Université Paris 8
2, rue de la Liberté
93526 Saint-Denis cedex

Cours
Programmation concurrente

Les Threads

Département Informatique et Technologies du Numérique
Master 1 Informatique
Parcours : Informatique & Big Data

Y. Touati
capaok@gmail.com

Généralités et quelques définitions

Threads ?

Histoire

- L'utilisation des mécanismes de création de processus sous Unix comme le **fork**, **wait**, et le **pipe** sont très onéreux en termes de coûts et ne peuvent être utilisés qu'avec un nombre restreint de processus.

- Introduction dès les années 90, diverses bibliothèques de processus légers :

  Threads Posix = Portable Operating System Interface

- Utilisation de La bibliothèque **pthread** avec plusieurs fonctions en C pour la programmation.

Threads ?

Définition

- Processus légers ou unités d'exécution qui opèrent dans le contexte d'un processus.
- Un processus peut contenir plusieurs threads qui exécutent tous le même programme et partagent la même mémoire
  (Segments de code, les data et le tas pour les allocations dynamiques)
- Partage entre les différentes threads d'un même processus.
- Chaque thread dispose de sa propre pile (Poursuite des chemins d'exécution différents).

Processus vs. Threads

Processus

- Lourds, nécessitant beaucoup de ressources.
- La commutation inter-processus nécessite une interaction avec l'OS.
- Environnement multiprocessus: chaque processus exécute le même code en utilisant ses propres ressources.
- Processus bloqué : Aucun processus ne peut s'exécuter tant que ce blocage persiste.
- Processus multiples sans threads : Nécessitée de beaucoup de ressources.

  (*Chaque processus fonctionne indépendamment des autres*)

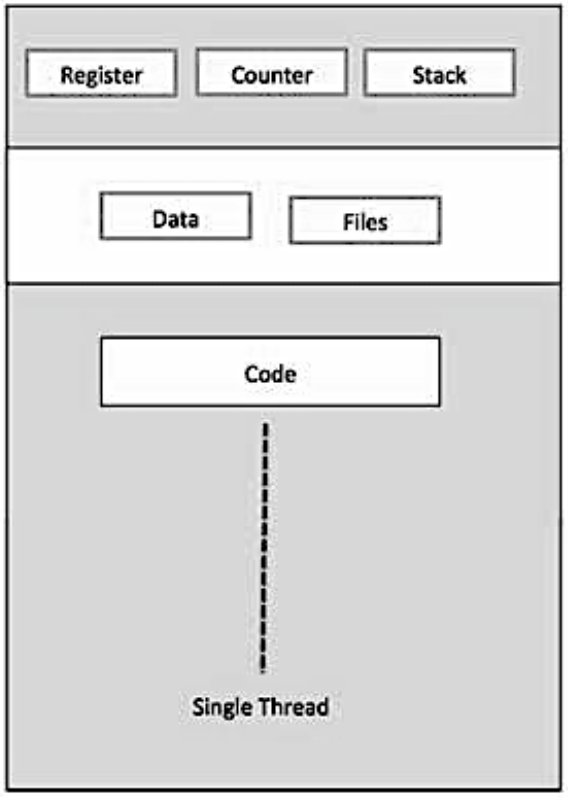
VS.

Threads

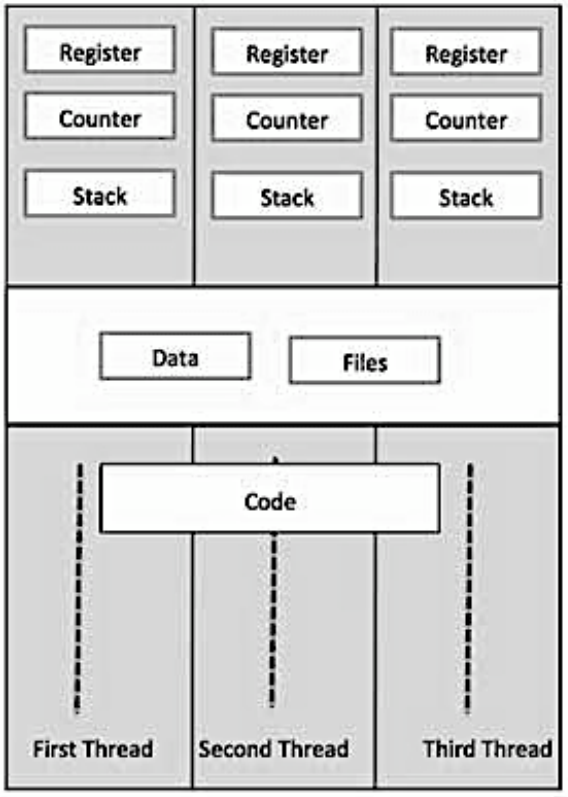
- Légers, avec moins de ressources qu'un processus.
- Aucune interaction avec l'OS.
- Les threads peuvent partager le même ensemble de fichiers ouverts (processus enfants).
- Si un thread est bloqué ou en attente, un second thread peut être exécuté.

  (Thread a*ppartenant à la même tâche*)
- Un processus multithreads utilise moins de ressources.
- Un thread peut lire, écrire ou modifier les données d'un autre thread.

Fil d'exécution des threads dans un processus



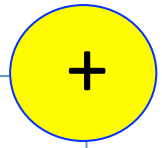
Processus unique avec thread unique



Processus unique avec 3 threads

Implémentation des Threads

Les plus



- Minimisation du temps de changement de contexte.
- Simultanéité au sein d'un processus.
- Communication efficace.
- Economie : Création de threads et changement de contexte.
- Efficacité d'utilisation d'architectures multiprocesseurs à plus grande échelle.

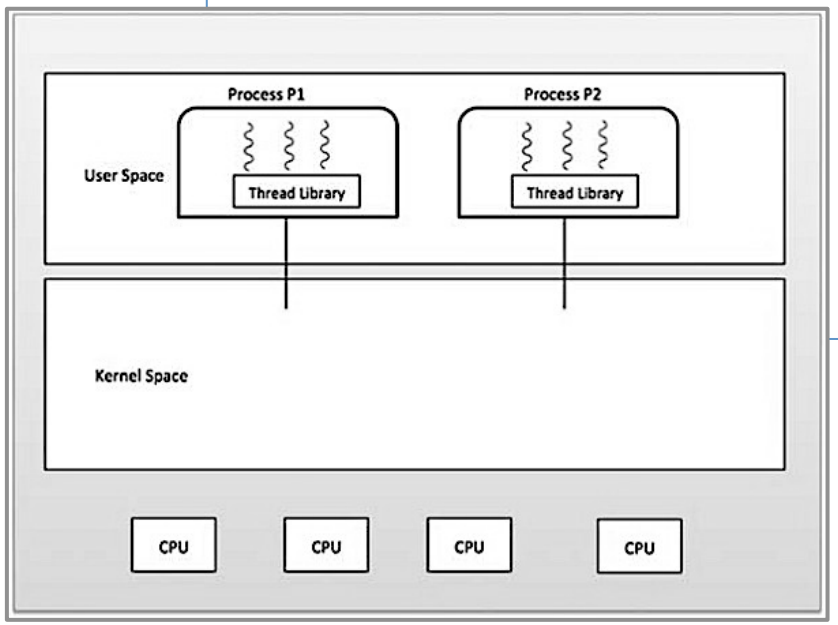
Types de Threads

Gestion concurrente et parallèles

- Threads niveau utilisateur.
  (Threads gérés par l'utilisateur, Pb de blocage?)
- Threads niveau noyau.
  (Threads gérés par l'OS agissant sur le noyau)
- Threads Multiprocesseurs.
  (Plusieurs processeurs tournent en parallèle, un processus-un thread)

Les Threads niveau utilisateur

- Le **noyau** n'a aucune connaissance sur l'existence de threads.
- Une **bibliothèque de threads** avec un code à plusieurs fonctionnalités :
  - Création et destruction de threads.
  - Transmission des messages et des données entre threads.
  - Planification de l'exécution des threads.
  - Sauvegarde et restauration de contextes des threads.



+

- La commutation ne nécessite aucun privilège.
- Un thread niveau utilisateur peut être exécuté sur n'importe quel OS.
- Rapidité : création et gestion.

Les Threads niveau noyau

- La gestion des threads par l'OS.
  - Conservation des informations de contexte pour un processus donné dans son ensemble et pour les threads individuellement, au sein du même processus.
  - Planification, création, et gestion des threads.
- Toute application peut être programmée pour être multithread.

+

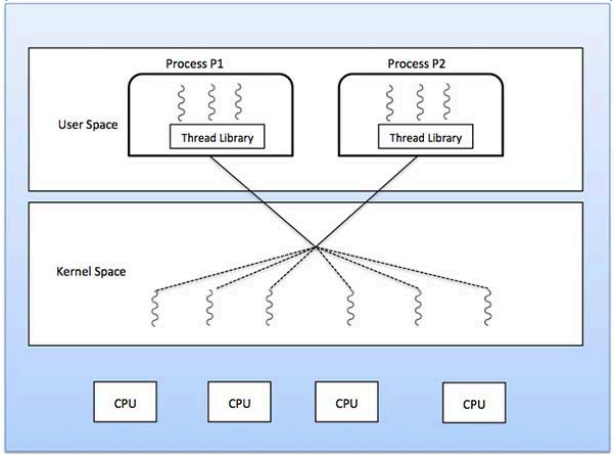
- Le noyau peut planifier simultanément plusieurs threads d'un processus donné sur plusieurs processus.
- Si un thread dans un processus est bloqué, le noyau peut planifier un autre thread du même processus.
- Les routines du noyau elles-mêmes peuvent être multithreads.

-

- Lenteur de création et de gestion des Threads vs. Côté utilisateurs.
- Nécessité d'une **commutation de mode** pour le transfert du contrôle d'un thread à un autre (dans un même processus).

Modèles Multithreads

MANY To MANY model

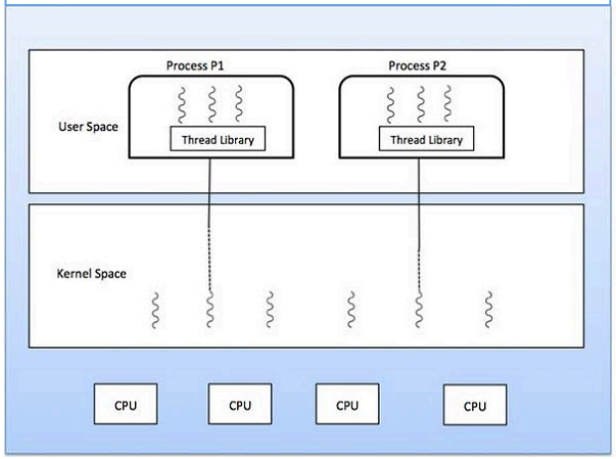


Les **threads utilisateur** réalisent un multiplexage avec des **threads noyau**.

Au niveau noyau: Exécution parallèle des threads dans un système multiprocesseurs.

Un thread effectuant un appel système bloquant, le noyau peut planifier l'exécution d'un autre thread.

MANY To ONE model

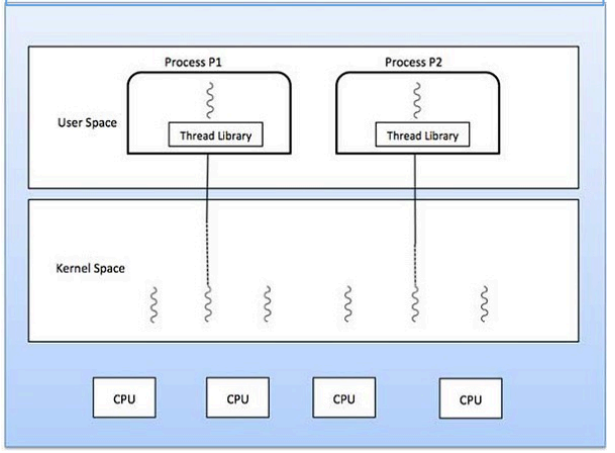


Modèle mappant plusieurs threads de niveau utilisateur en un thread au niveau noyau.

Un seul thread à la fois peut accéder au noyau.

Un thread effectuant un appel système bloquant, paralyse l'ensemble du processus.

ONE To ONE model



Un thread effectuant un appel système bloquant, un seul thread peut s'exécuter.

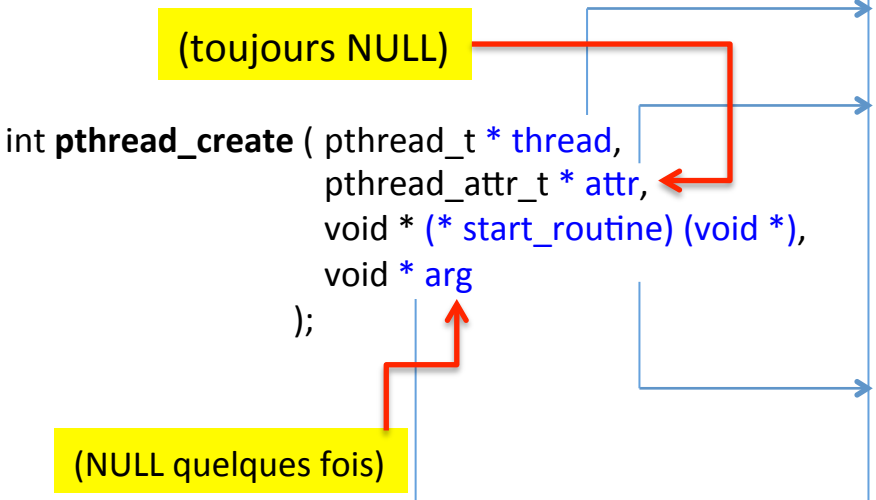
Système multiprocesseurs : Plusieurs threads s'exécutent en parallèle.

Meilleure précision en cas de simultanéité

Mise en pratique

Création de thread

Bibliothèque : **include** <*pthread.h*>
Avant la création : il faut le déclarer en tant que *thread* : **pthread_t** *thread*



1. pthread_t : Identifiant du thread qui sera créé avec un type opaque (sous Linux : unsigned long).
2. pthread_attr_t : définit des attributs spécifiques pour chaque thread. Il faut simplement savoir que l'on peut changer le comportement de la gestion des threads comme par exemple, les régler pour qu'ils tournent sur un système temps réel. En générale on se contente des attributs par défaut en mettant cet argument à NULL.
3. La raison de vivre du thread (fonction à exécuter). Cet argument permet de transmettre un pointeur sur la fonction qu'il devra exécuter.
4. Argument que l'on peut passer à la fonction que le thread doit exécuter.

- Si la création réussit, la fonction renvoie 0 (zéro) et l'identifiant du thread nouvellement créé est stocké à l'adresse fournie en premier argument.
- Si ERREUR, la valeur EAGAIN est retournée par la fonction (ressources système insuffisantes pour créer un nouveau thread ou bien si le nombre maximum de threads définit par la constante PTHREAD_THREADS_MAX est atteint ≅1024)

Exécution de thread

```
int pthread_join (    pthread_t thread,
                      void ** thread_return
                  );
```

1. *thread* : Thread à attendre.
2. Valeur de retour de la fonction du thread *thread*.

Objectif :
Permettre le cas échéant un échange avec d'autres THREADS
Récupérer la data de retour de la primitive : int **pthread_create**

- Cette fonction met en pause l'exécution du thread appelant jusqu'au retour de la fonction.
- **SUCCESS :**

  Elle retourne 0 et la valeur de retour du thread est passée à l'adresse indiquée (second argument) si différente de NULL.
- **ERROR** : la fonction retourne :
  - **ESRCH** : Aucun thread ne correspond à celui passé en argument.
  - **EINVAL** : Le thread a été détaché ou un autre thread attend déjà la fin du thread.
  - **EDEADLK** : Le thread passé en argument correspond au thread appelant.

Annulation et fin de thread

int **pthread_cancel** (pthread_t thread
);

1. Argument correspondant au thread à annuler.
   - Si la création réussit, la fonction renvoie 0 (zéro)
   - Si aucun thread ne correspond à celui passé en argument, , la valeur ESRCH est retournée

void **pthread_exit** (void * retval
);

1. Son seul argument est le retour de la fonction du thread appelant. Cet argument peut aussi être récupéré par la fonction **pthread_join**.

Récupère une valeur de la fonction « (* start_routine) (void *)» puis elle l'affiche et l'envoi à la primitive «**pthread_join**» en tant que second argument

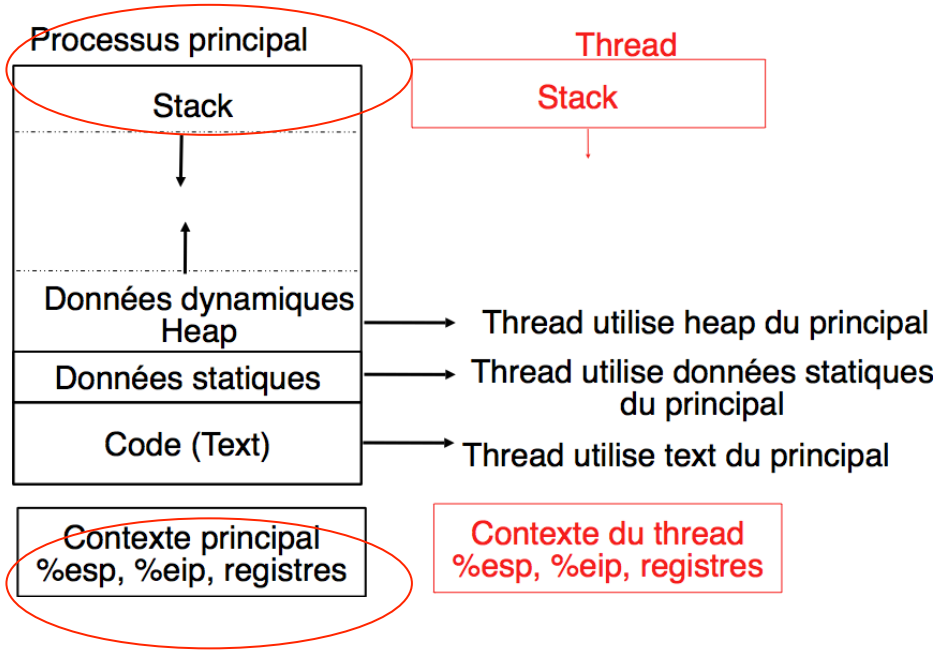
« (* start_routine) (void *)» -> «** thread_return»

Remarque :
Il existe d'autres primitives comme **pthread_execute** qui permet de tester l'exécution d'un THREAD

Communication inter-threads & processus

- Problème très complexe.
  (Lecture/écriture en simultanée en mémoire peut être une source de problèmes)
  L'exécution d'une ligne de code par un thread nécessite l'exécution de plusieurs instructions en assembleur
  …

- Problème d'accès concurrent à une zone de mémoire par plusieurs threads en mono et multiprocesseurs.
  - Exclusion mutuelle et Concurrence
  - Section critique
  - Verrous

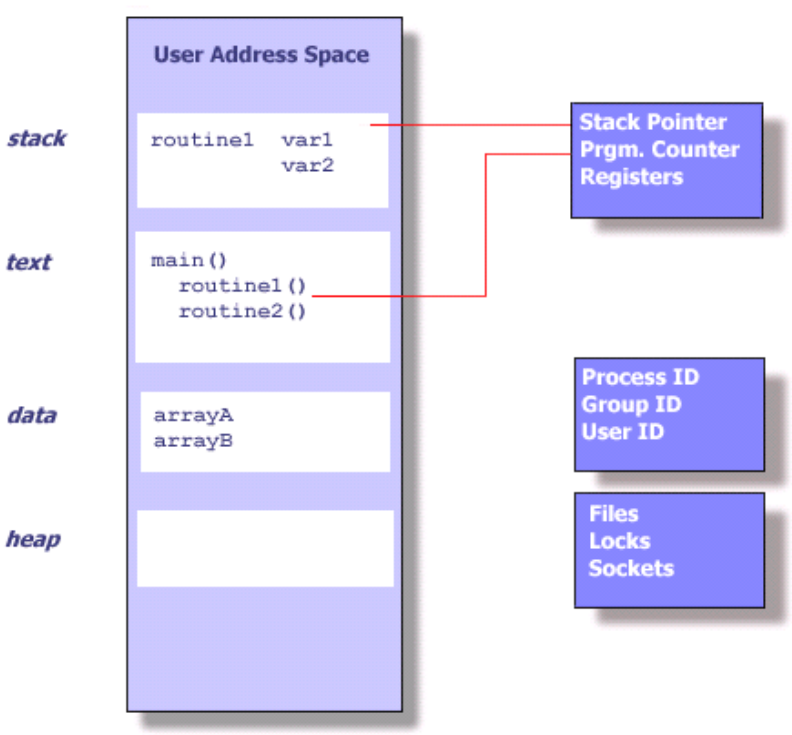


Données statiques, constantes, variables globales, chaînes de caractères

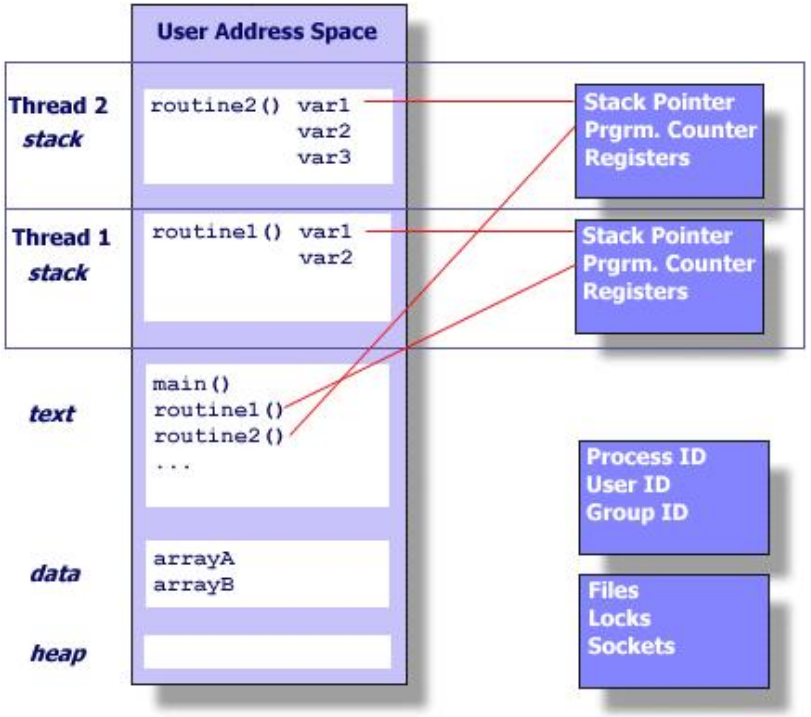
Instructions qui composent le programme

Communication inter-threads & processus

- Comme pour les processus, sur une machine à un seul processeur, les threads se partagent le processeur pour donner l'illusion de s'exécuter en même temps.



Processus Unix/Linux



Threads au sein d'un processus Unix/Linux

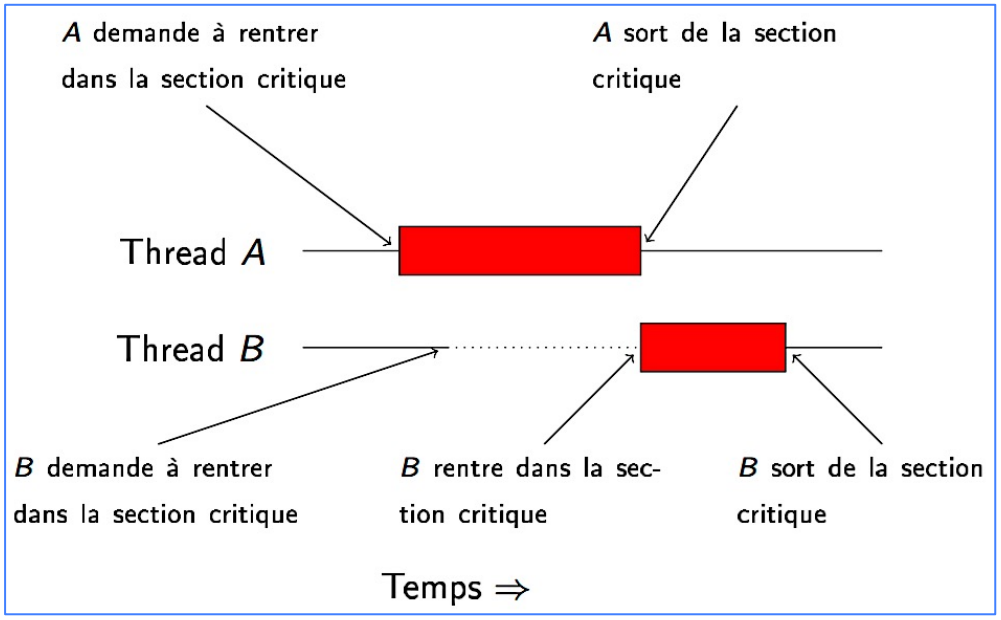
Accès concurrentiels

Remarque : Les threads ne tiennent en aucun cas compte qu'une fonction secondaire puisse avoir accès à une même variable (lecture ou écriture), impliquant de fait une **gestion d'accès concurrentiels**, très utile dans le contexte d'applications critiques.

Section critique

Section de code dans laquelle un seul thread au plus tourne.

Un Thread ***B*** souhaitant accéder à la section critique, et qu'un autre Thread ***A*** y est déjà, le Thread ***B*** doit attendre jusqu'à ce que le Thread ***A*** sorte de la section critique.



- Les différents Threads ne travaillant pas dans la section critique ne sont pas bloqués
- Plusieurs sections critiques différentes peuvent coexister sans interférence.

Accès concurrentiels

Mécanisme de gestion des sections critiques

- **Exclusion mutuelle**
  Deux Threads ne peuvent être fonctionnels au même temps dans une section critique
- **Progression**
  Continuité de fonctionnement du processus quelque soit les cas d'exécution
- **Attente bornée**
  Un Thread souhaitant rentrer dans une section critique ne peut attendre indéfiniment.

- **Attente active**
  Le Thread continue de tourner jusqu'à ce qu'il soit autorisé à accéder à la section critique.
- **Attente passive**
  Le Thread est mis en pause jusqu'à ce qu'il ait la possibilité de rentrer dans la section critique (Utilisation de l'ordonnanceur).
  Avantage : Le temps CPU est utilisé par d'autres Threads.

Primitive

Mécanisme à verrou : Mutex

Accès concurrentiels

Mutex

Système de verrou garantissant la viabilité des données manipulées par les threads dans la portion de code protégée ou zone critique. Une fois le thread terminé, le verrou est libéré. Ce dernier peut être repris à son tour, par un autre thread.

Pseudo-code :

```
Mutex m;

// section non critique
lock(m);
// section critique
unlock(m);
// section non critique
....
```

Blocage permettant à un THREAD de s'exécuter

Déblocage et sortie de la section critique

Création d'un mutex

- Déclarer une variable du type **pthread_mutex_t**
- Initialiser la variable avec la constante PTHREAD_MUTEX_INITIALIZER

Initialisation du verrou

```
static pthread_mutex_t mutex_thread = PTHREAD_MUTEX_INITIALIZER;
```

ou

Initialisation et déclaration du verrou

```
pthread_mutex_t mutex_thread
pthread_mutex_init (&mutex_thread, NULL);
```

Etats d'un mutex

Etat verrouillé

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

- Cette fonction permet de déterminer le début d'une zone critique. Son seul argument est l'adresse d'un mutex de type pthread_mutex_t.
- La fonction renvoie 0 en cas de succès, en cas d'échec :
  - **EINVAL** : mutex non initialisé.
  - **EDEADLK** : mutex déjà verrouillé par un autre thread.

Le THREAD s'est arrêter
Sa fonctionnalité est réalisée

Etat déverrouillé

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

- Cette fonction permet de relâcher le verrou passé en argument qui est l'adresse d'un mutex de type pthread_mutex_t.
- La fonction renvoie 0 en cas de succès, en cas d'échec :
  - **EINVAL** : mutex non initialisé.
  - **EPERM** : le thread n'a pas la main sur le mutex.

Exercices d’application