Linux Kernel Memory Management

Page Cache (Introduction), Watermarks, OOM, VMAs

Part 4 of 4

Linux Kernel : Memory Management series by kaiwanTECH

Part 1 : Introduction to Virtual Memory, Paging

Part 2 : Kernel and Process Segments

Part 3 : Memory Organization, Kernel Segment

Part 4 : Page Cache (Intro), Watermarks, OOM, Page Fault, VMAs

Important Notice : Courseware - Legal

kaiwanTECH Linux OS Corporate Training Programs
Please do check out our current offering of world-class, seriously-valuable, high on returns, technical Linux OS corporate training programs here: http://bit.ly/ktcorp

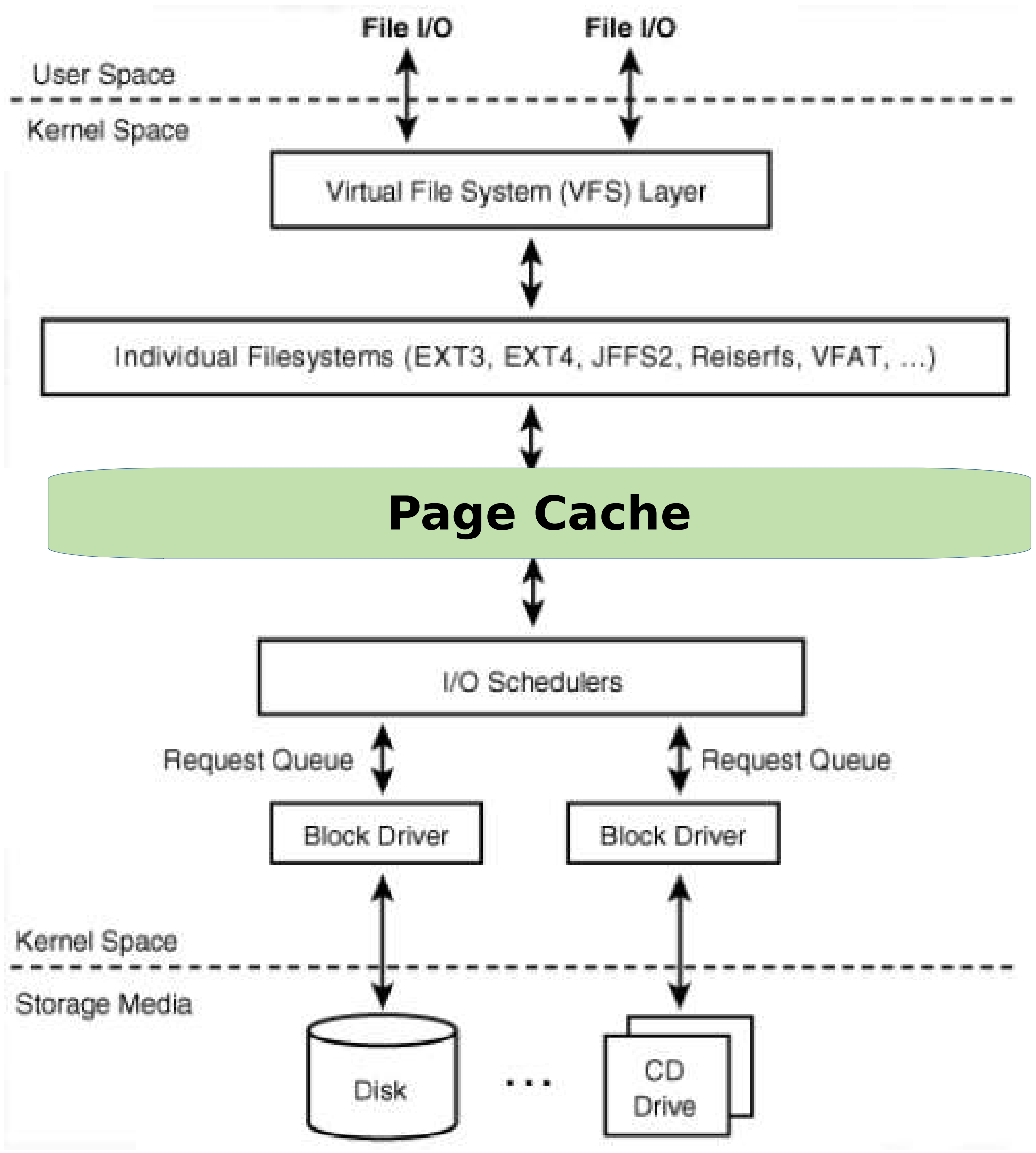
Table of Contents

The Page Cache – an Introduction

• Pretty much every page of user process RAM is kept in either the page cache or the swap cache (the swap cache is just the part of the page cache associated with swap devices).

• The purpose of the cache is simply to keep as much **useful (in use)** data in memory as possible, so that page faults may be serviced **quickly**.

• The cache is a layer between the kernel memory management code and the disk I/O code. Also, when the kernel *swaps pages out* of a task, they *do not get written immediately* to disk, but rather are added to the (swap) cache. The kernel then writes the cache pages out to disk as necessary in order to create free memory.



Basic Definitions:

- **Dirty page**: a page that has been written to and contents are not flushed to disk
- **Clean page**: updated page, i.e., contents are up-to-date on disk; thus, a clean page can be freed when deemed necessary;

Recall that the page structure holds the dirty "bit" for a page, as the flag PG_dirty.

- **Anonymous page**: a page that has no file-mapping is termed an anonymous page (or zero-mapped page); for example, a page that has been malloc'ed. Anonymous pages require some form of backing storage (like swap) to hold them when they are not in main memory.

A typical example of (heavy) page caching taking place is when we use tar to create an archive.

SIDEBAR :: ***See Linux page caching in action***

Test Case 1 : Tar'ring a (large-ish) folder)

We'll select a large folder (over 500 Mb) and tar it's contents. While 'tar' is running, we'll in parallel (on another terminal window) keep an eye on the output of 'free -m' particularly the last column (“cached” memory in MB).

```
$ time tar cvf doc.tar doc/
…
...
real    1m10.655s
user    0m0.392s
sys     0m4.336s
$



$ while [ true ]; do free -m|head -n2|tail -n1; sleep 1; done
<<           total       used       free     shared    buffers     cached >>
Mem:          2011        672       1339          0         67        335
Mem:          2011        672       1339          0         67        335
Mem:          2011        682       1329          0         67        344
Mem:          2011        736       1274          0         67        398
Mem:          2011        751       1259          0         67        413
--snip--
Mem:          2011        850       1161          0         67        508
Mem:          2011        856       1154          0         67        516
Mem:          2011        870       1141          0         67        529
--snip--
Mem:          2011       1069        942          0         67        725
Mem:          2011       1090        920          0         67        747
Mem:          2011       1098        912          0         67        754
--snip--
Mem:          2011       1201        810          0         67        855
Mem:          2011       1204        806          0         67        859
Mem:          2011       1232        779          0         67        886
--snip--
Mem:          2011       1528        483          0         68       1175
Mem:          2011       1536        475          0         68       1183
Mem:          2011       1544        466          0         68       1192
--snip--
Mem:          2011       1782        229          0         68       1425
Mem:          2011       1789        221          0         68       1433
```

```
Mem:          2011       1789        222          0         68       1433
^C
$
```

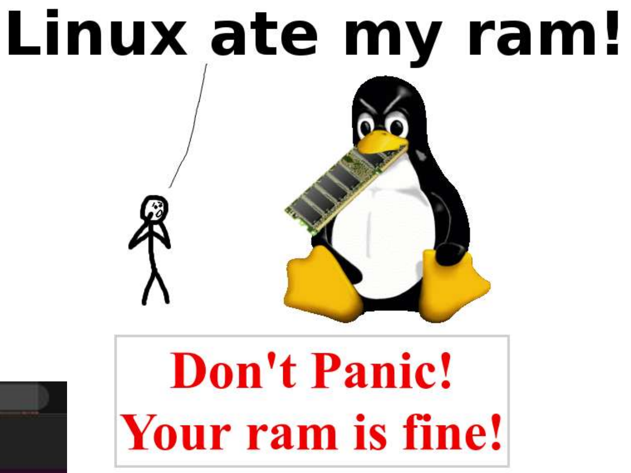
(From the System Monitor applet, we can also see that,as of now on the test system, memory (RAM) usage is: "16% in use by programs, 77% in use as cache").

Now for the interesting part: soon afterward (and without rebooting of course), we run the same 'tar' a second time:

```
$ time tar cvf doc.tar doc/
...
...
real    0m20.152s
user    0m0.184s
sys     0m2.264s
$
```

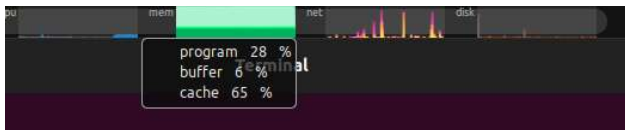
It's taken only about 20 seconds this time, as opposed to about 70 seconds the first time! This is primarily due to the performance effect of page caching.

http://www.linuxatemyram.com/



Please read: Linux Page Cache Basics
and,
Experiments and fun with the Linux disk (page) cache

--*snip*--



```
program  28  %
buffer   6   %
cache   65   %
```

How do I see how much free ram I really have?

To see how much ram is free to use for your applications, run free -m and look at the row that says "-/+ buffers/cache" in the column that says "free". That is your answer in megabytes:

```
$ free -m
             total       used       free     shared    buffers     cached
Mem:          1504       1491         13          0         91        764
-/+ buffers/cache:        635        869
Swap:         2047          6       2041
$
```

If you don't know how to read the numbers, you'll think the ram is 99% full when it's really just 42%.

--snip--

SIDEBAR - Resource
A must-read: "Page Cache, the Affair Between Memory and Files".

--snip--

First, even though this program uses regular read calls, three 4KB page frames are now in the page cache storing part of scene.dat. People are sometimes surprised by this, but all regular file I/O happens through the page cache. In x86 Linux, the kernel thinks of a file as a sequence of 4KB chunks. If you read a single byte from a file, the whole 4KB chunk containing the byte you asked for is read from disk and placed into the page cache. This makes sense because sustained disk throughput is pretty good and programs normally read more than just a few bytes from a file region. The page cache knows the position of each 4KB chunk within the file, depicted above as #0, #1, etc. Windows uses 256KB views analogous to pages in the Linux page cache.

Sadly, in a regular file read the kernel must copy the contents of the page cache into a user buffer, which not only takes cpu time and hurts the cpu caches, but also wastes physical memory with duplicate data. As per the diagram above, the scene.dat contents are stored twice, and each instance of the program would store the contents an additional time. We've mitigated the disk latency problem but failed miserably at everything else. Memory-mapped files are the way out of this madness:

When you use file mapping, the kernel maps your program's virtual pages directly onto the page cache. This can deliver a significant performance boost: Windows System Programming reports run time improvements of 30% and up relative to regular file reads, while similar figures are reported for Linux and Solaris in Advanced Programming in the Unix Environment. You might also save large amounts of physical memory, depending on the nature of your application.

--snip--

In Linux, if you request a large block of memory via malloc(), the C library will create an anonymous mapping (using mmap()) instead of using heap memory. 'Large' means larger than MMAP_THRESHOLD bytes, 128 kB by default and adjustable via mallopt().

--snip--

SIDEBAR :: Using the mmap(2) in applications

The above article "Page Cache, the Affair Between Memory and Files", clearly demonstrates why using memory-mapped IO is far superior to the "usual" approach – using the read/write system calls (or library equivalents) in a loop – especially for the use-case where the amount of IO to be performed is quite large.

But how exactly does one perform memory-mapped IO?

Ah. Do read this excellent tutorial: "Beej's Guide to UNIX IPC", (and for covering the mmap(2)) see the section "10. Memory Mapped Files".

The 'free' command summarizes free and used memory, including the memory usage for buffers and (page) cache:

Eg.

```
$ free --help

Usage:
 free [options]

Options:
 -b, --bytes         show output in bytes
 -k, --kilo          show output in kilobytes
 -m, --mega          show output in megabytes
 -g, --giga          show output in gigabytes
     --tera          show output in terabytes
 -h, --human         show human-readable output
     --si            use powers of 1000 not 1024
 -l, --lohi          show detailed low and high memory statistics
 -o, --old           use old format (without -/+buffers/cache line)
 -t, --total         show total for RAM + swap
 -s N, --seconds N   repeat printing every N seconds
 -c N, --count N     repeat printing N times, then exit

     --help     display this help and exit
 -V, --version  output version information and exit

For more details see free(1).

$ free -ht

             total       used       free     shared    buffers     cached
Mem:          2.8G       2.6G       130M       200M        17M       509M
-/+ buffers/cache:       2.1G       657M
Swap:         5.7G       1.1G       4.6G
Total:        8.5G       3.7G       4.8G
$
```

<< 657MB ~= free(130M) + buffers(17M) + cached(509M).
It is the totally available memory on the system, including buffers and cache. >>

BTW, looking up free memory, as separate 'low' and 'high' memory, basis can be done with the '-l' option to free (-h = human readable):

```
$ free -lh
             total       used       free     shared    buffers     cached
Mem:          7.5G       6.9G       571M       647M       233M       1.8G
Low:          7.5G       6.9G       571M
High:           0B         0B         0B
-/+ buffers/cache:       4.9G       2.6G
Swap:          15G       192M        15G
$
```

Interestingly, on a 32-bit x86 (IA-32) system with *more than* 896 MB RAM, you will notice that this makes a difference:

```
$ free -lh
             total       used       free     shared    buffers     cached
Mem:          2.8G       2.0G       846M       180M        18M       707M
Low:          838M       625M       213M
High:         2.0G       1.4G       633M
-/+ buffers/cache:       1.3G       1.5G
Swap:         5.7G       597M       5.1G
$
```

How to clean caches used by the Linux kernel

Short answer:
`sync && echo 1 > /proc/sys/vm/drop_caches`

From *Documentation/sysctl/vm.txt*
--snip--

drop_caches

Writing to this will cause the kernel to drop clean caches, dentries and inodes from memory, causing that memory to become free.
Eg.

```
# free -h
              total        used        free      shared  buff/cache   available
Mem:            15G        9.1G        290M        975M        6.2G        5.3G
Swap:          7.6G          0B        7.6G
Total:          23G        9.1G        7.9G
# sync
# echo 1 > /proc/sys/vm/drop_caches
# free -h
              total        used        free      shared  buff/cache   available
Mem:            15G        9.1G        4.8G         975M       1.7G        5.3G
Swap:          7.6G          0B        7.6G
Total:          23G        9.1G         12G
```

```
#


To free pagecache:
	echo 1 > /proc/sys/vm/drop_caches
To free dentries and inodes (cache):
	echo 2 > /proc/sys/vm/drop_caches
To free pagecache, dentries (cache) and inodes (cache):
	echo 3 > /proc/sys/vm/drop_caches

As this is a non-destructive operation and dirty objects are not freeable, the
user should run `sync' first.
--snip--
```

Application File IO Sync

- The sync(1) command – calls sync(2)
  - void sync(void) : causes all buffered modifications to file metadata and data to be written to the underlying filesystems.
- fsync(2) : int fsync(int fd);
- open(2)
  - O_DIRECT (Since Linux 2.4.10)

Try to minimize cache effects of the I/O to and from this file. In general this will degrade performance, but it is useful in special situations, such as when applications do their own caching. File I/O is done directly to/from user-space buffers. The O_DIRECT flag on its own makes an effort to transfer data synchronously, but does not give the guarantees of the O_SYNC flag that data and necessary metadata are transferred. To guarantee synchronous I/O, O_SYNC must be used in addition to O_DIRECT. **See NOTES** below for further discussion.

- O_SYNC The file is opened for synchronous I/O. Any write(2)s on the resulting file descriptor will block the calling process until the data has been physically written to the underlying hardware. But see NOTES below.

…

NOTES

…

POSIX provides for three different variants of synchronized I/O, corresponding to the flags O_SYNC, O_DSYNC, and O_RSYNC. Currently

(2.6.31), Linux implements only O_SYNC, but glibc maps O_DSYNC and O_RSYNC to the same numerical value as O_SYNC. Most Linux filesystems

don't actually implement the POSIX O_SYNC semantics, which require all metadata updates of a write to be on disk on returning to user space, but only the O_DSYNC semantics, which require only actual file data and metadata necessary to retrieve it to be on disk by the time the system call returns.

…

<< … lots of Notes on using O_DIRECT … >

"ZFS is the only modern filesystem we know of which uses its own read cache mechanism, rather than relying on its operating system's page cache to keep copies of recently-read blocks in RAM for it."
Src- ZFS 101—Understanding ZFS storage and performance, May 2020.

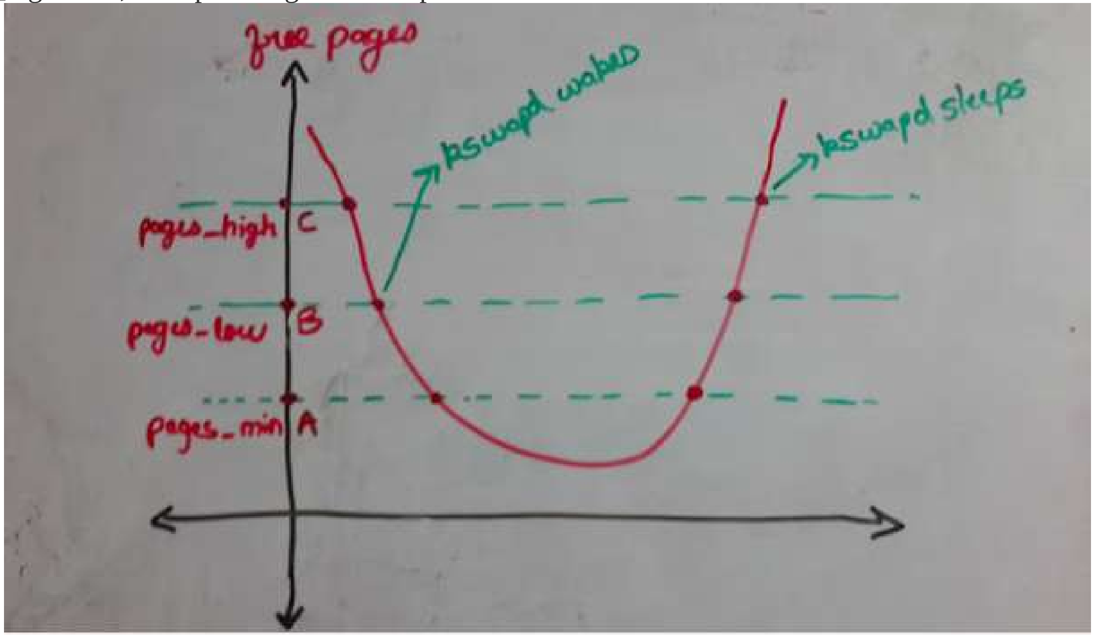
Watermarks

1. Another specific set of "trigger points" are available to the kernel, telling it when to initiate page reclamation. This is the per-node, and within that, per-zone basis triggers called "min", "low" and "high". They represent (in pages) the watermarks or threshold limits :

- When the amount of free RAM in a node:zone drops below high, the kernel starts swapping "gently" hoping to free sufficient pages to bring the level back above high.
- If it drops below the low watermark, the kernel starts swapping aggressively.
- If (rare case), this does not help and free pages go below the min watermark, the kernel now will swap aggressively and not allocate any RAM to userspace applications.

Incidentally, the "min" watermark (pages_min) is the amount of RAM kept reserved by the kernel for critical allocations, expressed in pages. [So, in the example below, in the 'normal' zone, min=936 => 936 * 4Kb = 3744 Kb.]

<<
Source
"Free memory in zones is controlled by various thresholds.
1)pages_min: If free memory falls below pages_min threshold, application demanding memory will also have to do the hard work of first reclaiming memory (synchronously). Once enough memory is reclaimed, request for memory is granted. This could turn out to be a very expensive call on application and you are likely to see performance impact.
2)pages_low: If free memory falls below pages_low threshold, kernel invokes kswapd demon to asynchronously work on reclaiming memory.
3)pages_high: Once enough memory is reclaimed by kswapd daemon, and free pages goes beyond pages_high limit, kswapd will go into sleep state."



>>

2. These settings can be viewed using proc (seen later):

<< Output below from an Ubuntu system, running the 2.6.24-16-generic kernel. Note that the output you see could vary with the kernel version; on some versions a particular proc entry may not even be present >>

```
$ cat /proc/zoneinfo
Node 0, zone      DMA
  pages free     3066
        min      17
        low      21
        high     25
        scanned  0 (a: 8 i: 8)
        spanned  4096
        present  4064
--snip--
  all_unreclaimable: 0
  prev_priority:     12
  start_pfn:         0
Node 0, zone   Normal
  pages free     87722
        min      936
        low      1170
        high     1404
        scanned  0 (a: 0 i: 0)
        spanned  225280
        present  223520
    nr_free_pages 87722
    nr_inactive  60749
    nr_active    26942
--snip--
$
```

Another way to see the above – including a wealth of memory information, is to trigger the Magic-SysRq key combo '<Alt><SysRq> <m>' as follows as root user (this of course assumes that the Magic-SysRq feature is built in and enabled in the running kernel):
(Also, fyi, don't get confused :-) . The output below was captured at a later date and on a different system from the one above; therefore the numbers don't match).

```
# echo m > /proc/sysrq-trigger
# dmesg
--snip--
[26038.210508] Mem-info:
...
[26038.210560] Active:286794 inactive:98690 dirty:0 writeback:0 unstable:0
[26038.210562]  free:68642 slab:36081 mapped:39657 pagetables:1246 bounce:0
[26038.210569] DMA free:10064kB min:68kB low:84kB high:100kB active:2144kB
inactive:56kB present:16256kB pages_scanned:0 all_unreclaimable? no
[26038.210575] lowmem_reserve[]: 0 873 2014 2014
[26038.210585] Normal free:247032kB min:3744kB low:4680kB high:5616kB
active:226992kB inactive:164276kB present:894080kB pages_scanned:0
all_unreclaimable? no
[26038.210591] lowmem_reserve[]: 0 0 9134 9134
```

```
[26038.210601] HighMem free:17472kB min:512kB low:1736kB high:2960kB
active:918040kB inactive:230428kB present:1169228kB pages_scanned:0
all_unreclaimable? no
[26038.210607] lowmem_reserve[]: 0 0 0 0
[26038.210614] DMA: 122*4kB 63*8kB 45*16kB 9*32kB 2*64kB 0*128kB 1*256kB
1*512kB 1*1024kB 1*2048kB 1*4096kB = 10064kB
[26038.210632] Normal: 6196*4kB 4435*8kB 3053*16kB 1624*32kB 663*64kB
202*128kB 41*256kB 6*512kB 0*1024kB 0*2048kB 1*4096kB = 247032kB
[26038.210650] HighMem: 280*4kB 230*8kB 55*16kB 28*32kB 13*64kB 47*128kB
17*256kB 1*512kB 1*1024kB 0*2048kB 0*4096kB = 17472kB
...
#
```

Using the “Magic” SysRq is interesting right? See *Documentation/sysrq.txt* for details.

In terms of code:
In *<linux/mmzone.h>*

```
...
enum zone_watermarks {
         WMARK_MIN,
         WMARK_LOW,
         WMARK_HIGH,
         NR_WMARK
};

#define min_wmark_pages(z) (z->watermark[WMARK_MIN])
#define low_wmark_pages(z) (z->watermark[WMARK_LOW])
#define high_wmark_pages(z) (z->watermark[WMARK_HIGH])

...

struct zone {
    /* Fields commonly accessed by the page allocator */
    /* zone watermarks, access with *_wmark_pages(zone) macros */
         unsigned long watermark[NR_WMARK];
…

} ____cacheline_internodealigned_in_smp;
```

A related question: what exactly determines the number of pages assigned to a zone watermark (min, low, high) level?

The answer is encoded in the function:
mm/page_alloc.c:setup_per_zone_wmarks

which in turn is dependent on (the value of *min_free_kbytes*) and invoked by:
init_per_zone_wmark_min()

The comment above the function *init_per_zone_wmark_min()* explains this:
…

```
/*
```

```
 * Initialise min_free_kbytes.
 *
 * For small machines we want it small (128k min).  For large machines
 * we want it large (64MB max).  But it is not linear, because network
 * bandwidth does not increase linearly with machine size.  We use
 *
 *  min_free_kbytes = 4 * sqrt(lowmem_kbytes), for better accuracy:
 *  min_free_kbytes = sqrt(lowmem_kbytes * 16)
 *
 * which yields
 *
 * 16MB:    512k
 * 32MB:    724k
 * 64MB:    1024k
 * 128MB:   1448k
 * 256MB:   2048k
 * 512MB:   2896k
 * 1024MB:  4096k
 * 2048MB:  5792k
 * 4096MB:  8192k
 * 8192MB:  11584k
 * 16384MB: 16384k
 */
int __meminit init_per_zone_wmark_min(void)
{
--snip--
}
```

From *Documentation/sysctl/vm.txt*
. . .
min_free_kbytes:

This is used to force the Linux VM to keep a minimum number
of kilobytes free. The VM uses this number to compute a
watermark[WMARK_MIN] value for each lowmem zone in the system.
Each lowmem zone gets a number of reserved free pages based
proportionally on its size.

Some minimal amount of memory is needed to satisfy PF_MEMALLOC
allocations; if you set this to lower than 1024KB, your system will
become subtly broken, and prone to deadlock under high loads.

Setting this too high will OOM your machine instantly.

...

Sysctl – some useful VM Tunables

*Source: **Kernel documentation on /proc/sys entries : Documentation/sysctl/vm.txt***

Also very useful: *http://balodeamit.blogspot.com/2015/11/deep-dive-into-linux-memory-management.html*

```
[...]
==============================================================

dirty_background_ratio

Contains, as a percentage of total available memory that contains free pages
and reclaimable pages, the number of pages at which the background kernel
flusher threads will start writing out dirty data.

The total available memory is not equal to total system memory.

# cat /proc/sys/vm/dirty_background_ratio
10
#
```

So, by default, the kernel will start background writing memory pages to disk when 10% of them are dirtied.

```
[...]

dirty_ratio

Contains, as a percentage of total available memory that contains free pages
and reclaimable pages, the number of pages at which a process which is
generating disk writes will itself start writing out dirty data.

The total available memory is not equal to total system memory.

# cat /proc/sys/vm/dirty_ratio
20
#
```

Default: 20

```
[...]

page-cluster
```

```
page-cluster controls the number of pages up to which consecutive pages
are read in from swap in a single attempt. This is the swap counterpart
to page cache readahead.
The mentioned consecutivity is not in terms of virtual/physical addresses,
but consecutive on swap space - that means they were swapped out together.

It is a logarithmic value - setting it to zero means "1 page", setting
it to 1 means "2 pages", setting it to 2 means "4 pages", etc.
Zero disables swap readahead completely.

The default value is three (eight pages at a time).  There may be some
small benefits in tuning this to a different value if your workload is
swap-intensive.

Lower values mean lower latencies for initial faults, but at the same time
extra faults and I/O delays for following faults if they would have been part
of that consecutive pages readahead would have brought in.
```

```
# cat /proc/sys/vm/page-cluster
3
#
```

[...]

swappiness

```
This control is used to define how aggressive(ly) the kernel will swap
memory pages.  Higher values will increase agressiveness, lower values
decrease the amount of swap.  A value of 0 instructs the kernel not to
initiate swap until the amount of free and file-backed pages is less
than the high water mark in a zone.

The default value is 60.
```

```
# cat /proc/sys/vm/swappiness
60
#
```

[...]

vfs_cache_pressure

```
This percentage value controls the tendency of the kernel to reclaim
the memory which is used for caching of directory and inode objects.

At the default value of vfs_cache_pressure=100 the kernel will attempt to
reclaim dentries and inodes at a "fair" rate with respect to pagecache and
swapcache reclaim.  Decreasing vfs_cache_pressure causes the kernel to prefer
to retain dentry and inode caches. When vfs_cache_pressure=0, the kernel will
```

```
never reclaim dentries and inodes due to memory pressure and this can easily
lead to out-of-memory conditions. Increasing vfs_cache_pressure beyond 100
causes the kernel to prefer to reclaim dentries and inodes.

Increasing vfs_cache_pressure significantly beyond 100 may have negative
performance impact. Reclaim code needs to take various locks to find freeable
directory and inode objects. With vfs_cache_pressure=1000, it will look for
ten times more freeable objects than there are.
```

```
# cat /proc/sys/vm/vfs_cache_pressure
100              << it's not a percentage >>
#
```

Tuning these values appropriate to your particular workload could yield possible benefits. It's a painstaking job and delicate balance though; wrong tuning could result in performance loss.
See this article for an example: *My Journey to Improve Disk Performance on the Raspberry Pi*.

OOM Killer

<< Notes below extracted from ULK3, pg 710. >>

Despite the PFRA (Page Frame Reclamation Algorithm) effort to keep a reserve of free page frames << calls *shrink_all_memory(nr_to_reclaim)*, which is a wrapper over *do_try_to_free_pages()*, when caches (and other reclaimable pages) must be freed up >>, it is possible for the pressure on the virtual memory subsystem to become so high that all available memory becomes exhausted. This situation could quickly induce a freeze of every activity in the system: the kernel keeps trying to free memory in order to satisfy some urgent request, but it does not succeed because the swap areas are full and all disk caches have already been shrunken. As a consequence, no process can proceed with its execution, thus no process will eventually free up the page frames that it owns.

<<
So, the important thing here is:

When the system runs out of *all* memory, i.e., when RAM *and* swap are (almost) completely full, the OOM killer pays a visit!

Also note that on typical *embedded* systems, there is no swap space!

Good Resources:

- *When Linux Runs Out of Memory.*
- *Surviving the Linux OOM Killer, Oct 2018*

>>

<<
Src
Script to display OOM Score and Oom Adj value of all processes alive

```
# Src: https://dev.to/rrampage/surviving-the-linux-oom-killer-2ki9
printf 'PID\tOOM Score\tOOM Adj\tCommand\n'
while read -r pid comm
do
   [ -f /proc/$pid/oom_score ] && [ $(cat /proc/$pid/oom_score) != 0 ] &&
       printf '%d\t%d\t\t%d\t%s\n' "$pid" "$(cat /proc/$pid/oom_score)" "$(cat
/proc/$pid/oom_score_adj)" "$comm"
done < <(ps -e -o pid= -o comm=) | sort -k 2nr
```

>>

To cope with this dramatic situation, the PFRA makes use of a so-called **out of memory (OOM) killer**, which selects a process in the system and abruptly kills it to free its page frames. The OOM killer is like a surgeon that amputates the limb of a man to save his life: losing a limb is not a nice thing, but sometimes there is nothing better to do.

The *out_of_memory( )* function is invoked by *_ _alloc_pages( )* when the free memory is very low and the PFRA has not succeeded in reclaiming any page frames (see the section "The Zone Allocator" in Chapter 8). The function invokes *select_bad_process( )* to select a victim among the existing processes, then invokes *oom_kill_process( )* to perform the sacrifice.

<<
Source – Memory FAQ
...
```
  Strict overcommit encourages the addition of gigabytes (even terabytes)
  of swap space, which leads to swap thrasing if it is ever used.  Systems
  with large amounts of swap space can literally thrash for days, during which
  they perform less computation than they could perform in a single minute
  during non-thrashing operation (such as if the OOM killer triggered a
  reboot).
...
```
>>

Of course, *select_bad_process( )* does not simply pick a process at random. The selected process should satisfy several requisites:

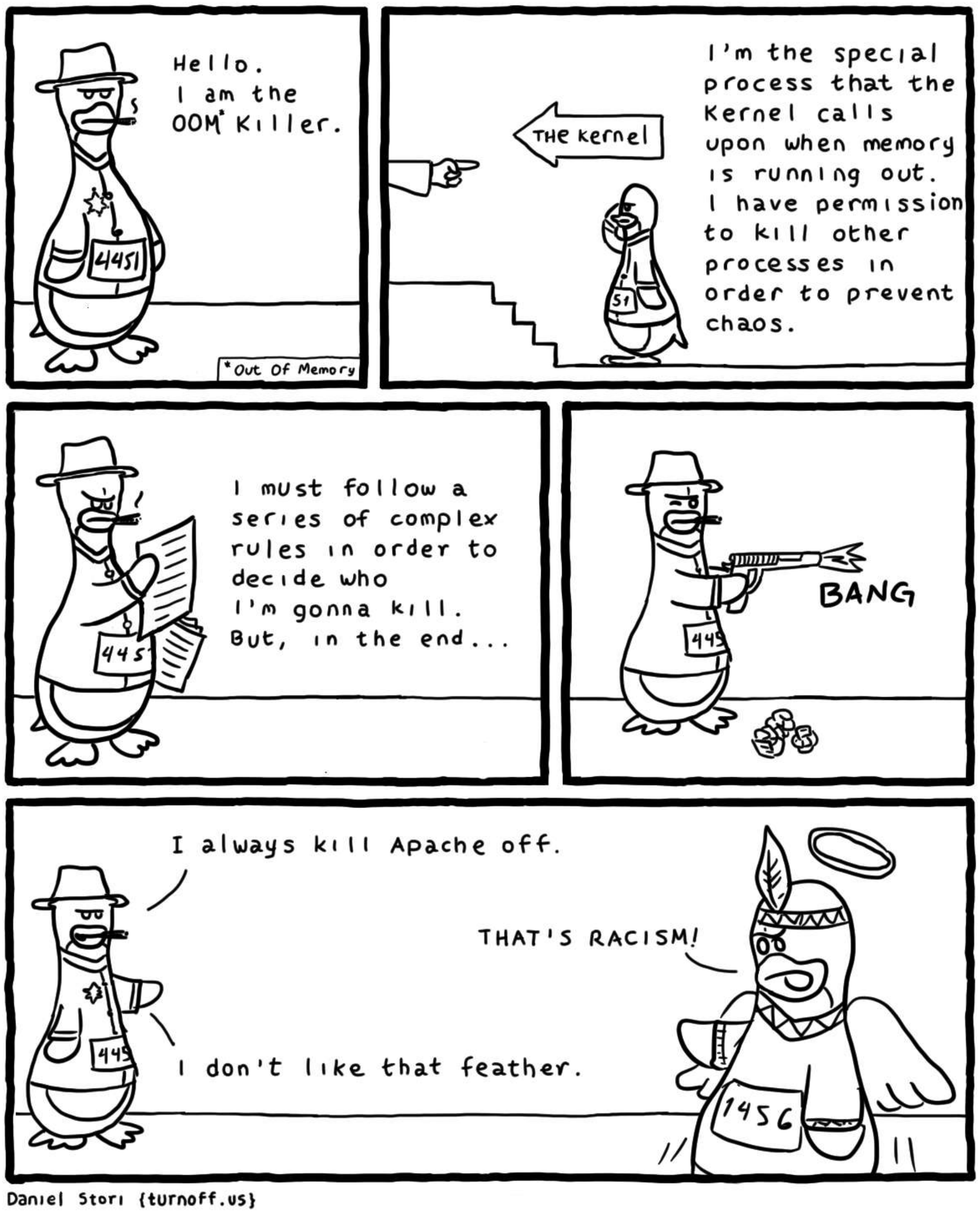
- The victim should own a large number of page frames, so that the amount of memory that can be freed is significant. (As a countermeasure against the "fork-bomb" processes, the function considers the amount of memory eaten by all children owned by the parent, too.)
- Killing the victim should lose a small amount of work - it is not a good idea to kill a batch process that has been working for hours or days.
- The victim should be a low static priority (nice value) process - the users tend to assign lower priorities to less important processes.
- The victim should not be a process with root privileges - they usually perform important tasks.
- The victim should not directly access hardware devices (such as the X Window server), because the hardware could be left in an unpredictable state.
- The victim cannot be swapper (process 0), init (process 1), or any other kernel thread.

  The *select_bad_process( )* function scans every process in the system, uses an empirical formula to compute from the above rules a value that denotes how good selecting that process is, and returns the process descriptor address of the "best" candidate for eviction. Then, the *out_of_memory( )* function invokes *oom_kill_process( )* to send a deadly signal (usually SIGKILL; see Chapter 11) either to a child of that process or, if this is not possible, to the process itself. The *oom_kill_process( )* function also kills all clones that share the same memory descriptor with the selected victim (basically, threads of the process).

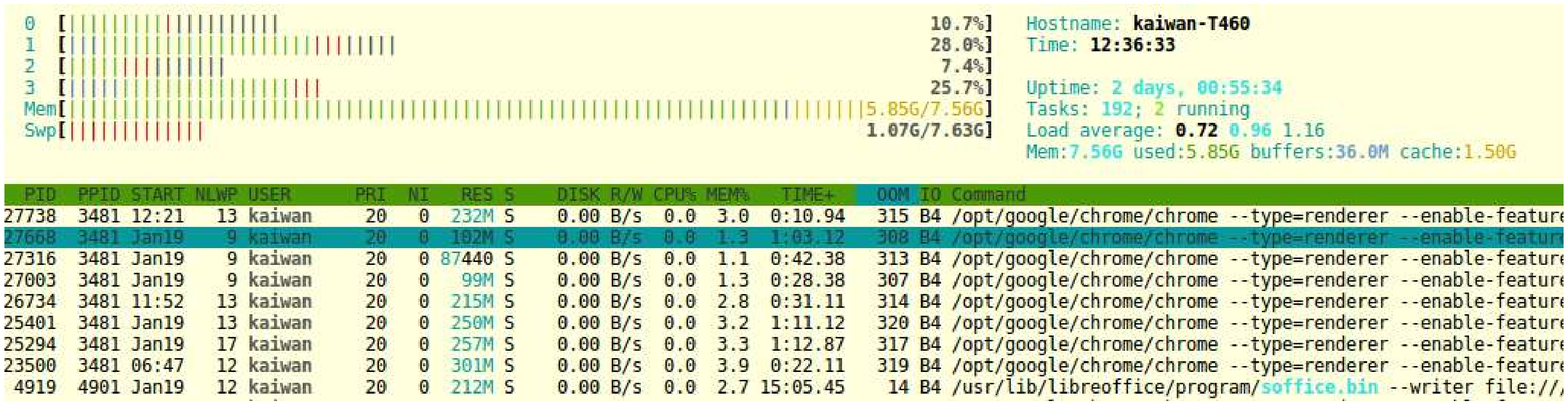
See
Introducing the OOM Killer [comic], Dzone :-)

Linux 2.6.36 :
“Recommended LWN article:
Another OOM killer rewrite

The Out of Memory Killer is the part of the VM that kills a process when there's no memory (both RAM and swap) left. The algorithm that decides what is the better process to be killed has been rewritten in this release and should make better decisions.
Code: (commit), (commit)”


Hello. I am the OOM* Killer.
* Out Of Memory
THE kernel
I'm the special process that the Kernel calls upon when memory is running out. I have permission to kill other processes in order to prevent chaos.
I must follow a series of complex rules in order to decide who I'm gonna kill. But, in the end...
BANG
I always kill Apache off.
THAT'S RACISM!
I don't like that feather.
Daniel Stori {turnoff.us}

Here’s a screenshot of *htop* output on a laptop sorted by “OOM score”:
(‘chrome’ (and/or ‘brave’ if alive) has the highest OOM scores; <eye-roll> :-))



```
  0  [|||||||||||||||||||                                               10.7%]   Hostname: kaiwan-T460
  1  [||||||||||||||||||||||||||||||                                    28.0%]   Time: 12:36:33
  2  [|||||||||||||                                                      7.4%]
  3  [|||||||||||||||||||||||                                           25.7%]   Uptime: 2 days, 00:55:34
  Mem[|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||5.85G/7.56G]   Tasks: 192; 2 running
  Swp[||||||||||||                                                1.07G/7.63G]   Load average: 0.72 0.96 1.16
                                                                                 Mem:7.56G used:5.85G buffers:36.0M cache:1.50G

  PID  PPID START NLWP USER      PRI  NI   RES S   DISK R/W CPU% MEM%   TIME+    OOM IO Command
27738  3481 12:21   13 kaiwan     20   0  232M S  0.00 B/s  0.0  3.0  0:10.94   315 B4 /opt/google/chrome/chrome --type=renderer --enable-feature
27668  3481 Jan19    9 kaiwan     20   0  102M S  0.00 B/s  0.0  1.3  1:03.12   308 B4 /opt/google/chrome/chrome --type=renderer --enable-feature
27316  3481 Jan19    9 kaiwan     20   0 87440 S  0.00 B/s  0.0  1.1  0:42.38   313 B4 /opt/google/chrome/chrome --type=renderer --enable-feature
27003  3481 Jan19    9 kaiwan     20   0   99M S  0.00 B/s  0.0  1.3  0:28.38   307 B4 /opt/google/chrome/chrome --type=renderer --enable-feature
26734  3481 11:52   13 kaiwan     20   0  215M S  0.00 B/s  0.0  2.8  0:31.11   314 B4 /opt/google/chrome/chrome --type=renderer --enable-feature
25401  3481 Jan19   13 kaiwan     20   0  250M S  0.00 B/s  0.0  3.2  1:11.12   320 B4 /opt/google/chrome/chrome --type=renderer --enable-feature
25294  3481 Jan19   17 kaiwan     20   0  257M S  0.00 B/s  0.0  3.3  1:12.87   317 B4 /opt/google/chrome/chrome --type=renderer --enable-feature
23500  3481 06:47   12 kaiwan     20   0  301M S  0.00 B/s  0.0  3.9  0:22.11   319 B4 /opt/google/chrome/chrome --type=renderer --enable-feature
 4919  4901 Jan19   12 kaiwan     20   0  212M S  0.00 B/s  0.0  2.7 15:05.45    14 B4 /usr/lib/libreoffice/program/soffice.bin --writer file://
```

From *Documentation/sysctl/vm.txt :*

...
oom_kill_allocating_task

This enables or disables killing the OOM-triggering task in out-of-memory situations.

If this is set to zero, the OOM killer will scan through the entire tasklist and select a task based on heuristics to kill. This normally selects a rogue memory-hogging task that frees up a large amount of memory when killed.

If this is set to non-zero, the OOM killer simply kills the task that triggered the out-of-memory condition. This avoids the expensive tasklist scan.

If panic_on_oom is selected, it takes precedence over whatever value is used in oom_kill_allocating_task.

The default value is 0.
...
==

FAQ :: Disabling OOM on a Particular Process

Source: http://unix.stackexchange.com/questions/153585/how-oom-killer-decides-which-process-to-kill-first

...
Higher the value of `oom_score` of any process the higher is its likelihood of getting killed by the*OOM Killer* in an out-of-memory situation.

But how is the `OOM_Score` calculated?

In David's patch set, the old badness() heuristics are almost entirely gone. Instead, the calculation turns into a simple question of what percentage of the available memory is being used by the process. If the system as a whole is short of memory, then "available memory" is the sum of all RAM and swap space available to the system. If, instead, the OOM situation is caused by exhausting the memory allowed to a given cpuset/control group, then "available memory" is the total amount allocated to that control group. A similar calculation is made if limits imposed by a memory policy have been exceeded. In each case, the memory use of the process is deemed to be the sum of its resident set (the number of RAM pages it is using) and its swap usage.

This calculation produces a percent-times-ten number as a result; a process which is using every byte of the memory available to it will have a score of 1000, while a process using no memory at all will get a score of zero. There are very few heuristic tweaks to this score, but the code does still subtract a small amount (30) from the score of root-owned processes on the notion that they are slightly more valuable than user-owned processes.

One other tweak which is applied is to add the value stored in each process's oom_score_adj variable, which can be adjusted via /proc. This knob allows the adjustment of each process's attractiveness to the OOM killer in user space; setting it to -1000 will disable OOM kills entirely, while setting to +1000 is the equivalent of painting a large target on the associated process.

--snip--

Resource-

How to Configure the Linux Out-of-Memory Killer, by Robert Chase, Oracle

An experiment : Invoking the kernel OOM-Killer

One way to invoke the kernel's OOM killer is using the program shown below - a very simple "crazy allocator".

An alternate (easy) way to invoke it is by using the 'Magic-SysRq' facility, specifically command key 'f' will call *oom_kill* to kill a memory hog process (i.e. doing):

```
# echo f > /proc/sysrq-trigger
```

!CAREFUL! Save your work first; some process(es) will die!

For details on SysRq usage, see *Documentation/sysrq.txt*).

```
$ cat oom_killer_try.c
/*
 * oom-killer-try.c
 * Demo the kernel OOM killer by having this process call malloc()
 * repetedly without freeing.
 * Ultimately, the kernel will kill it via OOM.
 * For a more "sure" kill, set the "force-page-fault" flag.
 *
 ** WARNING **
 * Be warned that running this intensively can/will cause
 * heavy swapping on your system and will probably necessitate a
 * reboot.
 *
 * Author: Kaiwan N Billimoria <kaiwan@designergraphix.com>
 * License: MIT
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define BLK                     8192   // 2 pages
#define SEE_MAPS_FLAG     1

int force_page_fault=0;

/*
 * This code (the cat,awk,.. commands) will actually stop working when
   memory ressure becomes high - as userspace app requests for memory will
   be ignored by the kernel.
 */
static void see_maps(void)
{
#if (SEE_MAPS_FLAG==1)
	int s;
	char cmd[128];
	sprintf(cmd, "cat /proc/%d/maps |awk '$6==\"[heap]\" {print $0}' ;
	 usleep 250000", getpid());
	//printf("cmd: %s\n", cmd);
	s = system(cmd);
	if (s == -1) {
		perror("system");
	}
	fflush(stdout);
#endif
}

int main( int argc, char **argv )
{
	char * p;
	int i=0, stepval=5000, verbose=0;

	if( argc < 3 ) {
		fprintf(stderr,"Usage: %s alloc-loop-count force-page-fault[0|1]
[verbose_flag[0|1]]\n",argv[0]);
		exit(1);
	}

	printf("%s: PID %d\n", argv[0], getpid());
	if( atoi(argv[2])==1)
		force_page_fault=1;

	if (argc>=4) {
		if (atoi(argv[3])==1)
			verbose=1;
	}

	do {
		p=(char *)malloc(BLK);  << only “virtual” memory has been
		  successfully allocated as of just now, not physical! >>
		if (verbose)
			printf("%06d\taddr p = %x   break = %x\n", ++i, (unsigned)p,
(unsigned)sbrk(0) );
		if( !p ) {
			fprintf (stderr, "%s: loop #%d: malloc failure.\n",
			  argv[0], i);
			break;
		}

/* Force the MMU to raise page fault exception by writing into  the page; (reading
or) writing a single byte, any byte, will                  do the trick!
This is as the virtual address referenced will have no PTE entry, causing the MMU to
raise a page fault. The OS fault handler, being intelligent, figures out it's a "good
fault" and allocates a page frame. Only now do we have physical memory!
		 */
		if( force_page_fault ) {
			p[4000] &= 0x14;
			p[8000] |= 'a';
		}

		if(!(i%stepval)) {		// every 'stepval' iterations..
			printf("%d...\n",i);
			see_maps();
```

```
                        usleep(250000);  // to have time for sampling free mem..
                }
                i++;
        } while( p && (i<atoi(argv[1])) );

        see_maps();
        return 0;
}
$
```

Source

…

Demand paging

The next question we need to address is how the page tables get created. Linux could create appropriate page-table entries whenever a range of virtual memory is allocated.

However, this would be wasteful because most programs allocate much more virtual memory than they ever use at any given time. For example, the text segment of a program often includes large amounts of error handling code that is seldom executed.

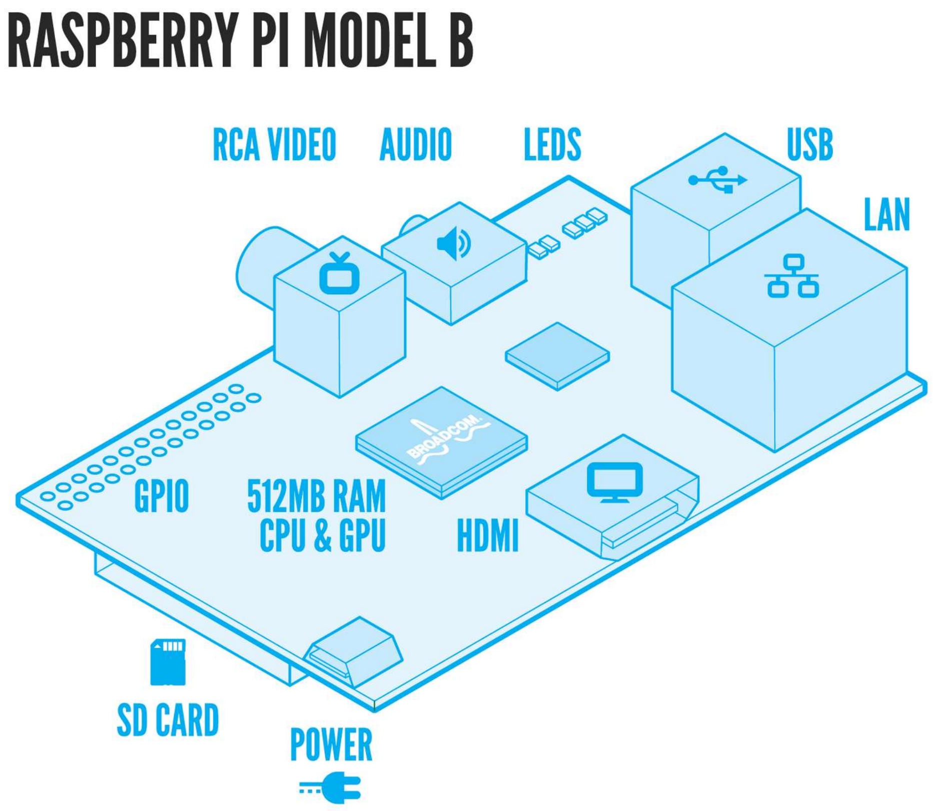
To avoid wasting memory on virtual pages that are never accessed, Linux uses a method called *demand paging*. With this method, the virtual address space starts out empty << we have only *virtual* pages, not physical >>. This means that, logically, all virtual pages are marked in the page table as **not present**.

When accessing a virtual page that is not present, the CPU (MMU, really) generates a *page fault*. This fault is intercepted by the Linux kernel and causes the page fault handler to be activated.

There, the kernel can allocate a new page frame, determine what the content of the accessed page should be (e.g., a new, zeroed page, a page loaded from the data section of a program, or a page loaded from the text segment of a program), load the page, and then update the page table to mark the page as **present**. Execution then resumes in the process with the instruction that caused the fault. Since the required page is now present, the instruction can now execute without causing a page fault.

...

Lets try out invoking the "OOM Killer" on an ARM-Linux system, a "Raspberry Pi" R2 Mode B board.



The system is based on a Broadcom BCM2835 SoC, has 464 MB RAM (actually 512MB but some is shared with a VideoCore GPU).
Thus, total RAM expressed in page frames = 464*1024/4 = 118,784.

Thus, if we allocate 2 pages in each malloc(), we can allocate a *theoretical maximum* of (118784 / 2) times = 59,392 times before we *completely* run out of memory! That too:

- we're ignoring the fact that some RAM is already in use
- embedded Linux systems that use flash chips as their secondary storage medium (the usual case), typically do *not* enable a swap partition. So, once we're out of RAM, we're out of virtual memory! (The R Pi running "Raspbian" Linux is an exception, though: it *does* have a default 100MB swap partition!).

```
RPi $ cat /etc/issue
Debian GNU/Linux 7.0 \n \l                    << Raspbian >>

RPi $ cat /proc/version
```

```
Linux version 3.6.11+ (kaiwan@asus-N53) (gcc version 4.4.1 (Sourcery G++ Lite 2010q1-
202) ) #1 PREEMPT Thu Feb 21 18:14:17 IST 2013
RPi $


RPi $ cat /proc/sys/vm/oom_kill_allocating_task
0
RPi $ cat /proc/sys/vm/oom_dump_tasks
1
RPi $
```

Experiment 1:
Worst case: (attempt to) loop 118,784 times, allocating 2 page frames per loop iteration to eat all memory. We say "page frames" as we shall turn On the "force-page-fault" flag, thus truly allocating two physical page frames per loop iteration!

(Think of this as having the OS instantiate page frames "on the fly" or "only when required" or "on demand"! More on this "memory overcommit feature" later...).

```
RPi $ free -m
             total       used       free     shared    buffers     cached
Mem:           464         98        366          0         12         45
-/+ buffers/cache:         40        424
Swap:           99          0         99
RPi $
RPi $ ./oom-killer-try
Usage: ./oom-killer-try alloc-loop-count force-page-fault[0|1] [verbose_flag[0|1]]
RPi $
RPi $ ./oom-killer-try 118784 1
./oom-killer-try: PID 2148
0...
01512000-01535000 rw-p 00000000 00:00 0          [heap]
5000...                                               ← loop iteration count
01512000-03c3c000 rw-p 00000000 00:00 0          [heap]
10000...
01512000-06364000 rw-p 00000000 00:00 0          [heap]
15000...
01512000-08a68000 rw-p 00000000 00:00 0          [heap]
20000...
01512000-0b190000 rw-p 00000000 00:00 0          [heap]
25000...
01512000-0d895000 rw-p 00000000 00:00 0          [heap]
30000...
01512000-0ffbd000 rw-p 00000000 00:00 0          [heap]
35000...
system: Cannot allocate memory      << perror output from the system(3) function >>
40000...
system: Cannot allocate memory
45000...
system: Cannot allocate memory
50000...
system: Cannot allocate memory
55000...
system: Cannot allocate memory
Killed
RPi $
```

```
<< Kernel Log >>
$ dmesg
. . .
[ 1316.727656] oom-killer-try invoked oom-killer: gfp_mask=0x200da, order=0, oom_adj=0,
oom_score_adj=0
<< Kernel-mode stack >>
[ 1316.727740] [<c001563c>] (unwind_backtrace+0x0/0xf4) from [<c00a36c0>]
  (dump_header+0x6c/0x200)
[ 1316.727777] [<c00a36c0>] (dump_header+0x6c/0x200) from [<c00a3e48>]
  (oom_kill_process+0x29c/0x3e4)
[ 1316.727811] [<c00a3e48>] (oom_kill_process+0x29c/0x3e4) from [<c00a42fc>]
  (out_of_memory+0x2d0/0x354)
[ 1316.727848] [<c00a42fc>] (out_of_memory+0x2d0/0x354) from [<c00a92b8>]
  (__alloc_pages_nodemask+0x5e4/0x61c)   << The __alloc_pages_nodemask() function's, i.e.,
the buddy system's, failure to allocate memory causes it to invoke the out_of_memory()
function ! >>
[ 1316.727893] [<c00a92b8>] (__alloc_pages_nodemask+0x5e4/0x61c) from [<c00be5ec>]
  (handle_pte_fault+0x550/0x738)         << the OS page fault handler 'understands' that this
is a legal request- thus it invokes  __alloc_pages_nodemask() (the heart of the buddy system
allocator) to allocate a page frame. Until right now the allocation has succeeded, but now
fails as the system hits OOM !!! >>
[ 1316.727928] [<c00be5ec>] (handle_pte_fault+0x550/0x738) from [<c00bec98>]
  (handle_mm_fault+0x98/0xcc)
[ 1316.727969] [<c00bec98>] (handle_mm_fault+0x98/0xcc) from [<c03f3c84>]
  (do_page_fault+0x2dc/0x414)
[ 1316.728006] [<c03f3c84>] (do_page_fault+0x2dc/0x414) from [<c00083d0>]
  (do_DataAbort+0x34/0x98)
[ 1316.728037] [<c00083d0>] (do_DataAbort+0x34/0x98) from [<c03f255c>]
  (__dabt_usr+0x3c/0x40)   << the write into the (as-of-now-invalid page frame triggers the
MMU to raise a Data Abort exception (on ARM), which in turn is hooked into the OS's page
fault handling code! >>
[ 1316.728057] Exception stack(0xdc841fb0 to 0xdc841ff8)
[ 1316.728079] 1fa0:                                     00002009 00016449 1e874bb8
  00000000
[ 1316.728104] 1fc0: 1e872bb8 00002009 b6f79288 1e872bb0 00018450 00000000 00002018
  00002710
[ 1316.728126] 1fe0: 00000004 bef2a618 b6ec072c b6ebe29c 60000010 ffffffff
[ 1316.728139] Mem-info:
[ 1316.728152] Normal per-cpu:
[ 1316.728167] CPU    0: hi:  186, btch:  31 usd:   0
[ 1316.728198] active_anon:56156 inactive_anon:56199 isolated_anon:0
[ 1316.728198]  active_file:27 inactive_file:42 isolated_file:0
[ 1316.728198]  unevictable:0 dirty:0 writeback:9716 unstable:0
[ 1316.728198]  free:2048 slab_reclaimable:594 slab_unreclaimable:1571

  [...]

[ 1316.754426] 2165 slab pages
[ 1316.754437] 370 pages shared
[ 1316.754447] 11998 pages swap cached


<< Table showing all processes with their memory usage and “oom score”; thus the kernel makes
a decision >>
                         << rss: resident-set-size: actual RAM usage (Kb) >>
[ 1316.754461] [ pid ]   uid  tgid total_vm      rss nr_ptes swapents oom_score_adj name
[ 1316.754507] [  142]     0   142      720        0       5      133         -1000 udevd
[ 1316.754532] [  256]     0   256      719        0       5      137         -1000 udevd
[ 1316.754560] [  263]     0   263      719        0       5      135         -1000 udevd
[ 1316.754585] [ 1362]     0  1362      436        7       4       18         -1000 ifplugd
[ 1316.754609] [ 1383]     0  1383      436        7       4       16         -1000 ifplugd
[ 1316.754635] [ 1632]     0  1632     1223        0       5      429         -1000 dhclient
[ 1316.754659] [ 1692]     0  1692     6992        1       7      110             0 rsyslogd
[ 1316.754685] [ 1711] 65534  1711      503        4       5       26             0 thd
[ 1316.754709] [ 1749]     0  1749      948        0       5       41             0 cron

--snip--

[ 1316.755475] [ 2060]     0  2060     2450        5       7      151             0 sshd
[ 1316.755500] [ 2074]  1000  2074     2450       23       6      134             0 sshd
[ 1316.755525] [ 2075]  1000  2075     1361        0       5      267             0 bash
[ 1316.755551] [ 2148]  1000  2148   120097    99945     238    19731             0 oom-
killer-try
[ 1316.755573] Out of memory: Kill process 2148 (oom-killer-try) score 829 or sacrifice child
[ 1316.755596] Killed process 2148 (oom-killer-try) total-vm:480388kB, anon-rss:399736kB,
file-rss:44kB
RPi $
```

<<

What exactly do the oom_score_adj and the "oom score" values mean?

Please look up documentation here:
https://www.kernel.org/doc/Documentation/filesystems/proc.txt section "/proc/<pid>/oom_adj & /proc/<pid>/oom_score_adj - Adjust the oom-killer score" for all details.

>>

```
RPi $ free -m
             total       used       free     shared    buffers     cached
Mem:           464         76        388          0          1         14
-/+ buffers/cache:         60        404
Swap:           99         59         40
RPi $
```

Experiment 2 :

(Attempt to) loop 118,784 times, allocating 2 virtual pages per loop iteration. We say "virtual pages" as we set the "force-page-fault" flag to zero; thus malloc() "succeeds" but only in a virtual sense- the virtual regions are succcessfully allocated but *no physical page frames* have been allocated (yet). This will happen only on a subsequent (future) access to the page(s).

```
RPi $ ./oom-killer-try
Usage: ./oom-killer-try alloc-loop-count force-page-fault[0|1] [verbose_flag[0|1]]

RPi $ ./oom-killer-try 118784 0
./oom-killer-try: PID 2169
0...
addr p = 0x00feb008  break = 0x0100e000
5000...
addr p = 0x03704c48  break = 0x03715000
10000...
addr p = 0x05e1e888  break = 0x05e3d000
15000...
addr p = 0x085384c8  break = 0x08541000
20000...
addr p = 0x0ac52108  break = 0x0ac69000
25000...
addr p = 0x0d36bd48  break = 0x0d36e000
30000...
addr p = 0x0fa85988  break = 0x0fa96000
```

```
35000...
addr p = 0x1219f5c8  break = 0x121be000
40000...
addr p = 0x148b9208  break = 0x148c2000
45000...
addr p = 0x16fd2e48  break = 0x16feb000
50000...
addr p = 0x196eca88  break = 0x196f1000
55000...
addr p = 0x1be066c8  break = 0x1be17000
60000...
addr p = 0x1e520308  break = 0x1e53f000
65000...
addr p = 0x20c39f48  break = 0x20c44000
70000...
addr p = 0x23353b88  break = 0x2336c000
75000...
addr p = 0x25a6d7c8  break = 0x25a72000
80000...
addr p = 0x28187408  break = 0x28198000
85000...
addr p = 0x2a8a1048  break = 0x2a8c0000
90000...
addr p = 0x2cfbac88  break = 0x2cfc7000
95000...
addr p = 0x2f6d48c8  break = 0x2f6ed000
100000...
addr p = 0x31dee508  break = 0x31df3000
105000...
addr p = 0x34508148  break = 0x3451b000
110000...
addr p = 0x36c21d88  break = 0x36c42000
115000...
addr p = 0x3933b9c8  break = 0x39348000
RPi $
```

Experiment 3:
READ-ONLY Worst case: (attempt to) loop 118,784 times, allocating 2 page frames per loop iteration to eat all memory. We say "page frames" as we shall turn On the "force-page-fault" flag, thus truly allocating two physical page frames per loop iteration!

However, this time we don't write to two bytes of the two pages (as we did in *Experiment 1*); instead, we just *read* 2 bytes from the 2 pages.

What do we find?
It *still* causes the OOM-killer to be invoked, as even a read operation on the malloc'ed page triggers a page fault!

```
             if (force_page_fault) {
#if 0
                    p[4000] &= 0x14;
                    p[8000] |= 'a';
#else
                    { char x, y;
                    x = p[4000];  // just read
```

```
                      y = p[8000];
                      }
#endif
```

```
<< On an x86_64 >>

$ ./rdpg_oom_try 9000000 1
./rdpg_oom_try: PID 3084
0...
5000...
10000...
15000...
20000...
25000...
30000...

--snip--

780000...
785000...
790000...
Killed
$
```

An actual case of the OOM-killer being invoked; on an x86_64 Ubuntu 16.04 desktop. What was running? A Yocto build! Heavyweight, indeed!

```
$ bitbake core-image-minimal

[...]

Initialising tasks: 100% |
###########################################################################################
#| Time: 0:00:04
NOTE: Executing SetScene Tasks
NOTE: Executing RunQueue Tasks
WARNING: zlib-native-1.2.8-r0 do_fetch: Failed to fetch URL
http://downloads.sourceforge.net/libpng/zlib/1.2.8/zlib-1.2.8.tar.xz, attempting MIRRORS if
available
...
ERROR: Worker process (12167) exited unexpectedly (-9), shutting down...
ERROR: Worker process (12167) exited unexpectedly (-9), shutting down...
ERROR: Worker process (12167) exited unexpectedly (-9), shutting down...
ERROR: Worker process (12167) exited unexpectedly (-9), shutting down…

$ cat /proc/sys/vm/oom_dump_tasks /proc/sys/vm/oom_kill_allocating_task
1
0
$ cat /proc/sys/vm/overcommit_kbytes /proc/sys/vm/overcommit_memory
/proc/sys/vm/overcommit_ratio
0
```

```
0
50
$

$ dmesg
[...]
 Out of memory: Kill process 9742 (chrome) score 304 or sacrifice child
 Killed process 9742 (chrome) total-vm:886128kB, anon-rss:25372kB, file-rss:50412kB
 Purging GPU memory, 59195392 bytes freed, 135168 bytes still pinned.
 38998016 and 0 bytes still available in the bound and unbound GPU page lists.
 Purging GPU memory, 26406912 bytes freed, 135168 bytes still pinned.
 35725312 and 0 bytes still available in the bound and unbound GPU page lists.

[...]

 CPU: 2 PID: 2758 Comm: compiz Tainted: G           OE   4.4.0-62-generic #83-Ubuntu
 Hardware name: LENOVO 20FMA089IG/20FMA089IG, BIOS R06ET39W (1.13 ) 07/11/2016
  0000000000000286 0000000013e88cca ffff8802195a7510 ffffffff813f7c63
  ffff8802195a76e8 ffff880213dc5a00 ffff8802195a7580 ffffffff8120ad4e
  0000000000000015 0000000000000000 ffff880237061e40 ffff8801b00f3f08
 Call Trace:
  [<ffffffff813f7c63>] dump_stack+0x63/0x90
  [<ffffffff8120ad4e>] dump_header+0x5a/0x1c5
  [<ffffffff811926c2>] oom_kill_process+0x202/0x3c0
  [<ffffffff81192ae9>] out_of_memory+0x219/0x460
  [<ffffffff81198a5d>] __alloc_pages_slowpath.constprop.88+0x8fd/0xa70
  [<ffffffff81198e56>] __alloc_pages_nodemask+0x286/0x2a0
  [<ffffffff811e270c>] alloc_pages_current+0x8c/0x110
  [<ffffffff81196ae9>] alloc_kmem_pages+0x19/0x90
  [<ffffffff811b448e>] kmalloc_order_trace+0x2e/0xe0
  [<ffffffff811ee48e>] __kmalloc+0x22e/0x250
  [<ffffffffc03bdc7f>] ? alloc_gen8_temp_bitmaps+0x2f/0x80 [i915_bpo]
  [<ffffffffc03bdc98>] alloc_gen8_temp_bitmaps+0x48/0x80 [i915_bpo]
  [<ffffffffc03c1230>] gen8_alloc_va_range_3lvl+0xa0/0x9d0 [i915_bpo]
  [<ffffffff811aad1a>] ? shmem_getpage_gfp+0xca/0x830
  [<ffffffff811e270c>] ? alloc_pages_current+0x8c/0x110
  [<ffffffff8140892a>] ? sg_init_table+0x1a/0x40
  [<ffffffff8142160d>] ? swiotlb_map_sg_attrs+0x6d/0x130
  [<ffffffffc03c1d9f>] gen8_alloc_va_range+0x23f/0x4a0 [i915_bpo]
  [<ffffffffc03c3e8b>] i915_vma_bind+0x9b/0x180 [i915_bpo]
  [<ffffffffc03cb4e3>] i915_gem_object_do_pin+0x873/0xae0 [i915_bpo]
  [<ffffffffc03cb77d>] i915_gem_object_pin+0x2d/0x30 [i915_bpo]
  [<ffffffffc03b8d89>] i915_gem_execbuffer_reserve_vma.isra.18+0x99/0x160 [i915_bpo]
  [<ffffffffc03b91d5>] i915_gem_execbuffer_reserve.isra.19+0x385/0x3c0 [i915_bpo]
  [<ffffffffc03ba65f>] i915_gem_do_execbuffer.isra.22+0x6bf/0x11e0 [i915_bpo]
  [<ffffffffc03c4e2d>] ? i915_gem_object_get_pages_gtt+0x23d/0x3f0 [i915_bpo]
  [<ffffffffc03a9134>] ? intel_runtime_pm_put+0x64/0xc0 [i915_bpo]
  [<ffffffffc03cce0d>] ? i915_gem_pwrite_ioctl+0xdd/0x910 [i915_bpo]
  [<ffffffffc03bbe02>] i915_gem_execbuffer2+0xc2/0x1e0 [i915_bpo]
  [<ffffffffc0054752>] drm_ioctl+0x152/0x540 [drm]
  [<ffffffffc03bbd40>] ? i915_gem_execbuffer+0x330/0x330 [i915_bpo]
  [<ffffffff812260ae>] ? dput+0x1ee/0x220
  [<ffffffff8122f7a4>] ? mntput+0x24/0x40
  [<ffffffff812107f0>] ? __fput+0x190/0x220
  [<ffffffff812227af>] do_vfs_ioctl+0x29f/0x490
  [<ffffffff812108be>] ? ____fput+0xe/0x10
  [<ffffffff8109efc6>] ? task_work_run+0x86/0xa0
  [<ffffffff81222a19>] SyS_ioctl+0x79/0x90
  [<ffffffff818385f2>] entry_SYSCALL_64_fastpath+0x16/0x71
 Mem-Info:
 active_anon:271031 inactive_anon:208811 isolated_anon:0
437391 isolated_file:0
52794 writeback:0 unstable:0

[...]

$
```

Recent Update: kernel ver 4.6 released on 15 May 2016
Source
…
1.2. Improve the reliability of the Out Of Memory task killer

In previous releases, the OOM killer (which tries to kill a task to free memory) tries to kill a single task in a good hope that the task will terminate in a reasonable time and frees up its memory. In practice, it has been shown that it's easy to find workloads which break that assumption, and the OOM victim might take unbounded amount of time to exit because it might be blocked in the uninterruptible state waiting for an event which is blocked by another task looping in the page allocator. This release adds a specialized kernel thread `oom_reaper` that tries to reclaim memory by preemptively reaping the anonymous or swapped out memory owned by the OOM victim, under an assumption that such a memory won't be needed when its owner is killed anyway.

Recommended LWN article: Toward more predictable and reliable out-of-memory handling

…

SIDEBAR ***[OPTIONAL]***
Q. What are the possible ways malloc might fail?

A. (By Robert Love)

malloc signifies failure by returning NULL. There are several failure cases:

Input validation. For example, you've asked for many gigabytes of memory in a single allocation. The exact limit (if any) differs by malloc implementation; POSIX says nothing about a maximum value, short of its type being a size_t.

Allocation failures. With a data segment-based malloc, this means brk failed. This occurs when the new data segment is invalid, the system is low on memory, or the process has exceeded its maximum data segment size (as specified by RLIMIT_DATA). With a mapping-based malloc, this means mmap failed. This occurs when the process is out of virtual memory (specified by RLIMIT_AS), has exceeded its limit on mappings, or has requested too large a mapping. Note most modern Unix systems use a combination of brk and mmap to implement malloc, choosing between the two based on the size of the requested allocation.

You asked for a zero-sized allocation. Per POSIX, malloc may return NULL if you call malloc with a parameter of zero.

Most modern Unix systems allocate memory opportunistically. This means that memory is committed to during malloc, but is in fact not actually allocated until first access. In such systems, which include Linux, malloc itself will rarely fail, as it does not actually allocate any memory. Instead, the failure is pushed forward to first access. That is, malloc will return a valid pointer but a subsequent dereference of that pointer will result in SEGV or an out-of-memory (OOM) condition and process termination. This is called overcommit and is an intentional design choice. Many systems, such as Linux, let you turn off overcommit for the rare workload that desires strict committing.

Note also that malloc may legitimately crash (i.e. it is by design and thus not a bug in malloc) if there is heap corruption. For example, in many malloc implementations, double freeing a pointer results in heap corruption that may not be realized until a subsequent malloc.

SIDEBAR
Found an interesting Android application that seems to make direct use of (a variation of) the kernel OOM Killer – it's called the "Free Memory Cleaner" application. It (in it's creator's own words): "frees up memory space using the feature that system kills running apps by low memory killer when available memory is low..."!   Variation on OOM? Yup, Android kernels have a Google-enhancement called the Android "low-memory killer". It's not the same as the classic OOM-killer; the low-memory killer classifies processes into "groups" and kills those processes in the lowest (lower) priority groups. Link.

Linux Memory Overcommit

Important: malloc()'s Behaviour

As you can see from the above "*oom-killer-try*" program, malloc() (via the kernel) ***really does not allocate physical RAM at the time of the call***; it merely marks a VM page(s) as reserved by you and applies COW semantics to share it with others (tasks) like you, as long as all of you treat it as read-only! It's only when you actually write data to the page, is the physical memory actually allocated and the page(s) become truly yours (COW on private pages).

From `man malloc` (on Linux):

```
...
NOTES

 By  default,  Linux follows an optimistic memory allocation strategy.  This means
      that when malloc() returns non-NULL there is no guarantee that the memory  really
      is available.  In case it turns out that the system is out of memory, one or more
      processes will be killed by the  OOM  killer.   For  more  information,  see  the
      description   of   /proc/sys/vm/overcommit_memory   and  /proc/sys/vm/oom_adj  in
      proc(5), and the Linux kernel source file Documentation/vm/overcommit-accounting.

      Normally, malloc() allocates memory from the heap, and adjusts the  size  of  the
      heap  as  required,  using sbrk(2).  When allocating blocks of memory larger than
      MMAP_THRESHOLD bytes, the glibc malloc() implementation allocates the memory as
      a private  anonymous  mapping  using mmap(2).  MMAP_THRESHOLD is 128 kB by
      default, but is adjustable using mallopt(3).   Allocations  performed  using  mmap(2)  are
      unaffected by the RLIMIT_DATA resource limit (see getrlimit(2)).

     [...]

      Crashes  in malloc(), calloc(), realloc(), or free() are almost always related to
      heap corruption, such as overflowing an  allocated  chunk  or  freeing  the  same
      pointer twice.

      The  malloc() implementation is tunable via environment variables; see mallopt(3)
      for details.
```

One can switch off this overcommitting behavior using a command like

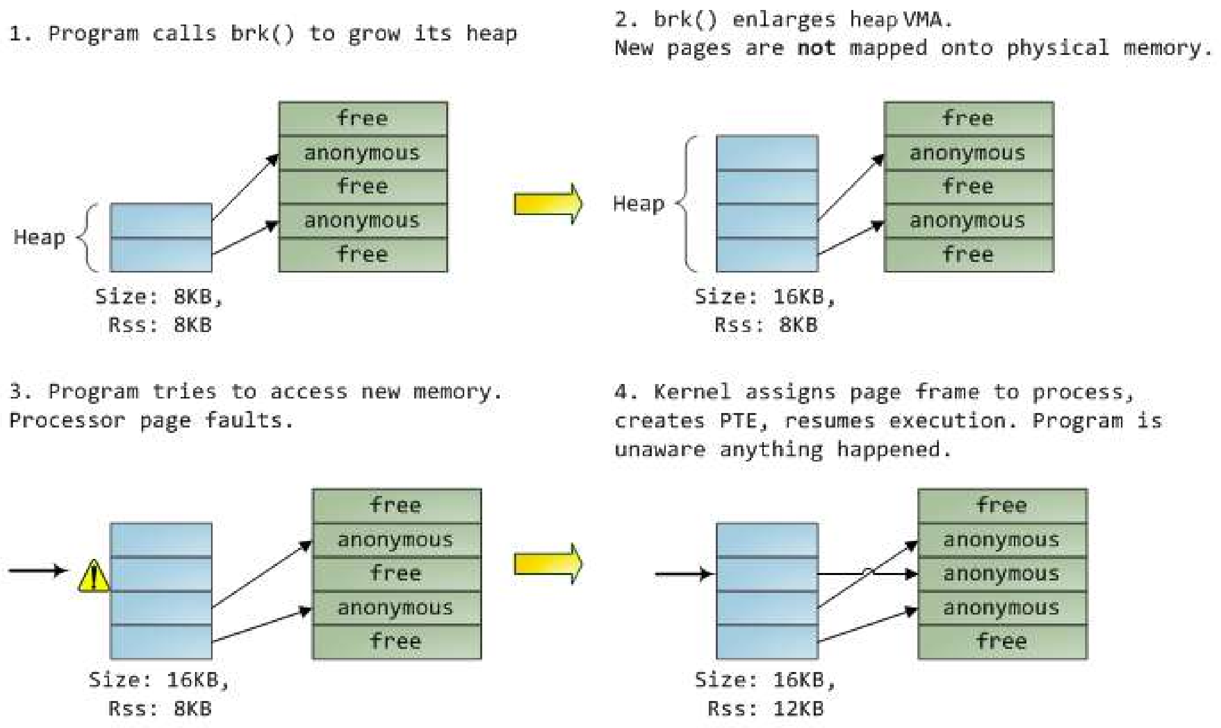
```
        # echo 2 > /proc/sys/vm/overcommit_memory
```

See also the kernel Documentation directory, files *vm/overcommit-accounting* and *sysctl/vm.txt* .

From one of Gustav Duarte's excellent blog articles:

--snip--

A VMA is like a contract between your program and the kernel. You ask for something to be done (memory allocated, a file mapped, etc.), the kernel says “sure”, and it creates or updates the appropriate VMA. But it does not actually honor the request right away, it waits until a page fault happens to do real work. The kernel is a lazy, deceitful sack of scum; this is the fundamental principle of virtual memory. It applies in most situations, some familiar and some surprising, but the rule is that VMAs record what has been agreed upon, while PTEs reflect what has actually been done by the lazy kernel. These two data structures together manage a program's memory; both play a role in resolving page faults, freeing memory, swapping memory out, and so on.


1. Program calls brk() to grow its heap
2. brk() enlarges heap VMA. New pages are not mapped onto physical memory.
3. Program tries to access new memory. Processor page faults.
4. Kernel assigns page frame to process, creates PTE, resumes execution. Program is unaware anything happened.
Heap
free
anonymous
Size: 8KB, Rss: 8KB
Size: 16KB, Rss: 8KB
Size: 16KB, Rss: 12KB

--snip--

Resources:
http://linuxtoolkit.blogspot.in/2011/08/tweaking-linux-kernel-overcommit.html
When Linux runs out of memory

Source: Professional Linux Kernel Architecture by Wolfgang Mauerer, Wrox Press.

…

sysctl_overcommit_memory can be set with the help of the *\/proc/sys/vm/overcommit_memory* .

```
# cat /etc/issue
Linux Mint 17 Qiana \n \l
# uname -r
3.13.0-24-generic
# cat /proc/sys/vm/overcommit_memory
0
# cat /proc/sys/vm/overcommit_ratio
50
#
```

Currently there are three overcommit options:

1 : allows an application to allocate as much memory as it wants, even more than is permitted by the address space of the system.

0 : << the default >> means that heuristic overcommitting is applied with the result that the number of usable pages is determined by adding together the pages in the page cache, the pages in the swap area, and the unused page frames; requests for allocation of a smaller number of pages are permitted.

2 : stands for the strictest mode, known as strict overcommitting, in which the permitted number of pages that can be allocated is calculated as follows:

```
allowed = (totalram_pages - hugetlb) * sysctl_overcommit_ratio / 100;
allowed += total_swap_pages;
```

Here sysctl_overcommit_ratio is a configurable kernel parameter that is usually set to 50. If the total number of pages used exceeds this value, the kernel refuses to perform further allocations.

Why does it make sense to allow an application to allocate more pages than can ever be handled in principle? This is sometimes required for scientific applications. Some tend to allocate huge amounts of memory without actually requiring it — but, in the opinion of the application authors, it seems good to have it just in case. If the memory will, indeed, never be used, no physical page frames will ever be allocated, and no problem arises.

Such a programming style is clearly bad practice, but unfortunately this is often no criterion for the value of software. Writing clean code is usually not rewarding in the scientific community outside computer science. There is only immediate interest that a program works for a given configuration, while efforts to make programs future-proof or portable do not seem to provide immediate benefits and are therefore often not valued at all.

...

From *Documentation/sysctl/vm.txt* :
==
...
overcommit_memory:

This value contains a flag that enables memory overcommitment.

When this flag is 0, the kernel attempts to estimate the amount of free memory left when userspace requests more memory.

When this flag is 1, the kernel pretends there is always enough memory until it actually runs out.

When this flag is 2, the kernel uses a "never overcommit" policy that attempts to prevent any overcommit of memory.

This feature can be very useful because there are a lot of programs that malloc() huge amounts of memory "just-in-case" and don't use much of it.

The default value is 0.

See *Documentation/vm/overcommit-accounting* and *smm/mmap.c::__vm_enough_memory()* for more information.

==

overcommit_ratio:

When overcommit_memory is set to 2, the committed address space is not permitted to exceed swap plus this percentage of physical RAM. See above.

==
…

Useful Reference on the linux-mm wiki site:
OverCommit Accounting

Interesting, see:
Quora: What are the disadvantages of disabling memory overcommit in Linux?

Snippets:
…
Nelson Elhage, Kernel hacker and security guy.
Written Jan 25, 2012

As I understand it, one of the biggest issues that, without memory overcommit, fork() must be accounted for as though it was *not* copy-on-write, since in theory the child could scribble over every writable page in its address space, and the kernel would have to allocate fresh pages for all of them.

Thus, a fork()ing server that spawns a new child for every request, or even has a request pool of N children, will effectively reserve N times as much memory (now unusable by anyone else), even though in practice the workers will never un-CoW most of those pages.

Similarly, as User mentioned, a process with a 10GB heap on a machine with 15G of RAM can't fork() and exec() children at all, since the kernel has no way of knowing that the child after the fork() won't go ahead and touch all 10G of that heap, and so the kernel must refuse the fork().

…

As an anecdote, I had coworker once who did make a point of turning off overcommit on his laptop. Almost everything worked fine, but he was unable to run **gitk** on linux-2.6.git, even though it worked fine under near-identical conditions with overcommit turned on. The problem, of course, is exactly the one I just described: gitk would consume around 2GB of his 4GB of RAM, and when it tried to fork() and exec() **git** to read information about the repository, the kernel would not allow it, because it had to account for the possibility of the **gitk** child consuming another 2GB.

As far as I can tell, many people who claim "malloc() should just return NULL" have not thought about the problem very hard, yet. The real issue is not just related to malloc(), but to the subtler issues of CoW mappings and other lazily-allocated mappings (including the anonymous mmap()s that often back malloc()).

…

This is not to say that overcommit is always the right answer -- in e.g. embedded environments, or tightly controlled and carefully provisioned server environments, where you can enforce rules about which primitives and workloads actually happen in practice, you have more options, and/or it may be more important to make sure that the OOM killer never strikes.

Mark Hahn, computer guy
Written Jan 26, 2012

first, fork does not cause a giant copy as described.

the default vm.overcommit_memory mode works well, because it's very normal for processes to have only a fraction of their virtual address space resident. do this:

```
ps -A --no-headers -o rss,vsz|awk '{r+=$1;v+=$2}END{print r,v,r/v}'
```

on my desktop, rss is 12% of vsz (in total). setting vm.overcommit=2 is the conservative approach, which lets you guarantee never to OOM. instead, an allocation too far will indeed return an error, which the program can handle.
…

[
On my Ubuntu 16.04 LTS desktop with 8GB RAM:

```
$ ps -A --no-headers -o rss,vsz|awk '{r+=$1;v+=$2}END{print r,v,r/v}'
6493128 79326172 0.0818535
$
```

Implying that the ratio of RSS:VSZ at this point in time is just about 8% !

]

Page Faults

Source: “Linux Kernel's Memory Management Unit API” by Bill Gatliff.

When a process tries to access memory in a page that is not known to the MMU, the MMU generates a page fault exception. The page fault exception handler examines the state of the MMU hardware and the currently running process’s memory information, and determines whether the fault is a "good" one, or a "bad" one. Good page faults cause the handler to give more memory to the process; bad faults cause the handler to terminate the process.

Good page faults are expected behavior, and occur whenever a program allocates dynamic memory, runs a section of code or writes a section of data for the first time, or increases its stack size. When the *process attempts to access* this new memory, the MMU declares a page fault, and the OS adds a fresh page of memory to the process’s page table. The interrupted process is then resumed.

Bad faults occur when a process follows a NULL pointer, or tries to access memory that it doesn’t own. Bad faults can also occur due to programming bugs in the kernel, in which case the handler will print an "oops" message before terminating the process.

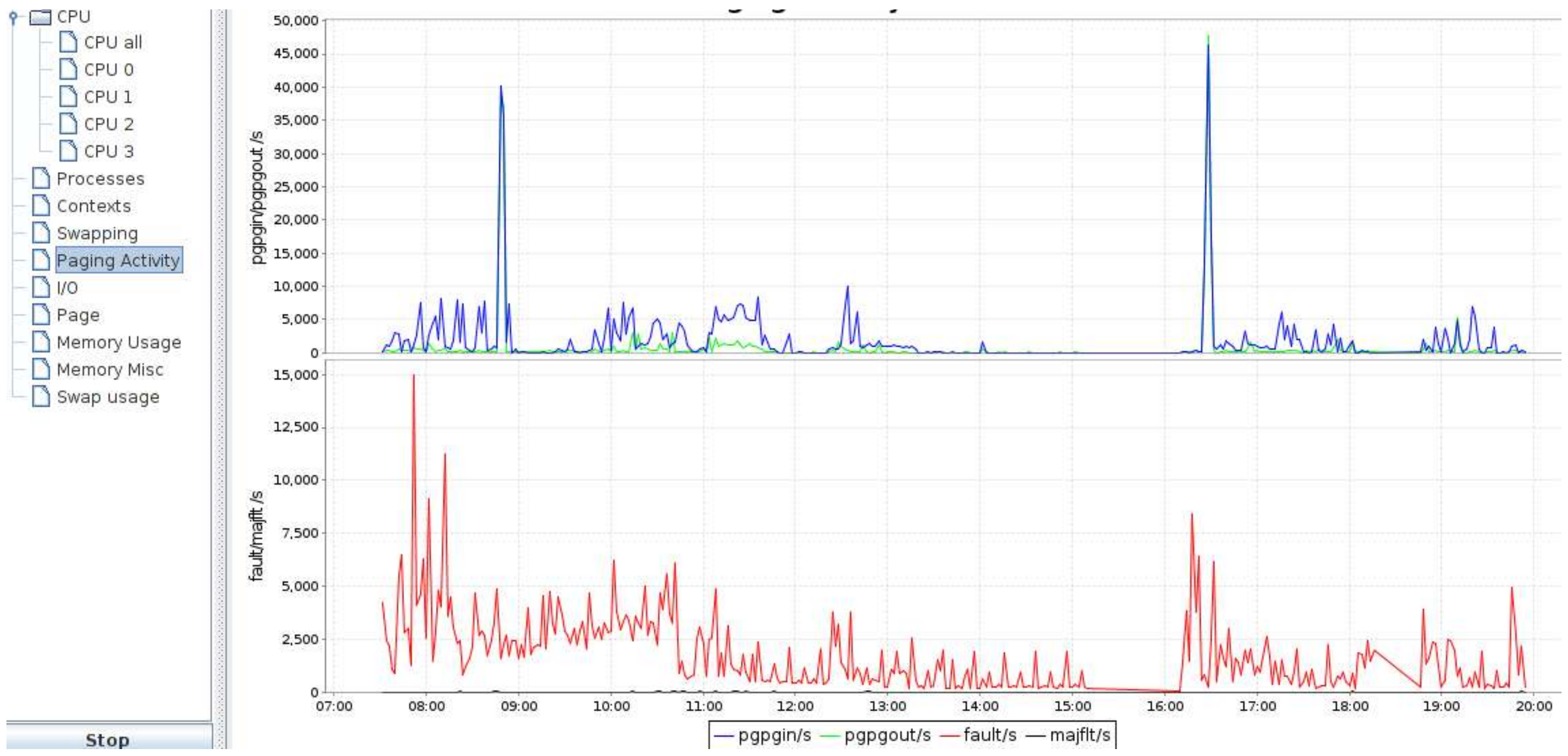
<<

The SAR (System Activity Report) project is useful to get a birds-eye view of system activity (usually for servers). Your author setup sar and captured the day’s stats; then viewed them via a useful GUI tool – kSar.

We take a peek at the ‘Paging Activity’ column and it’s graphing: the interesting thing to notice now:

- the high number of *minor* faults / sec (bottom graph red colour) occuring pretty often) – it’s “normal behavior”!
- The low number of *major* faults / sec (bottom graph black colour – it’s almost non-exitstant in fact); major faults imply a disk access and thus a major slowdown.

See the screenshot below:



>>

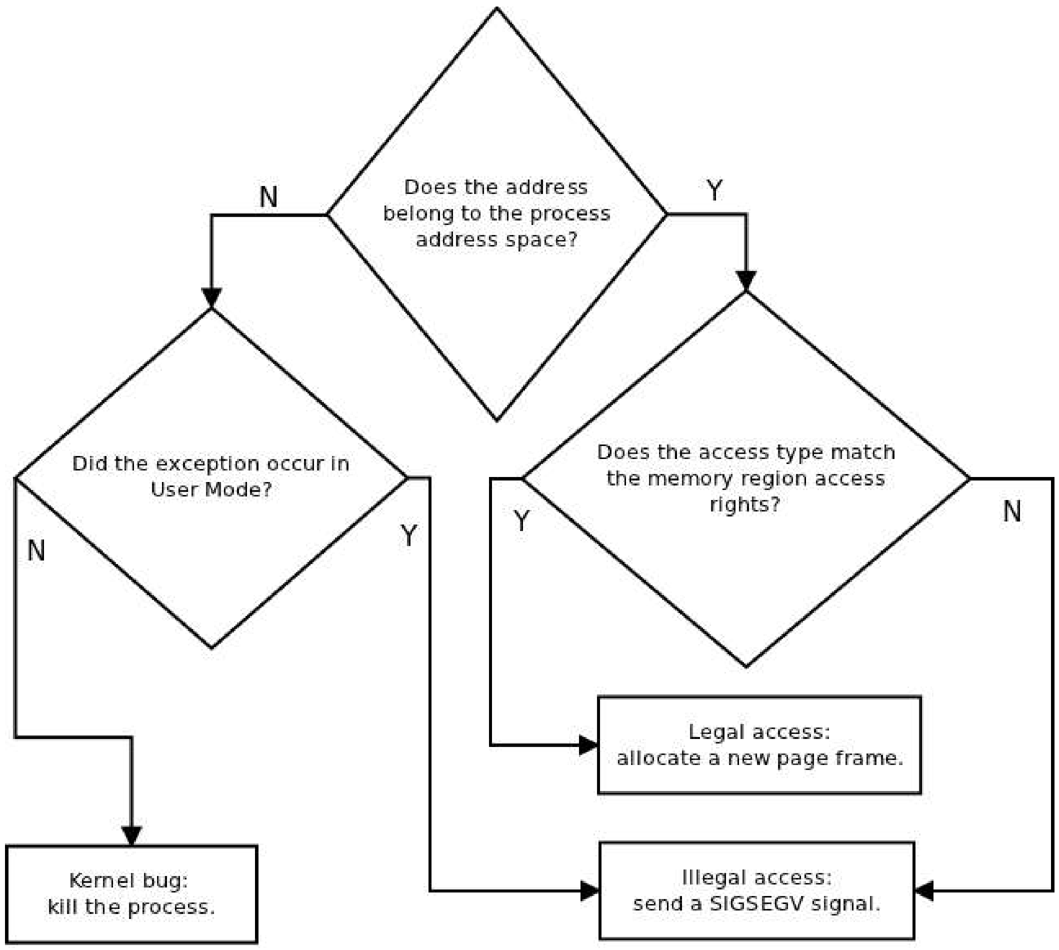
The Page Fault Exception Handler – In Brief

Source: ULK3

--snip--

The Linux Page Fault exception handler must distinguish exceptions caused by programming errors from those caused by a reference to a page that legitimately belongs to the process address space but simply hasn't been allocated yet.

The memory region descriptors allow the exception handler to perform its job quite efficiently. The `do_page_fault()` function, which is the Page Fault interrupt service routine for the 80 x 86 architecture, compares the linear address that caused the Page Fault against the memory regions of the `current` process; it can thus determine the proper way to handle the exception according to the scheme that is illustrated in the figure below.


Does the address belong to the process address space?
N
Y
Did the exception occur in User Mode?
N
Y
Does the access type match the memory region access rights?
Y
N
Legal access: allocate a new page frame.
Kernel bug: kill the process.
Illegal access: send a SIGSEGV signal.

Overall Scheme for the Page Fault handler

In practice, things are a lot more complex because the Page Fault handler must recognize several particular subcases that fit awkwardly into the overall scheme, and it must distinguish several kinds of legal access.

--snip--

Source – Memory FAQ
…

What is a page fault handler?

```
A page fault handler is an interrupt routine, called by the Memory
Management Unit in response an attempt to access virtual memory which
did not immediately succeed.

When a program attempts to read, write, or execute memory in a page that
```

hasn't got the appropriate permission bits set in its page table entry to allow that type of access, the instruction generates an interrupt. This calls the page fault handler to examines the registers and page tables of the interrupted process and determine what action to take to handle the fault.

The page fault handler may respond to a page fault in three ways:

1) The page fault handler can resolve the fault by immediately attaching a page of physical memory to the appropriate page table entry, adjusting the entry, and resuming the interrupted instruction. This is called a "soft fault".

2) When the fault handler can't immediately resolve the fault, it may suspend the interrupted process and switch to another while the system works to resolve the issue. This is called a "hard fault", and results when an I/O operation most be performed to prepare the physical page needed to resolve the fault.

3) If the page fault handler can't resolve the fault, it sends a signal (SIGSEGV) to the process, informing it of the failure. Although a process can install a SIGSEGV handler (debuggers and emulators tend to do this), the default behavior of an unhandled SIGSEGV is to kill the process with the message "bus error".

A page fault that occurs in an interrupt handler is called a "double fault", and usually panics the kernel. The double fault handler calls the kernel's panic() function to print error messages to help diagnose the problem. The process to be killed for accessing memory it shouldn't is the kernel itself, and thus the system is too confused to continue.

http://en.wikipedia.org/wiki/Double_fault

A triple fault cannot be handled in software. If a page fault occurrs in the double fault handler, the machine immediately reboots.

http://en.wikipedia.org/wiki/Triple_fault

How does the page fault handler allocate physical memory?

The Linux kernel uses lazy (on-demand) allocation of physical pages, deferring the allocation until necessary and avoiding allocating physical pages which will never actually be used.

Memory mappings generally start out with no physical pages attached. They define virtual address ranges without any associated physical memory. So malloc() and similar allocate space, but the actual memory is allocated later by the page fault handler.

Virtual pages with no associated physical page will have the read, write, and execute bits disabled in their page table entries. This causes any access to that address to generate a page fault, interrupting the program and calling the page fault handler.

When the page fault handler needs to allocate physical memory to handle a page fault, it zeroes a free physical page (or grabs a page from a pool of prezeroed pages), attaches that page of memory to the Page Table Entry associated with the fault, updates that PTE to allow the appropriate access, and resumes the faulting instruction.

Note that implementing this requires two sets of page table flags for read, write, execute, and share. The VM_READ, VM_WRITE, VM_EXEC, and VM_SHARED flags (in linux/mm.h) are used by the MMU to generate faults. The VM_MAYREAD, VM_MAYWRITE, VM_MAYEXEC, and VM_MAYSHARE flags are used by the page fault handler to determine whether the attempted access was legal and thus the fault handler should adjust the PTE to resolve the fault and allow the process to continue.

How does fork work?

The fork() system call creates a new process by copying an existing process. A new process is created with a copy of the page tables that calls fork(). These page tables are all copy on write mappings sharing the existing physical pages between parent and child.

How does exec work?

…

Minimally Testing the MMU and Linux OS page fault handling

The user-mode C program below will attempt various illegal memory accesses.

The MMU will thus trigger an appropriate exception condition and invoke the OS's page fault handler. The OS page fault handler will go through a very detailed algorithm (short flowchart version shown above), in order to determine the actual fault (minor or major, good or bad) and, in these cases, being a usermode 'bad' fault, will send SIGSEGV to 'current'!

```
$ cat segv_pgfault.c
/*
 * segv_pgfault.c
 *
 * Make a usermode process segfault by accessing invalid user/kernel-space addresses..
 * This in turn will have the MMU trigger an exception condition (Data Abort on
 * ARM), which will lead to the OS's page fault handler being invoked. *It* will
 * determine the actual fault (minor or major, good or bad) and, in this case, being
 * a usermode 'bad' fault, will send SIGSEGV to 'current'!
 *
 * Author(s) :  Kaiwan NB
 * License(s): MIT
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
#include <string.h>
#include <ctype.h>

/*---------------- Typedef's, constants, etc ------------------------*/
typedef unsigned int u32;
typedef long unsigned int u64;

/*---------------- Macros -------------------------------------------*/
#if __x86_64__   /* 64-bit; __x86_64__ works for gcc */
 #define ADDR_TYPE u64
 #define ADDR_FMT "%016lx"
 static u64 rubbish_uaddr = 0x100000L;
 static u64 rubbish_kaddr = 0xffff0a8700100000L;
#else
 #define ADDR_TYPE u32
 #define ADDR_FMT "%08lx"
 static u32 rubbish_uaddr = 0x100000L;
 static u32 rubbish_kaddr = 0xd0100000L;
#endif


/*---------------- Functions ----------------------------------------*/
static void myfault(int signum, siginfo_t * siginfo, void *rest)
{
	static int c = 0;

	printf("*** %s: [%d] received signal %d. errno=%d\n"
	       " Cause/Origin: (si_code=%d): ",
	       __func__, ++c, signum, siginfo->si_errno, siginfo->si_code);

	switch (siginfo->si_code) {
	case SI_USER:
		printf("user\n");
		break;
	case SI_KERNEL:
		printf("kernel\n");
		break;
	case SI_QUEUE:
		printf("queue\n");
		break;
	case SI_TIMER:
		printf("timer\n");
		break;
	case SI_MESGQ:
		printf("mesgq\n");
		break;
	case SI_ASYNCIO:
		printf("async io\n");
		break;
	case SI_SIGIO:
		printf("sigio\n");
		break;
	case SI_TKILL:
		printf("t[g]kill\n");
		break;
		// other poss values si_code can have for SIGSEGV
	case SEGV_MAPERR:
		printf("SEGV_MAPERR: address not mapped to object\n");
		break;
	case SEGV_ACCERR:
		printf("SEGV_ACCERR: invalid permissions for mapped object\n");
		break;
	default:
		printf("-none-\n");
	}
	printf(" Faulting addr=0x" ADDR_FMT "\n", (ADDR_TYPE) siginfo->si_addr);

#if 1
	exit (1);
#else
	abort();
#endif
}

static void usage(char *nm)
{
	fprintf(stderr, "Usage: %s u|k r|w\n"
		"u => user mode\n"
		"k => kernel mode\n"
		" r => read attempt\n" " w => write attempt\n", nm);
}
```

```
int main(int argc, char **argv)
{
	struct sigaction act;

	if (argc != 3) {
		usage(argv[0]);
		exit(1);
	}
	act.sa_sigaction = myfault;
	act.sa_flags = SA_RESTART | SA_SIGINFO;
	sigemptyset(&act.sa_mask);
	if (sigaction(SIGSEGV, &act, 0) == -1) {
		perror("sigaction");
		exit(1);
	}

	if ((tolower(argv[1][0]) == 'u') && tolower(argv[2][0] == 'r')) {
		ADDR_TYPE *uptr = (ADDR_TYPE *) rubbish_uaddr;
						// an arbitrary userspace virtual addr
		printf
		    ("Attempting to read contents of arbitrary usermode"
		     "va uptr = 0x" ADDR_FMT ":\n",
		     (ADDR_TYPE) uptr);
		printf("*uptr = 0x" ADDR_FMT "\n", *uptr);	// just reading
	} else if ((tolower(argv[1][0]) == 'u') && tolower(argv[2][0] == 'w')) {
		ADDR_TYPE *uptr = (ADDR_TYPE *) & main;
		printf
		    ("Attempting to write into arbitrary usermode"
		     "va uptr (&main actually) = 0x" ADDR_FMT ":\n",
		     (ADDR_TYPE) uptr);
		*uptr = 40;	// writing
	} else if ((tolower(argv[1][0]) == 'k') && tolower(argv[2][0] == 'r')) {
		ADDR_TYPE *kptr = (ADDR_TYPE *) rubbish_kaddr;
						// arbitrary kernel virtual addr
		printf
		    ("Attempting to read contents of arbitrary"
		     "kernel va kptr = 0x" ADDR_FMT ":\n",
		     (ADDR_TYPE) kptr);
		printf("*kptr = 0x" ADDR_FMT "\n", *kptr);	// just reading
	} else if ((tolower(argv[1][0]) == 'k') && tolower(argv[2][0] == 'w')) {
		ADDR_TYPE *kptr = (ADDR_TYPE *) rubbish_kaddr;
						// arbitrary kernel virtual addr
		printf
		    ("Attempting to write into arbitrary kernel va kptr = 0x"
		       ADDR_FMT ":\n",
		     (ADDR_TYPE) kptr);
		*kptr = 0x62;	// writing
	} else
		usage(argv[0]);
	exit(0);
}

/* vi: ts=4 */
```

```
$ ./segv_pgfault
Usage: ./segv_pgfault u|k r|w
```

```
u => user mode
k => kernel mode
 r => read attempt
 w => write attempt
$

On an x86_64
$ ./segv_pgfault u r
Attempting to read contents of arbitrary usermode va uptr = 0x0000000000100000:
*** myfault: [1] received signal 11. errno=0
 Cause/Origin: (si_code=1): SEGV_MAPERR: address not mapped to object
 Faulting addr=0x0000000000100000
$
$ ./segv_pgfault u w
Attempting to write into arbitrary usermode va uptr (&main actually) =
0x0000000000400904:
*** myfault: [1] received signal 11. errno=0
 Cause/Origin: (si_code=2): SEGV_ACCERR: invalid permissions for mapped object
 Faulting addr=0x0000000000400904
$
$ ./segv_pgfault k r
Attempting to read contents of arbitrary kernel va kptr = 0xffff0a8700100000:
*** myfault: [1] received signal 11. errno=0
 Cause/Origin: (si_code=128): kernel
 Faulting addr=0x0000000000000000   << security: the kernel va is not revealed >>
$
$ ./segv_pgfault k w
Attempting to write into arbitrary kernel va kptr = 0xffff0a8700100000:
*** myfault: [1] received signal 11. errno=0
 Cause/Origin: (si_code=128): kernel
 Faulting addr=0x0000000000000000   << security: the kernel va is not revealed >>
$
```

SIDEBAR :: KSM (Kernel Samepage Merging)

Source

Kernel SamePage Merging (KSM) (also: Kernel Shared Memory, Memory Merging) lets the hypervisor system share identical memory pages amongst different processes or virtualized guests. This is done by scanning through the memory finding duplicate pages. The duplicate pair is then merged into a single page, and mapped into both original locations. The page is also marked as "copy-on-write", so the kernel will automatically separate them again should one process modify its data.[1]

KSM was originally intended to run more virtual machines on one host by sharing memory between processes as well as virtual machines. Upon implementation, it was found to be useful for non-virtualized environments as well where memory is at a premium.[2][3] An experimental implementation of KSM by Red Hat found that 52 virtual instances of Windows XP with 1GB of memory, could run on a host computer that had only 16GB of RAM.[4]

...

Mainlined in 2.6.32.

KSM merges private anonymous pages, not pagecache ones. It’s very useful to any application which generates many instances of the same data – QEMU running guest VMs and using *kvm* for acceleration – is a great example.

Source

Enabling KSM

KSM only operates on those areas of address space which an application has advised to be likely candidates for merging, by using the madvise(2) system call:

int madvise(addr, length, MADV_MERGEABLE).

The app may call int madvise(addr, length, MADV_UNMERGEABLE) to cancel that advice and restore unshared pages: whereupon KSM unmerges whatever it merged in that range. Note: this unmerging call may suddenly require more memory than is available - possibly failing with EAGAIN, but more probably arousing the Out-Of-Memory killer.

More regarding KSM:

```
$ man madvise
NAME
       madvise - give advice about use of memory

SYNOPSIS
```

```
       #include <sys/mman.h>

       int madvise(void *addr, size_t length, int advice);
…
...
  MADV_MERGEABLE (since Linux 2.6.32)
              Enable Kernel Samepage Merging (KSM) for the pages in the range
specified by addr and length.  The kernel regularly scans those areas
of user memory that have been marked as mergeable, looking for pages with
identical content.  These are replaced by a  single  write-
protected  page  (which is automatically copied if a process later wants to
update the content of the page).

KSM merges only private anonymous pages (see mmap(2)).  The KSM feature is
intended for applications that generate many instances of  the  same  data
(e.g., virtualization systems such as KVM).

It can consume a lot of processing power; use with care.  See the Linux
kernel source file Documentation/vm/ksm.txt for more details.

The MADV_MERGEABLE and MADV_UNMERGEABLE operations are available only if the
kernel was configured with CONFIG_KSM.

  MADV_UNMERGEABLE (since Linux 2.6.32)
              Undo  the  effect of an earlier MADV_MERGEABLE operation on the
specified address range; KSM unmerges whatever pages it had merged in the
address range specified by addr and length.
…
```

Useful!
Excellent: “Best Practices for KVM”, IBM whitepaper. [PDF]
Relevant section: “Best practice: Over-commit memory resources by using page sharing or ballooning” pg 11-17 (covers using KSM for VMs best practices).

You can verify KSM is in action, by checking for the existence of some of its /sys files, under /sys/kernel/mm/ksm/

They are:

pages_shared	how many shared pages are being used
pages_sharing	how many more sites are sharing them i.e. how much saved
pages_unshared	how many pages unique but repeatedly checked for merging
pages_volatile	how many pages changing too fast to be placed in a tree
full_scans	how many times all mergeable areas have been scanned

[FYI / OPTIONAL]

Linux glibc malloc() Behaviour

The *first time* a process does an

`malloc(8);`

- the heap memory is not *physically* allocated yet, only *virtually* (remember, the kernel is a lazy guy!). (Technically, the kernel assigns or updates a VMA for this virtual region, marking it read-write, but *the PTE entry as read-only*. This is deliberate: it allows the kernel to catch the fault that will occur).

- Then, later, when the process attempts to write any address in this region, the MMU receives the virtual address. Now, the MMU will lookup the page tables to translate the virtual to a physical address; at this point, it will find the PTE is marked as read-only but a write is being attempted. Thus, the MMU raises a fault. Control goes to the the kernel's fault handler code; it "realizes" this is a *good fault* and allocates a page farme from the slab cache (possibly via the buddy allocator).
 This is mapped into the faulting process's heap..

- Another request is made by the process, say:
  malloc(32);

- Now, since the glibc memory manager (what we call malloc) now knows that a full page is available in the heap and only 8 bytes are actually used up, this request will *not* cause a brk() syscall to the kernel. This can be verified by checking the current break (using sbrk(0)).

  Also, seeking to within a page of the first request will – possibly, even probably - not cause a segfault even if the memory has not been programatically allocated! In fact, one finds that:

a) glibc allocates a lot more than a page in the heap; it's often in the region of 130 Kb or so

b) recent glibc does *not* grow the heap (and thus the data segment) when servicing malloc/calloc requests of more than MMAP_THRESHOLD (default: 128 kB). Instead, it uses the mmap(2) system call to setup a virtual region of the requested size. This is considered more efficient.

See below a program that helps us study how the system allocates memory dynamically to user-space applications via the usual malloc / calloc API.

```
$ cat malloc_brk_test.c
/*
 * malloc_brk_test.c
 * (c) Kaiwan NB.
 * License: MIT
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <malloc.h>

//#define USE_CALLOC
#undef USE_CALLOC

#define TRIES 5
void *heap_ptr[TRIES];
void *init_brk;

static void alloctest(int index, size_t num)
{
	void *p;

#ifndef USE_CALLOC
	p = malloc(num);
#else
	p = calloc(num, sizeof(char));
#endif
	if (!p) { printf("out of memory!\n"); exit(1); }

	heap_ptr[index] = p; // save ptr in order to free later..
#ifndef USE_CALLOC
	printf("\n%d: malloc", index);
#else
	printf("\n%d: calloc", index);
#endif
	printf("(%6u) successful. Heap pointer: %8p\n", num, p);
	printf("Current break: %8p [delta: %d]\n", sbrk(0), (sbrk(0)-init_brk));

	malloc_stats();
}

int main (int argc, char **argv)
{
	int i;
	char *q;

	init_brk = sbrk(0);
	printf("Current break: %8p\n", init_brk);
	alloctest(0, 8);

	q=heap_ptr[0];
	*(q+3000) = 'a'; /* "should" segfault but (probably) does not
```

```
                         bcoz a *page* (or more) is allocated
                          by the previous alloc, not just 8 bytes!
                         See value of prg break compared to this
                         pointer.
                           */
        printf("(q+3000) is the mem loc 0x%x. Mem here is: 0x%08x\n",
              (unsigned int)(q+3000), *(q+3000));

        alloctest(1, (getpagesize()-8-5));
        alloctest(2, 3);
        alloctest(3, (sbrk(0)-init_brk+1000));
        alloctest(4, 200000);

        for (i=0; i<TRIES; i++)
              free(heap_ptr[i]);
        exit (0);
}
/* vi: ts=4 */
$ ./malloc_brk_test
Current break: 0x8e3e000

0: malloc(     8) successful. Heap pointer: 0x8e3e008
Current break: 0x8e5f000 [delta: 135168]
Arena 0:
system bytes     =     135168
in use bytes     =         16
Total (incl. mmap):
system bytes     =     135168
in use bytes     =         16
max mmap regions =          0
max mmap bytes   =          0
(q+3000) is the mem loc 0x8e3ebc0. Mem here is: 0x00000061 << the
                                                'a' that we wrote in >>

1: malloc(  4083) successful. Heap pointer: 0x8e3e018
Current break: 0x8e5f000 [delta: 135168]
Arena 0:
system bytes     =     135168
in use bytes     =       4104
Total (incl. mmap):
system bytes     =     135168
in use bytes     =       4104
max mmap regions =          0
max mmap bytes   =          0

2: malloc(     3) successful. Heap pointer: 0x8e3f010
Current break: 0x8e5f000 [delta: 135168]
Arena 0:
system bytes     =     135168
in use bytes     =       4120
Total (incl. mmap):
system bytes     =     135168
in use bytes     =       4120
max mmap regions =          0
max mmap bytes   =          0
```

<< The 2nd and 3rd allocation above do not cause the brk syscall to be invoked, as there is sufficient memory on the heap to service the requests >>

```
3: malloc(136168) successful. Heap pointer: 0xb77f5008
Current break: 0x8e5f000 [delta: 135168]
Arena 0:
system bytes     =     135168
in use bytes     =       4120
Total (incl. mmap):
system bytes     =     274432
in use bytes     =     143384
max mmap regions =          1
max mmap bytes   =     139264

<< This (above) allocation request is a large one: ~132Kb. The 'mmap threshold' is
(default) 128Kb; thus, this causes an mmap() to the process virtual address space,
mapping in the virtually allocated region (which will later be mapped to physical
page frames via the MMU page-faulting on application access to these memory regions!
>>

4: malloc(200000) successful. Heap pointer: 0xb77c4008
Current break: 0x8e5f000 [delta: 135168]
Arena 0:
system bytes     =     135168
in use bytes     =       4120
Total (incl. mmap):
system bytes     =     475136
in use bytes     =     344088
max mmap regions =          2
max mmap bytes   =     339968

<< Similarly, the large alloc above causes another mmap to occour >>
$
```

See the man page on mallopt(3) for details.

When malloc runs out of heap space it has to resort to invoking the system call sbrk(). In reality, sbrk is a wrapper: it invokes the actual system call brk(2):

```
malloc() → sbrk() → brk() → sys_brk() → do_brk()
 \---      User Mode     ---/     \---   Kernel Mode     ---/
```

do_brk then either grows the existing VMA, or if it cannot, sets up a new VMA to mark the region as allocated.

If the requested allocation is >= MMAP_THRESHOLD (128Kb default), malloc internally uses the mmap(2) to setup a new VMA representing the (allocated) virtual address region.

For a (much) deeper understanding of the Linux heap, pl see:
Understanding the Heap & Exploiting Heap Overflows, Mathy Vanhoef
(good explanation, though the technique described is now a bit outdated by modern glibc).

Source – Memory FAQ

…

What are memory mappings?

```
  A memory mapping is a set of page table entries describing the properties
  of a consecutive virtual address range.  Each memory mapping has a
  start address and length, permissions (such as whether the program can
  read, write, or execute from that memory), and associated resources (such
  as physical pages, swap pages, file contents, and so on).

  Creating new memory mappings allocates virtual memory, but not physical
  memory (except for a small amount of physical memory needed to store the
  page table itself).  Physical pages are attached to memory mappings later,
  in response to page faults.  Physical memory is allocated on-demand by the
  page fault handler as necessary to resolve page faults.

  A page table can be thought of as a description of a set of memory
  mappings.  Each memory mapping can be anonymous, file backed, device backed,
  shared, or copy on write.

  The mmap() system call adds a new memory mapping to the current process's
  page tables.  The munmap() system call discards an existing mapping.

  Memory mappings cannot overlap << they are “tiled” >>.  The mmap() call
  returns an error if asked to create overlapping memory mappings.

  Virtual address ranges for which there is no current memory mapping are said
  to be "unmapped", and attempts to access them generate a page fault which
  cannot be handled.  The page fault handler sends a Segmentation
  Violation signal (SIGSEGV) to the program on any access to unmapped
  addresses.  This is generally the result of following a "wild pointer".

  Note that by default, Linux intentionally leaves the first few kilobytes
  (or even megabytes) of each process's virtual address space unmapped, so
  that attempts to dereference null pointers generate an unhandled page
  fault resulting in an immediate SIGSEGV, killing the process.
```

…

What is an anonymous mapping?

```
  Anonymous memory is a memory mapping with no file or device backing it.
  This is how programs allocate memory from the operating system for use
  by things like the stack and heap.

  Initially, an anonymous mapping only allocates virtual memory.  The new
```

mapping starts with a redundant copy on write mapping of the zero page. (The zero page is a single page of physical memory filled with zeroes, maintained by the operating system.) Every virtual page of the anonymous mapping is attached to this existing prezeroed page, so attempts to read from anywhere in the mapping return zeroed memory even though no new physical memory has been allocated to it yet.

Attempts to write to the page trigger the normal copy-on-write mechanism in the page fault handler, allocating fresh memory only when needed to allow the write to proceed. (Note, prezeroing optimizations change the implementation details here, but the theory's the same.) Thus "dirtying" anonymous pages allocates physical memory, the actual allocation call only allocates virtual memory.

Dirty anonymous pages can be written to swap space, but in the absence of swap they remain "pinned" in physical memory.

Anonymous mappings may be created by passing the MAP_ANONYMOUS flag to mmap().

What is a file backed mapping?

File backed mappings mirror the contents of an existing file. The mapping has some administrative data noting which file to map from, and at which offset, as well as permission bits indicating whether the pages may be read, written, or executed.

When page faults attach new physical pages to such a mapping, the contents of those pages is initialized by reading the contents of the file being mapped, at the appropriate offset for that page.

These physical pages are usually shared with the page cache, the kernel's disk cache of file contents. The kernel caches the contents of files when the page is read, so sharing those cache pages with the process reduces the total number of physical pages required by the system.

<< The “promise” of the page cache: exactly one physical page frame will be used at any point in time to map a page of a file. >>

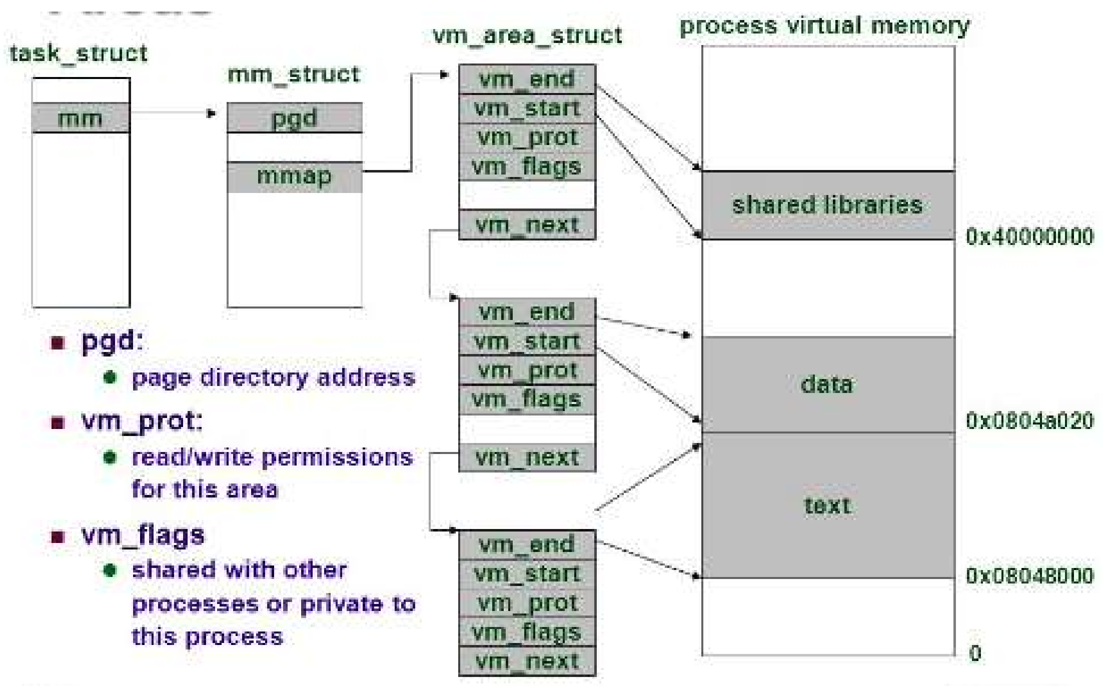
Writes to file mappings created with the MAP_SHARED flag update the page cache pages, making the updated file contents immediately visible to other processes using the file, and eventually the cache pages will be flushed to disk updating the on-disk copy of the file.

Writes to file mappings created with the MAP_PRIVATE flag perform a copy on write, allocating a new local copy of the page to store the changes. These changes are not made visible to other processes, and do not update the on-disk copy of the file.

```
Note that this means writes to MAP_SHARED pages do not allocate additonal
physical pages (the page was already faulted into the page cache by the
read, and the data can be flushed back to the file if the physical page
is needed elsewhere), but writes to MAP_PRIVATE pages do (the copy in
the page cache and the local copy the program needs diverge, so two pages
are needed to store them, and flushing the page cache copy back to disk
won't free up the local copy of the changed contents).
```

…

Process Virtual Memory Mapping



Source (above)

Very useful resource: ***How The Kernel Manages Your Memory,*** ***by Gustav Duarte.***

• All processes share the kernel page tables starting at PAGE_OFFSET (value 0xc0000000), and have their own *process-specific* page tables for the range 0...PAGE_OFFSET-1. Thus, the process page tables are valid in kernel mode, which simplifies user-->kernel transitions.

• For each user process, the kernel *maintains an* **mm_struct** *describing the process's address space* - this is in fact called the processes' ***memory descriptor***. It is referred to by the *mm* field of the process descriptor.

Shown below are some relevant members of the process **memory descriptor** - the ***mm_struct*** structure (*include/linux/sched.h*):

```
    struct vm_area_struct * mmap;    << pointer to head of list
                                 of memory regions (VMAs) >>
...
       unsigned long (*get_unmapped_area) (struct file *filp, ...)
...
       unsigned long mmap_base;      /* base of mmap area */
                    << start point for mmap's in the VA space >>
       unsigned long task_size;      /* size of task vm space */
...                                << = TASK_SIZE = 3GB normally >>
```

```
        pgd_t * pgd;              << Page Directory - (re)stored
                 into the CR3 register at context switch time >>

        atomic_t mm_users;    /* How many users with user space? */
        << mm_users: in effect, this is the number of threads in
           this process ! >>
      atomic_t mm_count;     /* How many references to "struct
                                 mm_struct" (users count as 1) */
```

<< *Related Q&A:* Linux Kernel: Why are we using two variables mm_users and mm_count in mm_struct ? >>

```
   atomic_long_t nr_ptes;          /* Page table pages */
   int map_count;                  /* number of VMAs */

   struct rw_semaphore mmap_sem;
...
   spinlock_t page_table_lock; /* Protects page tables and some counters */
...
        unsigned long start_code, end_code, start_data, end_data;
                                  << Code and Data segments >>
        unsigned long start_brk,  << Initial process heap address>>
                      brk,            <<Current final address of heap>>
         start_stack;  << Initial User Mode stack start address >>
        unsigned long arg_start, arg_end, << Cmd-line arguments >>
                 env_start, env_end;    << Environment variables >>
...
```

Note / Tips

1. **Process (approximate) physical memory usage**

   Try ps aux *and see for all the processes the total VM size (VSZ field) and RSS*.*
   Sort processes by (approx) physical memory usage with:
   `ps aux | head -n1 ; ps aux | sort -k6n`

*RSS – Resident Set Size:
 - RSS is resident set size, the non-swapped physical memory that a task has used (in kiloBytes).
 - The SIZE and RSS fields don't count some parts of a process including the page tables, kernel stack, struct thread_info, and struct task_struct. This is usually at least 20 KiB of memory that is always resident. SIZE is the virtual size of the process (code+data+stack).
- One can get detailed memory usage of a particular process (or indeed, thread) by :

```
    grep "^Vm" /proc/<pid>/status
Eg.
# grep "^Vm" /proc/1/status
VmPeak:  248200 kB
VmSize:  182664 kB
VmLck:        0 kB
VmPin:        0 kB
VmHWM:     5444 kB
VmRSS:     2544 kB
VmData:  148896 kB
VmStk:      136 kB
VmExe:     1260 kB
VmLib:     3448 kB
VmPTE:       96 kB
```

```
VmSwap:    1096 kB
#
```

<< Misc tip: quickly see the process hierarchy with PIDs using `pstree -p.` Note: this may not work on a busybox-based (or similar) Linux >>

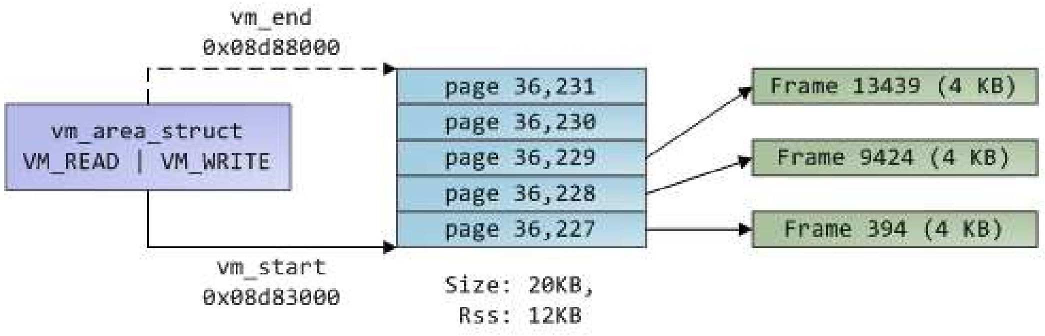
2. *mm->map_count is current # of VMAs (memory regions) that the process currently has. Maximum possible number of VMAs is tunable via:*
   ```
   $ cat /proc/sys/vm/max_map_count
   65530
   $
   ```

Virtual Memory Areas (VMAs)

Simply put, VMAs are the metadata structures the kernel uses to manage the "segments", or more accurately, the "regions" that make up the VAS (virtual address space) of a process. Each VMA manages a homegeneous regipon of VAS.

<<
How The Kernel Manages Your Memory *by Gustav Duarte.*

…
Let's put together virtual memory areas, page table entries and page frames to understand how this all works. Below is an example of a user heap:



Blue rectangles represent pages in the VMA range, while arrows represent page table entries mapping pages onto page frames. Some virtual pages lack arrows; this means their corresponding PTEs have the **Present** flag clear.

This could be because the pages have never been touched or because their contents have been swapped out. In either case access to these pages will lead to page faults, even though they are within the VMA. It may seem strange for the VMA and the page tables to disagree, yet this often happens.

…

>>

Ref: Optimizing VMA caching *<< advanced; patch set merged into ver 3.15 (see http://kernelnewbies.org/Linux_3.15) >>*

Vuln
StackRot (CVE-2023-3269): Linux kernel privilege escalation vulnerability
Ruihan Li, July 2023

… Previously, the VMAs were managed using red-black trees. However, starting from Linux kernel version 6.1, the migration **to maple trees** took place. Maple trees are RCU-safe B-tree data structures optimized for storing non-overlapping ranges. Nonetheless, their intricate nature adds complexity to the codebase and introduces the StackRot vulnerability. …

Linus:
Merge branch 'expand-stack' patch series (applied on 6.5, and subsequently back-ported to stable kernels (6.1.37, 6.3.11, and 6.4.1), effectively resolving the "Stack Rot" bug on July 1st.

Examining Memory Regions via the proc interface

Can we actually *see* for ourselves the memory regions of a Linux process? Yes, the **proc filesystem** (seen later) has a (pseudo) file that allows us to look up in detail the memory regions (VMAs) of any live process on the system.

The memory areas of a process can be seen by looking in ***/proc/<pid>/maps*** . As an example, here is the memory map of the init process (comments after a sharp sign): *<<Note: output below is from an IA-32 Linux box >>*

```
start   - end     perm    off   mj:mn inode        image
# sudo cat /proc/1/maps                      # lets look at init's memory regions
08048000-0804f000 r-xp 00000000 16:0b 193988     /sbin/init # text
0804f000-08050000 rw-p 00006000 16:0b 193988     /sbin/init # data
08050000-08071000 rw-p 08050000 00:00 0          [heap]   # heap
b7e21000-b7e22000 rw-p b7e21000 00:00 0   # zero-mapped bss for init*
b7e22000-b7f45000 r-xp 00000000 16:0b 294456     /lib/tls/libc-2.3.4.so
b7f45000-b7f46000 ---p 00123000 16:0b 294456     /lib/tls/libc-2.3.4.so
b7f46000-b7f47000 r--p 00123000 16:0b 294456     /lib/tls/libc-2.3.4.so
b7f47000-b7f4a000 rw-p 00124000 16:0b 294456     /lib/tls/libc-2.3.4.so
b7f4a000-b7f4c000 rw-p b7f4a000 00:00 0
b7f5c000-b7f5d000 rw-p b7f5c000 00:00 0
b7f5d000-b7f71000 r-xp 00000000 16:0b 290884     /lib/ld-2.3.4.so
b7f71000-b7f72000 r--p 00013000 16:0b 290884     /lib/ld-2.3.4.so
b7f72000-b7f73000 rw-p 00014000 16:0b 290884     /lib/ld-2.3.4.so
bff5b000-bff71000 rw-p bff5b000 00:00 0          [stack]
ffffe000-fffff000 ---p 00000000 00:00 0          [vdso]
#
start   - end     perm    off   mj:mn inode        image
```

Each field in */proc/*/maps* (except the image name) corresponds to a field in *struct vm_area_struct* .

<<
Similar facilities are available in most modern OS's:

"… On OpenSolaris you would use the pmap command— for example, *pmap –x <pid>*—whereas on Mac OS X you would execute the vmmap command - for instance, *vmmap <pid>* or *vmmap <procname>*, where *<procname>* is a string that will be matched against all the processes running on the system. If you are working on Windows, we suggest that you download the Sysinternals Suite by Mark Russinovich (http://technet.microsoft.com/en-us/sysinternals/bb842062.aspx), which provides a lot of very useful system and process analysis tools in addition to vmmap."
- *"A Guide to Kernel Exploitation", Perla, Oldani*
>>

Do use my procmap script (+ kernel module) to visualize the full virtual address space of any process!

<<
SIDEBAR :: /proc/iomem : physical address space

See this script for a CLI-based scaled view of the physical address space:

```
$ ./phymap_iomem.sh
                           /proc/iomem  ::  PHYSICAL ADDRESS SPACE        [embedded ver]
     Region                                   Phy Addr [hex]                    Size
                                        start       -       end    :       KB      MB      GB
```

```
 len=18320719871
+-----------------------------------+
| Reserved            [3 KB]
|                                   |
+-----------------------------------+
| System RAM          [347 KB]
|                                   |
+-----------------------------------+
| Reserved            [3 KB]
|                                   |
+-----------------------------------+
| System RAM          [203 KB]
|                                   |
+-----------------------------------+
| Reserved            [463 KB]
|                                   |
+-----------------------------------+
| PCI Bus 0000        [127 KB]
|                                   |
+-----------------------------------+
| Video ROM           [63 KB]
|                                   |
+-----------------------------------+
| System ROM          [63 KB]
|                                   |
+-----------------------------------+
```

--snip--

```
+-----------------------------------+
| Local APIC          [3 KB]
|                                   |
+-----------------------------------+
| System RAM          [13672447 KB]
|                                   |
|                                   |
|                                   |
|                                   |
|                                   |
~ . . . . . . . . . . . . . . . . .  ~
|                                   |
|                                   |
|                                   |
|                                   |
|                                   |
+-----------------------------------+
| Kernel code         [12300 KB]
|                                   |
|                                   |
|                                   |
+-----------------------------------+
| Kernel data         [7677 KB]
|                                   |
|                                   |
+-----------------------------------+
| Kernel bss          [1363 KB]
|                                  |
+----------------------------------+
| RAM buffer          [24575 KB]
|                                  |
|                                  |
|                                  |
|                                  |
|                                  |
|                                  |
|                                  |
+----------------------------------+
$
```

>>

```
$ man 5 proc

--snip--

  /proc/[pid]/maps
              A  file  containing the currently mapped memory regions and their access permissions.  See
mmap(2) for some further information about memory mappings.

...

The address field is the address space in the process  that  the  mapping  occupies.   The perms field is
a set of permissions:

                   r = read
                   w = write
                   x = execute
                   s = shared
                   p = private (copy on write)

              The  offset  field  is the offset into the file/whatever; dev is the device (major:minor);
inode is the inode on that device.  0 indicates that no inode is associated with the  memory region, as
would be the case with BSS (uninitialized data).

              The  pathname  field will usually be the file that is backing the mapping.  For ELF files,
you can easily coordinate with the offset field by looking at the Offset field in the  ELF program headers
(readelf -l).

              There are additional helpful pseudo-paths:

                   [stack] The initial process's (also known as the main thread's) stack.

                   [stack:<tid>] (since Linux 3.4)
                    A  thread's  stack  (where  the  <tid> is a thread ID).  It corresponds to the
/proc/[pid]/task/[tid]/ path.

                   [vdso] The virtual dynamically linked shared object.

                   [heap] The process's heap.

              If the pathname field is blank, this is an anonymous mapping as obtained via  the  mmap(2)
function.   There  is  no easy way to coordinate this back to a process's source, short of running it
through gdb(1), strace(1), or similar.

              Under Linux 2.0 there is no field giving pathname.
```

…

- * The name BSS is a historical relic, from an old assembly operator meaning "Block started by symbol." The BSS segment - basically the uninitialized data segment - of executable files isn't stored on disk, and the kernel maps the zero page (anonymous pages, typically filled by an malloc() request) to the BSS address range.

- Use *ldd(1)* to see what shared objects an executable links into

- The relevant code to look up and display information on the VMAs of a given PID is here: fs/proc/task_mmu.c : show_map_vma() .

- For security reasons, Linux supports a feature called **ASLR** (Address Space Layout Randomization). Tunable is:

/proc/sys/kernel/randomize_va_space

The following values are supported:

- 0 – No randomization. Everything is static.
- 1 – Conservative randomization. Shared libraries, stack, mmap(), VDSO and heap are randomized.
- 2 – Full randomization. In addition to elements listed in the previous point, memory managed through brk() is also randomized. [default]

Note on Optimizations

1. KSM candidates are aplenty: all the private anonymous regions (VMAs) of the VAS!

Eg. (seeing VMAs of Qemu via /proc/<pid>/maps):

```
…
7f981c000000-7f981e592000 rw-p 00000000 00:00 0
7f981e592000-7f9820000000 ---p 00000000 00:00 0
7f9828000000-7f9828021000 rw-p 00000000 00:00 0
7f9828021000-7f982c000000 ---p 00000000 00:00 0
7f982c000000-7f982c021000 rw-p 00000000 00:00 0
…
7f98fdc00000-7f98fdc20000 rw-p 00000000 00:00 0
7f98fde00000-7f98fde10000 rw-p 00000000 00:00 0
…
7f9915205000-7f9915268000 rw-p 00000000 00:00 0
7f99152a9000-7f99153db000 rw-p 00000000 00:00 0
…
```

2. The file-mapped pages in the page cache are already highly optimized; when a process uses a library (text/data), it is actually mmap'ed from the page cache into the VAS of the process. The page cache thus guarantees that exactly one copy of a file page is kept at any point in time (in effect, a de-dup kind of similar to that of KSM)!

Utilities to see aspects of process memory (based on procfs)

1. pmap(1)

pmap is part of the *procps* package – the one that provides ps(1). The pmap command reports the memory map of a process or processes.

```
# pmap -x 1
1:   /sbin/init splash
Address           Kbytes     RSS   Dirty Mode  Mapping
00007fbd54000000     164       0       0 rw---   [ anon ]
00007fbd54029000   65372       0       0 -----   [ anon ]
00007fbd5c000000     164       0       0 rw---   [ anon ]
00007fbd5c029000   65372       0       0 -----   [ anon ]
00007fbd61498000       4       0       0 -----   [ anon ]
00007fbd61499000    8192       0       0 rw---   [ anon ]
00007fbd61c99000       4       0       0 -----   [ anon ]
00007fbd61c9a000    8192       0       0 rw---   [ anon ]
00007fbd6249a000      16       0       0 r-x-- libuuid.so.1.3.0
00007fbd6249e000    2044       0       0 ----- libuuid.so.1.3.0
00007fbd6269d000       4       0       0 r---- libuuid.so.1.3.0
00007fbd6269e000       4       0       0 rw--- libuuid.so.1.3.0
00007fbd6269f000     236       0       0 r-x-- libblkid.so.1.1.0

--snip--

00007fbd6462a000       4       0       0 rw--- ld-2.21.so
00007fbd6462b000       4       0       0 rw---   [ anon ]
00007fbd6462c000    1260     640       0 r-x-- systemd
00007fbd6493b000      36       4       4 rw---   [ anon ]
00007fbd64964000      12       0       0 rw---   [ anon ]
00007fbd64967000     128      28       0 r---- systemd
00007fbd64987000       4       4       4 rw--- systemd
00007fbd64f0e000    1276     524     468 rw---   [ anon ]
00007ffc80623000     132      24      24 rw---   [ stack ]
00007ffc806a5000       8       0       0 r----   [ anon ]
00007ffc806a7000       8       8       0 r-x--   [ anon ]
ffffffffff600000       4       0       0 r-x--   [ anon ]
---------------- ------- ------- -------
total kB          182664    2544     516
#
```

In addition, *pmap* can be used to display *very verbose output* with the -X and -XX switches:

```
$ pmap -X $$

2982:   bash

         Address Perm   Offset Device   Inode   Size  Rss   Pss Referenced Anonymous LazyFree ShmemPmdMapped Shared_Hugetlb
Private_Hugetlb Swap SwapPss Locked Mapping

    55cbe67d7000 r-xp 00000000  08:06 1844860   1040  928  193        928         0        0              0              0
0    0       0    193 bash

    55cbe6ada000 r--p 00103000  08:06 1844860     16   16   16         16        16        0              0              0
0    0       0     16 bash
```

```
   55cbe6ade000 rw-p 00107000  08:06 1844860     36   36   36          36          36         0              0              0
0    0       0     36 bash


[…]


$ pmap -XX $$
2982:   bash
        Address Perm   Offset Device   Inode   Size KernelPageSize MMUPageSize  Rss   Pss Shared_Clean Shared_Dirty Private_Clean
Private_Dirty Referenced Anonymous LazyFree AnonHugePages ShmemPmdMapped Shared_Hugetlb Private_Hugetlb Swap SwapPss Locked
VmFlagsMapping
   55cbe67d7000 r-xp 00000000  08:06 1844860   1040              4           4  928  193          928            0             0
0       928         0        0             0             0             0              0    0       0    193    rd ex mr mw me dw sd
bash
   55cbe6ada000 r--p 00103000  08:06 1844860     16              4           4   16   16            0            0             0
16         16        16        0             0             0             0              0    0       0     16    rd mr mw me dw ac sd
bash


[…]
```

Ref: Cheat sheet: understanding the pmap(1) output

In kernels since 2.6.14, there is an additional verbose '***smaps***' entry under */proc/<pid>* ; this gives more detail on each segment (VMA).

To help interpret the various fields displayed, please take a look at the documentation: https://www.kernel.org/doc/Documentation/filesystems/proc.txt

Eg.:

```
# cat /proc/2012/smaps
00110000-00113000 r-xp 00000000 08:0a 9860       /usr/lib/libgthread-2.0.so.0.2600.1
Size:                 12 kB
Rss:                  12 kB
Pss:                   0 kB
Shared_Clean:         12 kB
Shared_Dirty:          0 kB
Private_Clean:         0 kB
Private_Dirty:         0 kB
Referenced:           12 kB
Swap:                  0 kB
KernelPageSize:        4 kB
MMUPageSize:           4 kB
00113000-00114000 r--p 00003000 08:0a 9860       /usr/lib/libgthread-2.0.so.0.2600.1
Size:                  4 kB
Rss:                   4 kB
```

```
Pss:                   4 kB
Shared_Clean:          0 kB
Shared_Dirty:          0 kB
Private_Clean:         0 kB
Private_Dirty:         4 kB
Referenced:            0 kB
Swap:                  0 kB
KernelPageSize:        4 kB
MMUPageSize:           4 kB

--snip--

bf8bd000-bf8e0000 rw-p 00000000 00:00 0          [stack]
Size:                144 kB
Rss:                 140 kB
Pss:                 140 kB
Shared_Clean:          0 kB
Shared_Dirty:          0 kB
Private_Clean:         0 kB
Private_Dirty:       140 kB
Referenced:           20 kB
Swap:                  0 kB
KernelPageSize:        4 kB
MMUPageSize:           4 kB
#
```

Source: https://www.kernel.org/doc/Documentation/filesystems/proc.txt

"...

The /proc/PID/smaps is an extension based on maps, showing the memory consumption for each of the process's mappings. For each of mappings there is a series of lines such as the following:

```
08048000-080bc000 r-xp 00000000 03:02 13130      /bin/bash
Size:               1084 kB
Rss:                 892 kB
Pss:                 374 kB
Shared_Clean:        892 kB
Shared_Dirty:          0 kB
Private_Clean:         0 kB
Private_Dirty:         0 kB
Referenced:          892 kB
Anonymous:             0 kB
Swap:                  0 kB
KernelPageSize:        4 kB
MMUPageSize:           4 kB
Locked:              374 kB
VmFlags: rd ex mr mw me de
```

the first of these lines shows the same information as is displayed for the mapping in /proc/PID/maps. The remaining lines show the :

<< formatted for readability >>

- size of the mapping (size),
- the amount of the mapping that is currently resident in RAM (RSS),
- the process' proportional share of this mapping (PSS), the number of clean and dirty private pages in the mapping. Note that even a page which is part of a MAP_SHARED mapping, but has only a single pte mapped, i.e. is currently used by only one process, is accounted as private and not as shared.

- "Referenced" indicates the amount of memory currently marked as referenced or accessed.
- "Anonymous" shows the amount of memory that does not belong to any file. Even a mapping associated with a file may contain anonymous pages: when MAP_PRIVATE and a page is modified, the file page is replaced by a private anonymous copy.
- "Swap" shows how much would-be-anonymous memory is also used, but out on swap.

"VmFlags" field deserves a separate description. This member represents the kernel flags associated with the particular virtual memory area in two letter encoded manner. The codes are the following:

```
    rd  - readable
    wr  - writeable
    ex  - executable
    sh  - shared
    mr  - may read
    mw  - may write
    me  - may execute
    ms  - may share
    gd  - stack segment growns down
    pf  - pure PFN range
    dw  - disabled write to the mapped file
    lo  - pages are locked in memory
    io  - memory mapped I/O area
    sr  - sequential read advise provided
    rr  - random read advise provided
    dc  - do not copy area on fork
    de  - do not expand area on remapping
    ac  - area is accountable
    nr  - swap space is not reserved for the area
    ht  - area uses huge tlb pages
    nl  - non-linear mapping
    ar  - architecture specific flag
    dd  - do not include area into core dump
    sd  - soft-dirty flag
    mm  - mixed map area
    hg  - huge page advise flag
    nh  - no-huge page advise flag
    mg  - mergable advise flag
```

Note that there is no guarantee that every flag and associated mnemonic will be present in all further kernel releases. Things get changed, the flags may be vanished or the reverse -- new added.

This file is only present if the CONFIG_MMU kernel configuration option is enabled.

The /proc/PID/clear_refs is used to reset the PG_Referenced and ACCESSED/YOUNG bits on both physical and virtual pages associated with a process, and the

soft-dirty bit on pte (see Documentation/vm/soft-dirty.txt for details).
To clear the bits for all the pages associated with the process
    > echo 1 > /proc/PID/clear_refs

To clear the bits for the anonymous pages associated with the process
    > echo 2 > /proc/PID/clear_refs

To clear the bits for the file mapped pages associated with the process
    > echo 3 > /proc/PID/clear_refs

To clear the soft-dirty bit
    > echo 4 > /proc/PID/clear_refs

Any other value written to /proc/PID/clear_refs will have no effect.

The /proc/pid/pagemap gives the PFN, which can be used to find the pageflags
using /proc/kpageflags and number of times a page is mapped using
/proc/kpagecount. For detailed explanation, see Documentation/vm/pagemap.txt.
..."

2. smem

From the smem(1) man page:
…
smem reports physical memory usage, taking shared memory pages into account. Unshared memory is reported as the USS (Unique Set Size). Shared memory is divided evenly among the processes sharing that memory. The unshared memory (USS) plus a process's proportion of shared memory is reported as the PSS (Proportional Set Size). The USS and PSS only include physical memory usage. They do not include memory that has been swapped out to disk.

<< RSS = Resident Set Size ; an (older) measure of physical memory usage by the process/mapping >>

<<
Source

One interesting metric is the **PSS value**, also known as "Proportional Share Size". This is the amount of memory which is private to the mapping, plus the partial amount of shared mappings of this process. For example, if the process has mapped 100 KiB of private memory, another 200 KiB of shared memory which is shared between two processes and another 150 KiB which is shared between three processes, the PSS is calculated like `100 KiB + 200 KiB / 2 + 150 KiB / 3 = 250 KiB`. [...]
>>

Memory can be reported by process, by user, by mapping, or systemwide. Both text mode and graphical output are available.
…

REQUIREMENTS

smem requires:

- Linux kernel providing 'Pss' metric in /proc/<pid>/smaps (generally 2.6.27 or newer).
- Python 2.x (at least 2.4 or so).
- The matplotlib library (only if you want to generate graphical charts).

A few examples follow below [run on the Seawolf Minimal for Developer's VM]

```
tty0 $ smem
  PID User     Command                         Swap      USS      PSS      RSS
  780 seawolf  /lib/systemd/systemd --user        0      956     2001     6400
  783 seawolf  -bash                              0     2032     2770     5392
 1167 seawolf  -bash                              0     2024     2773     5372
 1348 seawolf  /usr/bin/python /usr/bin/sm        0     6732     6969     8948

tty0 $ smem -k      << -k: Show unit suffixes (human-readable) >>
  PID User     Command                         Swap      USS      PSS      RSS
  780 seawolf  /lib/systemd/systemd --user        0   956.0K     2.0M     6.2M
  783 seawolf  -bash                              0     2.0M     2.7M     5.3M
 1167 seawolf  -bash                              0     2.0M     2.7M     5.2M
 1361 seawolf  /usr/bin/python /usr/bin/sm        0     6.5M     6.8M     8.7M

tty0 $ smem -k -w  << -w|--system : Report systemwide memory usage summary. >>
Area                           Used      Cache   Noncache
firmware/hardware                 0          0          0
kernel image                      0          0          0
kernel dynamic memory        380.8M     356.5M      24.3M
userspace memory              41.7M      20.5M      21.2M
free memory                  569.4M     569.4M          0

tty0 $ smem -k -w --realmem=1G  << --realmem: Amount of physical RAM.  This lets smem detect
the amount of memory used  by  firmware/hardware  in  the systemwide -w) output. If provided,
it will also be used as the total memory size to base percentages on. >>
Area                           Used      Cache   Noncache
firmware/hardware             32.1M          0      32.1M
kernel image                      0          0          0
kernel dynamic memory        380.8M     356.6M      24.3M
userspace memory              41.7M      20.5M      21.2M
free memory                  569.4M     569.4M          0
tty0 $
```

```
 REPORT BY
       If none of the following options are included, smem reports memory usage by process.

       -m, --mappings
              Report memory usage by mapping.
       -u, --users
              Report memory usage by user.
       -w, --system
              Report systemwide memory usage summary.
```

```
tty0 $ smem --users
```

```
User     Count     Swap      USS      PSS      RSS
seawolf      4        0    11984    14773    26344
tty0 $
tty0 $ smem --users -p          << -p : Show percentages. >>
User     Count     Swap      USS      PSS      RSS
seawolf      4    0.00%    1.18%    1.45%    2.60%
tty0 $
tty0 $ smem --mappings
Map                                        PIDs   AVGPSS      PSS
[vdso]                                        4        0        0
[vsyscall]                                    4        0        0
[vvar]                                        4        0        0
/lib/x86_64-linux-gnu/libcap.so.2.25          1        9        9
/lib/x86_64-linux-gnu/libattr.so.1.1.0        1       10       10
/lib/x86_64-linux-gnu/libcap-ng.so.0.0.0      1       10       10
/lib/x86_64-linux-gnu/librt-2.24.so           1       10       10
/lib/x86_64-linux-gnu/libutil-2.24.so         1       10       10
/lib/x86_64-linux-gnu/libacl.so.1.1.0         1       11       11
/lib/x86_64-linux-gnu/libuuid.so.1.3.0        1       11       11

--snip--

/lib/x86_64-linux-gnu/libc-2.24.so            4      103      415
/lib/systemd/systemd                          1      520      520
/bin/bash                                     2      488      976
/usr/bin/python2.7                            1     2308     2308
<anonymous>                                   4      786     3144
[heap]                                        4     1387     5548
tty0 $

tty0 $ smem --mappings -p
Map                                        PIDs   AVGPSS      PSS
[vdso]                                        4    0.00%    0.00%
[vsyscall]                                    4    0.00%    0.00%
[vvar]                                        4    0.00%    0.00%
/lib/x86_64-linux-gnu/libcap.so.2.25          1    0.00%    0.00%
/lib/x86_64-linux-gnu/libattr.so.1.1.0        1    0.00%    0.00%
/lib/x86_64-linux-gnu/libcap-ng.so.0.0.0      1    0.00%    0.00%
/lib/x86_64-linux-gnu/librt-2.24.so           1    0.00%    0.00%
/lib/x86_64-linux-gnu/libutil-2.24.so         1    0.00%    0.00%
/lib/x86_64-linux-gnu/libacl.so.1.1.0         1    0.00%    0.00%
/lib/x86_64-linux-gnu/libuuid.so.1.3.0        1    0.00%    0.00%

--snip--

/lib/x86_64-linux-gnu/libc-2.24.so            4    0.01%    0.04%
/lib/systemd/systemd                          1    0.05%    0.05%
/bin/bash                                     2    0.05%    0.10%
/usr/bin/python2.7                            1    0.23%    0.23%
<anonymous>                                   4    0.08%    0.31%
[heap]                                        4    0.14%    0.55%
tty0 $
```

3. smemcap

```
$ man smem
…
EMBEDDED USAGE

       To capture memory statistics on resource-constrained systems, the the smemcap
package includes a utility  named smemcap.   smemcap captures all /proc entries
required by smem and outputs them as an uncompressed .tar file to STDOUT.  smem can
analyze the output using the --source option.  smemcap is small and does not require
Python.

       To use smemcap:

       1.     Install package smemcap on target system.

       2.     Run smemcap on the target system and save the output:
              smemcap > memorycapture.tar

       3.     Copy the output to another machine and run smem on it:
              smem -S memorycapture.tar
```

The End?
You wish. :-)

Ponder and think, with these articles as food for thought:

- Transcendent Memory
  - *Resource: Slide Presentation on Transcendent Memory, Oracle Linux.*
  - *LWN article Transcendent memory in a nutshell, by Dan Magenheimer, Aug 2011.*
- Linux MM: rmap (reverse mapping); article "The case of the overly anonymous anon_vma" [LWN]
- Virtualization technology brings up interesting usage of Linux MM; take a look at this article on how KVM (qemu-kvm) internally uses memory (it's old-ish; as of qemu v0.12)
- Source: "There are some cases where a desktop system could be really unresponsive while doing things such as writing to a very slow USB storage device and some memory pressure. This release includes a small patch that improves the VM heuristics to solve this problem.
  Code: (commit)"
- Memory Management Notifiers, LWN, Jon Corbet, June 2008
- … and lots more! A Google search on "Linux memory management articles" does indeed throw up a lot! :-)

Also, some FAQs on Memory Management.

<<

The APPENDICES are in a separate document:
"Appendices | Linux VM"*.*

>>

<< End of Linux Memory Management, Part 4 >>