Next Previous Contents

9. Memory

9.1 Physical and virtual memory

Traditionally, one has *physical memory*, that is, memory that is actually present in the machine, and *virtual memory*, that is, address space. Usually the virtual memory is much larger than the physical memory, and some hardware or software mechanism makes sure that a program can transparently use this much larger virtual space while in fact only the physical memory is available.

Nowadays things are reversed: on a Pentium II one can have 64 GB physical memory, while addresses have 32 bits, so that the virtual memory has a size of 4 GB. We'll have to wait for a 64-bit architecture to get large amounts of virtual memory again. The present situation on a Pentium with more than 4 GB is that using the PAE (Physical Address Extension) it is possible to place the addressable 4 GB anywhere in the available memory, but it is impossible to have access to more than 4 GB at once.

9.2 Kinds of memory

Kernel and user space work with virtual addresses (also called linear addresses) that are mapped to physical addresses by the memory management hardware. This mapping is defined by page tables, set up by the operating system.

DMA devices use bus addresses. On an i386 PC, bus addresses are the same as physical addresses, but other architectures may have special address mapping hardware to convert bus addresses to physical addresses.

Under Linux one has

```
#include <asm/io.h>

phys_addr = virt_to_phys(virt_addr);
virt_addr = phys_to_virt(phys_addr);
 bus_addr = virt_to_bus(virt_addr);
virt_addr = bus_to_virt(bus_addr);
```

All this is about accessing ordinary memory. There is also "shared memory" on the PCI or ISA bus. It can be mapped inside a 32-bit address space using `ioremap()`, and then used via the `readb()`, `writeb()` (etc.) functions.

Life is complicated by the fact that there are various caches around, so that different ways to access the same physical address need not give the same result.

See `asm/io.h` and `Documentation/{IO-mapping.txt,DMA-mapping.txt,DMA-API.txt}`.

9.3 Kernel memory handling

Pages

The basic unit of memory is the *page*. Nobody knows how large a page is (that is why the Linux swapspace structure, with a signature at the end of a page, is so unfortunate), this is architecture-dependent, but typically PAGE_SIZE = 4096. (PAGE_SIZE equals 1 << PAGE_SHIFT, and PAGE_SHIFT is 12, 13, 14, 15, 16 on the various architectures). If one is lucky, the `getpagesize()` system call returns the page size.

Usually, the page size is determined by the hardware: the relation between virtual addresses and physical addresses is given by page tables, and when a virtual address is referenced that does not (yet) correspond to a physical address, a *page fault* occurs, and the operating system can take appropriate action. Most hardware allows a very limited choice of page sizes. (For example, a Pentium II knows about 4KiB and 4MiB pages.)

Kernel memory allocation

Buddy system

The kernel uses a buddy system with power-of-two sizes. For order 0, 1, 2, ..., 9 it has lists of areas containing 2^order pages. If a small area is needed and only a larger area is available, the larger area is split into two halves (buddies), possibly repeatedly. In this way the waste is at most 50%. Since 2.5.40, the number of free areas of each order can be seen in `/proc/buddyinfo`. When an area is freed, it is checked whether its buddy is free as well, and if so they are merged. Read the code in `mm/page_alloc.c`.

get_free_page

The routine `__get_free_page()` will give us a page. The routine `__get_free_pages()` will give a number of consecutive pages. (A power of two, from 1 to 512 or so. The above buddy system is used.)

kmalloc

The routine `kmalloc()` is good for an area of unknown, arbitrary, smallish length, in the range 32-131072 (more precisely: 1/128 of a page up to 32 pages), preferably below 4096. For the sizes, see `<linux/kmalloc_sizes.h>`. Because of fragmentation, it will be difficult

to get large consecutive areas from `kmalloc()`. These days `kmalloc()` returns memory from one of a series of slab caches (see below) with names like "size-32", ..., "size-131072".

Priority

Each of the above routines has a `flags` parameter (a bit mask) indicating what behaviour is allowed. Deadlock is possible when, in order to free memory, some pages must be swapped out, or some I/O must be completed, and the driver needs memory for that. This parameter also indicated where we want the memory (below 16M for ISA DMA, or ordinary memory, or high memory). Finally, there is a bit specifying whether we would like a hot or a cold page (that is, a page likely to be in the CPU cache, or a page not likely to be there). If the page will be used by the CPU, a hot page will be faster. If the page will be used for device DMA the CPU cache would be invalidated anyway, and a cold page does not waste precious cache contents.

Some common flag values (see `linux/gfp.h`):

```
/* Zone modifiers in GFP_ZONEMASK (see linux/mmzone.h - low four bits) */
#define __GFP_DMA       0x01

/* Action modifiers - doesn't change the zoning */
#define __GFP_WAIT      0x10    /* Can wait and reschedule? */
#define __GFP_HIGH      0x20    /* Should access emergency pools? */
#define __GFP_IO        0x40    /* Can start low memory physical IO? */
#define __GFP_FS        0x100   /* Can call down to low-level FS? */
#define __GFP_COLD      0x200   /* Cache-cold page required */

#define GFP_NOIO        (__GFP_WAIT)
#define GFP_NOFS        (__GFP_WAIT | __GFP_IO )
#define GFP_ATOMIC      (__GFP_HIGH)
#define GFP_USER        (__GFP_WAIT | __GFP_IO | __GFP_FS)
#define GFP_KERNEL      (__GFP_WAIT | __GFP_IO | __GFP_FS)
```

Uses:

- `GFP_KERNEL` is the default flag. Sleeping is allowed.
- `GFP_ATOMIC` is used in interrupt handlers. Never sleeps.
- `GFP_USER` for user mode allocations. Low priority, and may sleep. (Today equal to `GFP_KERNEL`.)
- `GFP_NOIO` must not call down to drivers (since it is used from drivers).
- `GFP_NOFS` must not call down to filesystems (since it is used from filesystems -- see, e.g., `dcache.c:shrink_dcache_memory` and `inode.c:shrink_icache_memory`).

vmalloc

The routine `vmalloc()` has a similar purpose, but has a better chance of being able to return larger consecutive areas, and is more expensive. It uses page table manipulation to create an

area of memory that is consecutive in virtual memory, but not necessarily in physical memory. Device I/O to such an area is a bad idea. It uses the above calls with GFP_KERNEL to get its memory, so cannot be used in interrupt context.

bigphysarea

There is a patch around, the " bigphysarea patch", allowing one to reserve a large area at boot time. Some devices need a lot of room (for video frame grabbing, scanned images, wavetable synthesis, etc.). See, e.g., `video/zr36067.c`.

The slab cache

The routine `kmalloc` is general purpose, and may waste up to 50% space. If certain size areas are needed very frequently, it makes sense to have separate pools for them. For example, Linux has separate pools for inodes, dentries, buffer heads. The pool is created using `kmem_cache_create()`, and allocation is by `kmem_cache_alloc()`.

The number of special purpose caches is increasing quickly. I have here a 2.4.18 system with 63 slab caches, and a 2.5.56 one with 149 slab caches. Probably this number should be reduced a bit again.

Statistics is found in `/proc/slabinfo` (and here also statistics on the `kmalloc` caches is given). From the man page:

```
% man 5 slabinfo
...
Frequently used objects in the Linux kernel (buffer heads,
inodes, dentries, etc.)  have their own  cache.  The  file
/proc/slabinfo gives statistics. For example:

       % cat /proc/slabinfo
       slabinfo - version: 1.1
       kmem_cache            60      78    100    2    2    1
       blkdev_requests     5120    5120     96  128  128    1
       mnt_cache             20      40     96    1    1    1
       inode_cache         7005   14792    480 1598 1849    1
       dentry_cache        5469    5880    128  183  196    1
       filp                 726     760     96   19   19    1
       buffer_head        67131   71240     96 1776 1781    1
       vm_area_struct      1204    1652     64   23   28    1
       ...
       size-8192              1      17   8192    1   17    2
       size-4096             41      73   4096   41   73    1
       ...

For  each  slab  cache, the cache name, the number of cur
rently active  objects,  the  total  number  of  available
objects,  the  size of each object in bytes, the number of
pages with at least one active object, the total number of
```

```
allocated pages, and the number of pages per slab are
given.
```

The utility `slabtop` displays slab usage.

Question *One of the reasons given for a slab cache is that it will give areas of the precise required size, and thus reduce memory waste. In the man page example above, look at how many pages are used for each slab cache, and how much space it would have cost to kmalloc() everything. Does the slab cache do better?*

Having a slab cache has some other advantages besides reducing waste. Allocation and freeing is faster than with the buddy system. There is an attempt at doing some *cache colouring*.

9.4 i386 addressing

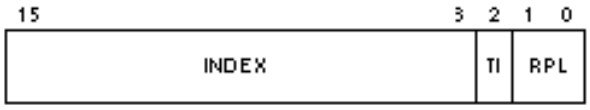
Let us look at an explicit machine to see how addressing is done. This is only a very brief description. For all details, see some Intel manual. (I have here 24547209.pdf, entitled *IA-32 Intel Architecture Software Developer's Manual Volume 3*, with Chapter 3: *Protected Mode Memory Management*. It can be downloaded from developer.intel.com.)

Addressing goes in two stages: a *logical address* determined by segment and offset is translated into a *linear address*, and next the linear address is translated into a *physical address*.

Real mode assembler programmers may be familiar with the first stage: one has a 16-bit segment register with value S and 16- or 32-bit offset O, that produce a 20- or 32-bit linear address A = 16*S + O, and this is also the physical address. There are six segment registers: CS, SS, DS, ES, FS, GS. The first is used when instructions are fetched, the second for stack access, the third for data access, and override prefixes allow one to specify any of the six.

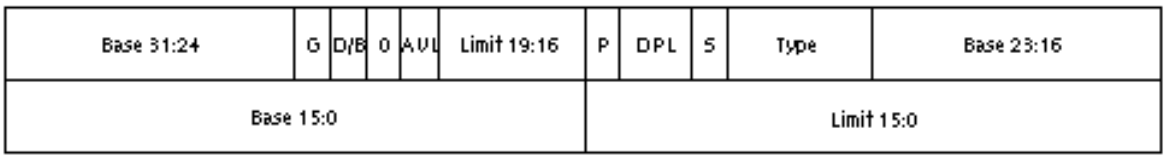
Segmentation

In protected mode one still has segmentation (although most systems do not use it - the corresponding hardware is just wasted and present only for compatibility reasons), but one also has paging. The paging is optional (it can be enabled/disabled), but this unused segmentation is always present, and much more complicated than in real mode.



Segment Selector

Instead of a single 16-bit value, the segmentation register S is now a 13-bit index I followed by the single bit value TI (Table Index) and the 2-bit value RPL (Requested Protection Level). The bit TI selects one of two tables: the GDT (Global Descriptor Table) or the LDT (Local Descriptor Table). Each can have up to 8192 entries, and the 13-bit index I selects an entry in one of these tables.



Segment Descriptor

Such an entry is an 8-byte struct that has a 32-byte field `base`, and a 20-byte field `limit` and several small fields. Here `base` gives the start of the segment, and `limit` its length minus one. One of the small fields is a 1-bit value G (Granularity). If this is zero then `limit` is measured in bytes, if it is one then `limit` is measured in 4 KiB units. Now that we have the base address B of the segment, add the 32-bit offset O to obtain the linear address.

Paging

When paging is enabled (as it is under Linux) we are only halfway, and this useless segmentation stuff is followed by the extremely useful paging. The linear address A is divided into three parts: a 10-bit *directory* AD, a 10-bit *table* AT and a 12-bit *offset* AO. The corresponding physical address is now found in four stages.

(i) The CR3 processor register points at the *page directory*.

(ii) Index the page directory with AD to find a *page table*.

(iii) Index the page table with AT to find the *page*.

(iv) Index the page with AO to find the physical address.

Thus, the page directory is a table with up to 1024 entries. Each of these entries points to a page table that itself has up to 1024 entries. Each of these point to a page. The entries in page directory and page table have 32 bits, but only the first 20 are needed for the address since by convention pages and page tables start at a 4 KiB boundary so that the last 12 address bits are zero. In page directory and page table these last bits are used for bookkeeping purposes. By far the most important bit is the "present" bit. If the user program refers to a page that is not actually present (according to this bit in the page table) a *page fault* is generated, and the operating system might fetch the page, map it somewhere in memory, and continue the user program.

Apart from the "present" bit there is the "dirty" bit, that indicates whether the page was modified, the "accessed" bit, that indicates that the page was read, and permission bits.

The above was for 4 KiB pages. There are also 4 MiB pages, where one indexing stage is absent and the offset part has 22 bits. (With PAE one also has 2 MiB pages.)

For many pages of detail, see the Intel manuals.

9.5 Reference

Recently Mel Gorman released his thesis, a detailed description and commentary of the virtual memory management of Linux 2.4.

9.6 Overcommit and OOM

Normally, a user-space program reserves (virtual) memory by calling `malloc()`. If the return value is NULL, the program knows that no more memory is available, and can do something appropriate. Most programs will print an error message and exit, some first need to clean up lockfiles or so, and some smarter programs can do garbage collection, or adapt the computation to the amount of available memory. This is life under Unix, and all is well.

Linux on the other hand is seriously broken. It will by default answer "yes" to most requests for memory, in the hope that programs ask for more than they actually need. If the hope is fulfilled Linux can run more programs in the same memory, or can run a program that requires more virtual memory than is available. And if not then very bad things happen.

What happens is that the *OOM killer* (OOM = out-of-memory) is invoked, and it will select some process and kill it. One holds long discussions about the choice of the victim. Maybe not a root process, maybe not a process doing raw I/O, maybe not a process that has already spent weeks doing some computation. And thus it can happen that one's emacs is killed when someone else starts more stuff than the kernel can handle. Ach. Very, very primitive.

Of course, the very existence of an OOM killer is a bug.

A typical case: I do `umount -a` in a situation where 30000 filesystems are mounted. Now `umount` runs out of memory and the kernel log reports

```
Sep 19 00:33:10 mette kernel: Out of Memory: Killed process 8631 (xterm).
Sep 19 00:33:34 mette kernel: Out of Memory: Killed process 9154 (xterm).
Sep 19 00:34:05 mette kernel: Out of Memory: Killed process 6840 (xterm).
Sep 19 00:34:42 mette kernel: Out of Memory: Killed process 9066 (xterm).
Sep 19 00:35:15 mette kernel: Out of Memory: Killed process 9269 (xterm).
Sep 19 00:35:43 mette kernel: Out of Memory: Killed process 9351 (xterm).
Sep 19 00:36:05 mette kernel: Out of Memory: Killed process 6752 (xterm).
```

Randomly `xterm` windows are killed, until the `xterm` window that was `x`'s console is killed. Then `x` exits and all user processes die, including the `umount` process that caused all this.

OK. This is very bad. People lose long-running processes, lose weeks of computation, just because the kernel is an optimist.

Demo program 1: allocate memory without using it.

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

int main (void) {
        int n = 0;

        while (1) {
                if (malloc(1<<20) == NULL) {
                        printf("malloc failure after %d MiB\n", n);
                        return 0;
                }
                printf ("got %d MiB\n", ++n);
        }
}
```

Demo program 2: allocate memory and actually touch it all.

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

int main (void) {
        int n = 0;
        char *p;

        while (1) {
                if ((p = malloc(1<<20)) == NULL) {
                        printf("malloc failure after %d MiB\n", n);
                        return 0;
                }
                memset (p, 0, (1<<20));
                printf ("got %d MiB\n", ++n);
        }
}
```

Demo program 3: first allocate, and use later.

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

#define N       10000

int main (void) {
```

```
        int i, n = 0;
        char *pp[N];

        for (n = 0; n < N; n++) {
                pp[n] = malloc(1<<20);
                if (pp[n] == NULL)
                        break;
        }
        printf("malloc failure after %d MiB\n", n);

        for (i = 0; i < n; i++) {
                memset (pp[i], 0, (1<<20));
                printf("%d\n", i+1);
        }

        return 0;
}
```

Typically, the first demo program will get a very large amount of memory before `malloc()` returns `NULL`. The second demo program will get a much smaller amount of memory, now that earlier obtained memory is actually used. The third program will get the same large amount as the first program, and then is killed when it wants to use its memory. (On a well-functioning system, like Solaris, the three demo programs obtain the same amount of memory and do not crash but see `malloc()` return `NULL`.)

For example:

- On an 8 MiB machine without swap running 1.2.11:
  demo1: 274 MiB, demo2: 4 MiB, demo3: 270 / oom after 1 MiB: `Killed.`
- Idem, with 32 MiB swap:
  demo1: 1528 MiB, demo2: 36 MiB, demo3: 1528 / oom after 23 MiB: `Killed.`
- On a 32 MiB machine without swap running 2.0.34:
  demo1: 1919 MiB, demo2: 11 MiB, demo3: 1919 / oom after 4 MiB: `Bus error.`
- Idem, with 62 MiB swap:
  demo1: 1919 MiB, demo2: 81 MiB, demo3: 1919 / oom after 74 MiB: The machine hangs. After several seconds: `Out of memory for bash. Out of memory for crond. Bus error.`
- On a 256 MiB machine without swap running 2.6.8.1:
  demo1: 2933 MiB, demo2: after 98 MiB: `Killed.` Also: `Out of Memory: Killed process 17384 (java_vm).` demo3: 2933 / oom after 135 MiB: `Killed.`
- Idem, with 539 MiB swap:
  demo1: 2933 MiB, demo2: after 635 MiB: `Killed.` demo3: oom after 624 MiB: `Killed.`

Other kernels have other strategies. Sometimes processes get a segfault when accessing memory that the kernel is unable to provide, sometimes they are killed, sometimes other processes are killed, sometimes the kernel hangs.

Turning off overcommit

Going in the wrong direction

Since 2.1.27 there are a sysctl `VM_OVERCOMMIT_MEMORY` and proc file `/proc/sys/vm/overcommit_memory` with values 1: do overcommit, and 0 (default): don't. Unfortunately, this does not allow you to tell the kernel to be more careful, it only allows you to tell the kernel to be less careful. With `overcommit_memory` set to 1 every `malloc()` will succeed. When set to 0 the old heuristics are used, the kernel still overcommits.

Going in the right direction

Since 2.5.30 the values are: 0 (default): as before: guess about how much overcommitment is reasonable, 1: never refuse any `malloc()`, 2: be precise about the overcommit - never commit a virtual address space larger than swap space plus a fraction `overcommit_ratio` of the physical memory. Here `/proc/sys/vm/overcommit_ratio` (by default 50) is another user-settable parameter. It is possible to set `overcommit_ratio` to values larger than 100. (See also `Documentation/vm/overcommit-accounting`.)

After

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

all three demo programs were able to obtain 498 MiB on this 2.6.8.1 machine (256 MiB, 539 MiB swap, lots of other active processes), very satisfactory.

However, without swap, no more processes could be started - already more than half of the memory was committed. After

```
# echo 80 > /proc/sys/vm/overcommit_ratio
```

all three demo programs were able to obtain 34 MiB. (Exercise: solve two equations with two unknowns and conclude that main memory was 250 MiB, and the other processes took 166 MiB.)

One can view the currently committed amount of memory in `/proc/meminfo`, in the field `Committed_AS`.

9.7 Stack overflow

Processes use memory on the stack and on the heap. Heap memory is provided by `malloc()` or the underlying mechanisms. The stack grows until it no longer can, and the process is hit by a SIGSEGV signal, a segmentation violation (because of the access of a location just beyond the end of the stack), and is killed. Sometimes people quote this as a phenomenon roughly similar to OOM. But this is very different. The OOM killer will kill some random

process, say `rpm` or `syslog`, because the system is short on memory, and the programmer is unable to do anything about it. A stack overflow happens because of something the program does itself, and if necessary the programmer can do something.

Each process has a soft and a hard limit on the stack size. The soft limit can be changed, but must stay below the hard limit. The hard limit can be decreased, but can be increased only by root.

```
% ulimit -s
unlimited
% ulimit -s 10
% ls
Segmentation fault
% ulimit -s 100
-bash: ulimit: stack size: cannot modify limit: Operation not permitted
% su
Segmentation fault :-)
```

One can get or set such limits via getrlimit/setrlimit:

```
/* getstklim.c */
#include <stdio.h>
#include <sys/resource.h>

int main() {
        struct rlimit rlim;

        if (getrlimit(RLIMIT_STACK, &rlim))
                return 1;
        printf("stack size: current %ld max: %ld\n",
                rlim.rlim_cur, rlim.rlim_max);
        return 0;
}
```

The bash builtin `ulimit` works in KiB, the system calls getrlimit/setrlimit in bytes.

```
% ulimit -s
8192
% ./getstklim
stack size: current 8388608 max: -1
% ulimit -s 100
% ./getstklim
stack size: current 102400 max: 102400
% su
# ulimit -s unlimited
# ./getstklim
stack size: current -1 max: -1
```

The old limit of 8 MiB was introduced in kernel version 1.3.7. Today it is ridiculously low. If you get random segfaults, put "ulimit -s unlimited" in your `.profile`.

It is not common to worry about stack overflow, and sometimes one can compromise the security of a system by starting a setuid binary with low resource limits:

```
% ulimit -s 15; mount
Segmentation fault
% ulimit -s 10; mount
Killed
```

But a programmer can catch signals and do something about the SIGSEGV. (This was even the technique the Bourne shell used for memory management: don't use malloc but just store stuff in the data area past the end of what was used before. If that gives a SIGSEGV, then extend the data area in the signal handler and return from the signal handler. Unfortunately, as it turned out, this would not correctly restart the faulting instruction on all architectures.)

A simple demo that catches SIGSEGV:

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

void segfault(int dummy) {
        printf("Help!\n");
        exit(1);
}

int main() {
        int *p = 0;

        signal(SIGSEGV, segfault);
        *p = 17;
        return 0;
}
```

Without the `exit()` here, this demo will loop because the illegal assignment is restarted.

This simple demo fails to catch stack overflow, because there is no stack space for a call frame for the segfault() interrupt handler. If it is desired to catch stack overflow one first must set up an alternative stack. As follows:

```
...
int main() {
        char myaltstack[SIGSTKSZ];
        struct sigaction act;
        stack_t ss;

        ss.ss_sp = myaltstack;
        ss.ss_size = sizeof(myaltstack);
        ss.ss_flags = 0;
        if (sigaltstack(&ss, NULL))
                errexit("sigaltstack failed");
```

```
        act.sa_handler = segfault;
        act.sa_flags = SA_ONSTACK;
        if (sigaction(SIGSEGV, &act, NULL))
                errexit("sigaction failed");
...
```

This will at least allow one to print an error message or to do some cleanup upon stack overflow.

Next Previous Contents