LDC, LDV pp 53
IRET : pp 118

CONCEPTION DES SYSTEMES CONCURRENTS

Tome 2

4ème Edition **Janvier 2007**

1ère édition. **6/12/1993**

Polycopié. N°: IAI 94-03

Pierre MOUKELI
Maître Assistant

pmoukeli@yahoo.fr

Institut Africain d'Informatique
IAI, B.P. 2263 Libreville
GABON

Chapitre 9. Exclusion mutuelle et synchronisation des processus.

L'exécution concurrente de processus sur une machine conduit inévitablement à des conflits de partage des ressources. En effet, celles-ci sont presque toujours en quantité limitée par rapport aux processus dont le nombre peut croître de manière exponentielle et stochastique. Les concepteurs de systèmes d'exploitation doivent donc mettre en oeuvre une politique d'arbitrage pour rationaliser l'exploitation des ressources, c'est à dire en réglementer l'occupation et la libération.

Dans ce chapitre, après avoir exposé les problèmes liés au partage des ressources et la notion de séquence critique, nous présentons quelques techniques de mise en œuvre de l'exclusion mutuelle (verrous et sémaphores) et un exemple de protocole. Enfin, nous présentons quelques cas d'utilisation des techniques d'exclusion pour la mise en œuvre de protocoles de synchronisations des processus.

IX.1. Notions de base de l'exclusion mutuelle.

IX.1.1. Problèmes liés au partage de ressources.

Considérons les deux séquences suivantes qui utilisent les variables x et y supposées être initialisées à 0.

seq1:	seq2 :
x = 15 ;	y = x + 3 ;
Ecrire (x) ;	Ecrire (y) ;

Supposons que les séquences *seq1* et *seq2* qui partagent la même variable x soient exécutées respectivement par deux processus P1 et P2. L'exécution du processus P1 donnera toujours le même résultat 15 : il est dit déterministe. L'exécution de P2 donnera soit 3, soit 18, et cela de manière imprévisible, selon que P1 s'exécutera le premier ou pas. Le comportement non-

déterministe de P2 résulte du partage de la variable x. Pour rendre ce comportement déterministe, il faut imposer un ordre d'accès à la variable x.

Supposons maintenant que x et y désignent un fichier F accessible en écriture pour P1 et en lecture pour P2, sachant que b et b' sont des buffers :

seq1 :	seq2 :
remplir (b) ; x = ouvrir (F, ECRITURE) ; Ecrire (x, b) ;	y = ouvrir (F, LECTURE) ; Lire (y, b') ; traiter (b') ;

Cette fois le comportement de P2 est totalement imprévisible. En effet, le fichier F peut être vide, en cours de remplissage, ou contenir d'anciennes données. Une coordination entre les deux processus P1 et P2 est nécessaire pour exploiter rationnellement le fichier F. Par exemple, P1 ne peut écrire que lorsque P2 n'est pas entrain de lire le fichier, et réciproquement. C'est ce qu'on appelle exclusion mutuelle (l'état d'écriture interdit, exclut l'état de lecture, et réciproquement). Un tel fichier est une ressource critique.

Plus généralement, une ressource critique est une ressource partagée dont l'utilisation nécessite un arbitrage. La notion de ressource généralise celle de variable.

IX.1.2. La séquence critique.

Une séquence (ou section) critique associée à une ressource est une séquence de code indivisible du point de vue de l'utilisation de cette ressource. Elle est constituée d'un point d'entrée, d'un code dit critique utilisant la ressource, et d'un point de sortie. L'entrée et la sortie servent à arbitrer (par exemple : séquentialiser) l'accès à la ressource. Le code critique sert à l'utilisation de la ressource ; un seul processus à la fois y est admis.

Soit r une ressource. La séquence critique relative à son utilisation est schématisée par :

```
Séquence_critique :
        Demander (r) ;      /* point d'entrée */
        utiliser (r) ;      /* code critique */
        libérer (r) ;       /* point de sortie */
```

C'est dans les fonctions points d'entrée et de sortie qu'est mise en oeuvre l'exclusion mutuelle. Toute utilisation de la ressource critique ne doit être envisagée que dans une séquence critique.

Le système d'exploitation doit fournir des outils permettant d'assurer l'exclusion mutuelle. C'est ensuite aux utilisateurs de s'appuyer sur ces outils pour mettre en oeuvre des politiques de partage des ressources. La solution apportée par le système doit remplir les conditions suivantes pour chaque ressource partagée :

1. **Atomicité de l'utilisation (ou exclusion mutuelle)** : à tout moment, un seul processus exécute le code critique associé à l'utilisation d'une ressource.

2. **Absence de blocage** : si aucun processus n'est engagé dans la section critique associée à une ressource, aucun processus ne doit être bloqué à cause de l'exclusion mutuelle sur cette ressource.

3. **Equité** : à priorité égale, tout processus doit pouvoir entrer en séquence critique et en sortir.

4. **Egalité** : le code d'entrée et de sortie de séquence critique doit être le même pour tous les processus pour une même ressource.

IX.2. Techniques de mise en oeuvre de l'exclusion mutuelle.

Il existe plusieurs techniques de mise en œuvre de l'exclusion mutuelle. Dans cette section présentons quelques unes, à savoir l'attente active avec l'algorithme de DEKKER et les verrous, et les sémaphores.

IX.2.1. L'attente active.

L'attente active consiste à associer un drapeau à chaque ressource partagée. Tout processus qui veut obtenir une ressource partagée doit tester le drapeau associé à celle-ci, pour s'assurer qu'elle est libre. Ensuite, il prend la ressource en positionnant le drapeau. Le processus libère le drapeau après l'utilisation de la ressource. L'obtention de la ressource nécessite souvent un temps d'attente pendant lequel le processus va continuellement tester le drapeau ; c'est ce qu'on appelle ***attente active***. Cette attente consomme naturellement le temps du processeur. De plus, elle pose le problème de l'atomicité des instructions. Nous ne nous étendrons pas sur ces problèmes du fait

que cette technique est très peu utilisée. Cependant l'aspect algorithmique mérite qu'on s'y attarde.

IX.2.1.1. Algorithme de Dekker.

La solution algorithmique de DEKKER est basée sur l'atomicité de l'accès mémoire, et utilise trois variables partagées par deux processus numérotés 0 et 1. Ces numéros sont en fait des indices d'un tableau. L'algorithme utilise trois fonctions :

- *Init* : permet d'initialiser les variables ;
- *Get* : point d'entrée de la séquence critique, permet d'attendre et prendre la ressource ;
- *Release* : point de sortie de la séquence critique, cette fonction permet de libérer la ressource..

Les fonctions *Get* et *Release* sont appelées par les processus 0 et 1, et jouent respectivement le rôle des fonctions *demander* et *libérer* du paragraphe IX.1.2. L'algorithme s'écrit :

```
int   flag [2] ;
char t ;

Init () { flag[0] = flag[1] = t = 0 ; }                    /* Initialise les drapeaux */

Get (int i) {                                              /* demande la ressource */
        flag [i] = 1 ;
        t        = 1 - i ;
        while ((flag[1-i] == 1) && (t == 1-i)) ;
        }

Release (int i) { flag[i] = 0 ; }                          /* libère la ressource */
```

L'efficacité de cet algorithme dépend de ce que l'opération d'accès mémoire doit être atomique. Pour s'en apercevoir, il suffit d'imaginer le processus interrompu dans l'affectation. A titre d'exercice, c'est algorithme peut-être facilement mis en œuvre est testé sous UNIX en utilisant

un segment de mémoire partagée, ou dans Linux avec les variables partagés et des processus légers..

Le lecteur étendra utilement cet algorithme aux cas de plus de deux processus.

IX.2.1.2. Le verrou.

La solution algorithmique devient très vite prohibitive dans le cas de plusieurs processus. La plupart des constructeurs fournissent sur leurs processeurs l'instruction assembleur TAS (Test and Set) faisant de la consultation et la modification d'un mot mémoire une instruction atomique. Un tel mot est appelé verrou. Par convention, le verrou vaut initialement 0. L'instruction TAS copie la valeur du verrou dans un registre R, et met le verrou à 1. Par exemple, si *v* est un verrou et R un registre, alors l'algorithme de TAS s'écrit :

```
TAS v :
        R = v ;
        v = 1 ;
```

Il faut noter que cette séquence de deux affectations est indivisible (atomique).

Un verrou est associé à chaque ressource. Le verrou peut être n'importe quelle zone mémoire. Il reflète par convention l'état de la ressource : si le verrou est trouvé à 0 alors la ressource est considérée comme étant libre; sinon, elle est occupée.

En utilisant l'instruction TAS, on peut écrire les fonctions d'entrée et de sortie de séquence critique, en supposant que le registre *R* (utilisée par TAS) est le registre de retour de valeur d'une fonction :

```
demander_ressource (VERROU v)  { while (TAS (v)) ; }
libérer_ressource (VERROU v)   { *v = 0 ; }
```

Dans le 80386, les instructions BTS (Bit Test and Set), BTR (Bit Test and Reset) et BTC (Bit Test and complément), permettent de mettre en oeuvre l'instruction TAS sur un bit. Chacune de ces instructions transfert le bit verrou dans le flag CF du registre EFLAGS, et met le bit verrou à 1, ou à 0 ou à son complément selon qu'on utilise BTS, BTR ou BTC. Il suffit alors de tester l'état de CF en utilisant au choix les instructions de saut conditionnel JC (CF égal à 1) ou JNC (CF égal à 0).

Exemple : Code assembleur 80386 mettant en oeuvre la fonction TAS.

```
        NAME exemple
        PUBLIC TAS
; --- Structure de la pile : elle contient l'adresse de retour et l'adresse du verrou
seg_pile    STRUC
retour      DD?
adr_verrou  DD?
seg_pile    ENDS
; --- Segment de code
seg_code    SEGMENT ER PUBLIC
TAS         PROC
            MOV EAX,[ESP].adr_verrou    ; EAX pointe sur le verrou
            XOR  EBX,EBX                ; indice du bit : 0
            BTS  [EAX],EBX              ; test and set du bit 0 du verrou
test:       JC etiq1                    ; si le bit dans CF vaut 1
            MOV EAX, 0                  ; retourne 0 dans EAX: verrou libre
            JMP eti2
etiq1:      MOV EAX, 1                  ; retourne 1 dans EAX: verrou occupé
etiq2:      RET 6                       ; retour au programme appelant, ajuste la pile
TAS         ENDP
seg_code    ENDS
            END.
```

IX.2.2. Les sémaphores.

L'exclusion mutuelle réalisée par attente active est très gourmande en temps d'unité centrale. Pour minimiser ce temps, les concepteurs système ont introduit la notion de sémaphore.

IX.2.2.1. Description d'un sémaphore.

Un sémaphore est un mécanisme système permettant de séquentialiser l'accès à une ressource. Il est constitué d'une file d'attente de type FIFO, d'un compteur, et de deux fonctions P et V représentant respectivement l'entrée et la sortie de séquence critique. Un sémaphore étant associé à une ressource, la fonction P sert à demander la ressource, mais si celle-ci n'est pas disponible, le processus demandeur se bloque dans la fonction P en attendent que la ressource se libère. Les processus ainsi bloqués sont insérés dans la file d'attente associée au sémaphore. Quant à la fonction V, elle sert à libérer la ressource en réveillant éventuellement le processus qui est en tête de la file d'attente, pour lui céder la ressource. Enfin, en ce qui concerne le compteur, quand sa valeur est positive ou nulle, il indique le nombre d'exemplaires de la ressource disponibles ; et si

sa valeur est négative, celle-ci indique en valeur absolue le nombre de processus bloqués en attente de la ressource.

En résumé, un sémaphore est caractérisé par les propriétés suivantes :

- La valeur initiale du compteur indique le nombre d'exemplaires de la ressource disponibles. En général ce nombre est 1 ;
- Si la valeur de son compteur est positive, alors elle indique le nombre d'exemplaires disponibles de la ressource; si elle est négative, alors elle désigne en valeur absolue le nombre de processus en attente.
- L'exécution de P est bloquante si la ressource n'est pas disponible.
- La fonction V n'est pas bloquante.

Soit *s* un sémaphore ; supposons qu'initialement sa file *s.f* est vide, et le compteur *s.c* contient le nombre d'exemplaires de la ressource disponibles. Les fonctions P et V s'écrivent de la façon suivante :

```
P (SEMAPHORE s) {
        PDESC       *ptr ;
        s.c     = s.c - 1 ;
        if (s.c < 0) {
                ptr = lire_desc () ;
                ptr->état = bloqué ;
                insérer (ptr, s.f)
                }
        }
```

```
V (SEMAPHORE s) {
        PDESC       *ptr ;
        s.c     = s.c + 1 ;
        if (s.c <= 0) {
                ptr  = retirer (s.f);
                ptr->état = actif ;
                insérer_ord (ptr) ;
                }
        }
```

Dans ces fonctions nous avons supposé que :

- *Insérer* : insère un processus dans la file d'attente du sémaphore en ayant sauvegardé son contexte, et appelle le superviseur pour libérer le processeur (commutation de contexte) ;
- *retirer* : retourne le descripteur du processus qui est en tête de la file du sémaphore ;
- *Lire_desc* : retourne le pointeur du descripteur du processus ;
- *insérer_ord* : insère un processus dans la file d'ordonnancement.

Le lecteur avisé constatera que si un processus est interrompu dans la première instruction de P (modification du compteur), alors l'exclusion mutuelle peut échouer si dans l'intervalle un nouveau processus exécute P. Pour y remédier, les fonctions P et V doivent être indivisibles (par exemple : exécution en mode superviseur, passage en haute priorité).

IX.2.2.2. Mise en oeuvre des sémaphores à l'aide de verrous.

Dans les algorithmes qui suivent, la mise en oeuvre des fonctions P et V s'appuie sur l'utilisation d'un verrou pour assurer l'exclusion mutuelle d'accès à la structure de données du sémaphore. Cette structure est constituée de variables du sémaphore et d'une file des processus en attente des ressources gérées par le sémaphore. Nous supposons qu'un processus prend un exemplaire d'une ressource à la fois; nous supposons aussi disposer des fonctions système suivantes :

- *InsererFifo* et *RetirerFifo* : permettent respectivement d'insérer et de retirer un élément d'une FIFO ;
- *TAS* : Test and Set, fonction (instruction) pour positionner un verrou ;
- *GetPdesc* : retourne le descripteur du processus qui l'appelle ;
- *Suspendre* et *Réveiller* : permettent respectivement de suspendre et de réveiller un processus de descripteur donné ;
- *ProcReschedule* : remet un processus en fin de file d'ordonnancement.

```
        /* Structure des variables d'un sémaphore */
struct SemStruct {
        int     SemCtr ;            /* Compteur */
        int     SemVal ;            /* Nombre de ressources */
        int     SemVerrou;          /* Verrou de partage des données du sémaphore */
        struct Pdesc* SemFifo ;
        } ;

        /* Création et initialisation d'un sémaphore pour n exemplaires de la ressource */
struct SemStruct *SemInit (int n) {
        struct SemStruct *ptr ;

        if (n <= 0)
                return (0) ;
        ptr     = (struct SemStruct*) malloc (sizeof(struct SemStruct)) ;
        if (!ptr)
                return (0) ;            /* Memory overflow */
        else {
                ptr->SemCtr             = n ;
                ptr->SemVal             = nb ;
                ptr-> SemVerrou          = 0 ;
                ptr->SemFifo            = 0 ;
                return (ptr) ;
                }
        }

int SemP (struct SemStruct *sem) {
        struct Pdesc    *ptr ;

        if (!sem)
                return (-1) ;
        while (TAS (sem-> SemVerrou))           /* Attente libération de la structure sem*/
                ProcReschedule () ;
        sem->SemCtr-- ;
        if (sem->SemCtr >= 0) {                 /* Une ressource est disponible */
                sem-> SemVerrou     = 0 ;       /* Libère la structure du sem */
                return (0) ;
                }
        else {                                  /* Ressource occupée */
                ptr     = GetPdesc () ;
                InsererFifo (sem->SemFifo, ptr) ;
                sem->SemVerrou      = 0 ;       /* Libère structure du sem */
                Suspendre (GetPdesc ()) ;
                return (0) ;
                }
        }

void SemV (struct SemStruct *sem) {
```

```
struct SemElem        *ptr ;

if (!sem)
        return ;
while (TAS (sem-> SemVerrou))           /* Attente libération de la structure sem*/
        ProcReschedule () ;
if (sem->SemCtr == sem->SemVal) {
        sem-> SemVerrou     = 0 ;       /* Libère la structure du sem */
        return  ;
        }
else {
        sem->SemCtr++ ;
        if (sem->SemCtr > 0) {          /* Pas de processus en attente
                sem-> SemVerrou     = 0 ;   /* Libère la structure du sem */
                return  ;
                }
        else {                          /* Reveil d'un processus en attente */
                ptr             = RetirerFifo (sem->SemFifo) ;
                sem-> SemVerrou     = 0 ;   /* Libère la structure du sem */
                Reveiller (ptr->Pdesc) ;
                return ;
                }
        }
}
```

Dans cet algorithme, la perte de temps occasionnée par l'attente active sur le verrou n'est pas très préjudiciable du fait qu'elle est couverte par la remise du processus en fin de file d'ordonnancement.

IX.2.2.3. Problèmes liés aux sémaphores.

L'utilisation des sémaphores assure bien l'exclusion mutuelle, mais elle peut poser des problèmes algorithmiques du fait que c'est l'utilisateur qui écrit le code critique. Les problèmes les plus courants sont l'attente infinie et l'interblocage :

- **Attente infinie**. Le processus détenant le sémaphore peut rester indéfiniment dans le code critique du fait d'une boucle infinie (défaut d'algorithme), ou se terminer prématurément (e.g. cas de déroutement). Dans les deux cas, le processus ne peut pas libérer la ressource, c'est à dire le sémaphore associé. Cela conduira les autres processus à attendre indéfiniment s'ils exécutent P sur le sémaphore en cause.

- **Interblocage**. L'interblocage est la situation stable dans laquelle plusieurs processus se trouvent indéfiniment en attente les uns des autres, chaque processus attendant qu'un autre libère la ressource dont il a besoin. Soient par exemple deux processus P1 et P2 qui utilisent deux ressources auxquelles sont associés les sémaphores s1 et s2. Considérons les séquences critiques suivantes :

Processus P1 :	Processus P2 :
... P(s1) ; ... P(s2) ; ... code critique ... V(s2) ; ... V(s1) ;	... P(s2) ; ... P(s1) ; ... code critique ... V(s1) ; ... V(s2) ;

Pendant l'exécution, l'un des deux processus peut se suspendre après avoir obtenu uniquement sa première ressource (premier appel de P). Le second processus peut alors entrer en séquence critique et obtenir lui aussi sa première ressource. Dans cette hypothèse, les deux processus seront bloqués indéfiniment, chacun attendant la deuxième ressource déjà occupée par l'autre. En effet, l'un ne peut être débloqué que par l'autre. Une telle situation est appelée interblocage, ou étreinte fatale (en anglais, *deadlock*).

En y regardant de plus près, on constate que chacune des séquences critiques précédentes est une imbrication de deux séquences critiques. Pour supprimer le risque d'interblocage, il suffit de supprimer la séquence critique la plus interne. Cela revient à considérer que les deux ressources ne font qu'une, et doivent être gérées par un seul sémaphore.

IX.3. Protocole d'exclusion mutuelle : les moniteurs.

Les problèmes liés aux sémaphores résultent pour l'essentiel de mauvais choix algorithmiques, car les séquences critiques sont écrites par les utilisateurs. Pour résoudre ces problèmes, les concepteurs système ont proposé de rassembler toutes les séquences critiques à l'intérieur d'un protocole d'utilisation des ressources appelé moniteur.

IX.3.1. Principe.

Un moniteur est constitué d'une file d'attente, de deux fonctions entrée et sortie, et d'un ensemble de variables de condition (ou conditions). Le moniteur n'est accessible qu'à l'aide des fonctions entrée et sortie. Les variables de condition sont exclusivement accessibles à l'intérieur du moniteur. Chaque condition correspond à une ressource; et on lui associée une file d'attente et un ensemble de fonctions internes au moniteur.

IX.3.2. Mise en oeuvre.

***Loi des moniteurs** : à tout moment, un seul processus peut être actif dans le moniteur.*

Quand un processus entre dans le moniteur, la ressource qu'il veut utiliser peut être occupée. Dans ce cas, il doit se suspendre en attendant la libération, et permettre à un nouveau processus d'entrer dans le moniteur. Quand un processus libère une ressource, avant de quitter le moniteur, il doit réveiller un processus demandeur de cette ressource. Mais alors, deux processus sont actifs dans le moniteur; ce qui viole la loi. Pour y remédier, nous considérerons que le processus réveillé est mis en fin de file d'ordonnancement, et que le processus qui l'a réveillé quitte tout de suite le moniteur.

Un moyen efficace d'implémenter un moniteur consiste à utiliser des sémaphores : un sémaphore pour filtrer l'accès au moniteur, un sémaphore par condition, et quatre fonctions de base :

```
entrer (moniteur m) :          /* seule point d'entrée dans le moniteur m */
        P(m) ;

sortie (moniteur m) :          /* seule point de sortie du moniteur m */
        V(m) ;

Attendre (ressource  r, moniteur m) /* pour d'attendre la ressource  r du moniteur m */
        si r.état == occupé alors
                V(m) ;
        P(r.sem)

signaler (ressource r)         /* permet de libérer une ressource r du moniteur m */
        si r.file == vide alors
                V(r.sem)
                V(m)
        sinon
                V(r.sem)
```

IX.4. Synchronisation avec les sémaphores.

La synchronisation est le mécanisme par lequel plusieurs processus s'accordent à exécuter une action générale en même temps. Lorsqu'elle n'implique que deux processus (ou quelque fois un petit groupe), on parle de rendez-vous ; sinon on parle de barrière de synchronisation. Dans la suite de cette partie, nous examinons chacun des deux scénarios et leur mise en œuvre avec des sémaphores.

IX.4.1. Rendez-vous.

IX.4.1.1. Description et mise en œuvre d'un rendez-vous.

Un rendez-vous un mécanisme par lequel deux processus doivent s'attendre mutuellement, de sorte que quand les deux ont atteint le point de rendez-vous, le système les réveille. Ce mécanisme peut être mis en œuvre avec deux sémaphores et une variable d'état, de la façon suivante

```
Sémaphore s, s' ;                 /* s pour l'exclusion mutuelle est initialisé à 1,
                                     et s' pour l'attente, initialisé à 0 */
Etat E = 0 ;
Rendez-vous ()
    |   P (s) ;                   /* exclusion mutuelle sur la variable d'état */
    |   E      = E + 1 ;          /* je suis au rendez-vous */
    |   Si (E == 1)               /* je suis le 1er arrivé */
    |       |   V (s) ;
    |       |   P (s') ;         /* j'attends l'autre */
    |   Sinon                     /* l'autre m'attendait déjà
    |       |   E      = 0 ;
    |       |   V (s) ;          /* libération de la variable d'état */
    |       |   V (s') ;         /* je réveille l'autre */
    |
```

Dans cette fonction, le 1er processus qui arrive se bloque sur le sémaphore s' dont le compteur est initialisé à 0, pour attendre l'autre. Le second arrivé réveille le premier qui est en attente au rendez-vous.

IX.4.1.2. Producteur - consommateur.

Le protocole producteur consommateur met en présence deux processus : l'un le producteur, qui produit une donnée, dont le deuxième processus, le consommateur a besoin. Le producteur génère la donnée et la rend disponible au consommateur ; puis il attend celui-ci. Le

consommateur à son tour se met en attente de la donnée. Dès qu'elle est disponible, il la récupère, ce qui libère le producteur. Ainsi donc, le premier des deux partenaires qui arrive le premier attend l'autre. Un tel protocole peut être envisagé entre un processus qui remplit un fichier, et un second qui le lit.

En utilisant le mécanisme de rendez-vous, il est possible de mettre en œuvre le protocole producteur – consommateur, de la façon suivante.

On suppose que la donnée est échangée entre le producteur et le consommateur à travers une variable partagée. La fonction produire appelée par le producteur génère la donnée, et la fonction consommer exécutée par le consommateur permet d'utiliser cette donnée. Ces fonctions sont supposées être fournies par l'utilisateur.

```
Etat E = 0 ;
Variable V ;
Produire () ;
Consommer () ;
Sémaphore s, s' ;          /* s :  sémaphore d'exclusion mutuelle dont le compteur est
                                   initialisé à 1 */
                           /* s' : sémaphore d'attente dont le compteur est initialisé à 0  */
```

```
Producteur ()
    P (s) ;                    /* exclusion mutuelle sur la variable d'état */
    E      = E + 1 ;           /* je suis au rendez-vous */
    V      = Produire () ;
    Si (E == 1)                /* je suis le 1er arrivé */
        V (s) ;
        P (s') ;               /* j'attends l'autre */
    Sinon
        E      = 0 ;
        V (s') ;               /* je réveille l'autre */
```

```
Consommateur ()
    P (s) ;                 /* exclusion mutuelle sur la variable d'état */
    E      = E + 1 ;        /* je suis au rendez-vous */
    Si (E == 1)             /* je suis le 1er arrivé */
        V (s) ;
        P (s') ;            /* j'attends l'autre */
        Consommer (V) ;
        V (s) ;
    Sinon
        E      = 0 ;
        Consommer (V) ;
        V (s) ;             /* libération de la variable d'état */
        V (s') ;            /* je réveille l'autre */
```

IX.4.1.3. Client - serveur.

Le protocole client - serveur met en présence un serveur auquel s'adressent des clients pour solliciter ses services. Le serveur reste en attente permanente d'une demande de service. Quand il en reçoit une, il exécute le service sollicité et acquitte le client demandeur. Quand un client arrive, il adresse une demande de service au serveur. Puis il se met en attente d'un acquittement le notifiant que le service a été rendu par le serveur. Plusieurs clients peuvent adresser leurs demandes simultanément ; elles sont alors traitées par le serveur en premier arrivé, premier servi.

Il y a là donc un mécanisme de rendez-vous entre le client qui attend l'acquittement et le serveur qui attend le client. Ce protocole peut être mis en œuvre en adaptant légèrement le protocole de rendez-vous de la section IX.4.1.1, comme décrit dans l'algorithme ci-dessous.

Le protocole ci-dessous est utilisable dans le cas où le serveur gère une ressource à laquelle il est le seul à accéder.

```
Variable E, E', X ;          /* E : Nombre de client ; E' : état du serveur ;
                                X : service*/
Sémaphore s1, s2, s3, s4 ;   /* s1 : sémaphore d'exclusion mutuelle,
                                initialisé à 1 */
                             /* s2 : sémaphore d'attente du
                                serveur quand il n'y pas de
                                client, initialisé à 0 */
                             /* s3 : sémaphore d'attente des clients,
                                initialisé à 0 */
                             /* s4 : sémaphore d'attente du service
                                demandé, initialisé à 0 */
Rendre_service (service x) ;
Demander_service (service x) ;
```

```
Serveur ()
    Eternellement :
        P (s1) ;             /* exclusion mutuelle sur la variable d'état */
        Si (E == 0)          /* il n'y a pas de client */
            E' = 1 ;
            V (s1) ;
            P (s2) ;         /* j'attends un client */
            P (s1) ;         /* exclusion mutuelle sur la variable d'état */
        E = E – 1 ;          /* je vais réveiller un client */
        V (s1) ;
        V (s3) ;             /* je réveille un client */
        P (s2) ;             /* j'attends le type de service du client */
        Rendre_service () ;  /* le type de service est dans X */
        V (s4) ;             /* je réveille un client qui demandé le service V' */
```

```
Client (service x)
    De temps en temps :
        P (s1) ;             /* exclusion mutuelle sur la variable d'état */
        E = E + 1 ;          /* je suis un client de plus */
        Si (E' == 1)         /* le serveur m'attendait, moi le premier client */
            E' = 0 ;
            V (s2) ;         /* je réveille le serveur */
        V (s1) ;
        P (s3) ;             /* j'attends mon tour d'être servi */
        Demander_service (x, X) ; /* je spécifie le service à demander dans X*/
        V (s2) ;             /* je réveille le serveur pour mon service*/
        P (s4) ;             /* j'attends la fin de mon service */
```

IX.4.2. Barrière de synchronisation.

Une barrière de synchronisation est un protocole permettant à un groupe ou clique de processus, de s'arrêter tous ensemble à un point de synchronisation : la barrière. Quand tous les processus sont arrêtés, la barrière s'ouvre et tous les processus sont réveillés.

Ce protocole peut être mis en œuvre en utilisant les sémaphores. Les processus s'arrêtent en appelant la fonction P sur un sémaphore dont le compteur est initialisé à 0. Tout le problème est de connaître quand tous les processus sont arrêtés. Trois techniques peuvent être utilisées :

- Tous les processus du système doivent être arrêtés. Il suffit que les fonctions de création et de terminaison des processus tiennent un décompte du nombre de processus actifs, et que celui-ci soit comparé au nombre de processus arrêté à un moment donné.

- Le nombre de processus de la clique est connu d'avance. Dans ce cas chaque processus qui s'arrête incrémente un compteur à comparer avec la taille de la clique. Il faut bien sûr que tous les processus de la clique soient actifs en même temps ; car si l'un manque à l'arrêt, la barrière ne s'ouvrira pas, à moins qu'il y ait un délai de garde.

- Chaque processus s'inscrit d'avance à la barrière. A l'arrêt, on contrôle l'identité de chaque processus, pour s'assurer que le nombre d'inscrits est atteint. Il faut là aussi que tous les processus de la clique soient actifs en même temps ; car si l'un manque à l'arrêt, la barrière ne s'ouvrira pas, à moins d'utiliser un délai de garde.

Supposons que la taille de la clique est connue. Nous allons utiliser une structure donnée comportant :

```
Structure barrière
    Entier      taille ;    /* taille de la clique * /
    Entier      ctr ;       /* nombre de processus arrêtés */
    Sémaphore   mutex ;     /* sémaphore d'exclusion mutuelle initialisé à 1 */
    Sémaphore   sem ;       /* sémaphore d'arrêt initialisé à 0 */
```

La fonction ci-dessous permet d'initialiser la structure de donnée relative à la barrière de synchronisation, qui comporte la taille de la clique, le compteur des processus arrêtés, le sémaphore d'exclusion mutuelle et le sémaphore d'arrêt.

```
Initialiser_barrière (taille t, barrière b)
    b.taille      = t ;
    b.ctr         = 0 ;
    b.mutex.ctr   = 1 ;
    b.sem.ctr     = 0 ;
```

La fonction ci-dessous est exécutée par tout processus de la clique pour s'arrêter à la barrière. Le dernier processus à atteindre la barrière réveille tous les autres. Il est possible d'intégrer une action à exécuter lorsque tous les processus sont arrêtés. Nous allons supposer que cette action est exécutée par le dernier processus à s'arrêter, avant qu'il ne réveille tous les autres.

```
Barrière_de_synchronisation (barrière b, action a)
    P (b.mutex) ;
    b.ctr    = b.ctr + 1 ;
    Si (b.ctr < b.taille)          /* quorum non atteint */
        V (b.mutex) ;
        P (b.sem) ;                /* je m'arrête */
    Sinon                          /* je suis le dernier et je réveille les autres */
        Tant que (b.ctr > 0)
            V (b.sem) ;
            b.ctr    = b.ctr – 1 ;
        V (b.mutex) ;
```

IX.5. Exercices et Travaux Pratiques.

IX.5.1. Producteur Consommateur.

Enoncé.

Une zone mémoire est générée sous la forme d'un tableau de tampons de même taille, utilisables entre des producteurs et des consommateurs. Le premier tampon est utilisé comme un tableau de flags indiquant l'état (libre ou occupé) des autres tampons; sachant que le flag n° i indique l'état du tampon n° i. Ces flags sont initialement à l'état libre. La zone mémoire est accessible en concurrence par les processus. Avant d'utiliser un tampon, un processus producteur doit le réserver. Un tampon ne peut être occupé que par un seul producteur à la fois, et ne peut être libéré que par celui-ci.

1) Ecrire une fonction permettant de réserver (occuper) un tampon.

2) Ecrire une fonction permettant de libérer un tampon.

Un tampon n'est partagé qu'entre un seul producteur et un seul consommateur.

3) Ecrire une fonction permettant d'inscrire une donnée dans un tampon donné.

4) Ecrire une fonction permettant de lire une donnée d'un tampon donné.

Solution.

Nous supposons disposer de deux sémaphores :

- SemT : initialisé à N-1 indique le nombre de tampons libres.
- Mutex: initialisé à 1 permet d'assurer l'exclusion mutuelle pour l'accès au tableau de flags.

Pour authentifier un tampon, le processus possesseur le signe en marquant le tableau des flags à l'entrée correspondante.

```
int reserver_tampon () {
        int i ;
        p (SemT) ;
        p(Mutex) ;
        i       = 1 ;
        while (Tampon[0)][i] != 0)
                i++ ;
        Tampon[0)][i]= getpid () ;
        v(Mutex)
        return (i) ;
        }

void libérer_tampon (int i) {
        p(Mutex) ;
        if (Tampon[0][i] == getpid())
                Tampon[0][i]  = 0 ;
        v(Mutex) ;
        v(SemT) ;               /* libère le tampon et réveille un demandeur */
        }
```

Nous supposons disposer d'un tableau de sémaphores SemP pour les producteurs, et SemC pour les consommateurs. Ces sémaphores sont initialisés à 0. SemP permet au producteur d'attendre le consommateur, et SemC permet au consommateur d'attendre le producteur.

```
void producteur (char* donnée) {
        int     i ;
```

```
        i       = reserver_tampon () ;
        copier (Tampon[i], donnée
        v (SemC[i]) ;                   /* réveille le consommateur */
        p(SemP[i]) ;                    /* attend la fin de la consommation */
        libérer_tampon (i) ;
        }

void consommateur (char* buf, int i) {
        p(SemC[i]) ;                    /* attend la présence de la donnée */
        copier(buf, Tampon[i]) ;
        v(SemP[i]) ;                    /* réveille de producteur pour libérer le tampon */
        }
```

Le lecteur vérifiera que les deux dernières répondent bien la condition qu'un seul producteur et un seul consommateur utilisent le même tampon. Le défaut de cette solution est que le consommateur peut attendre indéfiniment un hypothétique producteur; de même pour le producteur. Mais ce problème ne peut être résolu que dans le cadre d'un protocole bien précis.

IX.5.2. TP. Gestion du feu rouge de LONDON.

Le carrefour du quartier LONDON à LIBREVILLE est devenu dangereux pour les piétons et très encombré pour les automobilistes. Le Maire de la commune décide d'y implanter un feu tricolore pour synchroniser la traversée de ce carrefour. Mais au préalable, il demande de voir un modèle de ce feu rouge fonctionnant sur son ordinateur. Le carrefour est formé de la croisée de deux voix à double sens.

Eléments de résolution du problème.

- Le modèle sera simulé sous UNIX SYSTEM V en utilisant le bibliothèque IPC. Le carrefour pourra être matérialisé sur l'écran par une fenêtre gérée avec la bibliothèque *curses*. Les voitures y seront matérialisées par des jetons parcourant les voies; et les piétons, des jetons traversant les voies.

- Du fait que les feux doivent fonctionner indépendamment des voitures et piétons, le gestionnaire des feux peut être représenté par un automate d'état fini. La communication entre cet automate est les processus voitures et piétons pourra se faire à travers un segment de mémoire partagée dans lequel les piétons et les voitures lisent l'état du feu.

- On utilisera des sémaphores pour gérer l'accès à ce segment de mémoire.
- D'autres sémaphores serviront à mettre les processus en attente.
- Remarquer qu'avec la bibliothèque *curses*, un seul processus peut gérer l'écran. Il lui reviendra donc de servir de processus de visualisation de tout le système
- Les occurrences des voitures et des piétons obéissent à deux phénomènes stochastiques indépendants, auquel il serait intéressant d'intégrer la notion d'heure de pointe.

IX.5.3. TP. Algorithme de DEKKER sous UNIX.

Mettre en oeuvre l'algorithme de DEKKER en utilisant un segment de mémoire partagée dans le système d'exploitation UNIX.

Références bibliographiques.

BEAUQUIER Joffroy, BERARD Béatrice
Systèmes d'Exploitation : Concepts et Algorithmes
McGRAW-HILL

BEN-ARI M.
Processus Concurrents
MASSON

BOUZEFRANE Samia.
Les Systèmes d'Exploitation UNIX, LINUX, Windows XP avec C et Java.
DUNOD, SCIENCES SUP

DANCEA Ioan, MARCHAND Pierre
Architecture des Ordinateurs
Gaëtan MORIN, Editeur

DAUDEL Olivier
UNIX Système V, Configuration et modification du Noyau.
DUNOD TECH

DELACROIX Joëlle
LINUX, Programmation Système et Réseau.
DUNOD, SCIENCES SUP

FONTAINE A.B., HAMMES Ph.
UNIX système V, Système et environnement.
MASSON

GRIFFITHS Michael, VAYSSADE Michel
Architecture des Systèmes d'Exploitation
HERMES Traité des Nouvelle Technologies, série Informatique

MERCIER Philippe
Assembleur Facile, une Découverte Pas à Pas
MARABOUT Informatique

MOUKELI Pierre
Eléments de la Théorie des Langages
Support de cours IAI
BIBLIOTHEQUE IAI, BP 2263 Libreville –GABON

MOUKELI Pierre
Programmer avec LEX et YACC
Support de cours IAI
BIBLIOTHEQUE IAI, BP 2263 Libreville -GABON

MOUKELI Pierre
Système d'exploitation UNIX : Initiation et Administration
Support de cours IAI
BIBLIOTHEQUE IAI, BP 2263 Libreville -GABON

PATERSON David, HENNESSY John
Organisation et Conception des Ordinateurs, l'Interface Matériel / Logiciel
DUNOD

RAYNAL Michel, HELARY Jean-Michel
Synchronisation et Contrôle des Systèmes et des Programmes Répartis
EYROLLES

RIFFLET Jean-Marie
La programmation sous UNIX
McGRAW-HILL

RISTORI R., UNGARO L.
Architecture et Organisation des Systèmes
Polycopié IFSIC Juillet 1995, N° 68

SILBERSHATZ Abraham, GALVIN Peter B.
Principes des Systèmes d'Exploitation UNIX, 5ème édition.
ADDISON WESLEY

TANENBAUM Adrew
Les Systèmes d'Exploitation : Conception et Mise en Œuvre
InterEdition (Informatique Intelligence Artificielle)

THORIN M.
Parallélisme : Génie Logiciel Temps Réel.
DUNOD

TISCHER Michael, JENNRICH Bruno
La Bible PC Programmation Système
MICRO APPLICATION