Noyau d'un système d'exploitation
INF2610

Chapitre 5: Synchronisation

Département de génie informatique et génie logiciel

POLYTECHNIQUE MONTRÉAL
AFFILIÉE À L'UNIVERSITÉ DE MONTRÉAL

Automne 2017

Synchronisation de processus

- Introduction
- Comment assurer l'exclusion mutuelle ?
- Sémaphores
- Problèmes classiques de synchronisation
  - Producteurs et consommateurs
  - Le dîner des philosophes
  - Lecteurs et rédacteurs
- Les moniteurs

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, fichiers, etc.).

- Le partage d'objets sans précaution pourrait conduire à des résultats imprévisibles. L'état final des données dépend de l'ordonnancement des processus (ou threads).

- 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

- Soient 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

Introduction (3) : Exemple

- Deux processus P1 et P2 (ou threads) exécutent les deux requêtes sur deux processeurs.
- La valeur attendue est 100.

Dates	CPU 0		CPU 1		solde
0	Inst.	Reg.	Inst.	Reg.	1000
1	load solde	1000			1000
2			load solde	1000	1000
3	sub 1000	0			1000
4			add 100	1100	1000
5	store solde	0			0
6			store solde	1100	1100

Dates	CPU 0		CPU 1		solde
0	Inst.	Reg.	Inst.	Reg.	1000
1	load solde	1000			1000
2			load solde	1000	1000
3	sub 1000	0			1000
4			add 100	1100	1000
5			store solde	1100	1100
6	store solde	0			0

Introduction (4) : Exemple

- Il faut empêcher l'utilisation simultanée de la variable commune solde. Lorsqu'un thread (ou un processus) exécute la séquence d'instructions de sa requête, l'autre doit attendre jusqu'à ce que le premier ait terminé sa requête.

Dates	CPU 0		CPU 1		solde
0	Inst.	Reg.	Inst.	Reg.	1000
1 2 3	load solde sub 1000 store solde	1000 0 0			1000 1000 0
4 5 6			load solde add 100 store solde	0 100 100	0 0 100

➔ assurer l'exclusion mutuelle

Comment assurer l'exclusion mutuelle ?

- 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 ? (2)

- Encadrer chaque section critique par des opérations spéciales qui visent à assurer l'utilisation exclusive des objets partagés.
- Si P1 est dans sa section critique et P2 désire entrer dans sa section critique, alors P2 attend que P1 quitte sa section critique

P1

```
entrer_Section_critique(….);
/* début la section critique */
int tmp = solde;
tmp = tmp – 1000;
solde = tmp;
/* fin de la section critique */
sortir_section_critique(…..);
```

P2

```
entrer_Section_critique( …..);
/* début la section critique */
int tmp = solde;
tmp = tmp + 100;
solde = tmp;
/* fin de la section critique */
sortir_section_critique(….);
```

Comment assurer l'exclusion mutuelle ? (3)

Quatre conditions sont nécessaires pour réaliser correctement une exclusion mutuelle :

1. Deux processus ne peuvent être en même temps dans leurs sections critiques.

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 de sa section critique ne doit bloquer les autres processus.

4. 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 restaure l'état (masqué ou non masqué) à la fin de la section critique.
- Il ne peut être alors suspendu durant l'exécution de la section critique.

Problèmes

- La désactivation est généralement réservée au système d'exploitation. Un processus en espace utilisateur ne peut pas désactiver les interruptions, sinon il pourrait empêcher l'ordonnanceur de s'exécuter en bloquant les interruptions de l'horloge.
- Si les interruptions ne sont pas réactivées, alors le système ne réagirait plus aux événements et ne fonctionnerait plus normalement.
- 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.

Masquage des interruptions (2) : exemple (Linux)

```
local_irq_disable();
/* interrupts are disabled .. */

   Section_critique ();

local_irq_enable();
```

```
unsigned long flags;
local_irq_save(flags);
/* interrupts are now disabled */

   Section_critique ();

local_irq_restore(flags);
/* interrupts are restored to their previous
state */
```

```
/* linux/kernel/sched/core.c */
static int migration_cpu_stop(void *data)
{
    struct migration_arg *arg = data; ...
    local_irq_disable();
    __migrate_task(.arg->task, smp_processor_id(),  arg->dest_cpu);
    local_irq_enable();
    return 0;
}
```

http://lxr.free-electrons.com/source/kernel/sched/core.c?a=powerpc#L4684

Masquer les interruptions pendant la migration de la tâche

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.
- 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.

```
Processus P1
while (1)
{    /*attente active*/
     entrer_region(1) ;
     section_critique_P1() ;
     quitter_region(1);
….
}
```

```
Processus P2
while (1)
{    /*attente active*/
     entrer_region(2) ;
     section_critique_P2() ;
     quitter_region(2);
….
}
```

Solution de Peterson (2)

```
// Les variables « tour » et « interesse » sont partagées par les deux processus..
int tour;
Bool interesse[2];
void entrer_region(int process)
{    int autre;
     if(process==1) autre = 2; else autre=1;    //l'autre processus
     interesse[process] = TRUE;  //indiquer qu'on est intéressé
     tour = process; //la course pour entrer se gagne ici
     while (tour == process && interesse[autre] == TRUE);
}

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

Problème : attente active = consommation du temps CPU

Instructions atomiques

- Instruction exécutant plusieurs opérations de manière indivisible
  - Fetch-And-Add (Fetch_Add)
  - Test-And-Set-Lock (TSL)
  - Compare-And-Swap (CmpXchg)

Le bus est verrouillé durant l'exécution de leurs instructions.

- Permet d'implémenter des primitives de synchronisation efficacement

```
int Fetch_Add(int &x)
 {
    int tmp = x;
    x = tmp + 1;
    return tmp;
}
```

```
int TSL(int &x)
 {
    int tmp = x;
    x =  1;
    return tmp;
}
```

```
int CmpXchg (int& r, int& d, int s)
 {
     if (r == d)
     {  // valeur lue dans r non modifiée
          d = s;
          return 1;
     }
     r = d;
     return 0;
 }
```

Instructions atomiques (2)

Implémentation d'un verrou actif (spin_lock) en utilisant TSL :

La boucle se répète tant que le verrou n'est pas obtenu.

```
int lock = 0;
```

```
Processus P1
while (1)
{
   while(TSL(lock)!=0);
   section_critique_P1() ;
    lock=0;
}
```

```
Processus P2
while (1)
{
   while(TSL(lock)!=0);
   section_critique_P2() ;
    lock=0;
}
```

```
int TSL(int &x)
 {
    int tmp = x;
    x =  1;
    return tmp;
}
```

Instructions atomiques (3)

Implémentation d'un verrou actif (spin_lock) en utilisant CmpXchg :

La boucle se répète tant que le verrou n'est pas obtenu.

```
int lock = 0;
```

```
Processus P1
while (1)
{   int r= 0;
   while(CmpXchg(r,lock,1)==0) r=0;
   section_critique_P1() ;
    lock=0;
}
```

```
Processus P2
while (1)
{   int r= 0;
   while(CmpXchg(r,lock,1)==0) r=0;
   section_critique_P2() ;
    lock=0;
}
```

```
CmpXchg (r = 0, lock, 1)
 {
     if (r == lock) // lock==0
     {    lock = 1; return 1;}
     r = lock;
     return 0;
 }
```

Instructions atomiques (4) : Problèmes

- attente active = consommation du temps CPU

- un processus en attente active peut rester en permanence dans la boucle (boucle infinie). Supposez que :

  1. 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.
  2. Pendant que H est en E/S, le processus B entre en section critique.
  3. Ensuite, le processus H devient prêt alors que B est toujours en section critique.
  4. 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).
  5. B ne peut plus sortir de sa critique et H boucle indéfiniment.

➔ Inversion de priorités

Les primitives SLEEP et WAKEUP

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

```
int lock = 0;
```

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

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

- 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 pourrait dormir pour toujours.

Résumé des problèmes

- Attentes actives → consommation du temps CPU.
- Masquage des interruptions → dangereuse pour le système
- TSL → attente active + inversion des priorités => boucle active 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 au moins un nom et une valeur initiale.

- Les sémaphores sont manipulés au moyen des opérations :
  - P() désigné aussi par down ou wait (Proberen)
  - V() désigné aussi par up ou signal (Verhogen)

Sémaphores (2)

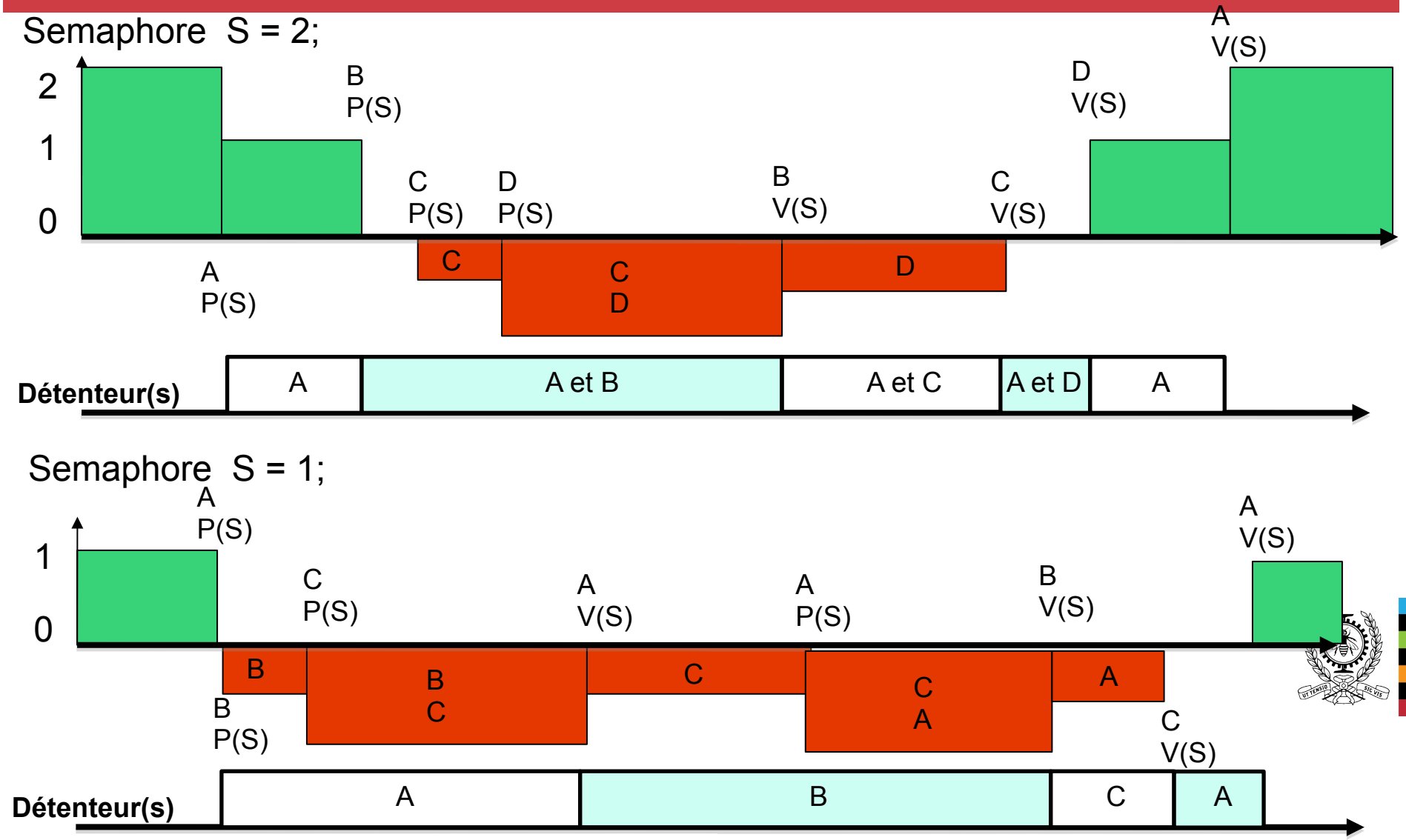
- L'opération P(S) décrémente la valeur du sémaphore S si cette dernière est supérieure à 0. Sinon le processus appelant est mis en attente du sémaphore.

- 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 est choisi et redevient prêt.

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

- Chaque sémaphore a une file d'attente.
  - File FIFO : sémaphore fort (par défaut)
  - File LIFO : sémaphore faible (famine possible)

Sémaphores (3)



Sémaphores (4) : 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

Exercice 1

a) Considérez les processus A et B suivants :

Semaphore x=1 ;

A	B
P(x)	P(x)
a	b
c	d
V(x)	V(x)

Donnez, sous forme d'un arbre, les différents ordres d'exécution possibles des actions atomiques a, b, c et d dans le cas où les processus sont lancés en concurrence.

b) Considérez les 3 processus A, B et C suivants :

Semaphore x=0 ;

A	B	C
a1	P(x)	P(x)
V(x)	b1	c1
	b2	c2
	V(x)	V(x)

Donnez, sous forme d'un arbre, les différents ordres d'exécution possibles des actions atomiques a1, b1, b2, c1 et c2, dans le cas où les processus sont lancés en concurrence.

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
     pthread_join(threadA,NULL); // attendre la fin des threads
       pthread_join(threadB,NULL);
     printf("ici main, fin threads : var_glob =%d \n",var_glob);
  return 0;
}
```

Sémaphores POSIX (4) : Exemple 1

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

Exercice 2

Trois threads concurrents T0, T1 et T2 exécutent chacun le bout de code suivant :

```
Ti ()  // i = 0,1,2
{       int c=0;
        while(true)
           printf("cycle %d de  %d", c++, i);
}
```

1) Synchronisez les cycles des threads à l'aide de sémaphores de manière à exécuter les cycles dans cet ordre :

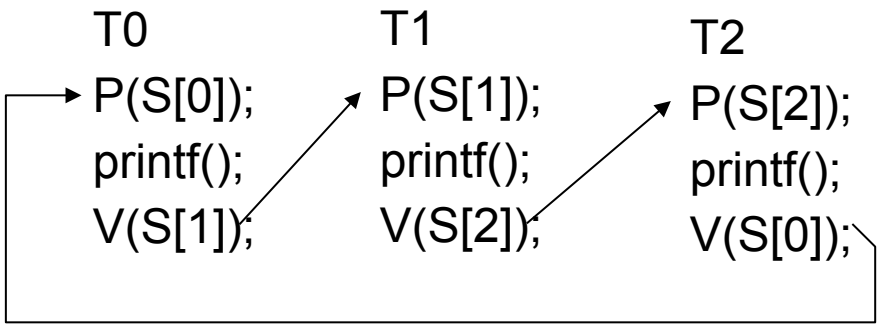
cycle de T0; cycle de T1; cycle de T2; cycle de T0; cycle de T1; ….

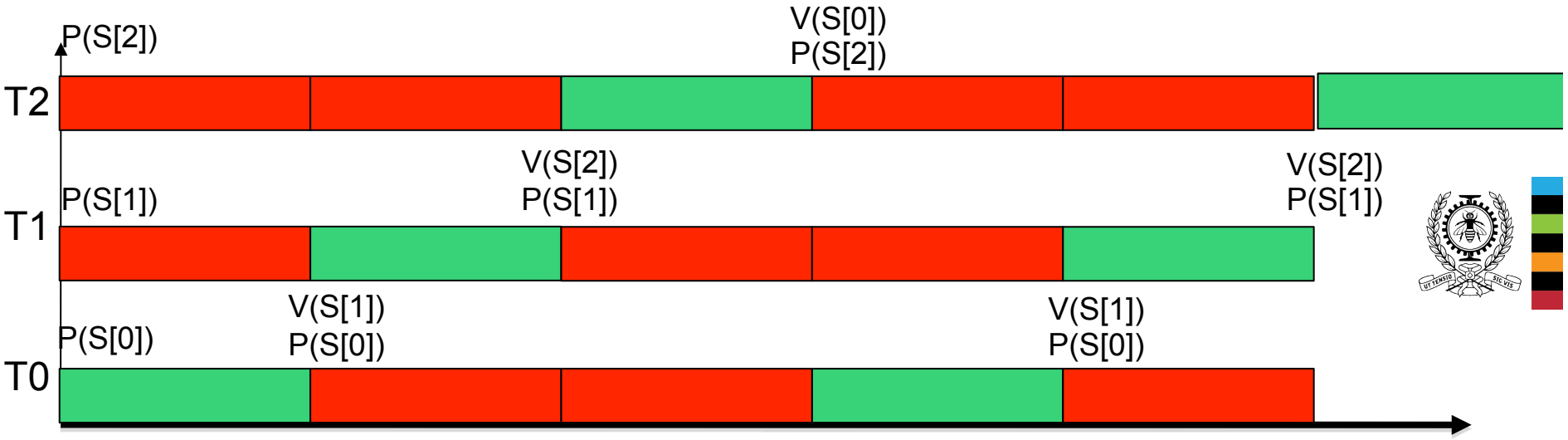
2) Synchronisez les cycles des threads à l'aide de sémaphores pour que :

- chaque cycle de T0 s'exécute en concurrence avec un cycle de T1, et
- le thread T2 exécute un cycle, lorsque T0 et T1 terminent tous les deux l'exécution d'un cycle et ainsi de suite.

Exercice 2.1 (solution)

```
const int n = 3;
sem_t S[n] ; // initialisation dans la fonction main S[0]=1, S[i]=0 pour i>0
void *tourAtour(void *num) {
    int  i = *((int*) num), c=0;
    while(1)) {
        sem_wait(&S[i]);
        printf("Cycle %d de %d\n", c++,i);
        sem_post(&S[(i+1)%n);
    }
}
```





Implémentation des sémaphores (1)
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 (2)
Utilisation de Test-and-Set-Lock (TSL)

Remarques :

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

- 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 ou priorité dynamique.

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).

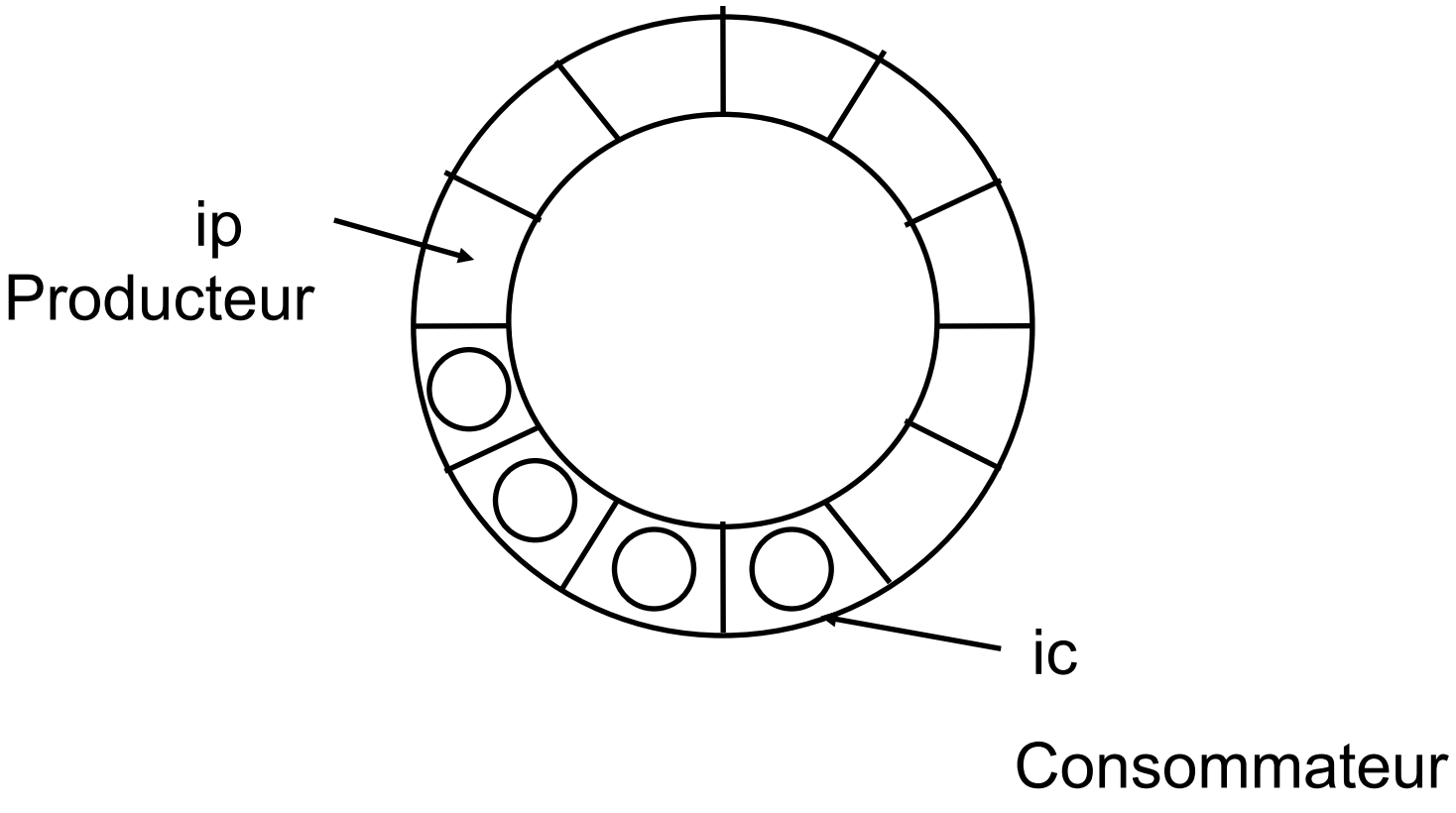
Producteur

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

Consommateur

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

Problème du producteur/consommateur (2)

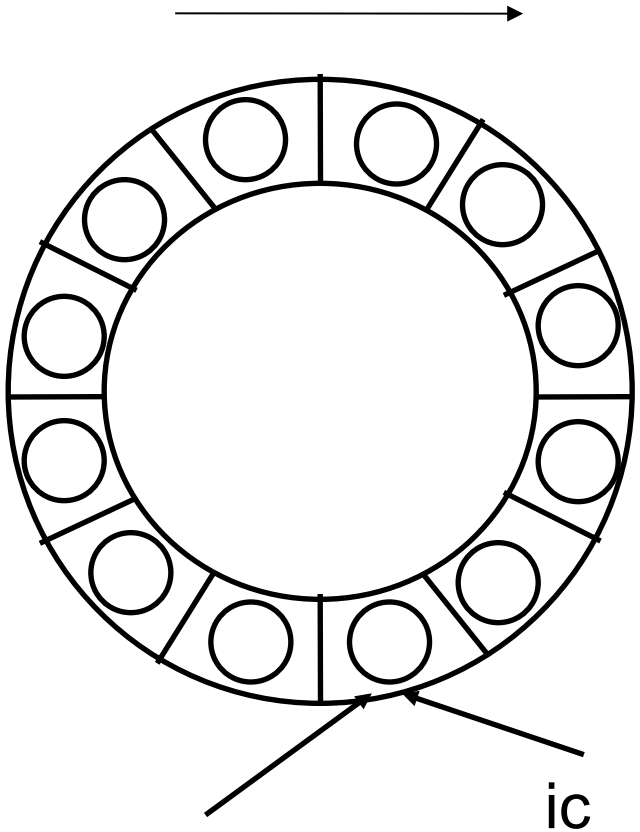


Problème du producteur/consommateur (3)

tampon plein

occupe = N;

libre = 0



ip
Producteur

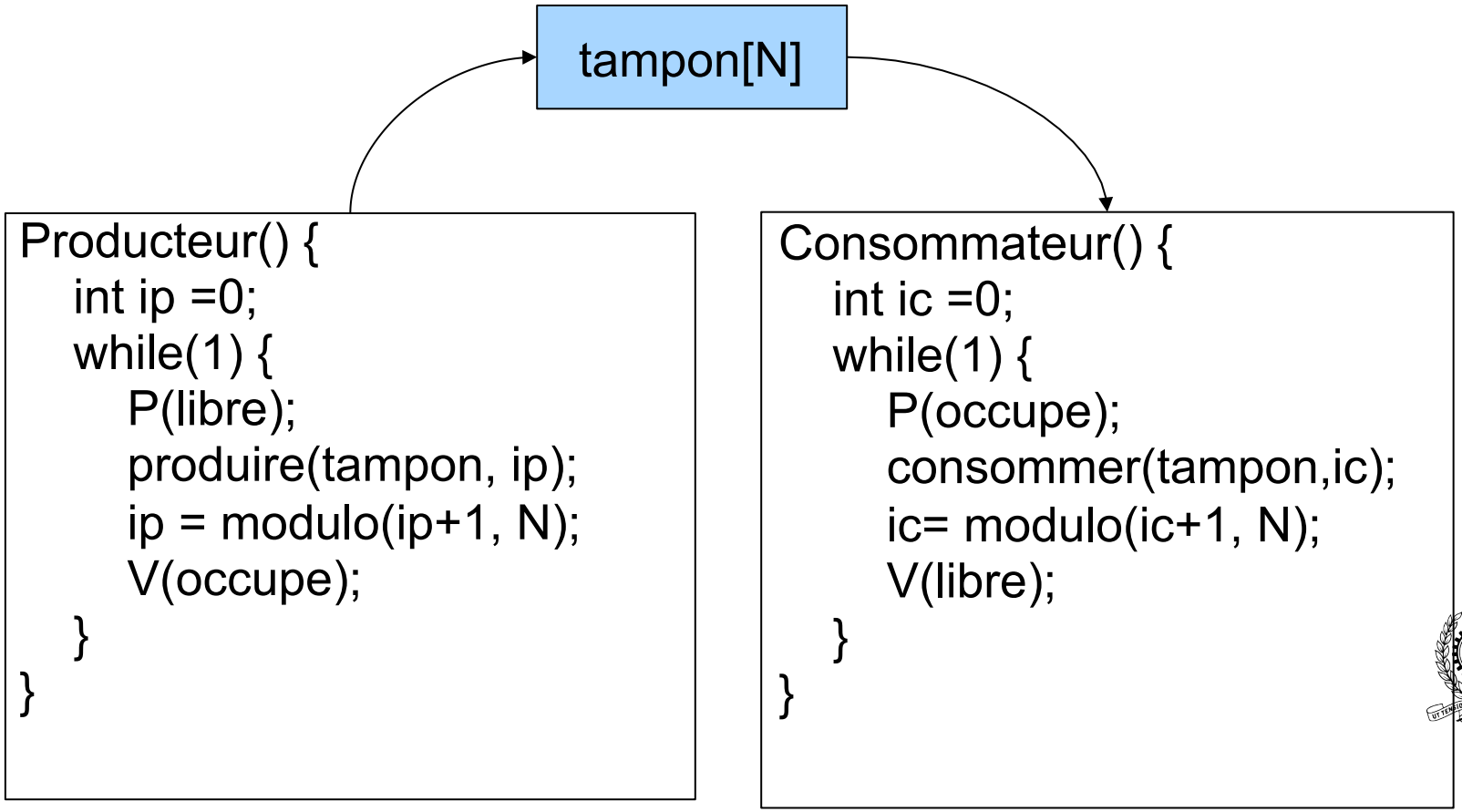
ic
Consommateur

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 étant la taille du tampon).
- Il sert à bloquer/débloquer le producteur (P(libre) et V(libre)).

Problème du producteur/consommateur (5)

Semaphore libre=N, occupe=0;



Problème du producteur/consommateur (6)

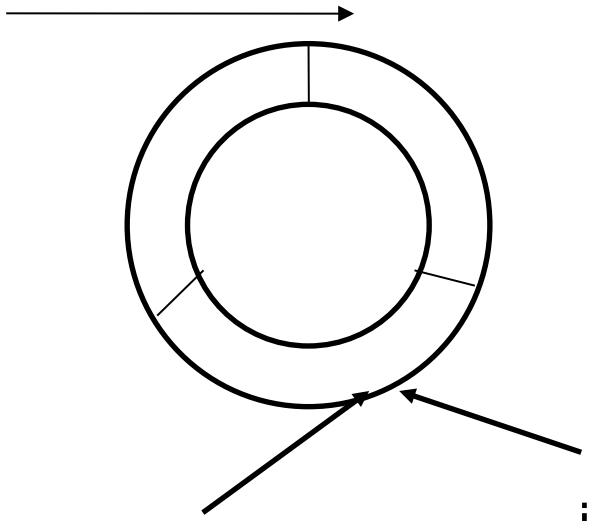
occupe = 0;

libre = 3

⇓

Consommateur bloqué

Producteur peut produire jusqu'à 3



ip
Producteur

ic
Consommateur

Une production

Problème du producteur/consommateur (7)

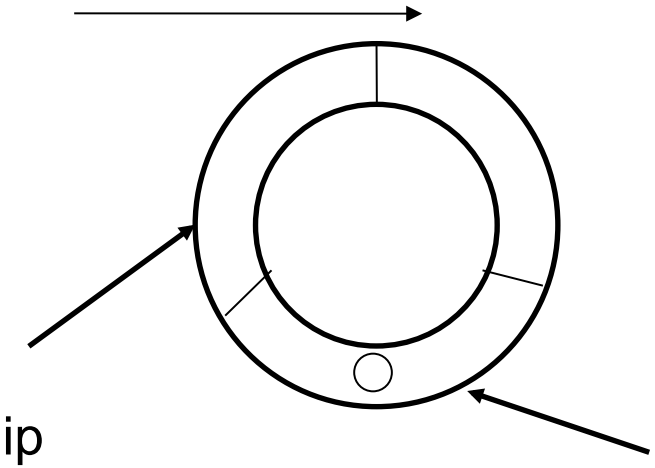
occupe = 1;

libre = 2

⇓

Consommateur peut consommer 1

Producteur peut produire 2



→ Une production

Problème du producteur/consommateur (8)

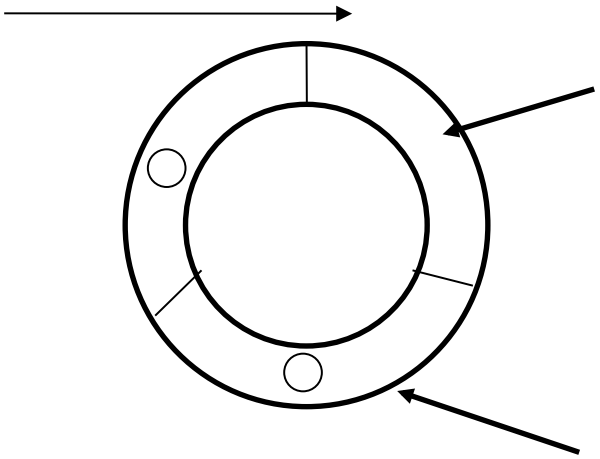
occupe = 2;

libre = 1

⇓

Consommateur peut consommer 2

Producteur peut produire 1



→ Une production

Problème du producteur/consommateur (9)

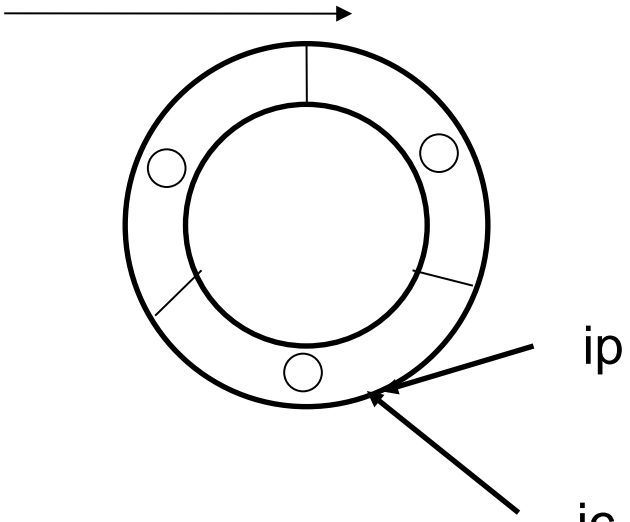
occupe = 3;

libre = 0

⇓

Consommateur peut consommer 3

Producteur bloqué par P(libre)



→ Une consommation

Problème du producteur/consommateur (10)

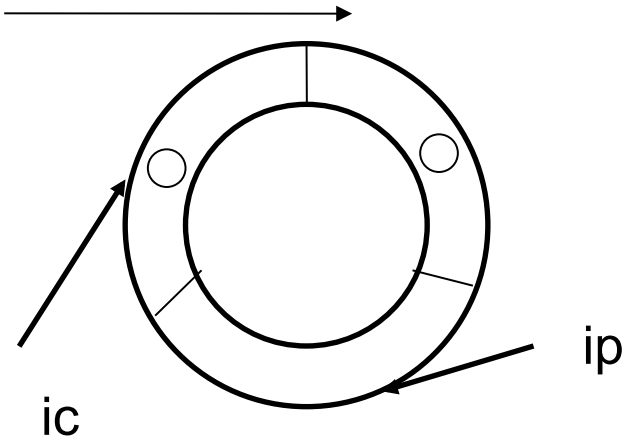
occupe = 2;

libre = 1

⇓

Consommateur peut consommer 2

Producteur peut produire 1



→ Une production

Problème du producteur/consommateur (11)

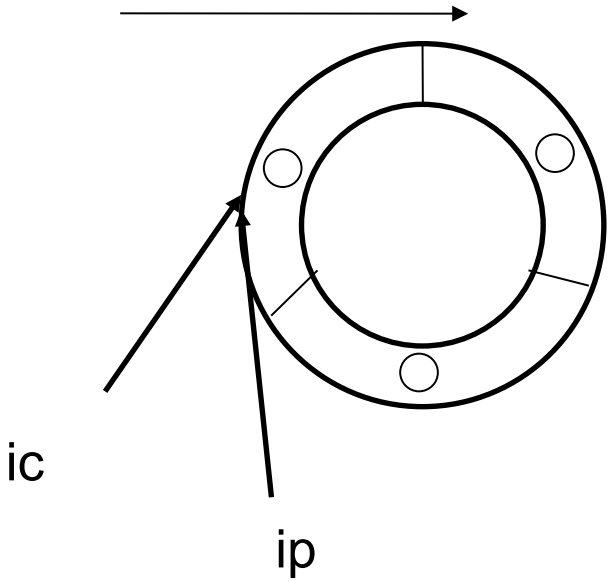
occupe = 3;

libre = 0

⇓

Consommateur peut consommer 3

Producteur bloqué



Problème du producteur/consommateur (12)

//progamme prodcons1.c

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

#define MAX 6
#define BUF_SIZE 3

typedef struct args {
    sem_t sem_free;
    sem_t sem_busy;
} args_t;
void *prod(void *arg);
void *cons(void *arg);
static int buf[BUF_SIZE];
```

```
int main( ) {
    int p, i;
    pthread_t t1, t2;
    args_t args;
    sem_init(&args.sem_free, 0, BUF_SIZE);
    sem_init(&args.sem_busy, 0, 0);
    pthread_create(&t1, NULL, prod, &args);
    pthread_create(&t2, NULL, cons, &args);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("exit\n");
    return EXIT_SUCCESS;
}
```

Problème du producteur/consommateur (13)

```
void *prod(void *arg) {
    int ip = 0, nbprod = 0, obj = 1001;
    args_t *args = (args_t *) arg;
    while(nbprod < MAX) {
        sem_wait(&args->sem_free);
        buf[ip] = obj;
        sem_post(&args->sem_busy);
        printf("prod: buf[%d]=%d\n", ip, obj);
        obj++;
        nbprod++;
        ip = (ip + 1) % BUF_SIZE;
    }
    return NULL;
}
```

```
void *cons(void *arg) {
    int ic = 0, nbcons = 0, obj;
    args_t *args = (args_t *) arg;
    while(nbcons < MAX) {
        sleep(1);
        sem_wait(&args->sem_busy);
        obj = buf[ic];
        sem_post(&args->sem_free);
        printf("cons: buf[%d]=%d\n", ic, obj);
        nbcons++;
        ic = (ic + 1) % BUF_SIZE;
    }
    return NULL;
}
```

Problème du producteur/consommateur (14)

```
$ ./src/prodcons1
prod: buf[0]=1001
prod: buf[1]=1002
prod: buf[2]=1003
cons: buf[0]=1001
prod: buf[0]=1004
cons: buf[1]=1002
prod: buf[1]=1005
cons: buf[2]=1003
prod: buf[2]=1006
cons: buf[0]=1004
cons: buf[1]=1005
cons: buf[2]=1006
exit
```

Le fil prod bloque quand le tampon est plein

Alternance entre prod et cons

Le fil cons traite tous les objets jusqu'à ce que le tampon soit vide.

Exercice 3 :

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

Cas de n producteurs / m consommateurs

```
int tampon [N];
int ip=0,ic=0;
Semaphore libre=N, occupe=0,
    mutex=1;
Producteur ()
{    while (1)
     {    P(libre) ;
          P(mutex);
          produire(tampon, ip);
          ip = modulo(ip +1,N);
          V(mutex);
          V(occupe);
     }
}
```

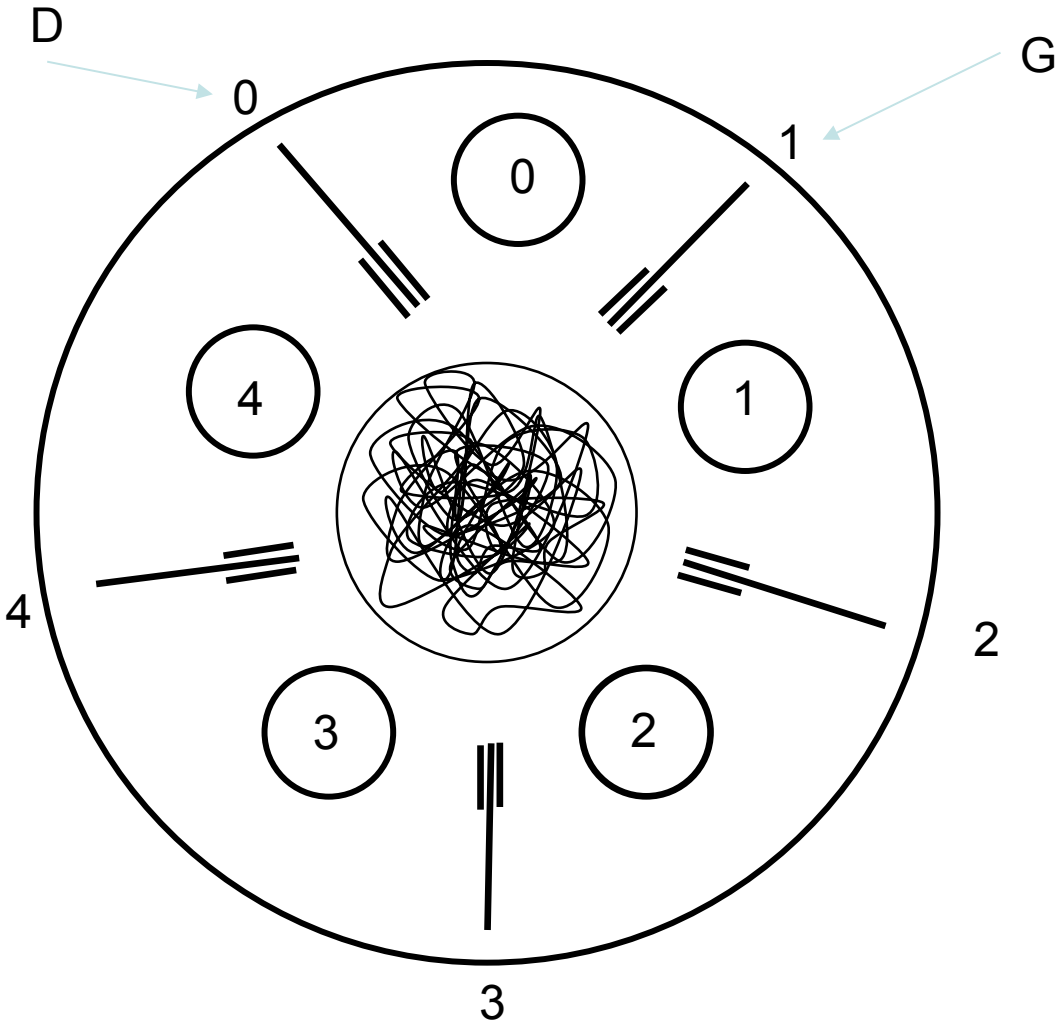
```
Consommateur ()
 {
        while(1)
        {   P(occupe);
            P(mutex);
            consommer(tampon,ic);
            ic= modulo(ic+1, N);
            V(mutex);
            V(libre);

        }
}
```

Problème des philosophes

- 5 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 doivent se faire en exclusion mutuelle. On utilisera le sémaphore mutex pour réaliser l'exclusion mutuelle.

Problème des philosophes (2)



Problème des philosophes (3)

```
// programme philosophe.c

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

#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;
```

Problème des philosophes (4)

```
int main() {
    int i, NumPhi[N] = { 0, 1, 2, 3, 4 };
    pthread_t th[N];

    sem_init(&mutex, 0, 1);

    // création des N philosophes
    for (i = 0; i < N; i++)
        pthread_create(&th[i], NULL, philosophe, &NumPhi[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)

```
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] == libre && fourch[D] == libre) {
            fourch[G] = occupe;
            fourch[D] = occupe;
            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] = libre;
            fourch[D] = libre;
            sem_post(&mutex);
        } else  sem_post(&mutex);
    }
}
```

Prendre les deux fourchettes ou aucune

Problème des philosophes (6)

```
$ ./src/philosophe
philosophe[4] mange
philosophe[2] mange
philosophe[4] a fini de manger
philosophe[2] a fini de manger
philosophe[1] mange
philosophe[4] mange
philosophe[1] a fini de manger
philosophe[4] a fini de manger
philosophe[3] mange
philosophe[1] mange
philosophe[3] a fini de manger
philosophe[1] a fini de manger
philosophe[0] mange
philosophe[3] mange
philosophe[0] a fini de manger
philosophe[3] a fini de manger
philosophe[2] mange
philosophe[0] mange
philosophe[2] a fini de manger
philosophe[0] a fini de manger
fin des  threads
```

Problème des philosophes (7) : Famine

- La solution précédente résout le problème d'interblocage. Mais, un philosophe peut mourir de faim, s'il ne parvient jamais à 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 (ou des ressources), il obtient satisfaction au bout d'un temps fini.

- Dans le cas des philosophes, le problème de famine peut être atténuée, en ajoutant N sémaphores (un sémaphore par fourchette). Les sémaphores sont initialisés à 1.

- Chaque philosophe i demande les deux fourchettes, une à une, en commençant par la plus petite en appelant dans le bon ordre P(f[G(i)]) et P(f[D(i)]).

- Lorsqu'un philosophe i termine de manger, il libère les fourchettes en réalisant V[f[G(i)]) et V[f[D(i)]).

Problème des philosophes (7) : Famine

```
#define N 5         // nombre de philosophes
#define G (i+1)%N   // fourchette gauche du philosophe i
#define D i         // fourchette droite du philosophe i

sem_t f[N]; // à initialiser dans main

void * philosophe(void * num) {
    int i = *(int *) num, nb = 2;
    while (nb) {
        sleep(1);           // penser
        if (G < D) {  sem_wait(&f[G]); sem_wait(&f[D]);}
        else {  sem_wait(&f[D]); sem_wait(&f[G]); }
            nb--;
            printf("philosophe[%d] mange\n", i);
            sleep(1);           // manger
            printf("philosophe[%d] a fini de manger\n", i);
            sem_post(&f[G]); sem_post(&f[D])
    }
}
```

Problème des Lecteurs / Rédacteurs

- Ce problème modélise les accès à une base de données. Plusieurs processus tentent périodiquement d'accéder à la base de données soit pour écrire ou pour lire des informations.
- Pour assurer la 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)

```
int db = 42;
int NbL = 0;
sem_t mutex;
sem_t redact;

void *lecteur(void *arg) {
    while(1) {
        sem_wait(&mutex);
        if (NbL == 0)
            sem_wait(&redact);
        NbL++;
        sem_post(&mutex);
        // lecture de la base
        sleep(1);
        printf("lecteur   bd=%d\n", db);
        // fin de l'accès à la base
        sem_wait(&mutex);
        NbL--;
        if(NbL == 0)
            sem_post(&redact);
        sem_post(&mutex);
    }
}
```

```
void *redacteur(void *arg) {
    while(1) {
        sem_wait(&redact);
        // modifier les données de la base
        db++;
        printf("redacteur bd=%d\n", db);
        sleep(2);
        sem_post(&redact);
    }
}
```

Exercice 4 :

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

Problème des Lecteurs / Rédacteurs (Solution)

Solution :
Semaphore **tour=1**, mutex=1, redact =1;

```
Redacteur () {
  while(1) {
    P(tour);   // attendre son tour
    P(redact); // assurer l'accès
               // exclusif à la base
    V(tour);
    Ecriture();
    V(redact);
  }
}
```

```
Lecteur () {
  while(1) {
    P(tour);  // attendre son tour
    P(mutex) ;
    Nbl++ ;
    if (Nbl==1) P(redact);
    V(mutex);
    V(tour);
    Lecture();
    P(mutex);
    Nbl--;
    if(Nbl==0) V(redact);
    V(mutex);
  }
}
```

Les moniteurs

- **Qu'est ce qu'un moniteur ?**
- **Variables de condition des moniteurs**
- **Exemple : Problème Producteur/consommateur**
- **Implémentation des moniteurs (au moyen de sémaphores)**
- **Moniteurs JAVA**

Qu'est ce qu'un moniteur ?

- Moniteur = { variables globales + sections critiques d'un même problème}
- Pour assurer l'exclusion mutuelle, à tout moment un processus au plus est actif dans le moniteur.
- Si un processus est actif dans le moniteur et un autre demande à y accéder (appelle une procédure du moniteur), ce dernier est mis en attente.
- C'est le compilateur qui se charge de cette gestion. Pour ce faire, il rajoute au moniteur le code nécessaire pour assurer l'exclusion mutuelle.
- 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.
- La majorité des compilateurs utilisés actuellement ne supportent pas les moniteurs (à l'exception de JAVA, C#).

Qu'est ce qu'un moniteur ? (2)

- Comment utiliser les moniteurs pour assurer l'exclusion mutuelle ?

```
Moniteur Compte
{  int solde  = 0 ;

   void Deposer (int  montant) // section critique pour le dépôt
   {  solde = solde + montant  ;
    }

   void retirer (int montant)    //section critique pour le retrait
   {  if (solde >= montant)
            solde = solde – montant ;
   }
}
```

Qu'est ce qu'un moniteur ? (3)
Risque d'interblocage

- Situation d'interblocage si le processus actif dans le moniteur a besoin d'une ressource détenue par un autre en attente du moniteur.

- Pour éviter des situations d'interblocage, il faut trouver un moyen permettant à un processus actif à l'intérieur d'un moniteur de se suspendre pour laisser place à un autre.

Exemple : Producteur/consommateur

- Le consommateur est actif dans le moniteur et le tampon est vide

=> il devrait se mettre en attente et laisser place au producteur.

- Le producteur est actif dans le moniteur et le tampon est plein

=> il devrait se mettre en attente et laisser place au consommateur.

Qu'est ce qu'un moniteur ? (4)
Risque d'interblocage

```
Moniteur ProducteurConsommateur
{    const int N= 100;
     int tampon[N];
     int compteur =0, ic=0, ip=0 ;

     void mettre (int objet) // section critique pour le dépôt
     {          while (compteur==N);
                tampon[ip] = objet ;
                ip = (ip+1)%N ;
                compteur++ ;
      }

     void retirer (int& objet) //section critique pour le retrait
     {          while (compteur ==0) ;
                objet = tampon[ic] ;
                ic = (ic+1)%N ;
                compteur -- ;
     }
}
```

Attente infinie dans le moniteur si le consommateur accède en premier alors que le tampon est vide.
Le producteur est en attente du moniteur.

Variables de condition des moniteurs

- Une variable de condition (des moniteurs) est une condition manipulée au moyen de deux opérations wait and signal.

- wait(x) :
  - suspend l'exécution du processus (thread) appelant (le met en attente de x);
  - autorise un autre processus en attente du moniteur à y entrer.

- signal(x) :
  débloque un processus en attente de la condition x.

- Le processus débloqué est :
  - mis en attente du moniteur (**sémantique « signal and continue »**), OU
  - activé dans le moniteur (**sémantique « signal and wait »**),
    => le processus appelant est dans ce cas mis en attente.

Suspendu dans le moniteur (proposée par Hoare)

Exemple : problème Producteur/consommateur

- Les sections critiques du problème du producteur et du consommateur sont les opérations de dépôt et de retrait du tampon partagé.

- Un dépôt est possible uniquement si le tampon n'est pas plein. Pour bloquer le producteur tant que le buffer est plein, il suffit d'utiliser une variable de condition nplein et de précéder chaque opération de dépôt par l'action wait(nplein), si le tampon est plein.

- L'action signal(nplein) sera appelée suite à un retrait d'un buffer plein.

- Un retrait est possible uniquement si le tampon n'est pas vide. Pour bloquer le consommateur tant que le buffer est vide, il suffit d'utiliser une variable de condition nvide et de précéder chaque opération de retrait par l'action wait(nvide), si le tampon est vide.

- L'action signal(nvide) sera appelée par l'opération de dépôt dans le cas d'un dépôt dans un buffer vide.

Exemple : problème Producteur/consommateur (2)

```
Moniteur ProducteurConsommateur
{   boolc nplein, nvide ; //variable de condition pour non plein et non vide
    const int N= 100;
    int tampon[N], compteur =0, ic=0, ip=0 ;

    void mettre (int objet)    // section critique pour le dépôt
    {   //attendre jusqu'à ce que le tampon devienne non plein
        if (compteur==N) wait(nplein) ;
                tampon[ip] = objet ;
        ip = (ip+1)%N ;
        compteur++ ;
      // si le tampon était vide, envoyer un signal pour réveiller le consommateur.
        if (compteur==1) signal(nvide) ;
    }
    void retirer (int& objet)  //section critique pour le retrait
    {   if (compteur ==0) wait(nvide) ;
        objet = tampon[ic] ;
        ic = (ic+1)%N ;
        compteur -- ;
        // si le tampon était plein, envoyer un signal pour réveiller le producteur.
        if(compteur==N-1) signal(nplein) ;
    }
}
```

Exercice 5

- Implémentez, en utilisant les moniteurs et les variables de conditions, les sémaphores (les primitives P et V).

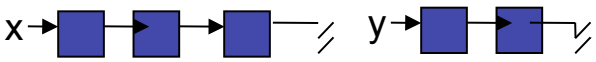
Implémentation des moniteurs

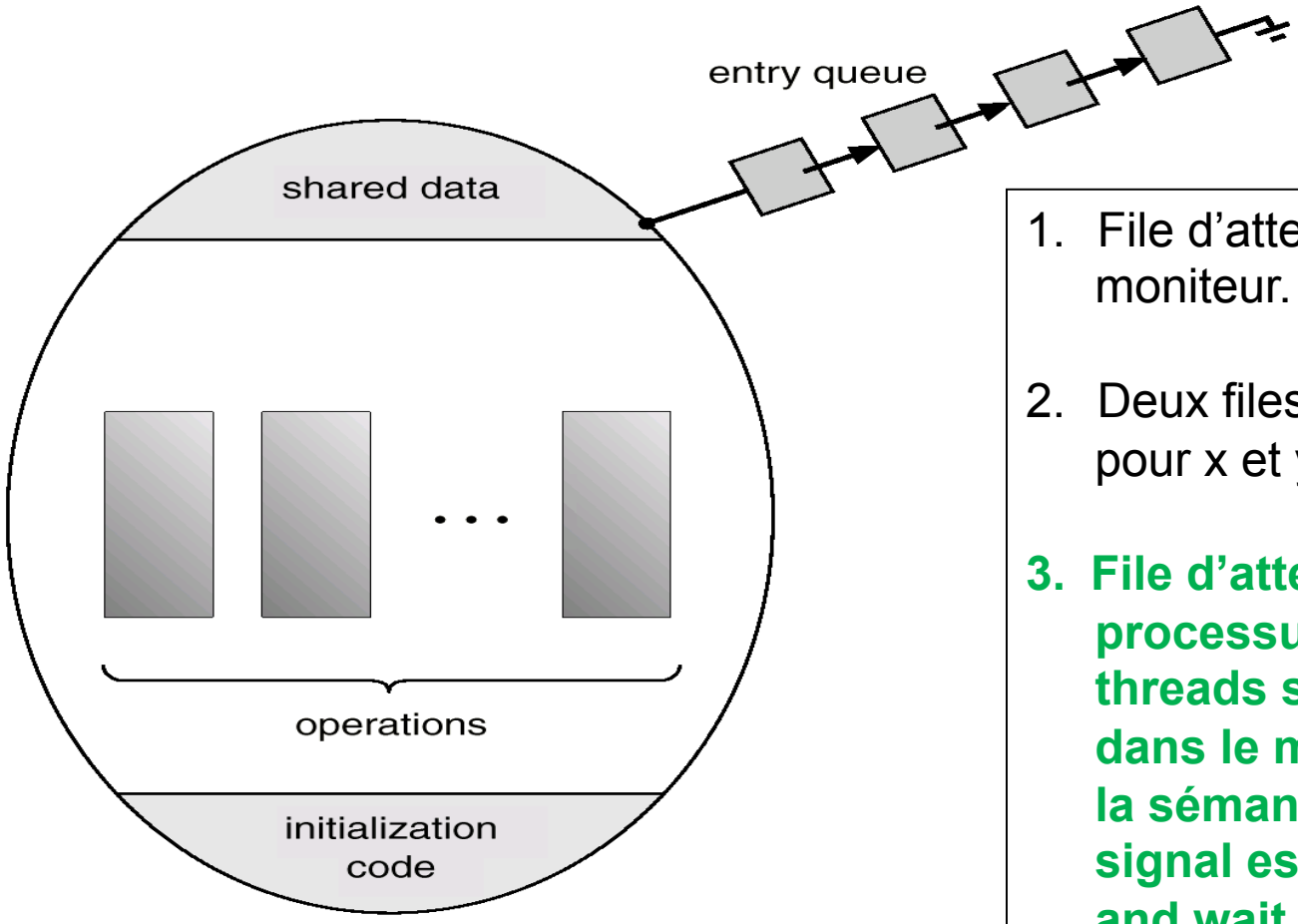
On utilise :

- Une file d'attente pour mémoriser les demandes d'accès au moniteur (file d'attente du moniteur).
- Une file d'attente pour chaque variable de condition (wait(x)).
- Éventuellement, une file d'attente des processus suspendus dans le moniteur suite à l'opération signal(x) (dans le cas où la sémantique est **« signal and wait »**).
- Si cette dernière file est gérée, elle est plus prioritaire que la file d'attente du moniteur.

Implémentation des moniteurs (2)

queues associated with x,y conditions





1. File d'attente du moniteur.
2. Deux files d'attente pour x et y.
3. **File d'attente des processus ou threads suspendus dans le moniteur (si la sémantique de signal est « signal and wait »).**

Implémentation des moniteurs (3) au moyen de sémaphores (cas de « signal and continue »)

- Un sémaphore binaire mutex, initialisé à 1, pour assurer l'accès en exclusion mutuelle au moniteur.

  File du mutex = File d'attente du moniteur

- Pour chaque variable de condition x du moniteur,
  - un sémaphore binaire x-sem initialisé à 0,
    File de x-sem = File d'attente de x
  - un compteur du nombre de processus en attente de x.

Implémentation des moniteurs (4) au moyen de sémaphores (cas de « signal and continue »)

Avant :

```
Function F (/* arguments.. */)
{


    /*corps de la fonction*/


}
```

Après :

```
Function F (/* arguments….*/)
{
    P(mutex);

    /*corps de la fonction*/

    V(mutex);
}
```

Implémentation des moniteurs (5) au moyen de sémaphores (cas de « signal and continue »)

Avant :

boolc x;

wait(x);

Après :

Semaphore x_sem=0;
int x_count = 0;

x_count = x_count + 1;
V(mutex);
P(x_sem);
P(mutex);

Implémentation des moniteurs (6) au moyen de sémaphores (cas de « signal and continue »)

Avant :

signal(x);

Après :

```
si (x_count > 0)
alors
{    x_count = x_count – 1;
     V(x_sem);
}
```

Moniteurs JAVA

- Un moniteur est associé à tout objet JAVA.
- Les méthodes de l'objet qui doivent accéder à l'objet en exclusion mutuelle sont déclarées de type « synchronized ».
- Une seule variable de condition «implicite » et les méthodes wait, notify (l'équivalent de signal) et notifyAll sont associées à tout objet JAVA.
- Les moniteurs JAVA sont de type « signal and continue ».
- Chaque objet JAVA a deux files d'attente :
  - Entry queue : file d'attente du moniteur
  - Wait queue : file d'attente de la variable de condition.

Variables de condition et mutex POSIX

- pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
- int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);
- int pthread_cond_signal(pthread_cond_t *cond);
- int pthread_cond_broadcast(pthread_cond_t *cond);
- int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
- int pthread_cond_timedwait(pthread_cond_t *cond,
  pthread_mutex_t *mutex, const struct timespec *abstime);
- int pthread_cond_destroy(pthread_cond_t *cond);

- **Exemple de code pour mutex, variables de condition et les barrières de pthread :**

  http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp?topic=%2Fapis%2Fusers_78.htm

  http://man7.org/tlpi/code/online/dist/threads/pthread_barrier_demo.c.html

- **Intel® Threading Building Blocks (Intel® TBB) :** est une libraire qui supporte la programmation parallèle et utilise C++.
  http://software.intel.com/sites/products/documentation/doclib/tbb_sa/help/index.htm

Lectures suggérées

- Notes de cours: Chapitre 6
(http://www.groupes.polymtl.ca/inf2610/documentation/notes/chap6.pdf)

- Chapitre 4 (pp 68- 82) et Chapitre 5 (pp 85 – 104)
M. Mitchell, J. Oldham, A. Samuel - Programmation Avancée sous Linux- *Traduction : Sébastien Le Ray* (2001) **Livre disponible dans le dossier Slides Automne 2017 du site moodle du cours.**