COURS SYSTEME D'EXPLOITATION II

UNIV IBN KHALDOUNE - TIARET

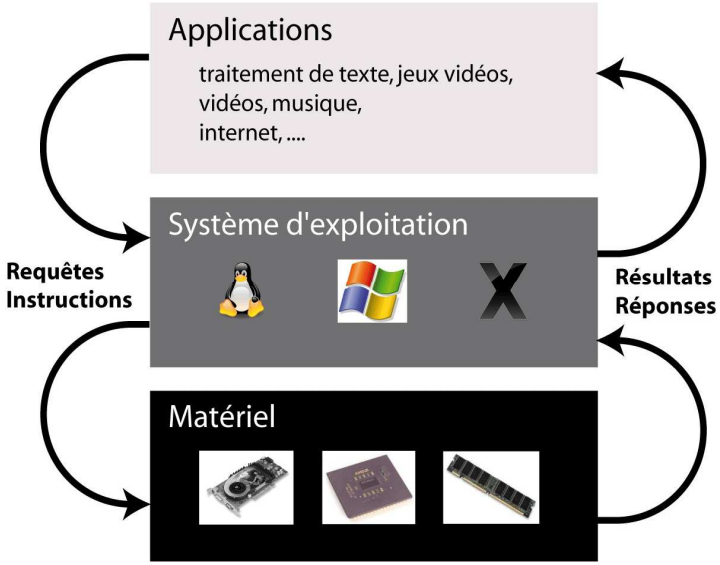
Version non corrigée

2020

Chapitre I : Notions sur les processus et la concurrence

Généralités sur les systèmes d'exploitation

1.Schéma d'un système informatique



Les logiciels peuvent être classés en deux catégories :

- les programmes d'application des utilisateurs

-les programmes système qui permettent le fonctionnement de l'ordinateur. Parmi ceux-ci, le système d'exploitation (SE dans la suite).

Le SE soustrait le matériel au regard du programmeur et offre une présentation agréable des fichiers. Un SE a ainsi deux objectifs principaux :

- **présentation** : Il propose à l'utilisateur une abstraction plus simple et plus agréable que le matériel : une **machine virtuelle**

- **gestion** : il ordonne et contrôle l'allocation des processeurs, des mémoires, des icônes et fenêtres, des périphériques, des réseaux entre les programmes qui les utilisent. Il assiste les programmes utilisateurs. Il protège les utilisateurs dans le cas d'usage partagé.

2. Elements de base d'un systeme d'exploitation

Les principales fonctions assurées par un SE sont les suivantes :

- gestion de la mémoire principale et des mémoires secondaires,
- exécution des E/S à faible débit (terminaux, imprimantes) ou haut débit (disques, bandes),
- multiprogrammation, temps partagé, parallélisme : interruption, ordonnancement, répartition en mémoire, partage des données

- lancement des outils du système (compilateurs, environnement utilisateur,...) et des outils pour l'administrateur du système (création de points d'entrée, modification de privilèges,...),

- lancement des travaux,

- protection, sécurité ; facturation des services,

- réseaux

L'interface entre un SE et les programmes utilisateurs est constituée d'un ensemble d'instructions étendues, spécifiques d'un SE, ou appels système. Généralement, les appels système concernent soit les processus, soit le système de gestion de fichiers (SGF).

3- Les differentes classes des SE

- **Systèmesà machines individuelles**: gestion de l'information, exécution des pgs
- **Systèmesà transactions**: ex: systèmes bancaires, systèmes de réservationPbs :simultanéité,cohérence
- **Systèmes de commande**: enregistrement, régulation, sécurité
- **Systèmes à temps partagé**: un ensemble de services, un ensemble d'usagers pbs : partage et allocation des ressources, gestion des infos partagés
- S**ystèmes à temps réel et à temps différé** : telephonie, mail

Avec la grande diffusion des micro-ordinateurs, l'évolution des performances des réseaux de télécommunications, deux nouvelles catégories de se sont apparus :

- **Les se en réseaux :** ils permettent à partir d'une machine de se connecter sur une machine distante, de transférer des données. Mais chaque machine dispose de son propre se
- **Les se distribués ou répartis** : l'utilisateur ne sait pas où sont physiquement ses données, ni où s'exécute son programme. Le se gère l'ensemble des machines connectées. Le système informatique apparaît comme un mono-processeur.

- **Les se embarques**un système embarqué peut être défini comme un système électronique et informatique autonome, qui est dédié à une tâche bien précise. Ses ressources disponibles sont généralement limitées. Cette limitation est généralement d'ordre spatial (taille limitée) et énergétique (consommation restreinte).

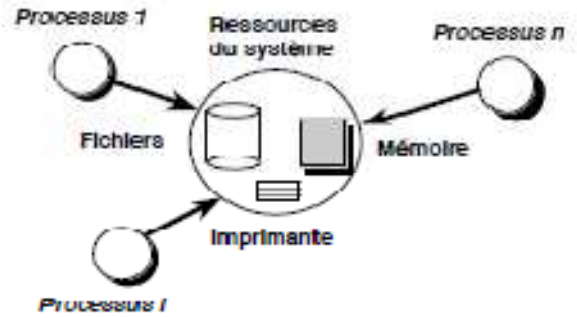
Concurrence,Processus et Ressources

1.Introduction

Deux processus sont concurrents s'ils existent en même temps

Les processus concurrents s'exécutant dans le système d'exploitation peuvent être desprocessus coopératifs ou indépendants.

Un système d'exploitation dispose de ressources (imprimantes, disques, mémoire, fichiers, base de données, ...), que les processuspeuvent vouloir partager.



Ils sont alors en situation de concurrence (race) vis-à-vis des ressources. Il faut synchroniser leurs actions sur les ressources partagées.

En effet, une architecture classique de machine, même mono-processeur, autorise le parallélisme des échanges d'information entre les périphériques (disques, bandes imprimantes, scanners, CD,. . .) et la mémoire centrale d'une part, et d'autre part, l'exécution d'un programme par le(s) processeur(s) avec cependant un problème de contention et de partage

d’accès à la mémoire centrale qui apparaît alors comme une ressource commune partagée

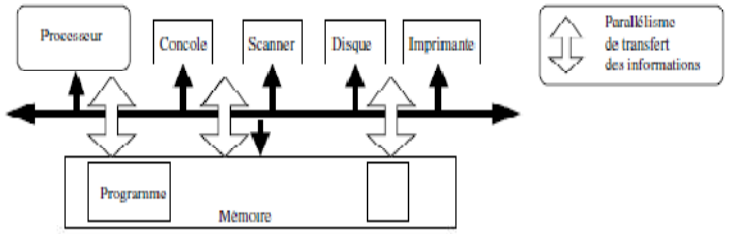


FIGURE 2.1 Parallélisme entre entrées/sorties et exécution d'un programme

2. Actualité de la concurrence

La concurrence entre processus est un sujet de grande actualité pour plusieurs raisons.

1. L'évolution des systèmes personnels : Dos, Windows, Mac, fait passer ceux-ci d'uneorganisation mono processus à des systèmes multiprogrammés.

2. Les systèmes classiques, MVS, AS 400 (IBM), VMS (Digital) , Solaris (Sun), Unix, Linux,qui sont multiprogrammés dès leur conception continuent à se développer et sont de plus enplus souvent multiprocesseurs.

3. Les applications temps réel, embarquées, etc se développent et requièrent des systèmestemps réel multiprocessus (on dit, dans ce métier, des exécutifs multitâches)

4. Les applications réparties se multiplient et chacun peut désormais écrire ou utiliser desprocessus concurrents à distance avec des applets ou des servlets Java. Les systèmesd'architecture client-serveur, les architectures réparties à base d'objets ou de composants vontaussi dans le sens d'une augmentation du nombre des processus concurrents.

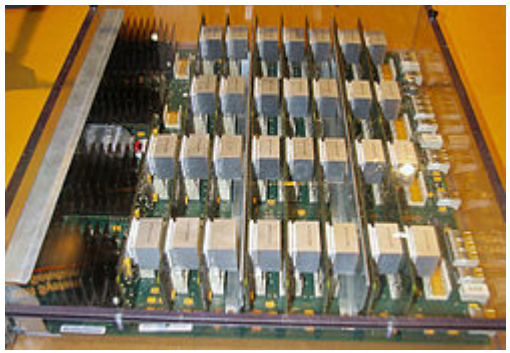
5. les clients nomades, les plates-formes mobiles, ne peuvent fonctionner sans utiliserdes processus mandataires ou proxys qui sont leurs représentants sur les serveurs fixes deressource.

6- L’arrivée des processeurs multicoeurs.

La connaissance des bons comportements et des bonnes méthodes de programmation etd'utilisation de ces processus concurrents est indispensable.

3. Architecture matérielle :

Systèmes à plusieurs processeurs

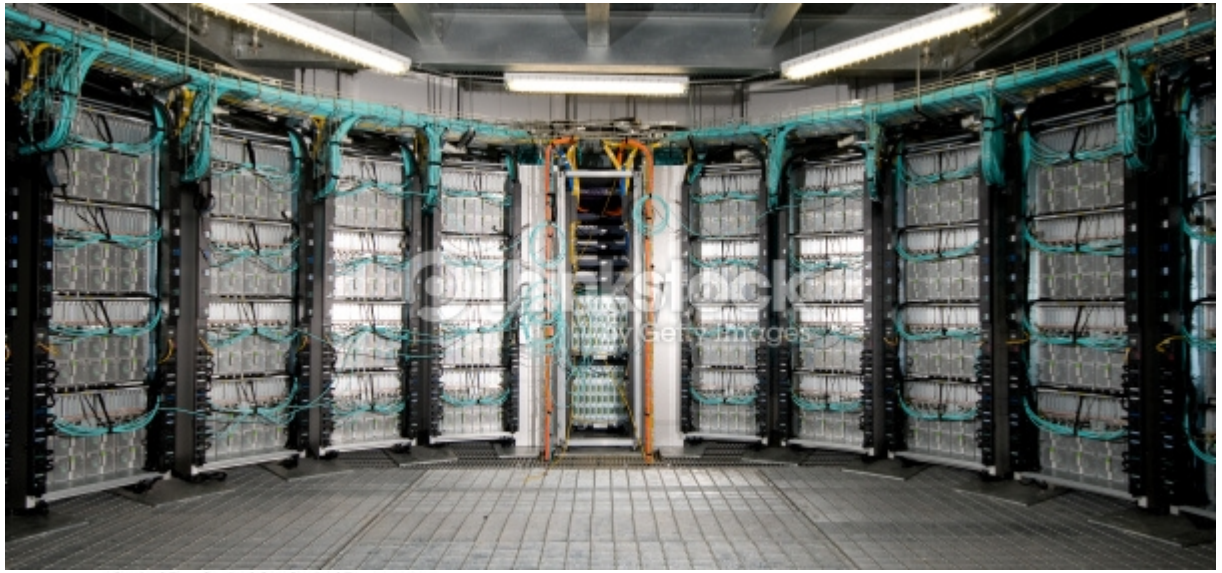


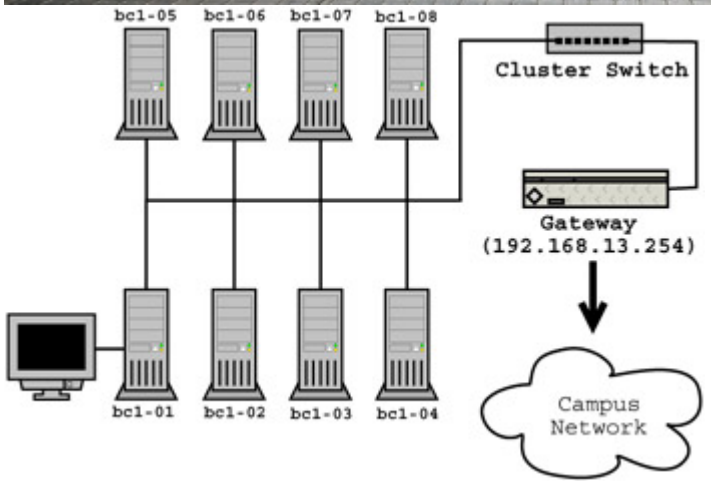
Le Blue Gene L dispose de seize cartes-nœuds pour un total de 440 cœurs organisés suivant le schéma de droite

Systèmes à plusieurs machines

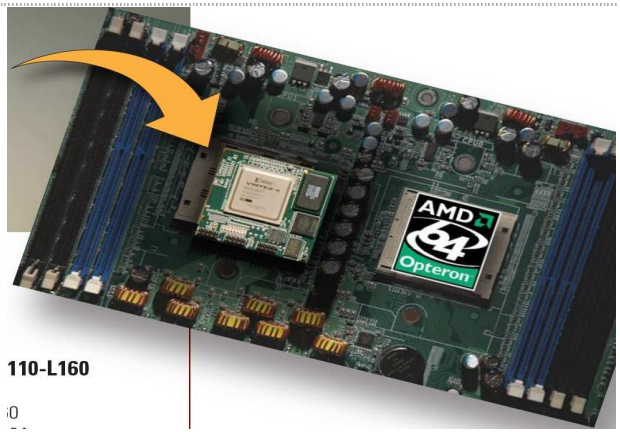
-Réseaux

- Cluster




bc1-05
bc1-06
bc1-07
bc1-08
Cluster Switch
Gateway
(192.168.13.254)
bc1-01
bc1-02
bc1-03
bc1-04
Campus
Network

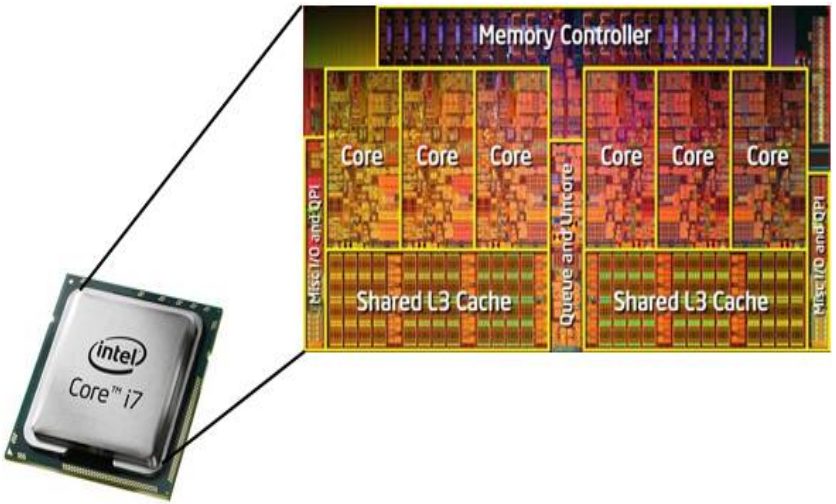
Coprocesseurs



une architecture classique de machine, même mono-processeur, autorise le parallélisme des échanges d'information entre les périphériques (disques, bandes imprimantes, scanners, CD,. . .) et la mémoire centrale d'une part, et d'autre part, l'exécution d'un programme par le(s) processeur(s) avec cependant un problème de contention et de partage d'accès à la mémoire centrale qui apparaît alors comme une ressource commune partagée

Processeurs multicœurs et multithreads

Avec l'arrivée massive des processeurs multi-cœur, le parallélisme peut devenir une nécessité pour une application afin exploiter toute la puissance disponible d'une machine proposant ce type d'architecture.



- plusieurs coeurs parallèles dans un même processeur
- spécifique par coeur : unités de contrôle/calcul + mémoire
- partagé : unités de contrôle communes + mémoire partagée

Exemple 10 : multiplication de matrices

Ci-dessous le code d'initialisation des matrices :

```
01 static double[,] InitializeMatrix(int rows, int cols)
02         {
03             double[,] matrix = new double[rows, cols];
04
05              Random r = new Random();
06              for (int i = 0; i < rows; i++)
07              {
08                  for (int j = 0; j < cols; j++)
09                  {
10                      matrix[i, j] = r.Next(100);
11                  }
12              }
13              return matrix;
14         }
```

La boucle séquentielle:

```
01 static void MultiplyMatricesSequential(double[,]
   matA, double[,] matB,
02  double[,] result)
03         {
04             int matACols = matA.GetLength(0);
05             int matBCols = matB.GetLength(0);
06             int matARows = matA.GetLength(0);
07
08             for (int i = 0; i < matARows; i++)
09             {
10                 for (int j = 0; j < matBCols; j++)
11                 {
```

```
12                      double temp = 0;
13                      for (int k = 0; k < matACols; k++)
14                      {
15                       temp += matA[i, k] * matB[k, j];
16                      }
17                      result[i, j] = temp;
18                  }
19              }
20          }
```

Et la boucle parallélisée:

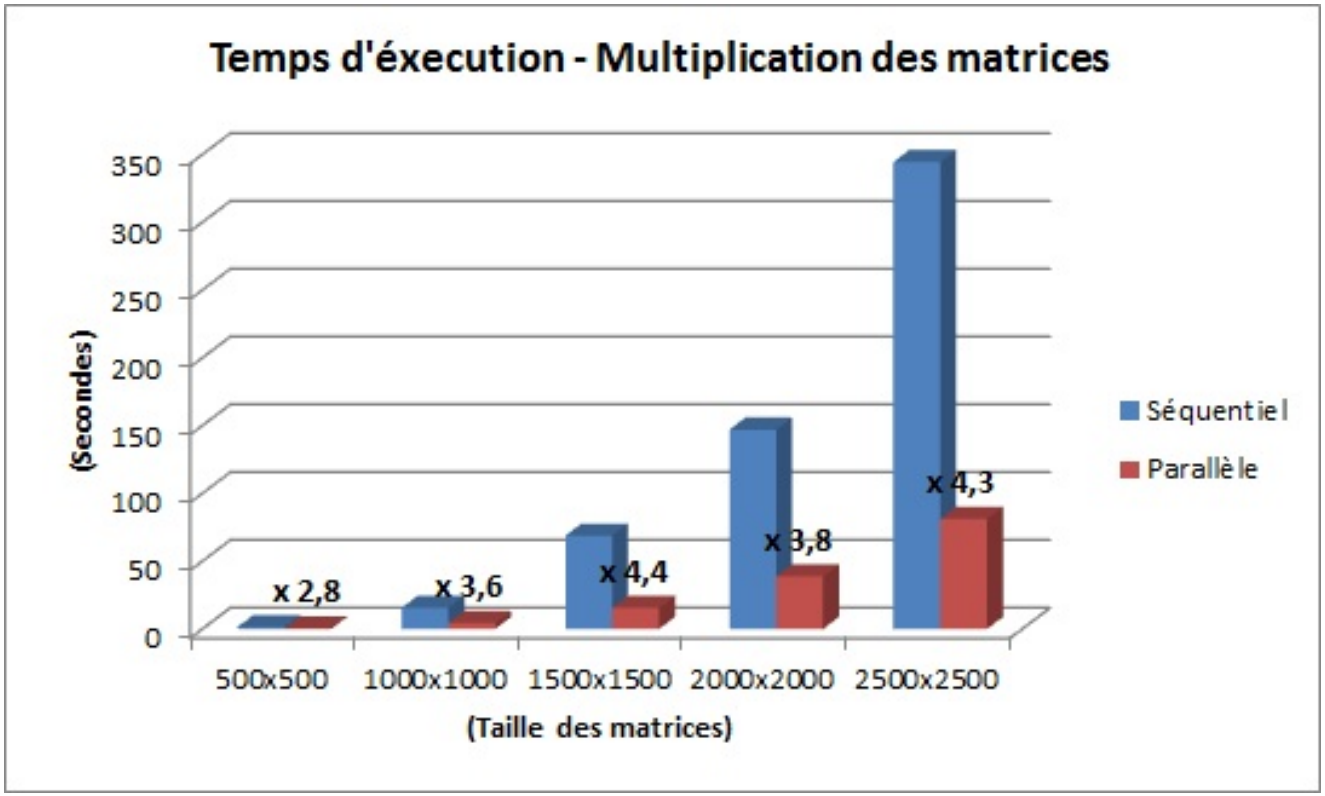
```
01  static void MultiplyMatricesParallel(double[,]
    matA, double[,] matB,double[,] result)
02          {
03              int matACols = matA.GetLength(1);
04              int matBCols = matB.GetLength(1);
05              int matARows = matA.GetLength(0);
06
07              // Multiplication des matrices
08      // Parallélisation de la boucle externe pour
    partitionner
09              //la matrice source par ligne.
```

```
10  Parallel.For(0, matARows, i =>
11              {
12                  for (int j = 0; j < matBCols; j++)
13                  {
14                      double temp = 0;
15                      for (int k = 0; k < matACols; k++)
16                      {
17                      temp += matA[i, k] * matB[k, j];
18                      }
19                      result[i, j] = temp;
20                  }
21              });
```

```
22          }
```

La machine sur laquelle les tests sont effectués embarque un processeur Intel Core i7 Quad-Core HT.

Les résultats sont logiquement ceux espérés.


Temps d'éxecution - Multiplication des matrices
(Secondes)
350
300
250
200
150
100
50
0
x 2,8
x 3,6
x 4,4
x 3,8
x 4,3
500x500
1000x1000
1500x1500
2000x2000
2500x2500
(Taille des matrices)
Séquentiel
Parallèle

4. Les processus

Un processus est un programme qui s'exécute, ainsi que ses données, sa pile, son compteur ordinal, son pointeur de pile et les autres contenus de registres nécessaires à son exécution.

Un processus est un programme en mémoire pourêtreexécuté.

Dans une entité logique unique, généralement un mot, le SE regroupe des informations-clés sur le fonctionnement du processeur : c'est le **mot d'état du processeur** PSW, Il comporte généralement :

- la valeur du compteur ordinal
- des informations sur les interruptions (masquées ou non)
- le privilège du processeur (mode maître ou esclave)
- etc.... (format spécifique à un processeur)

A chaque instant, un processus est caractérisé par son **état courant** : c'est l'ensemble des informations nécessaires à la poursuite de son exécution (valeur du compteur ordinal, contenu des différents registres, informations sur l'utilisation des ressources). A cet effet, à tout processus, on associe un **bloc de contrôle de processus** (BCP).

un **bloc de contrôle de processus** (BCP) comprend généralement :

- une copie du PSW au moment de la dernière interruption du processus
- l'état du processus : prêt à être exécuté, en attente, suspendu, ...
- des informations sur les ressources utilisées
  - mémoire principale
  - temps d'exécution
  - périphériques d'E/S en attente
  - files d'attente dans lesquelles le processus est inclus, etc...
- et toutes les informations nécessaires pour assurer la reprise du processus en cas d'interruption

Les BCP sont rangés dans une table en mémoire centrale à cause de leur manipulation fréquente.

5. Les interruptions

Une interruption est une commutation du mot d'état provoquée par un signal généré par le matériel. Ce signal est la conséquence d'un événement interne au processus, résultant de son exécution, ou bien extérieur et indépendant de son exécution.

6. Les ressources

On appelle **ressource** tout ce qui est nécessaire à l'avancement d'un processus (continuation ou progression de l'exécution) : processeur, mémoire, périphérique, bus, réseau, compilateur, fichier, message d'un autre processus, etc... Un défaut de ressource peut provoquer la mise en attente d'un processus.

Un processus demande au SE l'accès à une ressource. Certaines demandes sont implicites ou permanentes (la ressource processeur). Le SE **alloue** une ressource à un processus. Une fois une ressource allouée, le

processus a le droit de l'utiliser jusqu'à ce qu'il **libère** la ressource ou jusqu'à ce que le SE reprenne la ressource (on parle en ce cas de **ressource préemptible**, de préemption).

Une ressource est soit *locale*, soit *commune* à plusieurs processus

locale : utilisée par un seul processus. Ex.: fichier temporaire, variable de programme.

commune ou partageable . Ex. : disque, imprimante, fichier en lecture.

Une ressource peut posséder un ou plusieurs *points d'accès* à un moment donné :

* un seul :(**accès exclusif)** si elle ne peut être allouée à plus d'un processus à la fois. On parle de *ressource critique*

Ex: un lecteur de disquettes partagé entre plusieurs utilisateurs. Une zone mémoire partagée en écriture entre plusieurs utilisateurs.

* plusieurs points d'accès: (**accès partagé)** par exemple, un fichier en lecture utilisable par plusieurs processus.

Le SE doit contrôler l'utilisation des ressources dans une **table** indiquant si la ressource est disponible ou non, et, si elle est allouée, à quel processus. A chaque ressource est associée une **file d'attente**, pointée par la table précédente, contenant les BCP des processus qui l'attendent. Chaque fois

qu'un nouveau processus fait une demande de la ressource et que cette dernière n'est pas disponible, son BCP est ajouté en queue de la file d'attente. Lorsqu'une demande survient de la part d'un processus plus prioritaire que celui qui utilise la ressource, on empile l'état de la ressource, on la retire au processus en cours pour l'attribuer par **réquisition** au processus prioritaire.

Dans le cadre des ressources limitées gérées par le système, il n'est pas toujours possible d'attribuer à chaque processus, dès sa création, toutes les ressources nécessaires. Il peut y avoir **blocage** .

7. L'ordonnancement

On appelle ordonnancement la stratégie d'attribution des ressources aux processus qui en font la demande. Différents critères peuvent être pris en compte :

- temps moyen d'exécution minimal
- temps de réponse borné pour les systèmes interactifs
- taux d'utilisation élevé de l'UC
- respect de la date d'exécution au plus tard, pour le temps réel, etc...

Processus et Parallélisme

1 Parallélisme

On appelle **simultanéité** l'activation de plusieurs processus au même moment.

Si le nombre de processeurs est au moins égal au nombre de processus, on parle de simultanéité totale ou vraie, sinon de pseudo-simultanéité.

pseudo-simultanéité : c'est par exemple l'exécution enchevêtrée de plusieurs processus sur un seul processeur. La simultanéité est obtenue par commutation temporelle d'un

processus à l'autre sur le processeur. Si les basculements sont suffisamment fréquents, l'utilisateur a l'illusion d'une simultanéité totale.

Dans le langage de la programmation structurée, on encadre par les mots-clés **parbegin** et **parend** les sections de tâches pouvant s'exécuter en parallèle ou simultanément.

2.Etats de processus :

De façon simplifiée, on peut imaginer un SE dans lequel les processus pourraient être dans trois états :

- **élu** : en cours d'exécution. Un processus élu peut être arrêté, même s'il peut poursuivre son exécution, si le SE décide d'allouer le processeur à un autre processus
- **bloqué** : il attend un événement extérieur pour pouvoir continuer (par exemple une ressource; lorsque la ressource est disponible, il passe à l'état "prêt")
- **prêt** : suspendu provisoirement pour permettre l'exécution d'un autre processus

La gestion des interruptions, la suspension et la relance des processus sont l'affaire de l'ordonnanceur.

3. MODELE DE REPRESENTATION DE PROCESSUS

2.1 Décomposition en tâches

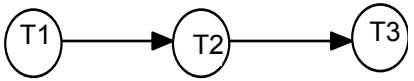
On appelle **tâche** une unité élémentaire de traitement ayant une cohérence logique. Si l'exécution du processus P est constituée de l'exécution séquentielle des tâches T_1, T_2,..., T_n, on écrit :

$$P = T_1\ T_2 ... T_n$$

A chaque tâche T_i, on associe sa date de début ou d'initialisation d_i et sa date de terminaison ou de fin f_i.

La relation $T_i < T_j$ entre tâches signifie que f_i inférieur à d_j entre dates. Si on n'a ni $T_i < T_j$,ni $T_j < T_i$, alors on dit que T_i et T_j sont **exécutables en parallèle**.

Une relation de précédence peut être représentée par un graphe orienté. Par exemple, la chaîne de tâches $S = ((T_1, T_2, T_3), (T_i < T_j$ pour i inférieur à j)) a pour graphe :



2.2 Parallèlisation de tâches dans un système de tâches

La mise en parallèle s'effectue par la structure algorithmique :

parbegin

.....

parend

Exemple :

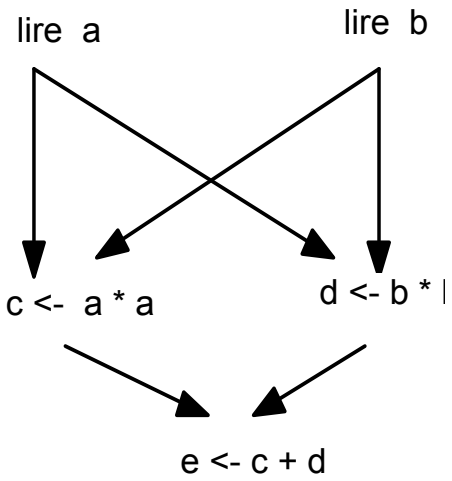
Lire a

Lire b

c<-a*a

d<-b*b

e<-c+d



soit debut

parbegin

lire a

lire b

parend

parbegin

c ← a * a

d ←b * b

parend

e ← c + d fin

exemple 2 : peut-on représenter le graphe suivant de tâches par un programme parallèle utilisant **parbegin** et **parend** ?

exemple 2 : (système S0)

T1 lire X

T2 lire Z

T3 X ← X + Z

T4 Y ← X + Z

T5 afficher Y

2. LE PROBLEME DE L'EXCLUSION MUTUELLE

Definitions

On appelle processus indépendants des processus ne faisant appel qu'à des ressources locales. On appelle processus parallèles pour une ressource des processus pouvant utiliser simultanément cette ressource. Lorsque la ressource est **critique** (ou en accès exclusif),

on parle d'**exclusion mutuelle** (par exemple, sur une machine monoprocesseur, l'UC est une ressource en exclusion mutuelle).

Def.: On appelle **section critique** la partie d'un programme où la ressource est seulement accessible par le processus en cours. **Il faut s'assurer que deux processus n'entrent jamais en même temps en section critique sur une même ressource.**

Exemple : la mise à jour d'un fichier (deux mises à jour simultanées d'un même compte client).La section critique comprend :

- lecture du compte dans le fichier,
- modification du compte,
- réécriture du compte dans le fichier.

Programmation multitache :

Definitions :

Def.:*programme multitâche* : ensemble de plusieurs processus séquentiels dont les exécutions sont imbriquées.

Def.: On appelle *interblocage* la situation où tous les processus sont bloqués.

Ex.: chacun attend que l'autre lui envoie un message pour continuer.

Ex.: chacun exécute une boucle d'attente en attendant une ressource disque indisponible. C'est la situation d'un carrefour avec priorité à droite pour tous et des arrivées continues de véhicules.

Def.: on appelle *privation* la situation où quelques processus progressent normalement en bloquant indéfiniment d'autres processus. C'est la situation d'un carrefour giratoire avec 4 voies d'accès dont deux ont des arrivées continues de véhicules.

Critères d'un programme mutlitache juste :

Règle 1: les processus doivent être en relation fortuite. **La défaillance d'un processus en dehors d'une section critique ne doit pas affecter les autres processus**.

Règle 2: un programme multitâche est *juste* s'il répond aux critères de *sécurité* comme l'exclusion mutuelle.

Règle 3 : Un programme multitâche est *juste* s'il répond aux critères de *viabilité* comme la non privation ou le non interblocage.

Les techniques de synchronisation entre processus concurrents peuvent être classés en deux catégories : par attente active et par attente passive.

Chapitre II : la synchronisation par attente active

Algorithmes de synchronisation par attente active :

1* masquage des interruptions : le processus qui entre en section critique masque les interruptions et ne peut donc plus être désalloué (exemple : accès du SE à la table des processus). Mais s'il y a plusieurs processeurs ou si le processus "oublie" de démasquer les interruptions, il y a problème

En ce cas, les processus restent dans la liste des processus prêts puisqu'ils ont toujours quelque chose à tester : d'où une consommation inutile de temps UC

2.1 La méthode des coroutines

Algorithme :

```
int tour ;/* variable de commutation de droit à la section critique */
          /* valeur 1 pour P1, 2 pour P2                              */
main ()
 {
     tour = 1;                /* P1 peut utiliser sa section critique   */
     parbegin
          p1();
          p2() ;
     parend          }
```

```
p1()                                    p2()
 {                                       {
  for ( ; ; )                             for (; ; )
  {                                       {
 while (tour == 2);   /* c'est au tour de P2 ;   while (tour == 1);    /* c'est au tour de P1 ;
                                  P1 attend    */                                  P2 attend      */
          crit1;                                  crit2;
  tour = 2;              /* on redonne       tour = 1;                  /* on redonne
  l'autorisation à P2       */                          l'autorisation à P1   */
  reste1;                                        reste2 ;
     } }                                              } }
```

Avantages :

- **l'exclusion mutuelle est satisfaite.** Pour chaque valeur de *tour* , une section critique et une seule peut s'exécuter, et ce jusqu'à son terme.

- **l'interblocage est impossible** puisque *tour* prend soit la valeur 1, soit la valeur 2 (Les deux processus ne peuvent pas être bloqués en même temps).

- **la privation est impossible**: un processus ne peut empêcher l'autre d'entrer en section critique puisque *tour* change de valeur à la fin de chaque section critique.

Inconvénients :

- P_1 et P_2 sont contraints de fonctionner avec la même fréquence d'entrée en section critique

- Si l'exécution de P_2 s'arrête, celle de P_1 s'arrête aussi: le programme est bloqué.

La dépendance de fonctionnement entre P_1 et P_2 leur confère le nom de **coroutines.**

2.2 Seconde solution

Chaque processus dispose d'une clé d'entrée en section critique (c1 pour P_1, c_2 pour P_2). P_1 n'entre en section critique que si la clé c_2 vaut 1. Alors, il affecte 0 à sa clé c_1 pour empêcher P_2 d'entrer en section critique.

Algorithme :

```
int c1, c2 ;        /* clés de P1 et P2 - valeur 0: le processus est en section critique */
                    /*                   valeur 1: il n'est pas  en section critique      */
    main ()
    {       c1=c2 = 1;    /* initialement, aucun processus en section critique */
        parbegin
            p1() ;
            p2 ();    parend            }
```

```
p1 ()
 {
    for ( ; ; )
{while (c2 == 0);    /* P2 en section
critique, P1 attend  */
c1 = 0  /* P1 entre en sectioncritique*/
    crit1 ;
    c1 = 1;
/* P1 n'est plus en section critique*/
reste1 ;            }     }
```

```
    p2 ()
{
for ( ; ; )
{   while (c1 == 0);       /* P1 en section
                          critique, P2 attend    */
c2 = 0;  /* P2 entre en section critique */
    crit2 ;
    c2 = 1 ;
        /* P2 n'est plus en section critique    */
          reste2;} }
```

Avantage : on rend moins dépendants les deux processus en attribuant une clé de section critique à chacun.

Inconvénients : Au début c_1 et c_2 sont à 1. P_1 prend connaissance de c_2 et met fin à la boucle while. Si la commutation de temps a lieu à ce moment, c_1 ne sera pas à 0 et P_2 évoluera pour mettre c_2 à 0, tout comme le fera irrémédiablement P_1 pour c_1.

La situation $c_1 = c_2 = 0$ qui en résultera fera entrer simultanément P_1 et P_2 en section critique : **l'exclusion mutuelle ne sera pas satisfaite.**

Si l'instruction $c_i = 0$ était placée avant la boucle d'attente, l'exclusion mutuelle serait satisfaite, mais on aurait cette fois interblocage.

A un moment donné, c_1 et c_2 seraient nuls simultanément P_1 et P_2 exécuteraient leurs boucles d'attente indéfiniment.

2.3 Troisième solution

Lorsque les deux processus veulent entrer en section critique au même moment, l'un des deux renonce temporairement.

Algorithme :

```
int  c1,c2;          /* 0 si le processus veut entrer en section critique,  1 sinon      */
main ()
{
     c1= c2 = 1;     /* ni P1,ni P2 ne veulent entrer en section critique au départ */
     parbegin
          p1();
          p2 () ;
     parend          }
```

```
p1(){ for ( ; ; )
 { c1 = 0; /* P1 veut entrer en section
 critique*/
 while (c2 == 0)  /* tant que P2 veut
 entrer en section critique ... */
 {c1 = 1;      /* ....P1 abandonne un
 temps son  intention... */
 c1 = 0;      /*.. puis la réaffirme   */}
 crit1;
 c1 = 1; /* fin de la section crit. de P1*/
    reste1 ;   } }
```

```
 p2() { for ( ; ; )
 {c2 = 0 ; /* P2 veut entrer en sectioncritique
 */
 while (c1 == 0)     /* tant que P1 veut aussi
 entrer en section critique */
  {c2 = 1;      /* .. P2 abandonne un temps
 sonintention .... */
 c2 = 0 ;     /* ..... puis la réaffirme */ }
     crit2;
c2 = 1 ;  /*  fin de la section critique de P2 */
                    reste2 ; } }
```

Commentaires :

- **l'exclusion mutuelle est satisfaite**. (cf. ci-dessus)

- il est possible d'aboutir à la situation où c_1 et c_2 sont nuls simultanément. Mais il n'y aura pas interblocage car cette situation instable ne sera pas durable (Elle est liée à la commutation de temps entre $c_i = 1$ et $c_i = 0$ dans while).

- il y aura donc d'inutiles pertes de temps par **famine limitée**.

2.4 Algorithme de DEKKER

DEKKER a proposé un algorithme issu des avantages des 1ère et 3ème solutions pour résoudre l'ensemble du problème sans aboutir à aucun inconvénient. Par rapport, à l'algorithme précédent, un processus peut réitérer sa demande d'entrée en section critique, si c'est son tour.

Algorithme :

```
int tour ,    /* valeur i si c'est au tour de Pi de pouvoir entrer en section critique  */
c1,c2 ;      /* valeur 0 si le processus veut entrer en section critique,1 sinon        */
main () {      c1 = c2 = tour = 1;   /* P1 peut entrer en section critique, mais... */
                    parbegin  /* ...  ni P1, ni P2 ne le demandent              */
               p1 () ; p2 () ;
                         parend}
```

```
p1 () {for ( ; ; )
{c1 = 0; /*  P1 veut entrer en SC*/
while (c2 == 0)/*  tant que P2 le veut*/
 if (tour == 2)  /* si c'est le tour de P2  */
 {c1 = 1;        /*  . P1 renonce          */
while (tour == 2) ;  /*jusqu'à son  tour*/
c1 = 0;  /*puis réaffirme son intention */}
 crit1;
 tour = 2;       /*  C'est le tour de P2     */
c1 = 1;   /* P1 a achevé sa section crit. */
  reste1;       }     }
```

```
p2 () {  for (; ; )
 {c2 = 0;   /* P2 veut entrer en SC*/
while (c1 == 0)  /* tant que P1 le veut aussi*/
 if (tour == 1)        /* .si c'est le tour de P1*/
    {c2 = 1;        /* ..... P2 renonce ...    */
 while (tour == 1); /*   jusqu'àson tour */
 c2 = 0;  /*  puis réaffirme son intention */}
 crit2;
  tour = 1;   /* C'est le tour de P1      */
 c2 = 1;  /* P2 a achevé sa section critique*/
          reste2; }       }
```

Remarques :

- Si p1 veut entrer en section critique ($c_1 = 0$), alors que p2 le veut aussi ($c_2 = 0$), et que c'est le tour de p1 (tour = 1), p1 insistera (*while* ($c_2 == 0$) sera actif). Dans p2, la même boucle aboutira à c2 = 1 (renoncement temporaire de p2).

- On démontre [Ben-Ari pages 51 à 53] que cet algorithme résout l'exclusion mutuelle sans privation, sans interblocage, sans blocage du programme par arrêt d'un processus.

- Cet algorithme peut être généralisé à n processus au prix d'une très grande complexité.

2.5 Algorithme de Peterson

```
Debut

Tour : entier ;

C1,c2 : entier ;

C1 :=C2 :=tour :=1 ;

Paregin

 P1()

P2()

Parend ;
```

```
Processus p1()
Debut
 Faire tj
    C1 :=0 ; Tour :=2 ;
Tq (c2=0 et tour=2) faire ftq
<SC>
C1 :=1 ;
Reste1 ;
fin
```

```
Processus p2()
Debut
 Faire tj
    C2 :=0 ; Tour :=1 ;
Tq (c1=0 et tour=1) faire ftq
<SC>
C2 :=1 ;
Reste1 ;
fin
```

Remarque : L'algorithme de peterson est un pg multitache juste mais il n'est valable que pour deux processus concurrents.

2.6. Algorithme de boulanger

Lamport a proposé l'algorithme de boulanger, une solution logicielle pour N processus

Debut

Tour[0..N-1] : table de booleen

Num[0..N-1]: table d'entiers

Pp :entier ;

Pour i :=2 jusqu'N faire

Debut

Tour[i] :=faux ;

Num[i]:=0;

fin

parbeginp0,p2,…,pn-1 ;parend

```
Processus pi()
var jj :=entier ;
debut
pp :=i ;/* 0 pour processus 0 ; etc
tour[pp] := vrai ;
num[pp] :=Max(NUm)+1
tour[pp ] :=Faux ;
pour jj :=0 jusqua' N faire
      tq(tour(jj) faire
      ftq
      tq ((num(jj)<>0) et   (( num(jj)< num(pp)
ou ((num(jj)=num(pp) et (jj<pp)))
ftq
fpour
<SC>
Num[pp] :=0 ;
Fin
```

3.7. Les instructions atomiques :

Introduction:

On utilise un verrou (une variable booléenne) pour verrouiller l'accès à la section critique.

```
Verrou := faux ;
Processus pi()
Debut
Tq (verrou) faire Ftq
  Verrou :=vrai ;
    <S C>
 Verrou :=faux ;
Fin
```

Il y a des processeurs qui fournissent des instructions câblées spéciales dont le rôle est de rendre atomique la séquence : lecture suivie d'écriture. Il devient alors impossible à deux processeurs exécutant des processus concurrents de lire le Verrou à Faux avant que l'un d'eux l'ait mis à Vrai. les instructions câblées atomiques (on dit aussi indivisibles) les plus usitées sont TAS ("test and set") et SWAP ("exchange to memory").

3.7.1. L'instruction Test And Set (TAS)

L'instruction TestAnd Set s'execute comme une action elementaire et indivisible ; elle peut etre définie comme suit :

```
Tas(verrou : booleen) : Booleen ;
Debut
    Si verrou alors retourne(vrai)
     Sinon
        Verrou := vrai ;
       Retourne (faux) ;
Fin
```

Une instruction Tas executée sur une UC doit s'achever avant qu'une autre puisse s'executer sur une autre UC.

```
Verrou : booleen
Verrou :=faux ;

Processus pi ()
Debut
Tq tas(verrou) ftq
<SC>
Verrou :=faux ;
Fin
```

2.8.2. L'instuction SWAP :

Elle est definie comme suit :

```
Procedure Swap( a,b : booleen)
C: booleen
Debut
C:=a;
 A:=b;
 B:=c
Fin
```

L’EM est assurée comme suit :

```
Verrou: booleen

Verrou :=faux ;

Processus pi()

Var cle : booleen  / * clé : un variable locale pour chaque processus

Debut

Cle : =vrai ;

Tq (clé) faire

  Swap(clé, verrou) ;

Ftq

      <SC>

Verrou :=faux ;Fin
```

Bilan de l'attente active : inconvénients et utilisation

L'attente active a comme inconvénients :

- Immobilisation du processeur simplement pour attendre.
- Risque d'attendre indéfiniment.
- Congestion du bus mémoire avec des accès pour vérifier que l'attente est toujours nécessaire et cela ralentit les accès mémoire, donc a puissance de calcul, du processeur en section critique.

L'attente active est cependant incontournable pour des sections critiques de courte durée

Nouvelle idée : mettre en sommeil un processus qui demande à entrer en section critique dès lors qu'un autre processus y est déjà. C'est l'attente passive.

Chapitre III : la synchronisation par attente passive

3.1 Les sémaphores

La résolution des problèmes multi-tâches a considérablement progressé avec l'invention des sémaphores par E.W. DIJKSTRA en 1965. Il s'agit d'un outil puissant, facile à implanter et à utiliser. Les sémaphores permettent de résoudre un grand nombre de problèmes liés à la programmation simultanée, notamment le problème de l'exclusion mutuelle.

Un **sémaphore** est une structure à deux champs :

- une variable entière, ou valeur du sémaphore. Un sémaphore est dit **binaire** si sa valeur ne peut être que 0 ou 1, **général** sinon.

- une file d'attente de processus ou de tâches.

Dans la plupart des cas, la valeur du sémaphore représente à un moment donné le nombre d'accès possibles à une ressource.

Seules deux fonctions permettent de manipuler un sémaphore :

- **P (s)** ou down (s) ou WAIT (s)

- **V (s)** ou up (s) ou SIGNAL (s)

3.1.1 P (s)

La fonction P(S) décrémente le sémaphore d'une unité à condition que sa valeur ne devienne pas négative.

P(s): SI s>0 alors s=s-1

Sinon suspendre l'exécution du processus en cours et le placer le processus dans la file d'attente

du sémaphore Fsi

3.1.2 V (s)

La fonction V(S) incrémente la valeur du sémaphore d'une unité si la file d'attente est vide et si cette incrémentation est possible.

V(s): Si Fat (file d attente) est vide alors s=s+1

Sinon reveiller un processus dans la file d'attente du sémaphore, Fsi

Remarques:

1. Du fait de la variable globale ***verrou***, les fonctions **P** et **V** sont **ininterruptibles** (on dit aussi **atomiques**). **Les deux fonctions P et V s'excluent mutuellement.** Si P et V sont appelées en même temps, elles sont exécutées l'une après l'autre dans un ordre imprévisible.

2. On supposera toujours que le processus réveillé par V est **le premier entré** dans la file d'attente, donc celui qui est en tête de la file d'attente.

3.1.3 Application des sémaphores à l'exclusion mutuelle

Avec le schéma de programme précédent :

```
        SEMAPHORE s;   /* déclaration très symbolique */
    main ()
     {
            SEMAB (s,1);    /* déclaration très symbolique ; initialise le sémaphore
binaire s à 1 */
            parbegin
            p1 () ;
                 p2 () ;       /* instructions très symboliques
            parend                                              */
         }
```

```
p1 ()       /*  premier processus   */
{
   for ( ; ; )
   {
   P (s);
   /* section critique de p1    */
   V (s);
    /* section non critique de p1 */
          }
    }
```

```
p2 ()       /*  second processus   */
   {
        for ( ; ; )
          {
        P (s);
    ......  /* section critique de p2        */
    V (s);
            ...  /* section non critique de p2   */
        }
        }
```

La démonstration formelle que l'algorithme résout l'exclusion mutuelle et ne crée pas d'interblocage est donnée dans Ben-Ari page 63. Si l'on étend cet algorithme à n processus (n > 2) , il peut y avoir privation (par exemple, P1 et P2 peuvent se réveiller mutuellement et suspendre indéfiniment P3 ou d'autres processus).

J.M. Morris a proposé en 1979 un algorithme de résolution de l'exclusion mutuelle sans privation pour n processus dans *Information Processing Letters, volume 8 , pages 76 à 80* .

Exemple :

Dans un système informatique, on dispose de trois fichiers F1, F2 et F3 et de trois processus dont les programmes A,B et C ont les structures suivantes :

Programme A	Programme B	Programme C
actions A1	actions B1	actions C1 (lire F3)
actions A2 (lire F2)	actions B2 (ecrire F3)	actions C2
actions A3	actions B 3(lire F1)	actions C 3
actions A4 (ecrire F3)	actions B4	actions C4 (ecrire F2)
actions A5		actions C5

Chaque fichier ne peut être lu et modifié en même temps.

1°) Donner pour chaque fichier, les sections critiques de A,B et C en exclusion mutuelle.

3°) Assurer l'exclusion mutuelle avec les sémaphores.

Exemple : avec deux processus p1 et p2, deux sémaphores s1 et s2 (s1 = s2 = 1)

p1	p2
....	.....
P (S1)	P (S2)
.....	.....
P (S2)	P (S1)

.....

V (S2)

Si p1 fait P (S1) alors S1 = 0

puis p2 fait P (S2) et S2 = 0

puis p1 fait P (S2) et p1 est en attente, ENDORMI

puis p2 fait P (S1) et p2 est ENDORMI

Il y a donc évidemment interblocage (puisque aucun des processus ne peut faire V (On dit aussi étreinte fatale ou deadlock)

Si les primitives P et V ne sont pas utilisées correctement (erreurs dans l'emploi, le partage ou l'initialisation des sémaphores), un problème peut survenir.

Propriétés des sémaphores

® intérêts des sémaphores

• mécanisme simple, pas d'attente active

• meilleur que le masquage des interruptions (conceptuellement)

• meilleur que l'attente active (ressources processeur et canal)

• mécanisme logique de base

• peu coûteux si bien implémenté au bon niveau (noyau)

® critiques des sémaphores

• A) difficile à utiliser avec fiabilité

l'oubli d'un V(mutex) peut conduire à interblocage

l'oubli d'un P(mutex) est une faute d'exclusion mutuelle

on retrouve ces problèmes pour tous les mécanismes quand les procédures ou

les appels systèmes peuvent être emboîtés

• B) mécanisme non structuré; la synchronisation peut apparaître n'importe oùdans le code - et non pas dans des régions bien localisées - donc la détectiond'erreurs et la mise au point sont plus difficile car la propagation des erreurs esttrès facilitée quand les sémaphores sont placés comme variables globales.

• C) mécanisme trop élémentaire au niveau abstraction

extensions simples des sémaphores

• sémaphores avec messages (utilisés dans HB 64 renommé DPS 7)

• sémaphores avec P non bloquant et retour d'un compte rendu booléen

• sémaphores avec P bloquant seulement pendant un délai maximal

• sémaphores avec P(S, n) augmentant l'entier E.S de n, V(S,m) diminuant l'entier E.S de m (m, n ! 0 et pas seulement n = m = 1)

Les moniteurs

1-Introduction:

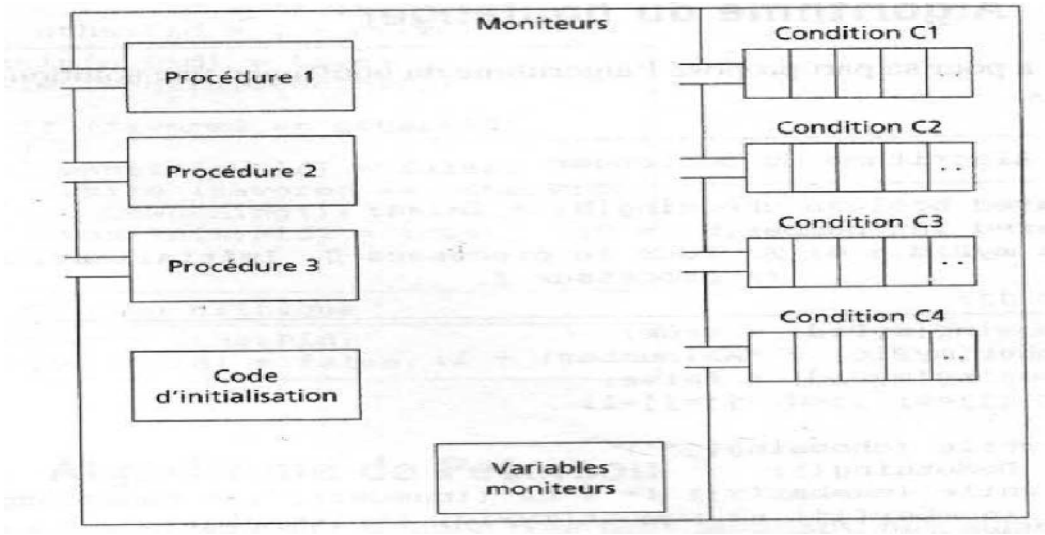
Bien que les semaphores proposent un mécanismegénéral pour contrôler l'accès aux sections critiques, leur utilisation ne garantit pas que cet acces est mutuellement exclusif ou que les inter blocages sont évités. On peut aisément imaginer qu'une erreur de programmation des primitives **P()** et **V()** ou l'oubli d'une de ces primitives peut être très grave : interblocage, incohérence des données, etc...

En tenant compte des critiques citées ci-dessus, on ne peut donc pas concevoir tout un système d'exploitation uniquement à partir des sémaphores. Un outil de programmation plus sûr s'avère nécessaire.

Définition : Un moniteur est une structure de variables et de procédures pouvant être paramétrée et partagée par plusieurs processus. Ce concept a été proposé par C.A.R. HOARE en 1974 et P. BRINCH-HANSEN en 1975. Le moniteur estune construction du langage de programmation qui garanti un accès approprie a des sections critiques. L'accès contrôle est propose par le langage et non par le programmeur.

Le type moniteur existe dans certains langages de programmation, tels que Concurrent Pascal.

Schéma



Le corps du moniteur est exécuté dès que le programme est lancé pour i**nitialiser** les variables du moniteur. Les variables moniteur ne sont accessibles qu'à travers les procédures moniteur.La seule manière pour un processus d'accéder à une variable moniteur est d'appeler une procédure moniteur.

On peut prévoir plusieurs moniteurs pour différentes tâches qui vont s'exécuter en parallèle. Chaque moniteur est chargé d'une tâche bien précise et chacun a ses données et ses instructions réservées. Si un moniteur M1 est le seul moniteur à avoir accès à la variable u1, on est sûr que u1 est en exclusion mutuelle. De plus, comme les seules opérations faites sur u1 sont celles programmées dans M1, il ne peut y avoir ni affectation, ni test accidentels.

On dit que l'entrée du moniteur par un processus exclut l'entrée du moniteur par un autre processus.

Les moniteurs présentent plusieurs avantages :

- au lieu d'être dispersées dans plusieurs processus, les sections critiques sont transformées en procédures d'un moniteur*modularité et de la systématisation de l'exclusion mutuelle et de la section critique*

- la gestion des sections critiques n'est plus à la charge de l'utilisateur, mais elle est réalisée par le moniteur, puisqu'en fait le moniteur tout entier est implanté comme une section critique.

Des exemples de moniteurs sont donnés dans Beauquier, p. 139-141.

Il utilise des variables de type **condition** et deux primitives agissant sur elles :

- WAIT : bloque le processus appelant et autorise un processus en attente à entrer dans le moniteur

- SIGNAL : réveille le processus endormi en tête de la file d'attente. Puis, ou bien le processus courant est endormi (solution de Hoare), ou bien le processus courant quitte le moniteur (solution de Brinch Hansen, la plus usitée), afin qu'il n'y ait pas deux processus actifs dans le moniteur.

La syntaxe d'un moniteur ressemble plus en moins au code ci apres:

```
Monit: moniteur
Debut
// declaration des variables
i: entier;c: condition
// decalaration des procedures
Procedure SCpoint1()
Debut
// code de procedure
fin
Procedure SCpoint2()
Debut
// code d eprocedure
fin
// intialisation
debut
fin
Fin du moniteur
```

Exemple:

(a+ b) * (c + d) - (e/f)

intérêts du moniteur

• introduction de la modularité et de la systématisation de l'exclusion mutuelle et de la section critique

• se prête bien à une formalisation (mieux que le sémaphore) car :

  • la section critique est déclarée,

  • on peut sérialiser les sections critiques,

  • on peut appliquer des preuves séquentielles à cette sérialisation

• on peut associer des invariants à un moniteur

inconvénients du moniteur

• la synchronisation par wait et signal peut devenir complexe et se révéler d'aussi bas niveau que les sémaphores(des essais pour introduire des sortes d'expressions gardées apportent des

améliorations de fiabilité, mais au coût d'une réévaluation dynamique des gardes,

à des endroits non systématiques)

• problème des appels emboîtés = problème général des exclusions mutuelles emboîtées:

- soit danger d'interblocage si appel de moniteur possible depuis un autre moniteur (P1 dans M1 appelle M2 "" P2 dans M2 appelle M1)

- soit un processus ne doit pouvoir n'appeler qu'un moniteur à la fois : contrainte à la programmation (explicite statique) ou à l'exécution(dynamique)

Chapitre 4 : COMMUNICATION INTER-PROCESSUS

Le modèle producteur-consommateur.

Introduction

Les processus ont besoin de communiquer, d'échanger des informations de façon plus élaborée et plus structurée que par le biais d'interruptions. Un modèle de communication entre processus avec partage de zone commune (tampon) est le **modèle producteur-consommateur.**

Le producteur doit pouvoir ranger en zone commune des données qu'il produit en attendant que le consommateur soit prêt à les consommer. Le consommateur ne doit pas essayer de consommer des données inexistantes.

Hypothèses :

- les données sont de taille constante

- les vitesses respectives des deux processus (producteur consommateur) sont quelconques.

Règle 1 : Le producteur ne peut pas ranger un objet si le tampon est **plein**

Règle 2 : Le consommateur ne peut pas prendre un objet si le tampon est **vide.**

PRODUCTEUR

CONSOMMATEUR

Faire toujours

produire un objet

si nb d'objets ds tampon < N

alors alors déposer l'objet ds le tampon

finsi

Fait

CONSOMMATEUR

Faire toujours

si nb d'objets ds tampon >0

prendre l'objet

finsi

consommer l'objet

Fait

Règle 3 : exclusion mutuelle au niveau de l'objet : le consommateur ne peut prélever un objet que le producteur est en train de ranger.

Règle 4 : si le producteur (resp. consommateur) est en attente parce que le tampon est plein (resp. vide), il doit être averti dès que cette condition cesse d'être vraie.

Le tampon peut être représenté par une liste circulaire. On introduit donc deux variables caractérisant l'état du tampon :

NPLEIN : nombre d'objets dans le tampon (début : 0)

NVIDE : nombre d'emplacements disponibles dans le tampon (N au début).

PRODUCTEUR :

Faire toujours Produire un objet

/* début d'atome ininterruptible */

si NVIDE >0 /* s'il existe un

emplacement vide dans le tampon */

alors NVIDE --sinon s'endormir

finsi /* fin d'atome ininterruptible */

ranger l'objet dans le tampon

/* début d'atome ininterruptible */

si consommateur endormi

alors réveiller le consommateur

sinon NPLEIN ++finsiFait

CONSOMMATEUR :

Faire toujours

si NPLEIN > 0 /* s'il existe au moins un

objet dans le tampon */

alors NPLEIN -- sinon s'endormir

finsi

prélever l'objet dans le tampon

si producteur endormi

alors réveiller le producteur

sinon NVIDE ++

finsi

consommer l'objetFait

2- Solution avec des sémaphores

On peut considérer NVIDE et NPLEIN comme des sémaphores :

PRODUCTEUR	**CONSOMMATEUR**
Faire toujours	Faire toujours
produire un objet	P (NPLEIN)
P (NVIDE)	prélever un objet
déposer un objet	V (NVIDE)
V (NPLEIN)	consommer l'objet
Fait	Fait

On démontre que le producteur et le consommateur ne peuvent être bloqués simultanément.

Cas où le nombre de producteur (consommateur) est supérieur à 1

Si plusieurs producteurs (consommateurs) opèrent sur le même tampon, il faut assurer l'exclusion mutuelle dans l' opération **déposer un objet** (**prélever un objet)** afin que le pointeur queue (tete) garde une valeur cohérente, de même que pour les objets pointés par queue (tete).

Si l'on veut s'assurer de plus qu'il n'y aura aucun problème dans l'accès au tampon, on peut décider que les opérations **prélever** et **déposer** ne s'exécutent pas simultanément. Déposer et prélever doivent donc figurer en section critique pour protéger les valeurs ressources (tampon, queue, tete). D'où l'utilisation d'un **sémaphore binaire :**

PRODUCTEURCONSOMMATEUR

```
produire un objet                    P  (NPLEIN)
P  (NVIDE)                       P  (MUTEX)
P  (MUTEX)                           objet = tampon [tete]
tampon[queue] = objet                    tete = (tete ++) % N
queue = (queue ++) % N               V  (MUTEX)
V (MUTEX)                        V  (NVIDE)
V (NPLEIN)                       consommer l'objet
```

3. Solution avec un compteur d'événements

D.P. REED et R.K. KANODIA ont proposé en 1979 une solution qui utilise une variable entière appelée compteur d'événements. Trois primitives permettent de manipuler une variable compteur d'événements E, commune à tous les processus concernés :

- Read (E) donne la valeur de E
- Advance (E) incrémente E de 1 de manière atomique
- Await (E , v) attend que E ≥ v

```
constante TAILLE   /* nombre de places dans le tampon */
compteur_d_événements          in = 0,     /* nb d'objets mis dans le tampon */
                               out  = 0   /* nb d'objets retirés du tampon    */
```

producteur

Faire toujours

produire l'objet suivant

nb_produits ++ /* nb_produits : nombre d'objets produits */

await (out, nb_produits - TAILLE)

mettre l'objet en pos(nb_produits - 1) % TAILLE

advance (in)

Fait

consommateur

Faire toujours

nb_retirés ++ /* nb_retirés : nombre d'objetsretirés */

await (in, nb_retirés)

retirer l'objet en position (nb_retirés - 1) % TAILLE

advance (out)

consommer l'objet

Fait

4-Solution avec moniteurs

Voici une solution du moniteur du problème producteur-consommateur :

```
        moniteur ProdCons                    /* moniteur, condition : types prédéfinis */
                condition plein, vide
                int compteur
                /* début du corps du moniteur */  compteur := 0
                /* fin du corps du moniteur
 procédure ajouter ()
{if compteur = N  then WAIT (plein) /* seul un SIGNAL (plein) réveillera le prcss */
                compteur ++
                if compteur = 1 then SIGNAL (vide)
                /* réveille un processus endormi parce que le tampon était vide */       }
```

```
procédure retirer ()
    {     if compteur = 0 then WAIT (vide) /* seul un SIGNAL (vide) réveillera le
                                                                    processus */
          ..... /* retirer l'objet du tampon */
                compteur --
                if compteur = N-1 then SIGNAL (plein)
                /* réveille un processus endormi parce que le tampon était plein */
          }
          fin du moniteur
```

```
procédure producteur ()
    {faire toujours
    produire (élément)
    ProdCons . ajouter ()
    fin faire}
```

```
procédure consommateur ()
{faire toujours
retirer (élément)
        ProdCons . retirer ()
        fin faire}
```

5- Solution avec échanges de messages

Certains ont estimé que les sémaphores sont de trop bas niveau et les moniteurs descriptibles dans un nombre trop restreint de langages. Ils ont proposé un mode de communication inter-processus qui repose sur deux primitives qui sont des appels système (à la différence des moniteurs) :

- send (destination , &message)
- receive (source , &message), où source peut prendre la valeur générale ANY

Généralement, pour éviter les problèmes dans les réseaux, le récepteur acquitte le message reçu. L'émetteur envoie à nouveau son message s'il ne reçoit pas d'acquittement. Le récepteur, s'il reçoit deux messages identiques, ne tient pas compte du second et en tire la conclusion que l'acquittement s'est perdu.

Dans le contexte d'un mode client-serveur, le message reçu contient le nom et les paramètres d'une procédure à lancer. Le processus appelant se bloque jusqu'à la fin de l'exécution de la procédure et le message en retour contient la liste des résultats de la procédure. On parle d'appel de procédure à distance.

On peut proposer une solution au problème producteur-consommateur par échange de messages avec les hypothèses suivantes :

- les messages ont tous la même taille
- les messages envoyés et pas encore reçus sont stockés par le SE dans une mémoire tampon
- le nombre maximal de messages est N

- chaque fois que le producteur veut délivrer un objet au consommateur, il prend un message vide, le remplit et l'envoie. Ainsi, le nombre total de messages dans le système reste constant dans le temps

- si le producteur travaille plus vite que le consommateur, tous les messages seront pleins et le producteur se bloquera dans l'attente d'un message vide ; si le consommateur travaille plus vite que le producteur, tous les messages seront vides et le consommateur se bloquera en attendant que le producteur en remplisse un.

producteur

```
Faire toujours
   produire_objet (&objet)
   /* produire un nouvel objet   */
   receive (consommateur , &m)
   /* attendre un message vide   */
   faire_message (&m , objet)
   /* construire un message à envoyer*/
   send (consommateur , &m)
   /* envoyer le message    */
Fait
```

consommateur

```
pour (i = 0 ; i < N ; i++) send (producteur,&m)    /* envoyer N messages vides   */
Faire toujours
   receive (producteur , &m)
   /* attendre un message            */
   retirer_objet (&m , &objet)
    /* retirer l'objet du message     */
   utiliser_objet (objet)
   send (producteur , &m)
```

/* renvoyer une réponse vide */Fait

On peut également imaginer une solution de type boîte aux lettres de capacité N messages ,avec un producteur se bloquant si la boîte est pleine et un consommateur se bloquant si la boîte est vide.

6- Propriétés des solutions précédentes

On démontre les résultats d'équivalence suivants entre sémaphores, moniteurs et échanges de messages :

- on peut utiliser des sémaphores pour construire un moniteur et un système d'échanges de messages

- on peut utiliser des moniteurs pour réaliser des sémaphores et un système d'échanges de messages

- on peut utiliser des messages pour réaliser des sémaphores et des moniteurs

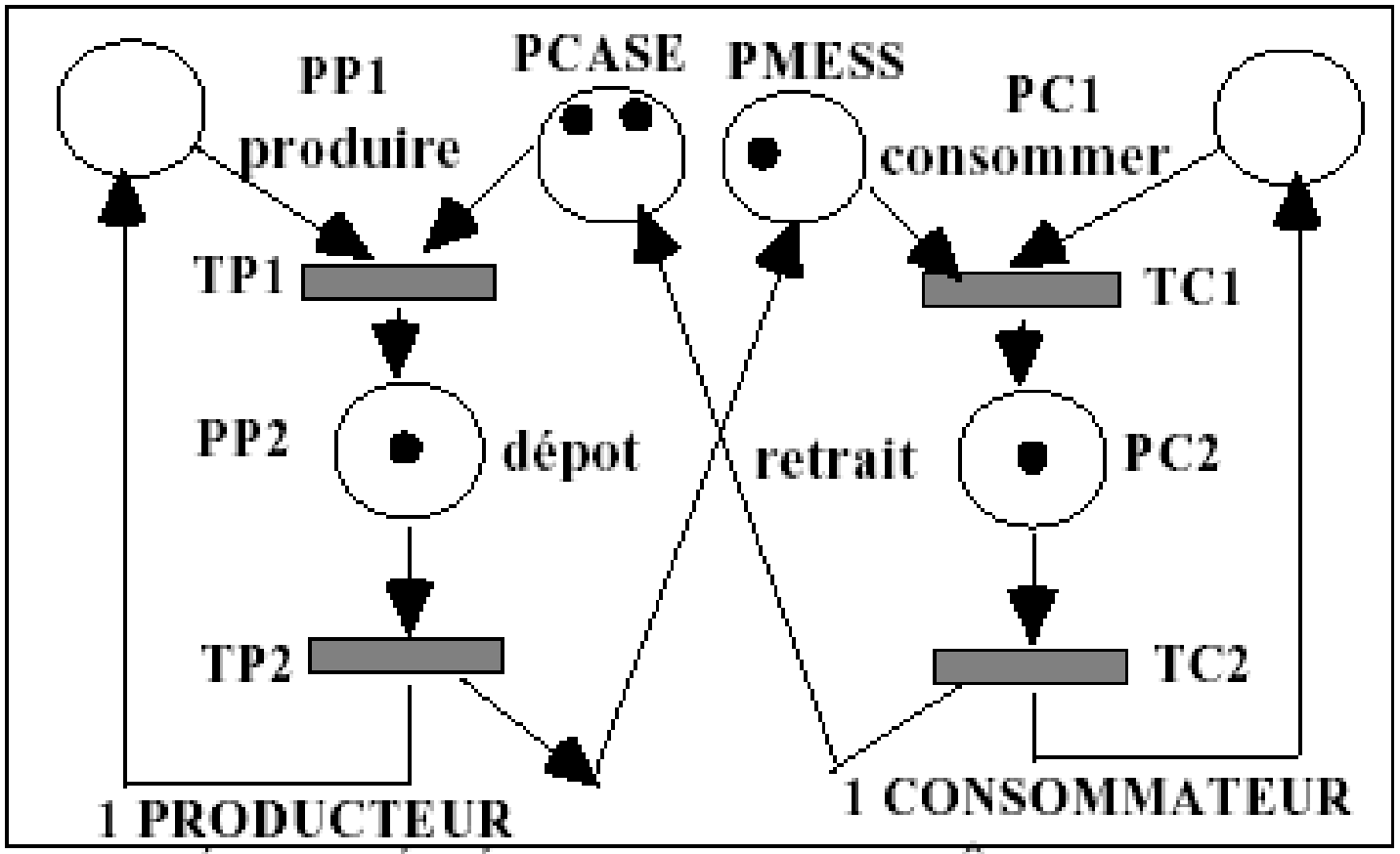


SCHÉMA GÉNÉRAL DE CONTRÔLE DE FLUX

Le problème du diner des philosophes

Introduction

Il s'agit d'un problème très ancien, dont DIJKSTRA a montré en 1965 qu'il modélise bien les processus en concurrence pour accéder à un nombre limité de ressources. *"5 philosophes sont assis autour d'une table ronde. Chaque philosophe a devant lui une assiette de spaghettis si glissants qu'il lui faut deux fourchettes pour les manger. Or, il n'y a qu'une fourchette entre deux assiettes consécutives. L'activité d'un philosophe est partagée entre manger et penser. Quand un philosophe a faim, il tente de prendre les deux fourchettes encadrant son assiette. S'il y parvient, il mange, puis il se remet à penser. Comment écrire un algorithme qui permette à chaque philosophe de ne jamais être bloqué ? "*

Il faut tout à la fois éviter les situations :

- d'interblocage : par exemple tous les philosophes prennent leur fourchette gauche en même temps et attendent que la fourchette droite se libère

- de privation : tous les philosophes prennent leur fourchette gauche, et, constatant que la droite n'est pas libre, la reposent, puis prennent la droite en même temps, etc...

Voici une solution (une autre est donnée dans Beauquier p. 155-156) :

Première solution

```
#define  N    5                    /* nombre de philosophes  */
```

```
#define  GAUCHE     (i - 1) % N   /* n° du voisin gauche de i */
#define  DROITE     (i + 1) % N   /* n° du voisin droite de     i */
#define PENSE   0                          /* il pense  */
#define FAIM       1                       /* il a faim   */
#define  MANGE 2                           /* il mange  */
typedef  int  semaphore;
int  etat [N];                                    /*  pour  mémoriser  les  états
des philosophes */
semaphore  mutex = 1,                  /* pour section critique */
           s [N];                      /* un sémaphore par philosophe */
/*****************************/
void philosophe (int  i)
```

```
{
    while (TRUE)
    {
        penser ();
        prendre_fourchettes (i);    /* prendre 2 fourchettes ou se bloquer */
        manger ();
        poser_fourchettes (i);
    }          }
```

```
void prendre_fourchettes (int  i)
{P (mutex);/* entrée en section critique */
        etat [i] = FAIM;
                test (i);
    /* tentative d'obtenir 2 fourchettes */
        V (mutex);/* sortie de la section crit.*/
```

```
P (s [i]);/* blocage si pas 2 fourchettes*/
   }
void poser_fourchettes (int i)
 {P (mutex);/* entrée en section crit.*/
        etat [i] = PENSE;
        test (GAUCHE);
        test (DROITE);
            V (mutex) ;
/* sortie de la section critique */}
void test (int  i)
{ if (etat [i] == FAIM && etat
[GAUCHE]!= MANGE && etat
[DROITE] != MANGE)
   {   etat [i] = MANGE;
              V (s [i] );
          }   }
```

Deuxieme solution

Baguette[5] :table de semapho=1

Chaise: semaphore=4

Philosophe(i:entier): processus

```
Philosophe (i:entier)
debut
//repas assis avec chaises pr 4 seulement
P(Chaise);
P(Baguette(I)); P(Baguette(I + 1)
Manger;
V(Baguette(I)); V(Baguette(I + 1));
V(Chaise);Fin
```

```
Parbegin
Philosophe(1),          Philosophe(2),
Philosophe(3),          Philosophe(4),
Philosophe(5)
parend

-- ni interblocage, ni famine
```

Le problème des lecteurs et des rédacteurs

Introduction :

Il s'agit d'un autre problème classique dont P.J. COURTOIS a montré en 1971 qu'il modélise bien les accès d'un processus à une base de données. On peut accepter que plusieurs processus lisent la base en même temps; mais dès qu'un processus écrit dans la base, aucun autre processus ne doit être autorisé à y accéder, en lecture ou en écriture.

Voici une solution qui donne la priorité aux lecteurs (COURTOIS a proposé une solution qui donne la priorité aux rédacteurs) :

```
typedef int  semaphore;
int  rc  = 0;          /* nb de processus qui lisent ou qui veulent écrire */
```

```
semaphore  mutex = 1,                  /* pour contrôler l'accès à rc     */
           bd = 1;                     /* contrôle de l'accès à la base */

/*******************************/
```

```
void lecteur ()                    | {P (mutex);/* pour un accès excl. à rc */
{while (TRUE)                      |      rc ++;    /* un lecteur de plus */
                                   |
```

```
if (rc == 1) P (bd);/* si c'est le 1er lect*/
V (mutex);
        lire_base_de_donnees ();
P (mutex);  /* pour un accès excl. à rc */
rc --;    /* un lecteur de moins */
if (rc == 0) V (bd);/* le dernier lect. */
V (mutex);utiliser_donnees_lues ();}}
        void redacteur ()
        {
```

```
        while (TRUE)
        {
            creer_donnees ();
            P (bd);
            ecrire_donnees ();
            V (bd);
        }
}
```

Autre bonne solution : Beauquier p. 146-148

Priorité totale aux lecteurs:

nl : entier / nl.=0 ;

Mutex1, Mutex2, w : semaphore [1]

```
Lecteurs ()
    Debut
      Faire tj
        P (mutex1);
                nl ++;
                si nl=1 alors P(w)
         V (mutex1);
        lire_base_de_donnees (bd);
        P (mutex1);
        nl --;       si nl=o alors V(w)
        V (mutex1);    faitFin
```

```
Redacteurs ()
      Debut
        Faire tj
         P(Mutex2)
              P(w)
                Ecrire_donnees (bd);
          V (w);
              V(Mutex2) ;
        Fait
    Fin
```

Chapitre 5 : Interblocage

Prévention et évitement de l'interblocage

1. **Introduction** :

Un système consiste en un nombre fini de ressources qui peuvent êtredistribuées aux processus en compétition.

Ressources : espaces mémoires, les cycles processeurs, fichiers , périphériquesd'entrées sorties..etc

SI un systèmepossède deux processeurs alors le type de ressource processeur a deux instances. (ils peuvent de eux ressources séparées).

Schema

P1→P2→P3→p1

R1 R2 R3

Les interblocages surviennent lorsque chaque processus d'un ensemble de processus contrôle une ressource requise par un autre processus de l ensemble. Chaque processus se bloque en attendant que la ressource requise soit disponible.

Exemple : problème d'interblocage sur le rover **Mars Pathfinder**

Mars Pathfinder est une sonde spatiale , développée par l'agence spatiale américaine, la NASA, qui s'est posée sur le sol de la planète Mars le 4 juillet 1997 .

Mais quelques jours plus tard, peu apres que Pathfinder a commencé à capter les informations météorologiques, le module spatial s'est mis à **se réinitialiser tout seul**, chaque reset entrainant une perte de données. !!

Trois tâches(threads) périodiques de priorités différentes :

– une tache météo de priorité basse, chargée de récolter les données météos et de les publier sur un bus de données

– une tâche de communication avec la NASA, de priorité moyenne.

– une tâche de gestion du bus de données, de priorité haute, qui transférait les données en entrée et en sortie du bus.

La tâche météo et la tâche de gestion du bus accédaient au bus en exclusion mutuelle grâceà un verrou mutex.

Un problème d'interblocage est survenu suite à un inversement de priorités.

2. Conditions d'interblocage :

Quatre conditions sont indispensables pour qu'un interblocage survienne.

C1. ***Exclusion mutuelle***. Chaque ressource peut êtrealloué a u seul processus a tout instant

C2. ***Détention et attente*** : le processus ne libèrent pas les ressources précédemment octroyées quand ils attendent que des requêtes imminentes soient octroyées.

C3. ***Pas de préemption*** : les ressources précédemmentoctroyées ne peuvent pas retirées des processus les détenant

C4. ***Attente circulaire*** : il existe une chaine de deux processus ou plus, de manièreà ce que chaque processus de la chaine contienne une ressource requise par le prochain processus de la chaine.

Interblocage ⇔c1^c2^c3^c4

Graphe d allocation de ressources :

Si un graphe d'allocation de ressources ne possède pas de cycle, alors le système n'est pas en état d'interblocage.

3. Prévenir l'interblocage :

Contraintes sur le modèle de programmation des processus.

Il est possible d'éviter l'interblocage à condition que les quatre conditions ne soient pas réunies.

1. La suppression de l'accès exclusif mutuel à toute ressource n'apparaît pas comme une solution pratique

- certaines ressources ne peuvent pas être partagées

  ex : imprimante (possible avec spooling)

2.La condition de détention et attente : s’applique en cas de requêtes de ressources par des processus sans ressources.

Sol1. On exige qu’un processus ne commence son exécution s’il ne requière pas toutes les ressourcespossibles.

Deux inconvénients majeurs :

- un processus ne sait pas forcement de quelles ressources il a besoin. Les ressources peuvent dépendre de son traitement
- famine

sol 2. demander aux processus de libérertoute ressource retenuelorsqu’ une requête est émise.

Inc. La plupart de temps, cette stratégie se révèle peu pratique.

Ex : allocation de disque puis imprimante. Risque de perdre ses données sur disque.

3.La suppression de la condition de non préemption : moins pratique que celle de la non exclusion. Possible pour quelques ressources (processeur) ou par dispositif virtuel.

4. L'élimination de la condition d'attente circulaire : la technique la plus prometteuse que les autres.

Une méthode consiste à associer à chaque ressource un numéro de priorité unique. Les processus ne peuvent requérir de ressources que si la priorité est plus élevée que toutes les ressources détenues.

Si la ressource n’a pas de prioritésupérieure à toutes les ressources détenues, celle d’entre elles qui est dotéepar la priorité la plus élevé doit êtrelibérée en premier.

Ex :

Imprimate 1, table tracante2 , modem 3

P1 imprimante table tracante

P2 modem imprimante

P3 table traçante modem

Inc. Il existe aucun numéro de priorité qui corresponde aux besoins de tous les processus.

3. Eviter les interblocages :

Analyse dynamique (à chaque demande de ressource) de l'état global d'allocation des ressources pour garantir une exécution sans interblocage.

Au lieu d'essayer d'éliminer l'une de conditions nécessairesà l'apparition de l'interblocage, il peut êtreévité ce dernier en n'allouant jamais de ressourcesi elleest susceptible d'induire à des interblocages. Une solution simple consiste allouer des ressources à un seul processus a la fois.

On dit un système est dans un état sur s'il ya une séquence d'exécution sure

Une séquenced’exécution sure est une séquence d’exécution dans laquelle tous les processus s’exécutententièrement.

Un état est dit sûr s'il existe une suite d'états ultérieurs qui permette à tous les processus d'obtenir toutes leurs ressources et de se terminer.

Sur / non sur (interblockage)

Relations entre les états sur, non sur et l’ineterblocage.

L’algorithme du banquier (Habermann et Dijkstra) est un algorithme d’évitement des interblocages qui s’applique dans le cas général où chaque type de ressources possède plusieurs instances. Le nom de l’algorithme a été choisi parce que cet

algorithme pourrait s’appliquer dans un système bancaire pour s’assurer que la banque ne prête jamais son argent disponible de telle sorte qu’elle ne puisse plus satisfaire tous ses clients.

Quand un nouveau processus entre dans le système, il doit déclarer le nombre maximal d’instances de chaque type de ressources dont il aura besoin. Ce nombre ne doit pas excéder le nombre total de ressources du système. Au cours de son exécution, quand un processus demande un ensemble de ressources, l’algorithme vérifie si cela gardera toujours le système dans un état sain. Dans l’affirmative la demande est accordée, dans la négative la demande est retardée.

L'algorithme de banquier ayant pour but de déterminer si un état est sur ou non d'un système avec P processus R ressources peut être récapitulé comme suit:

Algorithme du Banquier

Debut

max[P,R] :entier /* les besoins maximums des processus P en ressources R*/

courant [P,R] : Entier /* les allocations en cours*/

Disp[R]: entier;/* les disponibilités des ressources*/

Done[p]: booleen= "faux";

Sur: booleen;

Pp: entire:=0;

Tanque pp<P faire

Si done[pp] continue // passer au processus suivant

Sinon

// si les requêtes de ce processus sont accordées, peut il s'exécuter jusqu'à sa terminaison ?

Pour rr=1 jusuqu'à R faire

Si max[pp,rr]-courant [pp,rr]>Disp[rr] alors

Pp++ ;// non, pas suffisamment de ressources rr disponibles, se rend au prochain processus

Continue 2 // se rend à la prochaine itération de la boucle externe

Finsi

Fpour

// oui, réinitialiser le système à l'etat avec le nouveau etat des ressources en ajoutant les ressources libérées par pp

Pour rr=1 jusuqu'à R faire

Disp[rr]=disp[rr]+courant [pp,rr] ;

Fpour

Done[pp]= vrai ;

Pp=0 ;

Fsi

FTantque

// si tous les processus peuvent s’exécutercomplément , l’état du système est sur

Sur=vrai ;

Pour PP=1 jusuqu’à P faire

Si non Done[pp] alors sur =faux

FpourFin

L’algorithme du banquier prend la décision d’accorder une ressource en se demandant si l’octroi d’une requête va ou non placer le système dans un mode non sur.

Exemple :

Process	Allocations en	Besoins Maximums	Ressources

us	cours				disponibles	
	R1	R2	R1	R2	R1	R2
P1	1	1	4	1	1	0
P2	1	2	2	2		
P3	1	0	2	0		

Bien que trèsélégant dans sa conception, l'algorithme du banquier présente un inconvénient majeur : il lui faut connaître le nombre maximum de ressources que chaque processus peut requérir. Or, ce type de renseignements est rarement

disponible sur les systèmes : c’est pourquoi très peu de systèmes ont recours à cet algorithme.

Détection et correction de l'interblocage

Analyse dynamique et périodique de l'état d'allocation des ressources. Si détection d'interblocage à guérison.

5- Détecter les interblocages :

Plutôt que d'essayer de prévenir les interbocages , les systèmes peuvent également les laisser se produire (peu fréquent) pour les corriger lorsqu'ils surviennent. Une telle stratégie fait appel à des mécanismes de détection et de correction des interblocages.

Un graphe d'allocation des ressources peut être utilisé pour modéliser l'état des allocations de ressources et celui de requêtes.

Ex :

Processus	Allocations en cours			Requêtes émises			Ressources disponibles		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	3	0	0	0	0	0	0	0	0
P2	1	1	0	1	0	0			
P3	0	2	0	1	0	1			
P4	1	0	1	0	2	0			

Un graphe réduit d'allocation des ressources peut être utilisé pour déterminer si l'interblocage existe ou non. Pour y arriver, on suit les étapes suivantes d'une façon itérative :

*- si une ressource ne possède que des départs de flèches, il faut effacer toutes les flèches.

*- si un processus n' a que des arrivés des flèches, il faut également effacer toutes ses flèches.

*- si un processus a des flèches au départ, mais qu'un point de ressource (au moins) est disponible pour chacune d'elles dans la ressource dans laquelle pointe la flèche, il faut aussi supprimer toutes les flèches du processus.

Le système est en interblocage si et seulement s'il reste des flèches.

De l'état du système pour l'homme, main un ordinateur préfère une approche algorithmique. L'algorithme déterminant l'interblocage est similaire a celui qui détecte si le système est en état sur.

Algorithme

Debut

dem[P,R] :entier /* les besoins maximums des processus P en ressources R*/

courant [P,R] : Entier /* les allocations en cours*/

Disp[R]: entier;

Done[p]: booleen= "faux";

inetrblocage: booleen;

Pp: entire:=0;

Tanque pp<P faire

Si done[pp] contiune fsi

// si les requêtes de ce processus sont accordées , peut il s'exécuter jusqu'à sa terminaison ?

Pour rr=1 jusuqu'à R faire

si dem[pp,rr]>Disp[rr] alors

// non, pas suffisamment de ressources rr disponibles, se rend au prochain processus

Pp++ ;

Continue 2 // se rend à la prochaine itération de la boucle externe

Finsi

// oui, réinitialiser le système à l'etat avec le nouveau etat des ressources en ajoutant les ressources libérées par pp

Pour rr=1 jusuqu'à R faire

Disp[rr]=disp[rr]+courant [pp,rr] ;

Fpour

Done[pp]= vrai ;

Pp=0 ;

Fpour

// si tous les processus peuvent s'exécuter complément , l'état du système est sur interblocage=faux ;

Pour PP=1 jusuqu’à P faire

Si non Done[pp] alors interblocage =vrai

Fpour

Fin

6- Corriger les interblocages

Les trois approches permettant de corriger les interblocages sont la préemption automatique, la terminaison automatique ou l'intervention manuelle

a- Avec les préemption automatique, le système d'exploitation préempte un sous ensemble de ressources allouées. Trois problèmes majeurs doivent êtrerésolues pour mettre en œuvre une telle stratégie.

- Selection : quelles ressources de quels processus vont êtrepréemptées ?

  Des facteurs permettant de prendre la décision sont les suivantes :

- la priorité d'un processus ?
- Le temps d'exécution d'un processus

- Le nombre de ressources habituelles d'u processus
- Le nombre de processus affectés
  - Conséquence : qu'arrive-t-il aux processus qui ont leurs ressources préemptées ?

    Sauvegarde de la trace d'exécution : couteux(temps et stockage), peu pratique.
  - Famine : si le même processus est sans cesse victime de préemption.

    Solution : faire appel à un compteur de préemption qui est incrémenté chaque fois qu'un processus repasse à son étatantérieur.

b- La terminaison automatique fait disparaitre les interblocages en mettant fin au processus .

- Tous les processus ou sous ensemble
  - Tous ; une solution pour les systèmes qui favorisent la solution rapide.

- Sous ensemble : l
  - les problèmes de la sélection comme dans la préemptionautomatique.
  - La terminaison d'un processus peut répercuter sur les autres processus qui dépendent de son traitement ainsi la mise à jour partielle des fichiers.

c- L'intervention manuelle : c'est à l'operateur du système qu'il incombe de résoudre le problème.

- Compte tenu des limites de approches automatique, elle s'agit d'une solution intéressante

- Mais, certains systèmes fonctionnent sans operateur à plein temps, d'autres doivent réagir à une situation d'interblocage dès sa détection. Dans ces cas, l'intervention manuelle est peu commode

7- L’algorithme de l’autruche

Tous les mécanismes du traitement des interblocages présentés présentent tous un inconvénient majeur.(aucun moyen efficace). Pour la plupart des systèmes d’exploitation , l’apparition d’interblocages est fort rare. Par conséquent, dans nombreusessituations, le problème des interblocages est ignoré, à l’instar d’une autruche qui s’enfonce la tête dans le sable en espérant que le problème disparaisse.

Autrement, est ce que le cout de la solution est justifié par l’ampleur du problème à résoudre ?

BIBLIOGRAPHIE

J. BEAUQUIER, B. BERARD, Systèmes d'exploitation, Ediscience, 1993

M. BEN-ARI, Processus concurrents, Masson 1986

A. SCHIPER, Programmation concurrente, Presses Polytechnique Romandes 1986

A. TANENBAUM, Les systèmes d'exploitation, Prentice Hall, 1999

Crocus – Systèmes d'exploitation des ordinateurs.

Serie Schaum – Systèmes d'exploitation.