigcap O_1 = \varnothing$$

où $\varnothing$ représente l'ensemble vide. Les outputs de chaque processus doivent également être distincts :

$$O_1 \bigcap O_2 = \varnothing$$

Si les trois conditions peuvent être satisfaites, les deux processus peuvent être exécutés en parallèle. Les conditions de Bernstein peuvent être appliquées aux boucles d'un langage de haut niveau si ces boucles appliquent des traitements à des vecteurs ou des tableaux.

c) Pseudo-parallélisme

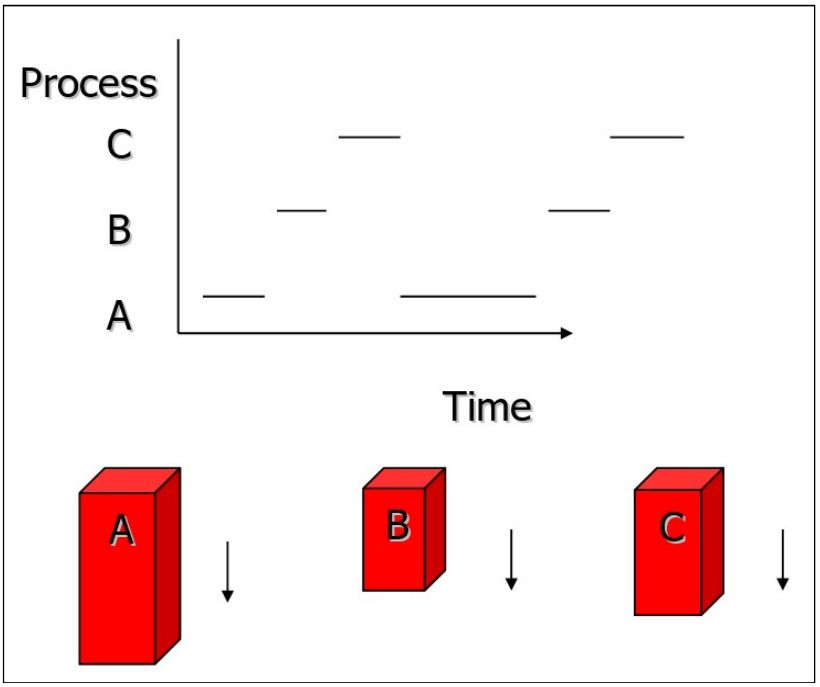

Fig. 21.Pseudo-parallélisme.

Un processeur unique peut donner l'impression de parallélisme suite aux interruptions subies par des programmes actifs et par l'activation d'un programme choisi suivant un algorithme dit algorithme de scheduling. Cet algorithme peut être basé sur différents

principes : round-robin, mean time to wait, ..., nous nous étendrons sur le sujet dans le chapitre consacré à la gestion des ressources.

Il ne faut pas confondre le pseudo-parallélisme avec le parallélisme hardware : un I/O se déroulant pendant que le processeur exécute d'autres instructions, pipelining des instructions dans un processeur.

Il est clair que ce parallélisme ne peut être avantageux que lorsque les tâches ou processus peuvent se chevaucher, sinon l'effort réalisé par l'OS pour créer plusieurs tâches et pour les contrôler, ajouté à l'effort nécessaire à la synchronisation et la communication entre processus serait inutile.

C. Systèmes d'exploitation.

Un système d'exploitation moderne possède deux caractéristiques fondamentales:

- Simultanéité ou pseudo-parallélisme
- Non-déterminisme

Il permet en effet de dérouler des tâches simultanément. Si le nombre de processeurs est plus grand que le nombre de tâches, pas de problèmes. Par contre si à l'inverse, le nombre de tâches à activer est supérieur au nombre de processeurs, le système d'exploitation active une tâche après l'autre (task switching et pseudo-parallélisme).

Le non-déterminisme signifie qu'une interruption peut survenir n'importe quand sans influencer le résultat du processus interrompu. Le déroulement d'un processus ne pouvant être influencé par le moment auquel l'interruption survient, il est nécessaire de sauvegarder l'état d'un processus que l'on appelle en général le contexte du processus.

D. Vie des processus

Les processus sont créés par le système d'exploitation en réponse à des appels systèmes (SVC, System Call…). Citons pour les systèmes d'exploitation les plus courants:

- Création: Attach (MVS), Fork (Unix), Load_and_exec (MS-Dos)
- Exécution: Exec
- Destruction: Detach (MVS), Exit (Unix), End_prog (MS-Dos)

Ces processus, une fois créés, traversent une série d'états:

- Running : actifs
- Ready : prêt à être activés
- Blocked : en attente d'un événement extérieur

Un processus est mis sur une *ready queue* et est activé suivant un algorithme de gestion des priorités implémenté dans le *scheduler*. Le passage direct de l'état blocked à l'état running n'est pas permis, un passage par la ready queue est obligatoire.

4. IPC: Inter-Process Communication

A. Généralités.

Des processus actifs (pseudo-)simultanément au sein d'un système d'exploitation sont soit:

- concurrents : suite au partage de ressources
- coopérants : car ils ont pour mission un partage du travail

a) Exclusion mutuelle

L'exclusion mutuelle doit être mise en œuvre pour des processus concurrents sur l'emploi de ressources non-partageables, soit des ressources physiques (hardware), soit des ressources logiques (software). Cette exclusion mutuelle est mise en œuvre au sein d'un processus via une section critique: une séquence d'instructions qui assure au processus l'emploi exclusif d'une ressource commune.

b) Synchronisation

La synchronisation doit être mise en œuvre pour des processus coopérants: un processus consommateur doit être mis en attente du résultat d'un processus producteur

c) Blocages

Les inter-blocages peuvent survenir en présence d'exclusions mutuelles sur au moins deux ressources. Imaginons, deux processus parallèles (Pa et Pb), utilisant de manière exclusive deux ressources (Ra et Rb). Un inter-blocage entre les deux processus peut survenir si la séquence d'emploi des ressources est la suivante:

- Pa exclut Pb de l'utilisation de la ressource Ra
- Pb exclut Pa de l'utilisation de la ressource Rb
- Pa demande l'utilisation de Rb
- Pb demande l'utilisation de Ra
- Deadlock

d) Race condition

En cas de nécessité d'exclusion mutuelle sur l'emploi d'une ressource commune et en l'absence de sections critiques, une condition dite 'Race condition' peut survenir. Tout processus sensible à cette condition se déroule dans un OS non-déterministe: le résultat de leur exécution différera en fonction du temps.

B. Section critique

Comme expliqué ci-dessus, l'emploi d'une ressource commune doit être protégé par une séquence d'instructions. Cette séquence d'instructions intitulée section critique doit obéir à certaines règles:

1. deux processus ne peuvent se trouver simultanément dans la même section critique
2. aucun processus hors d'une section critique ne peut bloquer les autres
3. aucun processus ne doit attendre indéfiniment avant d'entrer en section critique
4. aucune hypothèse ne peut être faite ni sur les vitesses relatives des processus ni sur le nombre de processeurs

Les sections critiques peuvent être réalisées de différentes manières:

1. Empêcher les interruptions
2. Tester une variable commune
3. Jouer l'alternance stricte
4. Dérouler la solution de Peterson et/ou utiliser TS ou CS
5. Utiliser les primitives Sleep et Wakeup
6. Employer des sémaphores

Les deux premières implémentations posent des problèmes divers.

- Il est dangereux d'empêcher les interruptions, un processus fautif peut en effet oublier de les permettre à nouveau et donc monopoliser un processeur. De plus cette solution est inapplicable dans un environnement multiprocesseurs. Une interruption peut être inhibée sur un processeur mais pas sur 10 ou 20 processeurs.

- Le test d'une variable commune ne peut être réalisé en une seule opération, les décisions prises sur base de ce test sont donc potentiellement fausses. Imaginons qu'un processus teste une variable lui permettant d'entrer dans sa section critique, qu'il se fasse interrompre immédiatement après le test et qu'un autre processus exécute la même opération, chacun des deux se trouvera en même temps dans la section critique.

Dcl shared lock=FALSE ;

Processus 0

```
While (TRUE)  {
        while (lock) /* attente */;
        lock = TRUE ;
        section_critique();
        lock = FALSE ;
        section_non_critique();
}
```

Processus 1

```
While (TRUE)  {
        while (lock) /* attente */;
        lock = TRUE ;
        section_critique();
        lock = FALSE ;
        section_non_critique();
}
```

Fig. 22. Variable commune.

Passons en revue les autres implémentations:

3. Alternance stricte

Processus 0

```
While (TRUE)  {
        while (tour != 0) /* attente */;
        section_critique();
        tour = 1;
        section_non_critique();
}
```

Processus 1

```
While (TRUE)  {
        while (tour != 1) /* attente */;
        section_critique();
        tour = 0;
        section_non_critique();
}
```

Fig. 23. Alternance stricte.

Une variable qui indique à qui le tour. Tant que ce n'est pas son tour, un processus attend. Le processus le plus rapide attend le processus le plus lent, ce qui viole une des règles émises plus haut.

4. Algorithme de Peterson

```
#define  N      2                          /* nombre de processus */

int tour ;                                 /* à qui le tour ? */
int intéressé[N] ;                         /* initialisé à 0 = FALSE */

void entrer_region (int process)       /* demander l'accès à une section critique */
{                                          /* numéro du processus : 0 ou 1 */
        int autre;

        autre = 1 - process;               /* numéro de l'autre processus */
        intéressé[process] = TRUE;         /* indiquer qu'on est intéressé */
        tour = process;                    /* positionner le drapeau */
        while (tour == process && intéressé[autre]== TRUE);     /* busy wait */
}

void quitter_region (int process)      /* indiquer sortie de la section critique */
{
intéressé[process] = FALSE;
}
```

Fig. 24. Solution de Peterson.

Cet algorithme combine l'emploi des variables « lock » (« intéressé ») et « tour ». La variable lock appartient à l'autre processus. Un processus entre dans sa section critique si c'est son tour et si l'autre processus n'est pas intéressé. Avantage, quand un processus n'est pas dans sa section critique, il ne bloque pas l'autre. Mais les inconvénients sont que le test « while » n'est pas indivisible donc une interruption peut survenir après le test sur « tour ». D'autre part, les cases du tableau « intéressé » peuvent être positionnées à True pour les deux processus qui peuvent donc être mis en attente définitive.

a) TS ou CS

Il existe dans le jeu d'instruction de nombreux processeurs des instructions qui sont indivisibles: elles ne peuvent être interrompues (comment interrompre une instruction machine ?), mais réalisent logiquement plusieurs opérations:

- lecture d'une adresse mémoire
- comparer et stocker la valeur dans un registre
- stocker une nouvelle valeur à l'adresse mémoire

Ces opérations sont indivisibles y compris entre plusieurs CPU.

Il existe deux instructions de ce type dans l'instruction set S/390:

TS (Test and Set) dont le principe est: le bit de gauche du byte à l'adresse du second operand est utilisé pour positionner le *Condition code (PSW)*, les bits constituant le byte à l'adresse du second operand sont positionnés à 1.

CS (Compare and Swap): le premier et le second operandes sont comparés. En cas d'égalité, le troisième opérande est stocké à l'adresse du second, sinon le second opérande est stocké à l'adresse du premier opérande. Le condition code indique le résultat de la comparaison.

```
entrer_region
        tsl registre,drapeau        | copier drapeau dans registre
                                    | et mettre drapeau à 1
        cmp registre,#0             | drapeau égal à 0 ?
        jnz entrer_region           | si différent de 0 (verrou mis), boucler
        ret                         \ retour à l'appelant ; entrée section critique

quitter_region
        mov drapeau,#0              | mettre drapeau à 0
        ret                         | retour à l'appelant
```

Fig. 25. Emploi de TS/CS/tsl.

La figure ci-dessus illustre l'emploi d'une instruction tirée de l'instruction set Intel. On remplace le test while qui n'était pas nécessairement indivisible par l'instruction TSL. Cette instruction positionne un flag qui est unique. Il suffit de vérifier le résultat de la lecture pour décider d'attendre car un processus est entré dans sa section critique.

La solution de Peterson et les solutions basées sur l'instruction de type TS ou CS sont de type « Busy Waiting ». En fait on boucle jusqu'à ce que le signal soit vert. De plus, ce mécanisme peut mener à une inversion des priorités, lorsqu'un processus à haute priorité commence à boucler en attente d'une section critique qu'il partage avec un processus à basse priorité.

5. Solution avec Sleep et wakeup

```
#define  N      100                  /* nombre d'emplacements dans le tampon */
int compteur = 0 ;                   /* nombre d'objets dans le tampon */

void producteur (void)
{
   int objet ;

   while (TRUE) {                    /* système sans fin ! */
        produire_objet(&objet);      /* produire l'objet suivant */
        if (compteur == N) sleep();  /* si tampon plein, dormir */
        mettre_objet(objet);         /* mettre l'objet dans le tampon */
        compteur = compteur + 1;     /* incrémenter compteur objets dans tampon */
        if (compteur == 1)           /* si le tampon était vide */
           wakeup(consommateur);     /* réveiller le consommateur */
   }
}

void consommateur (void)
{
   int objet ;

   while (TRUE) {                    /* système sans fin ! */
        if (compteur == 0) sleep();  /* si tampon vide, dormir */
        retirer_objet(objet);        /* retirer l'objet du tampon */
        compteur = compteur - 1;     /* décrémenter compteur objets dans tampon */
        if (compteur == N-1)         /* si le tampon était plein */
           wakeup(consommateur);     /* réveiller le producteur */
        consommer_objet(objet);      /* exploiter l'objet */
   }
}
```

Fig. 26. Solution basée sur Sleep et Wake-up.

Afin d'éviter le gaspillage de ressources entraîné par les solutions de type Busy Waiting, on peut faire appel à des primitives systèmes qui mettent les processus en attente ou qui permettent de les faire sortir de leur attente.

- Sleep : primitive qui permet de faire passer un processus de l'état Running à l'état blocked (similaire à la macro wait expliquée en première)
- Wakeup : primitive qui permet de faire passer un processus de l'état Blocked à Ready (similaire à la macro Post expliquée en première année)

Ces primitives, bien qu'évitant la condition busy waiting, n'empêchent pas l'apparition d'une race condition. En effet, si un processus se fait interrompre avant de dormir, le signal de wake-up peut être perdu et le processus ne se réveillera jamais.

6. Sémaphores

Pour résoudre les problèmes entraînés par les solutions précédentes, Dijkstra a introduit la notion de sémaphores. Les sémaphores peuvent être assimilés à des compteurs dans lesquels les appels à la primitive wake-up sont accumulés. La valeur de ce compteur est décrémentée suite à l'exécution d'une primitive Sleep. Un processus est mis en attente si la valeur d'un sémaphore sur lequel il exécute une primitive Sleep vaut 0. Il sera réactivé si ce même sémaphore subit une opération Wake-up du fait d'un autre processus.

Les sémaphores possèdent des propriétés intéressantes que nous allons étudier dans la suite. Une série de problèmes IPC trouvent une solution grâce à leur emploi.

i. Propriétés

La valeur d'un sémaphore Val(sem) obéit à tout moment à la relation suivante:

$$N(Wakeup) + C \geq N(Sleep)$$

N(Wakeup) = Nombre d'opérations Wakeup
N(Sleep) = Nombre d'opérations Sleep
C = la valeur initiale du sémaphore

L'égalité n'est vérifiée que pour une valeur du sémaphore égale à 0.

Démontrons cette relation,

Soit le nombre de sleep = nombre de wakeup + valeur initiale.
A tout instant, Val(Sem) = N(Wakeup) - N(Sleep) + C.
Par définition , Val(sem) est toujours ≥ 0 d'où la relation.

ii. Exclusion Mutuelle et sémaphores

L'exclusion mutuelle peut être obtenue par l'emploi des sémaphores:

```
Declare Mutex Semaphore
Sleep(Mutex)
Process Section_critique
Wakeup(Mutex)
```

Dans ce cas, la valeur initiale de Mutex doit être 1:

$$N\,(Sleep) - N\,(Wakeup) \leq 1$$

iii. Synchronisation et sémaphores

Imaginons deux processus (P1 et P2) dont l'un (P1) exploite le résultat de l'autre (P2):

Processus 1	Processus 2
Declare Proceed Semaphore;	*Declare Proceed Semaphore*
Sleep (Proceed);	*Process Process_2;*
Process Process_1;	*Wakeup(Proceed);*

La valeur initiale du sémaphore doit être égale à 0. Le processus P1 doit attendre la fin de l'exécution du processus P2 qui est marquée par le wakeup sur le sémaphore.

$$N\,(\text{Sleep}) \leq N\,(\text{Wakeup})$$

Cette relation exprime simplement que le nombre d'exécutions de Wakeup doit être plus élevé que le nombre d'exécutions de Sleep.

C. Problèmes IPC.

Nous allons passer en revue dans ce paragraphe des problèmes classiques de communication inter-processus et leur solutions basées sur l'emploi des sémaphores.

a) Producteur et consommateur.

Le problème du consommateur et du producteur se résume à deux processus accédant un réservoir commun (directory), le consommateur vide le réservoir des éléments (slots) placés par le producteur.

Adoptons les symboles suivants pour:

Capacité directory : N
Nombre de slots écrits : P
Nombre de slots lus : G

La relation suivante doit être respectée:

$$0 \leq P - G \leq N$$

Elle exprime simplement que le réservoir a une capacité limitée et qu'en conséquence, le nombre d'éléments présents à tout moment dans ce réservoir ne peut dépasser sa capacité.

Une section critique doit protéger les manipulations de la directory, on emploiera donc Sleep et Wakeup sur un sémaphore appelé Mutex.

Le producteur doit attendre sur la condition Buffer Full, il exécutera donc Sleep (#Free), avec #Free le nombre de slots libres. Le consommateur doit attendre sur la condition Buffer Empty. Il exécutera donc Sleep (#Full). Avec #Full le nombre de slots pleins.

La solution s'écrit donc

Producteur	Consommateur
Sleep(#Free)	*Sleep(#Full)*
Sleep(Mutex)	*Sleep(Mutex)*
PUT	*GET*
Wakeup(Mutex)	*Wakeup(Mutex)*
Wakeup(#Full)	*Wakeup(#Free)*

Démontrons la relation:

#Free = N
Mutex = 1
#Full = 0
W = Wakeup
S = Sleep

C(#Free) = N
C(#Full) = 0

1) $W(\#Free) + N \geq S(\#Free)$
2) $W(\#Full) + 0 \geq S(\#Full)$
3) $S(\#Free) \geq P \geq N(\#Full)$
4) $S(\#Full) \geq G \geq W(\#Free)$
5) $W(\#Free) + N \geq S(\#Free) >= P$
6) $W(\#Free) \geq P - N$
7) $G \geq P - N$
8) $[\ N \geq P - G\ ] \geq 0$

b) Les philosophes

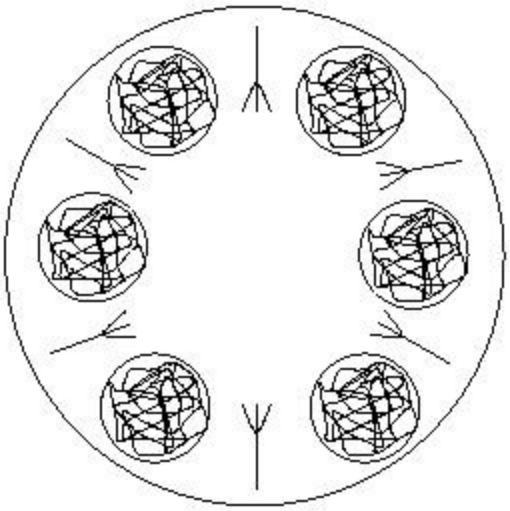

Fig. 27. Problème des philosophes.

Cinq philosophes sont disposés autour d'une table ronde. Ils disposent de cinq fourchettes pour manger chacun une assiette de spaghetti. Deux fourchettes sont nécessaires pour manger les spaghetti. Un philosophe ne fait que deux choses dans sa vie: il mange ou il pense ; mais il peut être dans trois états : mangeur, penseur ou affamé quand il a faim et qu'il attend de pouvoir manger !

```
#define  N            5          /* nombre de philosophes */
#define GAUCHE       (i-1)%N    /* (i-1) modulo N = N° du voisin de gauche */
#define DROITE       (i+1)%N    /* Numéro du voisin de droite */
#define  PENSE       0          /* le philosophe pense */
#define  FAIM        1          /* le philosophe a faim, attend les fourchettes */
#define  MANGE       2          /* le philosophes mange */

typedef int semaphore ;         /* le sémaphore = type d'entier */
int stat[N] ;                   /* pour mémoriser l'état des philosophes */
semaphore mutex = 1             /* pour excl. mutuelle dans section critique */
sema^hore s[N]                  /* un sémaphore par philosophe */

void philosophe (int i)         /* i : numéro du philosophe entre 0 et N-1 */
{
   while (TRUE) {               /* système sans fin ! */
        penser() ;              /* le philosophe commence par penser */
        prendre_fourchettes(i); /* demander 2 fourchettes pour philosophe i */
        manger() ;              /* le philosophe mange */
        poser_fourchettes(i);   /* redéposer les 2 fourchettes */
   }
}
void prendre_fourchettes(int i) ;   /* demander les 2 fourchettes du philosophe i */
{
   down(&mutex) ;               /* entrée dans la section critique */ ;
   stat[i] = FAIM ;             /* mémoriser philosophe i a faim */
   test(i) ;                    /* essaie d'obtenir 2 fourchettes */
   up(&mutex) ;                 /* sortie de la section critique */ ;
   down(&s[i]) ;                /* bloque si pas reçu les 2 fourchettes */
}
void poser_fourchettes(int i) ;     /* redéposer les 2 fourchettes du philosophe i */
{
   down(&mutex) ;               /* entrée dans la section critique */ ;
   stat[i] = PENSE ;            /* philosophe i a fini de manger */
   test(GAUCHE) ;               /* teste si voisin de gauche ou ... */
   test(DROITE) ;               /*... voisin de droite peuvent manger ... */
   up(&mutex) ;                 /* sortie de la section critique */ ;
}
void test(int i) ;              /* vérifier si philosophe i peut obtenir ... */
{                               /* ... ses 2 fourchettes */
   If (stat[i] == FAIM
     && stat[GAUCHE] != MANGE          /* fourchette de gauche est libre */
     && stat[DROITE] != MANGE {        /* fourchette de droite est libre */
        stat[i] = MANGE ;       /* le philosophe i peut manger */
        up(&s[i]) ;             /* le réveiller éventuellement */
   }
}
```

Fig. 28. Solution au problème des philosophes.

c) Lectures et mises à jour

Ecrivons un algorithme qui permette plusieurs lectures concurrentes et une seule mise à jour sans lecture simultanée dans une base de données par exemple.

```
typedef int semaphore ;

semaphore mutex = 1                 /* contrôle l'accès à rc */
semaphore bd = 1                    /* contrôle l'accès à la base de données */

int rc =0 ;                         /* nombre de processus lisant ou voulant lire */

void lecteur(void)                  /* processus pour un lecteur */
{
   while (TRUE) {                   /* système sans fin ! */
        down(&mutex) ;              /* obtenir l'accès exclusif à rc */ ;
        rc = rc + 1 ;               /* un lecteur de plus */
        if (rc == 1) down(&bd) ;    /* si c'est le premier lecteur… */
        up(&mutex) ;                /* libérer l'accès exclusif à rc */
        lire_base_de_donnees() ;
        down(&mutex) ;              /* obtenir l'accès exclusif à rc */ ;
        rc = rc - 1 ;               /* un lecteur de moins */
        if (rc == 0) up(&bd) ;      /* si c'est le dernier lecteur… */
        up(&mutex) ;                /* libérer l'accès exclusif à rc */
        utiliser_donnees_lues() ;   /* section non critique */
   }
}

void redacteur(void)                /* processus pour la mise à jour */
{
   while (TRUE) {                   /* système sans fin ! */
        creer_donnees()             /* préparer… section non critique */
        down(&bd) ;                 /* obtenir l'accès exclusif */
        ecrire_donnees ;            /* mettre à jour la base de données */
        up(&bd) ;                   /* libérer l'accès exclusif à bd */
   }
}
```

Fig. 29. Solution au problème lecture/mise à jour.

d) Le coiffeur endormi (the sleeping barber)

Un coiffeur attend les clients dans son salon, il s'endort en l'absence de client. Un client le réveille, s'assied sur une chaise ou s'en va suivant l'état du coiffeur et le nombre de chaises libres.

Fig. 30. Enoncé du problème du coiffeur endormi.

```
#define  CHAISES          5       /* nombre de chaises pour clients en attente */
typedef int semaphore ;

semaphore clients = 0             /* nombre de clients en attente */
semaphore coiffeurs = 0           /* nombre de coiffeurs en attente */
semaphore mutex = 1               /* pour l'exclusion mutuelle */
int attente =0 ;                  /* nombre de clients en attente */

void coiffeur(void)               /* processus pour un coiffeur */
{
   while (TRUE) {                 /* système sans fin ! */
        down(clients) ;           /* dormir si pas de clients */ ;
        down(mutex) ;             /* obtenir l'accès exclusif à 'attente' */ ;
        attente = attente - 1;    /* décrémenter compteur de clients en attente */
        up(coiffeurs) ;           /* un coiffeur est disponible */
        up(mutex) ;               /* libérer 'attente' */
        couper_cheveux() ;        /* en dehors de la section critique */
   }
}
void client(void)                 /* processus pour un client */
{
   down(mutex) ;                  /* obtenir l'accès exclusif à 'attente' */ ;
   if (attente < CHAISES) {       /* s'il reste des chaises pour attendre */
        attente = attente + 1;    /* incrémenter compteur de clients en attente */
        up(clients) ;             /* réveiller un coiffeur s'il dormait */
        up(mutex) ;               /* libérer 'attente' */
        down(coiffeurs) ;         /* dormir si pas de coiffeur disponible */
        obtenir_coupe() ;         /* s'asseoir et se faire couper les cheveux */
   } else {                       /* plus de chaise libre pour attendre */
        up(mutex) ;               /* libérer 'attente' !!!!! */
   }                              /* ne pas attendre et partir ...*/
}
```

Fig. 31. Solution au problème du coiffeur endormi.