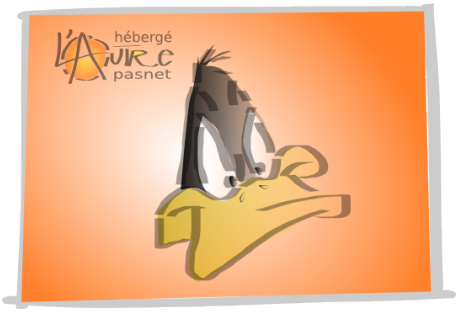
http://daffyduke.lautre.net/spip/?Linux-Memory-Q-A



Linux Memory Q&A

- 1- Blog-Notes - Au boulot -

Date de mise en ligne : lundi 12 septembre 2011

Question (Thread Stack Size)

A quoi correspond exactement la stack size d'un thread ? Comment la dimensionner au mieux autrement que par le test ?

Réponse

La stack size d'un thread fonctionne de la même façon que la stack size d'un processus (variables locales, paramètres de fonctions, adresses de retour, …)

Le dimensionnement correct de la TSS ne peut se faire que si l'on connaît parfaitement l'application. Lorsque cela n'est pas possible il reste l'empirisme.

Question (Mémoire et Stack Size)

Lorsque l'on lance bcp de threads, sous linux 2.6, le prg quitte avec un « out of memory ». Ok, la stack size est fixée à 8192 Ko ce qui entraîne le fait qu'il n'y a plus de mémoire dispo…. MAIS pourquoi alors, ce problème n'arrive t'il pas en 2.4 même si la stack size est elle aussi fixée à 8192Ko. En effet, lorsqu'en 2.6, un prg tout bête ne peut lancer que 382 threads avec une stack size par défaut de 8192ko avant de se tanker avec un beau out of memory, le même prg arrive à en lancer plus de 6100 sous un kernel 2.4 (cette limite est d'ailleurs sans doute due au ulimit –u puisqu'on le sait, un thread = un process sous 2.4).

Réponse

LC1 = kernel 2.4 + glibc 2.2 : l'implémentation des threads est faite par l'API linuxthreads-0.9.

LC4 = kernel 2.6 + glibc 2.3 : l'implémentation des threads est faite par les NPTL (Native POSIX Threads Library) + TLS (Thread Local Storage).

Par défaut sous LC1 la thread stack size est à 512ko – 1 page de 4 Ko (pour la structure thread kernel : descripteur du processus etc…) 512 * 1024 – 4 * 1024 = 520192

C'est vérifié ici à l'aide d'un pthread_attr_getstacksize (&attr, &stacksize).

```
 [root@lfslc1 /distrib/tmp/poluc_dev]# ./a.out
Default stack size = 520192
```

L'API linuxthreads déconseille l'utilisation de « pthread_attr_setstacksize », qui d'ailleurs ne fonctionne pas :

```
 [root@lfslc1 /distrib/tmp/poluc_dev]# ./a.out
Default stack size = 520192
Amount of stack needed per thread = 631072
Creating threads with stack size = 631072 bytes
Thread 0: stack size = 520192 bytes
```

Mon code récupère la TSS, la modifie, je crée ensuite autant de threads que possible jusqu'à me faire gentiment remercier par le OOM killer.

On constate ici que le « pthread_attr_setstacksize » ne renvoie pas d'erreur mais le système s'en moque puisqu'après création du premier thread, sa TSS est toujours à la valeur par défaut.

La TSS est initialisé durant la compilation dans le fichier glibc-2.2.5/linuxthreads/internals.h.

```
#define STACK_SIZE  (2 * 256 * 1024)
```

Le « pthread_attr_setstacksize » permet de diminuer la taille d'un thread mais pas de l'augmenter au-delà de ce seuil.

Maintenant si l'on compare LC1/LC4 en créant des threads de taille identique :

```
 LC1
Thread 6134: stack size = 520808 bytes
Thread 6132: stack size = 520808 bytes
Thread 6135: stack size = 520808 bytes
ERROR; return code from pthread_create() is 11
[root@lfslc1 /distrib/tmp/poluc_dev]#
```

```
 LC4
Thread 6135: stack size = 520808 bytes
Thread 6136: stack size = 520808 bytes
ERROR; return code from pthread_create() is 12
[root@lfslc4 /distrib/tmp/poluc_dev]#
```

Sous LC1 on crée autant de threads que sous LC4 et le comportement est strictement identique. (OOM killer, memory trace dans les logs etc…)

Question (Swap)

A quoi sert la mémoire virtuelle (ou swap ?) ? D'après un ulimit, elle est « illimitée » mais en fait pourquoi, le system ne l'utilise pas préférant un « out of memory ».

Réponse

La swap peut être limité par environnement, ce que permet le ulimit. En revanche, il n'est pas possible de swapper au-delà de la partition système déclarée.

La mémoire virtuelle est utilisée lorsque la RAM physique déborde, auquel cas, dés que le kernel dispose de son timeslice, il ordonnance ses processus et en passent certain en swap afin de libérer de l'espace RAM pour « l'élu ».

La LC1 et la LC4 swappent toutes les 2 correctement lorsque cela est nécessaire.

Afin d'expliquer le phénomène que tu décris (pas de swap utilisée), regardons de plus près nos 6136 threads allouant chacun 520808 bytes.

```
 [root@lfslc1 /distrib/tmp/poluc_dev]# echo "6136*520808/1024/1024"
| bc
3047
```

Ce qui fait à peu près 3 Go d'espace alloué. Sur une archi 32 bits, l'espace maximum d'adressage alloué à un processus est de 3Go. Le dernier Go est alloué au kernel.

Cette limite est inhérente à l'architecture matérielle (2^32 = 4294967296 adresses au total), à ne pas confondre avec le ZONE_HIGHMEM dispo dans les kernels…

Donc tu atteins les limites systèmes d'espace d'adressage de ton processus avant d'être autorisé à swapper.

Pour s'en convaincre voici ce qu'il se passe sur LC4 si tu augmentes la TSS :

(Notons que le code utilise l'espace alloué pour se rapprocher de conditions réelles, il ne contente pas d'allouer de l'espace ce qui ne serait pas pertinent.)

```
 Thread 203: stack size = 8888608 bytes
Thread 204: stack size = 8888608 bytes
Killed
[root@lfslc4 /distrib/tmp/poluc_dev]#
```

La TSS est plus importante, on ne crée que 204 threads et dans ce cas nous faisons un « out of memory ».

La limite système inhérente à un adressage sur 32 bits n'est pas atteinte :

```
 [root@lfslc1 /distrib/tmp/poluc_dev]# echo "8888608*204/1024/1024"
| bc
1729
[root@lfslc1 /distrib/tmp/poluc_dev]#
```

Et nous swappons :

```
 top - 19:19:11 up  9:36,  9 users,  load average: 12.70, 5.47, 2.83
Tasks:  55 total,  3 running,  51 sleeping,  0 stopped,  1
zombie
Cpu(s):  1.3% us, 73.9% sy,  0.0% ni,  0.0% id, 24.2% wa,  0.3% hi,
0.3% si
Mem:    256992k total,  253256k used,    3736k free,      60k
buffers
Swap:  524280k total,  524272k used,        8k free,    2712k
cached
PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND

6868 root      18  0 1740m 233m  320 D 63.5 93.1  0:05.96 a.out
```

NB : Le « top » est en accord avec notre calcul de 1729Mo utilisé par le processus a.out.

L'exception de la JVM

Sun a inclus une limite de 2Go de mémoire max dans sa JVM 32bits.

Il n'est donc pas possible d'allouer plus de mémoire à un process java en 32bits.

Cette barrière est levée avec une JVM 64bits.