Rapport du projet de système d'exploitation

11 avril 2011

Encadrant : Albert Jérémie

Elèves : Hichri Houda, Julien Python, Lissy BARRO
Nada El Ouadrhiri, Yassine El Ghali

ENSEIRB-MATMECA I2 2010/2011 semestre n°4 IT202

1 Introduction

L'objectif de ce projet est de mettre en oeuvre, sur un cas pratique, les notions et les acquis vus dans le module de système d'exploitation. Le but est dans un premier temps de construire une bibliothèque de gestion de threads proposant un ordonnancement coopératif (non pas préemptif). On devra donc tout d'abord définir une interface de threads permettant de créer, détruire, passer la main, passer la main à un thread particulier, attendre la/une terminaison, de manière à empêcher tout appel système d'un thread vers le noyau. Les threads ne communiqueront alors plus qu'avec les processus. Aussi, on veillera à rester relativement proche de *pthread.h* afin de pouvoir facilement comparer les deux implémentations avec des programmes similaires.

2 Structures de données utilisées

Pour construire la bibliothèque des `thread` demandée, on a choisi d'utiliser la structure de donnée `File` implémentée par une liste doublement chainée. La structure est appelée `fifoThread` et elle est définie dans le fichier `fifoThread.h` :

```
1 Structure fifoThread
2 |  element * head; | un pointeur vers le premier element de la file
3 |  element * tail; | un pointeur vers le dernier element de la file
```

La structure `element` est définie comme suit dans le fichier `fifoThread.h`

```
1 Structure element
2 | thread * th;                | un pointeur vers la stucture thread
3 | struct element * next;      | un pointeur vers l'element suivant
4 | struct element * previous;  | un pointeur vers l'element precedent
```

La structure `thread` représente toutes les informations concernant un thread et elle est définie dans le fichier `threadStructure.h` qui contient aussi toute les fonctions représentant les opérations élémentaires sur un thread.

```
1 Structure thread
2 |  int tid;                    | l'identifiant du thread
3 |  int priorite;               | la priorite du thread
4 |  ucontext_t * context;       | le context du thread de type ucontext_t*
5 |  status state;               | le status du thread qui est soit READY soit FINISHED
6 |  void * valRetour;           | la valeur de retour du thread
7 |  void ** adresseValRetour;   | l'adresse de la valeur de retour du thread
```

Pour stocker les threads nous utilisons deux `Files`, une pour les threads qui n'ont pas encore terminé leurs exécutions et l'autre pour les threads qui ont terminé.

3 Les fonctions de la bibliothèque

Toutes les structures de données définies plus haut ont servi à l'implémentation de fonctions principales de notre bibliothèque qui sont les suivantes :[1]

3.1 Fonction `thread_self`

Cette fonction retourne l'identifiant du thread courant. Le thread qui s'exécute est stocké dans une variable globale et donc à chaque appel de `thread_self` on retourne l'identifiant du thread référencé par la variable globale.

3.2 Fonction `thread_create`

Cette fonction permet de créer un thread. Comme défini plus haut, un thread est représenté par une structure et dans celle-ci se trouve un context qui correspond à un ensemble d'instructions que le thread exécute. A chaque appel de la fonction `thread_create`, le context du thread appelant est sauvegardé et remplacer par le contexte du thread créé. Ainsi un thread créé est automatiquement lancé et le thread appelant est mis en file d'attente.

3.3 Fonction `thread_yield`

À l'appel de cette fonction, le thread appelant entend rendre la main. Lorsqu'un thread est lancé, on garde dans le thread l'information sur le contexte dans lequel il a été lancé donc `thread_yield` sauvegarde dans le thread appelant son contexte et lance le contexte dans lequel celui-ci a été précédemment lancé.

3.4 Fonction `thread_exit`

À l'appel de cette fonction, le thread appelant est sensé avoir fini son exécution et cherche donc à stocker sa valeur de retour. La fonction `thread_exit` se contente de stocker dans la structure du thread appelant sa valeur de retour, de signaler que le thread a terminé en mettant son statut à STATUS_FINISH et de lancer le contexte qui a précédé le sien.

[1] Pour un détail sur les prototypes se réferer au fichier `thread.h`, nous donnons juste les principes des fonctions

3.5 Fonction `thread_join`

Cette fonction est utilisée pour attendre la terminaison d'un thread. À l'appel de cette fonction, le thread appelant est stoppé jusqu'à ce que le thread qu'il a décidé d'attendre termine. Ainsi sa valeur de retour est récupérée et stockée dans une variable passée en paramètre.

Commentaire

Il apparait clairement que les fonctions `thread_exit` et `thread_join` doivent être liées. En effet, un thread est sensé avoir terminé qu'après un appel (et un seul est possible) de la fonction `thread_exit`. Un appel `thread_join` n'est possible que sur les seuls thread qui font un appel `thread_exit` pendant leurs exécutions. Cependant l'ordre est sans importance : si on décide de faire un appel `thread_join` pour attendre un thread qui a déjà fait un `thread_exit` alors on récupère juste sa valeur de retour sinon on reste dans la fonction `thread_join` jusqu'à ce que le thread attendu face un `thread_exit` pour terminer. On comprend donc par là que c'est dans la fonction `thread_join` que la politique d'ordonnancement est mis en oeuvre pendant l'attente d'un thread et c'est elle qui libère les ressources utilisées par les threads qui ont terminé.

4 Tests :

Pour tester l'implémentation la bibliothèque *thread* que nous avons mis en place, nous avons créé plusieurs fichiers de tests comme demandé pour vérifier par exemple la bonne libération des ressources (et notamment en utilisant **Valgrind**),en plus d'applications qui utilisent et comparent les deux bibliothèques *thread* et *pthread*.

4.1 Les perfomances :

4.1.1 Coût de création et d'attente :

Pour comparer le coût de la création avec thread et pthread, le fichier `creationAttente.c` a été mis en place.
On y crée un thread et un pthread. On a aussi deux fonctions `pthreadfunc` et `threadfunc` qui utilisent respectivement les fonctions des bibliothèques *pthread* et *thread*.
Nous remarquons alors que le coût de la création et d'attente avec les fonctions de la bibliothèque *thread* est moindre par rapport aux autres. Nous pouvons expliquer ceci par le fait qu'avec un fonctionnement préemptif (utilisant la bibliothèque *pthread*), le coût des appels système (les threads communiquent directement avec le kernel) s'additionne, contrairement aux fonctions avec la bibliothèque *thread* qui elles, utilisent un ordonnancement coopératif.
Il est possible de compiler ce fichier à l'aide de la commande `make creationAttente`.

4.1.2 Coût du changement de contexte :

Pour comparer le coût du changement de contexte avec thread et pthread, le fichier `chgContext.c` a été mis en place.
On y crée deux threads et trois pthreads. On a aussi deux fonctions `pthreadfunc` et `threadfunc` qui utilisent respectivement les fonctions des bibliothèques *pthread* et *thread* comme ce qui a été vu précédemment.
Nous remarquons alors que le coût du changement de contexte avec les fonctions de la bibliothèque *pthread* est moindre par rapport aux autres.
Il est possible de compiler ce fichier à l'aide de la commande `make chgContexte`.

4.1.3 Coût de la destuction :

Pour comparer le coût de la destruction avec thread et pthread, le fichier `destruction.c` a été mis en place.
C'est un fichier semblable au programme exemple qui nous a été fourni, sauf que nous utilisons deux fonctions : `pthreadfunc` et `threadfunc` qui utilisent respectivement les fonctions des bibliothèques *pthread* et *thread*.
Nous remarquons alors que le coût de la destruction avec les fonctions de la bibliothèque *thread* est de loin inférieur à celui avec les autres fonctions.
Aussi, nous avons expliciter le fait qu'il soit possible, grâce à la nouvelle implémentation, de créer des threads qui font un `thread_join` avant le `thread_exit`, et vis versa. Pour ce, le fichier `freedMemory.c` a été créé. Il est possible de compiler ces fichiers à l'aide de la commande `make freedMemory` et `make destuction`.

4.2 Les applications :

4.2.1 Somme de tous les éléments d'un grand tableau par diviser-pour-régner :

Cette application se trouve dans le fichier `somme.c`.
Le principe est de diviser le tableau pour en calculer la somme plus rapidement. Notre tableau est de taille 1000, on le divise en dix tableaux de taille 100 en lançant un thread sur chacun des tableaux. Les threads calculent la somme de la partie du tableau qui les concerne, puis on retourne la somme totale.
Ceci est effectué avec les fonctions de la bibliothèque *pthread* et les fonctions implémentées de la bibliothèque *thread*, chose qui nous permet de comparer le coût du calcul de la somme des deux différentes manières.
On remarque alors que le fait d'ulitiser la bibliothèque *thread* est plus rapide puisque l'on peut en parallel calculer les différentes parties du tableau. Concernant la bibliothèque *pthread*, on peut justifier le fait qu'il soit plus lent relativement à l'autre bibliothèque par le fait qu'il ait besoin d'un appel système, c'est-à-dire qu'il va jusqu'au noyau pour faire toutes les opérations.

4.2.2 Calcul de la suite de Fibonacci :

Cette application se trouve dans le fichier **fibo.c**.
Pour le calcul du terme n de la suite, un thread principal est lancé. Sachant que pour ce faire ,on a besoin des deux termes précédents (terme $n-1$ et terme $n-2$), le thread principal lance deux autres sur les calculs de ces termes et attend qu'il terminent pour pouvoir utiliser leur valeur de retour et tous les threads se comportent comme un thread principal. Avant de lancer un thread sur le calcul d'un terme, le thread principal effectue un test au préalable pour verifier si la valeur du terme en question n'est déjà pas disponible afin d'éviter qu'un thread refasse un calcul déjà effectué par un autre et optise-t-on ainsi le nombre de threads utilisés dans le calcul des termes de la suite de fibonacci.

4.2.3 Tri d'un grand tableau :

Cette application se trouve dans le fichier **tri.c**.
Sans faire de fusion, nous nous sommes juste limités à trier le tableau portion par portion. Un tableau de taille 100 est divisé en des portions de 10 éléments chacune et un thread est lancé sur une portion pour l'ordonner de façon croissante. Après chaque itération un appel `thread_yield` est effectué pour attester le changement de contexte sans faille de notre bibliothèque. Comme

resultat, sur plusieurs exécutions on constate que notre bibliothèque est plus performante que la bibliothèque `lpthread`.

5 Objectifs libres :

En ce qui concerne les objectifs libres, nous envisageons de travailler sur :

- l'amélioration de l'ordonnancement des threads : Par exemple des priorités, des affinités entre threads, voire permettre de choisir entre différentes politiques d'ordonnanceurs (à la compilation ou dynamiquement).
- l'ajoût des fonctions de synchronisation de type sémaphores et/ou mutex pour permettre aux threads de manipuler des données partagées de manière sécurisée. On réfléchira à la validité de passer la main lorsqu'on tient un verrou et l'impact que cela peut avoir sur l'implémentation (attente active ou passive ?).
- Etudier le cas du thread principal (le main du programme) et être capable de la manipuler comme n'importe quel autre thread.
- Préemption : On pourra commencer par une préemption légère basée sur une coopération régulière des threads, avant de s'intéresser à une vraie préemption utilisant par exemple des signaux.

6 Conclusion :

Finalement, nous estimons que ce projet a été et est une bonne occasion pour mettre en pratique et pour renforcer nos acquis en matière d'assimilation des diverses notions et subtilités des systèmes d'exploitation et nous sommes heureux d'avoir réussi à achever les objectifs principaux tout en s'obligeant à toujours bien libérer les ressources et à respecter les consignes données.
Pour ce qui est des objectifs libres, nous espérons arriver à faire ce qu'on a prévu, voire plus si le temps nous le permet.