Programmation Système

Stéphanie Moreaud

stephanie.moreaud@labri.fr
Département d'informatique
IUT Bordeaux 1

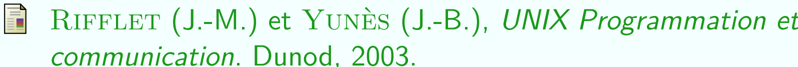
Plan du cours

1. Introduction
2. Complément de langage C
3. Vue générale d'un système d'exploitation
4. Généralités sur les processus UNIX
5. Gestion des processus sous UNIX
6. Communication par signaux
7. Communication des processus par tubes
8. Mémoire partagée
9. Processus légers (threads)

Plan de partie

1. Les processus légers (threads)
   - Utilisation de processus
   - Threads POSIX
   - Gestion des threads
   - Threads et partage de ressources
   - Section critique
2. Synchronisation
   - Sémaphores
   - Mutex
   - Conditions

Bibliographie



- Rifflet (J.-M.) et Yunès (J.-B.), *UNIX Programmation et communication*. Dunod, 2003.

- Billaud (M.), *Programmation système et réseau*.

  Polycopié de cours,
  http://www.labri.fr/perso/billaud/.

- Namyst (R.), *Cours de programmation système*.

- Billaud (M.) et Pierre (R.), *Supports de cours et TDs*.

- Zeitoun (M.), *Cours de programmation système*.

  http://www.labri.fr/perso/zeitoun/enseignement/current/INF355-0809.pdf.

Remerciements à Pierre Ramet, Michel Billaud et Kristian Kocher

Plan

1. Les processus légers (threads)
   - Utilisation de processus
   - Threads POSIX
   - Gestion des threads
   - Threads et partage de ressources
   - Section critique
2. Synchronisation
   - Sémaphores
   - Mutex
   - Conditions

Utilisation de processus

Processus UNIX classiques (*lourds*) créés par copies indépendantes d'un processus père.

- ➔ copie des ressources du processus père
- ➔ ressources séparées (espace mémoire, descripteurs,...)

Avantage

- Synchronisation "facile", pas de risque d'écrasement des données d'un autre processus.

Inconvénients

- Performances : duplication coûteuse
- Communication réduite
  - tubes
  - mémoire partagée (difficile)
  - ➔ nécessite des recopies, des protocoles compliqués...

Les processus légers (threads)

Définition d'un mécanisme permettant d'avoir plusieurs fils d'exécution (threads) dans un même espace de ressources *non dupliquées*

Les processus couvrent 2 aspects

- unité d'encapsulation : conteneur de certaines ressources
  - espace d'adressage, descripteurs de fichiers,...
- unités d'exécution, *thread* : exécution d'un programme, aspect pris en compte par l'ordonnanceur

→ contrôle d'exécution propre à chaque thread :

  - contexte d'exécution (état des registres), attribut d'ordonnancement, pile d'exécution, masque de signaux,...

Les processus légers (threads)

Plusieurs threads dans un processus :

- création des threads plus légère
- vocation à communiquer entre eux, partage de données, communication efficace !
- nécessité d'utiliser des mécanismes de synchronisation :
  - exclusion mutuelle (mutex), sémaphores, et conditions ...

Exemple d'utilisation :

- partage de données volumineuses
- parallélisme
- gestion des entrées utilisateurs (interfaces graphiques/programme principal), entrées multiples (multiplexage)
- simplification des protocoles de communication

Plan

1. Les processus légers (threads)
   - Utilisation de processus
   - Threads POSIX
   - Gestion des threads
   - Threads et partage de ressources
   - Section critique
2. Synchronisation
   - Sémaphores
   - Mutex
   - Conditions

Threads POSIX

Interface POSIX pour la gestion et la manipulation des threads

- assez grosse API (plus de 50 fonctions)
- bibliothèque pthread, en-tête :

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

  compilation avec `-lpthread`

- en général, valeur de retour est 0 si OK

Plus d'informations : `man pthreads`

Plan

1. Les processus légers (threads)
   - Utilisation de processus
   - Threads POSIX
   - Gestion des threads
   - Threads et partage de ressources
   - Section critique

2. Synchronisation
   - Sémaphores
   - Mutex
   - Conditions

Gestion des threads : Création

Création d'un nouveaux processus léger : pthread_create

pthread_create()

```
1 int pthread_create(pthread_t * thread,
2                    pthread_attr_t * attr,
3                    void * (*start_routine)(void *),
4                    void * arg);
```

- thread : identificateur du thread rempli par l'appel, (unique pour le processus)
- attr : permet de changer les attributs (priorité, ordonnancement, etc.), modifie le fonctionnement
- start_routine : fonction de démarrage du thread.
- arg : pointeur sur les arguments de cette fonction.

Gestion des threads : Terminaison

La terminaison du processus léger se fait avec code de retour :

- par appel à la fonction `pthread_exit()`
- lorsque la fonction associée au thread se termine par `return retval` (appel implicite à pthread_exit)

La fonction `pthread_join()` permet au processus d'attendre la fin d'un processus léger, et de récupérer son code de retour.

- équivalent du `wait()` pour les processus

Les ressources du processus ne sont libérées qu'à la terminaison du dernier thread.

pthread_join() et pthread_exit()

```
1 void pthread_exit(void *retval);
2 int pthread_join(pthread_t th, void **thread_return);
```

Gestion des threads

hello.c (1/2)

```
1  void *thread_hi(void* nb);
2  void *thread_boujour(void* nb);
3
4  int main(int argc, char**argv)
5  {
6    pthread_t tid[2];
7    int i=1, j=2;
8    void *ret_val, ret_val2;
9
10   /*creation deux deux threads */
11   printf("processus %i cree 2 threads et les attends\n", getpid());
12   int ret = pthread_create(&tid[0], NULL, thread_hi, (void*)i);
13   int ret2 = pthread_create(&tid[1], NULL, thread_boujour, (void*)j);
14   if(ret || ret2) {
15     fprintf(stderr, "Probleme lors de la creation d'un thread \n");
16     return 1;
17   }
18
19   ret = pthread_join(tid[0], &ret_val);
20   ret2 = pthread_join(tid[1], &ret_val2);
21   if(ret || ret2) {
22     fprintf(stderr, "Erreur lors de l'attente d'un thread\n");
23     return 1;
24   }
25   printf("Le thread 1 s'est termine avec le code %d\n", (int)ret_val);
26   printf("Le thread 2 s'est termine avec le code %d\n", (int)ret_val2);
27   return 0;
28 }
```

Gestion des threads

hello.c (2/2)

```
1  void *thread_hi(void* nb)
2  {
3    int id=(int)nb;
4    printf("Hi! je suis le thread %d du processus %i\n", id, getpid());
5    return (void*)id;
6  }
7
8  void *thread_boujour(void* nb)
9  {
10   int id=(int)nb;
11   printf("Bonjour, je suis le thread %d du processus %i\n", id, getpid());
12   return (void*)id;
13 }
```

Exécution

```
1 processus 3125 cree deux threads et les attend
2 Hi! je suis le thread 1 du processus 3125
3 Bonjour, je suis le thread 2 du processus 3125
4 Le thread 1 s'est termine avec le code 1
5 Le thread 2 s'est termine avec le code 2
```

Plan

1 Les processus légers (threads)
- Utilisation de processus
- Threads POSIX
- Gestion des threads
- Threads et partage de ressources
- Section critique

2 Synchronisation
- Sémaphores
- Mutex
- Conditions

Threads et partage de ressources

Les threads d'un même processus partagent de la mémoire

- ➔ facilite l'échange de données
- ➔ nécessite dans certains cas d'arbitrer l'accès aux données pour éviter les corruptions

Que se passe t-il si deux threads écrivent simultanément dans une variable ?

Threads et partage de ressources

Le code suivant est exécuté en concurrence par 2 threads.
La variable globale z est partagée entre les threads.

increment.c

```
1  volatile int z;
2
3  void increment_compteur(){
4    int i=0;
5    for(i=0; i<10000; i++){
6      z++;
7    }
8    return 1;
9  }
```

La valeur de z est affichée après un `pthread_join()` sur chaque thread.
Quelle valeur sera affichée ?

Threads et partage de ressources

increment.c

```
1  volatile int z;
2
3  void increment_compteur(){
4    int i=0;
5    for(i=0; i<10000; i++){
6      z++;
7    }
8    return 1;
9  }
```

Quelques résultats

```
1  La valeur de z est 12587
2  La valeur de z est 20000
3  La valeur de z est 12203
4  La valeur de z est 13831
5  La valeur de z est 12160
```

Plan

1. Les processus légers (threads)
   - Utilisation de processus
   - Threads POSIX
   - Gestion des threads
   - Threads et partage de ressources
   - Section critique
2. Synchronisation
   - Sémaphores
   - Mutex
   - Conditions

Les processus légers (threads)

Section critique : portion de code dans laquelle il doit être garanti qu'il n'y aura jamais plus d'un thread simultanément.

Il est nécessaire d'utiliser des sections critiques lorsqu'il y a accès à des ressources partagées par plusieurs threads.

Les mécanismes de synchronisation sont utilisés pour résoudre les problèmes de sections critiques et plus généralement pour bloquer et débloquer des threads suivant certaines conditions.

Plan

1. Les processus légers (threads)
   - Utilisation de processus
   - Threads POSIX
   - Gestion des threads
   - Threads et partage de ressources
   - Section critique
2. Synchronisation
   - Sémaphores
   - Mutex
   - Conditions

Sémaphores : principe

Sémaphore : mécanisme permettant de limiter l'accès concurrent à une section critique.

- modèle inventé par Dijkstra, s'appuie sur :
  - un compteur
  - une fonction d'*acquisition*, P()
  - une fonction de *libération*, V()
- P() décrémente le compteur
  - si le compteur est nul le processus/thread est bloqué.
- V() incrémente le compteur
  - débloque l'un des processus/thread bloqué

Sémaphores

Un sémaphore a une infinité d'états, positifs ou nuls

- ne peut pas être négatif
- acquisition d'un sémaphore de valeur nulle
  ➔ mise en attente du processus ou du thread jusqu'à ce qu'une libération se produise.
- "système de jetons"

Remarque : le compteur utilisé est modifiable via P() et V() par les threads/processus concurrents.

➔ implémentation spécifique de ces fonctions, assure que les modifications/lectures du compteur ne peuvent pas être effectuées simultanément par deux threads/processus.

Sémaphores : section critique

init_semaphore (1);

Déroulement **Thread 1** **Thread 2**

P();

section critique

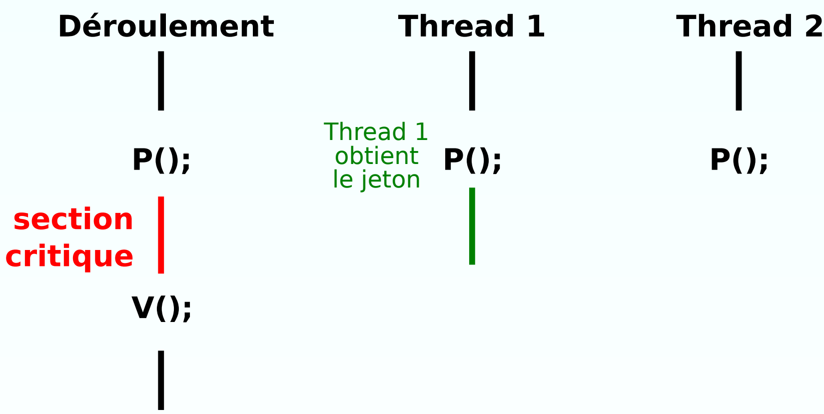
V();

Sémaphores : section critique

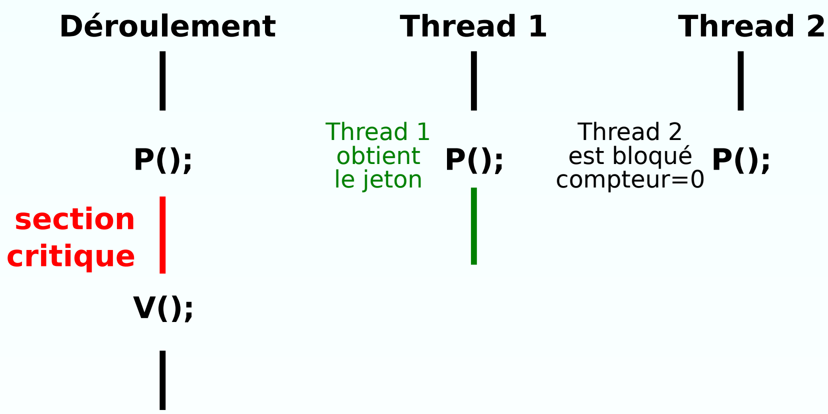
init_semaphore (1);

Déroulement	**Thread 1**	**Thread 2**
P();	**P();**	**P();**
section critique		
V();		

Sémaphores : section critique

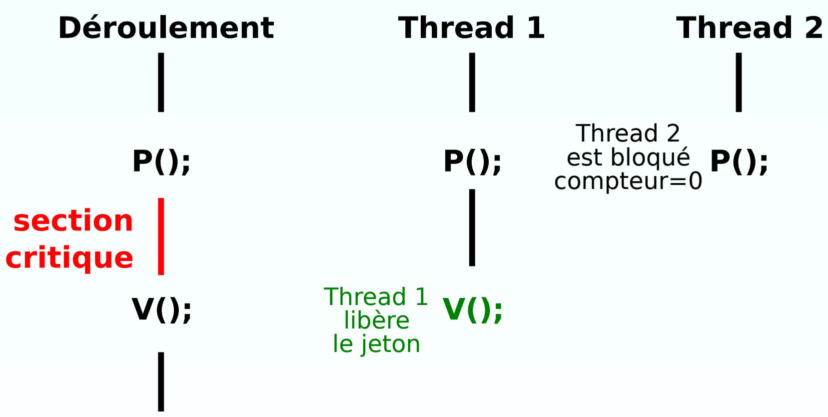


Sémaphores : section critique

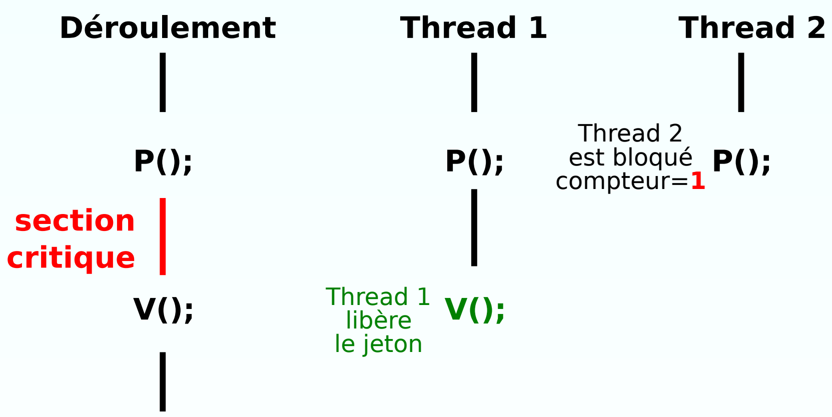


Sémaphores : section critique

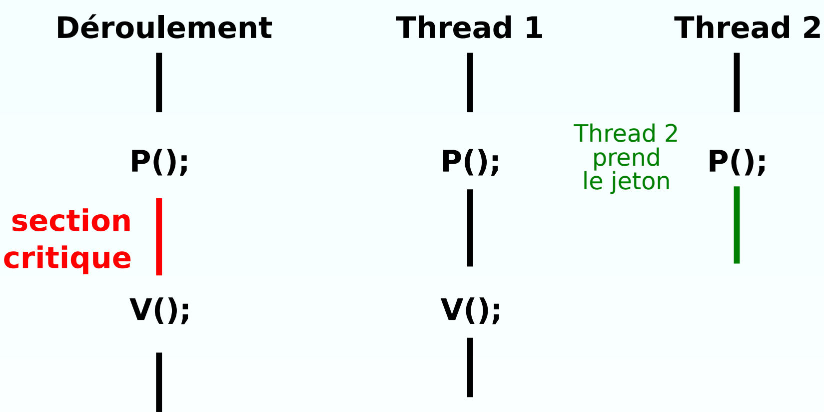
init_semaphore (1);



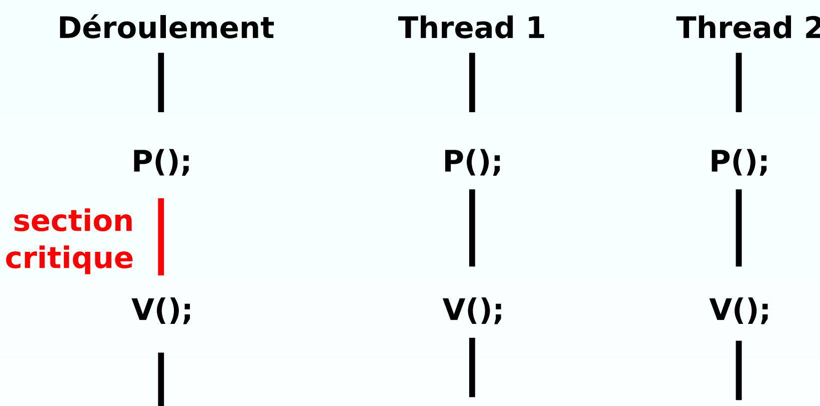
Sémaphores : section critique



Sémaphores : section critique

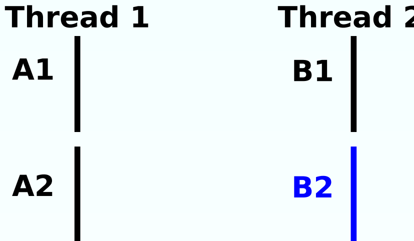


Sémaphores : section critique



Sémaphores : synchronisation

On considère l'exécution de deux threads concurrents comme suit :

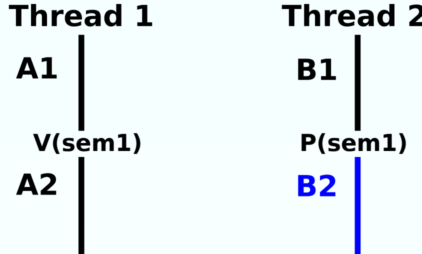


Dans cet exemple, l'interdépendance des threads fait que :

- B2 doit impérativement être exécutée après la section A1 (initialisation dans A1 de variables utilisées dans B2 par exemple)

Sémaphores : synchronisation

On considère l'exécution de deux threads concurrents comme suit :



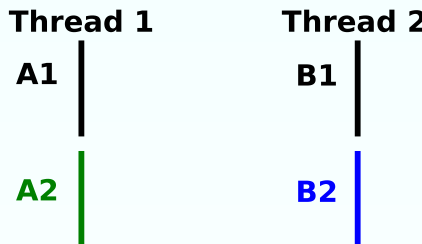
Dans cet exemple, l'interdépendance des threads fait que :

- B2 doit impérativement être exécutée après la section A1 (initialisation dans A1 de variables utilisées dans B2 par exemple)

Valeur initiale du sémaphore : 0

Sémaphores : interdépendance

On considère l'exécution de deux threads concurrents comme suit :

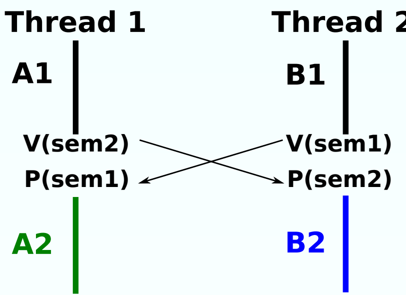


Dans cet exemple, l'interdépendance des threads fait que :

- B2 doit impérativement être exécutée après la section A1
- A2 doit impérativement être exécutée après la section B1

Sémaphores : interdépendance

On considère l'exécution de deux threads concurrents comme suit :



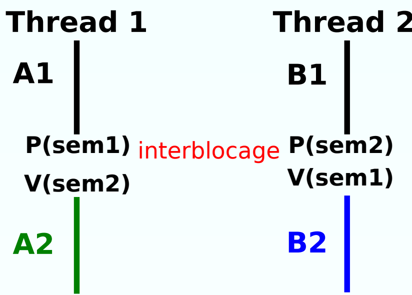
Dans cet exemple, l'interdépendance des threads fait que :

- B2 doit impérativement être exécutée après la section A1
- A2 doit impérativement être exécutée après la section B1

Valeur initiale des sémaphores : 0

Sémaphores : attention à l'interbloquage !

On considère l'exécution de deux threads concurrents comme suit :



Dans cet exemple, l'interdépendance des threads fait que :

- B2 doit impérativement être exécutée après la section A1
- A2 doit impérativement être exécutée après la section B1

Valeur initiale des sémaphores : 0

Sémaphores : POSIX

gestion des sémaphores

```
1 #include <semaphore.h>
2
3 int sem_init(sem_t *sem, int pshared, unsigned int value);
4 int sem_destroy(sem_t *sem);
5 int sem_wait(sem_t *sem);
6 int sem_trywait(sem_t *sem);
7 int sem_post(sem_t *sem);
8 int sem_getvalue(sem_t *sem, int *sval)
```

Remarque : les sémaphores, qui font partie de la norme POSIX, ne sont pas implémentés dans toutes les bibliothèques de threads.

Sémaphores : POSIX

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

➔ crée un sémaphore, place sont identificateur à l'endroit pointé par sem et l'initialise à value. Si pshared est nul, le sémaphore est local au processus lourd.

`int sem_wait(sem_t *sem); int sem_post(sem_t *sem);`

➔ sem_wait et sem_post équivalents respectifs des primitives P et V de Dijkstra.

`int sem_trywait(sem_t *sem);`

➔ échoue (au lieu de bloquer) si la valeur du sémaphore est nulle.

`int sem_getvalue(sem_t *sem, int *sval)`

➔ consulte la valeur courante du sémaphore.

Sémaphores : Exemple

shopping.c (1/2)

```
#define NB_CLIENTS 10
#define NB_CABINES 3
void *essayer(void *arg);
sem_t sem;

int main(int argc, char**argv) {
  // initialise le semaphore au nombre de cabines diponibles
  sem_init(&sem, 0, NB_CABINES);
  pthread_t tid[NB_CLIENTS];
  int i;
  printf(" on cree %i clients (threads) \n", NB_CLIENTS);
  for(i=0;i<NB_CLIENTS;i++) {
    int ret = pthread_create(&tid[i], NULL, essayer, (void*)i);
    if(ret) {
      fprintf(stderr, "Impossible de creer le thread %d\n", i);
      return 1;
    }
  }
  void *ret_val;
  for(i=0;i<NB_CLIENTS;i++) {
    int ret = pthread_join(tid[i], &ret_val);
    if(ret) {
      fprintf(stderr, "Erreur lors de l'attente du thread %d\n", i);
      return 1;
    }
  }
  printf(" Tous les clients (threads) ont effectue leur essayage\n");
  return 0;
}
```

Sémaphores : Exemple

shopping.c (2/2)

```
void *essayer(void *arg)
{
  int id=(int) arg;
  int val_sem;
  sem_getvalue(&sem, &val_sem);
  printf("client %i: j'ai constate qu'il y avait %i cabines
     libres, j'essaye d'en avoir une\n", id, val_sem);
  sem_wait(&sem);
  printf("client %i: j'ai eu une cabine, j'essaye\n", id);
  sleep(1);
  printf("client %i: je libere la place\n", id);
  sem_post(&sem);
  return (void*)id;
}
```

Excercice : modifiez le déroulement du programme en utilisant `sem_trywait` plutôt que le `sem_wait` bloquant.

Plan

1. Les processus légers (threads)
   - Utilisation de processus
   - Threads POSIX
   - Gestion des threads
   - Threads et partage de ressources
   - Section critique

2. Synchronisation
   - Sémaphores
   - Mutex
   - Conditions

Mutex

Un Mutex (*Mutual exclusion*) est une primitive de synchronisation permettant d'éviter que des ressources partagées d'un système ne soient utilisées en même temps.

➔ Implémentation efficace d'un sémaphore binaire.

Plusieurs algorithmes

- algorithme de Dekker
- algorithme de Peterson

Mutex : Pthread

Mutex pthread

```
#include <pthread.h>

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

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

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

➔ crée un verrou d'exclusion mutuelle (mutex).

- différents types (attributs pointés par mutexattr, par défaut mutexattr = NULL).
- identificateur du verrou placé dans la variable pointée par mutex.

Mutex

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

➔ détruit le verrou.

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

➔ tente de bloquer le verrou (met le thread en attente s'il est déjà bloqué).

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

➔ débloque le verrou.

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

➔ tente de bloquer le verrou, échoue s'il est déjà bloqué.

Mutex : Exemple

course2.c (1/2)

```c
#define NB_THREADS 5
#define MAX 20000
volatile int cpt;
pthread_mutex_t m;
void *courrir(void *arg);

int main(int argc, char**argv) {
  pthread_t tid[NB_THREADS];
  pthread_mutex_init(&m, NULL);
  cpt=0;
  int i;
  for(i=0;i<NB_THREADS;i++) {
    int ret = pthread_create(&tid[i], NULL, courrir, (void*)i);
    if(ret) {
      fprintf(stderr, "Impossible de creer le thread %d\n", i);
      return 1;
    }
  }
  void *ret_val;
  for(i=0;i<NB_THREADS;i++) {
    int ret = pthread_join(tid[i], &ret_val);
    if(ret) {
      fprintf(stderr, "Erreur lors de l'attente du thread %d\n", i);
      return 1;
    }
  }
  pthread_mutex_destroy(&m);
  printf("cpt vaut %d\n", cpt);
  return 0;
}
```

Mutex : Exemple

course2.c (2/2)

```
1  void *courrir(void *arg)
2  {
3    int id=(int)arg;
4    int i;
5    for(i=0;i<MAX;i++) {
6      {
7        // section critique
8        pthread_mutex_lock(&m);
9        cpt++;
10       pthread_mutex_unlock(&m);
11     }
12   }
13   return (void*)id;
14 }
```

Le programme course2.c est la version "thread-safe" (utilisant un mutex) du programme course.c, qui créer plusieurs threads incrémentant un même compteur global.
Testez les résultats de ces deux programmes.

Plan

1 Les processus légers (threads)
- Utilisation de processus
- Threads POSIX
- Gestion des threads
- Threads et partage de ressources
- Section critique

2 Synchronisation
- Sémaphores
- Mutex
- Conditions

Conditions

Les conditions permettent de mettre en attente des processus légers derrière un mutex.
➔ débloquage de tous les threads bloqués sur une même condition d'un seul coup.

Conditions pthread

```
1  #include <pthread.h>
2
3  int pthread_cond_init(pthread_cond_t *cond,
       pthread_condattr_t *cond_attr);
4  int pthread_cond_destroy(pthread_cond_t *cond);
5  int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t
        *mutex);
6  int pthread_cond_signal(pthread_cond_t *cond);
7  int pthread_cond_broadcast(pthread_cond_t *cond);
```

Conditions

```
int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *cond_attr);
```

➔ création d'une condition.

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

➔ destruction de la condition.

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
```

➔ mise en attente du thread sur la condition

```
int pthread_cond_signal(pthread_cond_t *cond);
```

➔ débloque un thread en attente sur la condition

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

➔ débloque tous les threads en attente sur la condition

Conditions : Exemple

Analysez le fichier source `course3.c`.

Exercice : modifiez celui-ci pour faire une vraie “attente sur condition”.
On s’appuira par exemple sur une variable partagée `ready` comptabilisant le nombre de threads créés, puis prêts, et déclancheur de la condition départ.