Système d'exploitation et programmation système

Cours 7

EISTI

2009/2010

Département informatique

multithreading

Processus

- Les processus fils et le processus père sont des processus indépendants ayant chacun :
  - son espace d'adressage
  - sa pile d'exécution
- Création coûteuse
- Changement de contexte : coûteux
- Partage de données délicat

Thread

- Un processus qui crée des *threads* ne crée pas d'autres processus, tout se passe dans le même espace d'adressage
- Chaque *thread* est associé à une pile d'exécution indépendante
- Handler
- Un *thread* ressemble fortement à un processus fils classique à la différence qu'il partage beaucoup plus de données avec le processus qui l'a créé:
  - Les variables globales
  - Les variables statiques locales
  - Les descripteurs de fichiers (file descriptor)
- Ce partage implique la gestion par le programmeur de la concurrence d'accès à ces variables

Application adaptée

- Client/serveur
  - Remplacement du fork par un *thread*
  - Permet une meilleure communication

Interface POSIX (**P**ortable **O**perating **S**ystem **I**nterface)

- Appelé également IEEE Std 1003.
- 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
- Interface POSIX**.1c**
  - Nom des types : pthread_objet_t
  - Nom des fonctions : pthread_objet_operation

Création

- *Thread* principal
  - Création d'un processus : création d'un *thread* natif
  - Execution du main dans ce *thread* principal
  - Si terminaison du thread principal➔ terminaison des autres *threads* du processus

Création

- Un thread POSIX est créé dynamiquement dans le processus auquel le thread appelant appartient grâce à la fonction pthread_create :

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

- la fonction retourne directement le code d'erreur:
  - n'emploie pas la variable **errno** comme les autres fonctions POSIX
  - valeur de retour 0 pour un succès sinon un code d'erreur non nul est renvoyé;

EAGAIN :

– il y a plus de **PTHREAD_THREADS_MAX** threads actifs.

– la valeur attr est invalide

Création (2)

- L’argument *pthread* , pointeur sur le thread créé
- L’argument *attr* indique les attributs du nouveau thread créé (ressource système, priorité,…):
  - Si *attr* est égal à **NULL**, les attributs par défaut sont utilisés : le *thread* créé est joignable (non détaché) et utilise la politique d'ordonnancement normale (pas temps-réel).
- L’argument *start_routine* la fonction exécuté par le thread
- L’argument *arg*, le premier argument passé à la fonction *start_routine*

Terminaison

- Le thread s’achève :
  - implicitement lorsque la fonction *start_routine* s'achève.
  - explicitement en appelant **pthread_exit** , cette fonction **pthread_exit** ne rend jamais la main.
- 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).

Primitives liées aux threads

- **void** pthread_exit(int *status): termine le thread appelant avec une valeur de retour égale à status; qui peut être consulté par un autre thread en utilisant pthread_join()
- **int** pthread_kill(pthread_t *thread*, **int** *sig*) : permet d'envoyer un signal à un thread (concept Unix, pas les signaux des moniteurs);
- **int** pthread_join(pthread_t *th*, void ***thread_return*) : suspend le thread appelant jusqu'à terminaison du thread désigné;
- pthread_t pthread_self(void) : retourne l'identificateur du thread
- **int** pthread_detach(pthread_t *th*) : place un thread en cours d'éxécution dans l'état détaché

Exemple avec un thread

```
#include <stdio.h>
 #include <stdlib.h>
#include <pthread.h>
static void * my_thread(void * vargp) {
        int * retval = malloc(sizeof (int));
        if(retval != NULL) { *retval = 42; }
        pthread_exit(&retval);
}
int main(void) {
          int * i = NULL;
          pthread_t tid;
          pthread_create(&tid, NULL, my_thread, NULL);
          pthread_join(tid, (void**)&i);
          if(i != NULL) {
                    printf("%d\n", *i);
                    free(i),
           }
           return 0;
 }
```

Primitives liées aux threads (2)

- La primitive pthread_attr_init(pthread_t *_pthread_) initialise l'ensemble des attributs référencé par *attr* d'un thread avec des valeurs par défaut.
- La primitive pthread_attr_destroy(pthread_t *_pthread_) détruit l'ensemble des attributs référencé par *attr*.
- Ces deux primitives retournent 0 en cas de succès, sinon un code erreur
- Pour plus d'informations (sous Unix):
  - man pthread
  - man pthread_create
  - man pthread_exit

Primitives de manipulation de l'argument *attr*

int pthread_attr_setdetachstate(pthread_attr_t **attr*, **int** *detachstate*);

int pthread_attr_getdetachstate(const pthread_attr_t **attr*, **int** **detachstate*);

- detachstate : Spécifie le comportement par défaut de la fin d'un thread
  - PTHREAD_CREATE_JOINABLE : identité et valeur de retour conservées jusqu'à ce qu'un *thread* en prenne connaissance
  - PTRHEAD_CREATE_DETACHED : la fin d'un *thread* entraîne la libération des ressources allouées

int pthread_attr_setschedpolicy(pthread_attr_t ***attr*, int** *policy*);

int pthread_attr_getschedpolicy(const pthread_attr_t ***attr*, int **policy*);

- schedpolicy : change la politique et les paramètres d'ordonnancement pour le thread :
  - SCHED_OTHER : scheduling Unix (par défaut, non temps réel)
  - SCHED_FIFO : scheduling Fifo (temps réel, premier dans la liste, premier exécuté)
  - SCHED_RR : scheduling Round-Robin

Exemple

```
pthread_attr_t attr;
pthread_attr_init(&attr); /* initialisation par défaut*/
pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
pthread_create(&tid, &attr, start_func, arg);
```

Synchronisation

```
#include <pthread.h>

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

- Se mettre en attente de terminaison

  pthread_join suspend l'exécution du *thread* appelant jusqu'à ce que le thread identifié par *th* achève son exécution :

  – soit après avoir été annulé.

  – soit en appelant pthread_exit()

- **VALEUR RENVOYÉE**

En cas de succès, 0 est renvoyé sinon un code d'erreur non nul est renvoyé :

– **ESRCH** : Aucun *thread* correspondant à *th* n' a pu être trouvé.

– **EINVAL** : Le *thread th* a été détaché.

– **EINVAL** : Un autre *thread* attend déjà la mort de *th*.

– **EDEADLK** : L'argument *th* représente le *thread* appelant.

Détachement

```
#include <pthread.h>
int pthread_detach(pthread_t th);
```

- Cette primitive place le thread *th* dans l'état détaché. Cela garantit que les ressources mémoire consommées par *th* seront immédiatement libérées lorsque l'exécution de *th* s'achèvera

Remarque : cela empêche les autres threads de se synchroniser sur la mort de *th* en utilisant **pthread_join**.

- **VALEUR RENVOYÉE**

En cas de succès, 0 est renvoyé, sinon un code d'erreur non nul est renvoyé :

ESRCH : Aucun *thread* ne correspond à celui indiqué par *th*

EINVAL : le *thread th* est déjà dans l'état détaché

Exemple : de 2 *threads* d'affichage

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void main (int argc, char **argv) {
   pthread_t th1, th2;
   void *ret;
   if (pthread_create (&th1, NULL, my_thread_process, "1") < 0) {
        fprintf (stderr, "pthread_create error for thread 1\n");
        exit (1);
   }
   if (pthread_create (&th2, NULL, my_thread_process, "2") < 0) {
        fprintf (stderr, "pthread_create error for thread 2\n");
        exit (1);
   }
   (void)pthread_join (th1, &ret); //attente de la fin du thread th1
   (void)pthread_join (th2, &ret); //attente de la fin du thread th2
 }
```

Exemple (suite)

```
void *my_thread_process (void * arg) {
    int i;
    for (i = 0 ; i < 5 ; i++) {
        printf ("Thread %s: %d\n", (char*)arg, i);
        sleep (1);
    }
    pthread_exit (0);
}
```

Compilation /Exécution

```
gcc  -o thread1 thread1.c -lpthread
./thread1
Thread 1: 0
Thread 2: 0
Thread 1: 1
Thread 2: 1
Thread 1: 2
Thread 2: 2
Thread 1: 3
Thread 2: 3
Thread 1: 4
Thread 2: 4
```

Accès concurrent

- Problématique
  - *thread* partage les données des processus
- Trois mécanismes de synchronisation
  - mutex: exclusion mutuelle (verrous)
  - sémaphores: (introduit lors de la 5ème révision du standard)
  - Les «signaux» liés aux moniteurs du cours de Programmation concurrente)

Création /Initialisation de mutex

- Les variables de type pthread_mutex_t peuvent aussi être initialisées de manière statique, en utilisant les constantes :**pthread_mutex_t monMutex =**
  - **PTHREAD_MUTEX_INITIALIZER** (mutex rapides),
  - **PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP** (mutex récursifs),
  - **PTHREAD_ERREURCHECK_MUTEX_INITIALIZER_NP** (mutex à vérification erreur).

- **int** pthread_mutex_init (pthread_mutex_t **mutex*, pthread_mutexattr_t * *attr*);

  Allocation mémoire et initialisation dynamique. Cette primitive initialise le mutex pointé par *mutex* selon les attributs de mutex spécifié par *attr*.

  Si *attr* vaut **NULL**, les paramètres par défaut sont utilisés.

- **int** pthread_mutexattr_init (pthread_mutexattr_t **attr*) ;

  Cette primitive initialise l'objet attributs_de_mutex *attr* et le remplit avec les valeurs par défaut

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

  Cette primitive détruit le mutex et restitue la mémoire.

Verrouillage/Déverrouillage d'un mutex

- Verrouillage

  int pthread_mutex_lock (pthread_mutex_t *_mutex_)

  – **EINVAL** : le mutex n'a pas été initialisé.

- Deverrouillage

  int pthread_mutex_unlock (pthread_mutex_t *_mutex_)

  – **EBUSY** : le mutex est déjà verrouillé.

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

Portée 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.

Exemple : 2 *threads* d'affichage

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
static pthread_mutex_t my_mutex;
static int tab[5];
void main (int argc, char **argv) {
   pthread_t th1, th2;
   void *ret;
   pthread_mutex_init (&my_mutex, NULL); //initialise le mutex par défaut
   if (pthread_create (&th1, NULL, write_tab_process, NULL) < 0) {
      fprintf (stderr, "pthread_create error for thread 1\n");
      exit (1);
   }
   if (pthread_create (&th2, NULL, read_tab_process, NULL) < 0) {
     fprintf (stderr, "pthread_create error for thread 2\n");
     exit (1);
   }
   (void)pthread_join (th1, &ret);
   (void)pthread_join (th2, &ret);
}
```

Exemple (suite)

```
void *read_tab_process (void * arg) {
   int i;
   pthread_mutex_lock (&my_mutex);
   for (i = 0 ; i != 5 ; i++)
         printf ("read_process, tab[%d] vaut %d\n", i, tab[i]);
   pthread_mutex_unlock (&my_mutex);
   pthread_exit (0);
}
void *write_tab_process (void * arg) {
   int i;
   pthread_mutex_lock (&my_mutex);
   for (i = 0 ; i != 5 ; i++) {
      tab[i] = 2 * i;
      printf ("write_process, tab[%d] vaut %d\n", i, tab[i]);
      sleep (1); /* Relentit le thread d'ecriture... */
    }
   pthread_mutex_unlock (&my_mutex);
   pthread_exit (0);
}
```

Compilation /Exécution

```
gcc  -o thread2 thread2.c -lpthread
./thread2
write_process, tab[0] vaut 0
write_process, tab[1] vaut 2
write_process, tab[2] vaut 4
write_process, tab[3] vaut 6
write_process, tab[4] vaut 8

read_process, tab[0] vaut 0
read_process, tab[1] vaut 2
read_process, tab[2] vaut 4
read_process, tab[3] vaut 6
read_process, tab[4] vaut 8
```

Sémaphore

- sem_init: allocation mémoire et initialisation;

int sem_init(sem_t *sem, int ***pshared*****, unsigned int** ***valeur*****);**

- 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_".

Exemple : de 2 *threads* d'affichage

```
#include <stdio.h>  #include <stdlib.h>  #include <pthread.h> #include<semaphore.h>

static sem_t my_sem;
int the_end;
main (int ac, char **av) {
      pthread_t th1, th2; void *ret;
      sem_init (&my_sem, 0, 0);

      if (pthread_create (&th1, NULL, thread1_process, NULL) < 0) {
           fprintf (stderr, "pthread_create error for thread 1\n");
           exit (1);
      }

     if (pthread_create (&th2, NULL, thread2_process, NULL) < 0) {
        fprintf (stderr, "pthread_create error for thread 2\n");
        exit (1);
     }

    (void)pthread_join (th1, &ret);     (void)pthread_join (th2, &ret);

} // fin de main
```

Exemple (suite)

```
void *thread1_process (void * arg) {
     while (!the_end) {
          printf ("Je t'attend !\n");
         sem_wait (&my_sem);
     }
    printf ("OK, je sors !\n");
    pthread_exit (0);
 }

void *thread2_process (void * arg) {
     register int i;
     for (i = 0 ; i < 5 ; i++) {
         printf ("J'arrive %d !\n", i);
         sem_post (&my_sem);
         sleep (1);
      }
     the_end = 1;
     sem_post (&my_sem); /* Pour debloquer le dernier sem_wait */
     pthread_exit (0);
  }
```

Exemple (suite)

Dans cet exemple, le thread numéro 1 attend le thread 2 par l'intermédiaire d'un sémaphore.

Après compilation on obtient la sortie suivante:

Compilation /Exécution

```
gcc  -o thread3 thread3.c -lpthread
./thread3
Je t'attend !
J'arrive 0 !
Je t'attend !
J'arrive 1 !
Je t'attend !
J'arrive 2 !
Je t'attend !
J'arrive 3 !
Je t'attend !
J'arrive 4 !
Je t'attend !
OK, je sors !
```

FIN