Table des matières

Introduction

Un processus est une suite d'instructions réalisant une opération que l'on considère comme élémentaire relativement à son algorithme et à l'ensemble des ressources qu'elle nécessite. Dans un système l'on assiste régulièrement au déroulement successif sur un même processeur ou bien simultané sur des processeurs distincts des processus. Ces derniers utilisent parfois des ressources partagées et elles peuvent être attribuées, au même instant, à n processus au plus. Un ensemble de processus peut :

- Entrer en compétition pour l'accès à une ressource ou à une information partagée ;
- Fonctionner en coopération pour mener à bien une application.

Dès l'instant où deux processus ont une ressource en commun, il se pose un problème d'accès concurrent à la ressource critique. D'où la nécessité de synchroniser les différents processus du système afin que ceux-ci n'altèrent pas accidentellement la donnée critique. Pour résoudre ce problème de synchronisation, multiples solutions ont été proposés parmi lesquelles nous présentons quelque unes. Pour le faire nous avons organisé notre document comme suit: en section 1 nous définissons le problème de la section critique, en section 2 nous présentons la solution du masquage des interruptions, en section 3 nous avons l'exclusion mutuelle par attente active, par la suite l'exclusion mutuelle par attente non active est explicitée en section 4, la section 5 présente l'exclusion mutuelle (mutex) des threads et enfin nous avons une conclusion.

1) Le problème de la section critique

Une **section critique** est un ensemble d'instructions qui opèrent sur un ou plusieurs objets critiques et qui peuvent produire des résultats imprévisibles lorsqu'elles sont exécutées simultanément par des processus différents. Lorsqu'un processus manipule une donnée ou ressource partagée, on dit qu'il se trouve en section critique (SC) (associée à cette donnée). A un moment donné, on peut assister à la demande d'une ressource partagée par plusieurs processus ; en d'autres termes des demandes d'accès en section critique par plusieurs processus.

Le problème de la section critique est de trouver un algorithme d`exclusion mutuelle de processus dans l'exécution de leurs Sections Critiques afin que le résultat de leurs actions ne dépendent pas de l'ordre d'entrelacement de leur exécution (avec un ou plusieurs processeurs).

L'exécution des sections critiques doit être mutuellement exclusive : à tout instant, un seul processus peut exécuter une section critique pour une variable donnée (même lorsqu'il y a plusieurs processus). Ceci peut être obtenu en mettant à l'entrée et respectivement à la sortie de la section critique une fonction de demande d'entrée (entree_section()) en section critique et respectivement une fonction de notification de sortie (sortie_section()). Ce qui nous donne l'algorithme suivant :

```
repeat
        entree_section() ;
                section_critique
        sortie_section() ;
        section non critique
forever
```

Figure 1 : prototype d'une solution d'exclusion mutuelle

Critères nécessaires pour solutions valides :

- Exclusion mutuelle : à tout instant, au plus un processus peut être dans une section critique (SC) pour une variable donnée.
- Non inférence : Si un processus s'arrête dans sa section non critique, ceci ne devrait pas affecter les autres processus.
- Progrès :

- il y a absence d'interblocage ;
- si un processus demande l'accès à une section critique à un moment où aucun autre processus n'en fait la requête, il devrait être en mesure d'y entrer.

- Absence de famine : aucun processus ne sera éternellement empêché d'atteindre sa section critique.

Il existe de nombreux mécanismes pour mettre en œuvre l'exclusion mutuelle, dont chacun a ses avantages et inconvénients.

2) Masquage des interruptions

L'une des solutions simples au problème de synchronisation des processus est le masquage des interruptions. Cette méthodes consiste en : Lorsqu'un processus entre en section critique, il masque toutes les sources d'interruptions ; tout le temps qu'il met en section critique. Dès lors il n'y a pas de chance qu'un autre processus ait accès en Section Critique et le problème de synchronisation est en partie résolu. Cependant cette solution est dangereuse car imaginons-nous que pour une raison ou une autre le processus en SC lors de sa sortie ne restaure pas les interruptions ; alors ce serait la fin du système. En plus cette solution n'est valable que pour un système monoprocesseur ; car le masquage des interruptions n'est valable que pour le processeur qui l'a demandé et non pour tous les processeurs. Les autres processus exécutés par les autres processeurs pourront donc avoir accès aux objets partagés.

3) Exclusion mutuelle par attente active

3.1 Les variables de verrouillage

Une autre tentative pour assurer l'exclusion mutuelle est d'utiliser une variable de verrou partagé. Lorsqu'un processus veut entrer en section critique, elle lit la variable de verrou si elle est égale à 0, il modifie verrou ← 1 et exécute sa section critique ; une fois terminé, il modifie encore verrou ← 0. Sinon il attend (attente active) que le verrou devienne 0. Cette dernière est illustrée par l'algorithme ci-dessous.

```
While verrou != 0 do
                ; // attente active
        End while
        Verrou ← 1
        Section_critique() ;
        Verrou ← 0
```

Figure 2 Algorithme 1 Verrouillage

Problèmes :

Cet algorithme n'assure pas l'exclusion mutuelle. Considérons qu'un processus P1 veut entrer en section critique il lit la valeur de la variable verrou égale à 0. Puis il est interrompu et P2 est élu il teste la variable de verrou et voit qu'elle est à 0 il la modifie verrou ← 1 et passe en section critique ; ensuite il est interrompu sans être sorti de la section critique et P1 est à nouveau élu il modifie verrou ← 1 et entre en section critique. Les deux processus sont alors en même temps en section critique.

On constate donc que cet algorithme n'est pas correct du fait que la lecture et la modification de la variable de verrou ne constituent pas une instruction atomique. Si on possède de façon hardware une instruction **test and set** qui lit la valeur d'une variable, teste si elle est égale à 0 et la modifie à 1 tout ceci en une seule instruction, alors on pourra résoudre facilement le problème d'exclusion mutuelle en utilisant une variable de verrou. Toutefois on n'a pas toujours disponible une telle instruction donc il faut chercher une solution logicielle.

3.2 L'alternance

L'alternance est un autre moyen d'assurer l'exclusion mutuelle. Elle fonctionne avec une variable tour initialisée à 0 qui donne les droits d'accès à la ressource critique à un processus. Une fois le processus sorti de la section critique, il la modifie donnant ainsi accès à un autre processus. Soit l'exemple suivant :

Supposons qu'un processus P1 lit la variable tour qui vaut 0 et entre dans sa section critique. Ensuite P1 est interrompu et P2 exécuté ; il lit la valeur de la variable tour elle vaut 0, il attend dans une boucle à ce que la valeur de tour soit modifiée à 1. P2 est interrompu et P1 réactivé ; il achève sa section critique modifie la valeur de tour à 1 et exécute sa section non critique. P1 est à nouveau interrompu et P2 réactivé il lit la valeur de tour elle est égale à 1 il entre en section critique finit et positionne tour à 0. P2 exécute sa section non critique

souhaite encore entrer en section critique alors que tour vaut 0 il attend dans une boucle la modification de tour par P1.

```
// Processus P1
while(1)
{ // attente active
while(tour!=0) ;
Section_critique() ;
tour = 1 ;
Section_noncritique() ;
...
}
```

Figure 3 : Algorithme du processus P1

```
// Processus P2
while(1)
{ // attente active
while(tour!=1) ;
Section_critique() ;
tour = 0 ;
Section_noncritique() ;
...
}
```

Figure 4 : Algorithme du processus P2

Problème : Nous constatons facilement que nous ne pouvons pas assister à la présence de deux processus en section critique. Toutefois le problème n'est pas résolu car il peut arriver que l'on ait un processus présentant un besoin régulier d'entrer en section critique plus que l'autre. L'algorithme l'obligera à attendre son tour bien que la section critique ne soit pas utilisée. Un processus sera bloqué par un autre processus qui n'est pas en section critique.

3.3 Variables d'alternance de Peterson

La solution proposée par Peterson améliore l'approche ci-dessus, en permettant au processus dont ce n'est pas le tour de rentrer quand même en section critique, lorsque l'autre n'est pas encore prêt à le faire. Elle est basée sur deux procédures, entrer_region et quitter_region, qui encadrent la section critique telle décrites à la figure 5.

Si l'un des processus désire entrer et que l'autre n'est pas intéressé à enter, le premier processus entre immédiatement en section critique.

Tant que l'un des processus est en section critique, l'autre ne peut y entrer.

Si les deux processus sont en compétition pour entrer en section critique, le dernier à vouloir entrer écrase la valeur de la variable tour positionnée par le premier et n'entre donc pas en section critique. En revanche, le premier processus, qui pouvait boucler en attente si le deuxième venait de positionner sa variable intéressé entre en section critique.

```
int tour //variable de tour
int interessé [2] //drapeaux
entrer_region(P)
{
        intéressé [P] = VRAI ; // veut entrer
        tour = p ; // est passé en dernier
        while((tour == P)&&(intéressé [1-P]==VRAI))  //attendre son tour
}
quitter_region(P)
{
        interessé[P] = FAUX
}
```

Figure 5 : prototype de Peterson

3.4 Instruction TSL (Test and Set Lock)

Les architectures multiprocesseurs utilisent l'instruction TSL (Test and Set Lock) qui est une instruction atomique indivisible.

tsl registre, verrou

Cette instruction exécute en un seul cycle, de manière indivisible une opération de chargement d'un mot mémoire dans un registre et le rangement d'une valeur non nulle à l'adresse du mot chargé. Cette instruction apporte une correction à la solution des variables de verrouillages.

4) Exclusion mutuelle sans attente active

L'exclusion mutuelle par attente active consomme beaucoup de **temps processeur** (Le processus exécute une boucle infinie jusqu'à ce qu'il soit autorisé à entrer en section critique). Pour éviter cela, les systèmes d'exploitation disposent de primitives IPC (Inter Process Communication) qui se bloquent au lieu de perdre du temps UC, lorsqu'elles ne sont pas autorisées à entrer en section critique. Ces primitives sont :

- **Les primitives SLEEP et WAKEUP**
- **Les sémaphores**

- **Les compteurs d'événements**
- **Les moniteurs**
- **L'échange des messages**

4.1 Primitives SLEEP et WAKEUP

La primitive SLEEP () est un appel système qui suspend l'appelant en attendant qu'un autre le réveille. La primitive WAKEUP (processus) est un appel système qui réveille un processus. Par exemple, un processus A qui veut entrer dans sa section critique est bloqué si un processus B est déjà présent dans sa section critique.

Problème : Si les tests et les mises à jour des conditions d'entrée en section critique ne sont pas exécutés en exclusion mutuelle, des problèmes peuvent survenir. Si le signal émis par WAKEUP () arrive avant que destinataire ne soit endormi, le processus peut dormir pour toujours.

Exemples

```
// Processus A
{
        If (cond == 0) SLEEP () ;
        cond = 0 ;
        section_critique () ;
        cond = 1 ;
        WAKEUP (B) ;
        …
}
```

Figure 6: Algorithme processus A

```
// Processus B
{
        If (cond == 0) SLEEP ();
        cond = 0 ;
        section_critique ();
        cond = 1 ;
        WAKEUP (A) ;
        …
}
```

Figure 7: Algorithme processus B

Cas 1 : Supposons que :

- A est exécuté et qu'il est suspendu juste après avoir lu la valeur de cond qui est égale à 1.
- B est élu et exécuté. Il entre dans sa section critique et est suspendu avant de la quitter.
- A est ensuite exécuté. Il entre dans sa section critique.
- Les deux processus sont dans leurs sections critiques.

Cas 2 : Supposons que

- A est exécuté en premier. Il entre dans sa section critique. Il est suspendu avant de la quitter.
- B est élu. Il lit cond qui est égale à 0 et est suspendu avant de la tester.
- A est élu de nouveau. Il quitte la section critique, modifie la valeur de cond et envoie un signal au processus B
- Lorsque B est élu de nouveau, comme il n'est pas endormi, il ignorera le signal et va s'endormir pour toujours.

Application au problème du producteur/consommateur

Dans le problème du producteur et du consommateur deux processus partagent une mémoire tampon de taille fixe. L'un des processus, le producteur, met des informations dans la mémoire tampon, et l'autre, les retire. Le producteur peut produire uniquement si le tampon n'est pas plein. Le producteur doit être bloqué tant et aussi longtemps que le tampon est plein. Le consommateur peut retirer un objet du tampon uniquement s'il n'est pas vide. Le consommateur doit être bloqué tant et aussi longtemps que le tampon est vide. Les deux processus ne doivent pas accéder en même temps au tampon.

Une solution classique au problème du producteur et du consommateur avec les primitives SLEEP et WAKEUP est présentée ci-dessous :

```
#include "prototype.h "
#define N 100    /*nombre d'emplacements dans le tampon*/
Int compteur = 0 ; /*nombre d'objet dans le tampon*/
Void producteur (void)
{
        Int objet ;
        while (true)
        {
                Produire_objet (&objet) ;
                If (compteur == N)
                        SLEEP () ;  /*si tampon plein dormir*/
                mettre_objet (objet) ; /*mettre l'objet dans le tampon*/
                compteur = compteur + 1 ; /*incrémenté le nombre d'objet*/
                if (compteur == 1) /*réveiller le consommateur si besoin*/
                        WAKEUP (consommateur) ;
        }
}
```

Figure 8 : prototype producteur.c

```
#include “prototype.h “
#define N 100    /*nombre d’emplacements dans le tampon*/
Int compteur = 0 ; /*nombre d’objet dans le tampon*/
Void consommateur (void)
{
        Int objet ;
        While (true)
        {
                If (compteur == 0)
                        SLEEP () ;  /*si tampon vide dormir*/
                retirer_objet (objet) ; /*retirer l’objet dans le tampon*/
                compteur = compteur - 1 ; /*décrémenter le nombre d’objet*/
                if (compteur == N-1) /*réveiller le producteur si besoin*/
                        WAKEUP (producteur) ;
                consommer_objet (objet) ; /* utiliser l’objet */
        }
}
```

Figure 9 : prototype consommateur.c

Cette solution ne marche pas car le signal WAKEUP envoyé à un processus qui ne dort pas est perdu:

- La mémoire tampon est vide et le consommateur vient de trouver la valeur 0 dans le compteur.
- L’ordonnanceur décide à cet instant de suspendre le consommateur avant qu’il n’ait eu le temps de s’endormir.
- Le producteur est alors relancé :
    - Il met un objet dans le tampon.
    - Il incrémente le compteur.
    - Il constate qu’il vaut désormais 1 et réveille donc le consommateur.
    - Le signal WAKEUP est perdu car le consommateur ne dormait pas.
- Le consommateur est relancé par l’ordonnanceur :
    - Il testera la valeur du compteur qu’il avait lu avant d’être suspendu.
    - Il trouvera 0 et se mettra en sommeil.
- Le producteur finira par remplir la mémoire et ira lui aussi dormir. Les 2 processus dormiront pour toujours.

4.2 Les Sémaphores

Un sémaphore est un compteur entier qui désigne le nombre d'autorisations d'accès à une section critique. Il a donc un nom et une valeur initiale, par exemple **sémaphore s = 10.**

Les sémaphores sont manipulés au moyen des opérations **p** ou **wait** pour l'acquisition et **v** ou **signal** pour la libération. L'opération **p(s)** décrémente la valeur du sémaphore **s** si cette dernière est supérieur à 0. Sinon le processus appelant est mis en attente. Le test du sémaphore, le changement de sa valeur et la mise en attente éventuelle sont effectués en une seule opération atomique indivisible. L'opération **v(s)** incrémente la valeur du sémaphore **s** si aucun processus n'est bloqué par l'opération **p(s).** Sinon l'un d'entre eux sera choisi et redeviendra prêt. **v** est aussi une opération indivisible.

Application au problème du producteur/consommateur.

```
#define N 100                       /*Taille de la file */
semaphore mutex ;                           /*sémaphore d'exclusion mutuelle */
semaphore vide ;                    /*Nombre de places vides*/
semaphore plein ;                           /*Nombre de places occupées*/
sem_init (mutex, 1) ;                       /*Un seul processus en section critique*/
sem_init (vide, N) ;                        /*La file est toute vide*/
sem_init (plein, 0) ;                       /*Aucun emplacement occupé*/
producteur () {
        while (1) {
                produire_objet (o) ;        /*Produire un objet*/
                sem_p (vide) ;              /*on veut un emplacement vide*/
                sem_p (mutex) ;  /*on bloque la file*/
                mettre_objet (o) ;/*mettre l'objet en file*/
                sem_v (mutex) ;  /*libération de la file*/
                sem_v (plein) ;    /*un objet est à prendre*/
        }
}
```

Figure 10 : prototype producteur.c

```
consommateur () {
        while (1) {
                sem_p (plein) ;              /*attente d'un objet*/
                sem_p (mutex) ;              /*on bloque la file*/
                retirer_objet (o) ;          /*consommer l'objet courant*/
                sem_v (mutex) ;              / *libération de la file*/
                sem_v (vide) ;                       /*une place est à prendre*/
                consommer_objet (o) ;        /*consommer l'objet courant*/
        }
}
```

Figure 11 : prototype consommateur.c

L'utilisation des sémaphores, bien qu'amenant une simplification conceptuelle de la gestion des exclusions mutuelles, est encore fastidieuse et propice aux erreurs. Ainsi, si l'on inverse par mégarde les deux p (wait ou down) dans le code du consommateur, et si la file est vide, le consommateur se bloque sur le sémaphore plein après avoir positionné le sémaphore mutex. Le producteur se bloque sur son appel à mutex sans pouvoir débloquer le sémaphore plein, réalisant ainsi un interblocage.

4.3Compteurs d'événements

Un compteur d'événements est un compteur lié à l'occurrence d'un évènement. Il peut être utilisé pour synchroniser les processus. Par exemple, pour réaliser un traitement, un processus doit attendre jusqu'à ce que le compteur d'événements atteigne une certaine valeur. Cette solution définit trois opérations pour un compteur d'événements E.

Read(E) : donne la valeur de E

Advance(E) : incrémente E de 1 de manière atomique

Await(E, v) : attend que E atteigne ou dépasse la valeur v.

Application au problème du producteur/consommation

Pour résoudre ce problème, nous considérons deux compteurs **in** et **out** permettant respectivement de compter le nombre d'insertion dans le tampon et le nombre de retrait du tampon depuis le début.

in : est incrémenté de 1 par le producteur après chaque insertion dans le tampon.

out est incrémenté de 1 par le consommateur après chaque retrait du tampon.

Nous avons les conditions suivantes :

Production : le producteur doit arrêter de produire lorsque le tampon est plein la condition de produire est N > in – out ; avec N la taille du tampon. Posons sequence = in + 1. La condition devient alors out > = sequence – N.

Consommation : Le consommateur ne peut consommer que si le tampon n'est pas vide ce qui inplique in – out > 0. Posons sequence = out + 1. La condition devient alors in > = sequence. Ceci nous permet de sortir les algorithmes suivants :

```
// producteur consommateur
#define N 100 // taille du tampon
int in = 0, out = 0 ;
void producteur(void)
{
        int objet, sequence =0, pos =0 ;
        while(1)
        {
                objet= objet+1 ;
                sequence = sequence + 1 ;
                // attendre que le tampon devienne non plein
                Await(out,sequence-N) ;
                tampon[pos]= objet;
                pos = (pos+1)%N ;
                // incrémenter le nombre de dépôts
                Advance(&int) ;
        }
}
```

Figure 12 : Prototype algorithme du producteur

```
void consommateur (void)
{
        int objet, sequence =0, pos =0 ;
        while(1)
        {
                sequence = sequence + 1 ;
                // attendre que le tampon devienne non vide
                Await(in,sequence) ;
                objet= tampon[pos] ;
                // incrémenter le compteur de retraits
                Advance(&out) ;
                printf(" objetconsommé [%d]:%d ", pos, objet) ;
                pos = (pos+1)%N ;
        }
}
```

Figure 13 : Prototype algorithme du consommateur

4.4Moniteurs

L'idée des moniteurs est de regrouper dans un module spécial, appelé **moniteur**, toutes les sections critiques d'un même problème. Un moniteur est un ensemble de procédures, de variables et de structures de données.

Les processus peuvent appeler les procédures du moniteur mais ils ne peuvent pas accéder à la structure interne du moniteur à partir des procédures externes. Pour assurer l'exclusion mutuelle, il suffit qu'il y ait à tout moment au plus un processus actif dans le moniteur. Cette tâche est à la charge du compilateur. Pour ce faire, il rajoute, au début de chaque procédure du moniteur un code qui réalise ce qui suit : s'il y a un processus actif dans le moniteur, alors le processus appelant est suspendu jusqu' à ce que le processus actif dans le moniteur se bloque en exécutant un **wait** sur une variable conditionnelle (**wait (c)**) ou quitte le moniteur. Le processus bloqué est réveillé par un autre processus en lui envoyant un signal sur la variable conditionnelle (**signal (c)**).

Cette solution est plus simple que les sémaphores et les compteurs d'événements, puisque le programmeur n'a pas à se préoccuper de contrôler les accès aux sections critiques.

Application au problème du producteur/consommateur

Les sections critiques du problème du producteur et du consommateur sont les opérations de dépôt et de retrait dans le tampon partagé. Le dépôt est possible si le tampon n'est pas plein. Le retrait est possible si le tampon n'est pas vide.

```
Moniteur ProducteurConsommateur
{
// Variable conditionnelle pour non plein et non vide
        bool nplein, nvide ;
        int compteur = 0, ic = 0, ip = 0 ;
// Section critique pour le dépôt
        void mettre (int objet )
        {
        If (compteur == N) wait (nplien) ;
        // Attendre jusqu' à ce que le tampon devienne non plein
        //déposer un objet dans le tampon
        tampon [ip] = objet ;
        ip = (ip +1)%N ;
        compteur++ ;
        // si le tampon était vide avant le dépôt, envoyer un signal
        // pour réveiller le consommateur
        if (compteur == 1) signal (nvide) ;
        }
        // section critique pour le retrait
        void retirer (int* objet)
        {
        If (compteur == 0) wait (nvide) ;
        Objet = tampon [ic] ;
        Compteur-- ;
        If (compteur == N-1) signal (nplien) ;
        }
}
```

Figure 14 : Prototype Moniteur ProducteurConsommateur

4.5Echanges de messages

Le problème de synchronisation sur une ressource partagée peut être reformulé en termes de communication interprocessus, et implémenté sous la forme de deux primitives (appels systèmes) **send** et **receive.** La primitive send envoie un message à un processus donné, alors que la primitive receive permet à un processus de lire un message provenant d'un processus, donné ou quelconque (au choix du récepteur) ; si aucun message n'est disponible, receive bloque le processus appelant.

Cette solution, qui permet l'échange d'informations entre processus d'une architecture à mémoire distribuée, est beaucoup plus complexe à mettre en œuvre que les précédentes, car elle nécessite de résoudre plusieurs problèmes majeurs :

- La perte possible de messages ;
- Le nommage des processus ;
- L'authentification des messages ;
- Le surcout dû à la latence des communications ;
- La tolérance aux pannes.

La perte possible de messages se traite en utilisant un protocole d'acquittement des messages émis : tout message reçu doit être suivi de l'envoi d'un message d'acquittement informant l'émetteur que son message a été reçu et compris. Pour éviter les doublons résultant de la perte d'un acquittement, les messages sont numérotés, afin qu'un récepteur recevant plusieurs fois le même message puisse ignorer les messages retransmis.

Le nommage des processus est un problème difficile. Chaque processus peut facilement disposer d'un nom unique sur chaque machine, attribué par le système d'exploitation (de type PID), mais offrir un service de nommage global permettant la communication entre processus non apparentés est un réel problème. Tout d'abord, cela nécessite un espace de nommage commun à l'ensemble des processus, avec tous les problèmes de goulot d'étranglement posés par l'accès à une ressource critique. De plus il est théoriquement impossible d'empêcher deux processus différents de vouloir porter le même nom, et donc d'empêcher les conflits entre applications différentes (ou instances de la même application) si par malheur leurs processus se déclarent avec des noms identiques ; seule la détection de telle conflits est possible.

L'authentification des messages peut être réalisée au moyen de crypto-systèmes à clés publiques et privées, mais cela nécessite ici encore un goulot d'étranglement.

La réalisation d'un système de type producteur-consommateur par envois de messages peut se réaliser assez simplement si l'on suppose que le système assure lui-même le stockage temporaire des messages en attente de lecture.

```
#define N 100
producteur () {
        while (1) {
                produire_objet (o) ;
                recevoir (consommateur, " ") ;
                envoyer (consommateur, " ") ;
        }
}
consommateur () {
        for (i = 0 ; i < N-1 ; i++)
                envoyer (producteur, " ");
        while (1) {
                envoyer (producteur, " ");
                recevoir (producteur, " ");
                consommer_objet (o);
        }
}
```

Figure 15 : Prototype producteur-consommateur

5) Exclusion mutuelle de threads

La cohérence des données ou des ressources partagées entre les processus légers est maintenue par des mécanismes de synchronisation. Il existe deux principaux mécanismes de synchronisation : mutex et variable condition.

5.1 Mutex

Un mutex (verrou d'exclusion mutuelle) a deux états : verrouillé ou non verrouillé. Quand le mutex n'est pas verrouillé, un fil d'exécution peut le verrouiller pour entrer dans une section critique de son code. En fin de section critique, le mutex est déverrouillé par le fil l'ayant verrouillé. Si un fil essaye de verrouiller un mutex déjà verrouillé ; il est soit bloqué jusqu'à ce que le mutex soit déverrouillé, soit l'appel au mécanisme signale l'indisponibilité du mutex.

Trois opérations sont associées à un mutex : lock pour verrouiller le mutex, unlock pour le déverrouiller et trylock équivalent à lock, mais qui en cas d'échec ne bloque pas le processus.

Un mutex est souvent considéré comme un sémaphore binaire (sémaphore ne pouvant qu'avoir la valeur 0 ou 1).

Un sémaphore est un compteur d'accès à une ressource critique. L'utilisation d'un sémaphore s'effectue par des opérations de demande et de libération de la ressource critique.

Par rapport à un mutex, les opérations utilisées pour un sémaphore peuvent être réalisées par un autre fil d'exécution (un fil n'est pas propriétaire d'un sémaphore quand il réalise des opérations sur celui-ci).

5.2 Variables conditions

Les variables conditions permettent de suspendre un fil d'exécution tant que des données partagées n'ont pas atteint un certain état. Une variable condition est souvent utilisée avec un mutex. Cette combinaison permet de protéger l'accès aux données partagées (et donc à la condition variable). Un fil bloqué en attente sur une variable condition est réveillé par un autre fil qui a terminé de modifier les données partagées.

Il existe également une attente limitée permettant de signaler automatiquement la variable condition si le réveil ne s'effectue pas pendant une durée déterminée. Deux opérations sont disponibles : **wait**, qui bloque le processus léger tant que la condition est fausse et **signal** qui prévient les processus bloqués que la condition est vraie.

Conclusion

Au terme de ce travail dont il était question de présenter les différentes techniques de synchronisation des processus pour résoudre le problème d'entrée en section des processus, il en ressort que, il existe plusieurs solutions de synchronisation de processus parmi lesquelles l'exclusion mutuelle par attente active et l'exclusion mutuelle par attente passive. Il est à noter que l'exclusion par attente active consomme assez de ressources processeur lors de l'attente de l'accès à la section critique. L'exclusion mutuelle par attente passive en revanche met les processus désirant accéder en section critique en état bloqué si cette dernière est déjà occupée.

Bibliographie

[1] Abraham Silberschatz, Peter Bear Galvin, Greg Gagne, Operating System concepts, John Wiley & Sons, 2013.

[2] Christophe BLAEES, programmation système en C sous Linux, Eyrolles, 2000.

[3] J. Gispert, J. Guizol, J.L. Massat, Support de cours système d'exploitation, Département d'informatique Faculté de Luminy, disponible sur http://jean-luc.massat.perso.luminy.univ-amu.fr/ens/systeme/poly.pdf

[4] Max Hailperin, Operating Systems and Middleware supporting controlled Interaction, free re-release, 2005-2010.

[5] MEGZARI Omar, Chap5 : Synchronisation des processus, Faculté des Sciences Rabat, Université Mohamed V, disponible sur http://www.fsr.ac.ma/cours/informatique/megzari/ch5-synchronisation.pdf.

[6] Patrick Cegielski, Conception des systèmes d'exploitation Le cas Linux, Eyrolles, 2nd Edition, 2004.

[7] Thomas Anderson and Michael Dahlin, Operating Systems: principles and practice, Beta Edition, T.Anderson and M.Dahlin.