`Nano-Kernel : A Bare Metal OS

## Part 10 - Memory Management and Paging

Memory management is one of the most important tasks for an operating system. So far, the memory used by the Nano Kernel has been unchecked – that is, you can use any chunk of memory in anyway that you want! Our goal is to build up to supporting user-mode tasks and have virtual memory to allocate them and protect them. Memory protection works by configuring and then enabling the MMU. Before we can do this, we need to find out what memory is available in the system.

### BIOS, ACPI, or Multiboot

The computer’s BIOS can provide information about the available memory, including its capacity and usability. The BIOS can only provide this **before** protected mode operation is enabled. After enabling protected mode, all of the BIOS’s built-in features are no longer available. By the time our kernel has started, Grub has already done this, so this is not an option.

The Advanced Configuration & Power Interface (ACPI) is a standard developed by Intel, Microsoft, and others. ACPI is implemented as a separate co-processor to the Intel CPU and works with the CPU, OS, and *chipset controllers* to manage power states, device configuration, and interrupt priorities. We could certainly go after the ACPI controller and use it, but this is an additional complication that we don’t really need at this point.

Instead, the Multiboot software protocol requires that the boot manager (Grub) create a Multiboot information structure in memory. Grub will use the BIOS to collect information about the system and build this structure. We’ve totally ignored this structure until now, but it contains several important things, most notably, the *physical memory map* for the computer system.

### Address Spaces and Physical Memory Maps

Recall that the Intel CPU provides a 32- or 64-bit address space. That is, a single pointer is either 32- or 64-bit wide, and the computer can theoretically address bytes. Computers may not have that much memory, and even if they did, some of that memory is reserved for *memory mapped I/O*. In fact, we’ve already seen that in the case of the video graphics adapter. Read-Only Memory (ROM) devices may also be mapped into the address space. Either way, there are *regions* of RAM we can use, and those that we cannot use.

Table 1 shows the memory mapping provided by the QEMU emulator that we’ve been using all along. The majority of the 32-bit address space is unassigned and cannot be used. We chose to have our kernel start at 1MB, or 0x0010\_0000, which is not accidental.

Table 1 - Address Space vs. Usable Physical Memory

|  |  |  |
| --- | --- | --- |
| Range | Length | Type |
| 0000\_0000 – 0009\_FBFF | 640KB | RAM, Usable |
| 0009\_FC00 – 0009\_FFFF | 1KB | ROM |
| 000A\_0000 – 000E\_FFFF | 320KB | Device I/O |
| 000F\_0000 – 000F\_FFFF | 64KB | ROM |
| 0010\_0000 – 07EE\_FFFF | 126MB | RAM, Usable |
| 0800\_0000 – FFFB\_FFFF | 4GB | Nothing, unused |
| FFFC0000 – FFFF\_FFFF | 256KB | ROM |

As a Multiboot compliant boot manager, Grub will compute the regions above and effectively create this table in memory. The Multiboot structures are defined in a standard header file. You can perform a Google search for “multiboot header file” or follow this link: <https://www.gnu.org/software/grub/manual/multiboot/html_node/multiboot_002eh.html>

You’ll need to download this .h file and include it into your project. A quick glance at its contents should show familiar keys, such as MULTIBOOT\_HEADER\_MAGIC which you’ve already included in your linker file. The file declares a “struct multiboot\_info," this has several interesting features we **could** choose to use, such as:

* Pointer to a string that identifies the “root file system device”
* Pointer to a command line for your kernel
* Pointer to a list of elf-file modules that your kernel should load
* Pointer to a list of BIOS-aware hard drives
* Video mode information selected by the boot manager
* ROM configuration table
* RAM configuration table

This last item is germane to our purposes right now. There are a pair of values that define the memory mapping information:

/\* Memory Mapping buffer \*/

multiboot\_uint32\_t mmap\_length;

multiboot\_uint32\_t mmap\_addr;

The “mmap\_addr” field is a 32-bit pointer to a “struct multiboot\_mmap\_entry” array. The mmap\_length parameter is the size of the array in bytes, not the length of the array. Each entry of the memory map array will describe a starting address (physical address), its length in bytes, and its type. This is exactly the information we need right now!

### Multiboot Information Exchange

The Multiboot protocol state that after the boot manager is completed, it will store the pointer to the multiboot structure in the %eax register, and the magic number in the %ebx register, and then it will jump into your first instruction. In our case, this is in “boot.S”, and its our \_start function.

#### Save the Registers, Save the World!

The first step is to open up your “boot.S”, and add two line of code to push the registers’ current value onto the call stack for your kernel main:

\_start:

# grub put multiboot info table and magic in %ebx and %eax respectively

movl $stack\_top, %esp

**push %eax**

**push %ebx**

Next, modify your kernel.c file, and “#include “multiboot.h” (that you downloaded above), and then modify your kernel main:

void kmain(multiboot\_info\_t \*multiboot\_ptr, uint32\_t multiboot\_magic)

That’s it! Your kernel’s main function now has access to this magic multiboot structure that was deposited in memory by Grub. Because the struct written by Grub is the same, byte-for-byte, as the struct used for reading, you can simply point at the same memory and, voila! you have access the configuration information written out by Grub.

The tricky-bit is what to do with this information. For now, we are interested in the memory that is usable.

### Free Memory

struct multiboot\_mmap\_entry

{

multiboot\_uint32\_t size;

multiboot\_uint64\_t addr;

multiboot\_uint64\_t len;

multiboot\_uint32\_t type;

} \_\_attribute\_\_((packed));

typedef struct multiboot\_mmap\_entry multiboot\_memory\_map\_t;

The multiboot information points to an array of memory maps. These are described by the “struct multiboot\_mmap\_entry,” which is typedef’d to be “multiboot\_memory\_map\_t”.

Since the multiboot header gives a pointer to the starting address, and the length (in bytes), you can work up some code that will walk the array and be able to build a picture of how memory is actually organized.

multiboot\_memory\_map\_t \*map = (multiboot\_memory\_map\_t \*) mbheader->mmap\_addr;

unsigned map\_length = mbheader->mmap\_length / sizeof(multiboot\_memory\_map\_t);

for (i = 0; i < map\_length; i++) {  
 kprintf(“%x - %x : %d\n”, map[i].addr, map[i].len, map[i].type);

}

The type field includes has five well known values:

* Type 1 – memory is available (its RAM)
* Type 2 – memory is marked “reserved” – cannot be used by your program
* Type 3 – memory is reserved for the ACPI BIOS
* Type 4 – memory is used by non-volatile memory (e.g. nv storage in apple)
* Type 5 – memory is marked as bad by the system BIOS

Technically, there is a sixth type – an entry that’s not listed is also assumed not present or not usable. For example, the device memory address space from 000A\_0000 – 000E\_FFFF should not be used for program storage, and won’t be listed in the memory map information.

There are two very important pieces of information we need from the memory map:

* What is the highest address of any *usable* memory?
* What memory addresses are reserved and what can be allocated to a program?

#### Highest Usable Memory

In this step, write a function that will walk the multiboot memory address information, and compute the largest address that is usable and return it. For example, if your QEMU is configured according to the default with 128MB available, your function should return 07EE\_FFFF, which corresponds to 128MB. Be careful, there is another map after this, way up at the end of memory, but is not usable, so don’t include it!

#### Bitmaps or Integer Arrays

Looking ahead, we know that memory is always allocated in 4KB pages. If the machine you were using had all 4GB of memory available, in the worst case, you would have , which is still 1 million entries. With smaller memories, you would need fewer pages. For example, with 128MB of memory, we would only have bytes of memory, and pages of memory.

Our kernel will need to keep track of what pages of memory are available to be used, and which pages have already been allocated, or are reserved by the kernel or hardware. There are two choices: a free bitmap, or an array.

**Bitmap**

The bitmap approach creates an array of 32-bit integers, where a single bit indicates if a page has been allocated or not. The number of bits must be greater than or equal to the number of pages. For example, with pages, we would need bits. If we use 32-bit integers, then we would need an array of integers.

Given a page number *n*, we can find which integer bit holds the allocation information:

Word(n) = n / 32

Bit(n) = n % 32

**Integer Array**

Instead of the bitmap approach, we could also use an array of integers, either 8-bit, 16-bit, or 32-bit, for each page. Then we can easily lookup the page availability by simply accessing the array based on page number, but then we also waste a lot of memory.

For example, if the bitmap needed entries, with each entry requiring 4-bytes (32-bit), then the whole map needs 4KB. Using 8-bit bytes, we would need bytes, which is, unsurprisingly, 8 times the size.

Whichever method you choose to build, as long as it works is fine for this project.

So, if we assume that when a bit is 1 the page is allocated, and 0 is when its available, we can write four functions:

* One to tell if the page is allocated or not
* One to mark the page as allocated
* One to mark the page as free
* One to find the first free page.

Finally, we need to initialize the memory arrays. Write a function, “memory\_init()” that will read the multiboot information, create the array (you can use kmalloc() for this), its probably best to mark all of the memory as “allocated,” so use “kmemset(array, 0xff, size)” to initialize all the bits to 1 (or whatever other method you are using). Then, walk the multiboot memory map and mark the pages of memory that are both usable according to multiboot AND greater than the memory address used by you kernel. Sample diagnostic information from my kernel is shown in Figure 1. To determine where the kernel begins and ends we need to get some information from the linker.

Return to your linker.ld file, and add some labels for where the kernel starts and ends. At the beginning:

/\* Begin putting sections at 1 MiB, a conventional place for kernels to be

loaded at by the bootloader. \*/

. = 1M;

**\_kernel\_start = .;**

And, again, at the end:

**\_kernel\_end = .;**

**\_kernel\_size = (\_kernel\_end - \_kernel\_start);**

/\* The compiler may produce other sections, put them in the proper place in

in this file, if you'd like to include them in the final kernel. \*/

}

Now, in the kernel, you can determine where the kernel starts and ends (according to the linker).

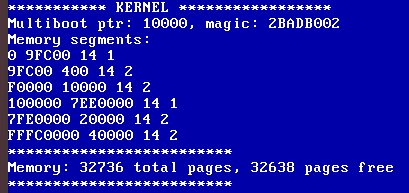


Figure 1

### Paging Memory

In the previous step, you created a view of the machine based on the idea of memory being divided into 4KB pages. In these next steps, you will use that to configure the MMU to create virtual memory pages that can be attached to physical pages.

In the Intel architecture, this is done by configuring the *paging* system. As you may recall, memory is divided into 4KB pages.[[1]](#footnote-1) Given a 32-bit memory address, we can divide the address into two parts:

|  |  |  |
| --- | --- | --- |
| bits: | 31 …. 12 | 11 … 0 |
| item: | page number | offset in page |

Consequently, given a 32-bit address, there can be at most addresses, or about 4GB. However, if we divide the memory into byte pages, then there are pages. Since the page-table configuration takes up memory itself, it is desirable to keep the size to a minimum. The Intel engineers came up with an interesting solution: a two-level page table.

### Page Directory Table

The page directory table has 1024 entries. If an entry is present, it contains the memory address of where the page table is located. When the MMU attempts to resolve a virtual address into a physical address, it takes the upper 10-bits of the address and looks in the corresponding row to see if the table is present or not. If its not, the MMU generates a page fault. If it is present, the MMU reads the address from the directory and reads the Page Table (see next section) to further resolve the address.

|  |  |
| --- | --- |
| Directory Row | Address Range |
| 0 | 0x0000\_0000 – 0x |
| 1 |  |
| 2 | 0x0080\_0000 – 0x00BF\_FFFF |
|  |  |
| 1022 | 0xFF80\_0000 – 0xFFBF\_FFFF |
| 1023 | 0xFFC0\_0000 – 0xFFFF\_FFFF |

This corresponds exactly to the upper 10-bits of an address. Given a 32-bit address, A, the CPU can find the index into the table by:

Index(A) = A >> 22

**Page Directory Entry**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Bits | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 31..24 | Page Address – 20 bits | | | | | | | |
| 23..16 |
| 15..8 |  | | | | Unused = 0 | | | 0 |
| 7..0 | S | 0 | A | D | W | U | R | P |

**Page Address<31:12>** – Page Table Entry Page Number

**S** – Page Size

* 1 – 4MB page size (requires PSE)
* 0 – 4KB page size

**A** – Accessed

* 1 – Set by hardware when page is accessed
* 0 – Clear by software, page has not been accessed

**D** – Cache Disabled

* 1 – Caching is disabled
* 0 – Caching is enabled

**W** – Write-Through Caching

* 1 – Write-Through caching
* 0 – Write-Back caching

**U** – User / Supervisory

* 1 – All pages in this block may be accessed by all (user and supervisor)
* 0 – All pages in this block may only be accessed by supervisor

**R** – Read / Write

* 1 – Page is Read/Write
* 0 – Page is Read only

**P** – Present

* 1 – Entry is present
* 0 – Entry is not present

Note that an entry that is all zeros will be “not present” and thus will not point to anything.

#### A Note About Page Addresses

When the directory entry is pointing at a valid page table, the 20 most significant bits of the **physical address** of the page table are stored here. This means that each page table entry **must be aligned** to a 4KB page boundary. In practice, this will not be difficult to achieve as we’ll see later.

### Page Table

The page table is the second-level of the paging system. Like the page directory, the page table includes 1024 entries. This time, the rows are indexed by the middle 10-bits of the memory address. Recall that the most significant 10-bits are used to index the page directory, so the next 10-bits are used by the CPU as an index to this table. The bottom 12-bits are used as the offset within a page, so are never stored (don’t worry, we’ll work through some examples).

In the previous narrative, we could describe exactly what addresses map to what rows in the page directory, but we cannot do that for the page table, because we don’t know the upper 10-bits. However, for the sake of a simple example, suppose we have the page table that is pointed to by page directory row #2. Row 2 works with addresses: 0x0080\_0000 through 0x00BF\_FFFF. So, this page table has 1024 rows:

|  |  |
| --- | --- |
| Table Row | Address Range |
| 0 | 0x0080\_0000 – 0x |
| 1 |  |
| 2 | 0x0080\_2000 – 0x0080\_2FFF |
|  |  |
| 1022 | 0x00BF\_E000 – 0x00BF\_EFFF |
| 1023 | 0x00BF\_E000 – 0x00BF\_FFFF |

Again, note the pattern, that the middle 10-bits are changing with each row, but upper 10-bits are determined by the page-directory, and the lowest 12-bits aren’t even stored. To get page table index for an address, *A*:

Index(A) = (A >> 12) & 0x3FF;

The logic here is that the we shift off the lowest 12-bits, leaving the upper 20-bits (of a 32-bit address). Then, we want to get only the least-significant 10-bits, which is 0x3FF in hexadecimal.

**Page Table Entry**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Bits | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 31..24 | Page Address – 20 bits | | | | | | | |
| 23..16 |
| 15..8 |  | | | | Unused = 0 | | | G |
| 7..0 | 0 | D | A | C | W | U | R | P |

**Page Address<31:12>** – Page Table Entry Page Number

**G** – Global

* 1 – Prevents the TLB from updating its address if CR3 changes
* 0 – Local to this process’s page table

**D** – Dirty

* 1 – Page was written to
* 0 – Page has not been written to

**A** – Accessed

* 1 – Set by hardware when page is accessed
* 0 – Clear by software, page has not been accessed

**C** – Cached

* 1 – Page is in cache
* 0 – Page is not in cache

**W** – Write-Through Caching

* 1 – Write-Through caching
* 0 – Write-Back caching

**U** – User / Supervisory

* 1 – All pages in this block may be accessed by all (user and supervisor)
* 0 – All pages in this block may only be accessed by supervisor

**R** – Read / Write

* 1 – Page is Read/Write
* 0 – Page is Read only

**P** – Present

* 1 – Entry is present
* 0 – Entry is not present

### An Example of Paging

A simple process uses three pages of memory, 4KB each. The following table shows the mapping of virtual memory address to physical memory address:

|  |  |  |
| --- | --- | --- |
| Page | Virtual | Physical |
| 0 | 0x1000\_0000 | 0x0103\_1000 |
| 1 | 0x1000\_1000 | 0x0020\_2000 |
| 2 | 0x7FFF\_0000 | 0x003C\_0000 |

To determine a minimum viable paging configuration, we first need to determine which rows in the directory need to be configured.

|  |  |  |
| --- | --- | --- |
| Page | Virtual | Directory Index |
| 0 | 0x1000\_0000 | 64 |
| 1 | 0x1000\_1000 | 64 |
| 2 | 0x7FFF\_0000 | 511 |

First, we need a page for the directory entry, suppose it is at 0x0000\_1000. Then, we need to get two pages of memory, one to hold the page table for row 64, and another to hold the page table for row 511. Lets suppose that we get page 0x0001\_0000 for row 64, and 0x0002\_0000 for row 511. Then the page directory will be:

Page Directory at Mem[0x0000\_1000]:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Row | Page Address <31:12> | Unused | S | 0 | A | D | W | U | R | P | Hex Word |
| 0 | 0000 0000 0000 0000 0000 | 0000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0000\_0000 |
| 1 | 0000 0000 0000 0000 0000 | 0000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0000\_0000 |
|  | | | | | | | | | | | |
| 64 | 0000 0000 0000 0001 0000 | 0000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0001\_0001 |
|  | | | | | | | | | | | |
| 511 | 0000 0000 0000 0010 0000 | 0000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0002\_0001 |
|  | | | | | | | | | | | |
| 1023 | 0000 0000 0000 0000 0000 | 0000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0000\_0000 |

Since there are two pages for the first page table, we can compute the page table address (middle 10 bits):

|  |  |  |  |
| --- | --- | --- | --- |
| Page | Virtual | Directory Index |  |
| 0 | 0x1000\_0000 | 0 | 0x0103\_1000 |
| 1 | 0x1000\_1000 | 1 | 0x0020\_2000 |

Page Table at Mem[0x0001\_0000]:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Row | Page Address <31:12> | Unused | G | 0 | D | A | C | W | U | R | P | Hex Word |
| 0 | 0000 0001 0000 0011 0001 | 000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0103\_1001 |
| 1 | 0000 0000 0010 0000 0010 | 000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0020\_2001 |
|  | | | | | | | | | | | | |

And, finally, the third page table, for the larger memory address:

|  |  |  |  |
| --- | --- | --- | --- |
| Page | Virtual | Directory Index | Physical Page |
| 2 | 0x7FFF\_0000 | 1008 | 0x003C\_0000 |

Page Table at Mem[0x0002\_0000]:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Row | Page Address <31:12> | Unused | G | 0 | D | A | C | W | U | R | P | Hex Word |
| 0 | 0000 0000 0000 0000 0000 | 000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0000\_0000 |
|  | | | | | | | | | | | | |
| 1008 | 0000 0011 1100 0000 0000 | 000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 003C\_0001 |
|  | | | | | | | | | | | | |

So, if we create these structures in the memory of the processor and then issue an instruction to load the address, the CPU’s MMU will use this to rewrite virtual addresses to physical addresses.

Suppose we try to load a value from memory address 0x1000\_103F.

The CPU will first find the page directory entry:

Index(0x1000\_103F) = 0x1000\_103F >> 22 = 64

So, the CPU will check the page directory, row 64, which it finds is:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Row | Page Address <31:12> | Unused | S | 0 | A | D | W | U | R | P | Hex Word |
| 64 | 0000 0000 0000 0001 0000 | 0000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0001\_0001 |

So, the entry I present (and readable, etc), so it now loads the page table, which it finds from the page address. The CPU will load the page at: 0000 0000 0000 0001 0000 0000 0000 0000, or 0x0001\_0000, which we said was the page table for this entry.

The CPU will then compute the index in this page table for the given address:

Index(0x1000\_103F) = (0x1000\_103F >> 12) & 0x3FF = 1

Next, the CPU will read the entry for row 1, which we configured:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Row | Page Address <31:12> | Unused | G | 0 | D | A | C | W | U | R | P | Hex Word |
| 1 | 0000 0000 0010 0000 0010 | 000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0020\_2001 |

The entry is present (and readable), so the MMU will, *finally*, convert this to a physical address. It will take the least 12-bits from the virtual address and the page address to construct a 32-bit adddress:

0000 0000 0010 0000 0010 0000 0011 1111 = 0x0020\_203F

### The MMU and the TLB

If the CPU really had to read three times to read one word from memory the system’s performance would be terrible. Instead, the MMU hardware maintains a cache of these page directory and page tables called the Translation Look-Aside Buffer (TLB). The CPU will manage this cache without any intervention on your behalf. The MMU will also work with the different instruction and data caches and the cacheability information stored in the tables to keep the caches and memories in sync.

So, while this isn’t something we have to worry about, just know that the performance of the system depends on the ability of the TLB to hold these structures in memory as much as possible. The good news is that the TLB is managed automatically and does not depend on any intervention on your part.

### Mapping a Page

Create a file, “memory.c”, and start by referencing the linker symbols created previously.

extern uint32\_t \_kernel\_size;

extern uint32\_t \_kernel\_start;

extern uint32\_t \_kernel\_end;

Create the related “memory.h” and define the page directory and entry data types. Normally, we would create these as a struct with a bitfield. This is one of those times were we may want to access the individual bits **and** access the 32-bit entry an integer. So, in my implementation, I started off with:

typedef union {

uint32\_t word\_value; // the 32-bit word unioned with…

struct { // the 32-bit bitfield containing:  
 unsigned address:20; // 20-bit page address

unsigned :3; // reserved

..

} page\_dirent\_t;

You’ll need one, similarly, for a page table entry as well. Then, it will help to create the “array” that will represent the page directory and page table, also in memory.h:

typedef struct {

page\_dirent\_t entry[1024];

} page\_directory\_t;

typedef struct {

page\_entry\_t entry[1024];

} page\_table\_t;

#### Discussion

Is this the only way to organize your page tables? No, it is definitely not. In fact, there are some good arguments to be made *against* using the union in the individual structs. One reason against this is that the correctness depends on the compiler’s optimizations. As long as the compiler *packs* all of the bitfields correctly *and* recognizes that the union and the structure members take the same space, in total, then this will work. One other argument *against* this that computers with different endianness can create problems, but the counter argument is that this is a structure that is specific to an Intel CPU, so portability is not an issue.

Instead, you could move the union into the arrays as follows.

typedef struct {

union {

uint32\_t intvals[1024];

page\_dirent\_t entry[1024];

uint8\_ bytevals[4096];

} page\_directory\_t;

Either way, we need to have structures that represent a whole page directory and whole page table, each with 1024 entries of their respective types.

#### Create the Page Directory

There are several things that give clues to how you should implement this. First, the *extern* for this implies that its storage is defined elsewhere, in this case, the linker. The *page\_directory\_t* is a data type that can be defined as a *bitmap*:

extern page\_directory\_t \*\_kernel\_page\_directory;

Create a set of functions to extract row indices from a virtual address. The first extracts the upper 10-bits of a virtual address as an integer, and this is used to index into the directory table. The second extracts the middle 10-bits, and the result is used to index into the page table.

unsigned int page\_directory\_for\_virtual(uint32\_t virtual);

unsigned int page\_table\_for\_virtual(uint32\_t virtual);

Initially, there will be an empty page directory that resides in memory at a location set by the linker (see the \_kernel\_page\_directory pointer above). We’ll add this definition to the linker later. The memory will be a 4096/4KB blob that holds the 1024 entries of the page directory, and initially all of its members will be zeroed out, which indicates that the row is “not present.”

We need a mechanism to add entries to the page directory that correspond to the memory that the kernel is actually, currently using. Later, as the kernel needs more memory for itself, it can add entries to the directory as well. To do this, we need to create a function that will fill in a “minimum viable” entry in the page directory on demand:

/\*\*

\* Given a page directory, allocate the second-level page table, and mark

\* set up its permissions. Set the top-level directory to point to that

\* new page. This does \*not\* set the entry in the page table,   
 \* @see page\_create\_map for that.

\* @param directory - the page table to update, can kernel's or users's

\* @param virtual - the virtual address that is being set up.

\* @returns address of the table that was allocated

\*\*/

static void page\_create\_dirent(page\_directory\_t \*directory, uint32\_t virtual)

Given a pointer to a page directory structure and a virtual 32-bit address, the function will do the following things:

1. Use page\_directory\_for\_virtual() to compute the index into the given page directory.
2. Look at the index in the table to determine if that row is present or not. Since the memory is contiguous, knowing where the 4KB block starts will allow us to treat the block as an array of page\_directory\_t.
3. If the row is present, then there is nothing further to do, just return.
4. If the row is not present:
   1. allocate a 4KB page that is aligned to a 4KB boundary – use the “find a free page and mark it as used” that you wrote in the previous section. This 4KB page will become the new page table for this range of memory controlled by the row of the page directory,
   2. zero out this new page of memory – setting all entries of the new table to zero
   3. store the upper 20-bits of this new page’s memory address in the address field of the page directory entry.
   4. mark the page “present” by setting a 1 in the P field.
   5. The remaining fields should all be zeros

#### Create the Page Table Entry

We were able to use the previous function to initialize an entry in the page directory, but it was created with all of its members zeroed out. In this step, we will map a virtual 32-bit address to a physical 32-bit address. Both of these addresses should really have their lowest 12-bits set to zeros.

Create another function that will create a virtual to physical mapping:

/\*\*

\* Map a virtual address to a physical address

\*\*/

static void page\_create\_map(page\_table\_t \*table, uint32\_t virtual, uint32\_t physical)

The page\_create\_map() function will, given a pointer to a page table, a virtual, and a physical address, update the page table to establish the mapping. This function must do the following:

1. Determine the 10-bit table address using page\_table\_for\_virtual
2. Verify that the address is not already mapped – if it is, throw an exception of some kind (again, more on this later)
3. Update the entry in the table for this address:
   1. Set the 20-bit address field to be he upper 20-bits of the physical address
   2. Set the P field to indicate that the page is present.
   3. All other fields should be zero for now.

#### Convenience is Key – Create a Function to Do Both

Finally, it would be handy to have a function that can just do whatever is required:

/\*

\* add an entry to map the physical and virtual address in the   
 \* indicated page table

\*/

static void page\_set\_map(page\_directory\_t \*directory,   
 uint32\_t virtual, uint32\_t physical)

This function will simply:

1. Call page\_create\_dirent() to ensure that the page directory is present
2. Use page\_directory\_for\_virtual()to find the row in the page directory for the virtual address
3. Now that you know that table is there (see #1), get the 20-bit address
4. Shift that 20-bit address left by 12-bits to make it a 32-bit physical address
5. Typecast that to be a pointer to a page\_table\_t.
6. Finally, call page\_create\_map with page\_table\_t pointer, and the virtual and physical addresses to update the table.

This should seem like a pretty simple bit of code because you’ve already built the complex functions. In fact, comparing my code to the text of the above list, the list is probably 3x the size of my code.

#### Initialize Paging

At this point, you should have the necessary functions to map a virtual page to a physical page. We still need to work out where the physical memory pages will be allocated from, and rest assured, we shall do that shortly. But, we have one final step to complete before we move on to that.

We need a paging initialization function that can be called from the kernel’s main():

/\*\*

\* Initialize the page-directory for the kernel itself

\*/

void page\_init( )

The point of this function is to loop over all of the kernel’s physical memory addresses and create mapping to the same physical address:

kprintf("Adding page mapping for %x to %x\n", start, end);

for (addr= start; addr < end; addr+= 4096) {

page\_set\_map(kpage\_dir, addr, addr);

}

Remember the linker symbols: \_kernel\_size, \_kernel\_start, and \_kernel\_end? Here’s where you’ll need them. So, this is where the linker put all of your kernel code and data. Its even where the linker put your heap. But, the linker has no idea where you put your *GDT* and *IDT*. I kept crashing the machine because of this. So, you can fix this the easy way or the hard way. The easy way is to start mapping virtual pages at address 0 until the kernel end. The more surgically precise way is to recall where you put the IDT and GDT and add a mapping for those two tables. Its up to you.

#### Enabling Page Mode

Create a function “page\_enable()” to turn on paging. There are two steps to follow here:

1. Load the 32-bit physical address of the page directory in control register CR3
2. Toggle bit 1 (paging) and only bit 1 of control register CR0

You’ve seen enough assembly language intrinsics to solve this yourself.

Your page\_init() function can call your page\_enable() function at the end. When we are ready to test this, what should happen? Nothing. Nothing at all. All you’ve done up to this point is create a virtual to physical mapping to all of the kernel’s memory. If you’ve done this right, the kernel should still have access to its own memory. If you’ve done this wrong, the machine will just crash. It may seem like a bit of a disappointment, but we’ll be exploiting paging in another section.

# Deliverables and Demos

Arrange a time for us to meet, and show be prepared to show me the following:

1. As usual, I want to see your code.
2. Demonstrate how you completed the task of connecting the mulitboot system information to your kernel and then pulled that information into something useful
3. Demonstrate your “bitmap” ADT (and any tests you have that show its working)
4. Demonstrate how many pages of memory are usable vs. unusable, how many are used by the kernel, and how many are free. Show me the code that computes that. Also, show me how that code keeps track of that information
5. Show me how your code will find the next free page of memory
6. Show me how your code will allocate the next free page of memory
7. Show me how you created and maintain the page-map for the kernel
8. Show me something you are really proud of in this project – each of you, and show me place where you both worked together to solve a problem.

Points: \_\_\_\_\_\_\_\_\_ / 50

1. This isn’t strictly true any more, Intel supports much larger pages, but we’ll stick with 4KB. [↑](#footnote-ref-1)