

Concurrence, l'exclusion mutuelle

Luigi Santocanale

Laboratoire d'Informatique Fondamentale,
Centre de Mathématiques et Informatique,
39, rue Joliot-Curie - F-13453 Marseille

13 décembre 2005

- 1 Le partage des ressources
 - Les « race conditions » : exemples
 - Exclusion mutuelle par attente active
- 2 L'exclusion mutuelle sans attente
 - Les primitives `sleep` et `wakeup`
 - Les sémaphores et les mutex
 - Les moniteurs
- 3 Aperçu des interfaces **SysV IPC** et **POSIX threads**
 - Les sémaphores dans le System V IPC
 - Les threads POSIX
- 4 La primitive `select`
 - La primitive `select`

Plan

- 1 Le partage des ressources
 - Les « race conditions »: exemples
 - Exclusion mutuelle par attente active
- 2 L'exclusion mutuelle sans attente
 - Les primitives `sleep` et `wakeup`
 - Les sémaphores et les mutex
 - Les moniteurs
- 3 Aperçu des interfaces **SysV IPC** et **POSIX threads**
 - Les sémaphores dans le System V IPC
 - Les threads POSIX
- 4 La primitive `select`
 - La primitive `select`

Les « race conditions »

État de concurrence :

le résultat des calculs dépende de l'entrelacement (ordonnancement) des processus.

Exemple :

```
shared int x = 0 ;
```

P1 :

```
x++;
```

```
y = x;
```

P2 :

```
x++;
```

```
z = x;
```

Valeur de y,z ?

Les races condition peuvent amener à l'inconsistance des données partagés.

Les « race conditions »

État de concurrence :

le résultat des calculs dépende de l'entrelacement (ordonnancement) des processus.

Exemple :

```
shared int x = 0 ;
```

P1 :

```
x++;
```

```
y = x;
```

P2 :

```
x++;
```

```
z = x;
```

Valeur de y,z ?

Les races condition peuvent amener à l'inconsistance des données partagés.

Les « race conditions »

État de concurrence :

le résultat des calculs dépende de l'entrelacement (ordonnancement) des processus.

Exemple :

```
shared int x = 0 ;
```

P1 :

```
x++;
```

```
y = x;
```

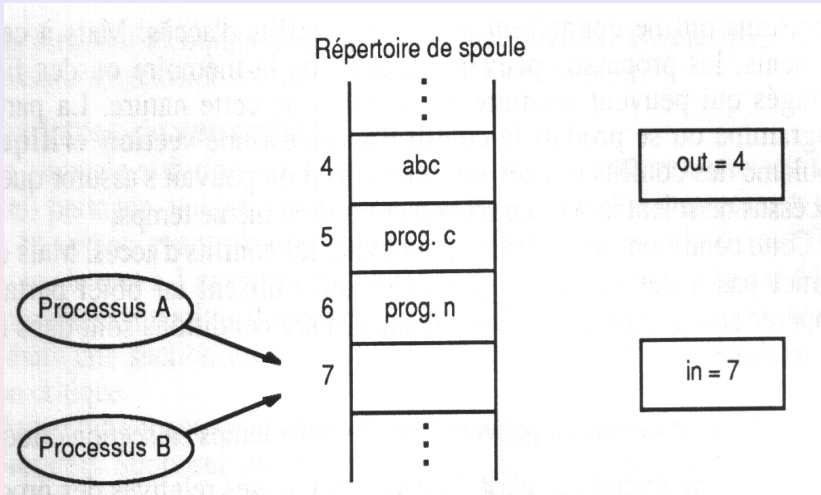
P2 :

```
x++;
```

```
z = x;
```

Valeur de y,z ?

Les races condition peuvent amener à l'inconsistance des données partagés.



Plan

- 1 Le partage des ressources
 - Les « race conditions »: exemples
 - Exclusion mutuelle par attente active
- 2 L'exclusion mutuelle sans attente
 - Les primitives `sleep` et `wakeup`
 - Les sémaphores et les mutex
 - Les moniteurs
- 3 Aperçu des interfaces **SysV IPC** et **POSIX threads**
 - Les sémaphores dans le System V IPC
 - Les threads POSIX
- 4 La primitive `select`
 - La primitive `select`

Exclusion mutuelle et section critique

Section critique :

- morceaux de code,
- pendant son execution on souhaite (il faut !!!) accéder à une ressource de façon exclusive (exclusion mutuelle).

Exemple :

```
for(;;){  
    /* code précédent */  
    ...  
  
    /* section critique */  
    in++;  
    write(ifd,&i,sizeof(in));  
    /* fin section critique */  
  
    /* code suivant */  
    ...  
}
```

Exclusion mutuelle et section critique

Section critique :

- morceaux de code,
- pendant son execution on souhaite (il faut !!!) accéder à une ressource de façon exclusive (exclusion mutuelle).

Exemple :

```
for(;;){  
    /* code précédent */  
    ...  
  
    /* section critique */  
    in++;  
    write(ifd,&i,sizeof(in));  
    /* fin section critique */  
  
    /* code suivant */  
    ...  
}
```

Réaliser l'exclusion : approche naïf

On peut le faire par :

- Masquage des interruptions.

Problèmes :

- ça marche seulement sur une architecture mono-processeur,
- pas souhaitable sur le cide de haut niveaux.

- Variables de verrouillage.

Problèmes :

- atomicité des opérations.

Réaliser l'exclusion : approche naïf

On peut le faire par :

- Masquage des interruptions.

Problèmes :

- ça marche seulement sur une architecture mono-processeur,
- pas souhaitable sur le cide de haut niveaux.

- Variables de verrouillage.

Problèmes :

- atomicité des opérations.

Réaliser l'exclusion : approche naïf

On peut le faire par :

- Masquage des interruptions.

Problèmes :

- ça marche seulement sur une architecture mono-processeur,
- pas souhaitable sur le cide de haut niveaux.

- Variables de verrouillage.

Problèmes :

- atomicité des opérations.

Réaliser l'exclusion : approche naïf

On peut le faire par :

- Masquage des interruptions.

Problèmes :

- ça marche seulement sur une architecture mono-processeur,
- pas souhaitable sur le cide de haut niveaux.

- Variables de verrouillage.

Problèmes :

- atomicité des opérations.

Atomicité

L'instruction

```
if (*x) x--;
```

devient

```
8      movl    8(%ebp), %eax
9      cmpl   $0, (%eax)
10     je     .L1
11     subl   $4, 8(%ebp)
12  .L1:
```

Atomicité

L'instruction

```
if (*x) x--;
```

devient

```
8          movl    8(%ebp), %eax
9          cmpl   $0, (%eax)
10         je     .L1
11         subl   $4, 8(%ebp)
12  .L1:
```


L'alternance

```

while (true)
{
    /* Ligne suiv: attente */
    while (tour != 0);
    section_critique();
    tour = 1;
    section_non_critique();
}

while (true)
{
    /* Ligne suiv: attente */
    while (tour != 1);
    section_critique();
    tour = 0;
    section_non_critique();
}

```

Problème :

si proc 0 a plus de taches (ou priorité majeure que) de proc 1.

L'alternance

```
while (true)                while (true)
{                            {
    /* Ligne suiv: attente */ /* Ligne suiv: attente */
    while (tour != 0);       while (tour != 1);
    section_critique();      section_critique();
    tour = 1;                tour = 0;
    section_non_critique();  section_non_critique();
}                            }
```

Problème :

si proc 0 a plus de taches (ou priorité majeure que) de proc 1.

La solution de Peterson

```
#define TRUE 1
#define FALSE 0
#define N 2 /* Nombre processus */

int tour;
int interesse[N];

entre_region(int process)
{
    int autre;
    autre = 1 - process;
    interesse[process] = TRUE;
    tour = process;
    /* Prochaine ligne : attente */
    while(tour == process && interesse[autre] == TRUE);
}

quitter_region(int process)
{
    interesse[process] = FALSE;
}
```

L'instruction TSL (Test, and set lock)

```
entrer_region:
```

```
    tsl registre , drapeau | copie de drapeau dans registre
                               | mettre drapeau a 1
    cmp registre , #0        | est registre egal à 0
    jnz entrer_region       | non : (verrous mis) on boucle
    ret                      | oui : retour,
                               | entree dans la section critique
```

```
quitter_region:
```

```
    mov drapeau , #0        | mettre 0 dans drapeau
    ret                      | retourner
```

Plan

- 1 Le partage des ressources
 - Les « race conditions »: exemples
 - Exclusion mutuelle par attente active
- 2 L'exclusion mutuelle sans attente
 - Les primitives sleep et wakeup
 - Les sémaphores et les mutex
 - Les moniteurs
- 3 Aperçu des interfaces **SysV IPC** et **POSIX threads**
 - Les sémaphores dans le System V IPC
 - Les threads POSIX
- 4 La primitive select
 - La primitive select

sleep et wakeup

Idée :

- Au lieu de l'attente active (= consommation de la CPU) on s'endort : `primitive`

`sleep,`

- Un processus peut réveiller un autre processus : `primitive`

`wakeup.`

sleep et wakeup

Idée :

- Au lieu de l'attente active (= consommation de la CPU) on s'endort : primitive

sleep,

- Un processus peut réveiller un autre processus : primitive

wakeup.

sleep et wakeup

Idée :

- Au lieu de l'attente active (= consommation de la CPU) on s'endort : primitive

sleep,

- Un processus peut reveiller un autre processus : primitive

wakeup.

sleep et wakeup

Idée :

- Au lieu de l'attente active (= consommation de la CPU) on s'endort : primitive

sleep,

- Un processus peut reveiller un autre processus : primitive

wakeup.

Le modèle producteur-consommateur

- n cases, chacune est libre ou pleine.
- Le producteur :
 - peut remplir une case s'il y en a libres, sinon il s'endort en attente d'être réveillé par le consommateur.
 - après avoir produit, il réveille le consommateur si une seule case est pleine.
- Le consommateur :
 - peut vider une case s'il y en a pleines, sinon il s'endort en attente d'être réveillé par le producteur.
 - après avoir consommé, il réveille le producteur si une seule case est vide.

Le modèle producteur-consommateur

- n cases, chacune est libre ou pleine.
- Le producteur :
 - peut remplir une case s'il y en a libres, sinon il s'endort en attente d'être réveillé par le consommateur.
 - après avoir produit, il réveille le consommateur si une seule case est pleine.
- Le consommateur :
 - peut vider une case s'il y en a pleines, sinon il s'endort en attente d'être réveillé par le producteur.
 - après avoir consommé, il réveille le producteur si une seule case est vide.

Le modèle producteur-consommateur

- n cases, chacune est libre ou pleine.
- Le producteur :
 - peut remplir une case s'il y en a libres, sinon il s'endort en attente d'être réveillé par le consommateur.
 - après avoir produit, il réveille le consommateur si une seule case est pleine.
- Le consommateur :
 - peut vider une case s'il y en a pleines, sinon il s'endort en attente d'être réveillé par le producteur.
 - après avoir consommé, il réveille le producteur si une seule case est vide.

Le modèle producteur-consommateur

- n cases, chacune est libre ou pleine.
- Le producteur :
 - peut remplir une case s'il y en a libres, sinon il s'endort en attente d'être réveillé par le consommateur.
 - après avoir produit, il réveille le consommateur si une seule case est pleine.
- Le consommateur :
 - peut vider une case s'il y en a pleines, sinon il s'endort en attente d'être réveillé par le producteur.
 - après avoir consommé, il réveille le producteur si une seule case est vide.

Le modèle producteur-consommateur

- n cases, chacune est libre ou pleine.
- Le producteur :
 - peut remplir une case s'il y en a libres, sinon il s'endort en attente d'être réveillé par le consommateur.
 - après avoir produit, il réveille le consommateur si une seule case est pleine.
- Le consommateur :
 - peut vider une case s'il y en a pleines, sinon il s'endort en attente d'être réveillé par le producteur.
 - après avoir consommé, il réveille le producteur si une seule case est vide.

Le modèle producteur-consommateur

- n cases, chacune est libre ou pleine.
- Le producteur :
 - peut remplir une case s'il y en a libres, sinon il s'endort en attente d'être réveillé par le consommateur.
 - après avoir produit, il réveille le consommateur si une seule case est pleine.
- Le consommateur :
 - peut vider une case s'il y en a pleines, sinon il s'endort en attente d'être réveillé par le producteur.
 - après avoir consommé, il réveille le producteur si une seule case est vide.

Le modèle producteur-consommateur

- n cases, chacune est libre ou pleine.
- Le producteur :
 - peut remplir une case s'il y en a libres, sinon il s'endort en attente d'être réveillé par le consommateur.
 - après avoir produit, il réveille le consommateur si une seule case est pleine.
- Le consommateur :
 - peut vider une case s'il y en a pleines, sinon il s'endort en attente d'être réveillé par le producteur.
 - après avoir consommé, il réveille le producteur si une seule case est vide.

pseudo-code

```
/* compteur est une variable partagé */  
/* entre le producteur et le consommateur */  
int compteur = 10; /* Nombre de cases pleines */
```

pseudo-code

```
/* compteur est une variable partagé */  
/* entre le producteur et le consommateur */  
int compteur = 10; /* Nombre de cases pleines */
```

```
producteur()  
{  
    while(1){  
        if(compteur == N) sleep();  
        compteur++;  
        if(compteur == 1)  
            wakeup(consommateur);  
    }  
}
```

pseudo-code

```
/* compteur est une variable partagé */  
/* entre le producteur et le consommateur */  
int compteur = 10; /* Nombre de cases pleines */
```

```
producteur()  
{  
    while(1){  
        if(compteur == N) sleep();  
        compteur++;  
        if(compteur == 1)  
            wakeup(consommateur);  
    }  
}
```

```
consommateur()  
{  
    while(1){  
        if(compteur == 0) sleep();  
        compteur--;  
        if(compteur == N-1)  
            wakeup(producteur);  
    }  
}
```

Problèmes

Assomption :

- un wakeup est **reçu si** le destinataire est **endormi**.

Problème :

- un wakeup est perdu si le destinataire n'est pas endormi.

Problèmes

Assomption :

- un wakeup est reçu si le destinataire est endormi.

Problème :

- un wakeup est **perdu** si le destinataire n'est **pas endormi**.

Exemples de perte de messages

Si `compteur == N`,
dans l'exécution parallèle de :

```

compteur--;
if(compteur == N) sleep();
if(compteur == N-1)
    wakeup(producteur);

```

De même, si `compteur == 0`,
dans l'exécution parallèle de :

```

compteur++;
if(compteur == 1)
    wakeup(consommateur);
if(compteur == 0) sleep();

```

Exemples de perte de messages

Si `compteur == N`,
dans l'exécution parallèle de :

```
if(compteur == N) sleep();
```

```
compteur--;
```

```
if(compteur == N-1)
    wakeup(producteur);
```

De même, si `compteur == 0`,
dans l'exécution parallèle de :

```
compteur++;
```

```
if(compteur == 1)
    wakeup(consommateur);
```

```
if(compteur == 0) sleep();
```

Exemples de perte de messages

Si `compteur == N`,
dans l'exécution parallèle de :

```
if(compteur == N) sleep();
```

```
compteur--;
```

```
if(compteur == N-1)
    wakeup(producteur);
```

De même, si `compteur == 0`,
dans l'exécution parallèle de :

```
compteur++;
```

```
if(compteur == 1)
    wakeup(consommateur);
```

```
if(compteur == 0) sleep();
```


Exemples de perte de messages

Si `compteur == N`,
dans l'exécution parallèle de :

```
if(compteur == N) sleep();
```

```
compteur--;
```

```
if(compteur == N-1)
    wakeup(producteur);
```

De même, si `compteur == 0`,
dans l'exécution parallèle de :

```
compteur++;
```

```
if(compteur == 1)
    wakeup(consommateur);
```

```
if(compteur == 0) sleep();
```

Solutions (I)

- un message envoyé à un processus pas endormi est mis en attente (masqué, bloqué) (Implementation : bit de réveil d'attente).
- Exécuter les codes

```
if(compteur == N) sleep();
```

```
compteur--;
```

```
if(compteur == N-1)
```

```
    wakeup(producteur);
```

```
compteur++;
```

```
if(compteur == 0) sleep();
```

```
if(compteur == 1)
```

```
    wakeup(consommateur);
```

en exclusion mutuelle.

Solutions (I)

- un message envoyé à un processus pas endormi est mis en attente (masqué, bloqué) (Implementation : bit de réveil d'attente).
- Exécuter les codes

```
if(compteur == N) sleep();
```

```
compteur--;
```

```
if(compteur == N-1)
```

```
    wakeup(producteur);
```

```
compteur++;
```

```
if(compteur == 1)
```

```
    wakeup(consommateur);
```

```
if(compteur == 0) sleep();
```

en exclusion mutuelle.

Solutions (I)

- un message envoyé à un processus pas endormi est mis en attente (masqué, bloqué) (Implementation : bit de réveil d'attente).
- Exécuter les codes

```
if(compteur == N) sleep();
```

```
compteur--;
```

```
if(compteur == N-1)
```

```
    wakeup(producteur);
```

```
compteur++;
```

```
if(compteur == 1)
```

```
    wakeup(consommateur);
```

```
if(compteur == 0) sleep();
```

en exclusion mutuelle.

Solutions (I)

- un message envoyé à un processus pas endormi est mis en attente (masqué, bloqué) (Implementation : bit de réveil d'attente).
- Exécuter les codes

```
if(compteur == N) sleep();
```

```
compteur--;
if(compteur == N-1)
    wakeup(producteur);
```

```
compteur++;
if(compteur == 1)
    wakeup(consommateur);
```

```
if(compteur == 0) sleep();
```

en exclusion mutuelle.

Solutions (II)

- Exécuter le code :

```
compteur--;  
if(compteur == N-1)  
    wakeup(producteur);
```

et le code :

```
compteur++;  
if(compteur == 1)  
    wakeup(consommateur);
```

de façon non interruptible, atomique.

Solutions (II)

- Exécuter le code :

```
compteur--;  
if(compteur == N-1)  
    wakeup(producteur);
```

et le code :

```
compteur++;  
if(compteur == 1)  
    wakeup(consommateur);
```

de façon non interruptible, atomique.

Solutions (II)

- Exécuter le code :

```
compteur--;  
if(compteur == N-1)  
    wakeup(producteur);
```

et le code :

```
compteur++;  
if(compteur == 1)  
    wakeup(consommateur);
```

de façon non interruptible, atomique.

Plan

- 1 Le partage des ressources
 - Les « race conditions »: exemples
 - Exclusion mutuelle par attente active
- 2 L'exclusion mutuelle sans attente
 - Les primitives sleep et wakeup
 - Les sémaphores et les mutex
 - Les moniteurs
- 3 Aperçu des interfaces **SysV IPC** et **POSIX threads**
 - Les sémaphores dans le System V IPC
 - Les threads POSIX
- 4 La primitive select
 - La primitive select

Les sémaphores

Un sémaphore est une variable

sem de type entier

- avec une opération down t.q. :
si $sem == 0$ et un processus fait $down(sem)$,
alors le processus s'endort.
Après : $sem == 0$.
- avec une opération up :
si $sem == 0$ et un processus fait $up(sem)$,
alors le processus réveille un autre processus à hasard,
endormi sur ce sémaphore,
Après : $sem == 1$.

Les opérations down et up sont atomiques.

Un *mutex* et un sémaphore qui

prend toujours un valeur entre 0, 1.

Les sémaphores

Un sémaphore est une variable

sem de type entier

- avec une opération down t.q. :
si $sem == 0$ et un processus fait $down(sem)$,
alors le processus s'endort.
Après : $sem == 0$.
- avec une opération up :
si $sem == 0$ et un processus fait $up(sem)$,
alors le processus réveille un autre processus à hasard,
endormi sur ce sémaphore,
Après : $sem == 1$.

Les opérations down et up sont atomiques.

Un *mutex* et un sémaphore qui

prend toujours un valeur entre 0, 1.

Les sémaphores

Un sémaphore est une variable

sem de type entier

- avec une opération down t.q. :
si $sem == 0$ et un processus fait $down(sem)$,
alors le processus s'endort.
Après : $sem == 0$.
- avec une opération up :
si $sem == 0$ et un processus fait $up(sem)$,
alors le processus réveille un autre processus à hasard,
endormi sur ce sémaphore,
Après : $sem == 1$.

Les opérations down et up sont atomiques.

Un *mutex* et un sémaphore qui

prend toujours un valeur entre 0,1.

Les sémaphores

Un sémaphore est une variable

sem de type entier

- avec une opération down t.q. :
si $sem == 0$ et un processus fait $down(sem)$,
alors le processus s'endort.
Après : $sem == 0$.
- avec une opération up :
si $sem == 0$ et un processus fait $up(sem)$,
alors le processus réveille un autre processus à hasard,
endormi sur ce sémaphore,
Après : $sem == 1$.

Les opérations down et up sont atomiques.

Un *mutex* et un sémaphore qui

prend toujours un valeur entre 0,1.

Les sémaphores

Un sémaphore est une variable

sem de type entier

- avec une opération down t.q. :
si $sem == 0$ et un processus fait $down(sem)$,
alors le processus s'endort.
Après : $sem == 0$.
- avec une opération up :
si $sem == 0$ et un processus fait $up(sem)$,
alors le processus réveille un autre processus à hasard,
endormi sur ce sémaphore,
Après : $sem == 1$.

Les opérations down et up sont atomiques.

Un *mutex* et un sémaphore qui

prend toujours un valeur entre 0, 1.

Programme : producteur-consommateur avec sémaphores

```
0 #define N 100
1
2 semaphore mutex, vide=N, plein=0;
3
4 producteur()
5 {
6     while(1){
7         down(vide);
8         down(mutex);
9         section_critique_producteur();
10        up(mutex);
11        up(plein);
12    }
13 }
```

Programme : producteur-consommateur avec sémaphores (II)

```
16 consommateur ()
17 {
18     while (1){
19         down(plein);
20         down(mutex);
21         section_critique_consommateur();
22         up(mutex);
23         up(vide);
24     }
25 }
```


Problèmes

- Pas évident à s'en servir :
- Si on échange les lignes 7-8, le système se bloque.
- Comment démontrer que le code ne se bloque jamais ?
(Dans ce cas, et dans le cas général).

Problèmes

- Pas évident à s'en servir :
- Si on échange les lignes 7-8, le système se bloque.
- Comment démontrer que le code ne se bloque jamais ?
(Dans ce cas, et dans le cas général).

Problèmes

- Pas évident à s'en servir :
- Si on échange les lignes 7-8, le système se bloque.
- Comment démontrer que le code ne se bloque jamais ?
(Dans ce cas, et dans le cas général).

Plan

- 1 Le partage des ressources
 - Les « race conditions »: exemples
 - Exclusion mutuelle par attente active
- 2 **L'exclusion mutuelle sans attente**
 - Les primitives sleep et wakeup
 - Les sémaphores et les mutex
 - **Les moniteurs**
- 3 Aperçu des interfaces **SysV IPC** et **POSIX threads**
 - Les sémaphores dans le System V IPC
 - Les threads POSIX
- 4 La primitive select
 - La primitive select

Les moniteurs

Morceaux de code contenant :

- variables partagées,
- une suite de fonctions :
 - le compilateur nous assure :
 - il y a toujours au plus un processus en train d'exécuter une de ces fonctions (il assure l'exclusion mutuelle),
- variables de conditions :
 - un processus peut s'endormir
 - `wait(condition)` – si cette variable est vraie,
 - un processus peut signaler cette variable
 - `signal(condition)` – il réveille un processus à hasard endormi sur cette variable, il sort du code du moniteur.

Problème :

comment garantir l'exclusion mutuelle des lecteurs ?

Les moniteurs

Morceaux de code contenant :

- variables partagées,
- une suite de fonctions :
 - le compilateur nous assure :
 - il y a toujours au plus un processus en train d'exécuter une de ces fonctions
 - (il assure l'exclusion mutuelle),
 - variables de conditions :
 - un processus peut s'endormir
 - `wait(condition)` –
 - si cette variable est vraie,
 - un processus peut signaler cette variable
 - `signal(condition)` –
 - il réveille un processus à hasard endormi sur cette variable,
 - il sort du code du moniteur.

Problème :

comment garantir l'exclusion mutuelle des processus ?

Les moniteurs

Morceaux de code contenant :

- variables partagées,
- une suite de fonctions :
 - le compilateur nous assure :
 - il y a toujours au plus un processus en train d'exécuter une de ces fonctions
 - (il assure l'exclusion mutuelle),
- variables de conditions :
 - un processus peut s'endormir
 - `wait(condition)` –
 - si cette variable est vraie,
 - un processus peut signaler cette variable
 - `signal(condition)` –
 - il réveille un processus à hasard endormi sur cette variable,
 - il sort du code du moniteur.

Problème :

comment garantir l'exclusion mutuelle des processus ?

Les moniteurs

Morceaux de code contenant :

- variables partagées,
- une suite de fonctions :
 - le compilateur nous assure :
 - il y a toujours au plus un processus en train d'exécuter une de ces fonctions
 - (il assure l'exclusion mutuelle),
- variables de conditions :
 - un processus peut s'endormir
 - `wait(condition)` –
 - si cette variable est vraie,
 - un processus peut signaler cette variable
 - `signal(condition)` –
 - il réveille un processus à hasard endormi sur cette variable,
 - il sort du code du moniteur.

Problème :

comment garantir l'exclusion mutuelle ?

Les moniteurs

Morceaux de code contenant :

- variables partagées,
- une suite de fonctions :
 - le compilateur nous assure :
il y a toujours au plus un processus en train d'exécuter une de ces fonctions
(il assure l'exclusion mutuelle),
- variables de conditions :
 - un processus peut s'endormir
 - `wait(condition)` –si cette variable est vraie,
 - un processus peut signaler cette variable
 - `signal(condition)` –il réveille un processus à hasard endormi sur cette variable,
il sort du code du moniteur.

Problème :

existence des moniteurs dépend des langages et des compilateurs

Les moniteurs

Morceaux de code contenant :

- variables partagées,
- une suite de fonctions :
 - le compilateur nous assure :
il y a toujours au plus un processus en train d'exécuter une de ces fonctions
(il assure l'exclusion mutuelle),
- variables de conditions :
 - un processus peut s'endormir
 - `wait(condition)` –si cette variable est vraie,
 - un processus peut signaler cette variable
 - `signal(condition)` –il réveille un processus à hasard endormi sur cette variable,
il sort du code du moniteur.

Problème :

existence des moniteurs dépend des langages et des compilateurs

Les moniteurs

Morceaux de code contenant :

- variables partagées,
- une suite de fonctions :
 - le compilateur nous assure :
il y a toujours au plus un processus en train d'exécuter une de ces fonctions
(il assure l'exclusion mutuelle),
- variables de conditions :
 - un processus peut s'endormir
 - `wait(condition)` –si cette variable est vraie,
 - un processus peut signaler cette variable
 - `signal(condition)` –il réveille un processus à hasard endormi sur cette variable,
il sort du code du moniteur.

Problème :

existence des moniteurs dépende des langages et des compilateurs

Les moniteurs

Morceaux de code contenant :

- variables partagées,
- une suite de fonctions :
 - le compilateur nous assure :
 - il y a toujours au plus un processus en train d'exécuter une de ces fonctions
 - (il assure l'exclusion mutuelle),
- variables de conditions :
 - un processus peut s'endormir
 - `wait(condition)` –
 - si cette variable est vraie,
 - un processus peut signaler cette variable
 - `signal(condition)` –
 - il réveille un processus à hasard endormi sur cette variable,
 - il sort du code du moniteur.

Problème :

existence des moniteurs dépende des langages et des compilateurs

Plan

- 1 Le partage des ressources
 - Les « race conditions »: exemples
 - Exclusion mutuelle par attente active
- 2 L'exclusion mutuelle sans attente
 - Les primitives `sleep` et `wakeup`
 - Les sémaphores et les mutex
 - Les moniteurs
- 3 Aperçu des interfaces **SysV IPC** et **POSIX threads**
 - Les sémaphores dans le System V IPC
 - Les threads POSIX
- 4 La primitive `select`
 - La primitive `select`

Le IPC, ou « Inter Process Communication »

Permet de :

- Partager des segments de mémoire entre processus.
- Utiliser des files de messages (« Message Queues »):
 - Message Queue :
 - Une chaîne de messages.
 - chacun d'une longueur fixé maximale.
 - Les messages sont typés.
 - Lecture et écriture de type FIFO.
- Utiliser des ensembles de sémaphores.

Le IPC, ou « Inter Process Communication »

Permet de :

- Partager des segments de mémoire entre processus.
- Utiliser des files de messages (« Message Queues »):
Message Queue :
 - Liste chaîné de messages,
 - chacune d'un longueur fixé maximale.
 - Les messages sont typées.
 - Lecture et écriture de type FIFO.
- Utiliser des ensembles de sémaphores.

Le IPC, ou « Inter Process Communication »

Permet de :

- Partager des segments de mémoire entre processus.
- Utiliser des files de messages (« Message Queues »):

Message Queue :

Liste chaîné de messages,

chacune d'un longueur fixé maximale.

Les messages sont typées.

Lecture et écriture de type FIFO.

- Utiliser des ensembles de sémaphores.

Le IPC, ou « Inter Process Communication »

Permet de :

- Partager des segments de mémoire entre processus.
- Utiliser des files de messages (« Message Queues »):

Message Queue :

Liste chaîné de messages,

chacune d'un longueur fixé maximale.

Les messages sont typées.

Lecture et écriture de type FIFO.

- Utiliser des ensembles de sémaphores.

Le IPC, ou « Inter Process Communication »

Permet de :

- Partager des segments de mémoire entre processus.
- Utiliser des files de messages (« Message Queues »):

Message Queue :

Liste chaîné de messages,

chacune d'un longueur fixé maximale.

Les messages sont typés.

Lecture et écriture de type FIFO.

- Utiliser des ensembles de sémaphores.

Le IPC, ou « Inter Process Communication »

Permet de :

- Partager des segments de mémoire entre processus.
- Utiliser des files de messages (« Message Queues »):
Message Queue :
 - Liste chaîné de messages,
chacune d'un longueur fixé maximale.
 - Les messages sont typées.
 - Lecture et écriture de type FIFO.
- Utiliser des ensembles de sémaphores.

Le IPC, ou « Inter Process Communication »

Permet de :

- Partager des segments de mémoire entre processus.
- Utiliser des files de messages (« Message Queues »):
Message Queue :
 - Liste chaîné de messages,
chacune d'un longueur fixé maximale.
 - Les messages sont typées.
 - Lecture et écriture de type FIFO.
- Utiliser des ensembles de sémaphores.

semget

```
#include <sys/sem.h>
int semget(key_t key, int nb_sem, int flags);
```

key : IPC_PRIVATE : permet la communication entre processus de la même famille.
Sinon, deux processus doivent se mettre d'accord sur une clé commune.

nb_sem : nombre de sémaphores dans l'ensemble.

flags : permissions lecture/écriture.

Retourne : un identificateur d'ensemble de sémaphores

Sommaire : crée un ensemble de sémaphores.

semget

```
#include <sys/sem.h>
int semget(key_t key, int nb_sem, int flags);
```

key : IPC_PRIVATE : permet la communication entre processus de la même famille.
Sinon, deux processus doivent se mettre d'accord sur une clé commune.

nb_sem : nombre de sémaphores dans l'ensemble.

flags : permissions lecture/écriture.

Retourne : un identificateur d'ensemble de sémaphores

Sommaire : crée un ensemble de sémaphores.

semget

```
#include <sys/sem.h>
int semget(key_t key, int nb_sem, int flags);
```

key : IPC_PRIVATE : permet la communication entre processus de la même famille.
Sinon, deux processus doivent se mettre d'accord sur une clé commune.

nb_sem : nombre de sémaphores dans l'ensemble.

flags : permissions lecture/écriture.

Retourne : un identificateur d'ensemble de sémaphores

Sommaire : crée un ensemble de sémaphores.

semget

```
#include <sys/sem.h>
int semget(key_t key, int nb_sem, int flags);
```

key : IPC_PRIVATE : permet la communication entre processus de la même famille.
Sinon, deux processus doivent se mettre d'accord sur une clé commune.

nb_sem : nombre de sémaphores dans l'ensemble.

flags : permissions lecture/écriture.

Retourne : un identificateur d'ensemble de sémaphores

Sommaire : crée un ensemble de sémaphores.

semget

```
#include <sys/sem.h>
int semget(key_t key, int nb_sem, int flags);
```

key : IPC_PRIVATE : permet la communication entre processus de la même famille.
Sinon, deux processus doivent se mettre d'accord sur une clé commune.

nb_sem : nombre de sémaphores dans l'ensemble.

flags : permissions lecture/écriture.

Retourne : un identificateur d'ensemble de sémaphores

Sommaire : crée un ensemble de sémaphores.

semget

```
#include <sys/sem.h>
int semget(key_t key, int nb_sem, int flags);
```

key : IPC_PRIVATE : permet la communication entre processus de la même famille.
Sinon, deux processus doivent se mettre d'accord sur une clé commune.

nb_sem : nombre de sémaphores dans l'ensemble.

flags : permissions lecture/écriture.

Retourne : un identificateur d'ensemble de sémaphores

Sommaire : crée un ensemble de sémaphores.

Autres opérations

```
#include <sys/sem.h>
```

```
int semctl(int semid, int nb, int cmd, ...);
```

semid : quel ensemble

nb : quel sémaphore dans l'ensemble

cmd : GETVAL, SETVAL, ...

Retourne : -1/0/la valeur du sémaphore

Sommaire : initialisation, lecture d'un sémaphore

```
int semop(int semid, struct sembuf * sops, size_t nsops);
```

semid : quel ensemble

sops : tableau d'opérations à faire de façon atomique

nsops : dimension de ce tableau

Retourne : -1/0

Sommaire : les opérations UP et DOWN

Autres opérations

```
#include <sys/sem.h>
```

```
int semctl(int semid, int nb, int cmd, ...);
```

semid : quel ensemble

nb : quel sémaphore dans l'ensemble

cmd : GETVAL, SETVAL, ...

Retourne : -1/0/la valeur du sémaphore

Sommaire : initialisation, lecture d'un sémaphore

```
int semop(int semid, struct sembuf * sops, size_t nsops);
```

semid : quel ensemble

sops : tableau d'opérations à faire de façon atomique

nsops : dimension de ce tableau

Retourne : -1/0

Sommaire : les opérations UP et DOWN

Autres opérations

```
#include <sys/sem.h>
```

```
int semctl(int semid, int nb, int cmd, ...);
```

semid : quel ensemble

nb : quel sémaphore dans l'ensemble

cmd : GETVAL, SETVAL, ...

Retourne : -1/0/la valeur du sémaphore

Sommaire : initialisation, lecture d'un sémaphore

```
int semop(int semid, struct sembuf * sops, size_t nsops);
```

semid : quel ensemble

sops : tableau d'opérations à faire de façon atomique

nsops : dimension de ce tableau

Retourne : -1/0

Sommaire : les opérations UP et DOWN

Programme : producteur-consommateur sous IPC

```
0 #include <sys/sem.h>
1 #include <unistd.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <time.h>
5 #include <sys/fcntl.h>
6
7 /* Gestion des sémaphores Sys V IPC*/
8
9 /* Créer un sémaphore */
10 #define SEMCREATE() semget(IPC_PRIVATE,1,0600)
11
12 /* L'index du seul sémaphore dans l'ensemble */
13 #define NOSEM 0
14
15 /* Utilisation de semop */
16
17 /* Une structure sembuf code
18     une action sur le seul sémaphore */
19 struct sembuf plus={NOSEM,1,0};
20 struct sembuf moins={NOSEM,-1,0};
21 #define DOWN(semid) semop(semid,&moins,1)
22 #define UP(semid) semop(semid,&plus,1)
```

Programme : producteur-consommateur sous IPC (II)

```
25 /* Utilisation de semop */
26
27 union semun {
28     int val;
29     struct semid_ds *buf;
30     unsigned short *array;
31 } arg;
32
33 int valeur(int semid){
34     return semctl(semid,NOSEM,GETVAL, arg);
35 }
36
37 int semset(int semid, int n) {
38     arg.val=n;
39     return semctl(semid,NOSEM,SETVAL, arg);
40 }
```

Programme : producteur-consommateur sous IPC (III)

```
42 /* Voici le consommateur/producteur */
43 #define NB_CASES 10
44 #define DELAIMAX 2
45
46 void producteur(void);
47 void consommateur(void);
48 int mutex,vide,plein;
49
50 int main(void)
51 {
52     if((mutex=SEMCREATE()) == -1
53         || (vide=SEMCREATE()) == -1
54         || (plein=SEMCREATE()) == -1)
55         exit(EXIT_FAILURE);
56
57     if((semset(mutex,1) == -1)
58         || (semset(vide,NB_CASES) == -1)
59         || (semset(plein,0) == -1))
60         exit(EXIT_FAILURE);
```


Programme : producteur-consommateur sous IPC (IV)

```
62     printf("Au debut : mutex %d, "  
63           "cases vides : %d,  pleines : %d.\n",  
64           valeur(mutex), valeur(vide), valeur(plein));  
65  
66     switch(fork())  
67     {  
68     case -1:  
69         exit(EXIT_FAILURE);  
70     case 0:  
71         producteur();  
72     default:  
73         consommateur();  
74     }  
75     exit(EXIT_FAILURE);  
76 }
```

Programme : producteur-consommateur sous IPC (V)

```
78 void producteur(void)
79 {
80     int pid=getpid();
81     srand(pid*time(NULL));
82     while(1){
83         sleep(rand()%DELAIMAX);
84         DOWN(vide);
85         DOWN(mutex);
86         sleep(rand()%DELAIMAX);
87         UP(mutex);
88         UP(plein);
89         printf("[%d] Cases vides : %d, cases pleines : %d\n",
90                pid,valeur(vide),valeur(plein));
91     }
92 }
```

Programme : producteur-consommateur sous IPC (VI)

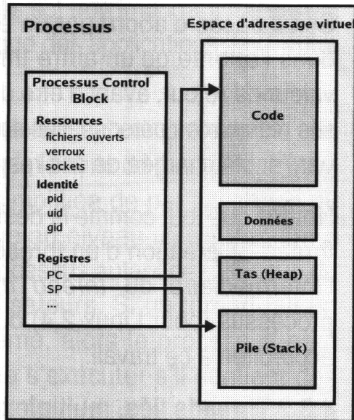
```
94 void consommateur(void)
95 {
96     int pid=getpid();
97     srand(getpid()*time(NULL));
98     while(1){
99         sleep(rand()%DELAIMAX);
100         DOWN(plein);
101         DOWN(mutex);
102         sleep(rand()%DELAIMAX);
103         UP(mutex);
104         UP(vide);
105         printf("[%d] Cases vides : %d, cases pleines : %d\n",
106                pid, valeur(vide), valeur(plein));
107     }
108 }
```

Session : prodcons

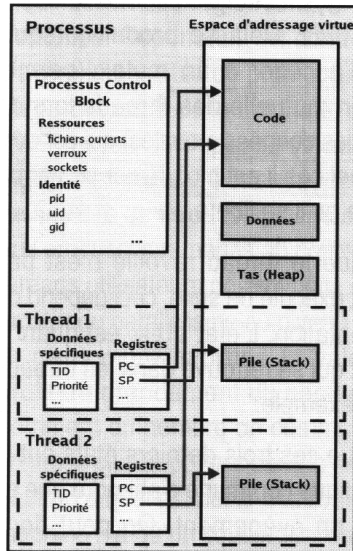
```
[lsantoca@localhost lecture10]$ a.out
Au debut : mutex 1, vide : 10, cases pleine : 0.
[7055] Cases vides : 9, cases pleines : 0.
[7055] Cases vides : 8, cases pleines : 1.
[7054] Cases vides : 9, cases pleines : 1.
[7055] Cases vides : 8, cases pleines : 2.
[7055] Cases vides : 7, cases pleines : 3.
[7054] Cases vides : 8, cases pleines : 2.
[7054] Cases vides : 8, cases pleines : 1.
[7055] Cases vides : 8, cases pleines : 1.
[7054] Cases vides : 8, cases pleines : 1.
[7055] Cases vides : 8, cases pleines : 1.
[7054] Cases vides : 8, cases pleines : 1.
[7055] Cases vides : 8, cases pleines : 1.
[7054] Cases vides : 8, cases pleines : 1.
[7055] Cases vides : 8, cases pleines : 2.
[7055] Cases vides : 7, cases pleines : 2.
[7054] Cases vides : 7, cases pleines : 2.
[7055] Cases vides : 7, cases pleines : 2.
[7054] Cases vides : 7, cases pleines : 2.
[7055] Cases vides : 7, cases pleines : 2.
[7054] Cases vides : 7, cases pleines : 2.
[7055] Cases vides : 7, cases pleines : 2.
[7054] Cases vides : 7, cases pleines : 2.
```

Plan

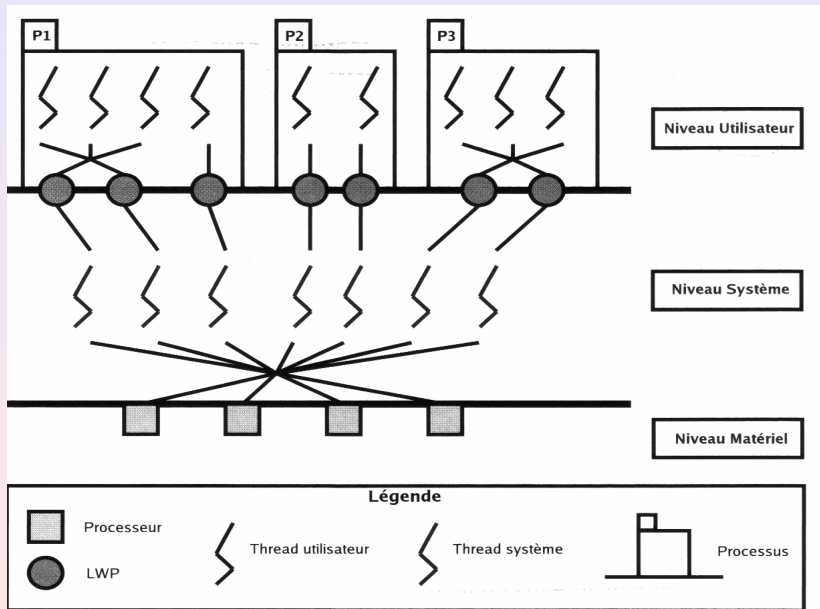
- 1 Le partage des ressources
 - Les « race conditions »: exemples
 - Exclusion mutuelle par attente active
- 2 L'exclusion mutuelle sans attente
 - Les primitives `sleep` et `wakeup`
 - Les sémaphores et les mutex
 - Les moniteurs
- 3 Aperçu des interfaces **SysV IPC** et **POSIX threads**
 - Les sémaphores dans le System V IPC
 - Les threads POSIX
- 4 La primitive `select`
 - La primitive `select`



Processus classiques



Processus multithreadé



Programme : partageress.c (I)

```
0 #include <pthread.h>
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4
5 #define MAX_RESOURCE 7
6 #define NB_THREADS 3
7
8 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
9 pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
10
11 unsigned int resources=MAX_RESOURCE;
12 unsigned int waiting=0;
```


Programme : partageress.c (II)

```
15 void get_resources(int amount)
16 {
17     pthread_mutex_lock(&mutex);
18
19     while(resources < amount){
20         waiting++;
21         pthread_cond_wait(&cond, &mutex);
22     }
23     resources -= amount;
24
25     pthread_mutex_unlock(&mutex);
26 }
```

Programme : partageress.c (III)

```
15 void get_resources(int amount)
16 {
17     pthread_mutex_lock(&mutex);
18
19     while(resources < amount){
20         waiting++;
21         pthread_cond_wait(&cond, &mutex);
22     }
23     resources -= amount;
24
25     pthread_mutex_unlock(&mutex);
26 }
```

Programme : partageress.c (IV)

```
28 void free_resources(int amount)
29 {
30     pthread_mutex_lock(&mutex);
31
32     resources += amount;
33     if(waiting > 0){
34         waiting = 0;
35         pthread_cond_broadcast(&cond);
36     }
37
38     pthread_mutex_unlock(&mutex);
39 }
```

Programme : partageress.c (V)

```
41 void *fonction(){
42     int i, tid, result;
43
44     tid = (int)pthread_self();
45     for (i = 3 ; i > 0 ; i--)
46     {
47         printf("thread %d : je demande %d ressources\n",tid,
48             get_resources(i);
49             sleep(1);
50             printf("thread %d : je libere %d ressources\n",tid,
51                 free_resources(i);
52     }
53     result=2;
54
55     pthread_exit((void *)result);
56 }
```

Programme : partageress.c (VI)

```
58 int main()
59 {
60     int i;
61     void *retour;
62     pthread_t thread[NB_THREADS];
63
64     for(i=0;i < NB_THREADS; i++)
65         pthread_create(&thread[i],NULL,fonction,NULL);
66
67     for(i=0;i < NB_THREADS; i++)
68     {
69         pthread_join(thread[i],&retour);
70         printf("thread main : le thread %d se termine (result %d\n",
71             (int)thread[i],(int)retour);
72     }
73
74     pthread_exit(NULL);
75 }
```

Session : partageress

```
[lsantoca@localhost lecture10]$ a.out
thread 16386 : je demande 3 ressources
thread 32771 : je demande 3 ressources
thread 49156 : je demande 3 ressources
thread 16386 : je libre 3 ressources
thread 16386 : je demande 2 ressources
thread 32771 : je libre 3 ressources
thread 32771 : je demande 2 ressources
thread 16386 : je libre 2 ressources
thread 16386 : je demande 1 ressources
thread 32771 : je libre 2 ressources
thread 32771 : je demande 1 ressources
thread 49156 : je libre 3 ressources
thread 49156 : je demande 2 ressources
thread 16386 : je libre 1 ressources
thread 32771 : je libre 1 ressources
thread 49156 : je libre 2 ressources
thread 49156 : je demande 1 ressources
thread main : le thread 16386 se termine (result 2)
thread main : le thread 32771 se termine (result 2)
thread 49156 : je libre 1 ressources
thread main : le thread 49156 se termine (result 2)
```

Plan

- 1 Le partage des ressources
 - Les « race conditions »: exemples
 - Exclusion mutuelle par attente active
- 2 L'exclusion mutuelle sans attente
 - Les primitives `sleep` et `wakeup`
 - Les sémaphores et les mutex
 - Les moniteurs
- 3 Aperçu des interfaces **SysV IPC** et **POSIX threads**
 - Les sémaphores dans le System V IPC
 - Les threads POSIX
- 4 **La primitive select**
 - La primitive select

select

```
#include <sys/select.h>
int select(int maxfd, fd_set * read, fd_set * write, fd_set *
err, struct timeval * t_out);
```

read : (pointeur à une structure de) ensemble de descripteurs de fichiers ouverts en lecture dont on s'en met en écoute.

Au retour cette structure est remplie avec les descripteurs prêts (non bloquantes).

NULL si on est pas intéressé à un tel ensemble.

write : ensemble de descripteurs de fichiers ouverts en lecture dont on s'en met en écoute.

Au retour cette structure est remplie avec les descripteurs prêts.

NULL si pas intéressé.

err : ensemble de descripteurs de fichiers (possiblement contenant un erreur) dont on s'en met en écoute.

Au retour cette structure est remplie avec les descripteurs prêts.

NULL si pas intéressé.

t_out : on se débloque après * t_out.

Si NULL on reste bloqué.

Retourne : no. descripteurs de fichier prêts, 0 si on est débloqué.

select

```
#include <sys/select.h>
int select(int maxfd, fd_set * read, fd_set * write, fd_set *
err, struct timeval * t_out);
```

read : (pointeur à une structure de) ensemble de descripteurs de fichiers ouverts en lecture dont on s'en met en écoute.

Au retour cette structure est remplie avec les descripteurs prêts (non bloquantes).

NULL si on est pas intéressé à un tel ensemble.

write : ensemble de descripteurs de fichiers ouverts en lecture dont on s'en met en écoute.

Au retour cette structure est remplie avec les descripteurs prêts.

NULL si pas intéressé.

err : ensemble de descripteurs de fichiers (possiblement contenant un erreur) dont on s'en met en écoute.

Au retour cette structure est remplie avec les descripteurs prêts.

NULL si pas intéressé.

t_out : on se débloque après * t_out.

Si NULL on reste bloqué.

Retourne : no. descripteurs de fichier prêts, 0 si on est débloqué

select

```
#include <sys/select.h>
int select(int maxfd, fd_set * read, fd_set * write, fd_set *
err, struct timeval * t_out);
```

read : (pointeur à une structure de) ensemble de descripteurs de fichiers ouverts en lecture dont on s'en met en écoute.

Au retour cette structure est remplie avec les descripteurs prêts (non bloquantes).

NULL si on est pas intéressé à un tel ensemble.

write : ensemble de descripteurs de fichiers ouverts en lecture dont on s'en met en écoute.

Au retour cette structure est remplie avec les descripteurs prêts.

NULL si pas intéressé.

err : ensemble de descripteurs de fichiers (possiblement contenant un erreur) dont on s'en met en écoute.

Au retour cette structure est remplie avec les descripteurs prêts.

NULL si pas intéressé.

t_out : on se débloque après * t_out.

Si NULL on reste bloqué.

Retourne : no. descripteurs de fichier prêts, 0 si on est débloqué par t_out, -1 si erreur

select

```
#include <sys/select.h>
int select(int maxfd, fd_set * read, fd_set * write, fd_set *
err, struct timeval * t_out);
```

read : (pointeur à une structure de) ensemble de descripteurs de fichiers ouverts en lecture dont on s'en met en écoute.

Au retour cette structure est remplie avec les descripteurs prêts (non bloquantes).

NULL si on est pas intéressé à un tel ensemble.

write : ensemble de descripteurs de fichiers ouverts en lecture dont on s'en met en écoute.

Au retour cette structure est remplie avec les descripteurs prêts.

NULL si pas intéressé.

err : ensemble de descripteurs de fichiers (possiblement contenant un erreur) dont on s'en met en écoute.

Au retour cette structure est remplie avec les descripteurs prêts.

NULL si pas intéressé.

t_out : on se débloque après * t_out.

Si NULL on reste bloqué.

Retourne : no. descripteurs de fichier prêts, 0 si on est débloqué par t_out, -1 si erreur.

select

```
#include <sys/select.h>
int select(int maxfd, fd_set * read, fd_set * write, fd_set *
err, struct timeval * t_out);
```

read : (pointeur à une structure de) ensemble de descripteurs de fichiers ouverts en lecture dont on s'en met en écoute.

Au retour cette structure est remplie avec les descripteurs prêts (non bloquantes).

NULL si on est pas intéressé à un tel ensemble.

write : ensemble de descripteurs de fichiers ouverts en lecture dont on s'en met en écoute.

Au retour cette structure est remplie avec les descripteurs prêts.

NULL si pas intéressé.

err : ensemble de descripteurs de fichiers (possiblement contenant un erreur) dont on s'en met en écoute.

Au retour cette structure est remplie avec les descripteurs prêts.

NULL si pas intéressé.

t_out : on se débloque après * t_out.

Si NULL on reste bloqué.

Retourne : no. descripteurs de fichier prêts, 0 si on est débloqué par t_out, -1 si erreur.

select

```
#include <sys/select.h>
int select(int maxfd, fd_set * read, fd_set * write, fd_set *
err, struct timeval * t_out);
```

read : (pointeur à une structure de) ensemble de descripteurs de fichiers ouverts en lecture dont on s'en met en écoute.

Au retour cette structure est remplie avec les descripteurs prêts (non bloquantes).

NULL si on est pas intéressé à un tel ensemble.

write : ensemble de descripteurs de fichiers ouverts en lecture dont on s'en met en écoute.

Au retour cette structure est remplie avec les descripteurs prêts.

NULL si pas intéressé.

err : ensemble de descripteurs de fichiers (possiblement contenant un erreur) dont on s'en met en écoute.

Au retour cette structure est remplie avec les descripteurs prêts.

NULL si pas intéressé.

t_out : on se débloque après * t_out.

Si NULL on reste bloqué.

Retourne : no. descripteurs de fichier prêts, 0 si on est débloqué par *t_out*, -1 si erreur.



La structure timeval

```
struct timeval {  
    time_t          tv_sec ;      /* Seconds. */  
    suseconds_t     tv_usec ;    /* Microseconds. */  
}
```

Les ensembles de descripteurs

```
#include <sys/select.h>
```

```
void FD_ZERO(fd_set * ens);
```

Sommaire : *ens* devient l'ensemble vide

```
void FD_SET(int fd, fd_set * ens);
```

Sommaire : add *fd* à *ens*

```
void FD_CLEAR(int fd, fd_set * ens);
```

Sommaire : efface *fd* de *ens*

```
int FD_ISSET(int fd, fd_set * ens);
```

Sommaire : vérifie si *fd* appartient à *ens*

Les ensembles de descripteurs

```
#include <sys/select.h>
```

```
void FD_ZERO(fd_set * ens);
```

Sommaire : *ens* devient l'ensemble vide

```
void FD_SET(int fd, fd_set * ens);
```

Sommaire : add *fd* à *ens*

```
void FD_CLEAR(int fd, fd_set * ens);
```

Sommaire : efface *fd* de *ens*

```
int FD_ISSET(int fd, fd_set * ens);
```

Sommaire : vérifie si *fd* appartient à *ens*

Les ensembles de descripteurs

```
#include <sys/select.h>
```

```
void FD_ZERO(fd_set * ens);
```

Sommaire : *ens* devient l'ensemble vide

```
void FD_SET(int fd, fd_set * ens);
```

Sommaire : add *fd* à *ens*

```
void FD_CLEAR(int fd, fd_set * ens);
```

Sommaire : efface *fd* de *ens*

```
int FD_ISSET(int fd, fd_set * ens);
```

Sommaire : vérifie si *fd* appartient à *ens*

Les ensembles de descripteurs

```
#include <sys/select.h>
```

```
void FD_ZERO(fd_set * ens);
```

Sommaire : *ens* devient l'ensemble vide

```
void FD_SET(int fd, fd_set * ens);
```

Sommaire : add *fd* à *ens*

```
void FD_CLEAR(int fd, fd_set * ens);
```

Sommaire : efface *fd* de *ens*

```
int FD_ISSET(int fd, fd_set * ens);
```

Sommaire : vérifie si *fd* appartient à *ens*

Les ensembles de descripteurs

```
#include <sys/select.h>
```

```
void FD_ZERO(fd_set * ens);
```

Sommaire : *ens* devient l'ensemble vide

```
void FD_SET(int fd, fd_set * ens);
```

Sommaire : add *fd* à *ens*

```
void FD_CLEAR(int fd, fd_set * ens);
```

Sommaire : efface *fd* de *ens*

```
int FD_ISSET(int fd, fd_set * ens);
```

Sommaire : vérifie si *fd* appartient à *ens*

Programme : exempleselect.c

```
0 #include <stdio.h>
1 #include <stdlib.h>
2 #include <unistd.h>
3 #include <sys/stat.h>
4 #include <sys/fcntl.h>
5 #include <sys/select.h>
6 #include <signal.h>
7
8 #define TUBE "fifo"
9 #define ERR 1
10 #define OK 0
11 #define NOTUBES 3
12 #define TIMEOUT 60
13 #define POURTOUTTUBE(i) for(i=0;i<NOTUBES;i++)
14 #define MAX(n,m) ((n < m)?m:n)
15
16 int fd[NOTUBES];
17 char nomtube[NOTUBES][256];
```

Programme : exempleselect.c (II)

```
20 void sortir(char *message,int err){
21     int i;
22
23     POURTOUTTUBE(i)
24         unlink(nomtube[i]);
25
26     if(err)
27     {
28         perror(message);
29         exit(EXIT_FAILURE);
30     }
31     printf("Terminaison du serveur : %s\n",message);
32     exit(EXIT_SUCCESS);
33 }
34
35 void handler(int sig){
36     sortir("Signaled",ERR);
37 }
```

Programme : exempleselect.c (III)

```
39 void traiter(fd_set *readfds)
40 {
41     int i,n;
42     char tampon[256];
43
44     POURTOUTTUBE(i)
45     {
46         if(FD_ISSET(fd[i],readfds))
47         {
48             n=read(fd[i],tampon,sizeof(tampon)-1);
49             tampon[n] = '\0';
50             printf("Lu %d caractères de %s :\n",n,nomtube
51             printf("%s\n",tampon);
52         }
53     }
54 }
```

Programme : exempleselect.c (IV)

```
56 int main(void)
57 {
58     int i,n,maxfd=0;
59     mode_t mode;
60     fd_set read;
61     struct timeval tv; /* Cette structure définie dans select.h */
62
63     for(i=1;i<NSIG;i++)
64         signal(i,handler);
65
66     POURTOUTTUBE(i)
67         sprintf(nomtube[i],"%s%d",TUBE,i+1);
68
69     mode = S_IRUSR| S_IWUSR | S_IWGRP | S_IWOTH;
70
71     POURTOUTTUBE(i)
72         if(mkfifo(nomtube[i],mode) == -1 || chmod(nomtube[i],mode) == -1)
73             sortir("mkfifo/chmod",ERR);
74
75     POURTOUTTUBE(i)
76         if((fd[i] = open(nomtube[i],O_RDONLY|O_NONBLOCK)) == -1)
77             sortir("open",ERR);
78     else
79     {
80         maxfd = MAX(fd[i],maxfd);
81         /* Prochaine ligne : rendre la lecture bloquante */
82         fcntl(fd[i],F_SETFL,fcntl(fd[i],F_GETFL) &!O_NONBLOCK);
83     }
```

Programme : exempleselect.c (V)

```
85     for(;;){
86         tv.tv_sec=TIMEOUT; /* Secondes */
87         tv.tv_usec=0; /* Microsecondes */
88         FD_ZERO(&read);
89         POURTOUTTUBE(i)
90             FD_SET(fd[i],&read);
91
92         n = select(maxfd + 1 ,&read,NULL,NULL,&tv);
93         switch(n){
94             case -1 :
95                 sortir("select",ERR);
96             case 0 :
97                 sortir("attente trop longue",OK);
98             default:
99                 traiter(&read);
100         }
101     }
102 }
```