

Threads POSIX

Introduction

Objectifs d'enseignement du cours:

- définir les notions de processus, de synchronisation et de communication entre processus;
- présenter et résoudre quelques problèmes types liés à la gestion de la concurrence;
- *illustrer quelques solutions avec des instruments de programmation auxquels les étudiants ont facilement accès.*

Les threads POSIX sont un tel instrument.

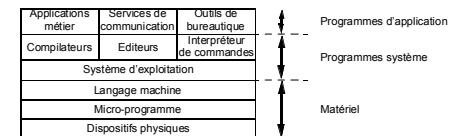
Plan

1. Introduction
2. Threads POSIX
3. Mécanismes de synchronisation définis par POSIX

Bibliographie:

- K.A. Robbins, S. Robbins *Practical UNIX Programming*, Prentice Hall, 1996
- C.J. Northrup *Programming with UNIX threads*, J. Wiley, 1996
- B. Lewis, D.J. Berg *Threads Primer*, Prentice-Hall, 1996
- B. Nichols, D. Buttlar, J. Proulx Farrell, *Pthreads Programming*, O'Reilly, 1998

Les deux fonctions d'un système d'exploitation



- fournir au programmeur une abstraction (une interface de programmation commode) de la machine sur laquelle il sévit;
- fournir les outils permettant de gérer les ressources de l'ordinateur et de partager et protéger l'information qui y réside.

POSIX = Portable Operating System Interface

- Appelé également IEEE Std 1003. *Qu'est-ce qu'un standard?*
- POSIX n'est pas limité aux systèmes UNIX (par ex. Windows)
- POSIX est un ensemble de standards:
 - POSIX.1 (1990): interface C pour la programmation système (API);
 - POSIX.1b (1993): extensions pour le temps réel (langage C);
 - **POSIX.1c** (1995): extension threads;
 - POSIX.2 (1992): shells et utilitaires;
 - etc.
- Toute implémentation "POSIX-compliant" doit offrir l'interface définie par POSIX.1.

Remarques

Un thread POSIX est créé dynamiquement grâce à la fonction `pthread_create`:

```
#include <pthread.h>
int pthread_create( pthread_t *pthread,
                  pthread_attr_t *attr,
                  void *(*fonction)(void *),
                  void *arg);
```

- correspond à la demande de création d'un nouveau thread dans le processus auquel le thread appelant appartient.
- la fonction retourne directement le code d'erreur:
 - une valeur `NULL` pour `*attr` donne une valeur par défaut aux attributs;
 - valeur 0 pour un succès;
 - n'emploie par la variable `errno` comme les autres fonctions POSIX.

Threads POSIX

Avec les threads POSIX on cherche à mettre à disposition du programmeur l'abstraction de processus au niveau du programme.

SUN Solaris et Linux offrent les threads POSIX.

Les threads POSIX et les threads SUN Solaris sont assez proches (mais des différences existent, par exemple, le réglage du taux de multiprogrammation des threads SUN Solaris, n'est pas défini dans POSIX).

En C, un thread POSIX est identifié par un descripteur de type `pthread_t` (peut être un entier non signé comme sous Solaris ou Linux, ou un autre type).

```
pthread_attr_t:
  scope          int      (PTHREAD_SCOPE_SYSTEM
                          PTHREAD_SCOPE_PROCESS)

  stackaddr      void *
  stacksize      size_t
  detachstate    int      (PTHREAD_CREATE_JOINABLE
                          PTHREAD_CREATE_DETACHED)

  inheritsched   int      (PTHREAD_INHERIT_SCHED
                          PTHREAD_EXPLICIT_SCHED)

  schedpolicy    int      (SCHED_OTHER /*système*/
                          SCHED_FIFO
                          SCHED_RR)

  schedparam     struct sched_param *
```

Terminaison d'un processus (qu'il soit constitué d'un seul ou de plusieurs threads):

- lorsqu'un thread exécute l'appel système `exit` (cf appels systèmes Unix);
- lorsque le thread qui exécute la routine `main` se termine;
- lorsqu'un signal provoquant la terminaison du processus est reçu (cf appels systèmes Unix).

Autres fonctions liés aux threads:

- `int pthread_exit(int *status)`: termine le thread appelant avec une valeur de retour égale à `status`;
- `pthread_kill`: permet d'envoyer un signal à un thread (concept Unix, pas les signaux des moniteurs);
- `pthread_join`: suspend le thread appelant jusqu'à terminaison du thread désigné;
- `pthread_self`: retourne l'ident du thread.

Pour plus d'informations (sous Unix):

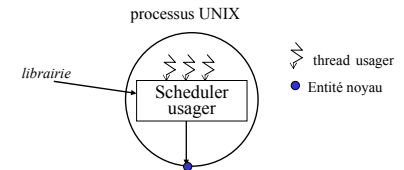
```
man pthread
man pthread_create
man pthread_exit
```

etc.

Threads usager vs. threads noyau

Threads usagers:

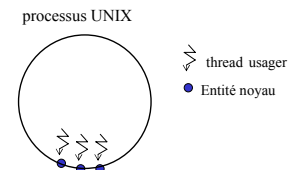
threads mis en œuvre au-dessus du noyau UNIX (invisibles au noyau système).



Exemples:

- 1) "threads" de SUN Solaris (*unbound thread* dans la terminologie SUN);
- 2) threads d'un programme Java (JVM = un processus UNIX).

Threads du noyau:



Exemple:

"lightweight processes" de SUN Solaris (*bound thread* dans la terminologie de SUN).

Avantages des threads usager:

- *threads usagers* plus efficaces (création, synchronisation).

Inconvénients des threads usager:

- appel système par un *thread usager*: si l'appel est bloquant, tous les autres *threads usagers* sont bloqués.

Pour éviter le problème, la librairie qui implémente les threads doit transformer l'appel système en un appel non bloquant (asynchrone): le thread qui fait l'appel est bloqué par la librairie de threads; il est débloqué lorsque l'appel se termine.

Mécanismes de synchronisation**Trois mécanismes****mutex:**

«verrous» du cours de Programmation concurrente;

sémaphores:

«sémaphores» du cours de Programmation concurrente;

variables de condition:

plus ou moins les «signaux» liés aux moniteurs du cours de Programmation concurrente.

Le modèle de threads de POSIX est un modèle flexible qui permet d'offrir à la fois les threads usagers et les threads du noyau.

Ceci est déterminé par l'attribut (contention) *scope*, égal soit à `pthread_scope_process`, soit à `pthread_scope_system` (cf `man pthread_create`).

Différentes politiques de scheduling peuvent être choisies (aussi bien pour les threads usagers que pour les threads du noyau):

FIFO, round robin, avec préemption (une priorité peut être attribuée à un thread: valeur élevée = haute priorité).

Mutex: (variable du type `pthread_mutex_t`)

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
(initialisation statique)
```

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                      pthread_mutexattr_t *attr);
(allocation mémoire et initialisation dynamique)
```

```
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
(définition des attributs du mutex)
```

- opération de verrouillage:
`int pthread_mutex_lock(pthread_mutex_t *mutex)`
- `pthread_mutex_unlock`: opération de déverrouillage;
- `pthread_mutex_trylock`: idem à `pthread_mutex_lock`, mais non bloquant (retourne 0 si le verrou est obtenu, un code d'erreur sinon);
- `pthread_mutex_destroy`: détruit le mutex et restitue la mémoire.

Seul le thread qui a effectué l'opération de verrouillage, peut effectuer l'opération de déverrouillage.

Sémaphore: (variable du type `sem_t`)

- `sem_init`: allocation mémoire et initialisation;
- `sem_wait`: opération P sur le sémaphore;
- `sem_post`: opération V sur le sémaphore;
- `sem_trywait`: idem à `sem_wait`, mais non bloquant (retourne 0 si le sémaphore est passé avec succès, un code d'erreur sinon);
- `sem_destroy`: détruit le sémaphore et restitue la mémoire.

Les opérations sur les sémaphores ne sont pas préfixées par "pthread_".

Portée du mutex: définie par un attribut lors de la création du mutex.

- **portée locale:** portée limitée au processus (défaut);
- **portée globale:** permet de synchroniser des threads appartenant à plusieurs processus. Requiert que le mutex soit alloué dans une zone de mémoire partagée.

Variables de condition:

- ressemblent aux signaux vus dans le contexte des moniteurs;
- à utiliser conjointement aux mutex;
- permet à un thread de se mettre en attente d'une condition tout en libérant l'exclusion mutuelle imposée par le mutex.

(Variable de) condition: variable du type `pthread_cond_t`

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

(initialisation statique)

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

(allocation mémoire et initialisation dynamique)

```
int pthread_condattr_init(pthread_condattr_t *attr);
```

(définition des attributs de la (variable de) condition)

-
- réalisation de la condition:

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

débloque un thread en attente de la condition. Si plusieurs threads sont en attente, le choix du thread débloqué dépend de la politique de scheduling;

- `pthread_cond_broadcast`: débloque tous les threads en attente du signal (les threads débloqués vont entrer en compétition pour l'exclusion mutuelle);
- `pthread_cond_destroy`: détruit la condition et restitue la mémoire.
 Portée de la variable de condition: idem au mutex.

-
- attente sur une condition:

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

prend en paramètre une **condition** et un **mutex**. Bloque le thread en attente sur la variable de condition et libère l'exclusion mutuelle du mutex. Lorsque le thread est débloqué, il est à nouveau en possession de l'exclusion mutuelle fournie par le mutex (la demande d'entrée en exclusion mutuelle est faite implicitement);

- `pthread_cond_timedwait`: idem à `pthread_cond_wait`, avec la possibilité de spécifier un time-out.

Remarque

Le mécanisme des variables de condition n'assure pas que si une expression booléenne `COND` est vraie au moment où un thread T_1 exécute `pthread_cond_signal`, l'expression `COND` est encore vraie au moment où le thread débloqué T_2 reprend son exécution (parce qu'un autre thread T_3 peut entrer en section critique avant T_2).

Cela conduit en général à adopter le style de programmation suivant:

```
while not COND {
    pthread_cond_wait (...);
}
```