IUT Orsay

DUT Informatique
Module Système S4-L
2010 / 2011

Travaux Dirigés nº 5 : 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 ou de sémaphores d'exclusion mutuelle.

1 Notes de cours

1.1 Rappels sur le contexte d'un processus

Le contexte d'un processus est l'ensemble des structures de données nécessaires au système pour assurer le contrôle de ce processus. Elles se répartissent en deux catégories :

1. Structures dédiées au contrôle des ressources :
   - Masque de création des fichiers (`umask`) qui spécifie les bits de permission à exclure lors de la création de fichiers ou répertoires.
   - Propriétaires et groupes propriétaires réels (correspondant à celui qui a demandé la création du processus) et effectifs (qui détermine les droits du processus par rapport aux fichiers du système, souvent identité du propriétaire du fichier contenant le binaire exécuté).
   - Liste des descripteurs des fichiers ouverts.
   - Répertoire de travail.
   - Implantation en mémoire des données et du programme, qui comprend le segment de code ou *text segment* (le code objet et exécutable du programme), le segment de données ou *data segment* (les variables globales et `static`), et le *tas* ou *heap segment* (la mémoire allouée dynamiquement par `malloc()`).
2. Structures dédiées à l'exécution du processus :
   - Informations nécessaires à l'ordonnanceur (priorité, politique d'ordonnancement...).
   - Valeur des registres, en particulier le compteur ordinal.
   - Informations relatives aux signaux.
   - Pile d'exécution ou *stack segment* (contenant la pile des appels de fonction, les arguments et les variables locales).

1.2 Threads

La création d'un nouveau processus par la primitive *fork()* nécessite la copie complète du contexte du processus père. Cela a l'avantage de la simplicité mais est d'une part particulièrement coûteux en temps d'exécution pour le système et d'autre part pas toujours adapté aux applications avec beaucoup de parallélisme. Les *threads*, qu'on appelle aussi *processus légers* ou *activités*, sont des unités d'exécution des processus : ils travaillent directement avec les structures de données dédiées au contrôle (voir Section 1.1, catégorie 1) du processus père, qu'on appellera plus justement *activité initiale*. Leur temps de création est minimal pour le

système (qui n'a plus besoin que de copier les structures de données dédiées à l'exécution, voir Section 1.1, catégorie 2). Cependant leur manipulation est plus délicate pour les programmeurs qui doivent être parfaitement conscients des problèmes liés au travail en mémoire partagée et à l'aise avec les solutions telles que les sémaphores d'exclusion mutuelle.

En particulier, on notera que **le code, les variables globales et la mémoire allouée dynamiquement sont partagés entre les différentes activités d'un même processus**. De même, bien qu'elle ne soit connue directement que d'une activité, **une variable locale dans la pile d'une activité peut être lue ou modifiée par une autre activité si elle en connaît l'adresse**.

1.2.1 Création

```
#include <pthread.h>
int pthread_create(
    pthread_t * p_tid,              /* Pointeur sur identite du thread */
    pthread_attr_t * p_attr,        /* NULL : attributs par defaut */
    void * (*fonction)(void * arg), /* Fonction executee par le thread */
    void * arg                      /* Parametre de << fonction >> */
);
```

La primitive `pthread_create` crée une nouvelle activié et renvoie son identité à l'adresse `p_tid` (il s'agît d'un numéro entier non signé 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), `fonction` est un pointeur sur la fonction qui sera exécutée par l'activité (cette fonction retourne nécessairement un `void *` et prend nécessairement un unique argument de type `void *`). Enfin, le dernier argument `arg` correspond à l'argument transmis à la fonction `fonction`. Cette primitive retourne 0 en cas de succès, un code d'erreur sinon.

1.2.2 Identité

Chaque processus a un numéro unique, le `pid`, qui est renvoyé par la primitive `getpid()`. Tout processus est lui-même décomposé en threads qui ont chacun leur identifiant unique pour un même processus, le `tid`. On notera l'activité `tid` (par exemple `5`) du processus `pid` (par exemple `1234`) sous la forme `pid.tid` (par exemple `1234.5`). Deux primitives sont dédiées à la manipulation des identités des activités :

```
#include <pthread.h>
pthread_t pthread_self(void);
int       pthread_equal(pthread_t tid_1, pthread_t tid_2);
```

1. `pthread_self()` retourne l'identificateur du thread courant dans le processus courant (le `tid`).
2. `pthread_equal(tid_1,tid_2)` retourne 0 si les deux identités transmises en argument sont identiques et une valeur non nulle sinon.

1.2.3 Terminaison

Tous les threads d'un même processus prennent fin si l'activité initiale prend fin ou si une des activités fait appel à la primitive `exit()` (ou `_exit()`). Une activité seule prend fin automatiquement quand la fonction passée en argument de la primitive `pthread_create` retourne. Les ressources allouées pour une activité ne sont jamais libérées automatiquement. Les primitives

de terminaison d'un thread (sans affecter les autres activités) et de libération de ses ressources sont les suivantes :

```
#include <pthread.h>
void pthread_exit(void * p_status);
int  pthread_detach(pthread_t tid);
```

1. `pthread_exit(p_status)` termine l'activité qui l'a appelée en indiquant une valeur de retour à l'adresse `p_status`. Cette primitive ne peut jamais retourner dans le thread qui l'a appelée.
2. `pthread_detach(tid)` indique au système qu'il pourra récupérer les ressources allouées au thread d'identifiant `tid` lorsqu'il terminera ou qu'il peut récupérer les ressources s'il est déjà terminé. Cette primitive ne provoque pas la terminaison du thread appelant. Elle retourne 0 en cas de succès, un code d'erreur sinon.

1.2.4 Synchronisation

Les threads disposent dans les grandes lignes des mêmes outils de synchronisation que les processus. Le premier d'entre eux est l'attente passive de la fin d'une autre activité qui rappelle les primitives `wait()` ou `waitpid()` liées aux processus. La primitive permettant ce comportement est la suivante :

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

Cette primitive suspend l'exécution de l'activité appelante jusqu'à la fin de l'activité d'identifiant `tid`. Si l'activité d'identifiant `tid` est déjà terminée, cette primitive retourne immédiatement. En cas de succès, elle retourne 0 et place à l'adresse `*status` la valeur de retour de l'activité attendue. En cas d'echec elle retourne un code d'erreur.

L'autre principal outil de synchronisation entre threads est le sémaphore d'exclusion mutuelle. On manipule les sémaphores pour les threads à l'aide des quatres 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 * p_mutex, NULL);
int pthread_mutex_lock(pthread_mutex_t * p_mutex);
int pthread_mutex_unlock(pthread_mutex_t * p_mutex);
int pthread_mutex_destroy(pthread_mutex_t * p_mutex);
```

1. `pthread_mutex_init(p_mutex,NULL)` réalise l'initialisation d'un sémaphore d'exclusion mutuelle de type `pthread_mutex_t` dont on aura passé l'adresse en premier argument. Le second argument indique les attributs du sémaphore, on indiquera `NULL` pour avoir les attributs par défaut qui initialisent le sémaphore avec un droit (le sémaphore n'est pas bloqué).
2. `pthread_mutex_lock(p_mutex)` réalise l'opération P (demande d'un droit) sur le sémaphore dont l'adresse est passée en argument.
3. `pthread_mutex_unlock(p_mutex)` réalise l'opération V (restitution d'un droit) sur le sémaphore dont l'adresse est passée en argument.
4. `pthread_mutex_destroy(p_mutex)` rend le sémaphore dont l'adresse est passée en argument inutilisable à moins d'un nouvel appel à `pthread_mutex_init(p_mutex,NULL)`.

2 Exercices

2.1 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 identité :

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

#define NB_THREADS 3

int thread_execute = 0;
pthread_t tid[NB_THREADS];

/* Fonction executee par les threads. Le type de retour et l'argument sont
 * obligatoirement de type void *, ce qui necessite souvent des casts.
 */
void * fonction(void * i) {
  int n = *((int *)i);
  printf("Thread numero %d, identite %d.%u\n",n,getpid(),pthread_self());
  thread_execute = 1;
}

int main() {
  int i;
  /* Boucle de creation des threads. */
  for (i=0; i<NB_THREADS; i++) {
    if (pthread_create(&tid[i],NULL,fonction,(void *)&i) == -1) {
      fprintf(stderr,"Erreur creation thread numero %d.\n",i);
      exit(1);
    }
  }
  printf("Thread initial d'identite %d.%u\n",getpid(),pthread_self());
  if (thread_execute)
    printf("Des threads annexes ont ete executes.\n");
  else
    printf("Aucun thread annexe n'a ete execute.\n");
  return 0;
}
```

Questions :

1. 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) ?
2. Listez toutes les variables et dire par quels threads elles sont directement utilisables.
3. Comment un thread pourrait lire ou modifier la variable `n` d'un autre thread ?
4. Expliquez le résultat d'exécution suivant où le numéro de chaque thread est le même. Proposez une solution.

   ```
   Thread numero 3, identite 13033.3084860304
   Thread numero 3, identite 13033.3076467600
   Thread numero 3, identite 13033.3068074896
   Thread initial d'identite 13033.3084863168
   Des threads annexes ont ete executes.
   ```

5. Expliquez le résultat d'exécution suivant où aucun thread n'a réalisé son affichage. Proposez une solution.

   ```
   Thread initial d'identite 13433.3084601024
   Aucun thread annexe n'a ete execute.
   ```

2.2 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...). 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 $i^{ème}$ élément du vecteur `y` est calculé à partir seulement de la $i^{ème}$ 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.
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.

```
for (i=0; i<NB_LIGNES; i++)
  for (j=0; j<NB_COLONNES; j++)
    y[i] += A[i][j] * x[j];
```

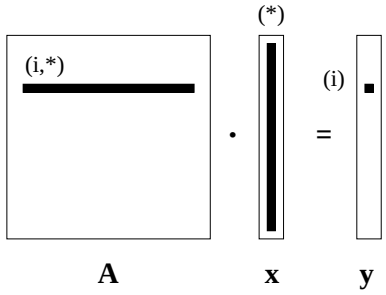

FIGURE 1 – Noyau de la multiplication matrice × vecteur

2.3 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).

2.4 Exercice 4 : 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 valeur du numéro du client sinon. Les requêtes des clients sont reçues au clavier par le thread initial du serveur qui attend la frappe 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.
Implantez un programme respectant cette spécification. Vous veillerez en particulier à mettre en place les synchronisations nécessaires.

3 Entraînement : exercice corrigé

3.1 Énoncé : le nombre fuyant

On cherche à implanter 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.

Note : pour les temporisations, utilisez seulement des primitives `sleep()`. L'utilisation du signal `SIGALRM` est possible (c'est d'ailleurs ce que fait `sleep()`) puisque les informations sur les signaux sont liées aux threads et non au processus. Cependant il n'est pas possible d'utiliser la primitive `signal()` qui ne fonctionne pas comme elle le devrait avec les threads (elle s'applique à tous les threads d'un même processus). À la place il faudrait utiliser la primitive `sigaction()` qui n'est pas au programme (voir pages de man pour les curieux !).

3.2 Correction (essayez d'abord ! ! !)

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 arrivé, 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 sémaphore d'exclusion mutuelle 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 activités.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.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 mystere */
pthread_mutex_t mutex;                      /* Semaphore de protection */

void * fuyeur() {                           /* Fonction du thread fuyeur */
  int t, x;
  while(1) {
    t = rand()%(T_SUP - T_INF + 1) + T_INF; /* Temps d'attente */
    x = rand()%(X_MAX + 1);                 /* Modification */
    sleep(t);
    pthread_mutex_lock(&mutex);             /* Protection modification */
    if (rand()%2) {                         /* On ajoute ou on retire */
      printf("Le nombre mystere a ete augmente de %d !\n",x);
      mystere = ((mystere + x) > N_SUP) ? N_SUP : mystere + x ;
    }
    else {
      printf("Le nombre mystere a ete diminue de %d !\n",x);
      mystere = ((mystere - x) < N_INF) ? N_INF : mystere - x ;
    }
    pthread_mutex_unlock(&mutex);           /* Fin de protection */
  }
}

void * timeout() {                          /* Fonction du thread timeout */
  sleep(TIMEOUT);
  printf("Temps ecoule ! Perdu !\n");
  exit(1);                                  /* exit() termine tout */
}

int main(int argc, char *argv[]) {
  int proposition = N_INF - 1;
  pthread_t t1, t2;

  srand(getpid());                          /* Initialisation generateur */
  pthread_mutex_init(&mutex,NULL);          /* Initialisation du semaphore */
  mystere = rand()%(N_SUP-N_INF+1) + N_INF; /* Initialisation du nb mystere */

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

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