Programmation systèmes et reseaux

Les Threads

Threads (Introduction)

Il est possible d'implémenter des algorithmes parallèles avec ce qu'on a vu jusque là (IPC), mais c'est lourd :

- fork – appel système lourd :
  - les processus fils et le processus père sont des processus indépendants
  - chaque processus a son espace d'adressage
  - chaque processus a sa pile d'exécution
  - changement de contexte (context switch) lent/coûteux
- communication inter-processus généralement lente
- partage de données délicat
- **Solutions** : les processus légers : « **threads** »

Processus légers : threads

- Un processus qui crée des threads ne crée pas d'autres processus, tout se passe dans le même espace d'adressage
  - un thread est une « partie » d'un processus
  - un processus est l'exécution d'un ensemble (>=1) de threads
- Chaque thread est associée à une pile d'exécution indépendante
- Différence entre un ensemble de threads et un ensemble de processus:
  - les threads partagent pratiquement tout : *les variables globales, les variables statiques locales, les descripteurs de fichiers ouverts, le PID, le PPID, les utilisateurs propriétaires, les handlers de signaux, le répertoire courant, le masque de fichiers, …)*
  - Ne partagent pas : *les identifiants des threads, la pile, le masque des signaux*
  - Ce partage implique la gestion par le programmeur de la concurrence d'accès à ces variables (*c-a-d., la synchronisation n'est plus gérée au niveau du système, mais est laissée à l'utilisateur*)

Threads: avantages et inconvénients

- Avantages par rapport aux processus
  - partage de la mémoire : mécanisme rapide de communication inter-thread
  - plus léger : moins de données système à recopier
  - plus rapide : le context-switch (le changement de contexte) est plus facile

- Les inconvénients sont (uniquement) des « difficultés » de programmation:
  - les threads utilisent les mêmes copies des librairies
  - il faut gérer la synchronisation (mutex, sémaphores...)
  - En cas de mauvaise synchronisation comportement « aléatoire » : bugs, segfaults, interblocage…

Threads noyau (kernel) et utilisateur

Où implémenter les threads ?

- Dans les bibliothèques utilisateur
  - contrôlés par l'utilisateur
  - POSIX Pthreads, Java threads
- Dans le noyau du SE
  - contrôlés par le noyau
  - Windows XP/2000, Solaris, Linux, UNIX, Mac OS X
- Solutions mixtes

Créer un thread

- Au départ, le processus est constitué d'un unique thread (thread principal) celui qui exécute la fonction main :
- Création d'un processus : création d'un thread natif
- Exécution du main dans ce thread principal
- Si terminaison du thread principal terminaison des autres threads du processus

Créer un thread (pthread_create())

```
#include <pthread.h>
int pthread_create(pthread_t *tid, const pthread_attr_t *attr,
void * (*func) (void*), void *arg)
```

- **pthread_t *tid:** pointeur vers l'adresse mémoire qui contient le TID (Thread Identifier) du pthread créé.
- **const pthread_attr_t *attr:** sert à préciser les attributs du pthread (taille de la pile, priorité, ressource systeme….).

  attr = NULL pour les attributs par défaut (*le thread créé est joignable (non détaché) et utilise la politique d'ordonnancement normale (pas temps-réel)*

- **void * (*func) (void*):** est la fonction à exécuter par le pthread

- **void *arg:** les arguments passé à de fonction func.

Attributs de thread

- Un pthread **n'a pas** de **PID propre** (ils partagent tous le même PID, celui du processus contenant les threads)

- L'identifiant d'un pthread est un objet du pthread_t : la norme ne dit pas ce qu'il y a dans pthread_t (objet opaque)

- L'appel renvoie **0** en cas de **succès**, sinon il renvoie une valeur non nulle identifiant l'erreur qui s'est produite
- **Erreurs**
  - **EAGAIN :** ressources insuffisantes pour créer un nouveau thread, ou une limite sur le nombre de threads imposée par le système a été atteinte
  - **EINVAL :** Paramètres invalides dans attr
  - **EPERM :** Permissions insuffisantes pour définir la politique d'ordonnancement et les paramètres spécifiés dans attr.

Attributs de thread

- Chaque thread est doté d'un certain nombre d'attributs, regroupés dans un type opaque **phtread_attr_t**. Les attributs sont fixés lors de la création du thread grâce au second argument de pthread_create( )

- Lorsque les attributs par défaut sont suffisants, on passe généralement un pointeur **NULL** sinon, il faut passer un pointeur sur un objet pthread_attr_t qui aura été préalablement configuré correctement.

- Dans un premier temps, il faut invoquer la fonction **pthread_attr_init(pthread_attr_t *attr)** en lui transmettant un pointeur sur une variable de type pthread_attr_t

Attributs de thread

- **pthread_attr_init(pthread_attr_t *attr):** initialise la structure d'attributs de thread (2ème paramètre de pthread_create) et la remplit avec les valeurs par défaut pour tous les attributs

- Une fois les attributs initialisés, on utilise les fonctions **pthread_attr_getXXX( )** et **pthread_attr_setXXX( )** pour consulter ou changer l'attribut correspondant

Exemple

pthread_setname_np(pthread_self(), "worker-1"): donner un nom à un thread

Primitives liées aux threads

Appel	Description
pthread_create	Créer un nouveau thread ou fil d'exécution
pthread_exit	Terminer le thread appelant
pthread_join	Attendre la fin d'un autre thread
pthread_mutex_init	Créer un mutex
pthread_mutex_destroy	Détruire un mutex
pthread_mutex_lock	Verrouiller un mutex
pthread_mutex_unlock	Relâcher un mutex

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

- **int pthread_join(pthread_t thread, void **retval)** : suspend le thread appelant jusqu'à terminaison du thread désigné (soit après avoir été annulé, soit terminé en appelant pthread_exit()); retval – un pointeur sur une variable (contenant un pointeur), où le code de retour sera copié

- **pthread_t pthread_self(void)** : retourne l'identificateur du thread

Primitives liées aux threads

- **int pthread_detach(pthread_t thread) :** place un thread en cours d'exécution dans l'état détaché sauf s'il un autre thread a déjà joint ce thread :
  - Cela garantie que les ressources mémoire consommées par thread seront immédiatement libérées lorsque l'exécution de thread s'achèvera
  - Cela empêche les autres threads de se synchroniser sur la mort de thread en utilisant pthread_joint()
  - Si on ne veut pas avoir à gérer la fin d'un thread, on peut le « détacher »

Fonction pthread_create()

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void * fonction_thread(void * arg);

int main (void)
{
 pthread_t thr;
 if (pthread_create(& thr, NULL, fonction_thread, NULL) != 0) {
 fprintf(stderr, "Erreur dans pthread_create\n");
 exit(EXIT_FAILURE);
}
while (1) {
 fprintf(stderr, "Thread Main\n");
 sleep(1);
 }
}
```

Fonction pthread_create()

```c
void * fonction_thread(void * arg)
{
while (1) {
 fprintf(stderr, "Nouveau Thread\n");
 sleep(1);
 }
}
```

```
$ gcc exemple-pthread-create-1.c -o exemple-pthread-create-1  -lpthread
$ ./exemple-pthread-create-1
Thread Main
Nouveau Thread
Thread Main
Nouveau Thread
Thread Main
Nouveau Thread
Thread Main
Nouveau Thread
Thread Main
Nouveau Thread
Thread Main
Nouveau Thread
Thread Main
Nouveau Thread
^C
```

Récupération du PID

- si on fait un **ps aux** sur un autre terminal **seul** **exemple-pthread-create-1** est visible

- Pour voir les threads il faut utiliser **ps maux**

Fin d'un thread

- Lorsque la fonction principale d'un thread se termine, celui-ci (le thread) est éliminé. La fonction doit renvoyer une valeur qui pourra être récupérée dans un autre thread.

- **Autre possibilité** :

void pthread_exit(void * retour);
Termine l'exécution du **pthread** et renvoie retour (éventuellement NULL)

Fin d'un thread

```
int main (void)
{
 pthread_t thr[3];
 int i;

for (i = 0; i < 3; i ++) {
 if (pthread_create(& thr[i], NULL, fonction_thread, (void *) i)
      != 0) {
  fprintf(stderr, "Erreur dans pthread_create\n");
  exit(EXIT_FAILURE);
 }
}
while (1) {
fprintf(stderr, "Thread Main\n");
sleep(1);
}
}
```

Fin d'un thread

```
void * fonction_thread(void * arg)
{
int num = (int) arg;
int i = 0;
while (1) {
 fprintf(stderr, "Thread %d, iteration %d\n", num, i);
 if ((num == 0) && (i == 1))
   pthread_exit(NULL);
 if ((num == 1) && (i == 3))
   return NULL;
 i ++;
 sleep(1);
}
}
```

3 threads :
- 1 terminé au bout de 2 secondes,
- un second au bout de 4 secondes

```
$./exemple-pthread-exit-1
Thread Main
Thread 0, iteration 0
Thread 1, iteration 0
Thread 2, iteration 0
Thread Main
Thread 0, iteration 1
Thread 2, iteration 1
Thread 1, iteration 1
Thread Main
Thread 2, iteration 2
Thread 1, iteration 2
Thread Main
```

Fin d'un thread

```
int main (void) {
pthread_t thr[3];
int i;

for (i = 0; i < 3; i ++) {
 if (pthread_create(& thr[i], NULL, fonction_thread, (void *) i)
     != 0) {
     fprintf(stderr, "Erreur dans pthread_create\n");
      exit(EXIT_FAILURE);
 }
}
pthread_exit(NULL);
}
void * fonction_thread(void * arg) {
int num = (int) arg;
while (1) {
 fprintf(stderr, "Thread %d\n", num);
 sleep(1);
}
}
```

Fin d'un thread

```
$./exemple-pthread-exit-2
Thread 0
Thread 1
Thread 2
Thread 0
Thread 2
Thread 1
Thread 2
```

- le thread principal se termine
- les autres continuent

Si c'est le comportement désiré alors utiliser pthread_exit dans le main (et pas return ou exit)

Fonctions pthread_join()

void pthread_join(pthread_t tid, void * *retour);

- Attend la fin d'un pthread. L'équivalent de **waitpid** des processus sauf qu'on doit spécifier le **tid** du **pthread** à attendre.
- retour sert à récupérer la valeur de retour et l'état de terminaison.
- Peut échouer si le **tid** n'existe pas, s'il est détaché

Fonctions pthread_join()

```
int main (int argc, char * argv[]) {
pthread_t * thr = NULL;
void * ptr;
int i;
int n;
thr = calloc(argc - 1, sizeof(pthread_t));
if (thr == NULL) {
 perror("calloc");
 exit(EXIT_FAILURE);
}
for (i = 1; i < argc; i ++) {
 n = atoi(argv[i]);
 if (pthread_create(& thr[i-1], NULL, fonction_thread,
     (void *) n) != 0) {
       fprintf(stderr, "Erreur dans pthread_create\n");
       exit(EXIT_FAILURE);
 }
}
```

Fonctions pthread_join()

```
...
for (i = 1; i < argc; i ++) {
 pthread_join(thr[i - 1], & ptr);
 fprintf(stderr, "%d -> %d\n", atoi(argv[i]), (int) ptr);
}
return EXIT_SUCCESS;
}

void * fonction_thread(void * arg)
{
int num;
 num = (int) arg;
 return (void *) (num * num);
}
```

Fonctions pthread_join()

```
$ ./exemple-pthread-join-1 1 2 3
1 -> 1
2 -> 4
3 -> 9
$
```

- autant de thread que d'arguments
- la fonction de chaque thread calcule le carré et le renvoie.
- le thread principal récupéré les résultats et affiche

Détachement des threads

- Un pthread est détaché par la fonction:

int pthread_detach(pthread_t tid)) ;

  - indique que l'on détache un thread (identifié par tid) pour que ses ressources soient libérées automatiquement à la fin de son exécution.
  - Echoue si le thread n'existe pas ou **déjà détaché**.
  - **Pourquoi cette fonction ?** parce que sinon les codes de retour (et ses piles) sont conservées : **risque de saturation** !
  - Avec **detach** la pile et le code de retour sont libérés dès la fin du thread.

Détachement des threads

```c
int main (int argc, char * argv[])
{
 pthread_t thr; /* La meme variable sera ecrasee a chaque fois *
 int nb_lances = 0;
 while (pthread_create(& thr, NULL, fonction_thread, NULL) == 0)
   pthread_detach(thr);
   nb_lances ++;
 if ((nb_lances % 100) == 0)
   fprintf(stderr, "%d thread lances\n", nb_lances);
 usleep(10000); /* 10 ms */
}
fprintf(stderr, "Echec de creation apres %d threads\n", nb_lance
return EXIT_SUCCESS;
}

void * fonction_thread(void * arg)
{
 return NULL;
}
```

Détachement des threads

```
$ ./exemple-pthread-detach
100 thread lances
...
3000 thread lances
^C
$
```

Avec le **detach** “**pas de limite**”

Synchronisations entre threads : mutex

- Un mutex est un objet d'exclusion mutuelle (**MUT**ual **EX**clusion), et est très pratique pour protéger des données partagées de modifications concurrentes et pour implémenter des sections critiques

- Deux états :
  - **Libre** (n'est pris par aucun thread) ou
  - **Verrouillé** (pris par un thread)

- Un thread qui verrouille un mutex tient le mutex

- Un seul thread peut tenir un mutex donné à la fois

- En général variables globales ou locales statiques.

Initialisation statique :

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Synchronisations entre threads : mutex

Initialisation Dynamique :

int pthread_mutex_init mutex(pthread_mutex_t * mutex, const pthread_mutexattr_t *attributs);

mutex : Un pointeur vers la variable de type **pthread_mutex_t** à initialiser.

attr : Un pointeur vers un objet **pthread_mutexattr_t** qui permet de spécifier des attributs particuliers pour le mutex (ou **NULL** pour utiliser les valeurs par défaut).

Libération :

int pthread_mutex_destroy(pthread_mutex_t *): détruit un mutex, libérant les ressources qu'il détient. Le mutex doit être déverrouillé.

Synchronisations entre threads : mutex

verrouillage :

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

- Si **mutex libre**, verrouillé et attribué au thread appelant
- Sinon blocage jusqu'à libération, puis verrouillé et attribué au thread appelant

Déverrouillage :

```
int pthread_mutex_unlock(pthread_mutex_t * mutex)
```

- Déverrouille le mutex

Remarque

Il faut éviter de verrouiller deux fois un même mutex dans le même thread sans le déverrouiller entre temps. Il y a un risque de blocage définitif du thread.

Synchronisations entre threads : mutex

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

static void * routine_threads (void * argument);
static int    aleatoire (int maximum);

pthread_mutex_t   mutex_stdout = PTHREAD_MUTEX_INITIALIZER;

int main (void)
{
int    i;
pthread_t  thread;

for (i = 0; i < 5; i ++)
 pthread_create(& thread, NULL, routine_threads, (void *) i);
 pthread_exit(NULL);
}
```

Synchronisations entre threads : mutex

```
static void * routine_threads (void * argument) {
 int numero = (int) argument;
 int nombre_iterations;
 int i;
 nombre_iterations = 1 + aleatoire(3);
 for (i = 0; i < nombre_iterations; i ++) {
  sleep(aleatoire(3));
  pthread_mutex_lock(& mutex_stdout);
  fprintf(stdout, "Le thread %d a obtenu le mutex\n", numero);
  sleep(aleatoire(3));
  fprintf(stdout, "Le thread %d relache le mutex\n", numero);
  pthread_mutex_unlock(& mutex_stdout);
 }
return NULL;
}

static int aleatoire (int maximum) {
double d;
d = (double) maximum * rand();
d = d / (RAND_MAX + 1.0);
return ((int) d);
}
```

Synchronisations entre threads : mutex

```
$ ./exemple-pthread-mutex-1
Le thread 0 a obtenu le mutex
Le thread 0 relache le mutex
Le thread 4 a obtenu le mutex
Le thread 4 relache le mutex
Le thread 1 a obtenu le mutex
Le thread 1 relache le mutex
Le thread 2 a obtenu le mutex
Le thread 2 relache le mutex
Le thread 3 a obtenu le mutex
Le thread 3 relache le mutex
Le thread 3 a obtenu le mutex
Le thread 3 relache le mutex
...
$
```

Synchronisations entre threads: sémaphores

<semaphore.h>

```
int sem_init (sem_t *semaphore, int partage, int valeur)

int sem_wait(sem_t *sem_id);

int sem_post(sem_t *sem_id);
```