Partage des ressources (processeurs et mémoire)

1 Processeurs

Important : les machines sont bicœur, c'est-à-dire que le processeur est en fait double. Pour pouvoir observer l'ordonnancement en conditions habituelles (un seul processeur), lancez la commande

`~sthibault/bin/sched_setaffinity 0`

ce qui ouvre un nouveau shell (avec un prompt `cpu0$`) dont tous les fils tourneront sur le processeur 0 seulement. Sauf indication contraire, lancez alors **systématiquement** les programmes du TP dans ce genre de shell.

1.1 Un cpu burner

Compilez et lancez le programme `burn` suivant :

```
int main(void) { while(1); }
```

Dans un autre terminal, lancez `top`. Vous pouvez constater que `burn` occupe presque 100% du cpu.

Lancez un autre `burn`, que se passe-t-il ? Généralisez.

Observez également la première valeur du «`load average`» monter progressivement jusqu'à 2. Que représente-t-elle donc ?

1.2 Un peu de gentillesse dans ce monde de brutes

Si vous ne l'avez pas déjà fait, tuez les burners précédents.

On voudrait que notre cpu burner n'occupe tout de même pas tant de % cpu. Au lieu de lancer simplement `./burn`, lancez `nice -n 19 ./burn`. Remarquez que dans `top`, il a effectivement un champ `NI` égal à 19. Remarquez aussi que son champ `PR` est plus fort (*i.e.* priorité plus faible)[1] Est-ce que le % cpu utilisé a diminué ? Pourquoi ?

1. La relation entre indice de gentillesse (nice) et priorité n'est en fait pas simple : elle fait également intervenir la notion d'interactivité

Observez les pourcentages de cpu utilisés par un burner très gentil et un burner normal tournant en même temps. Utilisez la commande `renice 19 <pid>` pour, « à chaud », rendre le burner normal gentil. Que se passe-t-il ? Peut-on utiliser `renice` pour enlever de la gentillesse à un burner, et donc le rendre plus agressif ?

Lancez en même temps des burner d'indices de gentillesse 0, 3, 6, 9, 12, 15 et 18. Remarquez les % cpu respectifs.

Regardez de nouveau le `load average`, confirmez vos soupçons.

1.3 Chez vos voisins

Utilisez `su` pour lancer des cpus burners sous votre identité chez votre voisin. L'équilibrage selon les gentillesses est-il le même ?

1.4 Et donc ?

Sous un système Linux, par défaut, il n'y a donc pas de protection sur le temps CPU utilisé. Si un utilisateur lance des dizaines de processus qui consomment du CPU, les quelques autres processus n'ont alors que peu de % CPU, *y compris ceux de root, a priori*. À noter l'existence d'une option du noyau `CONFIG_FAIR_USER_SCHED` , qui répartit d'abord le % CPU entre utilisateurs, puis redistribue ces temps parmis les processus de chaque utilisateur, étant ainsi plus équitable.

Cependant, puisqu'un utilisateur normal n'a pas le droit de diminuer sa gentillesse, un administrateur peut configurer dans `/etc/security/limits.conf` pour donner des indices plus grands à certains : tant qu'il sont seuls sur une machine ils peuvent travailler à pleine vitesse. Si quelqu'un avec un indice plus faible se connecte, ce dernier aura priorité (dans une certaine mesure).

Notez dans `ulimit -a` la présence de l'option `-t` (et l'équivalent dans `limits.conf`). Elle permet de limiter le temps CPU autorisé. En utilisant l'option `-S`, essayez avec `burn`. (Attention, sans l'option `-S`, vous établiriez une limite "dure" que vous ne pourriez plus annuler)

Note : la documentation d'ulimit est dans `man bash`

1.5 Fork bombs

Nota : les systèmes en salle de TD sont peu protégées contre les fork-bombs (cf la ligne `max user processes` produite par la commande `ulimit -a`). Ce qui suit rendra donc la machine inutilisable si vous ne faites pas attention (appuyez sur `ctrl-alt-backspace` pour redémarrer). Avant de lancer le programme ci-dessous, utilisez `ulimit -Su 128` pour vous limiter en nombre de processus.

```
int main(void) { while(1) printf("%d\n", fork()); }
```

Que fait le programme ci-dessus ?

D'autres variantes amusantes sont possibles : `while(fork());`, `while(!fork());`,... Quelles différences y a-t-il avec la première version ? Quelle variante est la plus difficile à tuer ?

Évitez de lancer une fork-bomb sur une machine partagée de l'ENSEIRB, telle que `ssh` : vous ne feriez que ne plus pouvoir y lancer de processus (elle est plus protégée) !...

1.6 cgroups

Les Linux récents fournissent un moyen de contrôler un peu mieux l'utilisation de % CPU : les cgroups. Ils servent de conteneurs dans lesquels on peut par exemple mettre les sessions des utilisateurs, l'ordonnanceur de Linux pouvant alors répartir le % CPU équitablement entre les cgroups avant de le répartir équitablement à l'intérieur d'un cgroup.

1.7 Et le multicœur ?

C'est essentiellement la même chose, sauf que l'on a autant de fois plus de centaines de % de CPUs à partager. Sur vos machines par exemple, le total est donc 200%. Essayez de lancer 3 processus `burn` sans utiliser `sched_setaffinity` (tapez 1 dans le `top` pour avoir le détail par processeur), est-ce que Linux parvient à atteindre 200/3 ? Pourquoi ?

Il est assez courant de fixer l'exécution de certains processus sur certains CPUs, pour bien découper l'utilisation de la machine par les différents utilisateurs. On l'a fait de manière souple dans ce TP à l'aide de `sched_setaffinity`, l'administrateur peut également l'imposer à l'aide de `cpuset`. Essayez de fixer deux `burn` sur un CPU et plusieurs `burn`s sur un autre CPU, constatez que chaque série se partage bien son propre CPU.

2 Mémoire

2.1 Quotas mémoire

Regardez la sortie de `ulimit -a` pour connaître les limites par défaut de votre poste en terme de mémoire (la documentation d'ulimit est dans `man bash`). Ce n'est pas du tout évident de définir des quotas d'utilisation effective de la mémoire (RSS, *resident set size*) : le code de la libc, typiquement, est utilisé par tous les processus, mais n'est mis en mémoire qu'une seule fois pour tous ces processus, quels que soient leur propriétaires. Il n'est donc même pas évident de pouvoir affirmer ce que chacun consomme, puisqu'il y a des partages !

Il est par exemple à noter que `ulimit -m` n'a pas d'effet sous Linux, essayez !

L'utilisation virtuelle de la mémoire (et donc les mallocs et autres mmaps géants que l'on pourrait vouloir faire, est par contre bien mesurable, et `ulimit -v` fonctionne bien, essayez.

2.2 Mémoire verrouillée

Un programme peut avoir des secrets à cacher : mot de passe, clé privée, etc. Or si le système se retrouve à manquer de mémoire, il va vouloir mettre des morceaux de processus dans le *swap* pour libérer de la mémoire physique. Le problème est que du coup, les secrets d'un processus peuvent se retrouver ainsi écrits tels quels sur le disque dur !

Unix fournit la fonction `mlock()` pour pallier ce problème : elle permet de verrouiller certaines zones du processus contre la mise en *swap*. Notez dans `ulimit` que vous avez une limitation de la quantité de mémoire ainsi verrouillée.

3 Bonus : Faille de sécurité de l'HyperThreading

Le Symmetric MultiThreading (SMT, HyperThreading est le mot marketing d'Intel) est une technologie un peu particulière : au lieu d'avoir plusieurs cœurs pour lancer différents programmes en parallèle, on lance différents programmes en même temps sur un même cœurs, les différents programmes se partageant donc les ressources du cœur : cache, unité de calcul, etc. Cela permet alors ce qu'on appelle un *canal caché* (*covert channels* : puisque les processus bataillent finement pour utiliser les ressources, un processus peut en observer finement un autre simplement en observant de quelle manière il parvient à utiliser les resources.

Ouvrez le document `http://www.daemonology.net/papers/htt.pdf`, vous pouvez notamment lire les parties 3 (*L1 cache missing*) et 4 (*L2 cache missing*), mais surtout la partie 5 (*OpenSSL key theft*), avec la figure 2 qui montre le genre de mesure que l'on peut obtenir.