Published on **Linux DevCenter** (http://www.linuxdevcenter.com/)
See this if you're having trouble printing code examples

When Linux Runs Out of Memory

by Mulyadi Santosa
11/30/2006

Perhaps you rarely face it, but once you do, you surely know what's wrong: lack of free memory, or Out of Memory (OOM). The results are typical: you can no longer allocate more memory and the kernel kills a task (usually the current running one). Heavy swapping usually accompanies this situation, so both screen and disk activity reflect this.

At the bottom of this problem lie other questions: how much memory do you want to allocate? How much does the operating system (OS) allocate for you? The basic reason of OOM is simple: you've asked for more than the available virtual memory space. I say "virtual" because RAM isn't the only place counted as free memory; any swap areas apply.

Exploring OOM

To begin exploring OOM, first type and run this code snippet that allocates huge blocks of memory:

```
#include <stdio.h>
#include <stdlib.h>

#define MEGABYTE 1024*1024

int main(int argc, char *argv[])
{
        void *myblock = NULL;
        int count = 0;

        while (1)
        {
                myblock = (void *) malloc(MEGABYTE);
                if (!myblock) break;
                printf("Currently allocating %d MB\n", ++count);
        }

        exit(0);
}
```

Compile the program, run it, and wait for a moment. Sooner or later it will go OOM. Now compile the next program, which allocates huge blocks and fills them with 1:

```
#include <stdio.h>
#include <stdlib.h>

#define MEGABYTE 1024*1024

int main(int argc, char *argv[])
{
```

```
        void *myblock = NULL;
        int count = 0;

        while(1)
        {
                myblock = (void *) malloc(MEGABYTE);
                if (!myblock) break;
                memset(myblock,1, MEGABYTE);
                printf("Currently allocating %d MB\n",++count);
        }
        exit(0);

}
```

Notice the difference? Likely, program A allocates more memory blocks than program B does. It's also obvious that you will see the word "Killed" not too long after executing program B. Both programs end for the same reason: there is no more space available. More specifically, program A ends gracefully because of a failed `malloc()`. Program B ends because of the Linux kernel's so-called OOM killer.

The first fact to observe is the amount of allocated blocks. Assume that you have 256MB of RAM and 888MB of swap (my current Linux settings). Program B ended at:

```
Currently allocating 1081 MB
```

On the other hand, program A ended at:

```
Currently allocating 3056 MB
```

Where did A get that extra 1975MB? Did I cheat? Of course not! If you look closer on both listings, you will find out that program B fills the allocated memory space with 1s, while A merely simply allocates without doing anything. This happens because Linux employs deferred page allocation. In other words, allocation doesn't actually happen until the last moment you really use it; for example, by writing data to the block. So, unless you touch the block, you can keep asking for more. The technical term for this is *optimistic memory allocation*.

Checking */proc/<pid>/status* on both programs will reveal the facts. Here's program A:

```
$ cat /proc/<pid of program A>/status
VmPeak:  3141876 kB
VmSize:  3141876 kB
VmLck:         0 kB
VmHWM:     12556 kB
VmRSS:     12556 kB
VmData:  3140564 kB
VmStk:        88 kB
VmExe:         4 kB
VmLib:      1204 kB
VmPTE:      3072 kB
```

Here's program B, shortly before the OOM killer struck:

```
$ cat /proc/<pid of program B>/status
VmPeak:  1072512 kB
VmSize:  1072512 kB
VmLck:         0 kB
VmHWM:    234636 kB
VmRSS:    204692 kB
VmData:  1071200 kB
VmStk:        88 kB
VmExe:         4 kB
VmLib:      1204 kB
VmPTE:      1064 kB
```

VmRSS deserves further explanation. RSS stands for "Resident Set Size." It explains how many of the allocated blocks owned by the task currently reside in RAM. Also note that before B reaches OOM, swap usage is almost 100 percent (most of the 888MB), while A uses no swap at all. It's clear that `malloc()` itself did nothing more than just preserve a memory area, nothing else.

Another question also arises. "Even without touching the pages, why is the allocation limit 3056MB?" This exposes an unseen limit. For every application in a 32-bit system, there is 4GB of address space available for usage. The Linux kernel usually splits the linear address to provide 0 to 3GB for user space and 3GB to 4GB for kernel space. User space is a room where a task can do anything it wants, while kernel space is solely for the kernel. If you try to cross this 3GB border, you will get a segmentation fault.

The conclusion is that OOM happens for two technical reasons:

1. No more pages are available in the VM.
2. No more user address space is available.
3. Both #1 and #2.

(Side note: There is a kernel patch that gives the whole 4GB to userspace, at the cost of some context-switching.)

Thus the strategies to prevent those circumstances are:

1. Know how large the user address space is.
2. Know how many pages are available.

When you ask for a memory block, usually by using `malloc()`, you're asking the runtime C library whether a preallocated block is available. This block's size must *at least* equal the user request. If there is already a memory block available, `malloc()` will assign this block to the user and mark it as "used." Otherwise, `malloc()` must allocate more memory by extending the heap. All requested blocks go in an area called the *heap*. Do not confuse it with the stack, because the stack stores local variable and function return addresses. These two sections have different jobs.

Where is the heap located in the address space? The process address map can tell you exactly where:

```
$ cat /proc/self/maps
0039d000-003b2000 r-xp 00000000 16:41 1080084    /lib/ld-2.3.3.so
003b2000-003b3000 r-xp 00014000 16:41 1080084    /lib/ld-2.3.3.so
003b3000-003b4000 rwxp 00015000 16:41 1080084    /lib/ld-2.3.3.so
003b6000-004cb000 r-xp 00000000 16:41 1080085    /lib/tls/libc-2.3.3.so
004cb000-004cd000 r-xp 00115000 16:41 1080085    /lib/tls/libc-2.3.3.so
004cd000-004cf000 rwxp 00117000 16:41 1080085    /lib/tls/libc-2.3.3.so
004cf000-004d1000 rwxp 004cf000 00:00 0
08048000-0804c000 r-xp 00000000 16:41 130592     /bin/cat
0804c000-0804d000 rwxp 00003000 16:41 130592     /bin/cat
0804d000-0806e000 rwxp 0804d000 00:00 0          [heap]
b7d95000-b7f95000 r-xp 00000000 16:41 2239455    /usr/lib/locale/locale-archive
b7f95000-b7f96000 rwxp b7f95000 00:00 0
b7fa9000-b7faa000 r-xp b7fa9000 00:00 0          [vdso]
bfe96000-bfeab000 rw-p bfe96000 00:00 0          [stack]
```

This is an actual address space layout shown for `cat`, but you may get different results. It is up to the Linux kernel and the runtime C library to arrange them. Notice that recent Linux kernel versions (2.6.x) kindly label the memory area, but don't completely rely on them.

The heap is basically free space not already given for program mapping and stack; thus, it narrows down the available address space. It's not a full 3GB, but it's 3GB minus everything else that's mapped. The bigger your program's code segment is, the less space you have for heap. The more dynamic libraries you link into your program, the less space you get for the heap. This is important to remember.

How does the map for program A look when it can't allocate more memory blocks? With a trivial change to pause the program (see *loop.c* and *loop-calloc.c*) just before it exits, the final map is:

```
0009a000-0039d000 rwxp 0009a000 00:00 0 ---------> (allocated block)
0039d000-003b2000 r-xp 00000000 16:41 1080084    /lib/ld-2.3.3.so
003b2000-003b3000 r-xp 00014000 16:41 1080084    /lib/ld-2.3.3.so
003b3000-003b4000 rwxp 00015000 16:41 1080084    /lib/ld-2.3.3.so
003b6000-004cb000 r-xp 00000000 16:41 1080085    /lib/tls/libc-2.3.3.so
004cb000-004cd000 r-xp 00115000 16:41 1080085    /lib/tls/libc-2.3.3.so
004cd000-004cf000 rwxp 00117000 16:41 1080085    /lib/tls/libc-2.3.3.so
004cf000-004d1000 rwxp 004cf000 00:00 0
005ce000-08048000 rwxp 005ce000 00:00 0 ---------> (allocated block)
08048000-08049000 r-xp 00000000 16:06 1267       /test-program/loop
08049000-0804a000 rwxp 00000000 16:06 1267       /test-program/loop
0806d000-b7f62000 rwxp 0806d000 00:00 0 ---------> (allocated block)
b7f73000-b7f75000 rwxp b7f73000 00:00 0 ---------> (allocated block)
b7f75000-b7f76000 r-xp b7f75000 00:00 0          [vdso]
b7f76000-bf7ee000 rwxp b7f76000 00:00 0 ---------> (allocated block)
bf80d000-bf822000 rw-p bf80d000 00:00 0          [stack]
bf822000-bff29000 rwxp bf822000 00:00 0 ---------> (allocated block)
```

Six Virtual Memory Areas, or VMAs, reflect the memory request. A VMA is a memory area that groups pages with the same access permission and/or the same backing file. VMAs can exist anywhere within user space, as long as that space is available.

Now you might think, "Why six? Why not a single big VMA containing all blocks?" There are two reasons. First, it is often impossible to find such a big "hole" to coalesce the blocks into a single VMA. Second, the program does not ask to allocate that approximately 3GB block all at once, but piece by piece. Thus, the `glibc` allocator has complete freedom to arrange the memory however it wants.

Why do I mention available pages? Memory allocation occurs in page-sized granularity. This is not a limit of the operating systems, but a feature of the Memory Management Unit (MMU) itself. Pages have various sizes, but the normal setting for x86 is 4K. You can discover the page size manually by using `getpagesize()` or `sysconf()` (with the `_SC_PAGESIZE` parameter) `libc` functions. The `libc` allocator manages each page: slicing them into smaller blocks, assigning them to processes, freeing them, and so on. For example, if your program uses 4097 bytes total, you need to use two pages, even though in reality the allocator gives you somewhere between 4105 to 4109 bytes.

With 256MB of RAM and no swap, you have 65536 available pages. Is that right? Not really. What you don't see is that some memory areas are in use by kernel code and data, so they're unavailable for any other need. There is also a reserved part of memory for emergencies or high-priority needs. `dmesg` reveals these numbers for you:

```
$ dmesg | grep -n kernel
36:Memory: 255716k/262080k available (2083k kernel code, 5772k reserved,
    637k data, 172k init, 0k highmem)
171:Freeing unused kernel memory: 172k freed
```

`init` refers to kernel code and data that is only necessary for the initialization stage; thus the kernel frees it when it is no longer useful. That leaves 2083 + 5772 + 637 = 8492KB. Practically speaking, 2123 pages are gone from the user's point of view. If you enable more kernel features or insert more kernel modules, you'll use up more pages for exclusive kernel use, so be wise.

Another kernel internal data structure is the page cache. The page cache buffers data recently read from block devices. The more caching work you do, the fewer free pages you actually have--but they are not really occupied, as the kernel will reclaim them when memory is tight.

From the kernel and hardware points of view, these are the important things to remember:

1. There is no guarantee that allocated memory area is physically contiguous; it's only virtually contiguous.

This "illusion" comes from the way address translation works. In a protected mode environment, users always work with virtual addresses, while hardware works with physical addresses. The page directory and page tables translate between these two. For example, two blocks with starting virtual addresses 0 and 4096 could map to the physical addresses 1024 and 8192.

This makes allocation easier, because in reality it is unlikely to always get continuous blocks, especially for large requests (megabytes or even gigabytes). The kernel will look everywhere for free pages to satisfy the request, not just adjacent free blocks. However, it will do a little more work to arrange page tables so that they appear virtually contiguous.

There is a price. Because memory blocks might be non-contiguous, sometimes the L1 and L2 caches go underused. Virtually adjacent memory blocks may be spread across different physical cache lines; this means slowing down (sequential) memory access.

2. Memory allocation takes two steps: first extending the length of memory area and then allocating pages when needed. This is demand paging. During VMA extension, the kernel merely checks whether the request overlaps existing VMA and if the range is still inside user space. By default, it omits the check whether actual allocation can occur.

   Thus it is not strange if your program asks for a 1GB block and gets it, even if in reality you have only 16MB of RAM and 64MB of swap. This "optimistic" style might not please everybody, because you might get the false hope of thinking that there are still free pages available. The Linux kernel offers tunable parameters to control this overcommit behavior.

3. There are two type of pages: anonymous pages and file-backed pages. A file-backed page originates from `mmap()`-ing a file in disk, whereas an anonymous page is the kind you get when doing `malloc()`. It has no relationship with any files at all. When the RAM becomes tight, the kernel swaps out anonymous pages to swap space and flushes file-backed pages to the file to give room for current requests. In other words, anonymous pages may consume swap area while file-backed pages don't. The only exception is for files `mmap()`-ed using the `MAP_PRIVATE` flag. In this case, file modification occurs in RAM only.

   This is where the understanding of swap as RAM extension comes from. Clearly, accessing the page requires bringing it back into RAM.

Inside the Allocator

The real work actually takes place inside the `glibc` memory allocator. The allocator hands out blocks to the application, carving them from the heap that comes (however infrequently) from the kernel.

The allocator is the manager, while the kernel is the worker. With this in mind, it's easy to understand that maximum efficiency comes from a good allocator, not from the kernel.

`glibc` uses an allocator named `ptmalloc`. Wolfram Gloger created it as a modified version of the original `malloc` library created by Doug Lea. The allocator manages the allocated blocks in terms of "chunks." Chunks represent the memory block you actually requested, but not its size. There is an extra header added inside this chunk besides the user data.

The allocator uses two functions to get a chunk of memory from the kernel:

- `brk()` sets the end of the process's data segment.
- `mmap()` creates a new VMA and passes it to the allocator.

Of course, `malloc()` uses these functions only if there are no more free chunks in the current pool.

The decision on whether to use `brk()` or `mmap()` requires one simple check. If the request is equal or larger than `M_MMAP_THRESHOLD`, the allocator uses `mmap()`. If it is smaller, the allocator calls `brk()`. By default, `M_MMAP_THRESHOLD` is 128KB, but you may freely change it by using `mallopt()`.

In the OOM context, how `ptmalloc` frees memory blocks is interesting. Blocks allocated via `mmap()` get freed via an `unmap()` call, and thus become completely released. Freeing blocks allocated via `brk()` means marking them as free, but they remain under the allocator's control. It can reassign free chunks to satisfy another `malloc()` if the request's size is less than or equal to the chunk's size. The allocator can consolidate multiple free chunks, as long as they are adjacent. It may even split a free chunk into smaller chunks to satisfy smaller future requests.

This implies that a free chunk may go abandoned if the allocator cannot fit future requests within it. Failure to coalesce free chunks may also trigger faster OOM. This is usually an indication of moderate to bad memory fragmentation.

Recovery

Once an OOM situation occurs, now what? The kernel will terminate one process for sure. Why kill? This is the only way to stop further memory requests. The kernel can not assume there is a sophisticated mechanism inside the process to stop further requests automatically, so it has no other choice but to kill it.

How does the kernel know exactly which process to kill? The answer lies inside *mm/oom_kill.c* of the Linux source code. This C code represents the so-called OOM killer of the Linux kernel. The function `badness()` give a score to each existing processes. The one with highest score will be the victim. The criteria are:

1. VM size. This is not the sum of all allocated pages, but the sum of the size of all VMAs owned by the process. The bigger the VM size, the higher the score.
2. Related to #1, the VM size of the process's children are important too. The VM size is cumulative if a process has one or more children.
3. Processes with task priorities smaller than zero (niced processes) get more points.
4. Superuser processes are important, by assumption; thus they have their scores reduced.
5. Process runtime. The longer it runs, the lower the score.
6. Processes that perform direct hardware access are more immune.
7. The swapper (pid 0) and init (pid 1) processes, as well as any kernel threads immune from the list of potential victims.

The process with the biggest score "wins" the election and the OOM killer will kill it very soon.

The heuristic isn't perfect, but usually it works well for most situations. Criteria #1 and #2 clearly show that it is the VMA size that matters, not the number of actual pages a process has. You might think that measuring VMA size will trigger a false alarm, but luckily it doesn't. The `badness()` call occurs inside the page allocation functions when there are few free pages left and page frame reclamation fails, so the VMA size closely matches the number of pages owned by the process.

Why not just count the actual number of pages? That would require more time and require the use of locks, thus making the procedure too expensive to make a fast decision. Knowing that OOM killer isn't perfect, you must be ready for a wrong kill.

The kernel uses the `SIGTERM` signal to inform the target process that it should stop.

How to Reduce OOM Risk

The simple rule to avoid OOM risk is actually simple: don't allocate beyond the machine's current free space. However, many factors come into play, so there are further refinements to the strategy.

Reduce Fragmentation by Properly Ordering Allocation

There is no need to use any sophisticated allocator. You can reduce fragmentation by properly ordering memory allocation and deallocation. As holes easily happen, employ the LIFO strategy: the last one you allocate is the first you need to free.

For example, instead of doing:

```
void *a;
void *b;
void *c;
............
a = malloc(1024);
b = malloc(5678);
c = malloc(4096);

.......................

free(b);
b = malloc(12345);
```

It's better to do:

```
a = malloc(1024);
c = malloc(4096);
b = malloc(5678);
.......................

free(b);
b = malloc(12345);
```

This way, there won't be any hole between the `a` and `c` chunks. You can also consider `realloc()` to resize any existing`malloc()`-ed blocks.

Two example programs (*fragmented1.c* and *fragmented2.c*) demonstrate the effect of allocation rearrangement. Reports at the end of both programs give the number of bytes allocated by the system (kernel and `glibc` allocator) and the number of bytes actually used. For example, on kernel 2.6.11.1, with `glibc` 2.3.3-27 and executing without giving an explicit parameter, `fragmented1` wasted 319858832 bytes (about 305 MB) while `fragmented2` wasted 2089200 bytes (about 2MB). That's 152 times smaller!

You can do further experiments by passing various numbers as the program parameter. This parameter acts as the request size of the `malloc()` call.

Tweak Kernel's Overcommit Behavior

You can change the behavior of the Linux kernel through the */proc* filesystem, as documented in *Documentation/vm/overcommit-accounting* in the Linux kernel's source code. You have three choices when tuning kernel overcommit, expressed as numbers in */proc/sys/vm/overcommit_memory*:

- `0` means that the kernel will use predefined heuristics when deciding whether to allow such an overcommit. This is the default.
- `1` always overcommits. Perhaps you now realize the danger of this mode.
- `2` prevents overcommit from exceeding a certain watermark. The watermark is also tunable through */proc/sys/vm/overcommit_ratio*. Within this mode, the total commit can not exceed the swap space(s) size + overcommit_ratio percent * RAM size. By default, the overcommit ratio is 50.

The default mode usually work quite fine in most situation, but mode #2 offers better protection toward

overcommit. On the other hand, mode #2 requires you to predict carefully how much space all running applications need. You certainly don't want to see your application unable to get more memory chunks just because the limit is too strict. However, mode #2 is a best way to avoid having a program killed suddenly.

Suppose that you have 256MB of RAM and 256MB of swap and you want to limit overcommit at 384MB. That means 256 + 50 percent * 256MB, so put 50 on */proc/sys/vm/overcommit_ratio*.

Check for `NULL` Pointer after Memory Allocation and Audit for Memory Leak

This is a simple rule, but it sometimes goes omitted. By checking for `NULL`, at least you know that the allocator *could* extend the memory area, although there is no obvious guarantee that it will allocate the needed pages later. Usually, you need to bail out or delay the allocation for a moment, depending on your scenarios. Together with overcommit tunables, you have a decent tool to anticipate OOM because `malloc()` will return `NULL` if it believes that it cannot acquire free pages later.

Memory leak is also a source of unnecessary memory consumption. A leaked memory block is one that the application no longer tracks, but that the kernel will not reclaim because, from the kernel's point of view, the task still has it under control. Valgrind is a nice tool to find out such occurrences inside your code without the need to re-code.

Always Consult Memory Allocation Statistics

The Linux kernel provides */proc/meminfo* as a way to find complete information about memory conditions. This */proc* entry is also an information source for utilities such as `top`, `free`, and `vmstat`.

What you need to check is the free and reclaimable memory. The word "free" needs no further explanation, but what does "reclaimable" mean? It refers to buffers and page caches--the disk cache. They are reclaimable because, when memory is tight, the Linux kernel can simply flush them out back to the disk. These are file-backed pages. I've lightly edited this example of memory statistics:

```
$ cat /proc/meminfo
MemTotal:       255944 kB
MemFree:          3668 kB
Buffers:         13640 kB
Cached:         171788 kB
SwapCached:          0 kB
HighTotal:           0 kB
HighFree:            0 kB
LowTotal:       255944 kB
LowFree:          3668 kB
SwapTotal:      909676 kB
SwapFree:       909676 kB
```

Based on this above output, the free virtual memory is MemFree + Buffers + Cached + SwapFree = 1098772 kB.

I failed to find any formalized C (`glibc`) function to find out free (including reclaimable) memory space. The closest I found is by using `get_avphys_pages()` or `sysconf()` (with the `_SC_AVPHYS_PAGES` parameter). They only report the amount of free memory, not the free + reclaimable amount.

That means to get *precise* information, you must programmatically parse the `/proc/meminfo` and calculate it by yourself. If you're lazy, take the `procps` source package as a reference on how to do it. This package contains tools such as `ps`, `top`, and `free`. It is available under the GPL.

Experiments with Alternative Memory Allocators

Different allocators yield different ways to manage memory chunks and to shrink, expand, and create virtual memory areas. Hoard is one example. Emery Berger from the University of Massachusetts wrote it as a high performance memory allocator. Hoard seems to work best for multi-threaded applications; it introduces the concept of per-CPU heap.

Use 64-bit Platforms

Users who need larger user address spaces can consider using 64-bit platforms. The Linux kernel no longer uses the 3:1 VM split for these machines. In other words, user space becomes quite large. It can be a good match for machines with more than 4GB of RAM.

This has no connection to extended addressing schemes, such as Intel's Physical Address Extension (PAE), which allows a 32-bit Intel processor to address up to 64GB of RAM. This addressing deals with physical address, while in the virtual address context, the user space itself is still 3GB (assuming the 3:1 VM split). This extra memory is reachable, but not all mappable into the address space. Unmappable portions of RAM are unusable.

Consider Packed Types on Structures

Packed attributes can help to squeeze the size of `structs`, `enums`, and `unions`. This is a way to save more bytes, especially for array of `structs`. Here is a declaration example:

```
struct test
   {
        char a;
        long b;
   } __attribute__ ((packed));
```

The con for this action is that it makes certain field(s) unaligned and thus it costs more CPU cycles to access the field. "Aligned" here means the variable's address is a multiple of its data type's natural size. The net result is that, depending on the data access frequency, the runtime may get relatively slower. However, take into account page alignment and cache coherence.

Use `ulimit()` for User Processes

With `ulimit -v`, you can limit the address space a process can allocate with `mmap()`. When you reach the limit, all `mmap()`, and hence `malloc()`, calls will return 0 and the kernel's OOM killer will never start. This is most useful in a multi-user environment where you cannot trust all of the users and want to avoid killing random processes.

Acknowledgement

The author gives credits to several people for their assistance and help: Peter Ziljtra, Wolfram Gloger, and Rene Hermant. Mr. Gloger also contributed the `ulimit()` technique.

References

1. "Dynamic Storage Allocation: A Survey and Critical Review," by Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Proceeding 1995 International Workshop of Memory Management.
2. *Hoard: A Scalable Memory Allocator for Multithreaded Applications*, by Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson
3. "Once upon a `free()`" by Anonymous, *Phrack* Volume 0x0b, Issue 0x39, Phile #0x09 of 0x12.
4. "Vudo: An Object Superstitiously Believed to Embody Magical Powers," by Michel "MaXX" Kaempf.

Phrack Volume 0x0b, Issue 0x39, Phile #0x08 of 0x12.
5. "Policy-Based Memory Allocation," by Andrei Alexandrescu and Emery Berger. *C/C++ Users Journal*.
6. "Security of memory allocators for C and C++," by Yves Younan, Wouter Joosen, Frank Piessens, and Hans Van den Eynden. *Report CW419*, July 2005
7. Lecture notes (CS360) about `malloc()`, by Jim Plank, Dept. of Computer Science, University of Tennessee.
8. "Inside Memory Management: The Choices, Tradeoffs, and Implementations of Dynamic Allocation," by Jonathan Bartlett
9. "The Malloc Maleficarum," by Phantasmal Phantasmagoria
10. *Understanding The Linux Kernel, 3rd edition*, by Daniel P. Bovet and Marco Cesati. O'Reilly Media, Inc.

Mulyadi Santosa is a freelance writer who lives in Indonesia.

Return to the Linux DevCenter.