Suspendre l'exécution d'un programme Unix pendant un temps précis ?

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 30 septembre 2014. Dernière mise à jour le 2 octobre 2014

https://www.bortzmeyer.org/sleep-unix.html

Supposons que vous développiez en C sur Unix et que vous deviez suspendre l'exécution du programme pendant exactement N [Caractère Unicode non montré [1]]s. Par exemple, vous voulez envoyer des paquets sur le réseau à un rythme donné. La réaction immédiate est d'utiliser `sleep`. Mais il y a en fait plein de pièges derrière ce but a priori simple.

Le plus évident est que `sleep` prend en argument un nombre entier de secondes :

```
unsigned int sleep(unsigned int seconds);
```

Sa résolution est donc très limitée. Qu'à cela ne tienne, se dit le programmeur courageux, je vais passer à `usleep` :

```
int usleep(useconds_t usec);
```

Celui-ci fournit une résolution exprimée en [Caractère Unicode non montré]s et, là, on va pouvoir attendre pendant une durée très courte. Testons cela en lançant un chronomètre avant l'appel à `usleep()` et en l'arrêtant après, pour mesurer le temps réellement écoulé :

```
% ./usleep 1000000
1 seconds and 67 microseconds elapsed
```

1. Car trop difficile à faire afficher par LaTeX

OK, pour une attente d'une seconde, le résultat est à peu près ce qu'on attendait. Mais pas à la [Caractère Unicode non montré]s près. Le noyau est un Linux, système multitâche préemptif et des tas d'autres tâches tournaient sur la machine, le noyau a donc d'autres choses à faire et un programme ne peut pas espérer avoir une durée d'attente parfaitement contrôlée. Ici, l'erreur n'était que de 0,0067 %. Mais si je demande des durées plus courtes :

```
% ./usleep 100
0 seconds and 168 microseconds elapsed
```

J'ai cette fois 68 % d'erreur. La durée écoulée en trop (le temps que l'ordonnanceur Linux remette mon programme en route) est la même mais cela fait bien plus mal sur de courtes durées. Bref, si `usleep()` a une résolution théorique de la microseconde, on ne peut pas espérer avoir d'aussi courtes durées d'attente :

```
% ./usleep 1
0 seconds and 65 microseconds elapsed
```

Avec une telle attente minimale, un programme qui, par exemple, enverrait à intervalles réguliers des paquets sur le réseau serait limité à environ 15 000 paquets par seconde. Pas assez pour certains usages.

Là, le programmeur va lire des choses sur le Web et se dire qu'il faut utiliser `nanosleep` :

```
struct timespec
  {
    __time_t tv_sec; /* Seconds.  */
    long int tv_nsec; /* Nanoseconds.  */
  };

int nanosleep(const struct timespec *req, struct timespec *rem);
```

Celui-ci permet d'exprimer des durées en nanosecondes, cela doit vouloir dire qu'il peut faire mieux que `usleep()`, non ?

```
% ./nsleep 1000
0 seconds and 67 microseconds elapsed
```

Eh bien non. Le problème n'est pas dans la résolution de la durée passée en argument, il est dans l'ordonnanceur de Linux. (À noter qu'il existe d'autres bonnes raisons d'utiliser `nanosleep()` plutôt que `usleep()`, liées au traitement des signaux.) La seule solution est donc de changer d'ordonnanceur. Il existe des tas de mécanismes pour cela, allant jusqu'à la recompilation du noyau avec des options davantage ≪ temps réel ≫. Ici, on se contentera d'appeler `sched_setscheduler()` qui permet de choisir un nouvel ordonnanceur. Attention, cela implique d'être root, d'où le changement d'invite dans les exemples :

```
# ./nsleep-realtime 1000
0 seconds and 16 microseconds elapsed
```

Le noyau utilisé n'est pas réellement temps-réel mais, en utilisant l'ordonnanceur `SCHED_RR`, on a gagné un peu et les durées d'attente sont plus proches de ce qu'on demandait. (Les gens qui veulent de la latence vraiment courte, par exemple pour le jeu vidéo, utilisent des noyaux spéciaux.)

Ce très court article ne fait qu'effleurer le problème compliqué de l'attente sur un système Unix. Il existe par exemple d'autres façons d'attendre (attente active, `select` sur aucun fichier, mais avec délai d'attente maximal, dormir mais jusqu'à un certain moment comme illustré par ce programme `<https://github.com/farsightsec/libmy/blob/master/my_rate.c#L123>`, etc). Je vous conseille la lecture de « "*High-resolution timing*" `<http://www.tldp.org/HOWTO/IO-Port-Programming-4.html>` » et de « "*Cyclictest*" `<https://rt.wiki.kernel.org/index.php/Cyclictest>` » ainsi évidemment que celle de StackOverflow `<http://stackoverflow.com/search?q=nanosleep>`. Le code source des programmes utilisé ici est (en ligne sur `https://www.bortzmeyer.org/files/usleep.c`) et (en ligne sur `https://www.bortzmeyer.org/files/nsleep.c`). Plus compliqué, le programme (en ligne sur `https://www.bortzmeyer.org/files/test_rt4.c`) utilise des "*timers*", des signaux, bloque le programme sur un seul processeur, interdit le "*swap*" et autres trucs pour améliorer la précision.

Merci à Michal Toma pour l'idée, à Laurent Thomas pour son code et à Robert Edmonds pour plein de suggestions.