CS 310 Paging Lab
Spring 2020

February 18, 2020

1 Mapping Memory to your Linux Process

Linux and Mac OS provide a function called `mmap` that allows a process to request a block of memory of a given size to be mapped to a particular virtual address. You can get complete documentation by typing `man mmap` at the terminal or in Google.

In this exercise, you will be writing a program that runs in Linux (or Mac OS) that uses `mmap` to request a block of virtual memory. The `mmap` function prototype is:

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

- `addr` is the virtual address that we are requesting to be the base address of the memory region.
- `length` is the size of the requested block in bytes.
- `prot` is protection flags. It tells the OS how this block of memory may be used. We can enable read, write, and execute operations on our memory block. In this exercise, we will set `prot = PROT_READ | PROT_WRITE`.
- `flags` is a list of options that we can pass to the OS. In this exerceise, we will set `flags = MAP_PRIVATE | MAP_ANONYMOUS`. This tells the OS that the memory region is private (can only be read by this process) and it is anonymous, meaning it is not backed by a file.
- `fd` and `offset` will not be used in this exercise. Set them to 0.

What you should do: Write a C program that:

1. Calls `mmap` with the appropriate options to map a region of virtual memory at some address that you choose.
2. Use `memset` to set all the bytes in your memory region to `0x00` or `0xFF` or whatever you want. See what happens if you try to call `memset` with a size that is bigger than the size you passed to `mmap`.
3. Print out the address that you mapped (the return value of `memset`).

2 Setting up Paging on the Bare Metal

In this exercise, we are going to set up paging on an x86 virtual machine. The coding for this exercise will not be done in Linux. It will be done in operating system mode, similar to homework 2. As we discussed in class, the page table is organized as a sparse two-level tree. The root node is the `CR3` register which points to the page directory. The page directory consists of 1024 32-bit pointers, each of which points to a different page table. Each page table has 1024 32-bit pointers, each of which points to a different physical address.

The Easy Way: Copy Somebody Else's Code Go to the x86 Bare Metal Examples page (linked in the course website under Coding Resources) and clone the repo into your Linux VM. `cd` into the directory and type `make`. This will compile all the example programs. The one that we are interested in is called `paging.XXX`. The git repo comes with an assembly source file, and when you compile it, it will generate a file called `paging.img`. You can run it in Qemu by typing: `qemu-system-i386 -hda paging.img`

Once you have the basic program running, modify it to change the virtual address of your mapped page. You can make Qemu print out the mapped pages by using the monitor function. First, start Qemu using:

```
 $ qemu-system-i386 -hda paging.img -monitor stdio
```

This will start the emulator, and it will also give you a monitor that you can type commands into. In the monitor, type `info mem` to print out mapped memory:

```
(qemu) info mem
0000000000000000-0000000000400000 0000000000400000 -rw
```

The Hard Way: In C Your job in this lab is to set up the page table. To do this, you need to take the following steps:

1. Allocate a 4 kilobyte block of memory for the page directory. This block of memory needs to be aligned on a 4k boundary, meaning that the least significant 12 bits should be zero. You can do this using a gloabal variable in the following way:

   ```
   struct page_directory_entry page_directory[1024] __attribute__((aligned(4096)));
   ```

2. Allocate a 4 kilobyte block of memory (also global variable) for a page table (also needs to be 4k-aligned).
3. Initialization of the page directory and page table (use a `for()` loop in your `main` function):
   (a) Zero out every element of the page directory. You could use `memset` for this.
   (b) Set the `present` and `rw` bits to 1 for page directory element 0.
   (c) Point the page directory element 0 to the page table your allocated by setting the frame field.
   (d) Set up identity mapping in the page table by setting the frame field in the page table entries equal to the index of the entry. For example, `page_table[0].frame = 0 page_table[1].frame = 1`, and so on. Do this for every element of the page table. Also set the `present` and `rw` bits to 1 for each element of the page table.
4. Load the CR3 register with the address of the page directory. This can be done in C in the following way:

   ```
       // Put the address of page_directory[] into CR3
       asm("mov %0,%%cr3"
           :
           : "r"(page_directory)
           :);
   ```

5. Enable paging by setting bits 0 and 31 in the `CR0` register:

   ```
       asm("mov %cr0, %eax\n"
           "or $0x80000001,%eax\n"
           "mov %eax,%cr0");
   ```

For details, see Chapter 5 of the Intel 80386 Programmer's Reference Manual (linked under Programming Resources on the course website).

3 x86 Data Structures Reference

Format of a Page Table Entry Each page table entry is a 32-bit (4-byte) data structure that contains a bunch of information about a page in memory. The page table consists of 1024 page table entries.

The page frame address is the 4k-aligned physical start address of the page. The `AVAIL` bits are unused by hardware and available for software to use. The same format is used for Page Directory Entries.

`D`	Dirty
`US`	Supervisor access only when set to 1
`RW`	Writable when set to 1. Otherwise read-only.
`P`	Present in memory. If this bit is 0, it indicates that the page is not mapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
`Page Frame Address`																				`AVAIL`			`0`	`0`	`D`	`A`	`0`	`0`	`US`	`RW`	`P`

C Data Structures to Implement Page Table and Page Directory Entries

```
struct page_directory_entry
{
   uint32_t present       : 1;   // Page present in memory
   uint32_t rw            : 1;   // Read-only if clear, R/W if set
   uint32_t user          : 1;   // Supervisor only if clear
   uint32_t writethru     : 1;   // Cache this directory as write-thru only
   uint32_t cachedisabled : 1;   // Disable cache on this page table?
   uint32_t accessed      : 1;   // Supervisor level only if clear
   uint32_t pagesize      : 1;   // Has the page been accessed since last refresh?
   uint32_t ignored       : 2;   // Has the page been written to since last refresh?
   uint32_t os_specific   : 3;   // Amalgamation of unused and reserved bits
   uint32_t frame         : 20;  // Frame address (shifted right 12 bits)
};


struct page
{
   uint32_t present    : 1;   // Page present in memory
   uint32_t rw         : 1;   // Read-only if clear, readwrite if set
   uint32_t user       : 1;   // Supervisor level only if clear
   uint32_t accessed   : 1;   // Has the page been accessed since last refresh?
   uint32_t dirty      : 1;   // Has the page been written to since last refresh?
   uint32_t unused     : 7;   // Amalgamation of unused and reserved bits
   uint32_t frame      : 20;  // Frame address (shifted right 12 bits)
};
```