Chapitre 6

Synchronisation des processus

DAns un système d'exploitation multiprogrammé en temps partagé, plusieurs processus s'exécutent en pseudo-parallèle ou en parallèle et partagent des objets (mémoires, imprimantes, etc.). Le partage d'objets sans précaution particulière peut conduire à des résultats imprévisibles. La solution au problème s'appelle **synchronisation des processus**.

6.1 Introduction

Considérez par exemple, le fonctionnement d'un *spool* d'impression montré sur la figure 6.1, qui peut être délicat.

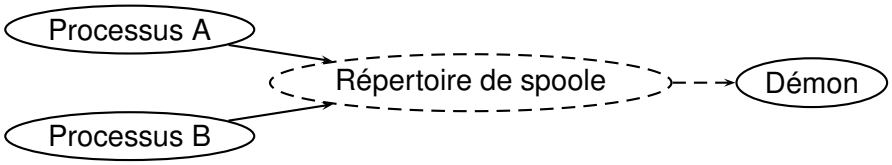

FIG. 6.1 – Spool d'une imprimante.

Quand un processus veut imprimer un fichier, il doit placer le nom de ce fichier dans un répertoire spécial, appelé répertoire de *spool*. Un autre processus, le démon d'impression, vérifie périodiquement s'il y a des fichiers à imprimer. Si c'est le cas, il les imprime et retire leur nom du répertoire. Supposons que :

– Le répertoire d'impression ait un très grand nombre d'emplacements, numérotés 0,1,2,...,N. Chaque emplacement peut contenir le nom d'un fichier à imprimer.

- Tous les processus partagent la variable `in` qui pointe sur le prochain emplacement libre du répertoire.
- Deux processus A et B veulent placer chacun un fichier dans la file d'impression et que la valeur de `in` est 7.
- Le processus A place son fichier à la position `in` qui vaut 7. Une interruption d'horloge arrive immédiatement après et le processus A est suspendu pour laisser place au processus B.
- Ce dernier place également son fichier à la position `in` qui vaut toujours 7 et met à jour `in` qui prend la valeur 8. Il efface ainsi le nom du fichier placé par le processus A.
- Lorsque le processus A sera relancé, il met à jour la valeur de `in` qui prend la valeur 9.

Sous ces conditions, deux problèmes peuvent se présenter :

- Le fichier du processus A ne sera jamais imprimé ;
- **in** ne pointe plus sur le prochain emplacement libre.

Comment éviter ces problèmes ? le problème provient du fait que le processus B a utilisé une variable partagée avant que A ait fini de s'en servir. Si l'on arrive à bien synchroniser les processus, le problème peut être résolu.

Lorsqu'un processus accède à la variable `in`, les autres processus ne doivent ni la lire, ni la modifier jusqu'à ce que le processus ait terminé de la mettre à jour. D'une manière générale, il faut empêcher les autres processus d'accéder à un objet partagé si cet objet est en train d'être utilisé par un processus, ce qu'on appelle d'assurer l'**exclusion mutuelle**. Les situations de ce type, où deux ou plusieurs processus lisent ou écrivent des données partagées et où le résultat dépend de l'ordonnancement des processus, sont qualifiées d'**accès concurrents**.

6.2 Objets et sections critiques

Nous allons utiliser les définitions suivantes :

Objet critique : Objet qui ne peut être accédé simultanément. Comme par exemple, les imprimantes, la mémoire, les fichiers, les variables etc. L'objet critique de l'exemple du *spool* d'impression précédent est la variable partagée `in`.

Section critique : Ensemble de suites 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.

Ainsi, la section critique de l'exemple du *spool* d'impression est :

– Lecture de la variable `in`.
– L'insertion d'un `nom de fichier` dans le répertoire.
– La mise à jour de `in`.

Imaginez maintenant qu'on désire effectuer une somme des N premiers nombres en utilisant des processus légers. Une vision simpliste pour résoudre ce problème, pourrait suggérer de diviser le travail en deux processus : le premier qui va calculer `S1=1+2+...+M` avec `M=N % 2` et un deuxième qui additionne `S2=M+(M+1)+...+N`, et finalement la variable `somme_totale=S1 + S2` fournira la réponse. La figure 6.2 illustre cette situation. Un code possible pour résoudre ce problème est :

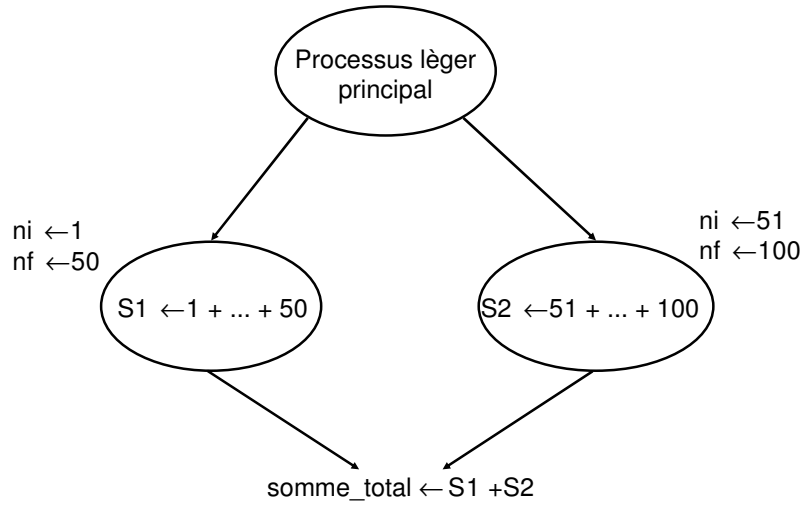

FIG. 6.2 – Somme de N nombres avec threads.

```
int somme_totale = 0;
void somme_partielle(int ni, int nf)
   {
       int j = 0;
       int somme_partielle = 0;
       for (j = ni; j <= nf; j++)
           somme_partielle += j;
       somme_totale += somme_partielle;
       pthread_exit(0);
   }
```

Mais, avec cet algorithme, si plusieurs processus exécutent concurremment ce code, on peut obtenir un résultat incorrect. La solution correcte nécessite l'utilisation des **sections critiques**.

Le problème de conflit d'accès serait résolu, si on pouvait assurer que deux processus ne soient jamais en section critique en même temps au moyen de l'**exclusion mutuelle**). En général, les processus qui exécutent des sections critiques sont structurés comme suit :

- Section non critique.
- Demande d'entrée en section critique.
- Section critique.
- Demande de sortie de la section critique.
- Section non critique.

Quatre conditions sont nécessaires pour réaliser correctement une exclusion mutuelle :

1. Deux processus ne peuvent être en même temps en section critique.
2. Aucune hypothèse ne doit être faite sur les vitesses relatives des processus et sur le nombre de processeurs.
3. Aucun processus suspendu en dehors d'une section critique ne doit bloquer les autres processus.
4. Aucun processus ne doit attendre trop longtemps avant d'entrer en section critique.

Comment assurer alors une coopération correcte et efficace des processus ?

6.3 Masquage des interruptions

Avant d'entrer dans une section critique, le processus **masque les interruptions**. Il les restaure à la fin de la section critique. Il ne peut être alors suspendu durant l'exécution de la section critique.

Cependant cette solution est dangereuse, car si le processus, pour une raison ou pour une autre, ne restaure pas les interruptions à la sortie de la section critique, ce serait la fin du système. La solution n'assure pas l'exclusion mutuelle, si le système n'est pas monoprocesseur car le masquage des interruptions concernera uniquement le processeur qui a demandé l'interdiction. Les autres processus exécutés par un autre processeur pourront donc accéder aux objets partagés.

En revanche, cette technique est parfois utilisée par le système d'exploitation pour mettre à jour des variables ou des listes partagées par ses processus, par exemple la liste des processus prêts.

6.4 Exclusion mutuelle par attente active

6.4.1 Les variables de verrouillage

Une autre tentative pour assurer l'exclusion mutuelle est d'utiliser une variable de verrouillage partagée `verrou`, unique, initialisée à 0. Pour rentrer en section critique (voir algorithme de verrouillage), un processus doit tester la valeur du `verrou`. Si elle est égale à 0, le processus modifie la valeur du `verrou` $\leftarrow 1$ et exécute sa section critique. À la fin de la section critique, il remet le `verrou` à 0. Sinon, il attend (par une attente active) que le `verrou` devienne égal à 0, c'est-à-dire : `while(verrou!=0);`

Algorithme verrouillage

Algorithm 1 Verrouillage

while verrou $\neq 0$ **do**
  ; // Attente active
end while
verrou $\leftarrow 1$
Section_critique();
verrou $\leftarrow 0$

Problèmes Cette méthode n'assure pas l'exclusion mutuelle : Supposons qu'un processus est suspendu juste après avoir lu la valeur du verrou qui est égal à 0. Ensuite, un autre processus est élu. Ce dernier teste le verrou qui est toujours égal à 0, met verrou $\leftarrow 1$ et entre dans sa section critique. Ce processus est suspendu avant de quitter la section critique. Le premier processus est alors réactivé, il entre dans sa section critique et met le verrou $\leftarrow 1$. Les deux processus sont en même temps en section critique.

Ainsi cet algorithme *n'est pas correct*. On voit ainsi qu'il est indispensable dans l'énoncé du problème de préciser quelles sont les **actions atomiques**, c'est-à-dire celles qui sont non divisibles. Si on possède sous forme hardware une instruction **test and set** qui sans être interrompue lit une variable, vérifie si elle est égale à 0 et dans ce cas transforme sa valeur en 1, alors il est facile de résoudre le problème de l'exclusion mutuelle en utilisant un verrou. Toutefois on n'a pas toujours disponible ce type d'instruction, et il faut alors trouver une solution logicielle.

6.4.2 L'alternance

Une autre proposition consiste à utiliser une variable `tour` qui mémorise le tour du processus qui doit entrer en section critique. `tour` est initialisée à `0`.

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

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

Supposons maintenant que le processus P1 lit la valeur de `tour` qui vaut 0 et entre dans sa section critique. Il est suspendu et P2 est exécuté. P2 teste la valeur de `tour` qui est toujours égale à 0. Il entre donc dans une boucle en attendant que `tour` prenne la valeur 1. Il est suspendu et P1 est élu de nouveau. `P1` quitte sa section critique, met `tour` à `1` et entame sa section non critique. Il est suspendu et P2 est exécuté. P2 exécute rapidement sa section critique, `tour = 0` et sa section non critique. Il teste `tour` qui vaut 0. Il attend que `tour` prenne la valeur 1.

Problème : On peut vérifier assez facilement que deux processus ne peuvent entrer en section critique en même temps, toutefois le problème n'est pas vraiment résolu car il est possible qu'un des deux processus ait plus souvent besoin d'entrer en section critique que l'autre ; l'algorithme lui fera attendre son tour bien que la section critique ne soit pas utilisée. Un processus peut être bloqué par un processus qui n'est pas en section critique.

6.4.3 Solution de Peterson

La solution de Peterson se base sur l'utilisation de deux fonctions : `entrer_region();quitter_region();`
Chaque processus doit, avant d'entrer dans sa section critique appeler la fonction `entrer_region()` en lui fournissant en paramètre son numéro de processus. Cet appel le fera attendre si nécessaire jusqu'à ce qu'il n'y ait plus de risque. A la fin de la section critique, le processus doit appeler `quitter_region()` pour indiquer qu'il quitte sa section critique et pour autoriser l'accès aux autres processus, comme le montre le code `peterson.c`.

Listing 6.1 – peterson.c

```
#define FALSE 0
#define TRUE  1
#define N     2     // Nb. processus

int  tour;          // Le tour du processus
int interesse[N];   // 0 initiale

void entrer_region(int processus)   // processus 0 ou 1
{
    int autre ;     // Autre processus
    autre = 1 - processus ;
    interesse[processus] = TRUE ; // On est éinteress
    tour = processus ;  // Positioner le drapeau
                        // Attendre
    while (tour == processus && interesse[autre] ==TRUE) ;
}

void quitter_region (int processus)
{
    // Processus quite region critique
    interesse[processus] = FALSE;
}
```

Prouver la correction de cet algorithme est équivalent à montrer le fait que deux processus ne peuvent pas entrer en section critique en même temps. Pour y arriver, nous devons réécrire le code, comme montré dans `peterson2.c`:

Listing 6.2 – peterson2.c

```
#define FALSE 0
#define TRUE  1
```

```
#define N     2     // Nb. processus

int  tour;          // Le tour du processus
int interesse[N];   // 0 initiale

void EntrerSortirRegion(processus)
 {
   autre = 1 - processus;
   interesse[processus] = TRUE;
   tour = processus;
   while ((tour == i) && interesse[autre]) ;
   Section_critique();
   interesse[processus] = FALSE;
 }
```

Pour vérifier que l'algorithme fonctionne correctement et donne accès aux deux processus à la région critique, il faut décomposer cet algorithme en actions atomiques. Voici celles pour le processus `P0` :

```
1. Écrire (true, interesse[0]);
2. Écrire (0, tour);
3. a0 = Lire (interesse[1]);
4. Si (!a0) aller à 7;
5. t0 = Lire(tour);
6. Si (t0 == 0) aller à 3;
7. Section_critique();
8. Écrire (false, interesse[0]);
9. Aller à 1;
```

Les actions atomiques de `P1` sont tout à fait semblables :

```
1'. Écrire (true, interesse[1]);
2'. Écrire (1, tour);
3'. a1 = Lire (interesse[0]);
4'. Si (!a1) aller à 7';
5'. t1 = Lire(tour);
6'. Si (t1 == 1) aller à 3';
7'. Section_critique();
8'. Écrire (false, interesse[1]);
9'. Aller à 1';
```

Il faut examiner tous les enchevêtrements possibles des deux suites d'actions atomiques. On peut pour des raisons de symétrie supposer que le processus `P0` exécute sa première instruction avant `P1`. Il faut toutefois examiner plusieurs cas par la suite :

Cas 1 : `P0` exécute `3` avant que `P1` n'exécute `1'` les valeurs obtenues par les deux processus sont alors : `a0 = false, t0 = 0, a1 = true, t1 = 1 ;` Dans ce cas on vérifie que `P0` entre en section critique, `P1` reste bloqué.

Cas 2 : P1 exécute 1' avant que `P0` n'exécute `3`, et `2'` après que `P0` ait exécuté `5`

`a0 = true, t0 = 0, a1 = true , t1 = 1 ;`

On constate qu'aucun des deux processus ne rentre dans la section critique, toutefois immédiatement après le processus `P0` obtient :

`a0 = true, t0 = 1`, ainsi `P0` entre en section critique, `P1` bloqué

Cas 3 : `P1` exécute `1'` avant que `P0` n'exécute `3`, et `2'` entre `2` et `5`. On a dans ce cas : `a0 = true, t0 = 1, a1 = true , t1 = 1 ;` ainsi `P0` entre en section critique, `P1` reste bloqué

Cas 4 : `P1` exécute 1' avant que `P0` n'exécute `2`, et `2'` avant `2`, donc `a0 = true, t0 = 0, a1 = true , t1 = 0 ;`

`P1` entre en section critique et `P0` reste bloqué

Ayant démontré la correction de l'algorithme, il faut aussi se poser la question de savoir si deux processus ne peuvent rester bloqués indéfiniment en attendant la section critique, on dirait alors qu'il y a interblocage. Des raisonnements analogues à ceux faits pour la correction montrent qu'il n'en est rien. Cependant, l'algorithme marche seulement pour deux processus.

6.4.4 L'instruction TSL

Les ordinateurs multiprocesseurs ont une instruction atomique indivisible appelée **TSL** (*Test and Set Lock*).

```
tsl registre, verrou
```

L'instruction **TSL** exécute en un seul cycle, de manière indivisible, le chargement d'un mot mémoire dans un registre et le rangement d'une valeur non nulle à l'adresse du mot chargé. Lorsqu'un processeur exécute l'instruction **TSL**, il verrouille le bus de données pour empêcher les autres processeurs d'accéder à la mémoire pendant la durée de l'opération. Cette instruction peut être utilisée pour établir et supprimer des verrous (assurer l'exclusion mutuelle).

```
entrer_region
  Label :
    tsl registre, verrou # Copier verrou dans reg.
                         # verrou = 1
    cmp registre, $0     # verrou = 0 ?
    jnz Label            # si different on boucle (verrou)
    ret                  # ret. entrer section critique

quitter_region
     mov verrou, $0      # mettre 0 dans verrou
     ret                 # quitter section critique
```

6.4.5 Commentaires sur l'attente active

L'attente active se traduit dans une consommation du temps UCT, et un processus en attente active peut rester en permanence dans une boucle infinie. Par exemple, supposons que :

- Deux processus H et B, tels que H est plus prioritaire que B partagent un objet commun. Les règles d'ordonnancement font que H sera exécuté dès qu'il se trouve dans l'état prêt.
- Pendant que H était en E/S, le processus B est entré dans sa section critique.
- Le processus H est redevenu prêt avant que B ait quitté sa section critique.
- Le processus B est ensuite suspendu et le processus H est élu. H effectue alors une attente active et B ne peut être sélectionné puisque H est plus prioritaire.

Conclusion ? le processus B ne peut plus sortir de sa section critique et H boucle indéfiniment.

Une dernière question est de vérifier qu'il y a un minimum d'équité entre les deux processus, c'est-à-dire que l'un d'entre eux ne peut pas attendre indéfiniment son tour d'accès à la section critique alors que l'autre y accède un nombre infini de fois. Cette question est plus difficile, de même que la généralisation de la solution à un nombre quelconque de processus.

6.5 Primitives `SLEEP` et `WAKEUP`

`SLEEP()` est un appel système qui suspend l'appelant en attendant qu'un autre le réveille. `WAKEUP(processus)` est un appel système qui réveille un `processus`. Par exemple, un processus H qui veut entrer dans

sa section critique est suspendu si un autre processus B est déjà dans sa section critique. Le processus H sera réveillé par le processus B, lorsqu'il quitte la 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 le destinataire ne soit endormi, le processus peut dormir pour toujours.

Exemples

```
// Processus P1
 {
    if(cond ==0 ) SLEEP() ;
    cond = 0 ;
    section_critique ( ) ;
    cond = 1 ;
    WAKEUP(P2) ;
    ...
 }

// Processus P2
{
    if (cond ==0 ) SLEEP() ;
    cond = 0 ;
    section_critique ( ) ;
    cond = 1 ;
    WAKEUP(P1) ;
    ...
}
```

Cas 1 : Supposons que :

- P1 est exécuté et il est suspendu juste après avoir lu la valeur de `cond` qui est égale à `1`.
- P2 est élu et exécuté. Il entre dans sa section critique et est suspendu avant de la quitter.
- P1 est ensuite exécuté. Il entre dans sa section critique.
- Les deux processus sont dans leurs sections critiques.

Cas 2 : Supposons que :

- P1 est exécuté en premier. Il entre dans sa section critique. Il est suspendu avant de la quitter ;
- P2 est élu. Il lit `cond` qui est égale à `0` et est suspendu avant de la tester.

- P1 est élu de nouveau. Il quitte la section critique, modifie la valeur de `cond` et envoie un signal au processus P2.
- Lorsque P2 est élu de nouveau, comme il n'est pas endormi, il ignorera le signal et va s'endormir pour toujours.

Dans le système Unix, les appels système qui assurent à peu près, les mêmes fonctions que les primitives `SLEEP` et `WAKEUP` sont :

- `pause( )` : suspend le processus appelant jusqu'à réception d'un signal.
- `kill(pid, SIGCONT)` : envoie le signal `SIGCONT` au processus `pid`.

L'émission d'un signal est possible si l'une des deux conditions est vérifiée :

1. L'émetteur et le destinataire du signal appartiennent au même propriétaire
2. L'émetteur du signal est le super-utilisateur.

6.6 Problème du producteur et du consommateur

Dans le problème du **producteur** et du **consommateur** deux processus partagent une mémoire tampon de taille fixe, comme montré à la figure 6.3. L'un d'entre eux, 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 si le tampon 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.

▶ **Exemple 1.** Une solution classique au problème du producteur et du consommateur [**?**] est montré sur le programme `schema-prod-cons.c` avec l'utilisation des primitives `SLEEP` et `WAKEUP`.

Listing 6.3 – schema-prod-cons.c

```
#define TRUE 1
#define N 50          // Nb. de places dans le tampon
int compteur = 0;     // Nb. objets dans le tampon

void producteur( )
{
 while(TRUE)
    {
        produire_objet();
```

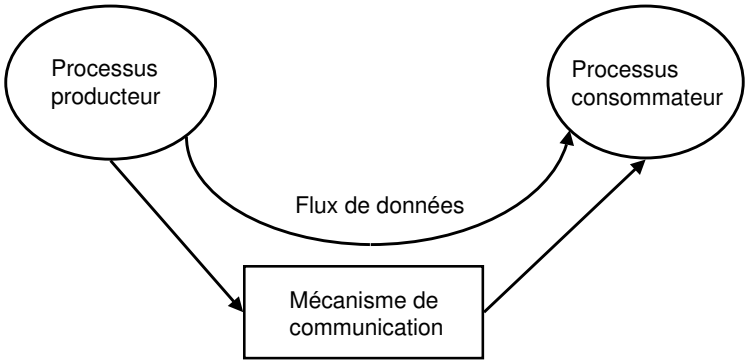

FIG. 6.3 – Problème du producteur/consommateur.

```
10          if(compteur == N)
               SLEEP();                 // Tampon plein
            mettre_objet();             // dans tampon
            compteur++;
            if(compteur == 1)           // Tampon vide
               WAKEUP(consomateur);
        }
    }

    void consomateur( )
20  {
     while(TRUE)
        {
            if(!compteur)
               SLEEP();                 // tampon vide
            retirer_objet();            // du tampon
            compteur--;
            if(compteur == N-1)         // érveiller producteur
               WAKEUP(producteur);
            consommer_objet();
30      }
    }
```

6.6.1 Critique des solutions précédentes

La généralisation de toutes ces solutions aux cas de plusieurs processus est bien complexe. Le mélange d'exclusion mutuelle et de suspension est toujours délicat à réaliser et à implanter.

6.7 Sémaphores

Pour contrôler les accès à un objet partagé, E. W. Dijkstra suggéra en 1965 l'emploi d'un nouveau type de variables appelées 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 `semaphore S = 10`.

Les sémaphores sont manipulés au moyen des opérations `P` ou wait et `V` ou signal. L'opération `P(S)` décrémente la valeur du sémaphore `S` si cette dernière est supérieure 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.

Les sémaphores permettent de réaliser des **exclusions mutuelles**, comme illustré à la figure 6.4.

semaphore mutex=1 ;		
processus P1 : P(mutex) section critique de P1 ; V(mutex) ;		processus P2 ; P(mutex) section critique de P2 ; V(mutex) ;

FIG. 6.4 – Exclusion mutuelle avec des sémaphores.

La figure 6.5 montre l'exclusion mutuelle d'une petite section critique `a++`.

int a ; semaphore mutex=1 ;		
processus P1 : P(mutex) a++ ; V(mutex) ;		processus P2 ; P(mutex) a++ ; V(mutex) ;

FIG. 6.5 – Exclusion mutuelle d'une section critique composée d'une variable a.

Sur la figure 6.6 on montre la valeur que les sémaphores prennent lors des appels des `P (wait)` et `V (signal)`, pour trois processus P_0, P_1 et

P_2.

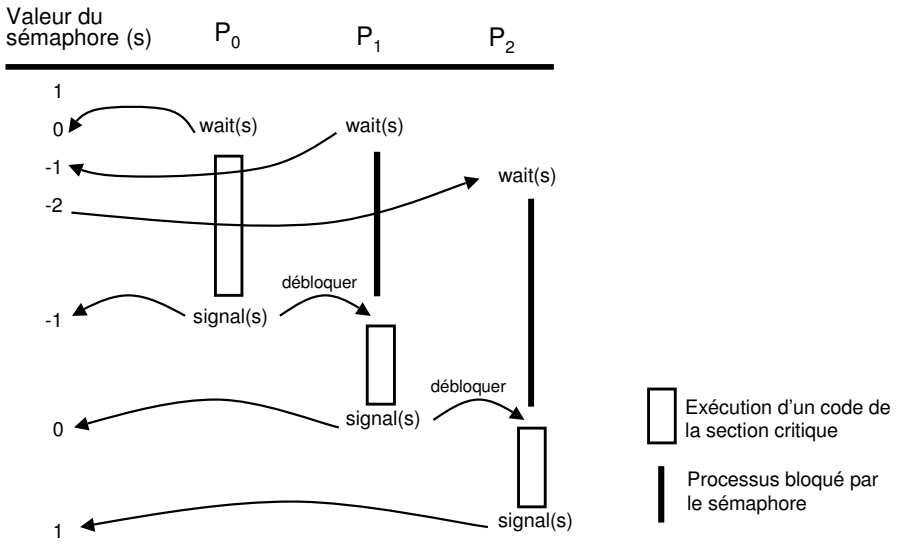

FIG. 6.6 – Valeurs de sémaphores lors des appels V (signal) et P (wait).

6.8 Services Posix sur les sémaphores

Le système d'exploitation Linux permet de créer et d'utiliser les **sémaphores** définis par le standard Posix. Les services Posix de manipulation des sémaphores se trouvent dans la librairie `<semaphore.h>`. Le type sémaphore est désigné par le type `sem_t`.

- `int sem_init(sem_t *sem, int pshared, unsigned int valeur)`
  Initialisation d'un sémaphore. `sem` est un pointeur sur le sémaphore à initialiser ; `value` est la valeur initiale du sémaphore ; `pshared` indique si le sémaphore est local au processus `pshared=0` ou partagé entre le père et le fils `pshared` $\neq$ `0`. Actuellement Linux ne supporte pas les sémaphores partagés.
- `int sem_wait(sem_t *sem)` est équivalente à l'opération `P` : retourne toujours `0`.
- `int sem_post(sem_t *sem)` est équivalente à l'opération `V` : retourne `0` en cas de succès ou `-1` autrement.

- `int sem_trywait(sem_t *sem)` décrémente la valeur du sémaphore `sem` si sa valeur est supérieure à `0` ; sinon elle retourne une erreur. C'est une opération atomique.
- `int sem_getvalue (sem_t* nom, int * sval)` ; récupérer la valeur d'un sémaphore : il retourne toujours `0`.
- `int sem_destroy (sem_t* nom)` ; détruire un sémaphore. Retourne `-1` s'il y a encore des `waits`.

Les sémaphores Posix servent donc à synchroniser les *communications entre threads*. Les sémaphores d'Unix **System V** sont un autre type de mécanisme qui permet la synchronisation entre processus. Sa gestion étant un peu plus complexe, on verra ce mécanisme dans le Chapitre **??**, *Communications IPC System V*.

▶ **Exemple 2.** Le programme `sem-posix.c` `sem-posix.c` montre l'utilisation de sémaphores Posix.

Listing 6.4 – sem-posix.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <unistd.h>
#include <semaphore.h>

sem_t my_sem;
#define SEMINITVALUE 1
void *critical_section(void *);
#define MAXTHREADS 100

int main(int argc, char *argv[])
{
   int n, i, error;
   pthread_t threadid[MAXTHREADS];
   if ((argc != 2) || ((n = atoi(argv[1])) <= 0))
   {
       printf("Usage: %s nombre de threads\n", argv[0]);
       exit(1);
   }
   // initialiser le ésmaphore
   sem_init(&my_sem, 0, SEMINITVALUE);
   // écration des threads
   for (i = 0; i < n; ++i)
     pthread_create(&threadid[i], NULL, critical_section, (void *)i);
   // attendre la fin des threads
   for (i = 0; i < n; ++i)
```

```
        pthread_join(threadid[i], NULL);
 exit(0);
}

void *critical_section(void *arg)
{
   int myindex, i, n;
   // extraire l'index des threads
   myindex = (int)arg;
   sem_wait(&my_sem);
   // section critique
   for (i=1; i < 5; i++)
   {
       printf("%d : %d\n", myindex, i);
       if ((n=sleep(1)) != 0) // dormir 1 seconde
       printf("interrupted, no of secs left %d\n",n);
   }
   // quiter section critique
   sem_post(&my_sem);
   pthread_exit(NULL);
}
```

Sortie du programme `posix-sem.c` :

```
leibnitz> gcc -o posix-sem posix-sem.c -lpthread
leibnitz> posix-sem 2
0 : 1
0 : 2
0 : 3
0 : 4
1 : 1
1 : 2
1 : 3
1 : 4
leibnitz>
```

▶ **Exemple 3.** Producteur/consommateur avec sémaphores. La solution du problème au moyen des sémaphores Posix utilise trois sémaphores dans le programme `prod-cons.cpp`. Le premier sémaphore, nommé `plein`, compte le nombre d'emplacements occupés. Il est initialisé à `0`. Le second, nommé `vide`, compte le nombre d'emplacements libres. Il est initialisé à `N` (la taille du tampon). Le dernier, nommé `mutex`, assure l'exclusion mutuelle pour l'accès au tampon (lorsqu'un processus utilise le tampon, les autres ne peuvent pas y accéder).

Listing 6.5 – prod-cons.cpp

```
#include <semaphore.h>
#include <unistd.h> //sleep
#include <pthread.h>
#include <stdio.h>

#define taille 3
sem_t plein, vide, mutex;
int tampon[taille];
pthread_t cons,prod;
void* consommateur(void *);
void* producteur(void *);

int main()
{
    // initiliser les ésmaphores
    sem_init(&plein,0,0);
    sem_init(&vide,0, taille);
    sem_init(&mutex,0, 1);
    //écrer les threads
    pthread_create(&cons,NULL,consommateur,NULL);
    pthread_create(&prod,NULL,producer,NULL);
    // attendre la fin des threads
    pthread_join(prod,NULL);
    pthread_join(cons,NULL);
    printf("fin des  thread \n");
    return 0;
}

void* consommateur(void *)
{
    int ic=0, nbcons = 0, objet;
    do {
         sem_wait(&plein);
         sem_wait(&mutex);
        // consommer
         objet = tampon[ic];
         sem_post(&mutex);
         sem_post(&vide);
         printf("ici cons. :tampon[%d]= %d\n", ic, objet);
         ic=(ic+1)%taille;
         nbcons++;
         sleep(2);
    } while ( nbcons<=5 );
    return (NULL);
}

void* producteur(void *)
```

```
{
   int ip=0, nbprod=0, objet=0;
   do {
         sem_wait(&vide);
         sem_wait(&mutex);
        // produire
         tampon[ip]=objet;
         sem_post(&mutex);
         sem_post(&plein);
         printf("ici prod. : tampon[%d]= %d\n", ip,objet);
         objet++;
         nbprod++;
         ip=(ip+1)%taille;
    } while ( nbprod<=5 );
    return NULL;
}
```

Sortie du programme `prod-cons.cpp` :

```
leibnitz> g++ -o prod-cons prod-cons.cpp -lpthread
leibnitz> prod-cons
ici cons. :tampon[0]= 0
ici prod. : tampon[0]= 0
ici prod. : tampon[1]= 1
ici prod. : tampon[2]= 2
ici prod. : tampon[0]= 3
ici cons. :tampon[1]= 1
ici prod. : tampon[1]= 4
ici cons. :tampon[2]= 2
ici prod. : tampon[2]= 5
ici cons. :tampon[0]= 3
ici cons. :tampon[1]= 4
ici cons. :tampon[2]= 5
fin des  thread
leibnitz>
```

6.8.1 Problème des philosophes

Il s'agit d'un problème théorique très interessant propossé et résolu aussi par Dijkstra. Cinq philosophes sont assis autour d'une table. Sur la table, il y a alternativement cinq plats de spaghettis et cinq fourchettes (figure 6.7). Un philosophe passe son temps à manger et à penser. Pour manger son plat de spaghettis, un philosophe a besoin de deux fourchettes qui sont de part et d'autre de son plat. Écrire un programme qui permette à

chaque philosophe de se livrer à ses activités (penser et manger) sans jamais être bloqué.

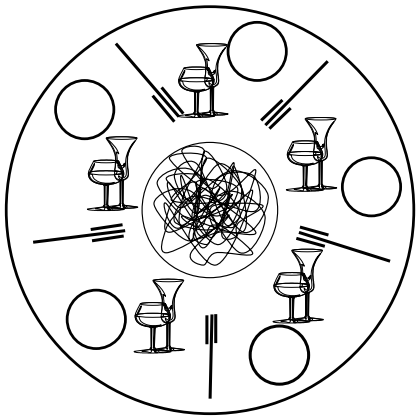

FIG. 6.7 – Problème des philosophes.

Il est important de signaler que si tous les philosophes prennent *en même temps* chacun une fourchette, aucun d'entre eux ne pourra prendre l'autre fourchette (situation d'interblocage). Pour éviter cette situation, un philosophe ne prend jamais une seule fourchette. Les fourchettes sont les objets partagés. L'accès et l'utilisation d'une fourchette doit se faire en exclusion mutuelle. On utilisera le sémaphore `mutex` pour réaliser l'exclusion mutuelle.

▶ **Exemple 4.** Une solution au problème des philosophes.

Listing 6.6 – philosophe.c

```
#include <stdio.h>
#include <unistd.h>     // pour sleep
#include <pthread.h>    // pour les threads
#include <semaphore.h> // pour les ésmaphores

#define N 5             // nombre de philosophes
#define G ((N+i-1)%N)   // philosophe de gauche de i
#define D   (i)         // philosophe de droite de i
#define  libre 1
#define occupe 0
int fourch[N] = {libre, libre, libre, libre, libre};
sem_t mutex ;
void * philosophe(void * ) ;
```

```
int main()
{
   int NumPhi[N] = {0,1,2,3,4};
   int i;
   pthread_t ph[N];
   sem_init(&mutex,0, 1) ;
   // écration des N philosophes
   for(i=0; i<N; i++)
        pthread_create(&ph[i],NULL,philosophe,&(NumPhi[i]));
   // attendre la fin des threads
   i=0;
   while(i<N && (pthread_join(ph[i++],NULL)==0));
   printf("fin des  threads \n");
   return 0;
}

// La fonction de chaque philosophe
void * philosophe(void *num)
{
   int i =*(int *) num;
   int nb = 2;
   while (nb)
   {
        // penser
           sleep(1);
        // essayer de prendre les fourchettes pour manger
           sem_wait(&mutex) ;
           if(fourch[G] && fourch[i])
           {
                  fourch[G] = 0 ;
                  fourch[i] =0 ;
                  printf("philosophe [%d] mange \n", i);
                  sem_post(&mutex) ;
                  nb--;
                 // manger
                  sleep (1) ;
                 // élibrer les fourchettes
                  sem_wait(&mutex) ;
                  fourch[G] = 1 ;
                  fourch[i] = 1 ;
                  printf("philosophe[%d] a fini de manger\n",i);
                  sem_post(&mutex) ;
           }
           else sem_post(&mutex);
    }
}
```

Problème : Cette solution résout le problème d'interblocage. Mais, un phi-

losophe peut mourir de faim car il ne pourra jamais obtenir les fourchettes nécessaires pour manger (problème de famine et d'équité).

Pour éviter le problème de famine il faut garantir que si un processus demande d'entrer en section critique, il obtient satisfaction au bout d'un temps fini. Dans le cas des philosophes, le problème de famine peut être évité, en ajoutant `N` sémaphores (un sémaphore pour chaque philosophe). Les sémaphores sont initialisés à `0`. Lorsqu'un philosophe `i` ne parvient pas à prendre les fourchettes, il se met en attente (`P(S[i])`). Lorsqu'un philosophe termine de manger, il vérifie si ses voisins sont en attente. Si c'est le cas, il réveille les voisins qui peuvent manger en appelant l'opération `V`. On distingue trois états pour les philosophes : penser, manger et faim.

▶ **Exemple 5.** Une meilleure solution au problème des philosophes.

Listing 6.7 – philosophe2.c

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

#define N 5
#define G ((N+i-1)%N)     // philosophe de gauche de i
#define D (i)             // philosophe de droite de i
enum etat { penser, faim, manger} ;
etat Etat[N] = {penser, penser, penser, penser, penser} ;
sem_t S[N] ;
sem_t mutex ;
void * philosophe(void * );
void Test(int i);

int main()
{
  int i, NumPhi[N] = {0,1,2,3,4};
  pthread_t ph[N];
  sem_init(&mutex, 0, 1) ;
  for(i=0 ; i<N ; i++)
     sem_init(&S[i],0,0) ;
  // écration des N philosophes
  for(i=0; i<N; i++)
     pthread_create(&ph[i],NULL,philosophe,&(NumPhi[i]));
  // attendre la fin des threads
  for(i=0; (i<N && pthread_join(ph[i],NULL)==0); i++);
  printf("fin des  threads \n");
  return 0;
}
```

```
void * philosophe( void * num)
{
  int i =* (int *) num ;
  int     nb = 2 ;
  while (nb)
  {
    nb--;
    // penser
    sleep( 1) ;
    // tenter de manger
    sem_wait(&mutex) ;
    Etat[i]=faim ;
    Test(i) ;
    sem_post(&mutex) ;
    sem_wait(&S[i]) ;
    // manger
    printf ("philosophe[%d] mange\n", i) ;
    sleep (1) ;  // manger
    printf("philosophe[%d] a fini de manger\n", i);
    // élibrer les fourchettes
    sem_wait(&mutex) ;
    Etat[i] = penser ;
    // évrifier si ses voisins peuvent manger
    Test(G) ;
    Test(D) ;
    sem_post(&mutex) ;
   }
}
// procedure qui évrifie si le philosophe i peut manger
void Test(int i)
{
  if( (Etat[i] == faim) && ( Etat[G] != manger)
      && (Etat[D] != manger) )
  {
      Etat[i] = manger ;
      sem_post(&S[i]) ;
  }
}
```

Exécution de `philosophe2.c`

```
leibnitz> g++ -o philosophe2 philosophe2.c -lpthread
leibnitz> philosophe2
philosophe[0] mange
philosophe[2] mange
philosophe[4] mange
philosophe[0] a fini de manger
```

```
philosophe[2] a fini de manger
philosophe[1] mange
philosophe[4] a fini de manger
philosophe[3] mange
philosophe[0] mange
philosophe[1] a fini de manger
philosophe[2] mange
philosophe[3] a fini de manger
philosophe[4] mange
philosophe[0] a fini de manger
philosophe[2] a fini de manger
philosophe[1] mange
philosophe[3] mange
philosophe[4] a fini de manger
philosophe[1] a fini de manger
philosophe[3] a fini de manger
fin des  threads
leibnitz>
```

6.8.2 Problème des rédacteurs et des lecteurs

Ce problème modélise les accès à une base de données. Un ensemble de processus tente constamment d'accéder à la base de données soit pour écrire, soit pour lire des informations. Pour assurer une certaine cohérence des données de la base, il faut interdire l'accès (en lecture et en écriture) à tous les processus, si un processus est en train de modifier la base (accède à la base en mode écriture). Par contre, plusieurs processus peuvent accéder à la base, en même temps, en mode lecture. Les rédacteurs représentent les processus qui demandent des accès en écriture à la base de données. Les lecteurs représentent les processus qui demandent des accès en lecture à la base de données (figure 6.8).

Pour contrôler les accès à la base, on a besoin de connaître le nombre de lecteurs (`NbL`) qui sont en train de lire. Le compteur `NbL` est un objet partagé par tous les lecteurs. L'accès à ce compteur doit être exclusif (sémaphore mutex). Un lecteur peut accéder à la base, s'il y a déjà un lecteur qui accède à la base (`NbL>0`) ou aucun rédacteur n'est en train d'utiliser la base. Un rédacteur peut accéder à la base, si elle n'est pas utilisée par les autres (un accès exclusif à la base). Pour assurer cet accès exclusif, on utilise un autre sémaphore : `Redact`. Des algorithmes adéquats pour le `lecteur()` et le `redacteur()` sont les suivants :

NbL $\leftarrow$ 0 // Nb Lecteurs

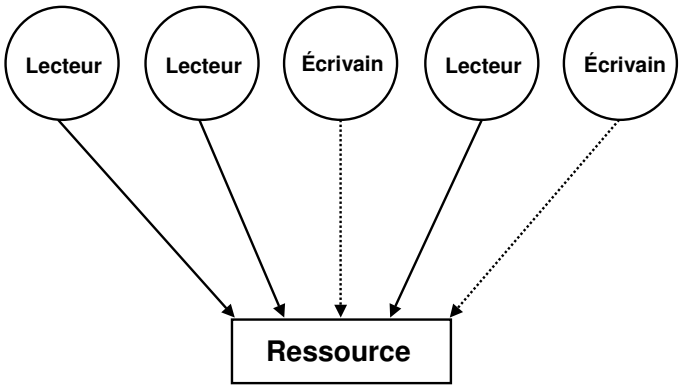

FIG. 6.8 – Problème des lecteurs et écrivains.

```
semaphore Redact // écrivain
semaphore Mutex // accès à la base
lecteur :
repeat
   P(Mutex)
   if (NbL == 0) then
      P(Redact) // Premier lecteur empêche écriture
   end if
   NbL ← NbL + 1
   V(Mutex)
   // lecture de la base
   // fin de l'accès à la base
   P(Mutex)
   NbL ← NbL - 1
   if (NbL == 0) then
      V(Redact) // Dernier lecteur habilite l'écriture
   end if
   V(Mutex)
until TRUE
redacteur :
repeat
   P(Redact)
   // modifier les données de la base
   V(Redact)
until TRUE
```

▶ **Exemple 6.** Le programme `red-lec.c` fait l'implantation de l'algorithme des rédacteurs et lecteurs. Il fixe le nombre de lecteurs et rédacteurs à `N=3`, ainsi que le nombre d'accès (afin d'éviter une boucle infinie) à `ACCES=4`. On a simulé des opérations de lecture de la base, des modifications des données et d'écriture dans la base avec des `sleep()` d'une seconde.

Listing 6.8 – red-lec.c

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

#define N 3
#define ACCES 4
int NbL = 0;
sem_t Redact;
sem_t mutex ;
void * lecteur(void *);
void * redacteur(void *);

int main()
{
   int i;
   int numred[N] = {0,1,2};
   int numlec[N] = {0,1,2};
   pthread_t red[N];
   pthread_t lec[N];
   sem_init(&mutex, 0,1);
   sem_init(&Redact,0,1);
   for( i=0; i<N; i++)
    {
   // écration des redacteurs
       pthread_create(&red[i],NULL,lecteur,&(numred[i]));
   // écration des lecteurs
       pthread_create(&lec[i],NULL,redacteur,&(numlec[i]));
    }
   // attendre la fin des threads
   for(i=0; i<N; i++)
   {
       pthread_join(red[i],NULL);
       pthread_join(lec[i],NULL);
   }
   printf("fin des threads\n");
   return 0;
}
```

```
40  void * lecteur(void * num)
    {
      int i =* (int *) num ;
      int x=ACCES;
      do
      {
          printf("Lecteur %d\n",i);
          sem_wait(&mutex);
          if(!NbL)
              sem_wait(&Redact);
50        NbL++;
          sem_post(&mutex);
          // lecture de la base
          sleep(1);
          // fin de l'èaccsà  la base
          sem_wait(&mutex);
          NbL--;
          if(!NbL)
              sem_post(&Redact);
          sem_post(&mutex);
60        // traitement des édonnes lues
          sleep(1);
      } while(--x);
    }

    void * redacteur(void * num)
    {
       int i =* (int *) num ;
       int x=ACCES;
       do
70     {
          printf("Redacteur %d\n",i);
          sem_wait(&Redact);
          // modifier les édonnes de la base
          sleep(1);
          sem_post(&Redact);
       } while(--x);
    }
```

Nous montrons ensuite une exécution de `red-lec.c` :

```
leibnitz> gcc -o red-lec red-lec.c -lpthread ; red-lec
Lecteur 0
Redacteur 0
Lecteur 1
Redacteur 1
Lecteur 2
Redacteur 2
```

```
Redacteur 0
Lecteur 0
Lecteur 1
Lecteur 2
Redacteur 1
Redacteur 2
Redacteur 0
Lecteur 0
Lecteur 1
Redacteur 1
Lecteur 2
Redacteur 2
Redacteur 0
Lecteur 0
Lecteur 1
Redacteur 1
Lecteur 2
Redacteur 2
fin des threads
leibnitz>
```

6.9 Compteurs d'événements

Un **compteur d'événements** est un compteur associé à un événement. Sa valeur indique le nombre d'occurrences de l'événement associé. Il peut être utilisé pour synchroniser des processus. Par exemple, pour réaliser un traitement un processus attend jusqu'à ce qu'un compteur d'événement atteigne une valeur. Trois opérations sont définies pour un compteur d'événement `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`

6.9.1 Producteur/consommateur

Pour le problème du producteur et du consommateur, nous pouvons utiliser deux compteurs d'événements `in` et `out`. Le premier compte le nombre d'insertions dans le tampon alors que le second compte le nombre de retraits du tampon depuis le début.

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

out : est incrémentée de `1` par le consommateur après chaque retrait du tampon.

Le producteur doit s'arrêter de produire lorsque le tampon est plein. La condition à satisfaire pour pouvoir produire est : `N > in - out`. Posons `sequence = in + 1`. La condition devient alors :
`out` $\geq$ `sequence - N`
De même, le consommateur peut consommer uniquement si le tampon n'est pas vide. La condition à satisfaire pour pouvoir consommer est : `in - out > 0`. Posons `sequence = out + 1`. La condition devient alors :
`in` $\geq$ `sequence`.

```
// 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) ;
    }
 }

 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(" objet consommé [%d]:%d ", pos, objet) ;
        pos = (pos+1)%N ;
    }
}
```

6.10 Moniteurs

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. C'est le compilateur qui se charge de cette tâche. Pour ce faire, il rajoute, au début de chaque procédure du moniteur un code qui réalise ce qui suit : s'il y a 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. Mais, malheureusement, à l'exception de JAVA, la majorité des compilateurs utilisés actuellement ne supportent pas les moniteurs.

6.10.1 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
 boolc nplein, nvide ;
 int compteur =0, ic=0, ip=0 ;
```

```
  // section critique pour le dépôt
  void mettre (int objet)
  {
    if (compteur==N) wait(nplein) ;
    //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(nplein) ;
  }
}
```

6.11 Échanges de messages

La synchronisation entre processus peut être réalisée au moyen d'**échange de messages**, dans Unix System V. Un processus peut attendre qu'un autre lui envoie un message.

L'avantage principale est qu'il n'y a pas de partage d'espace de données, et donc il n'y aura pas de conflit d'accès. Comme problèmes, on peut signaler que les messages envoyés peuvent se perdre. Lorsqu'un émetteur envoie un message, il attend pendant un certain délai la confirmation de réception du message par le récepteur (un acquittement). Si, au bout de ce délai, il ne reçoit pas l'acquittement, il renvoie le message. Il peut y avoir des messages en double en cas de perte du message d'acquittement. Pour éviter cela, on associe à chaque message une identification unique. L'authentification des messages se fait alors par l'utilisation des clés.

Le mecanisme detaillé d'échange de messages sera étudié dans le Chapitre **??**, *Communications IPC System V*.

6.12 Producteur/consommateur par tubes

Le producteur et le consommateur communiquent au moyen d'un tube de communication nommé. Le producteur dépose ses objets un à un dans le tube. Il se bloque lorsqu'il tente de déposer un objet et le tube n'a pas suffisamment de place. Il est réveillé quand le tube a suffisamment d'espace libre. De son côté, le consommateur récupère un à un des objets du tube. Il se bloque lorsqu'il tente de récupérer un objet à partir d'un tube vide. Il est réveillé lorsque le tube devient non vide. Le schéma de programme `p-c-pipe.c` montre l'utilisation de cette technique.

Listing 6.9 – p-c-pipe.c

```
#include <stdio.h>
#include <unistd.h>

int main(void)
{
 int fildes[2];   // pipe pour synchroniser
 char c;          // èscaractre pour synchroniser

 pipe(fildes);
 write(fildes[1], &c, 1);  // éncessaire pour entrer dans la
                           // section critique la èpremire fois
  if (fork() == 0)         // processus fils
  {
    for(;;)
    {
         read(fildes[0], &c, 1);  // éentre section critique
         // < Section critique >
         write(fildes[1], &c, 1);  // quiter section critique
    }
  }
  else  // processus èpre
  {
    for(;;)
    {
         read(fildes[0], &c, 1);   // éentre section critique
          // < section critique >
         write(fildes[1], &c, 1);  // quitter section critique
    }
  }
 return 0;
}
```

6.13 Exclusion mutuelle (mutex) de threads

Une synchronisation correcte entre plusieurs fils d'exécution est très importante. Pour accéder à des données globales partagées, il est indispensable de mettre en œuvre un mechanisme permettant de protèger une variable partagée par plusieurs threads. Ce mécanisme est appelé `mutex` (*MUTual EXclusion*) et repose sur l'utilisation du type `pthread_mutex_t`. Ainsi une variable sert de verrou à une zone de mémoire globale particulière. Les appels d'exclusion mutuelle de threads consistent notamment en l'initialisation, le verrouillage, le déverrouillage et la destruction.

Initialisation

On peut initialiser un mutex statique ou dynamique. Ainsi on peut initialiser un mutex statiquement comme :

```
phtread_mutex_t mutex = PHTREAD_MUTEX_INITIALIZER;
```

ou en utilisant l'initialisation dynamique avec l'appel système :

```
int pthread_mutex_init(pthread_mutex_t *mutex, const
pthread_mutexattr_t *attributs);
```

Avec les déclarations :

```
phtread_mutex_t mutex;
phtread_mutexattr_t mutexattr;
...
phtread_mutex_init(&mutex, &mutexattr);
...
```

Verrouillage

L'appel système de verrouillage des threads est :

```
phtread_mutex_lock(phtread_mutex_t *mutex);
```

Déverrouillage

L'appel système de déverrouillage des threads est :

```
phtread_mutex_unlock(phtread_mutex_t *mutex);
```

▶ **Exemple 7.** Le programme `sum.c` somme séquentiellement `N` nombres.

Listing 6.10 – sum.c

```
#include <stdio.h>

main(int argc, char **argv)
{
 double sum;
 int N, i;

 N = atoi(argv[1]);  // Nombre
 sum = 0;            // somme
 for (i=1; i<=N; ++i)
 {
   sum += i;
 }
 printf("Somme = %10.0f\n", sum);
 return(0);
}
```

Le programme `sum-thread.c` somme les mêmes `N` nombres, mais en forme parallèle avec deux threads qui utilisent mutex.

Listing 6.11 – sum-thread.c

```
#include <stdio.h>
#include <pthread.h>

volatile double somme = 0.0;  // somme (shared)
pthread_mutex_t somme_lock;   // Lock
volatile double N;            // N

void *processus(void *arg)
{
 double localsum;
 int i;
 int iproc = (*((char *) arg) - '0');
 localsum = 0;                // somme
 for (i=iproc; i<=N; i+=2)
 {
   localsum += i;
```

```
  }
  // Lock de somme, calculer et édbloquer
  pthread_mutex_lock(&somme_lock);
  somme += localsum;
  pthread_mutex_unlock(&somme_lock);
  return(NULL);
}

int main(int argc, char **argv)
{
  pthread_t thread0, thread1;
  void *retval;

  N = atoi(argv[1]);      // Nombre
  // Initialiser le lock de la somme
  pthread_mutex_init(&somme_lock, NULL);
  // 2 threads
  if (pthread_create(&thread0, NULL, processus, "0") ||
      pthread_create(&thread1, NULL, processus, "1"))
  {
    printf("%s: erreur thread\n", argv[0]);
    exit(1);
  }
  // Join les deux threads
  if (pthread_join(thread0, &retval) ||
      pthread_join(thread1, &retval))
    {
      printf("%s: erreur join\n", argv[0]);
      exit(1);
    }
  printf("Somme (épartage) = %10.0f\n", somme);
  return 0;
}
```

▶ **Exemple 8.** Soit un tableau M de N éléments rempli par un thread lent et lu par un autre plus rapide. Le thread de lecture doit attendre la fin du remplissage du tableau avant d'afficher son contenu. Les mutex peuvent protèger le tableau pendant le temps de son remplissage.

Listing 6.12 – mutex-threads2.c

```
#include <stdio.h>
#include <pthread.h>

#define N 5
static pthread_mutex_t mutex;
```

```
int M[N];

void *lecture(void * arg)
{
  int i;

  pthread_mutex_lock(&mutex);
  for (i=0; i < N; i++)
    printf("éLcture: M[%d] = %d\n", i, M[i]);
  pthread_mutex_unlock(&mutex);
  pthread_exit(0);
}

void *ecriture(void * arg)
{
  int i;

  pthread_mutex_lock(&mutex);
  for (i=0; i < N; i++)
  {
    M[i] = i;
    printf(É"criture: M[%d] = %d\n", i, M[i]);
    sleep(2); // Ralentir le thread d'ecriture
  }
  pthread_mutex_unlock(&mutex);
  pthread_exit(0);
}

int main(void)
{
  pthread_t th1, th2;
  void *ret;

  pthread_mutex_init(&mutex, NULL);

  pthread_create(&th1, NULL, ecriture, NULL);
  pthread_create(&th2, NULL, lecture, NULL);

  pthread_join(th1, &ret);
  pthread_join(th2, &ret);
  return 0;
}
```

Nous montrons ensuite une exécution de `mutex-threads2.c`:

```
leibnitz> gcc -o mt2 mutex-threads2.c -lpthread
leibnitz> mt2
Écriture: M[0] = 0
```

```
Écriture: M[1] = 1
Écriture: M[2] = 2
Écriture: M[3] = 3
Écriture: M[4] = 4
Lécture: M[0] = 0
Lécture: M[1] = 1
Lécture: M[2] = 2
Lécture: M[3] = 3
Lécture: M[4] = 4
leibnitz>
```

Sans l'utilisation des mutex, probablement on obtiendrait des sorties érronées comme la suivante :

```
Écriture: M[0] = 0
Lécture: M[0] = 0
Lécture: M[1] = 0
Lécture: M[2] = 0
Lécture: M[3] = 0
Lécture: M[4] = 0
Écriture: M[1] = 1
Écriture: M[2] = 2
Écriture: M[3] = 3
Écriture: M[4] = 4
```

6.14 Sémaphores avec System V (Solaris)

Le Unix System V sur Solaris permet de créer et d'utiliser les sémaphores. Les fonctions de manipulation des sémaphores sont dans la librairie `<synch.h>`. Le type sémaphore est désigné par le mot `sema_t`. L'initialisation d'un sémaphore est réalisée par l'appel système :

```
#include <synch.h>
int sema_init(sema_t*sp, unsigned int count, int type,
              NULL);
```

où `sp` est un pointeur sur le sémaphore à initialiser ; `count` est la valeur initiale du sémaphore ; `type` indique si le sémaphore est utilisé pour synchroniser des processus légers (threads) ou des processus. Le type peut être `USYNC_PROCESS` ou bien `USYNC_THREAD`. Par défaut, le type sera `USYNC_THREAD`.

– `int sema_wait (sema_t *sp)` est équivalente à l'opération `P`.

- `int sema_post(sema_t *sp)` est équivalente à l'opération `V`.
- `int sema_trywait(sema_t *sp)` décrémente la valeur du sémaphore `sp` si sa valeur est supérieure à `0` ; sinon elle retourne une erreur. C'est une opération atomique.

▶ **Exemple 9.** Usage de sémaphores System V.

Listing 6.13 – v-semaphore.c

```
#include <synch.h>  // ésmaphores
#include <thread.h> // thr_create et thr_join
#include <stdio.h>  // printf
#define val 1

sema_t mutex; // ésmaphore
int var_glob=0;
void* increment(void *);
void* decrement(void *);

int main()
{
   // initialiser mutex
   sema_init(&mutex,val,USYNC_THREAD,NULL) ;
   printf("ici main : var_glob = %d\n", var_glob);
   // écration d'un thread pour increment
   thr_create(NULL,0,increment,NULL,0,NULL);
   // écration d'un thread pour decrement
   thr_create(NULL,0,decrement,NULL,0,NULL);
   // attendre la fin des threads
   while(thr_join(NULL,NULL,NULL)==0);
   printf("ici main, fin threads : var_glob =%d \n",var_glob);
   return 0;
}

void* decrement(void *)
{
   int nb = 3;
   // attendre l'autorisation d'èaccs
   sema_wait(&mutex);
   while(nb--)
   {
     var_glob -= 1;
     printf("ici sc de decrement : var_glob=%d\n",
            var_glob);
   }
   sema_post(&mutex);
   return (NULL);
}
```

```
40
    void* increment (void *)
    {
       int nb=3;
       sema_wait(&mutex);
       while(nb--)
       {
         var_glob += 1;
         printf("ici sc de increment : var_glob = %d\n",
                var_glob);
50     }
       sema_post(&mutex);
       return (NULL);
    }
```

Exécution du programme `v-semaphore.c` sur la machine Unix nommé `jupiter`:

```
jupiter% gcc v-semaphore.c -lthread -o mutex
jupiter% mutex
ici main : var_glob = 0
ici sc de increment : var_glob = 1
ici sc de increment : var_glob = 2
ici sc de increment : var_glob = 3
ici sc de decrement : var_glob = 2
ici sc de decrement : var_glob = 1
ici sc de decrement : var_glob = 0
ici main, fin threads : var_glob = 0
```

► **Exemple 10.** Problème du Producteur/consommateur.

Listing 6.14 – v-prod-cons.c

```
#include <synch.h>
#include <unistd.h> // sleep
#include <thread.h>
#include <stdio.h>
#define taille 3

sema_t plein, vide,mutex;
int tampon[taille];
void* consommateur(void *);
10 void* producer(void *);

int main()
{
```

```
    // initiliser les ésmaphores
    sema_init(&plein,0,USYNC_THREAD,NULL);
    sema_init(&vide,taille,USYNC_THREAD,NULL);
    sema_init(&mutex,1,USYNC_THREAD,NULL);
    //écrer les threads
    thr_create(NULL,0,consommateur,NULL,0,NULL);
    thr_create(NULL,0,producer,NULL,0,NULL);
    // attendre la fin des threads
    while((thr_join(NULL,NULL,NULL)==0));
    printf("fin des thread \n");
    return 0;
}

void* consommateur(void *retrait)
{
    int ic=0, nbcons = 0, objet;
    do
        {
            sema_wait(&plein);
            sema_wait(&mutex);
            // consommer
            objet = tampon[ic];
            sema_post(&mutex);
            sema_post(&vide);
            printf("\n ici cons. :tampon[%d]= %d\n", ic,
                    objet);
            ic=(ic+1)%taille;
            nbcons++;
            sleep(2);
        } while ( nbcons<=5 );
    return (NULL);
}

void* producer(void *)
{
    int ip=0, nbprod=0, objet=0;
    do
        {
            sema_wait(&vide);
            sema_wait(&mutex);
            // produire
            tampon[ip]=objet;
            sema_post(&mutex);
            sema_post(&plein);
            printf("\n ici prod. : tampon[%d]= %d\n", ip,
                    objet);
            objet++;
            nbprod++;
            ip=(ip+1)%taille;
```

```
        } while ( nbprod<=5 );
    return NULL;
}
```

Exécution de producteurs et consommateurs :

```
jupiter% gcc prod_cons.c -lthread o- prod_cons
jupiter% prod_cons
ici prod. : tampon[0]= 0
ici prod. : tampon[1]= 1
ici prod. : tampon[2]= 2
ici cons. :tampon[0]= 0
ici prod. : tampon[0]= 3
ici cons. :tampon[1]= 1
ici prod. : tampon[1]= 4
ici cons. :tampon[2]= 2
ici prod. : tampon[2]= 5
ici cons. :tampon[0]= 3
ici cons. :tampon[1]= 4
ici cons. :tampon[2]= 5
fin des thread
jupiter%
```

▶ **Exemple 11.** Problème des philosophes.

Listing 6.15 – v-philosophe.c

```
#include <stdio.h>      // printf
#include <unistd.h>     // sleep
#include <thread.h>     // threads
#include <synch.h>      // ésmaphores

#define N 5             // nombre de philosophes
#define G (N+i-1)%N     // philosopheà  gauche de i
#define D (i+1)%N       // philosopheà  droite de i
#define libre 1
#define occupe 0
int fourch[N] = {libre, libre, libre, libre, libre};
sema_t mutex;
void * philosophe(void * );

int main( )
{
  int NumPhi[N] = {0,1,2,3,4};
  sema_init(&mutex, 1, NULL, NULL);
  // écration des N philosophes
```

```
    for(int i=0; i<N; i++)
        thr_create(NULL,0,philosophe,&(NumPhi[i]),0,NULL);
    // attendre la fin des threads
    while((thr_join(NULL,NULL,NULL)==0));
    printf("fin des threads\n");
    return 0;
}

// La fonction de chaque philosophe
 void * philosophe( void * num)
 {
    int i =*(int *) num;
    int nb = 2;
    while (nb)
    {
            // penser
            sleep( 1) ;
            // essayer de prendre les fourchettes pour manger
            sema_wait(&mutex) ;
            if(fourch[G] && fourch[i])
            {
                fourch[G] = 0 ;
                fourch[i] =0 ;
                sema_post(&mutex) ;
                nb--;
                // manger
                printf ("philosophe[%d] mange\n", i) ;
                sleep (1) ; // manger
                printf("philosophe[%d] a fini de manger\n", i) ;
                // élibrer les fourchettes
                sema_wait(&mutex) ;
                fourch[G] = 1 ;
                fourch[i] = 1 ;
                sema_post(&mutex) ;
            }
            else sema_post(&mutex);
    }
 }
```

Exécution de la première version des philosophes :

```
jupiter% v-philosophe.c -lthread -o philosophe
jupiter% philosophe
philosophe[0] mange
philosophe[2] mange
philosophe[0] a fini de manger
philosophe[2] a fini de manger
philosophe[4] mange
```

```
philosophe[1] mange
philosophe[4] a fini de manger
philosophe[4] mange
philosophe[1] a fini de manger
philosophe[1] mange
philosophe[4] a fini de manger
philosophe[3] mange
philosophe[1] a fini de manger
philosophe[3] a fini de manger
philosophe[2] mange
philosophe[0] mange
philosophe[0] a fini de manger
philosophe[2] a fini de manger
philosophe[3] mange
philosophe[3] a fini de manger
fin des threads
jupiter%
```

► **Exemple 12.** Solution améliorée des Philosophes (Solaris) version 2

Listing 6.16 – v-philosophe2.c

```
#include <stdio.h>
#include <unistd.h>
#include <thread.h>
#include <synch.h>
#define N 5
#define G (N+i-1)%N     // philosophe à gauche de i
#define D (i+1)%N       // philosophe à droite de i

enum etat { penser, faim, manger} ;
int fourch[N] = {libre, libre, libre, libre, libre};
etat Etat [N] = {penser, penser, penser, penser, penser} ;
sema_t S[N] ; sema_t mutex;
void * philosophe(void * );

int main()
{
   int i, NumPhi[N] = {0,1,2,3,4};
   sema_init(&mutex, 1, NULL, NULL) ;
   for (i=0 ; i<N ; i++)
       sema_int(&S[i],0,NULL,NULL) ;
   // écration des N philosophes
   for( i=0; i<N; i++)
   thr_create(NULL,0,philosophe,&(NumPhi[i]),0,NULL);
   // attendre la fin des threads
```

```
    while((thr_join(NULL,NULL,NULL)==0));
    printf("fin des threads \n");
    return 0;
}

void * philosophe( void * num)
{
    int i =*(int *) num ;
    int nb = 2 ;
    while (nb)
    {
        nb--;
        // penser
        sleep( 1) ;
        // tenter de manger
        sema_wait(&mutex) ;
        Etat[i]=faim ;
        test(i) ;
        sema_post(&mutex) ;
        sema_wait(&S[i]) ;
        // manger
        printf ("philosophe[%d] mange\n", i) ;
        sleep (1) ; // manger
        printf("philosophe[%d] a fini de manger\n", i) ;
        // élibrer les fourchettes
        sema_wait(&mutex) ;
        Etat[i] = penser ;
        // évrifier si ses voisins peuvent manger
        Test(G) ;
        Test(D) ;
        sema_post(&mutex) ;
    }
}

// éprocdure qui évrifie si le philosophe i peut manger
void Test(int i)
{
 if( (Etat[i] == faim) && ( Etat[G] != manger)
                       && (Etat[D] !=manger) )
        {
            Etat[i] = manger ;
            sema_post(&S[i]) ;
        }
}
```

Exécution de la deuxième version des philosophes :

```
jupiter% v-philosophe2.c -lthread -o philosophe2
jupiter% philosophe2
```

```
philosophe[0] mange
philosophe[2] mange
philosophe[0] a fini de manger
philosophe[2] a fini de manger
philosophe[4] mange
philosophe[1] mange
philosophe[4] a fini de manger
philosophe[3] mange
philosophe[1] a fini de manger
philosophe[0] mange
philosophe[0] a fini de manger
philosophe[1] mange
philosophe[3] a fini de manger
philosophe[4] mange
philosophe[4] a fini de manger
philosophe[3] mange
philosophe[1] a fini de manger
philosophe[3] a fini de manger
philosophe[2] mange
philosophe[2] a fini de manger
fin des threads
jupiter%
```

6.15 Exercises

1. Pourquoi le partage de données pose des problèmes dans un système multiprogrammé en temps partagé ?
2. Le système Unix permet-il de contrôler les accès aux données partagées ?
3. Qu'est-ce qu'une section critique ?
4. Expliquez la raison pour laquelle il est nécessaire de synchroniser les processus. Est-ce que tous les types de SE utilisent la synchronisation de processus ?
5. Identifiez et expliquez un problème typique qui nécessiterait la synchronisation de processus.
6. Identifiez le point (i.e. les énoncés) d'un processus qui représente sa section critique.
7. Identifiez le problème principal qui peut survenir lorsque l'accès à la section critique d'un processus est contrôlé par une variable booléenne de type verrou. Donnez une solution à ce problème.
8. Quel est le problème principal que résout un sémaphore en ce qui concerne l'accès à une section critique ?
9. Quand doit-on utiliser des sémaphores binaire et général ? Peut-on utiliser ces deux types de sémaphores dans un même processus et pour un même problème de synchronisation ? Expliquez.
10. Pourquoi un tube Unix utilise généralement deux canaux de communication entre deux processus, soit un canal d'entrée et un canal de sortie ? Existe-il des alternatives à ce mode de communication utilisant les tubes Unix ?
11. Considérez l'application suivante qui implante un système interactif qui lit dans une boucle des caractères entrés via le clavier et qui les affiche à l'écran. Le traitement se termine lorsque l'usager presse la touche `<Ctrl-d>` pour insérer le caractère de fin de fichier (`EOF`) dans le tampon.

```
// Processus parent
char *tampon;   // tampon partagé par les processus
int i=0, j=0;
// positions de lecture et d'écriture dans le tampon
// Processus lecture
   ...
```

```
   while (read(stdin, tampon, 1) != NULL)
   ...
// Processus écriture
   ...
   while (write(stdout, tampon, 1) != NULL)
   ...
```

(a) Synchronisez la communication entre les processus lecture et écriture à l'aide de tubes et d'appels système `fork`.

(b) Synchronisez la communication entre les threads lecture et écriture à l'aide de sémaphores.

(c) Considérez la solution suivante au problème d'accès à des ressources partagées en mode de lecture et d'écriture.
- Les processus en lecture et en écriture sont en compétition pour l'accès à une ressource partagée.
- Un seul processus à la fois peut être dans sa section critique, ceci est valable autant pour les processus en lecture et en écriture.
- Le système ne fait pas de différence entre les processus en lecture et en écriture, et chaque processus peut postuler une requête d'accès à la ressource qui lui sera accordée dès qu'un processus a fini de lire ou d'écrire sur la ressource.
- Cependant, le système définit des processus prioritaires pour l'écriture de la ressource, et les requêtes de ces processus doivent être honorées avant celles des processus normaux en lecture et en écriture dès que la section critique se libère.

Montrez la synchronisation des processus en lecture et en écriture, en utilisant des sémaphores.

12. Considérez le buffer circulaire montré sur la figure 6.9, qui répresente une variation du problème du producteur/consommateur. Résoudre le problème avec des sémaphores.

13. Le code suivant contient une solution proposée au problème d'accès à une section critique. Expliquez pourquoi cette solution serait correcte ou incorrecte, selon qu'elle respecte les 4 principes fondamentaux régissant l'accès à une section critique.

```
/* Variables partagées par les processus P0 et P1. */
 int flag[2];    // Tableau de booléens stockant 0 ou 1
            // Variable indiquant le processus actif,
            // i.e. 0 pour P0 ou 1 pour P1
```

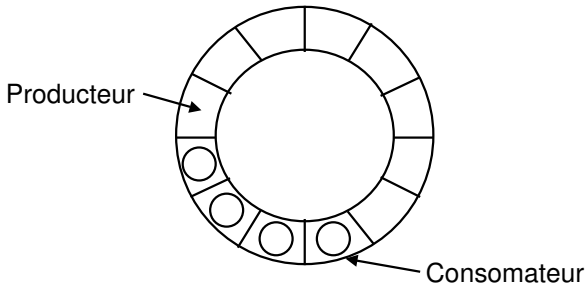

FIG. 6.9 – Buffer circulaire.

```
int turn;

// Code du processus P0
do
{
   flag[0] = 1;       // Mettre le flag de P0 à 1
   while (turn != 0)  // Si P0 est inactif alors, ...
   {
   while (flag[1]) // Attendre l'inactivité de P1
       {   /* skip */ }
       turn = 0;   // Indique que P0 est maintenant actif
   }
   ...
   <Section critique pour le processus P0>
   ...
   // Réinitialiser le flag de P0 afin que P1 puisse
   // s'activer dans sa section critique
   flag[0] = 0;
   ...
   <Autres énoncés pour P0>
   ...
} while (1);

/* Code du processus P1 */
do
{
   flag[1] = 1;    /* Mettre le flag de P1 à 1 */
   while (turn != 1)   /* Si P1 est inactif alors, ... */
   {
       while (flag[0]) /* Attendre l'inactivité de P0 */
       {   /* skip */ }
```

```
        turn = 1;   /* Indiquez que P1 est maintenant actif */
    }
    ...
    <Section critique pour le processus P1>
    ...
    // Réinitialiser le flag de P1 afin que P0 puisse
    // s'activer dans sa section critique
    flag[1] = 0;
    ...
    <Autres énoncés pour P1>
    ...
} while (1);
```

14. Dijkstra a proposé les 3 solutions suivantes comme étant des solutions logicielles au problème d'accès à une section critique. Par contre, aux vues de Dijkstra ces solutions n'étaient pas tout à fait correctes. Alors, expliquez pourquoi chacune des solutions est effectivement incorrecte.

a)
```
int turn;
proc(int i)
{   while (TRUE)
    {   compute;
        while (turn != i);
        <section critique>
        turn = (i+1) mod 2;
    }
}
turn = 1;
fork(proc, 1, 0);
fork(proc, 1, 1);
```

b)
```
boolean flag[2];
proc(int i)
{   while (TRUE)
    {   compute;
        while (flag[(i+1) mod 2]);
        flag[i] = TRUE;
        <section critique>
        flag[i] = FALSE;
    }
}
flag[0] = flag[1] = FALSE;
fork(proc, 1, 0);
fork(proc, 1, 1);
```

c)
```
boolean flag[2];
```

```
proc(int i)
{   while (TRUE)
    {   compute;
        flag[i] = TRUE;
        while (flag[(i+1) mod 2]);
        <section critique>
        flag[i] = FALSE;
    }
}
flag[0] = flag[1] = FALSE;
fork(proc, 1, 0);
fork(proc, 1, 1);
```

15. Considérez une route à deux voies (une voie par direction) et orientée selon un axe nord-sud. Cette route traverse un tunnel qui ne possède qu'une seule voie, et par conséquent son accès doit être partagé en temps par les voitures provenant de chacune des deux directions (nord ou sud). Afin de prévenir les collisions, des feux de circulation ont été installés à chacune des extrémités du tunnel. Un feu vert indique qu'il n'y a aucune voiture dans le tunnel provenant de la direction opposée, et un feu rouge indique qu'il y a une voiture à contresens dans le tunnel. Le trafic dans le tunnel est contrôlé par un ordinateur central qui est prévenu de l'arrivée de véhicules par des senseurs posés dans chaque direction. Lorsqu'un véhicule approche le tunnel en direction nord ou sud, le senseur externe exécute la fonction arrivée(direction) pour prévenir l'ordinateur central. Similairement, lorsqu'une voiture sort du tunnel le senseur interne exécute la fonction départ (direction). Concevez une esquisse de code pour synchroniser l'accès au tunnel afin que les feux de circulation fonctionnent correctement dans les deux directions.

16. On a 3 processus P_1, P_2 et P_3 concurrents. Le code *indépendant* de chaque processus $P_i (i = 1, 2, 3)$ est le suivant :

```
Pi()     // Processus Pi avec i=1,2,3
{
   while(TRUE)
       printf("je suis le processus" i );
}
```

On veut montrer à l'écran la sortie suivante :

```
je suis le processus 1
je suis le processus 2
je suis le processus 3
```

```
je suis le processus 1
je suis le processus 2
je suis le processus 3
...
```

Synchroniser les 3 processus à l'aide des sémaphores binaires.

17. Le programme suivant permet de calculer une estimation de la valeur de π avec un flot unique d'exécution. Parallélisez ce programme à l'aide d'exclusion mutuelle de threads et en respectant l'esprit de la boucle `for (i=0; i<intervals; ++i){...}`.

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char **argv)
{
 double width, sum;
 int intervals, i;

 // nombre d'intervalles
 intervals = atoi(argv[1]);
 width = 1.0 / intervals;

 // calcul de pi
 sum = 0;
 for (i=0; i<intervals; ++i)
 {
   double x = (i + 0.5) * width;
   sum += 4.0 / (1.0 + x * x);
 }
 sum *= width;
 printf("Estimation de pi = %10.9f\n", sum);
 return(0);
}
```