INF4033 - Partie 6 : Pour aller plus loin avec les Pthread

Alexandre BRIÈRE
alexandre.briere@esiea.fr

Plan

- Rappels
- Threads et gestion des processus
- Threads et signaux
- Les sémaphores
- Exécution unique d’une fonction
- Données spécifiques
- L’ordonnancement

Rappels (1/2)

Les threads sont des « processus légers » s'exécutant à l'intérieur d'un processus :

- partage de la mémoire (sauf pile)
- communications simplifiées (par passage de pointeurs)
- commutation entre threads plus rapide qu'entre processus

Rappels (2/2)

- Pthread
  - création : `pthread_create`
  - destruction : `pthread_exit`
  - attente : `pthread_join`
- Mutex
  - création : `pthread_mutex_init`
  - destruction : `pthread_mutex_destroy`
  - acquisition : `pthread_mutex_lock` ou `pthread_mutex_trylock`
  - relâchement : `pthread_mutex_unlock`
- Condition
  - création : `pthread_cond_init`
  - destruction : `pthread_cond_destroy`
  - attente : `pthread_cond_signal` ou `pthread_cond_broadcast`
  - réveil : `pthread_cond_wait`

—> exemple_1.c & exemple_2.c

Threads et gestion des processus

`fork`

Le nouveau processus possède :

- Uniquement le thread ayant appelé `fork`
- Les données allouées sur le tas par TOUS les threads du processus parent
  => mais plus les threads pour les désallouer
- Les différents verrous possédés par TOUS les threads du processus parent
  => sans même le savoir et sans jamais les relâcher

`exec`

L'appel à exec dans un thread termine tous les autres threads.
Le pthread appelant exécute le main du programme chargé.

`exit`

Un appel à exit dans un thread termine le processus et tous ses threads.

Threads et signaux

- Envoyer un signal à un autre thread via :

```
int pthread_kill(
  pthread_t id, // thread cible
  int signal)   // signal envoyé
```

- Possède son propre masque qu'il hérite lors de sa création

- Modifie son masque via :

```
int pthread_sigmask(
  int op,             // opération à réaliser
  sigset_t *newSig, // nouveau masque
  sigset_t *oldSig) // ancien masque
```

- Attend un signal via :

```
int sigwait(
  const sigset_t *ens, // ensemble des signaux attendus
  int *sig)            // signal reçu
```

Sémaphores

Les mutex sont des « sémaphores binaires »

- Mutex : un unique thread en section critique
- Sémaphore : de 0 à N threads en section critique

API

- `int sem_init(sem_t *sem, int portee, unsigned valeur)`
- `int sem_wait(sem_t *sem)`
- `int sem_trywait(sem_t *sem)`
- `int sem_post(sem_t *sem)`
- `int sem_destroy(sem_t * sem)`

Exécution unique

Pour les fonctions qui doivent s'exécuter une unique fois :

- ➡ Initialisation d'un mutex, d'une condition, etc.
- ➡ Ouverture d'un fichier
- ➡ Traitement particulier

`int pthread_once(pthread_once_t *once, void (*fonc))`

`once` : variable externe statique de type pthread_once_t

`fonc` : la fonction a n'exécuter qu'une seule fois

—> exemple_3.c

Données spécifiques

Réserver un ensemble de données réparti entre les threads.

Création de la clef : `pthread_key_create`

Stockage d'une valeur : `pthread_setspecific`

Récupération de la valeur : `pthread_getspecific`

Ordonnancement

Un thread possède trois attributs d'ordonnancement :

- `inheritsched` : défini si les deux autres sont hérités
- `schedpolicy` :
  - ➡ SCHED_OTHER : temps partagé
  - ➡ SCHED_FIFO : temps partagé avec priorité
  - ➡ SCHED_RR : temps partagé avec priorité et quantum
- `schedparam` : définie la priorité via le champs `sched_priority`

On peut connaitre les différente priorité grâce à :

```
int sched_get_priority_min(schedpolicy)
int sched_get_priority_max(schedpolicy)
```

—> exemple_4.c & exemple_5.c