Implementing Basic Paging

The Basics

First, we need to start by looking at basic paging terms and requirements.

Page Boundary
A block of memory that has a starting address which last 12 bits are *always* 0.
Page Frame
A 4kb unit of contiguous addresses of physical memory. A page frame begins on a page boundary and is of fixed size.
Page Directory
An array of 32-bit page table specifiers. 1 page directory takes up 4kb of memory to store.
Page Table
An array of 32-bit page specifiers. 1 page table maps 4MB of memory and takes up 4kb of memory to store.
Page Directory and Page Table Entries
An entry that tells about where a page points to, and its attributes:

31.....12	11...9	8...7	6	5	4...3	2	1	0
Page frame address	Avail	Reserved	D	A	Reserved	U/S	R/W	Present

Page frame address = Physical address of memory(either the physical address of the page, or the physical address of the page table)
Avail = Do what you want with this
Reserved = Reserved by Intel
D = Drity
A = Accessed
U/S = User or supervisor level
R/W = Read or read and write

Now there are two VERY important things to know. First, the paging bit of CR0(bit 31) when set to 1 enables paging. Second, before we set the paging bit to 1, we must put the address of the page directory into CR3.

Some Utility Functions

Since we will need to be able to read and write to the CR0 and CR3 registers, lets make some utility functions to simplify the process. The following NASM snippit(depending on your C compiler, you may have to remove the leading underscore for these functions to work) will work fine for what we need to do:

```
[global _read_cr0]
_read_cr0:
        mov eax, cr0
        retn

[global _write_cr0]
_write_cr0:
        push ebp
        mov ebp, esp
        mov eax, [ebp+8]
        mov cr0,  eax
        pop ebp
        retn

[global _read_cr3]
_read_cr3:
        mov eax, cr3
        retn

[global _write_cr3]
_write_cr3:
        push ebp
        mov ebp, esp
        mov eax, [ebp+8]
        mov cr3, eax
        pop ebp
        retn
```

The above code is self explainitory, so I'm not going to go over it any.

Setting up the Page Directory

Setting up the page directory is rather easy. We just need to find some free memory that is 4k aligned(if the address can be devided by 4096 and not have a remainder, the address is 4kb aligned). In the case, I have choosen 0x9C000 for the address(this may not work though depending on where your kernel is located at). Now we need to set up a pointer so that we can access that memory:

```
unsigned long *page_directory = (unsigned long *) 0x9C000;
```

Setting up the Page Table

We are only going to set up paging for the first 4MB of memory in this tutorial, so we are going to make one entry in our page directory and set it's attributes as *supervisor level, read/write, present*(011 in binary). All the other entries will be set as *supervisor level, read/write, not-present*(010 in binary). Before we do that though, we need to get the address of where we are going to put our page table(we will be using only one since we have set up the attributes of all the others to "not-present").

```
unsigned long *page_table = (unsigned long *) 0x9D000; // the page table comes right after the page directory
```

Now let's fill in the page table!

```
unsigned long address=0; // holds the physical address of where a page is
unsigned int i;

// map the first 4MB of memory
for(i=0; i<1024; i++)
{
        page_table[i] = address | 3; // attribute set to: supervisor level, read/write, present(011 in binary)
        address = address + 4096; // 4096 = 4kb
};
```

I think that the above code is self explanitory, so I'm not going to go over it. One that you should know though, you can have a page point to **anywhere** in the 4GB address space. In this example though, it is set up so that what appears to the programmer to be address 0x2250A000 really is 0x2250A000. But if we wanted to, we could make what appears to the programmer to be address 0x2250A000 really be 0x100000.

Now, we need to put an entry into our page directory that points to the page table:

Filling in the Page Directory Entries

```
// fill the first entry of the page directory
page_directory[0] = page_table; // attribute set to: supervisor level, read/write, present(011 in binary)
page_directory[0] = page_directory[0] | 3;
```

We still have 1023 entries to fill in on the page directory. However, we haven't set up page tables for them so we need to set the attribute of the remaining 1023 entries to *seupervisor level, read/write, not-present* which is 010 in binary(2 in decimal). Since these will be marked not-present, it doesn't matter what address we give it. In this case, we will just use the address 0:

```
for(i=1; i<1024; i++)
{
        page_directory[i] = 0 | 2; // attribute set to: supervisor level, read/write, not present(010 in binary)
};
```

You are probably wondering what happens if we try to access a page in memory marked *not-present*. What happens is the address that was trying to be accessed is put into CR2, and the page-fault handler is called(read Memory Management 2 for more about this).

Starting up Paging

We are almost done! All we have left to do is put the address of the page directory into CR3, and set the paging bit(bit 31)of CR0 to 1, and we will be using paging:

```
// write_cr3, read_cr3, write_cr0, and read_cr0 all come from the assembly functions
write_cr3(page_directory); // put that page directory address into CR3
write_cr0(read_cr0() | 0x80000000); // set the paging bit in CR0 to 1

// go celebrate or something 'cause PAGING IS ENABLED!!!!!!!!!!
```

Conclusion

Not as hard as it seems is it? Now there are still a ton of things that you can do, I suggest reading Memory Management 1 and Memory Management 2 to find out what else you should do(or might want to do). The complete source code for this tutorial may be downloaded here.

Written by K.J. - 2002 - thanks go to ByrdKernel for helping out with the definitions. Updated 2002.11.20 by K.J.