MIM 1 – Cours de Système – Travaux dirigés
À la découverte des Threads Linux

Emmanuel Agullo, Eddy Caron

10 février 2006

Note *Pour vous aider, les sources des canevas de programmes sont chargeables sous* `http://graal.ens-lyon.fr/~eagullo/systeme/threads/` *ou dans* `/home/eagullo/teach/systeme/threads`.

L'objectif de cette séance est de préparer la mise en place des *threads* (en français, *processus légers*) sous Nachos, votre premier travail d'envergure. Si vous maîtrisez déjà tout ce qui suit, ou si vous êtes vraiment pressé de vous replonger dans Nachos, vous pouvez oublier ce TD.

Il s'agit de vous familiariser avec l'utilisation de la notion de *threads* au travers de petits programmes d'exemple. L'accent sera mis sur l'interface standard *POSIX Threads* définissant un ensemble de fonctionnalités supportées par de nombreux systèmes Unix contemporains.

Partie I. Le *Hello world !* des *threads*

Le programme `thread` crée deux *threads* concurrents en plus du *thread* père.

Récuperez le source `threads.c` et compilez-le. Utilisez les options de compilation `-D_REENTRANT` (pour demander une version *thread-safe* de la bibliothèque standard) et `-lpthread`.

```
gcc -Wall -D_REENTRANT -lpthread threads.c -o threads
```

Alors ? Prenez le temps de bien comprendre ce qui se passe...

Action I.1. *Essayez avec* `sleep(0)`, `sleep(2)`, *etc. Faites dormir l'un des* threads *2 ou 3 fois plus que l'autre.*

Action I.2. *Remplacez l'appel à* `sleep` *par une boucle d'attente active. (comptez plusieurs milliards d'itérations, vous êtes à plusieurs GHz) Que constatez-vous ? Que pouvez-vous en déduire à propos de l'ordonnancement par défaut des processus légers ?*
Vérifiez qu'il est possible de forcer les changements de contexte à l'aide de la fonction `sched_yield()` *qui cède la main au* thread *suivant (*`#include <sched.h>`*).*

Action I.3. *Passez le* `int i` *de la fonction* `writer_func` *en* `static int i`. *Refaire les diverses expériences ci-dessus avec cette modification. Que constatez-vous ?*

Action I.4. *Essayez avec le programme* `update`. *Vous avez compris ce qui se passe ?*

Action I.5. *Dans* `threads`, *au lieu d'utiliser directement la chaîne de caractères* `message` *lors de la création du thread, utilisez un pointeur.*

```
char *s;
        ...
s = "Alice";
i = pthread_create(&writer1_pid, &attributes, writer_func, s);
        ...
s = "Bob";
i = pthread_create(&writer1_pid, &attributes, writer_func, s);
        ...
```

Essayez de prédire ce qui va se passer.

Action I.6. *Et maintenant, utilisez un tampon et non plus un pointeur.*

```
#include <string.h>
char s[1024];
        ...
strcpy(s, "Alice");
i = pthread_create(&writer1_pid, &attributes, writer_func, s);
        ...
strcpy(s, "Bob");
i = pthread_create(&writer1_pid, &attributes, writer_func, s);
        ...
```

Essayez de prédire ce qui va se passer.

Partie II. *Threads* et exclusion mutuelle

Les *threads* partagent les données globales du programme. Il est donc nécessaire de synchroniser les accès concurrents à ces données. Nous étudions d'abord l'exclusion mutuelle. Tous les détails sont obtenus par `man pthread_mutex_lock`, `pthread_cond_wait`, ... qui décrivent les verrous, sémaphores et moniteurs. Nous n'utilisons ici que les *mutex* intra-processus. Le programme `mutex` est un exemple simple d'utilisation des verrous.

Action II.1. *Exécutez le programme `mutex` et prenez le temps de bien comprendre ce qui se passe. Faites varier le `non_critical_work` : prendre par exemple `sleep(0)`, `sleep(10)`, `sched_yield()`, un travail dépendant du* thread*, etc.*

Action II.2. *Essayer d'enlever les verrous et regarder ce qui se passe dans les différents cas.*

Action II.3. *Inversez les `printf` et les appels à `mutex`.*

Partie III. *Threads* et variables de condition

Un verrou doit être acquis et libéré par le même *thread*. Par contre, une variable de condition (comme un sémaphore) peut être gérée en commun par plusieurs *threads*. Dans l'exemple `conditions`, les *threads* producteurs déposent des valeurs dans un tampon partagé ; les *threads* consommateurs sont bloqués en attente d'une valeur. Lorsqu'un consommateur a déposé la valeur, il signale que celle-ci est présente aux consommateurs par la variable condition `not_empty_cond`. Lorsque le consommateur a retiré la valeur, il signale aux producteurs que le tampon est disponible par la variable condition `not_full_cond`.

En fait, la solution ci-dessus est incomplète. La documentation précise que `pthread_cond_wait` peut être interrompu sur réception d'un signal Unix. Notez l'utilisation d'une boucle `while` pour prendre ce détail en compte : il n'est pas garanti que la condition soit bien vérifiée lorsqu'on sort de l'attente `pthread_cond_wait`...

Action III.1. *Exécutez le programme* `conditions`*. Modifiez les vitesses relatives par des appels à la fonction* `sleep` *pour rendre les comportements des* threads *très différents et variables dans le temps.*

Action III.2. *Adaptez votre code pour vérifier que les caractère entrés dans le tampons en sont bien extraits* dans le même ordre*, sans perte ni duplication (comportement* FIFO*).*

Partie IV. Utilisation "réelle" des *threads*

Action IV.1. Entrées/Sorties : *Ecrivez un petit programme lisant de manière simultanée sur plusieurs terminaux. Ouvrez deux (petites) fenêtres. Faites la commande* `tty` *dans chacune d'elles. Mettez-vous en attente de lecture* en même temps *dans chacune d'elles en utilisant deux* threads*, et écrivez les valeurs lues dans une troisième fenêtre. Attention aux conflits de lecture avec le* `shell`*...*

Attention, `gcc -Wall` *ne doit donner aucun avertissement. Soignez les conversions de type. Comparez la facilité d'utilisation par rapport à l'utilisation de* `select`*. Notez que les fenêtres peuvent être déportées sur d'autres machines via le protocole X.*

Action IV.2. Facilité d'emploi : *Ecrivez un petit programme effectuant un calcul récursif (typiquement Fibonacci) à l'aide de processus légers. Mesurez le temps utilisé. Comment pourait-on faire la même chose avec des processus Unix ?*

Action IV.3. Efficacité : *Mesurez le temps moyen de création+destruction d'un* thread *dont la fonction comportementale est vide. Comparez avec la création+destruction d'un processus Unix.*

Vous êtes maintenant tout à fait prêts pour attaquer la mise en place des *threads* en Nachos.