Setting Up Paging

From OSDev Wiki

This page is a work in progress and may thus be incomplete. Its content may be changed in the near future.

Difficulty level

🟩⬜⬜⬜
Beginner

This is a guide to setting up paging. It will teach you the basic concepts behind paging and how it can help you with your OS. This example will concentrate on Legacy Non-PSE Non-PAE paging (See also Setting Up Paging With PAE).

Paging is a term that refers to the management of the computer's virtual memory. If you have not yet created a physical memory manager, please read and follow Page Frame Allocation before continuing with this article.

Contents

Basic Paging

Paging allows you to have more than one virtual address space mapped into the physical address space. The MMU uses what is called a Page Directory to map virtual addresses to physical addresses.

Page Directory - A table in memory which the MMU uses to find the page tables.

Each index in the Page Directory is a pointer to a Page table.

Page Table - A table in memory that describes how the MMU should translate a certain range of addresses.

Each index in a Page Table contains the physical memory address to which a certain page should be mapped.

Creating a Blank Page Directory

The first step is to create a blank page directory. The page directory is blank because we have not yet created any page tables where the entries in the page directory can point.

Note that all of your paging structures need to be at page-aligned addresses (i.e. being a multiple of 4096). If you have already written a page frame allocator then you can use it to allocate the first free page after your kernel for the page directory. If you have not created a proper page allocator, simply finding the first free page-aligned address after the kernel will be fine, but you should write the page frame allocator as soon as possible. Another temporary solution (used in this tutorial) is to simply declare global objects with __attribute__((align(4096))). Note that this is a GCC extension. It allows you to declare data aligned with some mark, such as 4KiB here. We can use this because we are only using one page directory and one page table. Please note that on the real world, dynamic allocation is too basic to be missing, and paging structures are constantly being added, deleted, and modified. For now, just use static objects;

```
uint32_t page_directory[1024] __attribute__((aligned(4096)));
```

Now that we have a page directory, we need to blank it. The page directory should have exactly 1024 entries. We will set each entry to not present so that if the MMU looks for that page table, it will see that it is not there (...yet. We will add the first page table in a moment).

```
//set each entry to not present
int i;
for(i = 0; i < 1024; i++)
{
    // This sets the following flags to the pages:
    //   Supervisor: Only kernel-mode can access them
    //   Write Enabled: It can be both read from and written to
    //   Not Present: The page table is not present
    page_directory[i] = 0x00000002;
}
```

A page is "not present" is one which is not (intented to be) used. If the MMU finds one, it will Page Fault. Non-present pages are useful for technics such as Lazy Loading. It's also used when a page has been swapped to disk, so the Page Fault is not interpreted as an error by the OS. To the OS, it means someone needs a page it swapped to disk, so it is restored. A page fault over a page that was never swapped is a error by which the OS has a reason to kill the process.

Creating Your First Page Table

The second step is to create a basic page table. In this example we choose to fill up the whole first page table with addresses for the MMU. Because each page is 4 kilobytes large, and because each page table has exactly 1024 entries, filling up the whole table causes us to map 4 megabytes of memory. Also, the page directory is 1024 entries long, so everything can map up to 4GiB, the full 32-bit address space. Remembered the non-present page trick? Without it, we would use 16MiB per each paging structure. A single page directory needs 4KiB, but it can map some tables as non-present, effectively removing their space needs.

Now, its time to create a new page table.

```
uint32_t first_page_table[1024] __attribute__((aligned(4096)));
```

We now need to fill each index in the table with an address to which the MMU will map that page. Index 0 (zero) holds the address from where the first page will be mapped. Likewise, index 1 (one) holds the address for the second page and index 1023 holds the address of the 1024th page. That's for the first table. So, to get the page at which a certain index is mapped is as simple as (PageDirIndexOfTable * 1024) + PageTabIndexOfPage. If you multiply that by 4, you'll get the address (in KiB) at which the page will be loaded. For example, page index 123 in table index 456 will be mapped to (456 * 1024) + 123 = 467067. 467067 * 4 = 1868268 KiB = 1824.48046875 MiB = 1.781719207763671875 GiB. It's easy, right?

```
// holds the physical address where we want to start mapping the
// in this case, we want to map these pages to the very beginnin
unsigned int i;

//we will fill all 1024 entries in the table, mapping 4 megabyte
for(i = 0; i < 1024; i++)
{
    // As the address is page aligned, it will always leave 12 b
    // Those bits are used by the attributes ;)
    first_page_table[i] = (i * 0x1000) | 3; // attributes: super
}
```

Put the Page Table in the Page Directory

The third step is to put the newly created page table into our blank page directory. We do this by setting the first entry in the page directory to the address of our page table.

```
// attributes: supervisor level, read/write, present
page_directory[0] = ((unsigned int)first_page_table) | 3;
```

Enable Paging

The final step is to actually enable paging. First we tell the processor where to find our page directory by putting it's address into the CR3 register. Because C code cannot directly access the computer's registers, we will need to use assembly code to access CR3. The following assembly is written for GAS. If you use a different assembler then you will need to translate between this assembly format and the format supported by your assembler.

```
.text
.globl loadPageDirectory
loadPageDirectory:
push %ebp
mov %esp, %ebp
mov 8(%esp), %eax
mov %eax, %cr3
mov %ebp, %esp
```

```
pop %ebp
ret
```

This small assembly function takes one parameter: the address of the page directory. It then loads the address onto the CR3 register, where the MMU will find it. But wait! Paging is not still enabled. That's what we will do next. We must set the 32th bit in the CR0 register, the paging bit. This operation also requires assembly code. Once done, paging will be enabled.

```
.text
.globl enablePaging
enablePaging:
push %ebp
mov %esp, %ebp
mov %cr0, %eax
or $0x80000000, %eax
mov %eax, %cr0
mov %ebp, %esp
pop %ebp
ret
```

Now lets call the functions!

```
// This should go outside any function..
extern void loadPageDirectory(unsigned int*);
extern void enablePaging();
// And this inside a function
loadPageDirectory(page_directory);
enablePaging();
```

Paging should now be enabled. Try printing something to screen like "Hello, paging world!". If all goes well, congratulations! You've just learned the basics of paging. But there are lots of other things to do with it. You won't be able to almost all of them for now. Just remember that you have a little friend in the CR3 register that will help you one day.

More Advanced Paging Example

Add sections on how to dynamically get and free pages...