Programmation parallèle

IUT – Lyon 1

Module programmation répartie

Principe de la programmation parallèle

- Consiste à faire exécuter par des processus différents des parties de code d'un même programme
  - Par un même CPU/cœur
  - Par des cœurs/CPUs différents
  - Par des machines différentes (cloud computing)

Avantage de la programmation parallèle

- Réduire le temps d'exécution des problèmes
  - Résolution numérique par exemple
- Réduire les temps de réponse des services
  - Service réseau par exemple (web, etc.)
- Ne pas rester sur un appel bloquant
  - Service réseau par exemple (un processus est à l'écoute des connexions entrantes, d'autres processus traitent les clients)

Performance de la programmation parallèle

- Tout les programmes ou partie d'un programme ne sont pas parallélisables:
  - Evaluation numérique:
    - Certains calculs nécessitent le résultat de calculs précédents
  - Accès à une entrée/sortie
    - Les accès à un disque dur / clé USB se font en série
- Exemples de tâches parallélisables:
  - Gestion client pour un service réseau
  - Calcul numérique
    - interface graphique par exemple (GPU)
  - etc.
- Performance:
  - Le temps d'exécution d'un programme est minoré par le temps d'exécution de sa partie non parallélisable.

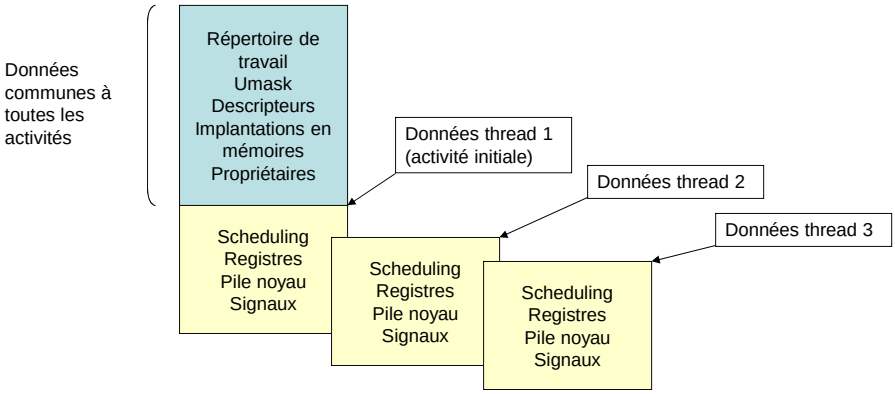
Les threads

Plan

- Propriétés des *threads*
- *Appel de base:*
  - *Création/Terminaison/synchronisation(join)*
- *Annulation*
- *Synchronisation des processus*
  - *Mutex / sémaphore*
  - *Variable de conditions*

Thread

- Un processus peut regrouper plusieurs threads
- L'ensemble des activités se partage alors une partie des données du processus initial (celui qui a lancé l'ensemble des threads)
- La création de ces threads est beaucoup plus légère que la création de processus avec un appel à *fork()*
- De plus, cela permet un partage/communication plus simple entre les threads.



Thread: propriétés

- Chaque thread est appelée pour effectuer une tâche particulière (une fonction)
  - La pile de cette fonction est propre à chaque thread
  - Pointeur sur le flot d'instruction propre à chaque thread
- Les variables globales sont partagées par tous les threads
- L'espace d'adressage est le même!
- Les tables des fichiers ouverts est commune à tous les threads.
- La terminaison de l'activité initiale termine l'ensemble des threads.

Les appels systèmes liés aux threads

- identification d'une activité pthread_self()
- création d'une nouvelle activité pthread_create()
- arrêt d'une activité pthread_exit() ou return (plus simple)
- la libération des ressources d'une activité pthread_detach()
- abandon d'une activité p_thread_cancel()
- synchronisation des activités pthread_join()
- Sémaphores d'exclusion mutuelle p_thread_mutex()

Création d'une nouvelle activité : pthread_create()

```
#include<pthread.h>

int pthread_create(pthread_t* p_tid, pthread_attr_t* attr, void *(*fonction)(void arg), void* arg );
```

Le paramètre ***p_tid*** spécifie le numéro de l'activité. Elle est de type *pthread_t* (un entier).

Le paramètre ***attr*** est un paramètre qui définit les attributs de l'activité. On passera en générale le pointeur *NULL* qui donne à l'activité les attributs standards.

Le paramètre ***fonction*** est un pointeur sur la fonction que va exécuter le thread.

Le paramètre ***arg*** est un pointeur sur les arguments de la fonction.

Valeur renvoyée: retourne 0 si la création a réussi ou -1 en cas d 'erreur.

Exemple

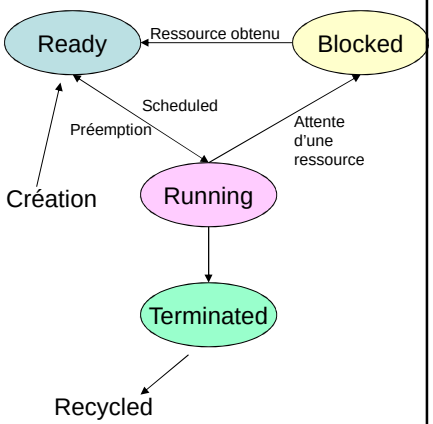
```
/* Exemple de création de threads */
#include<stdio.h>
#include<pthread.h>
#include<stdlib.h>
#include<errno.h>
pthread_t  pthread_id[3]; //tableau hébergeant les numéros des threads

/* Fonction excécutée par les threads */
void f_thread()
{
        printf( « Je suis le thread d'identite %d\n » ,getpid());
}

int main()
{
        int i;
        for(i=0;i<3;i++){
                if(pthread_create(pthread_id+i,NULL, (void*) f_thread,NULL)==-1)
                        perror( « Erreur lors de la création du thread\n » );
        }
        printf( « Je suis le processus initial de pid: %d\n » ,getpid());
        sleep(3); //permet d'attendre la terminaison de tous les threads
        exit(1);
}
```

gcc –lpthread …

Etat d'un thread

- Un *thread* peut se trouver dans un des quatre états suivants:

Ready	Le thread attend que le processeur soit disponible
Running	Le thread est en cours d'exécution.
Blocked	Le thread est bloqué. Le thread attend une ressource particulière (I/O), attend la libération d'un mutex, une variable de condition, etc.
Terminated	Le thread s'est terminé dans la mesure où l'exécution de la fonction appelé (par *pthread_create()*)est terminé. Mais il peut rester certaines ressources non libérées.



terminaison d'une activité : pthread_exit()

```
#include<pthread.h>

int pthread_exit (void* p_status);
```

Cette fonction termine l'activité (c'est l'activité qui appelle cette fonction qui se termine).

Le paramètre ***p_status*** correspond à un pointeur sur le code de retour du thread.

Valeur renvoyée: retourne 0 en cas de réussite ou -1 en cas d 'erreur.

Synchronisation des activités *:pthread_join()*

```
#include<pthread.h>

int pthread_join (pthread_t tid, void** status);
```

Cette fonction permet à une activité d'attendre la terminaison d'une autre activité et de récupérer le code de retour de cette activité (définit par l'appel pthread_exit()).

Le paramètre ***tid*** est l'identité de l'activité pour laquelle on attend la terminaison.

Le paramètre ***status*** est un pointeur sur un pointeur du code de retour. Attention, ce fonctionnement correspond à un retour effectué par *pthread_exit()*. Dans le cas d'un retour par *return()*, le second paramètre sera un pointeur sur le code de retour.

Valeur renvoyée: retourne 0 en cas de réussite ou -1 en cas d 'erreur.

libération des ressources d'une activité : pthread_detach()

```
#include<pthread.h>

int pthread_detach (pthread_t* p_tid);
```

Cette fonction libère les ressources d'une activité (ce n'est pas l'activité libéré qui appelle cette fonction). Elle ne termine pas l'activité si elle est en cours d'exécution, elle indique que les ressources devront être restitué une fois l'activité terminé.

Le paramètre ***p_tid*** est un pointeur sur l'identité de l'activité pour laquelle on libérera les ressources.

Valeur renvoyée: retourne 0 en cas de réussite ou -1 en cas d 'erreur.

Divers, mais important

- L'appel par une des activités à l'un des appels « *exec* » termine toutes les activités excepté celle qui a appelé « *exec* ».
- Un appel à exit() termine l'ensemble des activités quelque soit le *thread* appelant (initial ou non).

Synchronisation des processus:
Sémaphores, mutex, variables de conditions

Problème de synchronisation: zones critiques

- Une zone critique est une partie de code qui accède à une ressource partagée qui ne peut être utilisée par un autre processus.
- Exemple:

Thread 1

```
i=fonction()
resultat = tab[i]
```

Thread 2

```
i=fonction()
resultat = tab[i]
```

```
i=fonction()
i=fonction()
resultat = tab[i]
resultat = tab[i]
```

↓ temps

Problème de synchronisation: attente d'un résultat / condition

- L'exécution d'une partie d'un thread peut dépendre du résultat d'un autre thread (ou du processus initial)

Bout de code 1

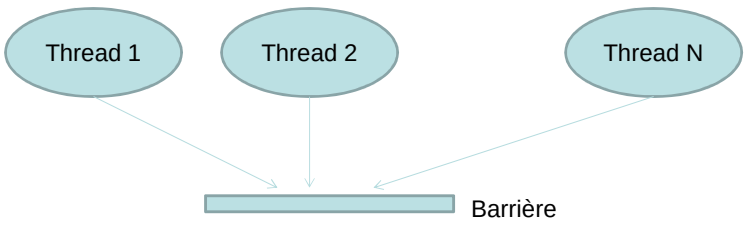
Le reste du code dépend du code 3 de l'autre thread

Bout de code 2

Bout de code 3

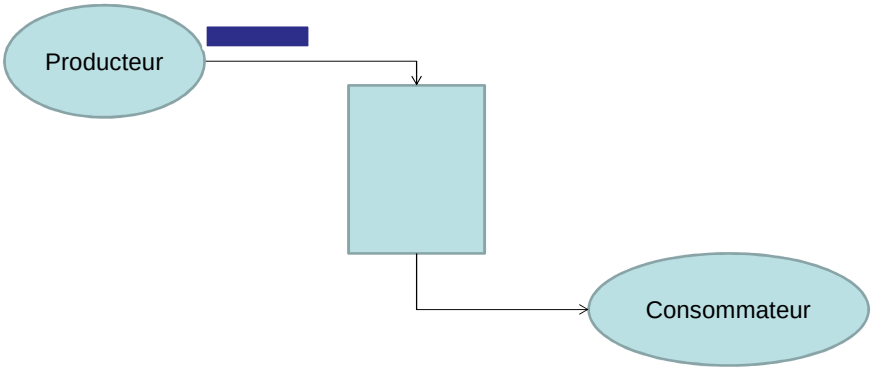
Barrière de synchronisation

- Un ensemble de threads doivent s'attendre mutuellement avant de continuer.



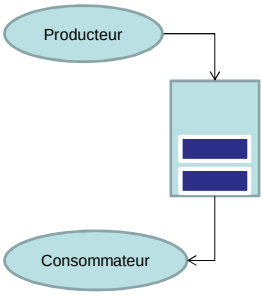
Principe du producteur consommateur

- Le système de producteur consommateur est un modèle classique de programmation parallèle.
  - Cas typique d'un processus qui ne peut s'exécuter que si les autres ont fait leur travail.

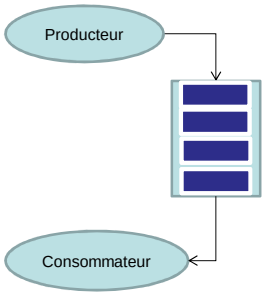


Principe du producteur consommateur: problématique

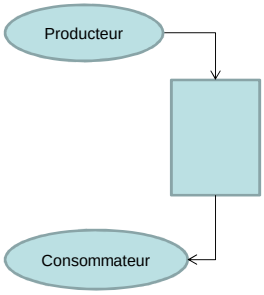
- La file est pleine: le producteur doit être mis en attente
- La file est vide: le consommateur doit être mis en attente
- Problème de synchronisation



Pas de problème: les deux processus peuvent produire et consommer.



Le producteur doit attendre que la file se libère. Il doit être débloqué quand c'est le cas.



Le consommateur doit attendre des ressources disponibles. Il doit être débloqué quand c'est le cas.

Les solutions techniques

- 3 outils: mutex / sémaphore /variables de conditions

Zone critique (zone d'exclusion mutuelle)	Producteur / Consommateur	Lecteur / rédacteur	Barrière de synchronisation
Mutex Sémaphore	Sémaphores Variables de conditions	Sémaphores Variables de conditions	Sémaphores

Mutex:
Sémaphore à exclusion mutuelle

Sémaphores d'exclusion mutuelle: mutex

- Un mutex est un objet d'exclusion mutuelle permettant
  - de protéger des données partager ne devant pas subir de modifications simultanées.
  - Implémenter des sections critiques
  - Synchroniser des *threads*.
- Un mutex a deux états possibles:
  - déverrouillé (pris par aucun thread)
  - verrouillé (pris par un thread).
- Un mutex ne peut être pris que par un seul thread à la fois. Un thread qui tente de verrouiller un mutex déjà verrouillé est suspendu jusqu'à ce que le mutex soit déverrouillé
- Le fonctionnement est le suivant
  - initialisation d'un mutex
  - verrouillage (lock) du mutex
    - permet à l'activité ayant bloqué le mutex de faire ce qu'elle veut sur la variable (ou une opération quelconque)
    - les autres activités vérifie au travers d'une fonction que le mutex est libre avant de modifier la variable (ou d'effectuer l'opération), la fonction renvoi un code d'erreur (-1) pour indiquer que le mutex n'est pas libre
  - libération du mutex (unlock)
  - Les autres activités peuvent maintenant verrouiller le mutex pour avoir accès à la variable (ou à l'opération).

pthread_mutex_init()

```
#include<pthread.h>

int pthread_mutex_init (pthread_mutex_t* p_mutex, pthread_mutexattr_t attr);
```

Cette fonction initialise un mutex.

Le paramètre ***p_mutex*** est un pointeur sur le mutex. Le mutex doit être déclaré au préalable, le type est *pthread_mutex_t*.

Le paramètre ***attr*** indique les attributs du mutex. Il est généralement initialisé par la constante ***pthread_mutex-attr_default*** qui décrit les attributs par défaut.

Valeur renvoyée: retourne 0 si on la création a réussi ou -1 en cas d 'erreur.

Initialisation statique

- Les variables de type **pthread_mutex_t** peuvent aussi être initialisées de manière statique, en utilisant les constantes **PTHREAD_MUTEX_INITIALIZER.**

```
pthread_mutex_t monMutex= PTHREAD_MUTEX_INITIALIZER;
```

pthread_mutex_lock()

```
#include<pthread.h>

int pthread_mutex_lock (pthread_mutex_t* p_mutex);
```

Cette fonction verrouille un mutex. Si le mutex a déjà été verrouillé par une autre activité, la fonction se met en attente jusqu'au déverrouillage par l'autre activité.

Le paramètre ***p_mutex*** est un pointeur sur le mutex. Le mutex doit être initialisé au préalable.

Valeur renvoyée: retourne 0 si le verrouillage a réussi ou -1 en cas d 'erreur.

pthread_mutex_trylock()

```
#include<pthread.h>

int pthread_mutex_trylock (pthread_mutex_t* p_mutex);
```

Cette fonction verrouille un mutex. Si le mutex a déjà été verrouillé par une autre activité, la fonction ne se met pas en attente mais renvoi un code de retour spécial (voir valeur renvoyée).

Le paramètre ***p_mutex*** est un pointeur sur le mutex. Le mutex doit être initialisé au préalable.

Valeur renvoyée: retourne 1 si le verrouillage a réussi, retourne 0 si le mutex a déjà été verrouillé par une autre activité et renvoi -1 en cas d 'erreur.

pthread_mutex_unlock()

```
#include<pthread.h>

int pthread_mutex_unlock (pthread_mutex_t* p_mutex);
```

Cette fonction déverrouille un mutex.

Le paramètre ***p_mutex*** est un pointeur sur le mutex. Le mutex doit être initialisé au préalable.

Valeur renvoyée: retourne 0 si le déverrouillage a réussi et renvoi -1 en cas d 'erreur.

Exemple d'utilisation d'un mutex

```
/* mutex sur une variable mutex 1 permettant l'affichages de deux messages dans le bon ordre */
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mutex1;
pthread_t       pthread_id;

void f_thread()
{
                pthread_mutex_lock(&mutex1);
                printf( « Thread %d\n » ,getpid());
                printf( « Message Numéro 2\n » );
                pthread_mutex_unlock(&mutex1);

}
int main()
{

                pthread_mutex_init(&mutex1, NULL);
                pthread_mutex_lock(&mutex1);
                if(pthread_create(&pthread_id, NULL, (void *) f_thread, NULL)==-1)
                        perror( « Erreur pthread_create()\n » );
                printf( « Message numéro 1\n » );
                pthread_mutex_unlock(&mutex1);

                //On attend la terminaison du thread avant d'arrêter le processus initial
                pthread_join(pthread_id,NULL);
                exit(0);
}
```

Le mutex ayant été déjà été verrouillé par l'activité initiale, le thread se met en attente. Une fois le premier message afficher, le mutex sera déverrouillé.

L'activité initiale verrouille le mutex.

Sémaphore

Sémaphore

- Un sémaphore est une variable gérée par le système.
- Il prend une valeur entière positive (au sens large)
- Il peut être incrémenté (P) ou décrémenté (V)
- Une décrémentation sur un sémaphore nul peut être bloquant (ou échoué)

Deux types d'implémentation

- Deux types d'implémentation des sémaphores :
  - System V
  - POSIX
- S'appliquent aux
  - processus lourds issus d'un fork (System V),
  - aux processus légers (System V ou Posix),
  - processus complètement indépendant (sémaphores nommés).

Liste des appels pour les sémaphores POSIX

- Déclaration: d'un sémaphore:

```
sem_t monSemaphore;
```

- Initialisation d'un sémaphore

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

- Décrémente un sémaphore

```
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
```

- Incrémente un sémaphore

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

- Obtenir la valeur d'un sémaphore

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

- Destruction du sémaphore

```
int sem_destroy(sem_t *sem);
```

Condition Variables

Variables de conditions: principes

- Les variables de conditions servent à synchroniser des threads sur une condition:
  - Un thread peut signaler aux autres threads qu'une condition est remplie
  - Un thread peut attendre qu'une condition soit remplie pour continuer à s'exécuter
- Il existe donc deux appels systèmes pour gérer ces conditions:
  - pthread_cond_signal : signal que la condition est maintenant satisfaite / un seul thread est débloqué
  - pthread_cond_broadcast : signal que la condition est maintenant satisfaite / tous les threads en attente sont débloqués
  - pthread_cond_wait: attend qu'un autre thread signal que la condition est satisfaite
- S'utilise conjointement avec un mutex
  - pthread_cond_wait déverouille le mutex pendant l'attente et le reprend à la réception du « signal ».
  - Pas d'effet sur les mutex pour les autres.