Linux memory management summary

Every part of process address space is some sort of mapping.

Memory accounting

ps terminology:

- %mem (pmem) – ratio of the process’s resident set size to the physical memory on the machine, expressed as a percentage;
- rss (rsz) – resident set size, the non-swapped physical memory that a task has used;
- size – approximate amount of swap space that would be required if the process were to dirty all writable pages and then be swapped out;
- vsz – virtual memory size of the process;

top terminology:

- VIRT – virtual Image; the total amount of virtual memory used by the task. It includes all code, data and shared libraries plus pages that have been swapped out. VIRT = SWAP + RES;
- SWAP – the swapped out portion of a task’s total virtual memory image;
- RES – resident size; the non-swapped physical memory a task has used. RES = CODE + DATA;
- CODE – code size; the amount of physical memory devoted to executable code, also known as the ’text resident set’ size or TRS;
- DATA – Data+Stack size; the amount of physical memory devoted to other than executable code, also known as the ’data resident set’ size or DRS;
- SHR – Shared Mem size; the amount of shared memory used by a task. It simply reflects memory that could be potentially shared with other processes;

/proc terminology:

- size – total size of the mapped regions, regardless of what is mapped and whether it is accessible; size >= rss + swap;
- rss – resident set size; size of physical memory, currently mapped into the region. rss >= shared_clean + shared_dirty + private_clean + private_dirty;
- pss – proportional set size; size of resident set, where each page shared by N processes is counted as 1/N'th part of page;
- shared_clean – size of memory actually shared (mapped into 2+ processes) and haven't been written to;
- shared_dirty – size of memory actually shared (mapped into 2+ processes) and changed;
- private_clean – size of memory mapped only to this process and haven't been written to;
- private_clean – size of memory mapped only to this process and changed;
- referenced – size of data that were accessed (either read or written);
- swap – size of region data that is currently in swap file;

- Charging memory to processes
- total_vm

Information sources

Per-process data:

- /proc/<PID>/maps – process memory map;
- /proc/<PID>/smaps – detailed process memory map;
- /proc/<PID>/statm – process memory usage summary;
- /proc/<PID>/status – process status;
- /proc/<PID>/oom_adj
- /proc/<PID>/oom_score

System-wide data:

- /proc/…

What all these memory types are

Memory is always mapped from some source. And after being mapped it is backed by some storage. There are the following cases:

- Shared named mapping – mapped from file and backed by this file;
- Private named mapping – mapped from file and backed by swap/physical memory;
- Shared anonymous mapping – mapped from /dev/zero and backed by swap/physical memory;
- Private anonymous mapping – mapped from /dev/zero and backed by swap/physical memory;

- Clean vs dirty
  - Clean pages for the given process are those, which haven't been modified by that process. They can always be restored from the mapping source.
  - Dirty pages are those modified by the process. They can only be restored from backing storage.
- Shared vs private
  - Shared mapping is synced back to the source and may always be restored from the source. Changes become visible in all other shared mappings of the file.
  - Changes made to private mapping are stored in separate backing storage and are never visible outside the process.
- Named vs anonymous
  - Named mapping is backed by the file. Clean pages may be restored from this file.
  - Anonymous mapping is backed either by the swap space or by the physical memory. Clean pages may be restored from /dev/zero.
- Resident vs all other
  - Resident is what currently in physical memory;
  - If not resident, it may be swapped or not backed by whatever storage at all.

What to expect

- of heap usage
  - private_dirty or swap size of [heap] region grows;
  - size of [heap] region grows;
- of file mapping

- of anonymous mapping
- of stack
- of child processes
    - what's duplicated in child, what in parent?
- of threads
    - tls?

OOM killing

When this happens

Who gets killed

Kernel threads or Init process never get killed by this mechanism.

For other processes we count their “score” and kill one that have maximal score. Current score for the given process may be read from /proc/<PID>/oom_score.

```
* The formula used is relatively simple and documented inline in the
* function. The main rationale is that we want to select a good task
* to kill when we run out of memory.
*
* Good in this context means that:
* 1) we lose the minimum amount of work done
* 2) we recover a large amount of memory
* 3) we don't kill anything innocent of eating tons of memory
* 4) we want to kill the minimum amount of processes (one)
* 5) we try to kill the process the user expects us to kill, this
*    algorithm has been meticulously tuned to meet the principle
*    of least surprise ... (be careful when you change it)
```

Process that currently executes swapoff system call is always the first candidate to be oom-killed with score of ULONG_MAX.

In other cases process score is counted as folows:

1. The memory size of the process is the basis for the badness;
    - points = total_vm
2. Take child processes into an account. Processes which fork a lot of child processes are likely a good choice. We add half the vmsize of the children if they have an own mm. This prevents forking servers to flood the machine with an endless amount of children. In case a single child is eating the vast majority of memory, adding only half to the parents will make the child our kill candidate of choice;
    - for each child process with own address space: points += (1 + child→total_vm/2)
3. Take process lifetime into an account. *(CPU time is in tens of seconds and run time is in thousands of seconds)*;
    - cpu_time = (user_time + system_time) / 8; *(that is, consumed cpu time in user and kernel*

mode, as reported by e.g. time)
    - run_time = (real time elapsed since process start) / 1024;
    - if (cpu_time > 0) points /= int_sqrt(cpu_time);
    - if (run_time > 0) points /= int_sqrt(int_sqrt(run_time));
4. Rise score for niced processes. *(Niced processes are most likely less important, so double their badness points)*;
    - if (task_nice > 0) points *= 2;
5. Lower score for superuser processes. *(Superuser processes are usually more important, so we make it less likely that we kill those)*;
    - if (has_capability_noaudit(p, CAP_SYS_ADMIN) || has_capability_noaudit(p, CAP_SYS_RESOURCE)) points /= 4;
6. Lower score for a process that have direct hardware access. *(We don't want to kill a process with direct hardware access. Not only could that mess up the hardware, but usually users tend to only have this flag set on applications they think of as important)*;
    - if (has_capability_noaudit(p, CAP_SYS_RAWIO)) points /= 4;
7. Finally adjust the score by oom_adj;
    - if (oom_adj > 0) points «= oom_adj; *(if points == 0 before shift, points = 1)*
    - if (oom_adj < 0) points »= -oom_adj;

How to control OOM-killer

The following parameter may be tuned in /proc on per-process basis:

- /proc/<PID>/oom_adj – signed decimal number. Positive values rise process oom_score, negative values make it lower.

The following parameters may be tuned through sysctl interface or /etc/sysctl.conf:

- vm.panic_on_oom – panic in case of OOM, instead of trying to kill some processes;
- vm.oom_kill_allocating_task – try first to kill task that issued request for memory that caused OOM condition;
- vm.oom_dump_tasks – dump memory summary, stack of the process caused oom and table of processes to log at before oom-killing;
- vm.would_have_oomkilled

Memleak detection

Direct memleak evidences

cat /proc/<PID>/smaps

From:
http://wiki.osll.ru/ - **Open Source & Linux Lab**

Permanent link:
http://wiki.osll.ru/doku.php/etc:users:jcmvbkbc:linux-mm?rev=1247494327

Last update: **2009/07/13 18:12**