Synchronisation de processus
Partie 1

- Introduction
- Sections critiques
- Comment assurer l'exclusion mutuelle ?
- Sémaphores
- Problème du « Producteur/consommateur »
- Problème des « Philosophes »
- Problème des « lecteurs/Rédacteurs »
- Problème du « client/serveurs »

Introduction

- 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. L'état final des données dépend de l'ordonnancement des processus.
- Lorsqu'un processus modifie un objet partagé, les autres processus ne doivent ni la lire, ni la modifier jusqu'à ce que le processus ait terminé de la modifier.
- Autrement dit, l'accès à l'objet doit se faire en **exclusion mutuelle**.

Introduction (2)
Exemple

- Considérons deux clients A et B d'une banque qui partagent un même compte. Le solde du compte est 1000$. Supposons qu'en même temps, les deux clients lancent chacun une opération. Le client A demande un retrait de 1000$ alors que le client B veut faire un dépôt de 100$.
- Les deux threads (ou les deux processus) exécutant les requêtes des clients partagent la variable solde.

```
Requête de A :
/*A1*/     if (solde >= 1000)
/*A2*/             solde -= 1000;
           else
             printf(« erreur... »);
```

```
Requête de B :
 /*B1*/     solde += 100;
```

- Les deux requêtes sont satisfaites et le solde pourrait passer à 1100$!

B : calcule solde+100 → Registre = 1100

A: exécute la requête au complet → solde = 0

B : poursuit sa requête → solde = 1100

Sections critiques

- Il faut empêcher l'utilisation simultanée de la variable commune solde. Lorsqu'un thread (ou un processus) exécute la séquence d'instructions (1 et 2), l'autre doit attendre jusqu'à ce que le premier ait terminé cette séquence.
- Section critique : suite d'instructions qui opèrent sur un ou plusieurs objets partagés et qui nécessitent une utilisation exclusive des objets partagés.
- Chaque thread (ou processus) a ses propres sections critiques.
- Les sections critiques des différents processus ou threads qui opèrent sur des objets communs doivent s'exécuter en exclusion mutuelle.
- Avant d'entamer l'exécution d'une de ses sections critiques, un thread (ou un processus) doit s'assurer de l'utilisation exclusive des objets partagés manipulés par la section critique

Comment assurer l'exclusion mutuelle?

- Encadrer chaque section critique par des opérations spéciales qui visent à assurer l'utilisation exclusive des objets partagés.
- Quatre conditions sont nécessaires pour réaliser correctement une exclusion mutuelle :
  - Deux processus ne peuvent être en même temps dans leurs sections critiques.
  - Aucune hypothèse ne doit être faite sur les vitesses relatives des processus et sur le nombre de processeurs.
  - Aucun processus suspendu en dehors de sa section critique ne doit bloquer les autres processus.
  - Aucun processus ne doit attendre trop longtemps avant d'entrer en section critique (attente bornée).

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.

Problèmes

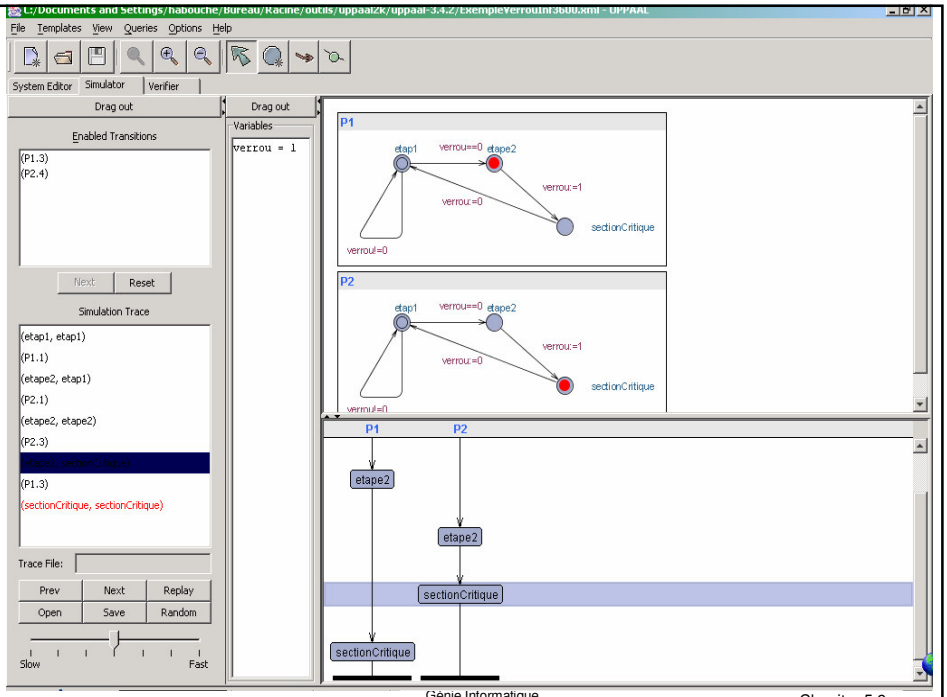
- Solution dangereuse : si le processus, pour une raison ou pour une autre, ne restaure pas les interruptions à la sortie de la section critique, ce serait la fin du système.
- Elle n'assure pas l'exclusion mutuelle, si le système n'est pas monoprocesseur (le masquage des interruptions concerne uniquement 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.

Attente active et verrouillage

- Utiliser une variable partagée **verrou**, unique, initialisée à 0.
- Pour rentrer en section critique, un processus doit tester la valeur du **verrou**.
- Si **verrou** = 0, le processus met le **verrou** à 1 puis exécute sa section critique.
- A la fin de la section critique , il remet le **verrou** à 0.
- Sinon, il attend (attente active) que le **verrou** devienne égal à 0 : **while(verrou !=0) ;**

Attente active et verrouillage (2) Problèmes

- Elle 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 le **verrou** à 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** à 1.
  - Les deux processus sont en même temps en section critique.
- L'attente active surcharge le CPU



Attente active avec alternance

- Utiliser une variable **tour** qui mémorise le tour du processus qui doit entrer en section critique.
- **tour** est initialisée à 0.

```
Processus P0
while (1)
{    /*attente active*/
     while (tour !=0) ;
     section_critique_P0() ;
     tour = 1 ;
....
}
```

```
Processus P1
while (1)
{    /*attente active*/
     while (tour !=1) ;
     section_critique_P1() ;
     tour = 0 ;
....
}
```

Attente active avec alternance (2)
Problèmes

- Un processus peut être bloqué par un processus qui n'est pas en section critique.
  - P0 lit la valeur de tour qui vaut 0 et entre dans sa section critique. Il est suspendu et P1 est exécuté.
  - P1 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 P0 est élu de nouveau.
  - P0 quitte sa section critique, met tour à 1 et entame sa section non critique. Il est suspendu et P1 est exécuté.
  - P1 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.
- Attente active consomme du temps CPU.

Solution de Peterson

- Cette solution se base sur deux fonctions entrer_region et 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, il doit appeler quitter_region pour indiquer qu'il quitte sa section critique et pour autoriser l'accès aux autres processus.

Solution de Peterson (2)

```
void entrer_region(int process)
{   int autre ;
    autre = 1 – process ;    //l'autre processus
    interesse[process] = TRUE ;  //indiquer qu'on est intéressé
    tour = process ;
    while (tour == process && interesse[autre] ==TRUE) ;
}

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

Problème : attente active = consommation du temps CPU

L'instruction Test and Set Lock (TSL)

- L'instruction Test and Set Lock « int TSL (int b) » exécute de manière indivisible les opérations suivantes :
  - récupère la valeur de b,
  - affecte la valeur 1 à b et
  - retourne l'ancienne valeur de b.
- 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'exécution.
- Cette instruction peut être utilisée pour établir et supprimer des verrous (assurer l'exclusion mutuelle).

L'instruction TSL (2)

- Valeur initiale de verrou est 0.
- entrer_region :
  ```
  while(TSL(verrou) ) ;
  ```
- quitter_region :
  ```
  verrou =0;
  ```


L'instruction TSL (3) Problèmes

- attente active = consommation du temps CPU
- un processus en attente active peut rester en permanence dans la boucle (boucle infinie). Supposons que :
  - Deux processus H et B, tels que H est plus prioritaire que B, partagent un objet. Les règles d'ordonnancement font que H est exécuté dès qu'il passe à l'état prêt.
  - Pendant que H est en E/S, le processus B entre en section critique.
  - Ensuite, le processus H devient prêt alors que B est toujours en section critique.
  - Le processus B est alors suspendu au profit du processus H. H effectue une attente active et B ne peut plus être réélu (puisque H est plus prioritaire).
  - B ne peut plus sortir de sa critique et H boucle indéfiniment.

L'instruction swap

- Plusieurs architectures supportent également l'instruction swap qui permet de permuter les valeurs de deux variables de manière atomique.

```
swap (a, b);
```

est équivalent à :

```
temp = a;
a = b;
b = temp;
```


Les primitives SLEEP et WAKEUP

- SLEEP est un appel système qui suspend l'appelant en attendant qu'un autre le réveille.
- WAKEUP(process) : est un appel système qui réveille le processus process.
- 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.

Les primitives SLEEP et WAKEUP (2)
Problèmes

- Les tests et les mises à jour des conditions d'entrée en section critique ne sont pas exécutés en exclusion mutuelle.
- Si le signal émis par WAKEUP(P2) arrive avant que P2 ne soit endormi, P2 pourra dormir pour toujours.

```
verrou = 0;
```

```
Processus P1
{    if(verrou ==1) SLEEP();
       verrou = 1;
       section_critique_P1( );
       verrou = 0;
       WAKEUP(P2)  ;
       ….
}
```

```
Processus P2
{   if (verrou ==1) SLEEP();
       verrou= 1;
       section_critique_P2( );
       verrou= 0;
       WAKEUP(P1)  ;
       ….
}
```


Résumé des problèmes

- Attentes actives → consommation du temps CPU.
- Masquage des interruptions → dangereuse pour le système
- TSL → attente active + inversion des priorités => boucle infinie.
- SLEEP et WAKEUP → synchronisation des signaux

Sémaphores

- Pour contrôler les accès à un objet partagé, E. W. Dijkstra (1965) suggéra l'emploi d'un nouveau type de variables appelées sémaphores.
- Un sémaphore est un compteur entier qui désigne le nombre d'autorisations d'accès disponibles.
- Chaque sémaphore a un nom et une valeur initiale.
- Les sémaphores sont manipulés au moyen des opérations :
  - P (désigné aussi par down ou wait) et
  - V (désigné aussi par up ou signal).

Sémaphores (2) Opérations P et V

- 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.
- 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.

```
Semaphore S = 1 ;
```

```
Processus P1 :
{       P(S)
        Section_critique _de_P1() ;
        V(S) ;
}
```

```
Processus P2 :
{   P(S)
    Section_critique_de_P2();
    V(S) ;
}
```

- Chacune de ces deux opérations doit être implémentée comme une opération indivisible.

Implémentation des sémaphores masquage des interruptions

```
struct Semaphore
{   int count; // valeur du sémaphore
    queue q;   // file d'attente
}

P(Semaphore *S)
{       Disable interrupts;
            if (S->count > 0)
            {           S->count-= 1;
                        Enable interrupts;
                        return;
            }
            Add process to S->q
            Enable interrupts;
            Redispatch;
}

V(Semaphore *S)
{    Disable interrupts;
     if (!S->q empty)
            Remove (S->q,ReadyQueue);
     else S->count += 1;
     Enable interrupts;
}
```


Implémentation des sémaphores (2) masquage des interruptions

Problème :

- Le masquage des interruptions n'assure pas l'exclusion mutuelle dans un système multiprocesseur.

Solution :

- Utilisation de l'instruction Test and Set Lock.

Implémentation des sémaphores (3)
Utilisation de Test-and-Set-Lock (TSL)

```
struct Semaphore
{    int count;
     queue q;
     int t;
     // t=0 si l'accès est libre aux attributs du
     // sémaphore
}
P(Semaphore *S)
{    Disable interrupts;
     while (TSL(S->t) != 0) /* do nothing */;
          if (S->count > 0) {
          S->count = S->count-1;
          S->t = 0;
          Enable interrupts;
          return;
     }
     Add process to S->q;
     S->t = 0;
     Enable interrupts;
     Redispatch;
}
```

```
V(Semaphore *S)
{    Disable interrupts;
     while (TSL(S->t) != 0) ;
     if (S->q empty)
          S->count += 1;
     else
        Remove (S->q, Readyqueue);
     S->t = 0;
      Enable interrupts;
}
```


Implémentation des sémaphores (4)
Utilisation de Test-and-Set-Lock (TSL)

Remarques :

- Le masquage des interruptions est utilisé ici pour éviter le problème de boucle infinie (ordonnanceur à priorité).

  Supposons que S->t = 0 et deux processus P1 et P2 tels que P2 est plus prioritaire que P1 ;

  - P1 exécute : TSL(S->t) S->t devient égal à 1
  - Interruption de l'exécution de P1 au profit de P2 (P2 est plus prioritaire)
  - P2 rentre dans la boucle while(TSL(S->t) ! =0);

- Une autre solution au problème est l'héritage de priorité.

Exclusion mutuelle au moyen de sémaphores

- Les sémaphores binaires permettent d'assurer l'exclusion mutuelle :

```
Semaphore mutex = 1 ;
```

```
Processus P1 :
{       P(mutex)
        Section_critique _de_P1() ;
        V(mutex) ;
}
```

```
Processus P2 :
{   P(mutex)
    Section_critique_de_P2();
    V(mutex) ;
}
```


Exclusion mutuelle au moyen de sémaphores

- Les sémaphores binaires permettent d'assurer l'exclusion mutuelle :

```
semaphore mutex = 1 ;
```

```
Processus P1 :
{       P(mutex)
        Section_critique _de_P1() ;
        V(mutex) ;
}
```

```
Processus P2 :
{   P(mutex)
    Section_critique_de_P2();
    V(mutex) ;
}
```

P1: P(mutex) → mutex=0
P1 : entame Section_critique_de_P1();
P2 : P(mutex) → P2 est bloqué → file de mutex
P1: poursuit et termine Section_critique_de_P1();
P1 : V(mutex)→ débloque P2 → file des processus prêts
P2: Section_critique_de_P2();
P2: V(mutex) → mutex=1

Sémaphores POSIX

- Les sémaphores POSIX sont implantés dans la librairie <semaphore.h>
- Le type sémaphore est désigné par le mot : sem_t.

On peut aussi créer un sémaphore nommé au moyen de sem_open

- L'initialisation d'un sémaphore est réalisée par l'appel système :

int sem_init (sem_t*sp, int pshared, unsigned int count) ;

où

  - sp est un pointeur sur le sémaphore à initialiser
  - count est la valeur initiale du sémaphore
  - pshared indique si le sémaphore est local au processus ou non (0 pour local et non null pour partagé).

- La suppression d'un sémaphore :
  int sem_destroy(sem_t * sem);

Sémaphores POSIX (2)

- int sem_wait (sem_t *sp) : est équivalente à l'opération P.
- int sem_post(sem_t *sp) : est équivalente à l'opération V.
- int sem_trywait(sem_t *sp) : décrémente la valeur du sémaphore sp, si sa valeur est supérieure à 0 ; sinon elle retourne une erreur (sem_wait non bloquant).
- int sem_getvalue(sem_t * sem, int * sval) : retourne dans sval la valeur courante du sémaphore.

Sémaphores POSIX (3)
Exemple 1

```
#include <semaphore.h>  // pour les sémaphores
#include <pthread.h> // pour pthread_create et pthread_join
#include <stdio.h>  // pour printf
#define val 1
sem_t  mutex;  // sémaphore
int var_glob=0;
void* increment( void *);
void* decrement( void *);
int main ( )
{     pthread_t threadA, threadB
      sem_init(&mutex, 0, val) ;                  // initialiser mutex
        printf("ici main : var_glob = %d\n", var_glob);
      pthread_create(&threadA, NULL, increment,NULL); // création d'un thread pour increment
      pthread_create(&threadB,NULL,decrement,NULL); // création d'un thread pour decrement
        // attendre la fin des threads
        pthread_join(threadA,NULL);
        pthread_join(threadB,NULL);
      printf("ici main, fin threads : var_glob =%d \n",var_glob);
  return 0;
}
```


Sémaphores POSIX (4)
Exemple 1 (suite)

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

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

Problème du producteur/consommateur

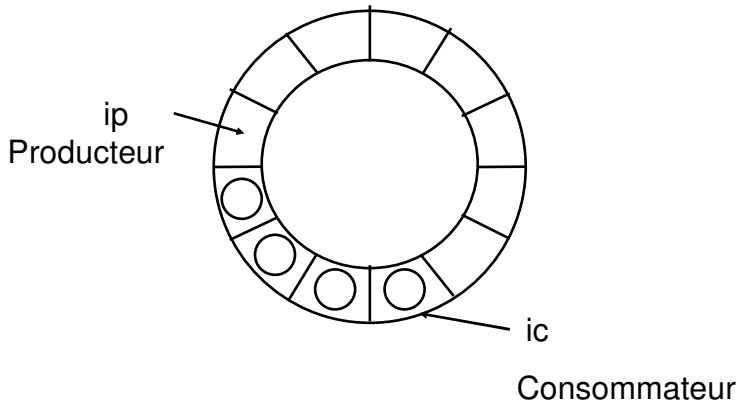
- Deux processus partagent une mémoire tampon de taille fixe N. L'un d'entre eux, le producteur, dépose des informations dans le tampon, et l'autre, le consommateur, les retire.
- Le tampon est géré comme une file circulaire ayant deux pointeurs (un pour les dépôts et l'autre pour les retraits).
- Producteur :

```
{   ip =0;
    Répeter
    {       S'il y a, au moins, une entrée libre dans le tampon
            alors   produire(tampon, ip); ip = Mod(ip +1,N);
            sinon   attendre jusqu'à ce qu'une entrée se libère.
    } tantque vrai
}
```

- Consommateur :

```
{   ic =0;
    Répéter
    {       S'il y a, au moins, une entrée occupée dans le tampon
            alors   consommer(tampon,ic); ic= Mod(ic+1, N);
            sinon   attendre jusqu'à ce qu'une entrée devienne occupée
    } tantque vrai
}
```


Problème du producteur/consommateur (2)

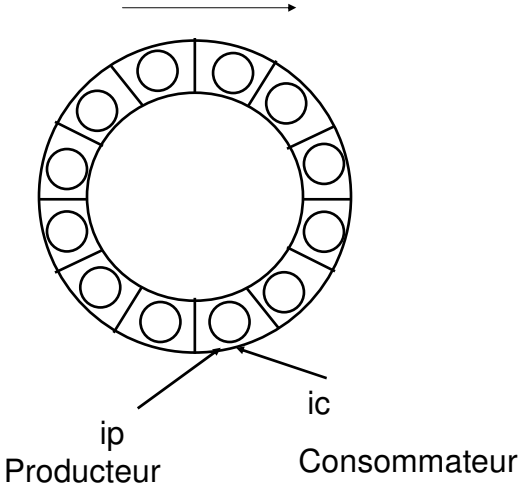


Problème du producteur/consommateur (3)

tampon plein

occupe = N;

libre = 0



Problème du producteur/consommateur (4)

- La solution du problème au moyen des sémaphores nécessite deux sémaphores.
- Le premier, nommé occupe, compte le nombre d'emplacements occupés. Il est initialisé à 0.
- Il sert à bloquer/débloquer le consommateur (P(occupe) et V(occupe)).
- Le second, nommé libre, compte le nombre d'emplacements libres. Il est initialisé à N (N est la taille du tampon).
- Il sert à bloquer/débloquer le producteur (P(libre) et V(libre)).

Problème du producteur/consommateur (5)

- tampon [N];
- Producteur :

```
{   int ip =0;
    Répeter
    {       P(libre) ;
            produire(tampon, ip);
            ip = Mod(ip +1,N);
            V(occupe);
    } tantque vrai
}
```

- Consommateur :

```
{   int ic =0;
    Répéter
    {       P(occupe);
            consommer(tampon,ic);
            ic= Mod(ic+1, N);
            V(libre);
    } tantque vrai
}
```


Problème du producteur/consommateur

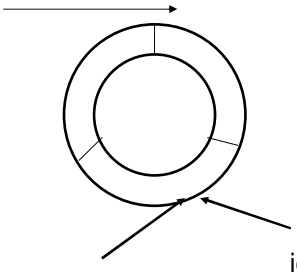
occupe = 0;

libre = 3

⇓

Consommateur bloqué

Producteur peut produire jusqu'à 3



ip
Producteur

ic
Consommateur

Une production

Problème du producteur/consommateur

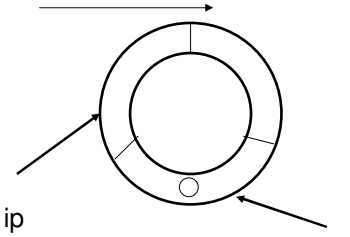
occupe = 1;

libre = 2

⇓

Consommateur peut consommer 1

Producteur peut produire 2



→ Une production

Problème du producteur/consommateur

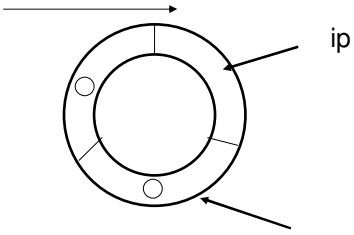
occupe = 2;

libre = 1

⇓

Consommateur peut consommer 2

Producteur peut produire 1



→ Une production

Problème du producteur/consommateur

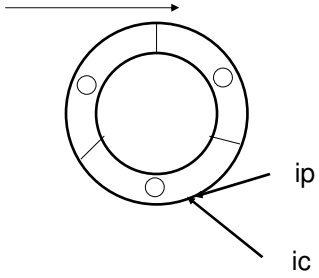
occupe = 3;

libre = 0

⇓

Consommateur peut consommer 3

Producteur bloqué



→ Une consommation

Problème du producteur/consommateur

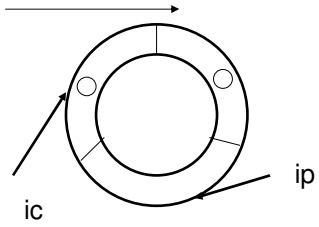
occupe = 2;

libre = 1

⇓

Consommateur peut consommer 2

Producteur peut produire 1



→ Une production

Problème du producteur/consommateur

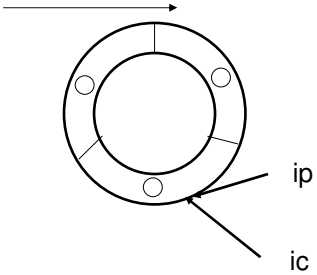
occupe = 3;

libre = 0

⇓

Consommateur peut consommer 3

Producteur bloqué



Problème du producteur/consommateur (6)

```
//progamme prod_cons.c
#include <semaphore.h>
#include <unistd.h> //pour sleep
#include <pthread.h>
#include <stdio.h>
#define N 3
sem_t occupe, libre ;
int tampon[N];
void* consommateur(void *);
void* producteur(void *);
int main()
{    pthread_t th1,th2;
     sem_init(&occupe,0,0); // initialiser les sémaphores
     sem_init(&libre,0,N);
     pthread_create(&th1, NULL,consommateur ,NULL); //créer les threads
     pthread_create(&th2,NULL,producteur,NULL);
     pthread_join(th1,NULL); // attendre la fin des threads
     pthread_join(th2,NULL);
     printf("ici main, fin des threads \n");
     return 0;
}
```

Problème du producteur/consommateur (7)
La fonction du thread producteur

```
void* producteur(void *depot)
{   int ip=0, nbprod=0, objet=0;
    do {
          sem_wait(&libre);
          // produire
          tampon[ip]=objet;
          sem_post(&occupe);
          printf("\n ici prod. : tampon[%d]= %d\n", ip,objet);
          objet++;       nbprod++;
          ip=(ip+1)%N;
      }  while ( nbprod<=5 );
          return NULL;
}
```


Problème du producteur/consommateur (8)
La fonction du thread consommateur

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

Problème du producteur/consommateur (9)
Exécution

```
jupiter% gcc prod_cons.c –lpthread -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
 ici main, fin des  threads
jupiter%
```

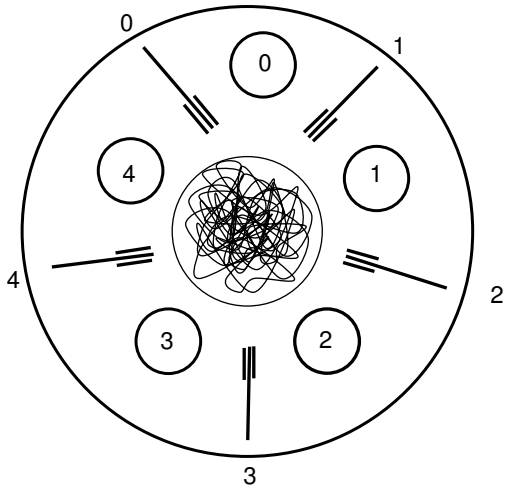
Exercice :

Généraliser à plusieurs producteurs et plusieurs consommateurs avec un seul tampon.

Problème des philosophes

- Cinq philosophes sont assis autour d'une table. Sur la table, il y a alternativement 5 plats de spaghettis et 5 fourchettes.
- 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.
- 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.

Problème des philosophes (2)



Problème des philosophes (3)
Implémentation

```
// programme philosophe.c
#include <stdio.h>                  // pour printf
#include <unistd.h>       // pour sleep
#include <pthread.h>      // pour les threads
#include <semaphore.h>            // pour les sémaphores
#define N 5                  // nombre de philosophes
#define G (i+1)%N         // fourchette gauche du philosophe i
#define D   i       // fourchette droite du philosophe i
#define  libre 1
#define occupe 0
int fourch[N] = {libre, libre, libre, libre, libre};
sem_t mutex ;
void * philosophe(void * ) ;
```

Problème des philosophes (4)
Implémentation

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


Problème des philosophes (5)
Implémentation

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

Problème des philosophes (6)
Exécution

```
jupiter% gcc philosophe.c –lpthread  –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
```


Problème des philosophes (7)
La famine

- La solution précédente résout le problème d'interblocage. Mais, un philosophe peut mourir de faim car il ne pourra jamais obtenir les fourchettes nécessaires pour manger (problème de famine et d'équité).
- Pour éviter ce problème, 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

Problème des philosophes (8)
La famine

```
philosophe( i in [0,5])
{ Répéter
      {     penser();
           // tenter de manger
           P(mutex) ;
            Etat[i]=faim ;
           Test(i);
           V(mutex) ;
           P(S[i]) ;
            manger();
           // libérer les fourchettes
              P(mutex) ;
               Etat[i] = penser ;
             // vérifier si ses voisins peuvent manger
              Test(G(i)) ;
              Test(D(i)) ;
               V(&mutex) ;
            }
}
```

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


Problème des Lecteurs / Rédacteurs

- Ce problème modélise les accès à une base de données. Plusieurs processus tentent 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 autre 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.

Problème des Lecteurs / Rédacteurs (2)

- Pour contrôler les accès à la base, on a besoin de connaître le nombre de lecteurs qui sont en train de lire de la base (NbL).
- 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).

Problème des Lecteurs / Rédacteurs (3)

```
void lecteur(void)
{  do {    sem_wait(&mutex) ;
           if ( NbL == 0)  sem_wait(&Redact) ;
           NbL ++;
           sem_post(&mutex) ;
           // lecture de la base
           sleep(2) ;
           // fin de l'accès à la base
           sem_wait(&mutex) ;
           NbL -- ;
           If(NbL ==0) sem_post(&Redact) ;
           sem_post(&mutex) ;
           // traitement des données lues
           sleep(2) ;
   } while(1)
}
```

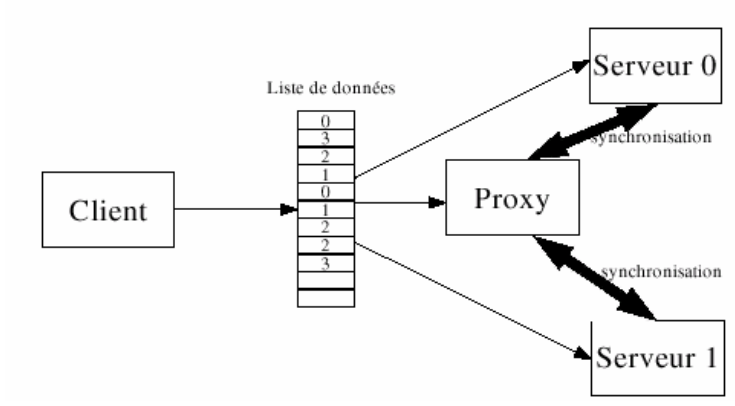
Problème des Lecteurs / Rédacteurs (4)

```
void redacteur(void)
{
    do {
        sem_wait(&Redact)
        // modifier les données de la base
        sleep(2) ;
        sem_post(&Redact) ;
        } while(1)
}
```

Exercice :

- Modifiez le code de manière à empêcher les lecteurs d'entrer dans la base si un au moins des rédacteurs est en attente.

Problème client / serveurs



{Numéro du serveur, nombre de paquets, les paquets de données}$^{+}$

Problème client / serveurs (2)

Sémaphores :

- libre, occupe et mutex pour contrôler le remplissage du tampon (modèle Producteur/consommateur).
- sem_proxy pour bloquer/debloquer le proxy.
- sem_serveur [NbServeurs] pour bloquer/débloquer un serveur.

Semaphore
libre= N, // taille du tampon
occupe=0,
mutex = 1,
sem_proxy = 1,
sem_serveur[i] = 0; 0 ≤ i < NbS // nombre de serveurs;

Problème client / serveurs (3)

```
Client
{   Repeter
    {       // récupérer les paquets à envoyer
            lire (nbp, paquets);
            // dépôt des paquets
            pour i=0 à nbp-1 pas 1
            {           P(libre);
                        P(mutex);
                        depot(paquets,i,ic);
                        V(mutex);
                        V(occupe);
            }
    }
}
// 1er  paquet <- numéro du serveur
// 2ième paquet <-  nombre de paquets de données
// Autres paquets <- données
```

```
Proxy
{   Repeter
    {       P(sem_proxy);
            P(occupe);
            P(mutex);
            nserv=retrait(ip);
            V(mutex);
            V(libre);
            V(sem_serveur[nserv]);
    }
}
```

Problème client / serveurs (4)

```
Serveur (numero id)
{    Repeter
     {       P(sem_serveur[id]);
             P(occupe);
             P(mutex);
             nbpaquets=retrait(is);
             V(mutex);
             V(libre);
             pour i=0 à nbpaquets-1 pas 1
             {           P(occupe);
                         P(mutex);
                         data[i] = retrait(is);
                         V(mutex);
                         V(libre);
             }
             V(sem_proxy);
     }
}
```


Sémaphores anonymes Windows XP

```
HANDLE hSemaphore;
LONG cMax = 10;

// Créer un sémaphore dont la valeur initiale est 10.
hSemaphore = CreateSemaphore(
   NULL,   // sécurité par défault
   cMax,   // compte initial
   cMax,   // compte maximum
   NULL);  // sémaphore anonyme

if (hSemaphore == NULL)
{printf("CreateSemaphore error: %d\n", GetLastError());}
```

Sémaphores anonymes
Windows XP

```
DWORD dwWaitResult;

// Essayer d'obtenir le sémaphore
dwWaitResult = WaitForSingleObject(
                                    hSemaphore,   // handle au sémaphore
                                    0);         // zero-second time-out interval

switch (dwWaitResult)
{
   // Le sémphore était signalé
   case WAIT_OBJECT_0: { ….
            break; }

   // Le sémpahore était non-signalé – time-out
   case WAIT_TIMEOUT: { …..
      break; }
}
```


Sémaphores anonymes
Windows XP

```
// Incrémenter la valeur du sémaphore
if (!ReleaseSemaphore(
       hSemaphore,  // handle au semaphore
       1,          // incrémenter la valeur de 1
       NULL) )      // ignorer la dernière incrémentation
{
   printf("ReleaseSemaphore error: %d\n", GetLastError());
}
```