Memory Management 2

Tim Robinson · timothy.robinson@ic.ac.uk · http://www.themoebius.org.uk/

Introduction

You might have already read my "Memory Management 1" tutorial; if not, then you should go and read it now, because this tutorial relies on it. Here I'll be describing the next level of memory management: turning the memory manager I wrote about in the first tutorial into something useful and general-purpose.

Remember that the low-level memory manager deals with the computer's physical memory: that is, the raw address space of your PC before things like paging come into effect. The low-level manager has three main components: an allocator (for allocating one physical page at a time), a deallocator (for deallocating those pages) and a memory mapper (for modifying mappings between the virtual address space and physical memory). Now it's time to make something useful out of these.

There are a few problems with the physical memory functions which make them unsuitable for everyday memory allocation like, say, `malloc()`:

- They only allocate whole pages at a time. This is fine if you only ever want to allocate memory in multiples of 4KB (on the x86), but not much use if you're using things like linked lists (which you will be).
- They deal with physical addresses, not virtual. This means that they don't naturally take advantage of a paged architecture, which would allow your kernel to arrange applications' address spaces independently of the layout of the computers RAM chips.
- They don't do much with the memory they allocate: they don't allow things like sharing, swapping to disk, memory-mapped files, integration with the disk cache, etc.

What we need to do is write a layer of functions on top of the physical memory manager which deal with these problems. Note that *this* memory manager, like most of the kernel, can be made portable: the things it deals with are generally the same across different architectures (that is, until you start dealing with things like the Z80 and the 80286).

Specifications

What we're going to do is, effectively, write two memory managers: one which can manage the virtual address space for user applications, and one which implements `malloc()` and `free()` (or the equivalents in your language of choice) for your kernel. A virtual memory manager is useful for providing a large-block memory interface to your user applications: sure, it would be OK to expose the physical memory manager directly to user mode, but that would mean that user apps needed to know about the physical architecture of the target machine, and surely we're writing a kernel to avoid that kind of thing?

It's good to write versions of `malloc()` and `free()` for the kernel because it makes writing kernel (and driver) code easier and more enjoyable. I'm a great believer in providing a good run-time library interface in kernel mode, even if it means that the kernel grows in size. Different kernels provide these RTL facilities to different extents; the Linux kernel provides a small subset of the C RTL, and Windows NT provides its own set of functions which mirror those provided by the C standard. Linux calls its `malloc()` `kmalloc()`; Windows NT has `ExAllocatePool()` and friends; I'll be sticking with the familiar `malloc()`.

`malloc()` and `free()`

I'll start with these two because they are probably the simplest to implement. The goal of any `malloc()` is to take a large slab of memory and divide it up into pieces as requested, and that is what we'll be doing in the kernel. Here, we need to set aside a portion of the kernel's address space (remember, the kernel's address space ought to be separate from the user address space, in the interests of memory protection) from where `malloc()`'s memory will come.

I like to split my kernel's address space into several regions. The Möbius kernel, for example, starts at address `0xC0000000`; that is, it occupies the top gigabyte. Kernel memory is split up into 256MB-regions as follows:

0xC0000000	Kernel code, data, bss, etc. Probably too much but, hey, we might see a 256MB kernel one day.
0xD0000000	Kernel heap
0xE0000000	Space reserved for device drivers
0xF0000000	Some physical memory (useful for video memory access) Page directory and page tables of the current process

Note that 256MB of address space, from `0xD0000000` to `0xE0000000` is devoted to the kernel's heap. This doesn't mean that 256MB of *physical* memory will be given over to the kernel heap at all times; it just means that there is a 256MB window through which heap memory is accessed.

I'm not going to go into telling you how to write the `malloc()` and `free()` functions themselves; other people can do that better than me. There is a sample implementation in the book *The C Programming Language* by Kernighan and Ritchie, and there are several slot-in implementations on the Internet. However, one thing all `malloc()` implementations should have in common is a hook into the OS kernel to request more memory. In the more conventional user-mode environment, this function (often called `morecore()` or `sbrk()`) will ask the kernel for a large block of memory in order to enlarge the process's address space. Here, we are the kernel, so we need to write our own `morecore()`.

Incidentally, there is a slight difference between `morecore()` (as featured in *K&R*) and `sbrk()` (as used by most Unix implementations):

`void *morecore(size_t n)`
: Requests a block of size *n* bytes (*K&R* uses multiples of the allocation header size) which might come from anywhere in the application's address space

`char *sbrk(size_t d)`
: Adjusts the size of the application's address space by *d* bytes and returns a pointer to the start of the new region. Returns `char*` because the earliest versions of C compilers had no `void*` type.

The difference is subtle but it makes sense to be clear about these definitions. I'll be using the `morecore()` interface here, but it should be trivial to adapt it for `sbrk()`; in fact, there's no reason why `morecore()` can't work the same way as `sbrk()` internally.

Remember: `malloc()` calls `morecore()` when it runs out of memory. If it succeeds, the new block is added to the free block list (if it fails, `malloc()` returns `NULL`). In the kernel we need to increase the amount of space set aside for the kernel's heap by *n* bytes and return a pointer to the start of the block. But `morecore()` wants to return an address inside the virtual address space, and so far we only know how to allocate physical pages. Well, the obvious thing to do would be to:

1. Allocate a page using the physical memory allocator
2. Map it at the end of the kernel's heap using the physical memory mapper
3. Adjust the kernel's end-of-heap pointer
4. Repeat from (1) until enough memory has been allocated to satisfy the request

Note that there is no provision made to free the physical memory associated with the heap. I don't know of any `malloc()` implementation which will ask the OS to free the memory is has been given (the RTL usually assumes that the program's memory will be freed when it exits) but it should be possible to walk the free block heap periodically and release any memory used. As is is, this algorithm will only allocate memory as it is used: if a lot of memory is suddenly allocated and then freed, it will remain allocated (as far as the kernel is concerned) but it will be re-used next time `malloc()` is called.

That's all there is to the kernel side of `malloc()`. Remember that this `malloc()` is only for use by the kernel and by kernel-mode device drivers. User applications can have their own funky implementation of `malloc()`, or `new`, or `GetMem()` as required. These user-mode allocators can call the kernel's virtual memory manager, which I'll conveniently cover next.

The Virtual Memory Manager

The kernel's `malloc()` is all well and good; we can go off and write `strdup()` and `new` and make linked list functions now. But it's not very sophisticated. We can make a much more powerful allocator which can do more than allocate and free arbitrary small blocks of memory: it can do all the things I talked about in the Introduction. This is where we write the virtual memory manager.

As with the physical memory manager from part 1, the virtual memory manager will be managing the address space. However, this address space is the virtual one: it is a constant 4GB long (on a 32-bit machine) and it needn't correspond directly with the underlying memory installed the machine. Parts of it can be mapped to the same physical memory (to share memory); parts of it can be mapped nowhere at all (to provide guard pages, for example, at the end of stacks); parts of it can be unmapped but can emulate real memory (like in Windows NT's `ntvdm.exe`, which emulates the PC's BIOS Data Area for real-mode apps).

The key to all of this is the page fault. If you've done much Windows programming you'll probably be familiar with the page fault (along with the access violation) as one of the symptoms of a spurious pointer; Linux programmers (and users) will know it as a segmentation violation or bus fault. However, as applications programmers, we only see the page fault when things go wrong. In reality, the page fault is the processor's way of telling the kernel that it doesn't know how to handle an access to a particular region of memory, and that some software handler needs to be invoked to sort it out. Under the correct circumstances, the kernel will be able to do so and continue the execution of the application.

The most benign kind of page fault occurs when the processor tries to read from or write to a page which has been marked as 'not present'. Remember that each entry in the page directory and each entry in each page table has a 'Present' bit. The processor checks this bit on every memory access (actually it will cache the PDEs and PTEs to speed up memory access). If it is set then the access can continue; if not, it writes the address in question to the register `CR2` and invokes the page fault handler. This is installed the same way as a normal interrupt handler; see the **interrupts** tutorial for more details(not yet availible). I'll assume that you have got your interrupt handler sufficiently advanced that control will be passed to a routine when an interrupt occurs.

So the page fault handler is executed when a page is accessed which is, as far as the MMU is concerned, not present. The page fault handler needs to:

1. Check what should be located at the faulting address and,
2. Make sure the address is accessible if it should be, or,
3. Take the appropriate action if the address is incorrect

The “appropriate action” in this case will probably involve terminating the application in question. Different operating systems take different approaches over this; Windows NT invokes the current set of Structured Exception Handlers, the default action of which is to terminate the application; Linux invokes the appropriate signal handler.

Allocator

In order to check what should be located at the faulting address we need to keep records of how the address space is organized. To start with, we can write a simple virtual memory allocator. Forget about page faults for a moment while I describe what our virtual memory allocator will do.

In short, we need to be able to make good use of the 4GB given to us by the hardware designers. This comes down to writing a routine which can allocate an arbitrary amount of virtual memory. Note that the physical memory behind it doesn’t need to be allocated all at once, and it doesn’t need to be contiguous. Of course the physical memory does eventually need to be there – the processor needs something to send over the address bus to the main memory – but it’s not essential to allocate it all at once. For now it’s enough to record all allocations that take place in a list. Our `malloc()` can come in useful here because you can allocate one record per allocation and store things like the virtual address of the allocation and how many pages were allocated. In fact, we need to record at least the allocation base address and the size so that we can allocate (“commit”) physical memory later. It’s usually more worthwhile dealing in pages here, rather than bytes, because it makes the kernel’s accounting easier. Remember that the user-mode RTL can implement a `malloc()`-style allocator itself which can divide large blocks of memory into smaller ones. As a reasonable minimum you ought to record for each allocation:

- Virtual base address (i.e. the start of the block in the current application’s address space)
- Size
- Protection (e.g. read/write, user vs. kernel)

At this point we have a function which can ‘mentally’ allocate blocks of memory and record them somewhere in a list. If you’re implementing process-based protection in your kernel you ought to keep one list per process, to avoid confusion.

Once some application has a block allocated it can try to access it. However, there’s no physical memory associated with the block yet – the page table entries (and, in fact, the page directory entry) for that region are marked as ‘not present’. So the processor invokes a page fault, and calls our page fault handler.

The first thing the handler needs to do is check the allocation list to find the faulting address. The x86 processor records the actual address accessed, not the address of the start of the page, but this shouldn’t make any difference here. We need to walk the list to find a block which spans the faulting address.

Once the block is found, we need to commit it. This will involve allocating the relevant physical pages and mapping them into the address space, using the physical memory allocator from part 1. It’s up to you whether you commit an entire block at once or commit it page-by-page. Imagine an allocation of 4MB: that’s 1,024 pages. On the one hand, the application might write one byte at the start and then forget about it. If we committed the whole block on the first access, we’d waste 3.99MB (that is, until the application freed the block). Alternatively we could commit one page at once, and invoke the page fault handler when each page was accessed. If the application allocated the 4MB block and wrote to every byte we’d end up faulting 1,024 times: each fault has its overhead which would be unnecessary if the whole block was committed on the first fault. As with so many design decisions, it's up to you.

I mentioned the processor's page table cache before. This is called the Translation Lookaside Buffer and it is an internal copy of the PDEs and PTEs as they were when the cache was last invalidated. The cache is invalidated when `CR3` is written to and when the `INVLPG` instruction is executed (on the 486 and above). Invalidating the whole TLB takes a while so it should only be done when necessary (e.g. when switching between two processes). The `INVLPG` instruction is useful because it instructs the processor to invalidate only the region of the TLB associated with one page. The TLB must be updated when any changes are made to the page directory or page tables (the processor doesn't do it automatically) so it is necessary after allocating and mapping the physical memory here.

Assuming that the faulting address was correctly identified as part of an allocated block, and the relevant physical memory was allocated and mapped successfully, the kernel is free to return from the page fault handler and continue execution of the current application. The processor will retry the faulting instruction and, if the kernel has done its job, all will be well.

Executables

If we load an executable program into memory, we'll have a record of the makeup of that executable – that is, what sections it contains, where they are located, and how big they are. They will be laid out in the application's address space in some order, and they needn't all be loaded at startup. This is similar to the virtual memory allocator I just described, except that we already have a record of the executable's sections in the header. Why not map the file into memory as it appears on disk, headers and all. Then it becomes unnecessary to allocate the executable's memory with the virtual allocator.

To do this we need to walk the list of executable image sections at the same time as we check the list of allocated blocks. This is a matter of starting at the base address of the executable (that is, the address of the start of the headers), parsing the headers and locating the section which spans the faulting address. It may be more efficient to cache the sections' details in some other format, but most executable formats (including COFF, ELF and PE) keep it reasonable. Then we can write a "separate executable section committer" which will load an executable section from disk, allocate physical storage and map it. We can make some more optimizations here – for example, if code pages are read-only, it will be unnecessary to swap them to disk if they are already present in the on-disk image. Read-only code pages can also be shared among multiple instances, and data pages can be shared until they are written to; once they are written to, a copy is made. The principle of copying pages only when they are written to ("dirtied") is known as Copy-On-Write, and is commonly used.

The principle behind COW is that the originals of such pages are made read-only. The processor will fault when the application attempts to write to it – because they are data pages there's no reason why the application should avoid writing to them – and the error code on the stack will reflect this. The page fault handler detects that the faulting address is contained within a COW page, so it can make a copy of the underlying physical memory, re-map it in the address space, and allow the application to continue. Because the mapping has changed, the application will continue with the copy, which the page fault handler has now made read-write.

Copy-on-write is also used in the common Unix `fork()` paradigm. The `fork()` system call, which has no parallel in standard Win32, creates a copy of the current process which is identical – it uses the same code pages, data pages and stack pages are shared, and execution continues at the same location. To save memory, as many pages are shared as possible. However, the new process and the old one are completely separate, so each is free to write to its shared copy of the data. Here COW is used to spawn new copies of the shared data when either process (parent or child) modifies them.

Swapping

Now that we're able to allocate memory on demand, we can do some more advanced things with our virtual memory manager. Granted, we can only implement the more advanced features when we've got things like a disk driver and a process manager going, but the theory's here. One useful feature is the ability of the virtual memory manager to swap pages to and from disk. This is present in most modern operating systems and it is responsible for the thrashing noise when the hard disk is accessed in a low-memory situation. The theory is simple: sooner or later, a block will need to be committed, and the physical memory allocator will run out of pages. We need to clear out some space for the new block: after all, the new block is being accessed *right now*, and there's probably blocks in the system which haven't been accessed for a while.

The swapper function needs to:

1. Find a block suitable for swapping out to disk. This will usually be the least recently used one, although Win9x will swap arbitrary blocks out randomly.
2. Make room for it in the swap file: talk to the file system and disk drivers for this. A fixed contiguous swap file is simplest (à la Windows 3.1): at startup, record the disk sector numbers associated with the swap file and talk directly to the disk driver when necessary. Alternatively you could access the swap file as a normal file, via the file system driver.
3. Write the old block to disk, and record that is has been swapped out
4. Repeat from (1) until the physical allocation request succeeds

Note that if any of this goes wrong, the commit will fail. Remember that if a page fault occurs in a block which has been swapped out, it must be swapped back in again.

Overview

A typical sequence of events after an illegal address is accessed might go as follows:

1. Processor triggers page fault.
2. Control is passed to the kernel, which calls the page fault handler.
3. The page fault handler looks through the list of allocated blocks for the current process.
   3.1. A block has been found which spans the faulting address; the "commit" function is called.
   3.2. The commit function allocates a physical page for each page in the block's range.
   3.3. If the block was in the swap file, each page of the block is read in from the swap file.
   3.4. The relevant mappings are made in the current process's address space, according to the protection flags given when the block was first allocated.
   3.5. Control is passed back to the application.
4. The page fault handler looks through the list of executable modules for the current process.
   4.1. An module has been found which spans the faulting address; the "dynamic load" function is called.
   4.2. The dynamic load function looks for a section in the module which spans the faulting address.
   4.2. It looks for a copy of that section already loaded into another process's address space.
      4.2a.1. Another copy of the section has been found. Its reference count is incremented.

4.2a.2. The physical memory for that section is mapped into the current address space read-only. A later write to the section will incur another page fault; then, the physical memory for the section is copied and unshared.

4.2b.1. The section has not been loaded anywhere else.

4.2b.2. Physical memory for the section is allocated and mapped into the current address space.

4.2b.3. The section is loaded from disk into memory.

4.3. Control is passed back to the application.

5. No valid block has been found; the current process is either terminated or control is passed to its exception/signal handlers.

I've gone through the major functionality of a reasonably-sophisticated virtual memory manager. The VMM is able to split up applications' address spaces into reasonable chunks, and the kernel `malloc()` makes writing the rest of the kernel a lot easier. But so far all we've got is still a bloated "Hello, world" program, albeit one which uses protected mode and which can allocate a full 4GB of address space. No operating systems ever sold themselves on the ability to switch from one archaic processor mode to another which became public in 1986, so it's time to give the kernel some work to do and start **scheduling some code**(not yet avalible).

Created 24/12/01; last updated 01/01/02.
Updated by K.J. 03/06/02

This tutorial is mirrored on this website with premission from Tim Robinson.