PS 03 Programmation Système (POSIX threads)

A faire TD 5 et TD 6.

Table des matières

1 Introduction

1.1 Processus lourds / légers

processus “lourd”

- espaces mémoires séparés (copie)
- dédoublement lors du `fork()`

processus léger

- fonction lancée “en parallèle”
- partage le **même espace mémoire**
- En C : API POSIX <thread.h>
- En C++11 : contrôlé par un objet `std::thread`
- En C11 : <c11threads.h>

1.2 Exemple 1 : Hello World

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

void* say_hello(void* data)
{
    char *str;
    str = (char*)data;
    while(1)
    {
        printf("%s\n",str);
        sleep(1);
    }
}

int main()
{
    pthread_t t1,t2;

    pthread_create(&t1, NULL, say_hello, "hello from thread 1");
    pthread_create(&t2, NULL, say_hello, "hello from thread 2");
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
}
```

1.3 Détails sur `pthread_create()`

- `pthread_t` est un entier - le thread ID (TID). Il est similaire à `pid_t` pour les processus.
- la routine `somme_thread` est une fonction qui **prend un** `void *` **et retourne un** `void *`. En pratique, on va passer et récupérer un pointeur sur une structure quelconque.
- `pthread_create()` prends 4 arguments qui sont tous des pointeurs.

```c
int pthread_create(
      pthread_t *pTID,             /* TID ptr. */
      const pthread_attr_t *pAttr, /* thread attribute ptr. */
      void *(*pRoutine) (void *),  /* thread routine function ptr. */
      void *pArg                   /* argument ptr. */
    );
```

  - Le premier argument `pTID` pointe sur la variable qui va stocker le TID,

  - Le second argument `pAttr` pointe sur une structure décrivant les attributs du nouveau thread, comme sa priorité par exemple. Il peut être `NULL`.
  - Le troisième argument `pRoutine` pointe sur la fonction que devra exécuté le thread, cette fonction peut bien sûr en appeler d'autres.
  - Le quatrième argument `pArg` pointe sur l'argument qui sera passé à la fonction (`pRoutine`). Il peut être `NULL`.
- Cette fonction retourne `0` en cas de succès; en cas d'erreur, elle retourne son code, et dans ce cas, `pTID` est indéfini.

1.4 Détails sur `pthread_join()`

- Si le thread principal, c.a.d. le thread du processus, termine, l'espace virtuel complet sera libéré, tuant les threads qui ont été créés. Il est donc nécessaire d'**attendre la terminaison** des threads actifs avant de laisser terminer le thread principal.

```
int pthread_join(pthread_t TID, void **ppRetval);
```

  - Le premier argument `TID` est l'ID du thread attendu. (note : ce n'est pas un pointeur).
  - Le second argument `ppRetval` est un pointeur sur le pointeur de la valeur de retour du thread.
- Si le thread attendu a déjà terminé, alors `pthread_join()` retourne immédiatement; sinon la fonction **bloque et attend**.
- En cas de succès, elle retourne `0`; en cas d'erreur, elle retourne son code, qui peut être :
  - `EINVAL`: un autre thread attends déjà sur ce thread ID, où le thread attendu n'est pas joinable.
  - `ESRCH`: aucun thread avec ce TID n'a pu être trouvé.
  - `EDEADLK`: un 'deadlock' a été détecté (c.a.d., deux threads essaient de s'attendre réciproquement), ou le `TID` indique le thread qui a fait l'appel lui-même.

1.5 Remarques

- Les threads ne sont pas obligés de débuter/terminer ensemble ou dans un ordre prédictible.
- Une fois qu'un thread est terminé, `pthread_join()` rejoint ce thread avec le thread appelant (dans l'exemple, le thread principal).
- Le thread spécifié par `TID` doit être **joinable**, c.a.d. non détaché. > Note : Quand un thread détaché termine, ses ressources sont automatiquement désallouées. `int pthread_detach(pthread_t TID);` permet de détacher un thread.
- Un thread peut faire explicitement appel à :
  - `pthread_t pthread_self()` afin de récupérer son thread ID,
  - `pthread_exit()` pour forcer la terminaison de son exécution.

1.6 Un exemple avec 2 threads

1.7 Exemple 2 : somme des éléments d'un tableau

```
**
 * Somme d'une tranche d'un tableau
 * @param tableau   le tableau
 * @param debut     indice de debut (compris)
 * @param fin       indice de fin (exclus)
 * @return          tableau[debut]+tableau[debut+1]+... tableau[fin-1]}
 */

float somme_tranche(float tableau[], int debut, int fin) {
    float somme = 0.;
    for (int i = debut; i < fin; i++) {
        somme += tableau[i];
    }
```

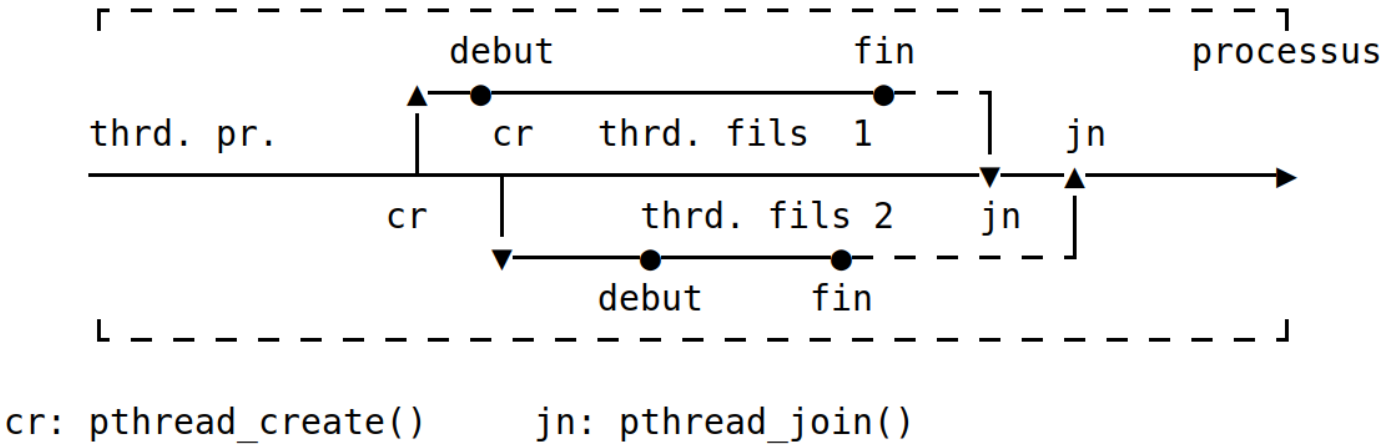

Figure 1 – thread

```
    return somme;
}
```

1.8 Calcul sur des “tranches” séparées pour chaque thread

```
struct arguments_somme_tranche {
    float * tableau;
    int debut, fin;
    float * adr_somme;
};

void * thread_somme_tranche(void * args) {
    struct arguments_somme_tranche *a = args;
    *(a->adr_somme) = somme_tranche(a->tableau, a->debut, a->fin);
    return NULL;
}
```

1.9 Vue d’ensemble

```
float *tableau = malloc(TAILLE_TABLEAU * sizeof (float));
remplir_tableau(tableau, TAILLE_TABLEAU);

float somme1, somme2;
struct arguments_somme_tranche
args_t1 = {
  .tableau = tableau,
  .debut = 0,
  .fin = TAILLE_TABLEAU / 2,
  .adr_somme = &somme1
},
args_t2 = {
  .tableau = tableau,
  .debut = TAILLE_TABLEAU / 2,
  .fin = TAILLE_TABLEAU,
  .adr_somme = &somme2
};
pthread_t t1, t2;
```

```
pthread_create(& t1, NULL, thread_somme_tranche, &args_t1);
pthread_create(& t2, NULL, thread_somme_tranche, &args_t2);

pthread_join(t1, NULL);
pthread_join(t2, NULL);

printf("la somme vaut = %d + %d = %d \n", somme1, somme2, somme1 + somme2);
```

1.10 Exercice : Est-ce rentable ?

- remplir un tableau de 10 millions de flottants
- calculer la somme de différentes façons : appel direct, 1 thread, 2 threads, 4 threads, ...
- **mesurer** les temps d'exécution (`gettimeofday`)

1.11 Exemple d'utilisation de `gettimeofday()`

```
#include <sys/time.h>
#include <time.h>

struct timeval tv1,tv2;

// noter les temps de début et de fin
gettimeofday(&tv1, NULL);
calcul();
gettimeofday(&tv2,NULL);

// calculer la durée en microsecondes
#define TIME_DIFF(t1, t2) \
        ((t2.tv_sec - t1.tv_sec) * 1000000 + (t2.tv_usec - t1.tv_usec))

// afficher sa valeur en millisecondes
printf("durée du calcul : %f ms\n", ((float)TIME_DIFF(tv1,tv2)/1000));
```

2 Synchronisation : sections critiques et verrous

2.1 Problème constaté (variable accumulation partagée en écriture)

```
void somme_tranche(float tableau[], int debut, int fin
                   float *s) // retour par reference
) {
    for (int i = debut; i < fin; i++) {
        *s += tableau[i];
    }
}

void * thread_somme_tranche(void * args) {
{
  struct arguments_somme_tranche *a = args;
  somme_tranche(a->tableau, a->debut, a->fin, a->adr_somme);
  return NULL;
}
```

2.2 Vue d'ensemble

```
float somme = 0.;              <--- variable partagée en écriture
struct arguments_somme_tranche
```

```
args_t1 = {
  .tableau = tableau,
  .debut = 0,
  .fin = TAILLE_TABLEAU / 2,
  .adr_somme = &somme          <---
},
args_t2 = {
  .tableau = tableau,
  .debut = TAILLE_TABLEAU / 2,
  .fin = TAILLE_TABLEAU,
  .adr_somme = &somme          <---
};
pthread_t t1, t2;

pthread_create(& t1, NULL, thread_somme_tranche, &args_t1);
pthread_create(& t2, NULL, thread_somme_tranche, &args_t2);
```

2.3 Symptôme : les écritures se mélangent parfois

— **cause** : une écriture peut commencer alors qu'une autre est en cours
— **solution** : rendre l'écriture **atomique**
— **moyen** : en faire une **section critique**

2.4 Comment corriger ?

Un *verrou* pour encadrer l'accès à une variable partagée (en écriture)

— de type `mutex`
— avec des opérations `lock()` et `unlock()`

Remarque: pour une variable **partagée en lecture**, si la variable est modifiée par un thread, elle doit être déclarée `volatile`.

2.5 Principe

Un verrou peut être

— **détenu** par un thread (un seul à la fois), qui a fait `lock()`
— ou **libre**.

```
pthread_mutex_t v;
pthread_mutex_init(&v, NULL);
```

	libre	détenu
`pthread_mutex_lock(&v);`	prend possession	attend libération
`pthread_mutex_unlock(&v);`	*indéfini*	libère

2.6 usage habituel : section critique

— Le code délimité par lock/unlock forme une **section critique**

```
pthread_mutex_lock(&v);                |  pthread_mutex_lock(&v);
// Section 1                           |  // Section 2
pthread_mutex_unlock(&v);              |  pthread_mutex_unlock(&v);
```

Invariant : il y a **au plus un** thread dans une des sections critiques contrôlées par le même verrou.

2.6.1 Solution 1

```
void somme_tranche(float tableau[], int debut, int fin
                   float *s) // retour par reference
{
  for (int i = debut; i < fin; i++) {
    pthread_mutex_lock(&mutex); // section critique
    *s += tableau[i];
    pthread_mutex_unlock(&mutex);
  }
}
```

2.6.2 Solution 2

```
void somme_tranche(float tableau[], int debut, int fin
                   float *s) // retour par reference
{
  float tmp = 0.;
  for (int i = debut; i < fin; i++) {
    tmp += tableau[i];
  }
  pthread_mutex_lock(&mutex); // section critique
  *s += tmp;
  pthread_mutex_unlock(&mutex);
}
```