FACULTÉ
DES SCIENCES
Unité de formation
et de recherche

UNIVERSITE
BRETAGNE
LOIRE

Système

Fabien GARREAU
sur la base du cours d'André Rossi

Université d'Angers

`fabien.garreau@univ-angers.fr`

L2 MPCIE, 2018–2019

Communication interprocessus

La communication interprocessus peut se faire par l'intermédiaire de fichiers partagés, mais des incohérences peuvent survenir, par exemple quand 2 processus écrivent en même temps dans le fichier. Les **sections critiques** d'un processus sont les parties de son code où il a accès à des ressources auxquelles un seul processus devrait pouvoir accéder à tout instant pour éviter les incohérences.
On utilise **l'exclusion mutuelle** pour assurer qu'un processus est le seul utilisateur des ressources concernées quand il entre dans une de ses sections critiques.
Considérons un compte bancaire dont le montant est à l'emplacement `A` du disque dur, et un programme consistant à ajouter 100 euros à ce compte (`N` est une variable locale de ce programme).

```
Lire(N, A)
N = N + 100;
Ecrire(N, A);
```

Communication interprocessus

On considère deux processus exécutant ce programme (solde initial du compte 1000 euros), l'enchaînement peut être :

Processus P_1	Processus P_2
`lire(N, A);`	
	`lire(N, A);`
	`N = N + 100;`
	`Ecrire(N, A);`
`N = N + 100;`	
`Ecrire(N, A);`	

À l'issue de cet enchaînement, le solde du compte est de 1100 euros, au lieu de 1200 euros, car chaque incrémentation se fait à partir de la valeur initiale égale à 1000.
Pour éviter cela, un processus au plus devrait pouvoir accéder au compte à tout instant : il faut imposer l'exclusion mutuelle.

Exclusion mutuelle

Imposer l'exclusion mutuelle signifie n'autoriser qu'un seul processus à modifier le solde du compte. Nous allons considérer quatre solutions pour tenter d'atteindre cet objectif :

1. Exclusion mutuelle par masquage des interruptions
2. Exclusion mutuelle par alternance stricte
3. Exclusion mutuelle sans alternance stricte
4. Exclusion mutuelle sans attente active

Solution 1 : Exclusion mutuelle par masquage des interruptions

Une première idée consiste à désactiver toutes les interruptions quand un processus entre en section critique. Les interruptions sont réactivées une fois sorti de la section critique. Ainsi, même l'horloge ne peut envoyer d'interruption, et l'ordonnanceur ne peut intervenir. Une telle solution est trop dangereuse : un processus utilisateur pourrait bloquer le système si le programmeur du processus oubliait de réactiver les interruptions.

Exclusion mutuelle

Solution 2 : Exclusion mutuelle par alternance stricte

Initialement, `turn = 0`.

Processus P_1	Processus P_2

`while(1){`
`while(turn != 0); /* boucle */`
`section_critique();`
`turn = 1;`
`section_noncritique(); }`
	`while(1){`
	`while(turn != 1); /* boucle */`
	`section_critique();`
	`turn = 0;`
	`section_noncritique(); }`

Le processus P_1 constate que `turn` vaut 0, il entre donc en section critique. Le processus P_2 teste en permanence la condition `turn != 1` en attendant que `turn` passe à 1. Cette situation se nomme **attente active** : P_2 n'est pas bloqué, et consomme du temps CPU inutilement. Le second inconvénient de cette solution est que si la section non critique de P_1 est très longue, alors P_2, après avoir exécuté sa section critique et sa section non critique une première fois, ne pourra pas entrer en section critique une seconde fois puisque que `turn` vaudra 0. On a des pertes de temps si les deux processus ont des temps d'exécution très différents.

Exclusion mutuelle

Solution 3 : Exclusion mutuelle sans alternance stricte

Solution de Peterson pour assurer l'exclusion mutuelle :

```
#define N 2
int tour; /* numéro du processus dont c'est le tour d'entrer en SC */
int demande[N]; /* 1 si le processus veut entrer en SC, 0 sinon */
void entrer_SC(int process) /* process vaut 0 ou 1 */
{
int autre; /* numéro de l'autre processus */
autre = N - 1 - process;
demande[process] = 1; /* le processus appelant demande à entrer en SC */
tour = process;
while(tour == process && demande[autre] == 1); /* on ne fait rien */
}
void sortir_SC(int process) /* process vaut 0 ou 1 */
{
demande[process] = 0; /* Sortie de la SC */
}
```

Exclusion mutuelle

On remarque que cette solution utilise elle aussi l'attente active (avec le `while` à la fin de `entrer_SC()`), mais sans imposer l'alternance stricte entre les processus P_0 et P_1.

À quoi sert-il de vérifier que `tour == process` sachant que la ligne précédente est `tour = process;` ?

Réponse : Si P_0 est interrompu jusque après `tour = process;` et que P_1 demande à entrer en section critique, alors, P_1 pourra entrer en section critique. Quand P_0 sera exécuté à nouveau, il ne pourra pas franchir le `while` car `tour` vaudra `1`.

Exclusion mutuelle

Solution 4 : Exclusion mutuelle sans attente active

A cette fin, on utilise un **verrou**. Il s'agit d'un objet système sur lequel deux opérations sont définies :

- *Verrouiller*(v) permet d'acquérir le verrou v. S'il n'est pas disponible, le processus est bloqué en attente de v
- *Deverrouiller*(v) permet au processus de libérer le verrou v qu'il possédait. Si un ou plusieurs processus étaient bloqués en attente de v, l'un d'eux est réveillé et reçoit le verrou.

En tant qu'opérations système, ces opérations ne sont pas interruptibles.

Supposons que ces opérations soient interruptibles, et que l'accès au solde du compte en banque soit protégé par un verrou `v`.

Exclusion mutuelle

`Verrouiller(v)` s'écrit :

```
if(v == 0)
v = 1;
else
Bloquer le processus;
```

`Deverrouiller(v)` s'écrit :

```
v = 0;
Réveiller un
processus bloqué;
```

Programme :

```
Verrouiller(v);
Lire(N, A);
N = N + 100;
Ecrire(N, A);
Déverrouiller(v);
```

Mais le déroulement suivant peut survenir :

Processus P_0	Processus P_1
`if(v == 0)`	
	`if(v == 0)`
	`v = 1;`
	`Lire(N, A);`
`v = 1;`	
`Lire(N, A);`	
`N = N + 100;`	
`Ecrire(N, A);`	
`v = 0;`	
	`N = N + 100;`
	`Ecrire(N, A);`
	`v = 0;`

Résultat final : le solde du compte est de 1100 au lieu de 1200.
Un verrou ne doit pas être programmé par l'utilisateur, il doit être implémenté comme une primitive système.

Exclusion mutuelle

Un **sémaphore** (introduit par Edsger Dijkstra en 1965) généralise un verrou. Il met à disposition un nombre fixé de jetons qui représentent chacun une unité de ressource. Les processus demandent des jetons et les restituent après utilisation des ressources. Un sémaphore avec un seul jeton est appelé **sémaphore binaire**, il se comporte comme un verrou. Son implémentation est appelée **mutex** (exclusion mutuelle). Trois opérations sont associées à un sémaphore :

- `Init(s, n)` qui initialise à `n` le nombre de jetons du sémaphore `s`
- `P(s)`, noté parfois `down(s)` pour demander un jeton au sémaphore `s`. Si le nombre de jetons disponibles est strictement positif, le jeton est attribué au processus demandeur. S'il est nul, le processus demandeur est endormi. **Non interruptible**
- `V(s)`, noté parfois `up(s)` pour restituer un jeton. Si le nombre de jetons disponibles est égal à `1` après cette restitution, l'OS réveille le processus endormi depuis la plus longue durée (FIFO) afin qu'il puisse prendre le jeton. **Non interruptible**

Exclusion mutuelle : producteur et consommateur

Deux processus partagent un tampon de taille fixe. L'un d'eux produit des données dans ce tampon (le producteur), un autre les consomme (le consommateur). Que se passe-t-il lorsque le tampon est plein, et que le producteur veut y stocker une donnée ? On veut qu'il se bloque, et ne soit réveillé que lorsque le consommateur aura consommé au moins une donnée. De même, si le consommateur veut lire une donnée dans le tampon alors qu'il est vide, celui-ci doit se bloquer et être réveillé quand le producteur aura produit au moins une donnée.
Soit `N` la capacité du tampon, et `compte` le nombre d'éléments dans le tampon à un instant donné.

- Si `compte == N` le producteur se bloque, sinon il ajoute une nouvelle donnée et incrémente `compte`
- Si `compte == 0` le consommateur se bloque, sinon il consomme une donnée et décrémente `compte`.

Chaque processus fait également un test pour savoir s'il doit réveiller l'autre processus.

Exclusion mutuelle : producteur et consommateur

```
#define N 100
int compte = 0;

void producteur(void)
{
int item;
while(1)
   {
   item = produire_item();
   if(compte == N) sleep();
   insere_item();
   compte++;
   if(compte == 1)
     wakeup(consommateur);
   }
}
```

```
void consommateur(void)
{
int item;
while(1)
   {
   if(compte == 0) sleep();
   item = retirer_item();
   compte--;
   if(compte == N - 1)
     wakeup(producteur);
   consommer_item();
   }
}
```

Cette solution est-elle satisfaisante ?

Exclusion mutuelle : producteur et consommateur

On considère que le tampon est vide, et que le consommateur le constate. À ce moment là, il est interrompu, et le producteur ajoute un élément. Comme `compte == 1`, il réveille le consommateur, qui ne s'était pas encore bloqué. Le signal `wakeup` est donc sans effet, et perdu. Quand le consommateur s'exécute à nouveau, il se met en sommeil puisqu'il avait fait le constat que le tampon était vide avant d'être interrompu. Quand le producteur aura rempli le tampon, il se mettra en sommeil, et ne pourra pas être réveillé.
La raison de ce blocage est que le signal `wakeup` a été perdu. On utilise deux sémaphores et un verrou pour résoudre le problème.

- `occupe` est un sémaphore dont les jetons représentent les places occupées dans le tampon, initialement, il n'y a aucun jeton.
- `libre` est un sémaphore dont les jetons représentent les places libres dans le tampon, initialement, il a `N` jetons.
- `mutex` est un sémaphore pour restreindre l'accès au tampon (initialement, un jeton).

Exclusion mutuelle : producteur et consommateur

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore libre = N;
semaphore occupe = 0;

void producteur(void)
{ int item;
while(1)
{
item = produire_item();
down(&libre); /* production */
down(&mutex); /* demande SC */
insere_item();
up(&mutex); /* quitte la SC */
up(&occupe); /* produit un
emplacement occupe */
}
}
```

```
void consommateur(void)
{ int item;
while(1)
{
down(&occupe); /* consommation */
down(&mutex); /* demande SC */
item = retirer_item();
up(&mutex); /* quitte la SC */
up(&libre); /* produit un
emplacement vide */
consommer_item();
}
}
```