Chapitre VI : Mutex et Sémaphore

Cours Système Exploitation II

1 Linfo

ISIMM

2022/2023

Batman et Robin création de threads

- Ecrire un code en c dans lequel thread Batman affiche j'utilise ma Batmobile et thread Robin affiche j'utilise ma moto .

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


void *batman(void *arg) {
   printf("J'utilise ma Batmobile.\n");
   pthread_exit(NULL);
}


void *robin(void *arg) {
   printf("J'utilise ma moto.\n");
   pthread_exit(NULL);
}

int main() {  // thread principal
   pthread_t batman_thread, robin_thread;
   int rc;

   rc = pthread_create(&batman_thread, NULL, batman,
NULL);  // création du thread Batman
 if (rc) {
      printf("Erreur lors de la création du thread Batman.
Code de retour : %d\n", rc);
      exit(-1);
   }

   rc = pthread_create(&robin_thread, NULL, robin, NULL);
// création du thread Robin
   if (rc) {
      printf("Erreur lors de la création du thread Robin. Code
de retour : %d\n", rc);
      exit(-1);
   }

   pthread_join(batman_thread, NULL);
   pthread_join(robin_thread, NULL);

   printf("Les threads Batman et Robin ont terminé leur
exécution.\n");

   pthread_exit(NULL);
}
```

Interblocage

- L'interblocage (deadlock) se produit lorsqu'un groupe de threads est bloqué et ne peut pas continuer à exécuter en raison d'un conflit d'accès aux ressources partagées.

Les ressources

- Ce sont des périphériques ou des variables partagées ou tout ce qui peut faire l'objet d'accès concurrents : mémoire vive, fichiers, les entrées/sorties, des variables globales.

Interblocage

L'**accès concurrent** aux ressources peut provoquer des **interblocages** de processus.

Exemple : Les processus A et B utilisent les ressources R1 et R2

1. A demande l'accès à R1 : **accordé**
2. B demande l'accès à R2 : **accordé**
3. A demande l'accès à R2 : **refusé** car utilisé par B => **A se bloque**
4. B demande l'accès à R1 : **refusé** car utilisé par A => **B se bloque**

Les 2 processus sont bloqués indéfiniment ainsi que les 2 ressources

Ce phénomène peut se produire dans des quantités de circonstances comme l'accès concurrent à des enregistrements d'une BD qui sont verrouillés par les processus.

Modélisation de l'interblocage

On dessine :

- un **processus** par un cercle
- une **ressource** par un carré

Un arc de ressource vers processus indique que cette ressource **est détenue** par ce processus

P détient R

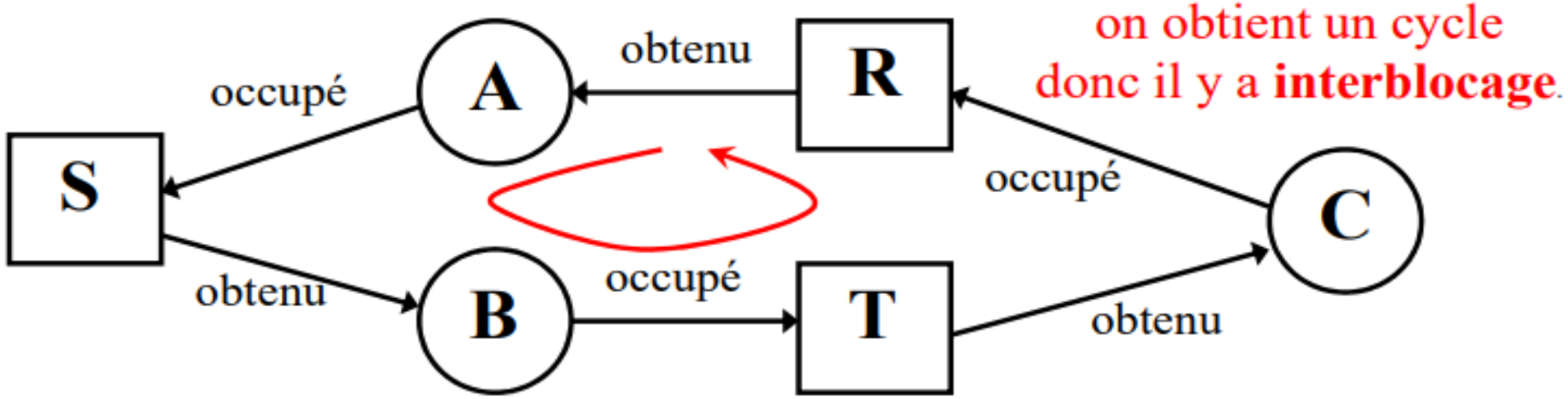
Un arc de processus vers ressource indique que ce processus **demande** cette ressource

P demande S

En faisant ce graphe pour tous les processus et toutes les ressources on met en évidence des cycles => des **interblocages**.

Exemple d'évolution de 3 processus et 3 ressources

Processus A	Processus B	Processus C
Demander R	Demander S	Demander T
Demander S	Demander T	Demander R
Libérer R	Libérer S	Libérer T
Libérer S	Libérer T	Libérer R



Remarque : si les 3 processus avaient été **ordonnancés différemment** sans rien changer à leur code l'interblocage aurait pu être évité : par exemple si A avait libéré R avant que C ne la demande.

Monitor (ou sémaphore)

- Utilisé pour synchroniser l'accès à une ressource partagée.
- Cette ressource peut-être un segment d'un code donné.
- Un thread accède à cette ressource par l'intermédiaire de ce moniteur.
- Ce moniteur est attribué à un seul thread à la fois.
- Pendant que le thread exécute la ressource partagée aucun autre thread ne peut y accéder.

Monitor (ou sémaphore)

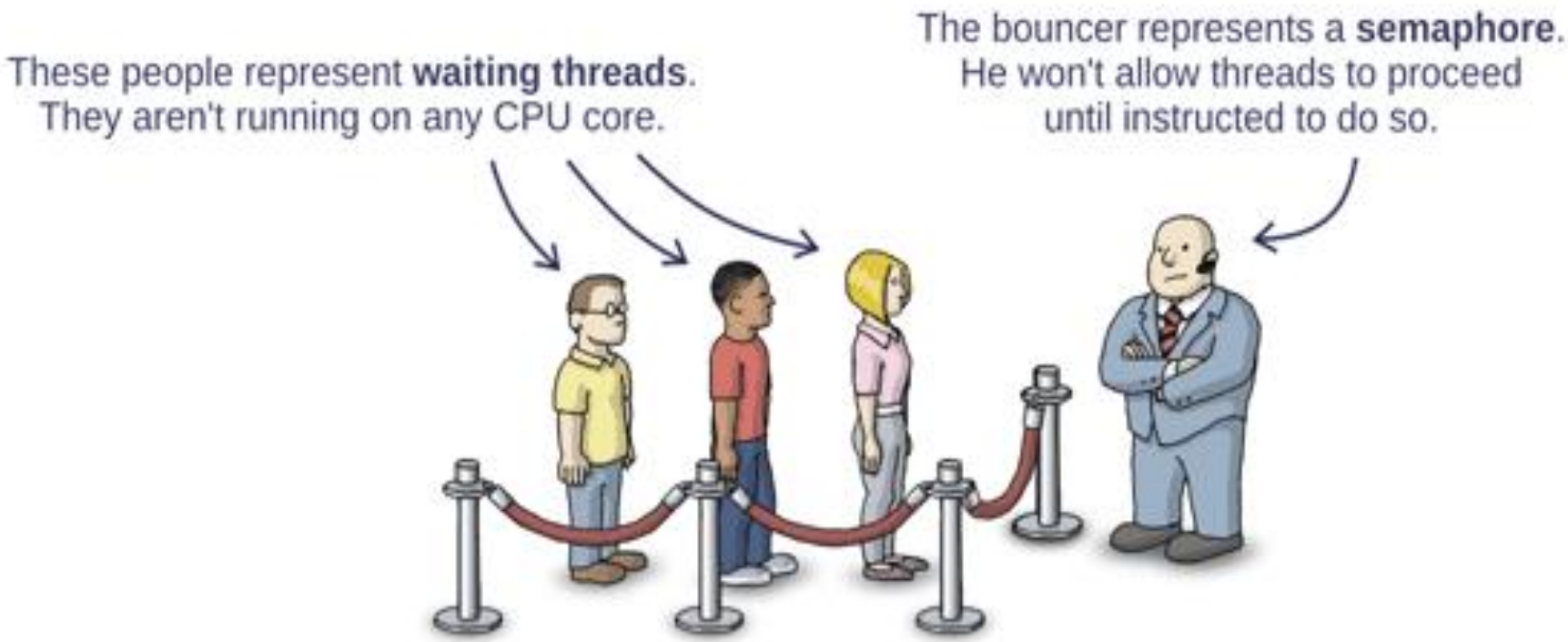
- Le thread libère le monitor dès qu'il a terminé l'exécution du code synchronisé.

- Nous assurons ainsi que chaque accès aux données partagées est bien

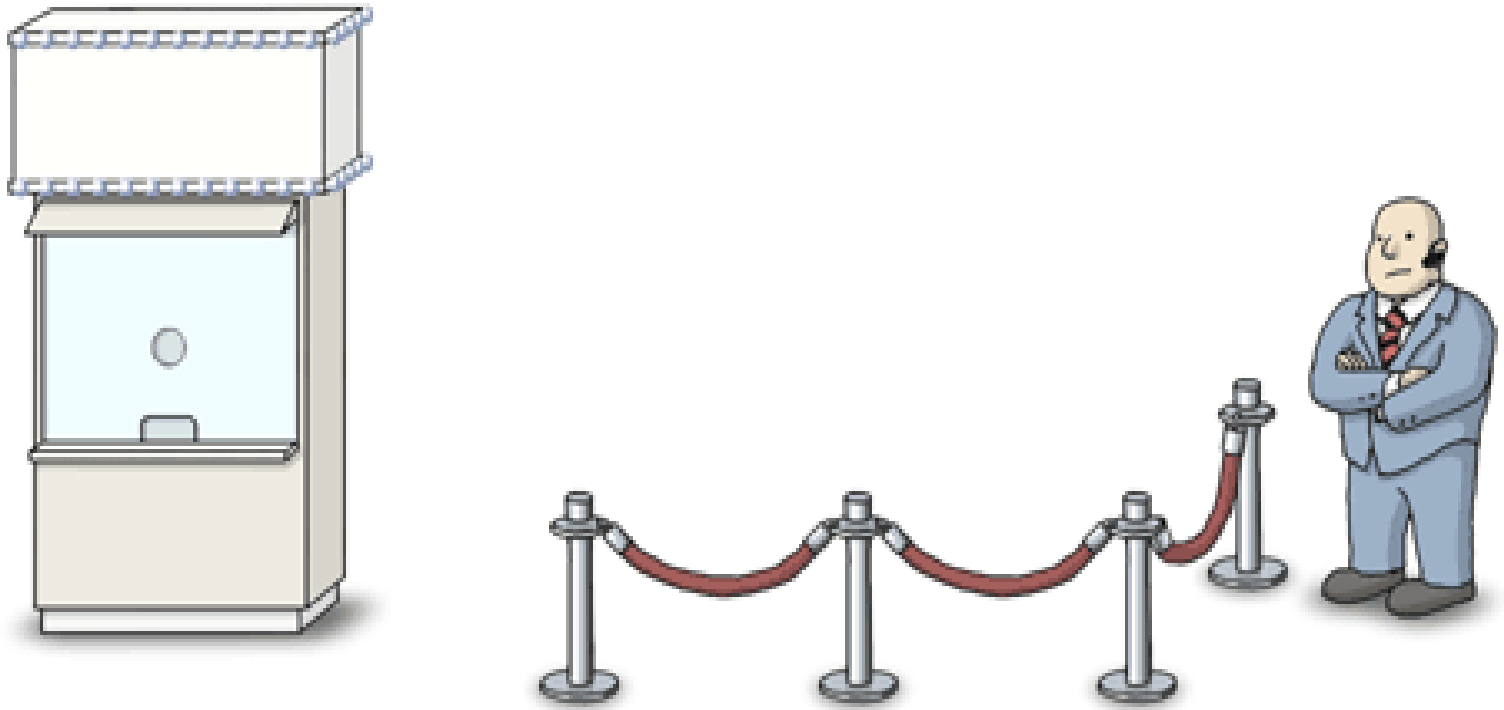
« mutuellement exclusif ».

- Nous allons utiliser pour cela les « mutex » une abréviation pour « Mutual Exclusion » ou « exlusion mutuelle ».

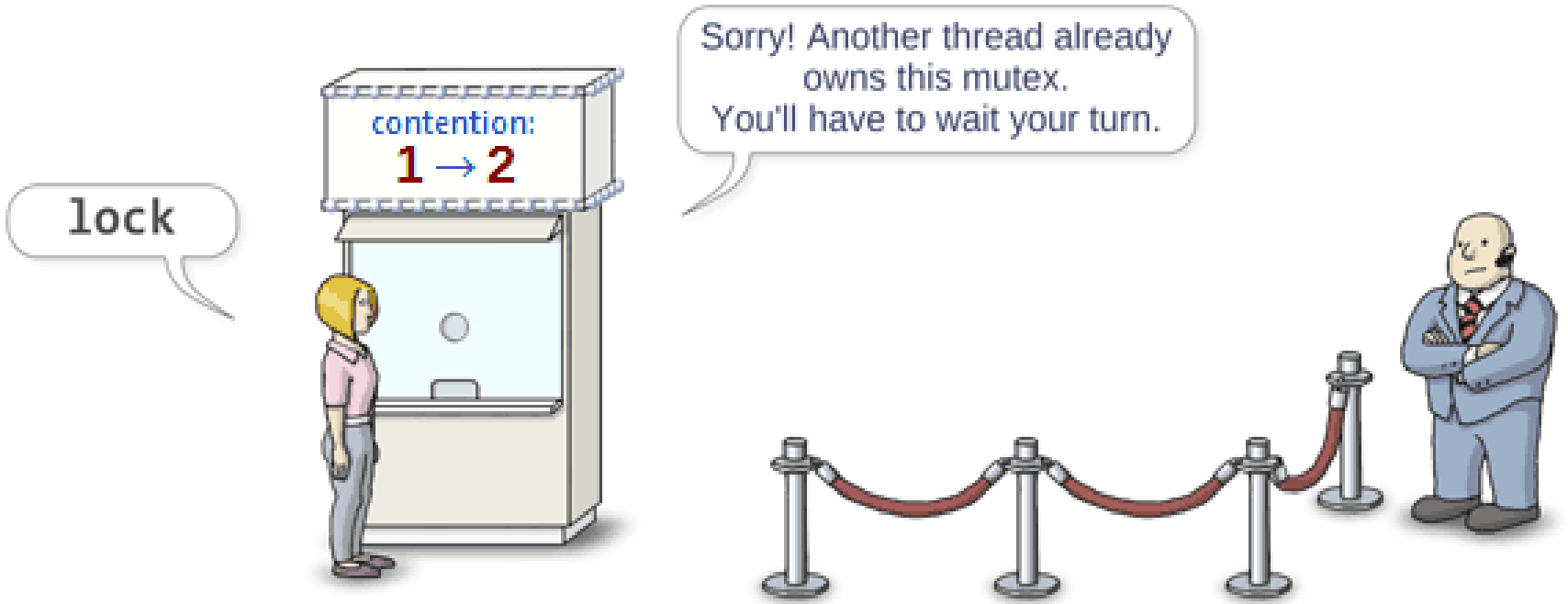
Monitor (ou sémaphore)



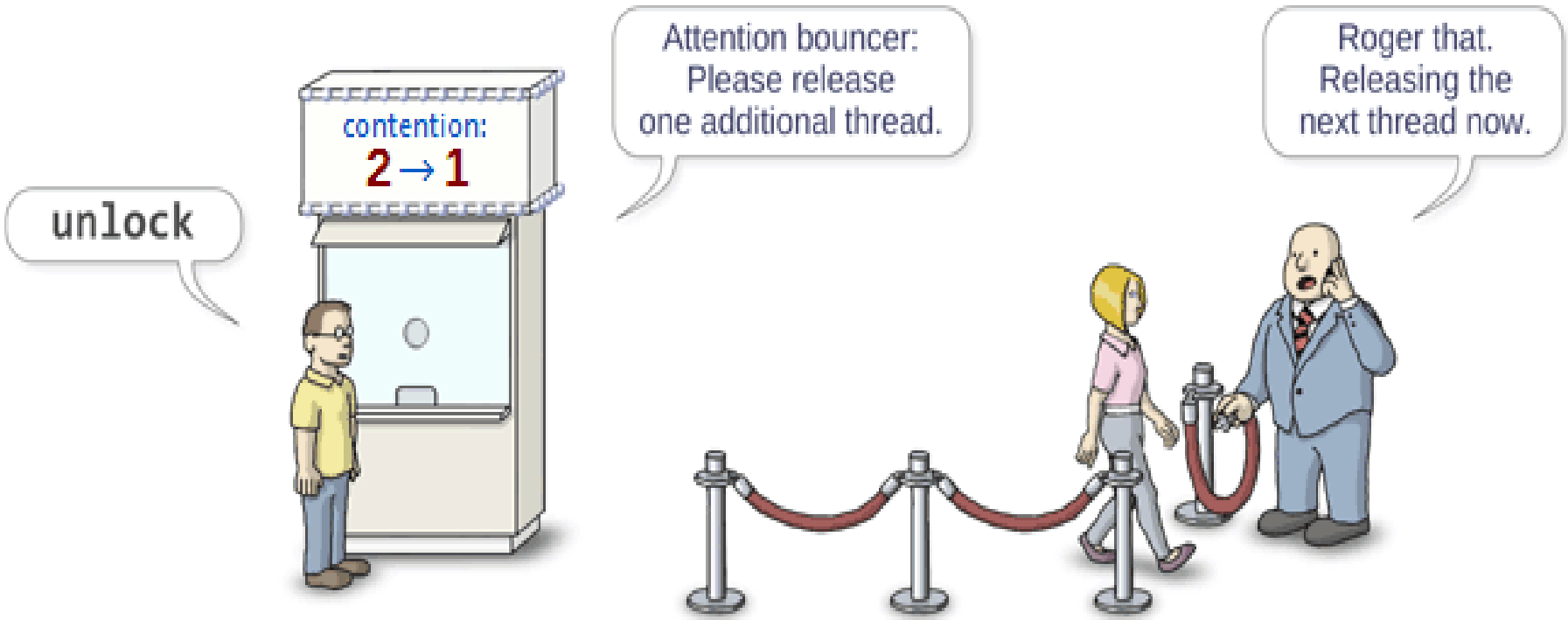
Monitor (ou sémaphore)



Monitor (ou sémaphore)



Monitor (ou sémaphore)



Mutex

- Un mutex est un verrou possédant deux états : déverrouillé (pour signifier qu'il est disponible) et verrouillé (pour signifier qu'il n'est pas disponible).

- Mutex est une variable d'exclusion mutuelle.

- Un mutex est du type « pthread_mutex_t ».


```
pthread_mutex_t mutex;
```

Mutex

- Avant de pouvoir utiliser un mutex, il faudra l'initialiser.
- Nous pouvons initialiser un mutex à l'aide d'une macro.

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

- Nous pouvons initialiser aussi un mutex à l'aide de la fonction «pthread_mutex_init ».

correspond au mutex à initialiser

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

les attributs à utiliser lors de l'initialisation. Nous pouvons utiliser la valeur « NULL ».

Mutex

- Pour détruire un mutex, nous utilisons la fonction «pthread_mutex_destroy».

```
int pthread_mutex_destroy (pthread_mutex_t* mutex);
```

Exemple 1 : Interblocage

Par exemple, si chaque thread attend que l'autre thread libère une ressource avant de pouvoir continuer, cela peut conduire à un interblocage.

Voici un exemple où cela se produit :

- Batman attend la libération de la variable partagée « moto » et incrémente la variable « Batmobile ».
- Robin attend la libération de la variable partagée « Batmobile » et incrémente la variable « moto ».

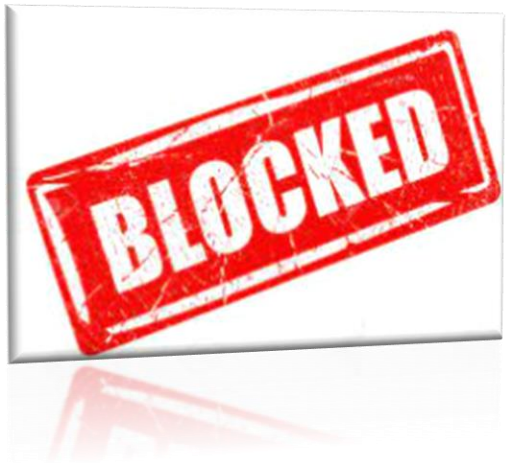
```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int batmobile = 0;  // variable partagée
int moto = 0;  // variable partagée

void *batman(void *arg) {
  printf("Batman utilise la Batmobile.\n");
   batmobile++;
  printf("Batman utilise la moto.\n");
   moto --;
 pthread_exit(NULL);
}

void *robin(void *arg) {
   printf("Robin utilise la moto.\n");
   moto++;
   printf("Robin attend la Batmobile.\n");
   batmobile --;
   printf("Nombre d'utilisations de la moto : %d\n", moto);
   pthread_exit(NULL);
}

int main() {  // thread principal
   pthread_t batman_thread, robin_thread;
   int rc;
```




```c
   rc = pthread_create(&batman_thread, NULL, batman, NULL);  // création du thread Batman
   if (rc) {
      printf("Erreur lors de la création du thread Batman. Code de retour : %d\n", rc);
      exit(-1);
   }

   rc = pthread_create(&robin_thread, NULL, robin, NULL);  // création du thread Robin
   if (rc) {
      printf("Erreur lors de la création du thread Robin. Code de retour : %d\n", rc);
      exit(-1);
   }

   pthread_join(batman_thread, NULL);
   pthread_join(robin_thread, NULL);

   printf("Les threads Batman et Robin ont terminé leur exécution.\n");

   pthread_exit(NULL);
}
```

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

int batmobile = 0;  // variable partagée
int moto = 0;  // variable partagée
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;  // mutex pour synchroniser l'accès aux variables partagées

void *batman(void *arg) {
    pthread_mutex_lock(&mutex);  // verrouillage du mutex
    printf("Batman utilise la Batmobile.\n");
    batmobile++;
   printf("Batman utilise la moto.\n");
    moto--;
   pthread_mutex_unlock(&mutex);  // déverrouillage du mutex
    pthread_exit(NULL);
}

void *robin(void *arg) {
    while (batmobile == 0) { }  // attente de la Batmobile
    pthread_mutex_lock(&mutex);  // verrouillage du mutex
    printf("Robin utilise la moto.\n");
    moto++;
    printf("Robin utilise la batmobile.\n");
    batmobile--;
    pthread_mutex_unlock(&mutex);  // déverrouillage du mutex
    pthread_exit(NULL);
}
```

Utiliser Mutex

```c
int main() {  // thread principal
    pthread_t batman_thread, robin_thread;
    int rc;
    rc = pthread_create(&batman_thread, NULL, batman, NULL);  // création du thread Batman
    if (rc) {
        printf("Erreur lors de la création du thread Batman. Code de retour : %d\n", rc);
        exit(-1);
    }

    rc = pthread_create(&robin_thread, NULL, robin, NULL);  // création du thread Robin
    if (rc) {
        printf("Erreur lors de la création du thread Robin. Code de retour : %d\n", rc);
        exit(-1);
    }

    pthread_join(batman_thread, NULL);
    pthread_join(robin_thread, NULL);

    printf("Les threads Batman et Robin ont terminé leur exécution.\n");

    pthread_exit(NULL);
}
```

Les sémaphores

- Un sémaphore est une variable accessible seulement à l'aide des opérations P (Proberen : tester) et V (Verhogen : incrémenter) suivantes:
- P(S) Puis-je ? → S= S-1
- Si S est nulle ou négative alors le processus/thread réalisant P(S) est bloqué dans un file d'attente spécifique de chaque sémaphore.
- P est utilisée pour prendre (acquérir) le sémaphore avant d'accéder à la ressource partagée.

- V(S) Vas-y → S=S+1
- Si S > 0 alors un des processus/thread de la file d'attente est débloqué.
- V est utilisée pour libérer (relâcher) le sémaphore après avoir terminé l'accès à la ressource partagée.

Sémaphore sous Linux

prend comme argument un pointeur vers le sémaphore à initialiser.
Value est la valeur initiale, positive ou nulle, du sémaphore.

```
#include <semaphore.h>

int sem_init(sem_t *sem, unsigned int value);

 int sem_destroy(sem_t *sem);

int sem_wait(sem_t *sem);

 int sem_post(sem_t *sem);
```

libérer un sémaphore qui a été initialisé

•Équivalent de P(S) : tester la valeur d'un sémaphore. Si la valeur du sémaphore est positive, elle est décrémentée d'une unité et la fonction réussit

Équivalent de V(S)

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

int batmobile = 0;  // variable partagée
int moto = 0;  // variable partagée
sem_t batmobile_sem, moto_sem; // sémaphores pour synchroniser l'accès aux variables partagées

void *batman(void *arg) {
   sem_wait(&batmobile_sem); // attente du sémaphore batmobile_sem
   printf("Batman utilise la Batmobile.\n");
   batmobile++;
   sem_post(&batmobile_sem); // libération du sémaphore batmobile_sem
   sem_wait(&moto_sem); // attente du sémaphore moto_sem
   printf("Batman utilise la moto.\n");
   moto--;
   sem_post(&moto_sem); // libération du sémaphore moto_sem
   pthread_exit(NULL);
}
```

```c
void *robin(void *arg) {
   sem_wait(&batmobile_sem); // attente du sémaphore batmobile_sem
   printf("Robin utilise la batmobile.\n");
   batmobile--;
   sem_post(&batmobile_sem); // libération du sémaphore batmobile_sem
   sem_wait(&moto_sem); // attente du sémaphore moto_sem
   printf("Robin utilise la moto.\n");
   moto++;
   sem_post(&moto_sem); // libération du sémaphore moto_sem
   pthread_exit(NULL);
}
```

```
int main() {
   sem_init(&batmobile_sem, 0, 1); // initialisation du sémaphore batmobile_sem à 1
   sem_init(&moto_sem, 0, 0); // initialisation du sémaphore moto_sem à 0

   pthread_t thread_batman, thread_robin;

   pthread_create(&thread_batman, NULL, batman, NULL);
   pthread_create(&thread_robin, NULL, robin, NULL);

   pthread_join(thread_batman, NULL);
   pthread_join(thread_robin, NULL);

   sem_destroy(&batmobile_sem); // destruction du sémaphore batmobile_sem
   sem_destroy(&moto_sem); // destruction du sémaphore moto_sem

   return 0;
}
```

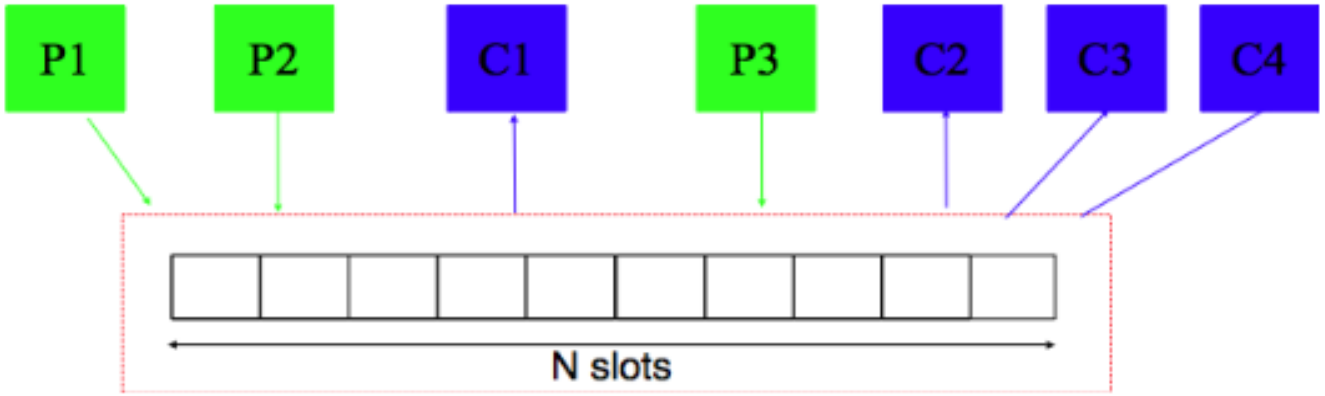
Problème des producteurs-consommateurs

Le problème des producteurs-consommateurs est un problème extrêmement fréquent et important dans les applications découpées en plusieurs threads. Il est courant de structurer une telle application, notamment si elle réalise de longs calculs, en deux types de threads :

•les *producteurs* : Ce sont des threads qui produisent des données et placent le résultat de leurs calculs dans une zone mémoire accessible aux consommateurs.

•les *consommateurs* : Ce sont des threads qui utilisent les valeurs calculées par les producteurs.

Ces deux types de threads communiquent en utilisant un buffer qui a une capacité limitée à *N* places comme illustré dans la figure ci-dessous.



Problème des producteurs-consommateurs

Conditions :
Le nombre de producteurs et de consommateurs ne doit pas nécessairement être connu à l'avance et ne doit pas être fixe. Un producteur peut arrêter de produire à n'importe quel moment.

Q: Quelle est la ressource partagée ?
R: Le buffer et il doit nécessairement être protégé par un mutex

Q: Quand est ce qu'un thread producteur se bloque ?
R: Lorsque le buffer est plein.

Q: Quand est ce qu'un thread consommateur se bloque ?
R: Lorsque le buffer est vide.

Problème des producteurs-consommateurs

Ce problème peut être résolu en utilisant deux sémaphores et un mutex.

L'accès au buffer, que ce soit par les consommateurs ou les producteurs est une section critique. Cet accès doit donc être protégé par l'utilisation d'un mutex.

Quant aux sémaphores :

- Le premier, nommé empty, sert à compter le nombre de places qui sont vides dans le buffer partagé. Ce sémaphore doit être initialisé à la taille du buffer puisque celui-ci est initialement vide.

- Le second sémaphore est nommé full. Sa valeur représente le nombre de places du buffer qui sont occupées. Il doit être initialisé à la valeur 0.

Problème des producteurs-consommateurs

```
// Initialisation
#define N 10 // places dans le buffer
pthread_mutex_t mutex;
sem_t empty;
sem_t full;

pthread_mutex_init(&mutex, NULL);
sem_init(&empty, 0 , N);  // buffer vide
sem_init(&full, 0 , 0);   // buffer vide
```

Problème des producteurs-consommateurs

```
// Producteur
void producer(void)
{
  int item;
  while(true)
  {
    item=produce(item);
    sem_wait(&empty); // attente d'une place libre
    pthread_mutex_lock(&mutex);
     // section critique
     insert_item();
    pthread_mutex_unlock(&mutex);
    sem_post(&full); // il y a une place remplie en plus
  }
}
```

Tout d'abord, le producteur est mis en attente sur le sémaphore empty. Il ne pourra passer que si il y a au moins une place du buffer qui est vide.

Lorsque la ligne sem_wait(&empty); réussit, le producteur s'approprie le mutex et modifie le buffer de façon à insérer l'élément produit (dans ce cas un entier).

Il libère ensuite le mutex pour sortir de sa section critique.

Problème des producteurs-consommateurs

```
// Consommateur
void consumer(void)
{
 int item;
 while(true)
 {
   sem_wait(&full); // attente d'une place remplie
   pthread_mutex_lock(&mutex);
    // section critique
    item=remove(item);
   pthread_mutex_unlock(&mutex);
   sem_post(&empty); // il y a une place libre en plus
 }
}
```

Le consommateur quant à lui essaie d'abord de prendre le sémaphore full.

Si celui-ci est positif, cela indique la présence d'au moins un élément dans le buffer partagé. Ensuite, il entre dans la section critique protégée par le mutex et récupère la donnée se trouvant dans le buffer.

Puis, il incrémente la valeur du sémaphore empty de façon à indiquer à un producteur qu'une nouvelle place est disponible dans le buffer.