



Sign In

Create Account

Edit Style

Community

Tutorials

Rules, Guidelines & FAQ

View New Content

New York Philharmonic

Hover to Expand



[tutorial] Paging: Memory Mapping With A Recursive Page Directory

Started By xWeasel, Jun 30 2008 04:01 PM

xWeasel

Posted 30 June 2008 - 04:01 PM

Introduction

I am writing this tutorial because I feel there is a lack of resources on the Internet which properly explain paging, more specifically virtual memory mapping using the recursive page directory technique. There are plenty of articles which provide working code to enable paging and fill out your initial page directory and tables, so I will not be going over that. The problem I find many people have is with what to do next – how to map memory to a given virtual address. Many tutorials offer a brief explanation of how the page directories and tables work, and some even mention recursive page directory mapping, which is (at least in my opinion) the proper way to do virtual memory mapping. However, I myself found when I was reading them that the information provided was not enough, and ended up wasting a couple of days coding an over-complicated memory mapper which I then redid in 1/5 the code and 20 minutes using a recursive page directory. I feel that this technique has many advantages over others that have been suggested and used by people, and that many would benefit from fully understanding it and being able to implement it into their operating system.

Virtual Memory - Overview

If you have ever worked with pointers while coding something, either on Windows or *nix, there's a great chance that you have noticed the addresses in memory at which your data resides. In a usermode program on Windows you are likely to see addresses of anywhere up to 0x80000000 – which is 2GB! If you've worked with kernel mode, then these memory addresses go up as high as 0xFFFFFFFF – 4GB. Many computers don't even have that much memory! So how can this possibly work? The answer is simple – all of these addresses are *virtual* – which means they don't indicate the location of the data in your computer's *physical* memory, but a virtual location which could be mapped to anywhere in physical memory, or even paged out to disk.

As your OS grows in size and complexity you will realize the need for proper memory management, and with this comes the need for paging. It is possible to code an operating system where all programs use physical memory directly, but this requires the application developer to know how much physical RAM is available on the system, which parts of it aren't reserved for hardware use, and so on. Additionally, once you implement multitasking and processes, you will see that with a physical memory management model every process can access the data and code of any other process, and, perhaps more frighteningly, that of the kernel, without restriction – which is a huge risk not only to system stability but also to user security. With paging, however, these problems are all solved. The user does not need to know anything about the inner workings of the low-level memory manager, and which physical addresses are accessible, but only has to deal with virtual addresses in the address space of the current process – virtual addresses that your kernel will map somewhere and return to the user. Processes will also not be able to interfere with each other's data, as they will run in different address spaces and not have access to each other's memory. For example, 0x30000000 in one process can be mapped to one location and to another in a different process. This way when one process reads from that address it sees different data than a different process would if it read from the same address. This is because the address is mapped to a different location in each process, so logically the data in these two different locations is different.

Anyways, I think that's enough for an overview of the concepts I'll be talking about in this article. There are already plenty of resources on the theoretical aspects and advantages of virtual memory and paging out there on the Internet, so if that's what interests you then go read those. I'll be concentrating more on the technical aspect and the practical implementation of paging.

Requirements

Okay, so if you're still reading this then by now you're probably wondering "how do I implement this into my operating system?" Before you do that, you have to make sure that you are at a point where you are ready to implement paging. When you're reading this you already have a physical memory manager done and working. If you don't yet have one, you should probably go write one. It really is only about twenty minutes of work at most, so it's nothing difficult. For this tutorial I'll assume that you have a function called `mm_allocatephyspage` which takes zero parameters and returns an unsigned long which is the physical address of the page that has been allocated. It is up to you how you write this function – you can use a stack or a bitmap, as long as you have some way of keeping track of which pages are used. This allocator does not need to be able to allocate memory blocks of various sizes; it will only be allocating one page at a time. Remember that one page is 4KB. You should also have a function to free the page (let's call this `mm_freephyspage`), which takes one parameter – the physical address of the page to be freed. This should simply either set a bit to 0 in a bitmap, or push/pop an address from a stack, depending on how you've implemented it. It would also benefit you to have working `memset` and `memcpy` functions, as those come in useful.

Paging

Enough of that, let's get to paging! Before I read my first tutorial on paging (Tim Robinson's Memory Management Part 1 & 2 on osdever.net – an excellent tutorial, much credit goes to the author), I had doubts regarding the implementation of this whole "virtual memory" thing, and how to take care of and keep track of the virtual-physical translations. It turns out, however, that the processor already does this for us! The x86 architecture (don't ask me what the first processor was to support paging, because I honestly don't know and don't care!) has support for something called "page directories" and "page tables", which assist in mapping virtual addresses to physical addresses. If you want to see an overwhelming amount of information on the topic then please have a look at the Intel manuals. I will try to explain this briefly but with enough detail to allow you to understand it fully.

Page Directories

The first paging structure with which you must familiarize yourself is the page directory. You can think of this as the "top-level" table. It covers the whole 4GB of addressable virtual address space – from `0x00000000` to `0xFFFFFFFF`. This table consists of 1024 32-bit entries, and takes up 4096 bytes in memory (you can see how that works, $1024 * 4 = 4096$). Not coincidentally this is also the size of one page – 4KB. Each entry in this table is called a PDE, or Page Directory Entry, and it contains the physical address of a page table, more on this later. This is what one of these 32-bit entries looks like:

31.....12	11...9	8...7	6	5	4...3	2	1	0
Physical Address	avail	Reserved	Dirty	Accessed	Reserved	U/S	R/W	Present

As you can see, the bottom 12 bits are used for various flags. This leaves us with the top 24 bits to use as an address of a page table. The address will look like `0x12345000`, or `0xAAAA0000` – as you can see these addresses are 4K-aligned, or aligned to a page boundary. You can't have a page table that starts in the middle of a physical page, because if you have an address such as `0x0011DEAD`, once you OR it with any of the bottom 12 bits the bits that are already set as part of the address will interfere with the flags. This is an important fact to remember when you're allocating space for a new page table or directory.

Page Tables

You're probably confused right now, but before you can fully understand it you need to first get slightly more confused. Alright, so, page tables. What are they? A page table is a structure very similar to a page directory, except each of the entries (called PTEs or Page Table Entries), contains the physical address of a page in memory. The format of the PTE is the same as that of a PDE, with the same flags. The only difference is that it specifies a page and not a page table. Recall that a page is 4KB. Recall also that a page table, just like a page directory, has 1024 entries. This means that one page table covers 4MB of the virtual address space. 1024 of these entries in a page directory is where you get your 4GB virtual address space ($1024 * 1024 * 4096 = 4294967296$ bytes = 4GB).

Mapping

As you've seen, only a *physical* address of a page is placed into a PTE, so you might think – how do I tell the processor where to map the page in *virtual* memory? The answer to this is quite simple, and lies simply in which page table and which page table entry you place your mapping. Confusing? Not really. Imagine you have a physical address – `0x00012000`, and you want to map it to the virtual address `0xDEAD7000`. The first thing you need to do is figure out the index of the page table that covers the area into which your virtual address falls. Don't forget – there are 1024 page tables, and each of them covers 4MB, so unless you have a magic address that's above 4GB, there is a page table which will cover the specified region. Once you've got that in mind, it's just simple math. Divide the virtual address (`0xDEAD7000`) by `0x400000` ($0x1000 * 0x1000 * 4 = 0x400000 = 4$ MB). You should end up with `0x37A`, or 890 in decimal. This means that the `0x37A`th (that's kind of hard to say so just go with 890th) page table is the one which covers the region you're looking for – remember, saying "the 890th page table" is the same thing as saying "the 890th page directory entry". At this point you're half done. Now that you've figured out which page table the mapping will go into, you still need to write the address into the correct page table entry. This means you have to find out the PTE index of the virtual page to which you want to map a physical page. This, again, is just more math. Divide the remainder of the previous operation (`0xDEAD7000 / 0x400000`) by `0x1000` – which is the size of one page. `0xDEAD7000 % 0x400000` results in `0x2D7000`. Divide by `0x1000` and you get `0x2D7`, which is 727 in decimal.

Now that you have the index of the page table and of the page corresponding to your desired virtual address, you're ready to modify the PTEs! The first thing you need to do is get a pointer to the page table. This shouldn't be hard, right? Here's some code you could theoretically use to access the page table.

[-] C Source

[copy] (#) [popup] (#) [collapse] (#) ? (#)

```

1 | unsigned long *page_table = (unsigned long *) page_directory[890]; // assume page_directory e
2 | page_table[727] = 0x00012000; // remember, this is the physical address to which we wanted to

```

That works, right? No, it doesn't. We forgot about flags. Let's try again:

```

[-] C Source [copy] (#) [popup] (#) [collapse] (#) ? (#)
1 | unsigned long *page_table = (unsigned long *) page_directory[890] & 0xFFFFF000; // strip away
2 | page_table[727] = 0x00012000 | 3; // 11b - we want to set the present and the r/w bit

```

Okay, that's better. You can see how bitwise AND was used to keep only the top 24 bits of the PDE to get the address of the page table, and how the physical address was OR'd with 3 to set the Present and R/W bit. Those ones, for now, are the only bits we will set when mapping anything. Once again though, this code is incomplete. Have you thought about what will happen if the page table we're trying to write to doesn't exist?! When paging is initialized, the page directory is zeroed, or set to 2's (r/w, not present). So we'll be getting the address 0x00000000 for our page table and then writing to the 727th unsigned long from there. The result? A page fault, probably. Well, so what do we do about this? Recall that a page table actually needs to exist somewhere in physical memory, it can't just be a random address to nowhere. Also remember that a page table is 4KB, which is the size of one page. Yep, this is where the physical memory manager you wrote earlier comes in. We need to allocate a physical page and get the address of it – exactly what your memory manager does. Let's see what this code looks like now.

```

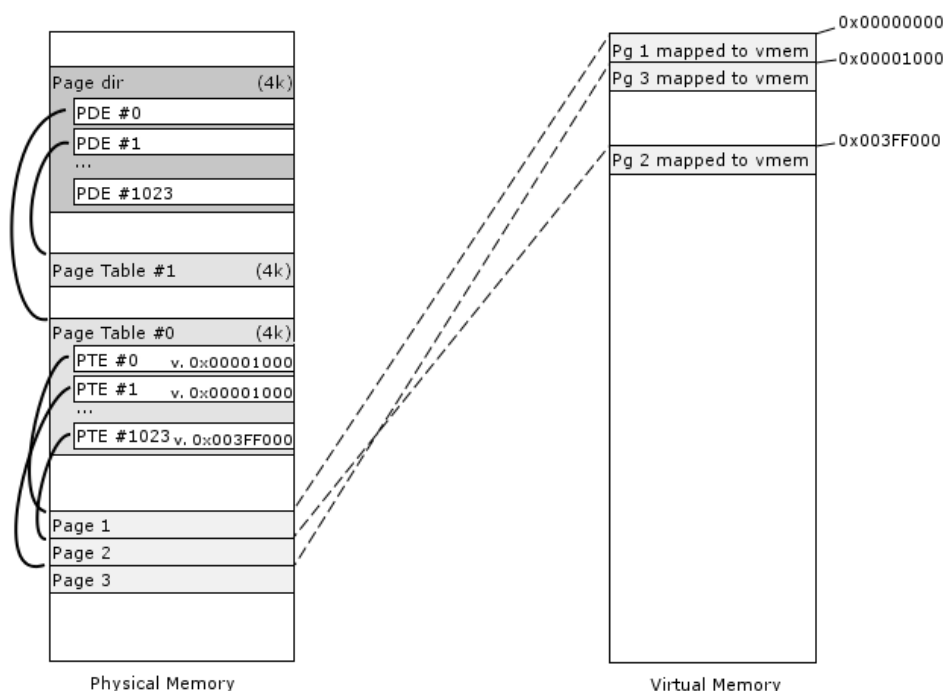
[-] C Source [copy] (#) [popup] (#) [collapse] (#) ? (#)
1 | // check if the present bit is set
2 | if(page_directory[890] & 1){
3 |     // page table exists, continue as we did earlier
4 |     unsigned long *page_table = (unsigned long *) page_directory[890] & 0xFFFFF000;
5 |     page_table[727] = 0x00012000 | 3;
6 | }else{
7 |     // page table doesn't exist, so create one
8 |     unsigned long *page_table = (unsigned long *) mm_allocphyspage();
9 |     memset(page_table, 2, 1024); // set 1024 dwords to 2 (r/w, not present)
10 |    page_table[727] = 0x00012000 | 3;
11 | }

```

Woot. We're done! ... - that's what you might think. We're far from it. There's another problem here - a big problem, too. If you've already realized what it is, then good job. If not, remember I said earlier that PDEs and PTEs are physical addresses of page tables and pages! By the time the mapping code runs it's assumed that we are already in paged mode (well duh! What would we be mapping if we weren't?!), and that means we can no longer directly write to or read from physical addresses. And what we did in that code right there was get the physical address of the page table and write to it. Ahem, page fault.

Recap

The next part will be somewhat difficult to grasp, so before I start explaining that I want to recap what I've said so far. I feel something like this is best explained with diagrams and examples, so I've taken the time to draw a half-assed diagram of how a simple paging setup might look.



I've tried to sort of colour-code stuff and remain consistent with lines and symbols. The left is the physical memory, and the right is the virtual memory. The black lines at the left side represent pointers from one place to another, and the dashed lines between physical and virtual are virtual-physical mappings. So, let's start at the top. The first thing we have in our physical memory is the page directory, which takes up 4KB. The first entry in it (PDE 0) points to an existing page table – Page Table 0. Remember that PDE 0 will simply be the physical memory address of Page Table 0, with flags in the lower bits. PDE 1 also points to an existing page table, but it appears none of the PTEs in that page table are filled in, so accessing any memory covered by that page table (4MB – 8MB in virtual memory) will likely result in a page fault. In page table 0 there are three PTEs that are filled in. PTE 0 contains the address of Page 1, which is simply a page somewhere in physical memory. PTEs 1 and 1023 also point to physical pages. Remember that if a PTE contains a valid physical address and the present bit is set, and the page table which contains the PTE is pointed to by a valid entry in the page directory, then you can access the page by a virtual address!

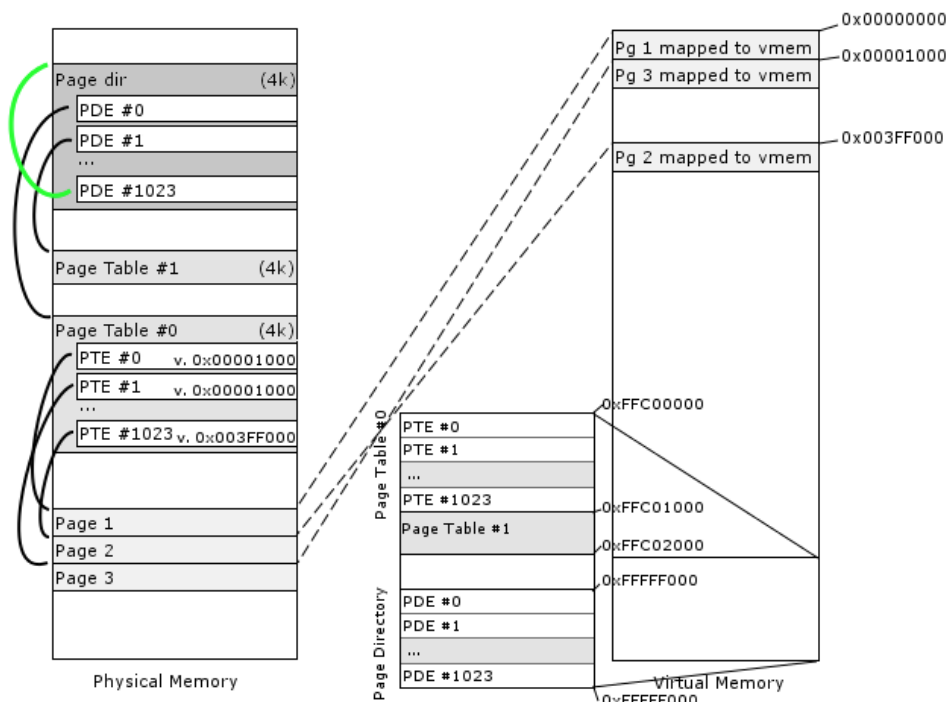
So let's see what happens when you access the virtual address 0x00000000 with the given setup. The processor first looks for the page table that corresponds to the address. Well that's easy, it's the first one! So, okay, now the processor knows that Page Table #0 is the one in which it must look for an entry that corresponds to address 0x00000000. Once again, we know right away that this will be the first entry in the page table, or PTE #0. The processor then takes the value from PTE #0 in Page Table #0, which is the address of Page 1 OR'd with several flags. This way, when you access 0x00000000 you are actually accessing the physical address of Page 1. You don't, however, know this physical address, but you also don't need to. The whole point of paging is to avoid the complexities that arise when working with physical memory directly. If you look further in the diagram, you can see that Page 3 is mapped consecutively after Page 1 into virtual memory at address 0x00001000. Page 2, which in physical memory comes after Page 1, is actually mapped way past Page 3, at virtual address 0x003FF000. This way paging can create the appearance of contiguous memory, while the underlying physical pages don't need to be consecutive at all.

Recursive Page Directory

Now let's get back to the problem at hand. We need a way to read from and write to page tables. This means accessing them by a virtual address, since we're using paging. There are a number of ways to do this. The way I did it originally was to keep part of the virtual address space for mapping page tables into it. This way, whenever a new page table was created, the physical address would be written into the page directory as part of the PDE, but it would also be mapped somewhere in virtual memory. This works, but there is a large amount of code and logic required to take care of mapping the page table somewhere, writing to it when necessary, and making sure it gets unmapped properly when it is no longer required. The problems with this approach became overwhelming when I began to implement separate address spaces for processes/threads, and I gave up on it. The way I ended up doing it eventually is the way suggested in Tim Robinson's memory management tutorial on osdever.net, and I found this way to be significantly more efficient. Once I switched over to this, the amount of code in my virtual memory mapper went from 500 lines to about 100.

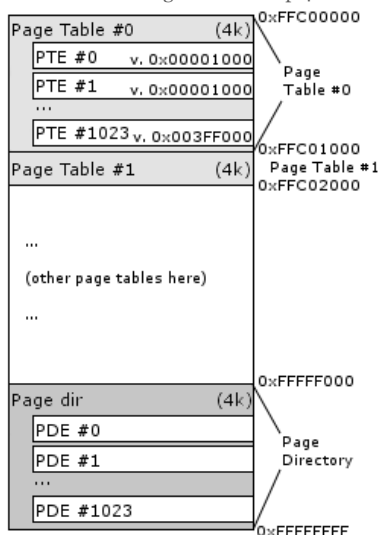
The way it works appears difficult to understand at first, but if you've got a good understanding of the paging architecture then you won't have any problems. This 'recursive page directory' memory mapping approach is based on mapping the page directory into itself. Yes, it sounds weird. Allow me to explain. During initialization of the virtual memory manager, the last PDE in the page directory is set to the physical address of the page directory itself. Remember that the processor looks for a page table at the address specified in the page directory entry. Also remember that a page table is 1024 32-bit values. Since the page directory itself is also 1024 32-bit values, there is nothing that stops the page directory from acting as a page table!

Since each page table covers 4MB of the address space, the last page directory entry will point to a page table that covers the top 4MB of the address space – that's 0xFFC00000 and above. Now, since we have the page directory acting as a page table, what happens when you write to 0xFFC00000? The processor first finds the page table that corresponds to that address – yep, that's our page directory. It then looks for the first entry in the "page table" (in this case it's looking at the first entry in the page directory). And what's the first entry in the page directory? Why, it's a pointer to the first page table – that's PDE #0, combined with the necessary bits set to show that it is present. The processor sees this PDE as a PTE which describes the page that is mapped to 0xFFC00000 – and that page turns out to be Page Table #0! So we now have access to the entire page table in our virtual address space, and we can modify any of the PTEs in it without having to worry about page faulting. Because of the way that the page directory structure is identical to the page table structure, we now have ALL of the existing page tables mapped to the top 4MB of our virtual address space. Conveniently, the top 4KB of the address space (the last page) is mapped to the page directory (recall how we set the last entry of the page directory to the address of the page directory, and how we have all the page tables mapped into the top 4MB). This way, by accessing memory at offsets from 0xFFFFF000, you can modify the PDEs in the page directory.



Notice the green line, which is the new mapping of the page directory into itself. In the middle is shown a zoomed-in view of the top 4MB of the virtual address space – the mapped page tables and the page directory at the end. You can see with the addresses at the side that each page table takes up 4KB in the top 4MB of the memory, so we can calculate the virtual address of a given page table as $(0xFFC00000 + (\text{page_table_idx} * 0x1000))$.

Here's another diagram of the top 4MB of virtual address space with the recursive page directory implemented:



Code

Now that we've got that all sorted out, we can finally write some code! Before we code our map page function, there's one thing we need.

```
[ - ] C Source [copy] (#) [popup] (#) [collapse] (#) ? (#)
1  typedef struct __attribute__((packed)) {
2      int pagetable;
3      int page;
4  }pageinfo, *ppageinfo;
5  pageinfo mm_virtaddrtopageindex(unsigned long addr) {
6      pageinfo pginf;
7
8      //align address to 4k (highest 20-bits of address)
9      addr &= ~0xFFF;
10     pginf.pagetable = addr / 0x400000; // each page table covers 0x400000 bytes in memory
11     pginf.page = (addr % 0x400000) / 0x1000; //0x1000 = page size
12     return pginf;
13 }
```

That's just a function which calculates the PDE index and PTE index for the given virtual address to assist us later in mapping. And

now the actual mapping functions!

```
[...] C Source [copy] (#) [popup] (#) [collapse] (#) ? (#)

1 unsigned long *kernel_page_dir = (unsigned long*) 0xFFFFF000; // last page in vmem is mapped
2 int mm_mmappage(unsigned long phys_address, unsigned long virt_address){
3     pageinfo pginf = mm_virtaddrtopageindex(virt_address); // get the PDE and PTE indexes fo
4
5     if(kernel_page_dir[pginf.pagetable] & 1){
6         // page table exists.
7         unsigned long *page_table = (unsigned long *) (0xFFC00000 + (pginf.pagetable * 0x100
8         if(!page_table[pginf.page] & 1){
9             // page isn't mapped
10            page_table[pginf.page] = phys_address | 3;
11        }else{
12            // page is already mapped
13            return status_error;
14        }
15    }else{
16        // doesn't exist, so alloc a page and add into pdir
17        unsigned long *new_page_table = (unsigned long *) mm_allocphyspage();
18        unsigned long *page_table = (unsigned long *) (0xFFC00000 + (pginf.pagetable * 0x100
19
20        kernel_page_dir[pginf.pagetable] = (unsigned long) new_page_table | 3; // add the ne
21        page_table[pginf.page] = phys_address | 3; // map the page!
22    }
23    return status_success;
24 }
25
26 void mm_unmappage(unsigned long virt_address){
27     pageinfo pginf = mm_virtaddrtopageindex(virt_address);
28
29     if(kernel_page_dir[pginf.pagetable] & 1){
30         int i;
31         unsigned long *page_table = (unsigned long *) (0xFFC00000 + (pginf.pagetable * 0x100
32         if(page_table[pginf.page] & 1){
33             // page is mapped, so unmap it
34             page_table[pginf.page] = 2; // r/w, not present
35         }
36
37         // check if there are any more present PTEs in this page table
38         for(i = 0; i < 1024; i++){
39             if(page_table[i] & 1) break;
40         }
41
42         // if there are none, then free the space allocated to the page table and delete map
43         if(i == 1024){
44             mm_freephyspage(kernel_page_dir[pginf.pagetable] & 0xFFFFF000);
45             kernel_page_dir[pginf.pagetable] = 2;
46         }
47     }
48 }
```

Conclusion

I hope that my explanation has benefited those of you who were confused about paging and memory mapping. Paging truly is a great feature of the CPU, and it is difficult to write a decent operating system without its implementation. If you have comments, questions, or if you've found mistakes in this article, please post in this thread or PM me.

Naltax

Posted 30 June 2008 - 05:09 PM

great article, im sure this will be great help to many people 😊.

```
[...] C Source [copy] (#) [popup] (#) [collapse] (#) ? (#)

1 | pginf.pagetable = addr / 0x400000; // each page table contains 0x400000 addresses
```

i think what you ment to say is each page table holds 1024 page adresses and each page is 0x1000 therefore each page table covers 0x400000 of memory.

also, you can calculate the pde and pte using bit shifting:

```
pginf.pagetable = addr >> 22;
pginf.page = (addr << 10) >> 22;
```

Edited by Naltax, 30 June 2008 - 05:18 PM.

xWeasel

Posted 30 June 2008 - 06:24 PM

Naltax is absolutely right. You can use bit shifting to get the index of the PDE and the PTE. Also, I fixed the typo in the comment. Thanks.

Echo

Posted 01 July 2008 - 03:49 AM

Nice read.

Spunkymonkey

Posted 18 August 2008 - 10:18 PM

Thanks for that, xWeasel - just about made sense of it 😊

Only started even thinking about playing with OSdev about a week ago and have a basic "boot, text onscreen and read the keyboard" system sorted so, going back to slightly more basic concepts, can I just check where this fits into the overall memory management plan as that seems to be the next thing that needs tackling.

As I (think I) understand it, first thing I need is a low-level memory allocator which is ONLY interested in marking which pages in physical memory are allocated. That's then used to supply physical addresses for mapping in my page directory / page tables as & when needed? As (well, if....) I get to the point of wanting more processes running, I then set up a complete new page directory / table structure for each process to provide it's own virtual memory space. Or have I misunderstood that last bit? 😊

Echo

Posted 19 August 2008 - 02:00 AM

- 1.) Physical Memory manager
- 2.) Create Page Tables
- 3.) Deal with PTEs
- 4.) Mapping
- etc..

Don't forget you need to make a page stack for this method.
Memory management is horribly tough so be careful.
Later you can allocate pages and stuff.

[e] - Page tables should be specified before you enter your kernel's entry point.
Also each process will be allocated a chunk physical memory which will be used in its own virtual memory space, thus preventing shared memory (once you get there) in user mode.

Edited by Echo, 19 August 2008 - 02:02 AM.

xWeasel

Posted 19 August 2008 - 05:25 PM

Spunkymonkey, on Aug 18 2008, 06:18 PM, said:

As I (think I) understand it, first thing I need is a low-level memory allocator which is ONLY interested in marking which pages in physical memory are allocated. That's then used to supply physical addresses for mapping in my page directory / page tables as & when needed? As (well, if....) I get to the point of wanting more processes running, I then set up a complete new page directory / table structure for each process to provide it's own virtual memory space. Or have I misunderstood that last bit? 😊

Yes - when you create a new process you must create a page directory for this process, and put the physical address of this page directory into the cr3 register. That is, if you want to have separate virtual address spaces for your processes, which I strongly suggest. You should also have your kernel memory mapped into every address space, for which you will likely want to implement a higher half kernel, ie. kernel memory is from 0x80000000 (2GB) and upwards. There are articles on the osdev.org wiki on this which you should read.

yilmaz

Posted 29 April 2009 - 04:05 PM

Thanks

nav143Posted 19 January 2012 - 07:48 AM

A small correction when you had explained about the flags When we use 12-bits for flags in a PDE entry , then what remains of a 32-bit entry is 20 bits and not 24-bits as u had mentioned in your tutorial.

Thankyou

NapalmPosted 20 January 2012 - 09:56 PM

You are correct in I think its a mistake. But do not dismiss the 24-bit as this could be 36bit addressing.

Napalm

azru0512Posted 16 August 2012 - 08:27 AM


Hi, Napalm. Nice article, but I have a question about what you said about your original implementation.

Quote

This way, whenever a new page table was created, the physical address would be written into the page directory as part of the PDE, but it would also be mapped somewhere in virtual memory.

When you write PDE into the page directory, doesn't it means the page table already had been mapped in virtual memory? I don't understand what else you need to do besides writing PDE. Could you explain a bit little more? Thanks.

[Back to Tutorials](#)

**Fios 50 Mbps Internet, Custom TV & phone**

\$ 79.99 /mo

for 2 yrs. + taxes, equip.
charges, FDV & other fees.

[Check Availability](#)

[See All Offers](#)

[Offer Pricing & Details](#)

[rohitab.com - Forums](#) → [Community](#) → [Tutorials](#)