Systèmes d'exploitation

Série d'exercices N°3

Consignes aux étudiants :

- Vous pouvez travailler par binôme.
- N'hésitez pas d'appeler votre professeur pour vous expliquer et interpréter le résultat.
- Vous devez rendre un compte rendu à la fin de la science.

Objectifs :

- Les Threads : Fonctionnement, Communication, Synchronisation et Clonage

---

I. Ordonnancement et Synchronisation des Threads.

Exercice 1 :

1. Ecrire un programme qui a comme objectif principal de créer un thread fils qui affiche un caractère **`'x'`** de façon continue (dans une boucle infinie) sur la sortie standard. Le thread principal lui aussi affiche un autre caractère **`'o'`** infiniment sur la sortie standard.

2. Exécuter ce programme et expliquer le motif obtenu par les affichages alternatives des caractères **`'x'`** et **`'o'`**. (Prouvez-vous prédire combien de fois sera affiché chaque caractère **`'x'`** et **`'o'`**)

Exercice 2 :

1. Ecrire un programme dans lequel on déclare une structure composée de deux champs : le premier est de type **`char`** et le deuxième est de type **`int`**. Dans la fonction main, créer deux threads fils en leur passant comme paramètre de la fonction du thread deux instances de cette structure. La fonction de thread se charge d'afficher le caractère du premier champ (**`character`**) **`count`** fois. Où **`count`** est la valeur du deuxième champ.

```
struct parms {
    char character;
    int count;
};
```

```
struct parms p1,p2;

p1.character = 'x'; p1.count = 3000;
p2.character = 'o'; p2.count = 2000;
```

```
pthread_create (..., ..., ..., &p1);
pthread_create (..., ..., ..., &p2);
```

2. Exécuter votre programme dans un premier temps sans utiliser la fonction *pthread_join*. Expliquer pourquoi son exécution s'arrête avant même que l'affichage des deux threads ne soit terminé.

3. Qu'est-ce vous pouvez proposer comme solution pour que l'affichage demandé à travers les deux threads soit terminé.

II. Synchronisation et Sections Critiques

Exercice 3 :

On désire créer un programme ayant une série de tâches à traiter par plusieurs threads concurrents. La file d'attente des tâches est représentée par une liste chaînée d'objets ***struct_job*** sous la forme :

```
struct job {
      struct job* next;
      char character;
      int count;
};
struct job* job_queue;        /* Liste chaînée de tâches en attente. */
```

A la fin de l'exécution de chaque tâche (un nœud de la liste), le thread vérifie la file pour voir si une nouvelle tâche est disponible. Si *job_queue* n'est pas **NULL**, le thread supprime et récupère la tête de la liste chaînée et fait pointer *job_queue* vers la prochaine tâche de la liste.

La fonction de thread qui traite les tâches de la liste ressemble au code suivant :

```
void* thread_function (void* arg) {
    while (job_queue != NULL) {
        struct job* next_job = job_queue;   /* Récupère la tâche suivante.*/
        job_queue = job_queue->next;     /* Supprime cette tâche de la liste.*/
        process_job (next_job);         /* Traite la tâche.*/
        free (next_job);             /* Libération des ressources.*/
    }
    return NULL;
}
```

Compléter ce code comme suit :

1. Ajouter une fonction **`void enqueue_job (char chart, int count)`** permettant d'insérer des nouvelles tâches à la liste **`job_queue`**.
2. Définir le corps de la fonction **`process_job (next_job)`** dans l'objectif d'afficher le caractère **`chart`** dans la sortie standard **`count`** fois.
3. Ajouter une fonction principale main() dans laquelle vous allez créer au moins deux threads permettant d'exécuter les tâches de la liste **`job_queue`**.

Exécuter ce programme pour la première fois. Puis changer la fonction **`process_job(next_job)`** par l'ajout de l'instruction **`sleep(2)`**. Expliquer pourquoi le programme se plante.

Exercice 4 : Mutexes

Afin d'assurer l'exclusion mutuelle, Linux propose l'utilisation des ***mutexes***. Un ***mutex*** (*MUTual EXclusion locks*) est un verrouillage spécial qu'un seul thread peut utiliser à la fois. Si un thread verrouille un mutex puis qu'un second tente de verrouiller le même mutex, ce dernier est *bloqué* ou *suspendu*. Le second thread est *débloqué* uniquement lorsque le premier déverrouille le mutex. Ce qui permet la reprise de l'exécution. Linux assure qu'il n'y aura pas de condition de concurrence critique entre deux threads tentant de verrouiller un mutex; seul un thread obtiendra le verrouillage et tous les autres seront bloqués.

Pour créer un mutex, on utilise la syntaxe suivante :

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

Ou encore :

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Un thread peut essayer de verrouiller un mutex en appelant *pthread_mutex_lock*. Si le mutex était déverrouillé, il devient verrouillé et la fonction se termine immédiatement. Si le mutex était verrouillé par un autre thread, *pthread_mutex_lock* bloque l'exécution et ne se termine que lorsque le mutex est déverrouillé par l'autre thread. Lorsque le mutex est déverrouillé, seul un des threads suspendus (choisi aléatoirement) est débloqué et autorisé à accéder au mutex; les autres threads restent bloqués. Un appel à *pthread_mutex_unlock* déverrouille un mutex. Cette fonction doit toujours être appelée par le thread qui a verrouillé le mutex.

1. Changer le programme créé précédemment en une autre version où l'accès à la liste des tâches (`job_queue`) doit être protégé par un ***mutex***. Avant d'accéder à la file (que ce soit pour une lecture ou une écriture), chaque thread commence par verrouiller le mutex. Ce n'est que lorsque la séquence de vérification de la file et de suppression de la tâche est terminée que le mutex est débloqué.
2. verrouiller le ***mutex*** deux fois successives et ré-exécuter votre programme. Que se passe-t-il ? Pourquoi ?

Exercice 5 : Sémaphores

Un sémaphore est un compteur qui peut être utilisé pour synchroniser plusieurs threads. Comme avec les mutexes, Linux garantit que la vérification ou la modification de la valeur d'un sémaphore peut être accomplie en toute sécurité, sans risque de concurrence critique.

Un sémaphore est représenté par une variable *sem_t*. Avant de l'utiliser, on doit l'initialiser par le biais de la fonction *sem_init*, à laquelle on passe un pointeur sur la variable *sem_t*. Le second paramètre doit être à zéro *(Une valeur différente de zéro indiquerait que le sémaphore peut être partagé entre les processus, ce qui n'est pas supporté sous Linux pour ce type de sémaphore.)*, et le troisième paramètre est la valeur initiale du sémaphore. Si vous n'avez plus besoin d'un sémaphore, il est conseillé de libérer les ressources qu'il occupe avec *sem_destroy*.

Pour vous mettre en attente sur un sémaphore, utilisez *sem_wait*. Pour effectuer une opération de réveil, utilisez *sem_post*. Une fonction permettant une mise en attente non bloquante, *sem_trywait*,

est également fournie. si l'attente devait bloquer en raison d'un sémaphore à zéro, la fonction se termine immédiatement, avec le code d'erreur *EAGAIN*, au lieu de bloquer le thread.

1. Changer le programme de l'exercice 3 en une autre version de la file de tâches dans laquelle vous devez :
   - utiliser un sémaphore qui compte le nombre de tâches en attente dans la file. *sem_t job_queue_count;*
   - Définir une fonction *void initialize_job_queue()* permettant d'initialiser *job_queue* par *NULL* et le sémaphore *job_queue_count* par *zéro.*
   - Dans la fonction de thread, avant de prélever une tâche en tête de file, chaque thread se mettra en attente sur le sémaphore. Ce qui veut dire, si la valeur de sémaphore *job_queue_count* égale à zéro, indiquant que la file est vide, le thread sera tout simplement bloqué jusqu'à ce que le sémaphore devienne positif, indiquant que la tâche a été ajoutée à la file.
   - Dans la fonction *enqueue_job* et après avoir ajouté une tâche à la file, envoyer un signal de réveil au sémaphore pour indiquer qu'une nouvelle tâche est disponible.
   - La fonction *enqueue_job* et la fonction de thread (*thread_function*) doivent verrouiller le mutex de la file des taches avant de la modifier.

Exercice 6 : Variables de Condition

Une *variable de condition* est un troisième dispositif de synchronisation que fournit Linux et qui permet de spécifier une condition qui lorsqu'elle est remplie autorise l'exécution du thread et inversement, une condition qui lorsqu'elle est remplie bloque le thread. Du moment que tous les threads susceptibles de modifier la condition utilisent la variable de condition correctement, Linux garantit que les threads bloqués à cause de la condition seront débloqués lorsque la condition change. Une variable de condition est représentée par une instance de *pthread_cond_t*. Une variable de condition doit être accompagnée d'un mutex. Voici les fonctions qui manipulent les variables de condition:

- *pthread_cond_init* initialise une variable de condition. Le premier argument est un pointeur vers une variable *pthead_cond_t*. Le second argument, un pointeur vers un objet d'attributs de variable de condition, est ignoré par GNU/Linux. Le mutex doit être initialisé à part, comme indiqué dans la Section 4.4.2, « Mutexes ».
- *pthread_cond_signal* valide une variable de condition. Un seul des threads bloqués sur la variable de condition est débloqué. Si aucun thread n'est bloqué sur la variable de condition, le signal est ignoré. L'argument est un pointeur vers la variable *pthread_cond_t*. Un appel similaire, *pthread_cond_broadcast*, débloque *tous* les threads bloqués sur une variable de condition, au lieu d'un seul
- *pthread_cond_wait* bloque l'appelant jusqu'à ce que la variable de condition soit validée. L'argument est un pointeur vers la variable *pthread_cond_t*. Le second argument est un pointeur vers la variable *pthread_mutex_t*. Lorsque *pthread_cond_wait* est appelée, le mutex doit déjà être verrouillé par le thread appelant. Cette fonction déverrouille automatiquement le mutex et se met en attente sur la variable de condition. Lorsque la

variable de condition est validée et que le thread appelant est débloqué, *pthread_cond_wait* réacquiert automatiquement un verrou sur le mutex.

Lorsqu'un programme effectue une action qui pourrait modifier la condition protégé avec la variable de condition, il doit suivre les étapes suivantes :

- Verrouiller le mutex accompagnant la variable de condition.
- Effectuer l'action qui pourrait modifier la condition.
- Valider ou effectuer un broadcast sur la variable de condition, selon le comportement désiré.
- Déverrouiller le mutex accompagnant la variable de condition.

Soit le programme suivant :

```
#include <pthread.h>

int thread_flag;
pthread_mutex_t thread_flag_mutex;

void initialize_flag (){
      pthread_mutex_init (&thread_flag_mutex, NULL);
      thread_flag = 0;
}

/* Appelle do_work de façon répétée tant que l'indicateur est actif ;
sinon, tourne dans la boucle. */
void* thread_function (void* thread_arg) {
      while (1) {
            int flag_is_set;
            /* Protège l'indicateur avec un mutex. */
            pthread_mutex_lock (&thread_flag_mutex);
            flag_is_set = thread_flag;
            pthread_mutex_unlock (&thread_flag_mutex);
            if (flag_is_set)
            do_work ();
            /* Rien à faire sinon, à part boucler. */
      }
      return NULL;
}

/* Positionne la valeur de l'indicateur de thread à FLAG_VALUE. */
void set_thread_flag (int flag_value) {
      /* Protège l'indicateur avec un verrouillage de mutex. */
      pthread_mutex_lock (&thread_flag_mutex);
      thread_flag = flag_value;
      pthread_mutex_unlock (&thread_flag_mutex);
}
```

Changer le programme suivant en une autre version qui utilise une variable de condition pour protéger l'indicateur.

Exercice 7 : Clonage

Il existe un autre appel système sous linux qui permet de réaliser un dédoublement du processus ou thread comme fork ou creat_thread, d'où son nom de clone. Cet appel système permet de préciser exactement ce qu'on partage entre le père et le fils.

Le prototype de cette fonction est le suivant :

```
#include <sched.h>
int clone (int (*fn) (void *), void *child_stack, int flags, void *arg);
```

Avec :

- **`int (*fn) (void *)`:** L'argument **`fn`** est un pointeur sur la fonction appelée par le processus fils lors de son démarrage.
- **`void *child_stack` :**
  cet argument indique l'emplacement de la pile utilisée par le processus fils. En effet, c'est un espace mémoire préparé par le processus appelant pour stocker la pile de son fils. Il est donc transmit à **`clone()`** comme un pointeur sur **`void`** dans cet emplacement.
- **`int flags`** : ce parametre permet de préciser ce qui sera partagé entre le père et le fils, en effectuant un **`OU`** binaire entre ***zéro*** ou ***plusieurs*** constantes:
  - **`CLONE_VM`** : espace mémoire
  - **`CLONE_FS`** : root, cwd, umask
  - **`CLONE_FILES`** : fichiers ouverts
  - **`CLONE_PARENT`** : même parent
  - **`CLONE_PID`** : même pid (uniquement si parent = 0, lors du boot)
  - **`CLONE_PTRACE`** : « ptracé » comme parent
  - **`CLONE_SIGHAND`** : même traitants d'interruptions
  - **`CLONE_THREAD`** : même groupe que parent (= thread POSIX)
  - **`CLONE_SIGNAL`** : CLONE_THREAD + CLOSE_SIGHAND
  - **`CLONE_VFORK`** : bloque parent jusqu'à mort ou exec du fils
- **`void *arg`:** cet argument est transmis à la fonction **`fn`** lors de son invocation.

En cas de réussite de la fonction **`clone()`**, le **`PID`** du processus fils est renvoyé dans le thread d'exécution de l'appelant. En cas d'échec, **`-1`** est renvoyé dans le contexte de l'appelant, aucun fils n'est créé.

Question

Ecrire un programme qui calcule et affiche la table de multiplication d'un nombre donnée. Cette table est créée dans le processus fils et affichée dans la sortie standard par le processus père. L'entier N sera passé comme argument à la fonction main lors de l'exécution du programme.

Dans cet exercice, les paramètres de clone sont les suivants :

- le premier est la fonction qui sera exécuté par le processus fils
- le deuxième est l'espace mémoire de **`1024 * 1024`** créé par **`malloc`**.
  `void *pchild_stack = malloc(1024 * 1024);`
- le troisième **`flags`** sera **`SIGCHLD`**
- le quatrième est l'entier **`N`** contenu dans **`argv[1]`**