Architecture et Systèmes

Stefan Schwoon

Cours L3, 2016/17, ENS Cachan

Les threads

Un thread est un fil d'exécution *à l'intérieur d'un processus*.

(aussi appelé processus léger)

Rappel: Les processus sont des unités d'exécution avec leurs propres ressources (mémoire, signaux, ...) qui sont entièrement indépendant l'un de l'autre.

Par contre, les threads d'un processus se partagent la plupart de leurs ressources.

Ressources privés et partagés

Un thread possède :

- son propre compteur de programme, son jeux de registres et sa propre pile
- ses variables locales (non `static`), qui sont allouées sur la pile
- code de retour (un pointeur `void*`)

Les threads d'un même processus se partagent :

- tout le reste de la mémoire, en particulier le tas
- leurs fichiers ouverts
- (partiellement) les signaux

Travailler avec les threads

Librairie standard : Posix threads (Pthreads), standard défini en 1995

Pour compiler, inclure `pthreads.h` et utiliser `gcc -pthread`.

Voir `pthreads(7)` pour un survol de la thématique.

Remarques

On peut imaginer un processes comme une collection de threads
(par défaut, il n'ya qu'un seul).

Aucune hiérarchie entre les threads d'un même processus.

Les threads obtiennent leur temps de calcul par le noyau (*kernel threads*).
Certains versions de Linux connaissent aussi les *user threads*, où l'ordonnancement entre threads se fait par un ordonnanceur à l'intérieur du processus.

Un thread possède un identifiant (type `pthread_t`).
Cet identifiant peut être utilisé avec les fonctions de la famille pthread.

Création d'un thread

N'importe quel thread peut en créer un autre avec `pthread_create`.

Contrairement à `fork`, le thread actuel n'est pas dupliqué ;
le nouveau thread commence sa vie dans une fonction de départ fournie comme paramètre.

Cette fonction de départ prend un argument de type `void*` et renvoie un `void*`.

Terminaison d'un thread

Un thread est vivant jusqu'à ce qu'il termine sa fonction de départ.
Il peut aussi terminer avec `phread_exit`.

Dans les deux cas, on enregistre une valeur de sortie.

N'importe quel thread peut attendre la terminaison d'un autre thread et rattraper sa valeur de sortie, avec `pthread_join`.

Un thread peut en tuer un autre – en inclus le thread de départ – avec `pthread_cancel`.

Divers

`pstree` affiche les threads en accolades.

`pthread_detach` rend un thread “non-joignable”:

lors de la terminaison, il sera supprimé entièrement toute suit;
il n’entrera pas dans un état de “zombie”.

par contre, ce thread ne pourra pas laisser une valeur de sortie.

Vie et mort d'un processus avec threads

Un processus est vivant lorsqu'il possède au moins un thread vivant.

Par ailleurs, les actions suivants tuent le processus avec tous ses threads :

- un appel d'`exit` (par n'importe quel thread);
- le thread principal termine la fonction `main` (ce qui appelle `exit` implicitement) ;
- le processus ou l'un de ses threads re**c**coit un signal terminant.

Threads et Signaux

La relation entre threads et signaux a très évolué au fil des années. Le suivant est la spécification POSIX (qui n'est pas respectée par toutes les implémentations) :

La disposition des signaux (Term, Ign, etc) est *par processus*, c'est à dire partagé par tous les threads.

Du coup un signal terminant (p.ex. SIGINT par défaut, une violation de mémoire SIGSEGV etc) terminent le processus à part entier.

Chaque thread possède son propre masque de bloquage.

Un signal peut être envoyé à un process (*process-directed signal*). Si la disposition du signal n'est pas "Ignore", il sera livré à un thread quelconque qui ne bloque pas ce signal. Si un tel thread n'existe pas, le signal reste 'en attente'.

Un signal peut être envoyé à un thread (*thread-directed signal*). Dans ce cas, le signal ne peut être livré qu'à ce thread. Si ce thread le bloque actuellement, le signal reste en attente.

`pthread_kill` envoi des signaux à un thread..

Avantages et inconvenients

Advantages des threads par rapport aux processus :

- Communication entre threads plus faciles (par mémoire partagée au lieu des fichiers/tubes/signaux).
- Création des threads moins couteux pour le système.
- Bref, c'est plus efficace.

Inconvenients :

- Moins sécure - une erreur dans un seul thread peut tuer le processus entier.
- Certains appel système ne sont pas thread-safe, deux threads ne peuvent pas s'en servir simultanément (voir *pthreads(7)* pour une liste).
- Attention lors des accès mémoire (*situations de compétition*) !