5 - SEMAPHORES

1. RAPPELS

Une **ressource critique** est une ressource partagée entre plusieurs processus et dont l'accès doit être contrôlé pour préserver la cohérence de son contenu (variable, fichier, etc…). En particulier, ce contrôle peut impliquer un accès en **exclusion mutuelle**, c'est-à-dire que la ressource n'est jamais accédée par plus d'un processus à la fois.

Une **section critique** est une séquence d'instructions au cours de laquelle un processus accède à une ressource critique et qui doit être exécutée de manière indivisible.

L'**indivisibilité** signifie que deux sections critiques concernant une même ressource ne sont jamais exécutées simultanément mais toujours séquentiellement (dans un ordre arbitraire). Il y a ainsi exclusivité entre ces séquences d'instructions.

L'exclusion mutuelle n'est qu'un cas particulier de synchronisation. Une **synchronisation** est une opération influant sur l'avancement d'un ensemble de processus:

- établissement d'un ordre d'occurrence pour certaines opérations (envoyer / recevoir).
- respect d'une condition (rendez-vous, exclusion mutuelle, etc…).

L'attente active est la mobilisation d'un processeur pour l'exécution répétitive d'une primitive de synchronisation jusqu'à ce que la condition de synchronisation permette la continuation normale du processus.

Un sémaphore est un objet permettant de contrôler l'accès à une ressource en évitant l'attente active. Il est constitué d'un compteur et d'une file d'attente. La valeur du compteur dénombre, lorsqu'elle est positive, le nombre de ressources disponibles, lorsqu'elle est négative le nombre de processus en attente de ressource. La politique de gestion de la file d'attente est définie par le concepteur du système. A la création d'un sémaphore, la file est vide. Un sémaphore est manipulé à l'aide des opérations indivisibles suivantes :

`SEM *CS(cpt)` : Création d'un sémaphore dont le compteur est initialisé à "cpt".

`DS(sem)` : Destruction d'un sémaphore. Si la file n'est pas vide un traitement d'erreur doit être effectué.

`P(sem)` : Demande d'acquisition d'une ressource. Si aucune ressource n'est disponible, le processus est bloqué.

`V(sem)` : Libération d'une ressource. Si la file d'attente n'est pas vide, un processus est débloqué.

1.1.

Donnez le code des primitives P et V en utilisant les opérations suivantes :

- TACH **courant*() : désigne la tâche en cours d'exécution (la tâche ELUE).
- void *insérer*(TACH tache, FILE *file) : insère une tâche dans la file
- TACH **extraire*(FILE *file) : extrait une tâche de la file.

```
      P(SEM *sem)                          V(SEM *sem)
      {                { TACH *tache;
(a)   sem->cpt--;                  (x)   sem->cpt++;
(b)   if (sem->cpt < 0) {          (y)   if (sem->cpt <= 0) {
(c)   courant()->état = BLOQUE;    (z)   tache = extraire(sem->file);
(d)   insérer (courant(), sem->file); (t) tache->état = PRET;
      commut()}                    }
      }                }
```

1.2.

Expliquez pourquoi ces primitives doivent être rendues indivisibles et comment on peut réaliser cette indivisibilité.

Supposons que la valeur initiale du compteur soit 1. Si deux processus exécutent la séquence d'instructions (a1 a2 b2 b1), il y a deux processus bloqués mais le compteur est à -1. L'un des deux ne sera donc jamais débloqué. Pire, si le sémaphore est utilisé pour gérer une exclusion mutuelle, tout processus qui tentera d'accéder à la ressource sera bloqué définitivement puisque personne n'exécutera jamais le V.

La solution pour rendre P et V indivisibles est de masquer les interruptions pendant leur exécution. Ceci n'est autorisé que parce que P et V sont des fonctions *système*.

1.3.

Pourquoi n'utilise-t-on pas le masquage/démasquage des interruptions pour réaliser une section critique en mode utilisateur ?

C'est trop dangereux ! Un processus risque d'en profiter pour monopoliser le processeur, puisqu'une fois qu'il a masqué les interruptions, il s'exécutera jusqu'au démasquage. S'il n'effectue pas ce démasquage, le système ne peut pas reprendre la main.

1.4.

Un processus en train d'exécuter une section critique peut-il être interrompu par un autre processus ? Expliquez ce qui se passe alors.

Oui bien sûr. Si le nouveau processus élu n'accède pas à la ressource critique, il peut s'exécuter normalement. S'il accède à la ressource, il se trouvera bloqué à l'entrée de la section critique et perdra le processeur au profit d'un autre processus.

Si la section critique était réalisée au moyen du masquage des interruptions, les tâches qui n'accèdent pas à la ressource critique seraient pénalisées puisqu'elles devraient attendre la fin de l'exécution de la section critique pour accéder au processeur (masquage de l'interruption horloge).

On considère les trois processus suivants qui s'exécutent de manière concurrente, et le sémaphore Mutex initialisé à 1.

Processus A	*Processus B*	*Processus C*
`(a) P(Mutex);`	`(d) P(Mutex);`	`(g) P(Mutex);`
`(b) x = x + 1;`	`(e) x = x * 2;`	`(h) x = x – 4;`
`(c) V(Mutex);`	`(f) V(Mutex);`	`(i) V(Mutex);`

1.5.

On considère le scénario suivant : a d b g c e f h i

Donnez après chaque opération sur le sémaphore Mutex

- la valeur du compteur,
- le contenu de la file du sémaphore,
- l'état de chacun des processus (élu, prêt, bloqué).

Le scénario suivant est-il possible ? Justifiez : a d e b c f g h i

NB : j'ai volontairement omis les modifications de l'état des processus dues à une élection pour qu'il n'y ait pas de confusion avec celles dues au sémaphore.

Opération	Compteur	Etat de la file	Etat des processus
a	Mutex.cpt = 0		A : élu B, C : prêt
d	Mutex.cpt = -1	B	B : élu -> bloqué A, C : prêt
g	Mutex.cpt = -2	B C	C : élu -> bloqué A : prêt B : bloqué
c	Mutex.cpt = -1	C	A : élu B : bloqué -> prêt C : bloqué
f	Mutex.cpt = 0		B : élu C : bloqué -> prêt
i	Mutex.cpt = 1		C : élu

Le deuxième scénario est impossible : lorsque B exécute (d), il se trouve bloqué et seule l'exécution de (c) par A lui permettra de poursuivre. Il ne peut donc exécuter (e) immédiatement.

On considère les trois processus suivants qui s'exécutent de manière concurrente sur une machine. La variable **a** est une variable partagée, et on a les initialisations suivantes :

int a = 6; S1 = CS(1); S2 = CS(0);

Processus A	*Processus B*	*Processus C*
`P(S1);`	`a = a – 5;`	`V(S1);`
`a = a + 7;`		`P(S2);`
`V(S2);`		`a = a * 3;`

1.6.

Quelles sont les valeurs finales possibles de la variable a ? Donner pour chaque valeur la (les) suite(s) d'instruction qui amènent à cette valeur.

ABC -> 24
ACB -> 34
BAC -> 24
NB : le sémaphore S1 est ici parfaitement inutile.

1.7.

- Ajoutez dans les programmes des processus les sémaphores nécessaires (et ceux-là seulement), pour obtenir dans toute exécution a = 34. Donnez les initialisations des sémaphores ajoutés.
- Même question pour a = 24.

Une seule séquence permet d'obtenir a = 34. Il suffit d'imposer que B ait lieu après C. On ajoute un sémaphore S3 initialisé à 0 :

```
Processus A      Processus B      Processus C
P(S1);           P(S3);           V(S1);
a = a + 7;       a = a - 5;       P(S2);
V(S2);           a = a * 3;
                 V(S3);
```

Les deux séquences qui permettent d'obtenir a = 24 sont caractérisées par le fait que C a lieu en dernier. On ajoute un sémaphore S3 initialisé à 0 (ou on réutilise S2) :

```
Processus A      Processus B      Processus C
P(S1);           a = a - 5;       V(S1);
a = a + 7;       V(S3);           P(S2);
V(S2);           P(S3);
                 a = a * 3;
```

1.8.

Quel sémaphore pourrait être supprimé sans modifier l'exécution de l'ensemble des processus ? Justifiez.

Le sémaphore S1 n'est jamais bloquant. Il est donc inutile.

1.9.

On modifie la synchronisation de la manière suivante :

```
Processus A      Processus B      Processus C
P(S1);           P(S2);           P(S2);
P(S2);           a = a - 5;       P(S1);
a = a + 7;       V(S2);           a = a * 3;
V(S1);           V(S1);           V(S2);
V(S2);
```

S1 et S2 sont **tous les deux initialisés à 1**. Quelles sont les conséquences de cette modification ? Justifiez.

Certaines séquences d'exécution peuvent mener à un interblocage.

Par exemple :

A : P(S1) C : P(S2) B : P(S2) -> bloqué A : P(S2) -> bloqué C : P(S1) -> bloqué

2. SYNCHRONISATION ET INTERBLOCAGE

Soient N processus utilisateurs U_1 à U_N partageant un ensemble de X serveurs et Y ressources (X > 0, Y > 0, N > X + Y). Ces processus sont synchronisés au moyen de 2 sémaphores S et R, représentant la disponibilité des serveurs et des ressources, selon le programme suivant.

```
Boucle
     Prologue:  P(S);  P(R);  V(S);
     Utilisation de ressource;
     Epilogue:  P(S);  V(R);  V(S);
Fin boucle
```

Le sémaphore S est initialisé à X et le sémaphore R est initialisé à Y.

2.1.

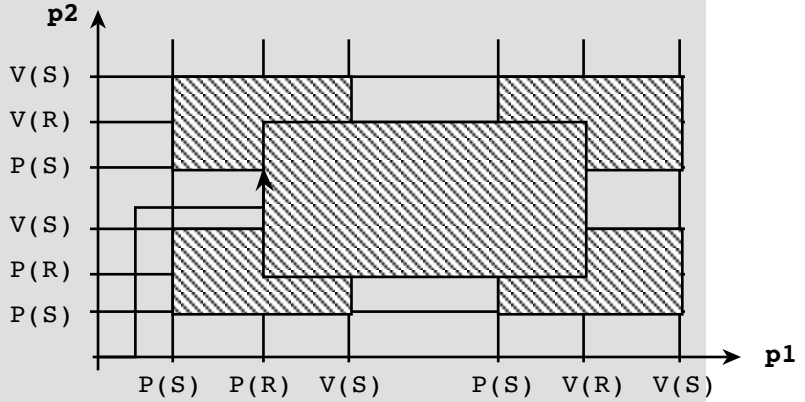
Quel est le nombre maximum n de processus pouvant se trouver simultanément en train d'utiliser une ressource ?

Dans tous les cas, ce nombre ne peut pas être supérieur au nombre de ressources : n ≤ Y. Le nombre de serveurs n'intervient pas : en effet, un processus qui prend une ressource libère le serveur avant l'utilisation de la ressource : avec un unique serveur, on peut tout de même avoir Y processus utilisant des ressources en même temps. Donc n = Y

2.2.

Montrez, pour N = 2 et X = Y = 1 que l'exécution de ce système peut conduire à un interblocage. Pour X > 0, Y > 0 quelconques, quelle est la valeur minimum de N qui peut amener à un interblocage ?

L'exécution des deux processus peut être représentée sur un graphe (graphe de progrès), où les contraintes définies par les sémaphores interdisent l'accès à certaines zones. Une exécution est représentée par une ligne brisée représentant la progression des deux processus dans leur exécution. Lorsqu'on entre dans certaines zones du graphe, on ne peut plus en sortir et il y a donc interblocage. La figure ci-dessous représente un cas de ce type.



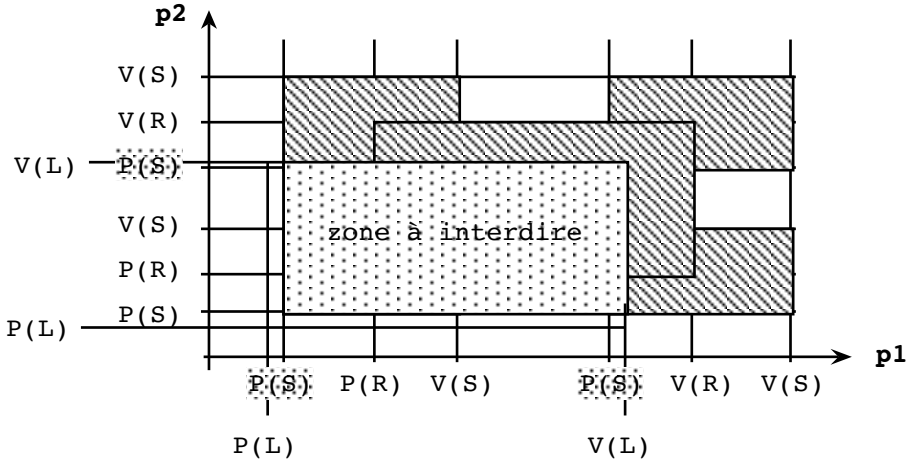
Ces états de blocage sont caractérisés par le fait que la ressource est occupée, et le serveur n'est pas disponible pour la libérer. Si l'on généralise à X et Y quelconques, il faut que les Y ressources soient occupées, et les X serveurs bloqués par des processus qui ne possèdent pas de ressource. Il faut donc un minimum de N = X + Y processus, exécutant la séquence suivante :

Y * Prologue + X * Prologue_P(S) + Epilogue_P(S)

2.3.

L'interblocage peut être évité en introduisant un nouveau sémaphore L, pour limiter le nombre de processus autorisés à entrer dans le prologue. Introduisez les opérations P(L) et V(L) nécessaires dans le prologue et l'épilogue. Précisez l'initialisation du sémaphore L.

Pour éviter l'interblocage, il faut interdire lez zones à risque. Dans le cas de deux processus, la zone à éviter est représentée ci-dessous. On la bloque avec un sémaphore L initialisé à 1, tel que P(L) est placé dans le prologue avant P(S) et V(L) dans l'épiloque après P(S). Autrement dit, lorsque la ressource est occupée, on empêche l'autre processus d'utiliser le serveur.



Dans le cas général, lorsque les Y ressources sont occupées, on doit conserver un serveur disponible pour permettre la libération, donc ne pas autoriser plus de (X-1) nouveaux processus à exécuter le prologue. L'initialisation du compteur est donc de (X+Y-1).