Memory Management 1

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

Introduction

I'm writing this tutorial in an attempt to help you to write a memory manager for your own OS. I assume that you've decided to try and write your own OS kernel, and that you've got past the "boot-and-print-a-message" stage. Congratulations – you've got further than most people.

A memory manager (that is, a physical memory manager) is one of the components at the lowest level of every operating system, and it is vital for the splitting-up of your computer's memory. Note that there might be several memory managers in your kernel, each working at a different level. I'll be explaining the lowest-level one here. The allocator here won't work as a `malloc()` unless you're happy with a 4096-byte granularity.

I'm not going to tell you exactly what to do. I'll give some examples, and you're free to copy what I've done, but every OS does things slightly differently. And if you're writing your own operating system, you're probably not just copying one somebody else has written.

I'm assuming that your kernel is running in supervisor mode on some protected architecture. On the x86, this means that you're running in privilege level 0 (PL0) in protected mode on a 386 or higher. I'm not assuming an x86 processor, but until unless somebody gives me anything else, I'll keep giving x86 examples. I'll also be giving examples in C: you're free to write kernels in assembler, or Pascal (is this possible?) but I believe that C is a good semi-universal language for doing these things in. It also happens that I wrote my own kernel in C.

I apologize if this 'tutorial' is long and waffly. Instead of teaching you what to do (there's already plenty of hardware manuals, architecture tutorials and sample kernels out there to teach you that), I want to teach you how and why you're doing it. I hope this is of some help.

Specifications

The low-level memory manager is going to be fairly simple. When control is passed from the boot loader to the kernel code, you've got full control of all the hardware in the computer system. Your code is running in some part of memory (put there by the boot loader), and you've got some kind of stack and data area set up. In short, you've got the operating environment set up enough to print call your `main()`, print "Hello world" and stop.

The physical memory manager should to one of the first pieces of code to run (apart from the code which writes your name to the screen, of course). If you've got your kernel running to an extent that you can reference global variables (implying that your data section exists), call procedures (implying a stack) and write to the screen (implying that you can look at the rest of physical memory too), you may be tempted to just muddle through, switch to graphics mode and write a GUI. This may well be possible (it was the Windows philosophy until 1995) but you're going to come unstuck once you try more sophisticated OS things (again, the Win9x philosophy).

The physical memory manager's task is reasonably simple: to split up your computer's physical address space into reasonably-sized chunks and give it out to the various parts of your kernel that need it. For simplicity, we'll be calling the chunks "pages": on the x86, one page is (by default) 4KB in size.

The x86 is fairly unique among modern architectures in that it deals in both segments and pages; that is, memory is referenced via segments, which in turn reference either raw bytes or chunks of memory in pages. There are two main advantages in using the paged architecture:

- Most other 32-bit architectures split their 4GB address space into pages, as the 386 does. The x86 introduces segmentation, which allows the OS to reference memory as code, data, stack etc., but in this respect the x86 is different to most other processors. It would be nearly impossible to port a segmented x86 operating system to another architecture; also, no mainstream compilers can handle segmentation in 32-bit mode, so you're restricted to assembler on a segmented OS.
- Without using paging, the x86 can only address 16MB of physical memory. This comes back to the 286: as a 16-bit processor running on a 24-bit bus, it could only address 16MB. Without segmentation, you were restricted to 64KB. When the 386 came out it became possible to divide memory into 4KB pages, the advantage of which was that it was now able to look at the full 4GB address space allowed by a 32-bit processor. The disadvantage of this was that it was now not possible to allocate memory in blocks smaller than 4KB if you wanted more than 64KB (although, if you read on, this becomes less of a problem).

So we're going to be using a paged memory manager on a 32-bit architecture. There are some people who swear by segmentation. Me, I swear at segmentation; you're free to join them if you prefer. Just remember that you won't be getting any segmented memory management tutorials from me, OK?

Good. Now we can get down to some specifics. My aim here is to virtually eliminate physical addressing from pretty much everywhere in the kernel except the memory manager. This way your OS can be independent of things like how much memory the machine has, and how it is split up. For example, the Pentium and above have an option giving you 4MB pages, with a 36-bit physical address space giving you 64GB (although the virtual address space is still restricted to 4GB). With a properly separated physical memory manager your OS will be able to take full advantage of a machine with huge amounts of memory with no changes outside of the memory manager; without, you'd probably need to recompile all your applications (not just the kernel) which could get tedious, to say the least.

Organization

Fundamentally, your memory manager needs to set aside a region of memory to manage the rest of memory. This might seem like a chicken-and-egg scenario: how do I allocate memory control information (such as length, etc.) before the allocator can work? I can see two ways of doing this:

1. Set aside part of each block allocated to store allocation information, in a header
2. Set aside a region of memory for each block allocated

If you're allocating small blocks (that is, with a resolution of one byte), option 1 is useful. This is often used by `malloc()`-style allocators. However, we want to manage memory in pages, and to put a few bytes of control information at the start of each page would be wasteful. In this scheme the memory control information could also be corrupted by buggy user applications: by writing to one byte before the start of your allocated buffer, you could crash the allocator.

Therefore, for a low-level allocator, option 2 is often best. Remember at this point we can access all physical memory as if it were global data: we just need to point a pointer at a region of memory and we can read from and write to it. At this point I'd like to cover a handy way of locating your kernel (on the x86 at least). You can skip this next bit if you don't understand, or don't care.

Your boot loader might load your kernel to, say, the 1MB mark (out of the way of the PC BIOS) and jump to it. So you'd link your kernel so that it started looking for code and data at 1MB and above. Later on you're going to start loading and running user applications. You're going to want to split your address space into user and kernel regions: the user region for all the applications' code and data, and the kernel space for the kernel and device drivers. The kernel should be inaccessible to the user apps. There's nothing to stop you keeping your kernel at 1MB and putting your user space higher up (starting at, say, 2GB – remember your virtual address space is completely independent of your physical address space). A lot of operating systems (Win9x, NT and Linux at least) like to have the kernel starting at 2GB and have the user space contained within the bottom 2GB. How do we accomplish this if the kernel is, in reality, loaded at 1MB? With paging this is easy: just point the relevant page table entries at the correct physical addresses, and link the kernel so that it starts at, say, address `0xC0000000`.

But putting the memory manager in the boot loader isn't practical. We need to have the processor seeing address `0xC0000000` as `0x100000` without enabling paging. Later, we want to enable paging and avoid relocating the kernel. We can do this by fiddling the base addresses of the kernel's code and data. Remember that the processor forms physical addresses by adding the virtual address (e.g. `0xC0000000`) to the segment's base address and sending that to the address bus (or, if paging is enabled, to the MMU). So as long as our address space is continuous (that is, there's no gaps or jumps) we can get the processor to associate any virtual address with any physical address. In our example before, with the kernel located at `0xC0000000` but loaded at `0x100000`, we need a segment base address which turns `0xC0000000` into `0x1000000`; that is, `0xC0000000 + base = 0x1000000`. This is easy: our segment base address needs to be `0x41000000`. When we refer to a global variable at address `0xC0001000`, the CPU adds `0x41000000` and gets the address `0x1001000`. This happens to be where the boot loader loaded part of our kernel, and everyone's happy. If we later enable paging, and map address `0xC0000000` to `0x1000000` (and use a segment with base address zero), the same addresses will continue to be used.

Here we rejoin the people reading this who don't care. If you recall, we're going to set aside a separate piece of kernel memory to manage the rest of it. Remember we'll be managing memory in largish pieces – 4KB on the x86 – and if we use less control info per block than the size of each block (page), it's going to work.

At the very least we need to know which page is allocated and which isn't. Immediately we can think of using a bitmap. We need one bit for each page in the system – on an x86, we're only using one 32,768th of physical memory to manage it. On a 256MB system, we'd need a 8192-byte bitmap to manage all 65,536 pages. The low-level memory manager only needs to know whether a page is or isn't allocated. Eventually, we may need to know things like how big the block is, which process allocated it, and which users have access to the block. However it's easy to write a higher-level memory manager based on this low-level one. All we're concerned with here is dicing up the physical address space and providing an interface to the higher-level bits of the kernel.

One advantage of the bitmap is its space efficiency, and its simplicity. Each page only needs one bit of control information: either the page is allocated, or it isn't. However we need to search the entire bitmap each time a page is allocated; on a large system, search times might become significant.

An alternative (which I believe is used by Linux and Windows NT) is to use a stack of pages. The (physical) addresses of free pages are pushed onto the stack; when a page is allocated, the next address is popped off the top of the stack and used. When a page is freed, its address is pushed back onto the top of the stack. With this, an allocation (or deallocation) becomes a matter of incrementing (or decrementing) some pointer. However, although most allocations don't require the physical addresses to be continuous (the MMU can make discontinuous physical addresses continuous to user apps), things like DMA do. If something requires physical addresses to be continuous then the memory manager needs to take addresses out of the middle of the stack, which complicates things.

The stack approach also makes it harder to choose where physical memory comes from. Returning to the DMA example, ISA DMA requires addresses to come from the first 16MB of memory (due to the 24-bit address bus). If we were to write a floppy drive controller driver, we'd need to allocate a buffer below the 16MB mark. A simple solution to this would be to maintain two stacks: one below 16MB, and one for the rest of memory. To take full advantage of the system's memory we could take "main" memory from the "low" stack as well: we usually don't need 16 megabytes for a floppy drive transfer buffer; what if the system only has 8MB in the first place?

Initialization

To set the ball rolling we need to allocate some memory for our control information, whether it is a bitmap or one or more stacks. The amount of control info will be proportional to the amount of memory in the system: the best way to determine this is to get the boot loader to call the BIOS (on the PC) and determine it that way. It is possible to read the CMOS NVRAM for the amount of memory installed (whilst remaining in protected mode), but that will only tell you about the first 64MB. You'll need to call BIOS interrupt 15h to do it properly; without running in VM86 mode, the best way to do this is in the boot loader.

So we know how much memory the user has in their system: now to set aside a region of memory for the memory manager. How do we do this before we've even written an allocator? We want some memory that is outside of the allocator's jurisdiction: some reserved memory. Remember that we have some more reserved memory in the (PC) system: the BIOS, the BDA, the kernel itself. All of these should not be touched. Why not tack our bitmap or stack onto the end of the kernel's memory and treat it as such: as a kind of dynamic global variable?

We know the address of the start and end of our kernel (you can set a symbol to the end of the kernel image in a GNU ld script, for example). From this we can work out how big the kernel is so that we can avoid allocating memory from the kernel's space – to allocate memory from the middle of the kernel would be fatal. So we can enlarge the kernel's recorded size by the size of the bitmap/stack region, point a pointer at the start of the region and mark that space as reserved in the bitmap/stack itself. In the case of a bitmap we need to set the bits for that region to all ones; for a stack, we push all addresses except the reserved region. The same goes for the BDA (the memory below address `0x500` which is needed to run a VM86 monitor later) and the BIOS/ROMs. Your initialization function also needs to enable paging, which I'll cover later.

Allocation

Our first memory manager function is the physical page allocator. This will mark one physical page as used and return its address. If we're using a stack this is easy: we keep track of the number of free pages and, when a page is allocated, decrement the number of free pages and return the address of the page at the top of the stack. For a bitmap, we'll need to do some bit twiddling to scan through the array of bits and mark one as used. I prefer the stack approach.

Going back to the problem of allocating specific addresses, such as an address below 16MB for ISA DMA: this is easy with a bitmap (just stop at a certain offset). With a stack, you could either maintain two page stacks, or write a second allocator that scanned though the main stack for a certain address and mark it as used. I prefer the two-stack approach. Also note that it is difficult to allocate a continuous range of addresses from a stack. Luckily most times we can manage with repeated allocations of one page at a time.

What do we do if there aren't any free pages left? Well, if we have a swap file on disk, we need to swap pages out until we have enough space free. However, to do that, we need to write a lot more kernel framework (at least disk and filing system drivers), so, for now, it's enough to panic on out of memory. It's not as if you're going to allocate all 1GB of your machine's memory on a hello world kernel, is it?

The address returned by your low-level allocator is a physical address. As such, you can read from/write to it in your kernel as much as you like. You can put data there. However, your memory manager becomes much more powerful one you start mapping pages from your virtual address space to physical pages in memory.

Deallocation

Deallocation is effectively the reverse of allocation so I'm not going to cover it thoroughly. Either clear a bit in your bitmap or push an address onto your stack. Easy.

Mapping

This is where the paging architecture starts to get useful. In short (and the processor manual explains this better), paging allows you to control the address space from software. You can map pages anywhere you like; you can apply protection to pages; and you can allocate and map pages on demand via page faults (or even emulate memory access completely).

I'll leave the MMU description to the hardware manuals, and for this, I'll stick to the x86 (although other architectures are fairly similar). The 386 and above use a three-level translation scheme: the PDBR (Page Directory Base Register, called CR3) contains the physical address of a page directory. The page directory is a page in memory split into 1,024 32-bit words, or Page Directory Entries, each of which is the physical address of a page table. Each page table is, too, split into 1,024 Page Table Entries; each of these words is the physical address of a page in memory.

Note that, to span 4GB of 4096-byte pages, you don't need 32 bits for each physical address. The PDBR, PDEs and PTEs only need 20 bits each, so the bottom 12 bits are used as flags. These flags can apply protection (read/write and user vs. supervisor) to pages and page tables, and they can mark individual pages and page tables as present or not present. There is also an 'accessed' flag, which the processor sets when a page or page table is accessed. By switching the PDBR you can use different page directories and page tables in different contexts, thereby causing separate applications to use different address spaces.

Paging is immensely powerful, and its true benefits aren't fully realized until you start writing user applications to make use of the address space. Now we need to write functions in our memory manager to maintain the various page directories and page tables.

Remember that page directories and page tables are just normal pages in memory; the processor needs to know their physical addresses. There's only one page directory per process (or per address space) and there can be up to 1,024 page tables per page directory. We can allocate these as normal from our physical allocator we wrote before. Remember that page directory and page table addresses need to be page-aligned: that is, the bottom 12 bits need to be zero (otherwise they would interfere with the read/write/protection/accessed bits).

Before we enable paging we need valid a PDBR and page directory, and a valid page table for wherever we're executing. The current page table needs to identity map the current EIP: when we set the paging bit in CR0, the CPU will still be executing from the same address, so it had better have some valid instructions at wherever it's executing. This is why it's good to locate the kernel at whatever address it's going to end up once paging is enabled.

So before we enable paging, we need to allocate a page directory and one page table. You could use the page allocator or you could reserve giant (4096-byte) global variables: either way it's the same. If you're using global variables for your page directory/page table, be sure that they're page-aligned.

Zero all the entries you're not using. This will clear the present bit so that accessing the addresses they cover will cause a page fault. You could set the other bits to some distinctive pattern if you want your page faults to look good. Remember that each PDE (and page table) covers 4MB of your address space, which should be enough for your kernel – if not, I'd like to see it. Each PTE covers only 4KB, so you'll probably need several PTEs to cover your kernel (again, if not, I'd like to see it…). Assuming you're not relocating your kernel dynamically (and why? Since it's the first user of the address space, it's not likely to clash), you can just poke numbers into the page directory and page table you've allocated as appropriate. You'll be making your memory-mapping function more sophisticated later.

All of this needs to go into your memory manager initialization routine, which I covered before. Since your memory manager should separate all the machine-specific memory stuff from the rest of the kernel, it's polite to enable paging before your return, and it's polite to keep the address format the same (if not, then the return address on the stack from the last function call will be wrong). So set up your page directory and page tables, enable paging, do a far jump (to reload CS) and reload the other selectors (SS, DS, ES, FS and GS) as appropriate. Now that you've enabled paging it makes sense to use segments whose base addresses are zero, although there's no reason why this should be mandatory.

With luck, we're now running with paging enabled. All addresses now are going through the page directory and page tables we established before. Note that the rest of physical memory is not necessarily mapped into our new address space – all that is necessary is the part of code that the CPU is currently executing (in practice, you'd map the whole kernel: data, stack and all). We'd like to map in video memory so we can print some more messages, and the full memory-mapping routine comes in useful.

Memory mapping

At first glance this appears pretty easy. Just poke words into the right pages in the current page directory and page table; allocate a new page table if required. However, the CPU uses physical addresses for page directories and page tables; we're using virtual addresses for everything. There's a number of ways around this:

1. Map all physical memory into the address space. This can either be done 1:1 (that is, physical memory is addressed by the bottom of the address space) or at some offset (that is, physical memory is accessible starting at, say, `0xD0000000`). This approach's advantage is its simplicity (Win9x uses this); however, its disadvantage is the fact that the user may have any amount of memory installed in their system, all of which must be addressable. Imagine if the user had 4GB installed: there would be no address space left…
2. Map each page into the address space and keep track of their virtual addresses in a virtual page directory parallel to the real one. The virtual page directory can store the virtual addresses of each of the page tables while the real page directory stores their physical addresses. This is good if the only pieces of physical memory which must be addressed directly are the page directories/page tables; however, it increases the amount of space taken up just by mapping – not good in a small system.
3. Map the page directory into itself. This might seem like a kind of weird fractal memory mapper, but it works well in practice. By setting one of the fixed PDEs to the physical address of the associated page directory, you can address PDEs and PTEs as separate addresses. If you set element 1023 of each page directory to the physical address of the page directory itself, the processor will see the page directory as the last page table. It will see the PDEs as PTEs, and it will see the PTEs as individual 32-bit words in

the top 4MB of the address space. You can use the top 4KB of the address space as the entries in the original page directory. This has the advantage of being beautiful yet simple; it has the disadvantage that you can only access page mappings inside the current address space.

By way of an example, Windows NT maps up to 512MB of physical memory into the kernel's address space (as in option 1) while mapping the page tables directly into the address space (as in option 3). Personally, I'd go for the third option, even though it takes some thinking about to get your head round it. The first option also has its advantages for a simple kernel. Either way, the page mapper's job is simple. Remember to apply the correct protection at each stage: on the PTE, apply the desired protection for that page; on the PDE, apply the desired protection for that 4MB region. Each PDE should normally be made read/write and user-visible unless you have a good reason for making the whole 4MB region inaccessible from user mode.

Overview

That's about it for our simple physical memory manager. If you want HAL-style machine abstraction you'll probably need an address space switching function callable from the scheduler (on the x86, this is just `MOV CR3, addr`). If you're going for a full VMS- or NT-style asynchronous device architecture you'll need a routine which can lock a buffer in the virtual address space and record the physical pages associated with it; however, that comes under 'advanced' design and you probably won't need it for a simple Minix or Linux model.

Before we can use our address fully, we'll need a more sophisticated memory manager, so here's a look forward to part 2 of the memory management tutorial.

Created 23/12/01; last updated 01/01/02.
Updated by K.J. 02/14/02

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