Chapitre 12

Les processus légers ou threads

Un thread est une séquence (ou fil) d'exécution du code d'un programme au sein d'un processus. Un processus classique (lourd) ne contient qu'un seul thread. Un processus multithreadé peut contenir plusieurs flots d'exécutions simultanés.

Idée : partager les différentes ressources d'un processus par plusieurs flots d'exécution.

Référence : `http://www.llnl.gov/computing/tutorials/pthreads/`

12.1 Présentation

12.1.1 Principe

Un thread n'existe qu'au sein d'un processus lourd. Ses ressources sont partagées par les différents threads qu'il contient :

- code
- mémoire
- fichiers
- droits Unix
- environnement shell, répertoire de travail
- ...

Chaque thread possède :

- sa propre pile d'exécution
- un identificateur de thread
- un pointeur d'instruction

Cela est évidemment nécessaire pour que chaque thread ait un flot d'exécution distinct.

Un processus (Unix) peut contenir plusieurs centaines de threads.

12.1.2 Cycle de vie d'un thread

Au lancement d'un processus, un seul thread est créé (celui du processus initial). La création de chaque nouveau thread est paramétrée par une fonction à exécuter, avec éventuellement des paramètres. Le thread se termine soit explicitement, soit en étant tué par un autre thread ou par le système.

Les différents threads d'un processus se partagent le temps alloué au processus par le SE. On peut ajuster les priorité des threads au besoin.

12.1.3 Critiques des threads

Avantages

Le partage des ressources entre les threads est automatique, il évite donc des mécanismes sophistiqués de communication entre processus (mémoire partagée, tubes, ...)

Le temps de commutation entre deux threads d'un même processus est peu coûteux, car il consiste simplement à changer de flot d'exécution.

La vitesse d'exécution peut être améliorée. Supposons que l'on exécute des requêtes répondant avec un certain délai. Avec un seul thread, une attente est nécessaire à chaque requête au système. Si plusieurs threads font chacun un requête, attentes se font en parallèle.

Multiprocesseurs . Sur un SE adapté, les threads peuvent être exécutés en parallèle ... sous condition

La réactivité d'interfaces utilisateurs peut être améliorée. Un thread pour réafficher l'interface graphique (raffraichissement des images, affichage d'un curseur d'avancement), un autre thread ayant lancé un calcul lourd. L'application reste réactive.

La réactivité d'un serveur peut être améliorée. Avec un seul thread, un serveur doit gérer plusieurs requêtes : répondre aux requêtes, gérer les priorités de requêtes. Sur un serveur multithreadé, on crée un thread par requête. Le réglage de priorité (ordonnancement des threads) peut être laissé au système.

L'écriture de programmes peut être facilitée. Prenons l'exemple simplifié d'un tableur.

Pseudo code avec un thread :

```
while (1) {
  while (maj_necessaire() && ! entree_utilisateur_disponible()) {
    recalculer_une_formule();
  }
  cmd = get_entree_utilisateur();
  traiter_entree_utilisateur(cmd);
}
```

Avec un code à deux threads :

```
// thread clavier
while (1) {
  cmd = get_entree_utilisateur();
  traiter_entree_utilisateur(cmd);
}

//thread calcul
while (1) {
  attendre_un_changement();
  recalculer_toutes_les_formules();
}
```

L'expression est plus naturelle. La raison principale est ici qu'on peut bloquer chaque thread sur un événement, et le traiter quand il se produit. Chaque thread a également une tâche bien précise, et le code est mieux découpé.

Inconvénients

La concurrence entre les threads doit être correctement gérée. Risque de bogues occasionels, d'étreinte fatales (dead-lock), ... Il faut analyser finement la concurrence des threads.

La pile d'exécution peut être difficile à régler. Comme chaque thread en possède une, le réglage de sa taille dépend de l'activité du thread (appels récursifs profonds, variables locales nombreuses, ...).

Réentrance : propriété pour une fonction d'être lancée simultanément par plusieurs tâches. Si plusieurs threads appellent une même fonction non prévue pour cela, le résultat de la fonction risque d'être erroné ! Le problème se produit avec les fonctions ayant des effets de bords, utilisant des variables statiques.

Une application multi-threadée ne peut donc pas toujours appeler une librairie non prévue pour cela, sans prendre de précaution (protéger les appels par des verrous)

12.1.4 Quand choisir les threads

Il est intéressant d'utiliser les threads si un processus
- se bloque pour des délais potentiellement longs (temps perdu)
- résout des tâches distinctes en parallèle
- doit répondre à des événements asynchrones (saisie clavier, répétition d'un tâche à intervalle régulier, ...)
-

12.2 La librairie pthread

La librairie `pthread` satisfait la norme POSIX 1.c.

12.2.1 Création d'un thread

```
#include <pthread.h>

int  pthread_create(pthread_t  *  thread, pthread_attr_t * attr, void *
                    (*start_routine)(void *), void * arg);
```

- crée un thread qui exécute immédiatement la fonction `start_routine` en lui passant la valeur de `arg`

– `attr` permet de modifier le comportement du thread. Si `attr` vaut `NULL`, des paramètres par défaut sont utilisés (le thread est joignable, priorité non temps-réel, ...)
– renvoie un entier non nul si la création a échoué (nombre maximum de threads `PTHREAD_THREADS_MAX` déjà atteint, pas assez de ressources ...).
– renvoie 0 si ok. `*thread` est alors modifié (sert à identifier le thread).

La variable `arg` doit pointer sur une zone sûre (qui restera accessible, et dont la valeur ne risque pas de changer). Solution simple : passer une zone allouée par `malloc` au thread, et la `libérer` dans `start_routine`.

Un thread peut connaître son propre identifiant avec :

```
#include <pthread.h>

pthread_t pthread_self(void);
```

12.2.2 Terminaison d'un thread

Un thread peut se terminer explicitement en utilisant :

```
#include <pthread.h>

void pthread_exit(void *value_ptr);
```

– `value_ptr` est la valeur de retour du thread
– l'appel à `pthread_exit` peut se faire en dehors de la fonction utilisée à la création du thread.

Un thread se termine également lorsque la fonction principale (`start_routine`) du thread se termine : la valeur de retour du thread est celle de `start_routine`.

Attention, le pointeur passée comme valeur de retour ne doit pas pointer sur une variable locale de la fonction du thread !

12.2.3 Attente de la fin d'un thread

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **value_ptr);
```

– attend que le thread `thread` se termine
– quand le thread est terminé, `*value_ptr` reçoit la valeur de retour du thread.
– renvoie 0 si ok, un nombre non nul indiquant la cause de l'erreur sinon (thread incorrect, dead lock détecté, ...)
– récupère

Tant que le thread n'a pas été (re)joint, il reste comptabilisé dans la table des threads (comme pour les processus zombis).

12.2.4 Exemple

threads/exemples/bonjour.c

```
/*************************************************************************
 *bonjour.c
 *
 *  (François lemaire) <lemaire@lifl.fr>
 * Time-stamp: <2007-03-14 16:57:19 lemaire>
 *************************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <pthread.h>
#include <unistd.h>

void* bonjour(void *arg) {
  int i;
  i = *((int *)arg);
  free(arg);
  printf("Bonjour %d!\n", i);

  pthread_exit(NULL);
}

int main(int argc, char *argv[]) {

  int i;
  int *arg;
  pthread_t ths[5];

  for (i=0;i<5;i++) {
    arg = (int*)malloc(sizeof(int));
    *arg = i;
    if (pthread_create(&ths[i], NULL, bonjour, (void*)arg)) {
```

```
      fprintf(stderr, "Erreur de création");
      exit(EXIT_FAILURE);
    }
  }

  for (i=0;i<5;i++) {
    pthread_join(ths[i], NULL);
  }


  exit(EXIT_SUCCESS);
}
```

Se compile avec :

```
$ gcc -Wall -D_REENTRANT -o bonjour bonjour.c -lpthread
$ ./bonjour
Bonjour 0!
Bonjour 1!
Bonjour 2!
Bonjour 3!
Bonjour 4!
$
```

12.3 Verrous

La librairie `pthread` fournit des verrous (type `pthread_mutex_t`) pour les threads.

Initialisation Se fait avec :

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Voir aussi `pthread_mutex_init` pour plus de flexibilité.

Verrouillage Se fait avec :

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

L'appel est bloquant jusqu'à ce que `mutex` puisse être verrouillé. Le thread appelant devient propriétaire du verrou.

Déverrouillage Se fait avec :

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

Seul le propriétaire peut déverrouiller `mutex`.

Destruction Se fait avec :

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

Libère `mutex`. Le verrou doit être déverrouillé.

Les trois fonctions renvoient 0 si ok, et un code d'erreur sinon.

threads/exemples/verrou.c

```
/*****************************************************************
 *bonjour.c
 *
 *  (François lemaire) <lemaire@lifl.fr>
 * Time-stamp: <2007-03-14 16:57:19 lemaire>
 *****************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <pthread.h>
#include <unistd.h>

int valeur;
pthread_mutex_t V = PTHREAD_MUTEX_INITIALIZER;

void inc(int *v) {
  int res;
  res = *v;
  res = res + 1;
  sleep(1);
  *v = res;
}

void* bonjour(void *arg) {
  int i;
  pthread_mutex_lock(&V);
```

```
  inc(&valeur);
  printf("%d\n", valeur);
  pthread_mutex_unlock(&V);
  pthread_exit(NULL);
}

int main(int argc, char *argv[]) {

  int i;
  pthread_t ths[5];

  for (i=0;i<5;i++) {
    if (pthread_create(&ths[i], NULL, bonjour, NULL)) {
      fprintf(stderr, "Erreur de création");
      exit(EXIT_FAILURE);
    }
  }

  for (i=0;i<5;i++) {
    pthread_join(ths[i], NULL);
  }

  pthread_mutex_destroy(&V);
  exit(EXIT_SUCCESS);
}

/*
$ ./verrou
1
2
3
4
5

Sans le lock/unlock

$ ./verrou
1
1
1
1
1

*/
```

12.4 Sémaphores

La librairie `pthread` fournit aussi des sémaphores faciles d'emploi (les sémaphores System V sont plus ardus). C'est le type : `(sem_t`. Contrairement aux verrous, il n'y a pas de notion de propriétaire d'un sémaphore.

Initialisation Se fait avec

```
#include <semaphore.h>

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

- si `pshared` vaut 0, `sem` est supposé être une variable partagée entre les threads (variable globale, variable allouée dans le tas, ...). sinon, `sem` doit être située en mémoire partagée.
- `value` est la valeur initiale.

Puis-je ? se fait avec

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

Vas-y ! se fait avec

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

Destruction se fait avec

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

Les quatre fonctions précédentes renvoient 0 si ok, et -1 sinon. Attention, la fonction `sem_wait()` peut-être interrompue par un signal, elle renvoie -1 dans ce cas. Parade : appeler `sem_wait` dans une boucle.

threads/exemples/sem_aff_un_deux.c

```
/*************************************************************************
 *sem_test.c
 *
 *  (François lemaire) <lemaire@lifl.fr>
 * Time-stamp: <2007-03-14 16:42:32 lemaire>
 *************************************************************************/
```

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <semaphore.h>
#include <pthread.h>

sem_t S;

void* mes1(void* unused) {
  sem_wait(&S);
  printf("Et de deux !\n");
}


void* mes2(void* unused) {
  sleep(1);
  printf("Et de un, ...\n");
  sem_post(&S);
}

int main(int argc, char *argv[]) {

  pthread_t th1, th2;

  if (sem_init(&S, 0, 0)) {
    perror("sem_init");
    exit(EXIT_FAILURE);
  }

  if (pthread_create(&th1, NULL, mes1, NULL)) {
    fprintf(stderr,"thread 1");
    exit(EXIT_FAILURE);
  }

  if (pthread_create(&th2, NULL, mes2, NULL)) {
    fprintf(stderr,"thread 2");
    exit(EXIT_FAILURE);
  }


  pthread_join(th1, NULL);
  pthread_join(th2, NULL);
  sem_destroy(&S);

  printf("Fini !\n");
  exit(EXIT_SUCCESS);
}
```

12.5 Considérations pratiques

En pratique, deux erreurs assez surprenantes et inattendues peuvent se produire. La première est liée à la manière dont les variables sont modifiées en mémoire au niveau du processeur. La seconde est lié aux optimisations du compilateur.

12.5.1 Les variables atomiques

Lorsque que deux variables sont rangées côté à côte en mémoire, il se peut que la manipulation d'une variable ait un effet de bord sur l'autre. Plus précisément, les lectures et écritures de variables ne sont pas atomiques. Cela est lié au phénomène suivant :

```c
  unsigned char a,b;

  /* supposons a=0, b=0 */
  a=1;
```

L'affectation `a=1` risque de se faire de la manière suivante :
- chargement depuis la mémoire dans un registre du processeur des 4 octets contenant les variables `a` et `b`
- mise à 1 de la valeur de `a` dans le registre
- recopie en mémoire du registre, qui écrase l'ancienne valeur de `a` et qui remet 0 dans la variable `b`

Ainsi, les deux variables ne sont pas indépendantes. Par conséquent, si un thread ne manipule que `a`, et si un autre ne manipule que `b`, des conflits peuvent quand même se produire, même si `a` et `b` sont utilisées de manière indépendantes.

La solution consiste soit :
- soit à protéger l'accès aux variables visibles par les différents threads par un verrou ;
- soit à utiliser le mot clé `atomic_t` lors de la déclaration des variables (plus de détail dans `atomic.h`.

12.5.2 Les variables volatiles

Le mot clé `volatile` permet d'indiquer au compilateur que le contenu d'une variable risque de changer de manière intempestive.

```
int i=0;
while (i!=0);
```

Si on compile le code précédent avec optimisation (`-O2` par exemple), la variable `i` sera supprimée et le code deviendra quelque chose ressemblant à

```
while (1);
```

Si la variable `i` peut être modifiée de manière externe, cette optimisation est incorrecte.

Si on déclare une variable avec mot-clé `volatile`, le compilateur considère que le contenu de la variable peut changer à tout moment, et ne fait donc plus de suppositions sur sa valeur.

Ce mot clé est utile pour les variables

- en mémoire partagée (variables d'un controleur, ...)
- situées entre `setjmp` et `longjmp`
- partagées entre threads
- ...