Chapitre 5 PROCESSUS ET RESSOURCES

1. GENERALITES

Un processus à exécuter a besoin de *ressources* : procédures et données, mémoire, processeur (UC , et d'autres éventuellement), périphériques, fichiers, ...

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 : 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: 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** .

2. LE PROBLEME DE L'EXCLUSION MUTUELLE

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

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. A ce sujet, aucune hypothèse ne doit être faite sur les vitesses relatives des processus**.

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.

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

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.

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.

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

Examinons quelques solutions au problème de l'exclusion mutuelle. On s'intéresse symboliquement ici au cas de deux processus P_1 et P_2 exécutant chacun une boucle infinie divisée en deux parties :

- la section critique ($crit_1$, $crit_2$ respectivement),
- le reste du programme ($reste_1$, $reste_2$ respectivement).

Les exécutions de $crit_1$ et de $crit_2$ ne doivent pas être interrompues et ne doivent pas se chevaucher.

Les algorithmes peuvent être classés en deux catégories :

- **algorithmes par attente active :**

  * 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 !

  * algorithmes 2.1 à 2.4

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

- **algorithmes par attente passive :** les sémaphores

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()
{
        for ( ; ; )
        {
                while (tour == 2);      /* c'est au tour de P2 ; P1 attend   */
                crit1;
                tour = 2;               /* on redonne l'autorisation à P2    */
                reste1;
        }
}
/*************************************************************************/
p2()
{
        for (; ; )
        {
                while (tour == 1);   /* c'est au tour de P1 ; P2 attend     */
                crit2;
                tour = 1;            /* on redonne l'autorisation à P1     */
                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 c_1, c_2 ;

/* clés de P_1 et P_2 - 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 section critique       */
                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 aussi entrer en section critique ... */
                {
                        c1 = 1;      /* ....P1 abandonne un temps son intention... */
                        c1 = 0;      /* ..... puis la réaffirme                          */
                 }
                crit1;
                c1 = 1;                /* fin de la section critique de P1               */
                 reste1 ;
        }
 }
/*************************************************************************/
 p2()
  {
         for ( ; ; )
        {
                c2 = 0 ;                /* P2 veut entrer en section critique            */
                while (c1 == 0)     /* tant que P1 veut aussi entrer en section critique ... */
                 {
                        c2 = 1;      /* .. P2 abandonne un temps son intention .... */
                        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 section critique        */
                while (c2 == 0)       /*  tant que P2 le veut aussi.....            */
                if (tour == 2)        /*  ..... si c'est le tour de P2 .....        */
                {
                        c1 = 1;       /*  ..... P1 renonce .....                    */
                        while (tour == 2) ;  /* ..... jusqu'à ce que ce soit son  tour ..... */
                        c1 = 0;              /*  ..... puis réaffirme son intention  */
                }
                crit1;
                tour = 2;             /*  C'est le tour de P2                       */
                c1 = 1;               /* P1 a achevé sa section critique            */
                 reste1;
        }
}
/*************************************************************************/
p2 ()
 {
         for (; ; )
        {
                c2 = 0;               /* P2 veut entrer en section critique         */
                 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'à ce que ce soit son 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.

- De manière générale, les algorithmes par attente active présentent un défaut commun : les boucles d'attente et le recours fréquent à la variable *tour* gaspillent du temps UC. 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.

En outre, l'utilisation par les algorithmes de variables globales n'est pas d'une grande élégance.

2.5 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)

2.5.1 P (s)

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

```
début
   a ← 1 /* a est un registre */
   TestAndSet (&a, &verrou)  /* copie le contenu de verrou (variable globale initialisée à 0) dans a
                et range 1 dans le mot mémoire verrou. TestAndSet est ininterruptible Exécutée par
le matériel, elle permet de lire et d'écrire un mot mémoire */
   tant que a = 1 /* si a = 0, le verrou est libre et on passe, sinon le verrou est mis */
         TestAndSet (&a, &verrou)
   fin tant que
   si  (valeur du sémaphore > 0)
         alors   décrémenter cette valeur
   sinon  - suspendre l'exécution du processus en cours qui a appelé P(s),
         - placer le processus dans la file d'attente du  sémaphore,
```

```
            - le processus passe de l'état ACTIF à l'état  ENDORMI.
        finsi
        verrou ← 0
    fin
```

2.5.2 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.

```
début
        a ← 1  /* a est un registre */
        TestAndSet (&a, &verrou)  /* cf. ci-dessus*/
        tant que a = 1 /* si a = 0, le verrou est libre et on passe, sinon le verrou est mis */
                TestAndSet (&a, &verrou)
        fin tant que
        si  (file d'attente non vide)
                alors
                        - choisir un processus dans la file d'attente du  sémaphore,
                        - réveiller ce processus. Il passe de l'état ENDORMI à  l'état ACTIVABLE.
        sinon    incrémenter la valeur du sémaphore si c'est possible
         finsi
        verrou ← 0
fin
```

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. Dans un système multiprocesseur, les accès à la variable partagée ***verrou*** peuvent s'entrelacer. Il est donc nécessaire de verrouiller le bus mémoire chaque fois qu'un processeur exécute TestAndSet.

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. L'attente active sur ***a*** consomme du temps UC.

2.5.3 Application des sémaphores à l'exclusion mutuelle

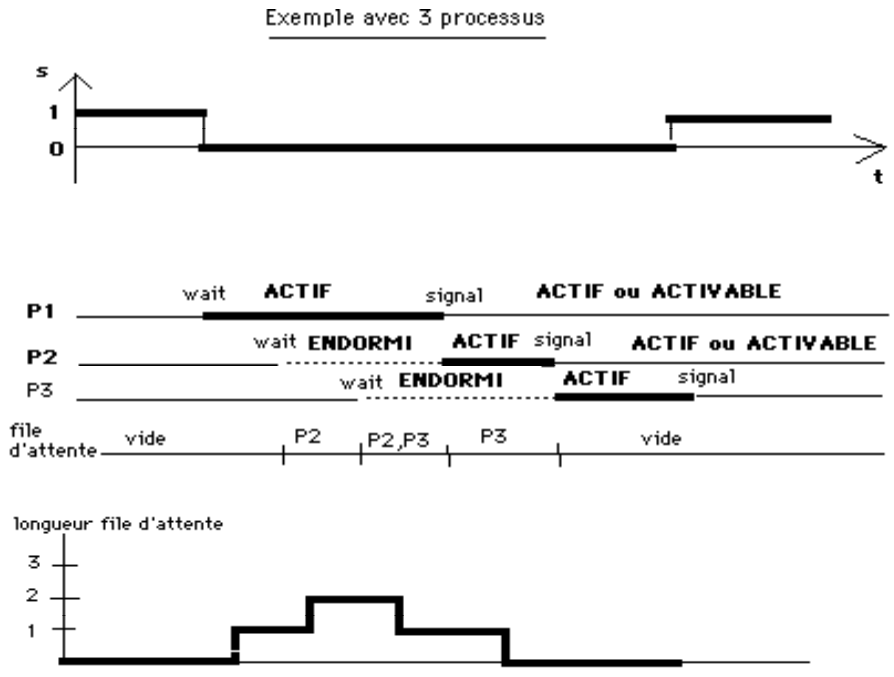
Avec le schéma de programme précédent :

```
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* .



3. COMMUNICATION INTER-PROCESSUS

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

```
Faire toujours
        produire un objet
        si nb d'objets ds tampon < N
                alors déposer l'objet ds le tampon
        finsi
Fait
```

CONSOMMATEUR

```
Faire toujours
        si nb d'objets ds tampon >0
                alors prendre l'objet
                        consommer l'objet
        finsi
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 au moins 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 ++
        finsi
Fait
```

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'objet
Fait
```

3.1 Solution avec des sémaphores

On peut considérer NVIDE et NPLEIN comme des sémaphores :

PRODUCTEUR

```
Faire toujours
        produire un objet
        P  (NVIDE)
        déposer un objet
        V (NPLEIN)
Fait
```

CONSOMMATEUR

```
Faire toujours
P  (NPLEIN)
prélever un objet
V  (NVIDE)
consommer l'objet
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 :**

PRODUCTEUR

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

CONSOMMATEUR

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

3.2 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 position (nb_produits - 1) % TAILLE
                advance (in)
        Fait


consommateur
        Faire toujours
                nb_retirés ++    /* nb_retirés : nombre d'objets retirés */
                await (in, nb_retirés)
                retirer l'objet en position (nb_retirés - 1) % TAILLE
                advance (out)
                consommer l'objet
        Fait
```

3.3 Solution avec un moniteur

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

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
- 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.

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 processus */
        ..... /* ranger l'objet dans le tampon */
        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
}
```

3.4 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.

3.5 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

3.6 Le problème des philosophes

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) :

```
#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 critique */
        P (s [i]);                              /* blocage si pas 2 fourchettes */
}
```

```
/*****************************/
void poser_fourchettes (int i)
{
        P (mutex);                                      /* entrée en section critique */
        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] );
        }
}
```

3.7 Le problème des lecteurs et des rédacteurs

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 ()
{
        while (TRUE)
        {
                P (mutex);                              /* pour obtenir un accès exclusif à rc */
                rc ++;                                  /* un lecteur de plus */
                if (rc == 1) P (bd);                    /* si c'est le 1er lecteur */
                V (mutex);
                lire_base_de_donnees ();
                P (mutex);                              /* pour obtenir un accès exclusif à rc */
                rc --;                                  /* un lecteur de moins */
                if (rc == 0) V (bd);                    /* si c'est le dernier lecteur */
                V (mutex);
                utiliser_donnees_lues ();
        }
}
/*****************************/
void redacteur ()
{
        while (TRUE)
        {
                creer_donnees ();
                P (bd);
                ecrire_donnees ();
                V (bd);
        }
```

}

Autre bonne solution : Beauquier p. 146-148

4. APPLICATION DES SEMAPHORES A LA SYNCHRONISATION

4.1 Synchronisation

Def.: On dit qu'un processus P est **synchronisé** avec un processus Q lorsque l'activité de P (resp. Q) dépend d'un événement modifié par Q (resp. P).

La synchronisation exige la mise en place d'un mécanisme permettant à un processus actif de bloquer un autre processus ou de se bloquer lui-même ou d'activer un autre processus. On distingue:

- les actions directes : le processus agit directement sur le processus cible par des primitives telles que **bloque** (pid) ou **réveill**e (pid) ou encore **dort** () qui s'applique à lui-même

- les actions indirectes : le processus utilise un objet commun, tel que sémaphore

Exemple :

Soient 3 processus P1, P2, P3 chargés du calcul de (a+ b) * (c + d) - (e/f)
P2 calcule c + d, P3 calcule e/f et P1 le résultat.
On initialise les sémaphores s1 et s2 à 0.

P1	P2	P3
t1 = a + b	t2 = c + d	t3 = e/f
• P (s1)	V (s1)	V(s2)
t4 = t1 * t2		
Δ P (s2)		
res = t4 - t3		

P1 ne peut se poursuivre au-delà de • tant que P2 n'a pas exécuté V
P1 ne peut se poursuivre au-delà de Δ tant que P3 n'a pas exécuté V

Plus généralement, si l'on veut empêcher un processus A d'exécuter son instruction K avant qu'un autre processus B n'ait exécuté son instruction J, on initialise un sémaphore binaire s à 0. On place P (s) avant l'instruction K de A et V (s) après l'instruction J de B.

4.2 Interblocage

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.

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)

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

Exercices sur le chapitre 4

1. Expliquer pourquoi l'exclusion mutuelle n'est plus assurée si les opérations P et V sur un sémaphore ne sont plus atomiques.

2. Une machine possède une instruction swap qui échange de manière atomique le contenu de deux variables entières. Soient les variables suivantes globales à n processus :

```
int libre = 0;
int tour [n];    /* tableau initialisé à 1 */
```

Le code source du processus n° i (i = 1 à n) est :

```
main ()
{
        while (1)
        {
                do {
                swap (libre, tour [i]);
                i++;
                } while (tour [i-1] == 1);
                /* puis section critique */
                ……
                swap (libre, tour [i]);
                /* puis section non critique */
        }
}
```

L'exclusion mutuelle est-elle satisfaite ?

Un processus désirant entrer en section critique peut-il attendre indéfiniment cette entrée ?