UNIVERSITÉ PARIS SUD
IUT D'ORSAY

DUT Informatique
M3101 Système S3
2015 / 2016

Travaux Dirigés n° 4 : Threads

Objectifs : apprendre à créer, travailler avec et arrêter des threads (ou processus légers). Savoir reconnaître les données partagées entre différents threads. Être capable d'orchestrer la synchronisation de threads au moyen des primitives de terminaison et d'exclusion mutuelle.

1 Threads

Les *threads*, qu'on appelle aussi *processus légers*, sont des unités d'exécution qui opèrent dans le contexte d'un processus. Un processus peut contenir plusieurs threads qui exécutent tous le même programme et partagent la même mémoire. Plus précisément, les segments de code (le programme), de données (les variables globales) et le tas (les allocations dynamiques) sont partagés entre les différentes threads d'un même processus. Chaque thread dispose de sa propre pile, ce qui leur permet de poursuivre les chemins d'exécution différents, mais comme ces piles habitent la mémoire commune, une variable locale dans la pile d'un thread peut être lue et modifiée par un autre thread s'il en connaît l'adresse.

La mémoire partagée permet à des threads d'un processus d'échanger des données sans recourir à des outils de communication entre processus tels que tubes ou sockets. La création d'un nouveau thread est aussi moins couteuse que la création d'un nouveau processus. En effet, même si le noyau ne duplique pas le contenu de mémoire d'un processus au moment de `fork()`, il doit tout de même faire une copie de sa table de pages. Pour la même raison, le changement de contexte entre deux threads d'un processus est plus rapide qu'entre deux processus.

Cependant, la programmation avec des threads est une affaire délicate : l'accès simultané aux variables partagées peut amener à des erreurs d'*interférence* qui sont à la fois

- difficiles à détecter, car l'apparition ou non de l'erreur dépend de décisions d'ordonnancement ainsi que de détails d'architecture dans le cas d'une machine multiprocesseur ;
- difficiles à analyser, car les exécutions consécutives d'un même programme ne se comportent pas de la même façon, et d'ailleurs le débogage même — l'addition du code de debug ou l'utilisation d'un débogueur — peut empêcher à l'erreur de se manifester (ce cauchemar de programmeur à reçu le nom de *Heisenbug*) ;
- difficiles à corriger, car la gestion correcte d'accès aux données partagées peut nécessiter des modifications majeures dans le code du programme, jusqu'à la réimplémentation complète des parties concernées.

Ainsi, la conception et le développement d'une application multi-threadée exige que le programmeur soit parfaitement conscient des problèmes liés au travail en mémoire partagée et à l'aise avec les solutions telles que l'exclusion mutuelle, les moniteurs, les structures de données adaptées, etc.

Dans ce cours nous allons utiliser les threads POSIX (appelés *pthreads*) et leur implémentation actuelle dans Linux.

2 Création et identification de threads

```
#include <pthread.h>
int pthread_create(
  pthread_t* thread_id            /* pointeur sur l'identifiant du thread */
  pthread_attr_t* attr,           /* NULL: attributs par défaut */
  void* (*routine) (void* arg),   /* fonction exécutée par le thread */
  void* arg                       /* paramètre de `routine' */);
```

La primitive `pthread_create()` crée un nouveau thread et écrit son identifiant dans `thread_id` (il s'agît d'une valeur opaque qui servira par la suite à la gestion du thread). Son argument `attr` définit les attributs du thread : nous utiliserons toujours NULL, qui donne les attributs par défaut. L'argument `routine` est un pointeur sur la fonction qui sera exécutée par le thread. Cette fonction, qu'on appelle *routine de départ* (*start routine*), retourne un `void*` et prend un unique argument de type `void*`. Enfin, le dernier argument `arg` correspond à l'argument transmis à la fonction `routine`. Cette primitive renvoie 0 en cas de succès, un code d'erreur sinon.

Le nouveau thread a le même PID et PPID que le thread créateur. Il partage également ses permissions (UID, GID, EUID, EGID), ses fichiers ouverts, ses gestionnaires de signaux (mais non pas la liste de signaux bloqués), son répertoire courant, ses variables d'environnement et d'autres propriétés (voir `man 7 pthreads` pour la liste complète).

Chaque thread a un identifiant unique de type `pthread_t`. Pour obtenir l'identifiant d'un thread ou comparer deux identifiants on utilise les primitives suivantes :

```
#include <pthread.h>
pthread_t pthread_self(void);
      int pthread_equal(pthread_t tid1, pthread_t tid2);
```

La primitive `pthread_equal()` renvoie une valeur non-nulle si `tid1` et `tid2` sont égaux et 0 sinon.

3 Terminaison de threads

Tous les threads d'un processus prennent fin si un des threads appelle `exit()` ou `_exit()`, ou bien si le thread initial (celui qui existait seul dans le processus avant tout appel à `pthread_create()`) retourne de la fonction `main()`. Un thread seul prend fin automatiquement quand sa routine de départ (la fonction passée en argument de `pthread_create()`) retourne ou quand le thread appelle la primitive `pthread_exit()` :

```
#include <pthread.h>
void pthread_exit(void* retval);
```

qui prend en argument la valeur de retour du thread. La primitive `pthread_exit()` ne peut jamais retourner dans le thread appelant. La valeur de retour peut être un entier converti en `void*` : par exemple, on peut renvoyer `((void*) 1)` de la routine de départ.

Tout thread est par défaut considéré *joignable*. Cela veut dire qu'à la terminaison de ce thread ses ressources ne sont pas libérées avant qu'un autre thread n'appele la primitive `pthread_join()` :

```
#include <pthread.h>
int pthread_join(pthread_t thread_id, void** retval);
```

La primitive suspend l'exécution du thread appelant jusqu'à la fin du thread d'identifiant `thread_id`. Si le thread d'identifiant `thread_id` est déjà terminé, elle retourne immédiatement. En cas de succès,

elle renvoie 0 et place à l'adresse `*retval` la valeur de retour du thread attendu. En cas d'échec, elle renvoie un code d'erreur. La primitive `pthread_join()` joue ainsi le même rôle pour les threads que la primitive `waitpid()` pour les processus.

Il est possible de déclarer un thread non-joignable, ou *détaché*, en utilisant `pthread_detach()` :

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

Cette primitive indique au noyau qu'il pourra récupérer les ressources allouées au thread `thread_id` lorsqu'il terminera (immédiatement s'il est déjà terminé). Un appel de `pthread_join()` sur un thread détaché ou déjà attendu par un autre thread renvoie le code d'erreur `EINVAL`.

4 Exclusion mutuelle

Un outil principal de synchronisation entre threads est l'objet d'exclusion mutuelle, ou *mutex*. Les mutex permettent d'assurer que plusieurs threads ne peuvent pas accéder aux valeurs partagées en même temps.

À tout moment donné, un mutex est soit pris par aucun thread (on dit *déverrouillé*), soit pris par un et un seul thread (on dit *verrouillé*). Tout thread qui essaie de prendre un mutex déjà verrouillé doit attendre jusqu'à ce que le mutex se libère.

On manipule les mutex à l'aide des quatre primitives fondamentales suivantes qui retournent toutes 0 en cas de succès et un code d'erreur sinon :

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex, NULL);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

La primitive `pthread_mutex_init()` initialise un mutex dont on aura passé l'adresse en premier argument. Le second argument indique les attributs du mutex, on indiquera `NULL` pour avoir les attributs par défaut qui initialisent le mutex en état déverrouillé.

La primitive `pthread_mutex_lock()` verrouille le mutex passé en argument. Si le mutex est déverrouillé, il devient verrouillé et `pthread_mutex_lock()` retourne immédiatement. Si le mutex est déjà pris par un autre thread, le thread appelant est suspendu jusqu'à ce que le mutex soit déverrouillé. Si le mutex est déjà pris par le thread appelant, le comportement de `pthread_mutex_lock()` dépend des attributs du mutex. Par défaut, le thread appelant est bloqué indéfiniment.

La primitive `pthread_mutex_unlock()` déverrouille le mutex passé en argument, qui doit être verrouillé par le thread appelant au moment de l'appel. Si en ce moment il y a d'autres threads qui essaient de prendre le mutex avec `pthread_mutex_lock()`, un de ces threads verrouillera le mutex et continuera l'exécution.

La primitive `pthread_mutex_destroy()` détruit le mutex passé en argument, qui doit être déverrouillé au moment de l'appel. Il est possible de réinitialiser un mutex détruit avec un nouvel appel à `pthread_mutex_init()`.

5 Exercices

Exercice 1 : Partage des données et terminaison. On considère le code suivant où plusieurs threads sont crées, chacun ayant pour unique travail d'afficher son identifiant :

```
#define NB_THREADS 3
pthread_t tid[NB_THREADS];
int thread_execute = 0;

/* Fonction executée par les threads. Le type de retour
   et l'argument sont de type void*, ce qui nécessite
   des conversions de type. */
void * routine(void * i) {
  int n = *((int *) i);
  // aujourd'hui sous Linux les identifiants sont numériques
  printf("Thread numéro %d, ID %lu\n", n, pthread_self());
  thread_execute = 1;
  return NULL;
}

int main() {
  int i;
  for (i = 0; i < NB_THREADS; i++) {
    if (pthread_create(&tid[i], NULL, routine, (void *) &i) != 0)
    { fprintf(stderr, "Erreur création thread numéro %d.\n", i); exit(1); }
  }
  printf("Thread initial d'ID %lu\n", pthread_self());
  if (thread_execute)
    printf("Des threads annexes ont été exécutés.\n");
  else
    printf("Aucun thread annexe n'a été exécuté.\n");
  return 0;
}
```

Combien au maximum, avec ce programme, y a-t-il de threads s'exécutant en parallèle (ou en concurrence s'il n'y a pas assez de ressources) ?

Énumérez toutes les variables et dites par quels threads elles sont directement utilisables.

Comment un thread pourrait lire ou modifier la variable `n` d'un autre thread ?

Expliquez l'exécution suivante où le numéro de chaque thread est le même. Proposez une solution.

```
Thread numéro 3, ID 3084860304
Thread numéro 3, ID 3076467600
Thread numéro 3, ID 3068074896
Thread initial d'ID 3084863168
Des threads annexes ont été exécutés.
```

Expliquez l'exécution suivante où aucun thread n'a réalisé son affichage. Proposez une solution.

```
Thread initial d'ID 3084601024
Aucun thread annexe n'a été exécuté.
```

Exercice 2 : Parallélisation de la multiplication matrice × vecteur. L'opération de multiplication d'une matrice par un vecteur est l'une des plus utiles en informatique (calcul scientifique, infographie, etc.). Il est très intéressant de la paralléliser pour en améliorer la performance. Elle est réalisée simplement par les deux boucles montrées en Figure 1 où on voit comment le vecteur résultat y est calculé à partir de la matrice A et du vecteur x. Plus précisément, on voit que le i-ième élément du vecteur y est calculé à partir seulement de la i-ième ligne de la matrice A et de tout le vecteur x. Puisque A et x restent constants, on peut calculer chaque élément du vecteur y indépendamment les uns des autres.

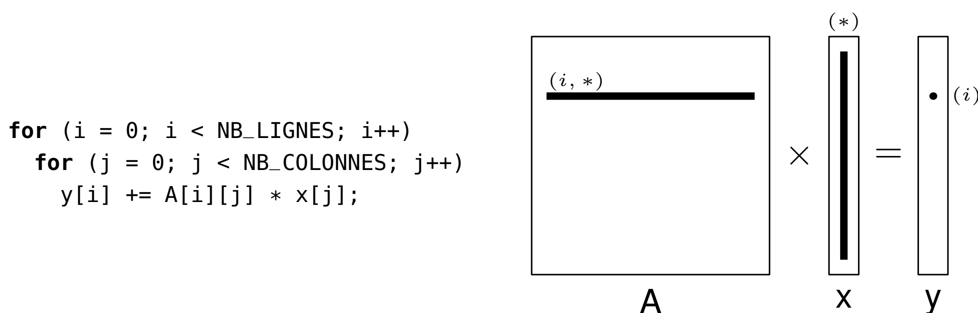

FIGURE 1 – Noyau de la multiplication matrice × vecteur

Proposez un programme utilisant les threads pour le calcul du vecteur y tel que chaque élément de ce vecteur soit calculé en parallèle par rapport aux autres.

Exercice 3 : Synchronisation de threads. Le but de cet exercice est d'écrire un programme dans lequel le thread initial et un thread annexe, chacun de leur côté, incrémentent une variable partagée initialisée à 0. Le thread initial affiche la valeur finale de la variable partagée avant de terminer. Discutez les risques d'un manque de synchronisation dans un tel programme. Écrivez un programme réalisant ces opérations de manière sûre (avec les synchronisations adéquates).

Exercice 4 : Application client-serveur. On désire simuler un mécanisme client-serveur de réservation de places. Il y a 100 places qui sont représentées par un tableau `place` de 100 entiers : `place[i]` vaut 0 si la place est libre et vaut la le numéro du client sinon. Les requêtes des clients sont reçues au clavier par le thread initial du serveur qui attend la saisie d'un entier. Si cet entier est supérieur ou égal à 0, il indique le nombre de places demandées par le client, sinon il indique l'arrêt des demandes de réservation et provoque l'affichage final du tableau de places. Après chaque demande de réservation, le thread initial créera un thread annexe pour traiter la demande et se remettra en attente d'une nouvelle requête. Les clients sont numérotés par ordre d'arrivée. Implémentez un programme respectant cette spécification. Vous veillerez en particulier à mettre en place les synchronisations nécessaires.

6 Entraînement : exercice corrigé

Le nombre fuyant. On cherche à implémenter un jeu de « nombre mystère fuyant ». Il s'agît d'une variante du jeu du nombre mystère où l'ordinateur choisit un nombre entier aléatoirement et ne répond aux propositions d'un joueur que par « *Trop petit !* », « *Trop grand !* » ou « *Gagné !* ». Dans cette variante, toutes les t secondes l'ordinateur change le nombre mystère en lui ajoutant ou en lui retirant un nombre x. Le joueur en est informé par un message (par exemple : « *Le nombre mystère a été augmenté de 13.* »). Les nombres t et x sont définis aléatoirement et changent à chaque fois qu'on les utilise (par exemple après 5 secondes de jeu l'ordinateur ajoute 32 au nombre mystère, puis au bout de 11 secondes, il lui retire 13, etc.). Le joueur n'aura de plus qu'un temps limité pour trouver le nombre fuyant.

Réalisez l'application implantant le jeu du nombre fuyant à l'aide de trois threads. Le thread initial réalisera le jeu du nombre mystère classique. Un premier thread annexe se chargera des modifications du nombre mystère dans le temps. Un second thread annexe se chargera du respect du temps limite. Pour l'implantation, le nombre mystère sera choisi entre 0 et 200, t entre 5 et 10, x entre 0 et 50 sera ajouté ou retiré (choix aléatoire) avec la contrainte de préserver le nombre mystère entre 0 et 200. Enfin, le temps limite sera de 40 secondes.

Correction. Le programme est relativement simple : il faut commencer par écrire le code du jeu du nombre mystère habituel. Ensuite on intègre un premier thread simple pour le temps maximal du jeu. Quand le temps maximum est écoulé, ce thread peut quitter tout le programme par un appel à la primitive `exit()`. Enfin on ajoute la dimension « fuyante » par un nouveau thread. Le besoin en synchronisation est centré sur le nombre mystère. On utilise un mutex pour assurer qu'un seul thread pourra accéder en lecture comme en écriture au nombre mystère : le thread initial doit tester si la proposition du joueur est correcte (accès en lecture), et le thread annexe modifie ce nombre (accès en écriture). Lorsque le joueur a gagné, le thread initial termine, mettant ainsi immédiatement fin aux autres threads.

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

#define N_INF 0
#define N_SUP 200
#define X_MAX 50
#define T_INF 5
#define T_SUP 10
#define TIMEOUT 40

int mystere;           /* nombre mystère */
pthread_mutex_t mutex;  /* mutex de protection */

void * fuyeur(void * arg) {   /* routine de départ du thread fuyeur */
  int t, x;
  while (1) {
    t = random() % (T_SUP - T_INF + 1) + T_INF;  /* temps d'attente */
    x = random() % (X_MAX + 1);                     /* modification */
    sleep(t);
    pthread_mutex_lock(&mutex);          /* protection modification */
    if (random() % 2) {                  /* on ajoute ou on retire */
      printf("Le nombre mystère a été augmenté de %d !\n",x);
      mystere = ((mystere + x) > N_SUP) ? N_SUP : mystere + x;
    } else {
      printf("Le nombre mystère a été diminué de %d !\n",x);
      mystere = ((mystere - x) < N_INF) ? N_INF : mystere - x;
    }
    pthread_mutex_unlock(&mutex);             /* fin de protection */
  }
}

void * timeout(void * arg) { /* routine de départ du thread timeout */
  sleep(TIMEOUT);
  printf("Temps écoulé ! Perdu !\n");
  exit(1);                                  /* exit() termine tout */
}

int main(int argc, char ** argv) {
  int mystere_local, proposition;
  pthread_t t1, t2;

  srandom(getpid());                /* initialisation du générateur */
  pthread_mutex_init(&mutex, NULL);      /* initialisation du mutex */
  mystere = random() % (N_SUP - N_INF + 1) + N_INF;      /* mystère */

  pthread_create(&t1, NULL, fuyeur, NULL); /* lancement des threads */
  pthread_create(&t2, NULL, timeout, NULL);

  while (1) {                    /* jeu classique du nombre mystère */
    printf("Proposition ?\n");
    scanf(" %d", &proposition);
    pthread_mutex_lock(&mutex);               /* protection lecture */
    mystere_local = mystere;
    pthread_mutex_unlock(&mutex);             /* fin de protection */
    if (proposition > mystere_local)
      printf("Trop grand !\n");
    else if (proposition < mystere_local)
      printf("Trop petit !\n");
    else
      break;
  }
  printf("Gagne !\n");
  return 0;
}
```