1 Introduction

This assignment has the following goals:

1. To configure the MMU that supports paging in a real processor
2. To experiment with demand paging in a real system
3. To explore how paging is used to optimize OS system calls

This is an *individual* assignment.

You are *encouraged* to help (and seek help from) your peers. However, you are not allowed to share code, and your final assignment must be your own individual work.

This assignment requires you to add several new system calls which will modify the configuration of the processor's MMU.

2 Assigned Reading

Before starting this assignment, you should read the following material. It will better help you understand how 64-bit paging support (which Intel calls IA-32E paging) works:

- The section on IA-32E paging in the chapter on the MMU in the Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3.
- The "Memory Addressing" chapter in the *Understanding the Linux Kernel* book.
- Text on bit-wise shift, bit-wise AND, and bit-wise OR operations in the C programming language. Example books include *A Book on C* and *C: A Reference Manual*.

3 Files

The files you need for this assignment are located in the git repository on Github. You can use the following command to download the files:

```
% git clone https://github.com/jtcriswell/csc256-mp3.git
```

The QEMU emulator and OpenWRT disk images are the same as those for MP2, so you should not need to re-download them. However, if you need to download new copies, you can do so using the getfiles.sh script.

4 Description

For this assignment, you are to implement two new system calls which directly read and manipulate the processor's page tables. One system call, named `readMMU`, will simply read information from the page tables. The other system called, `writeMMU`, will modify page table entries. You must also implement two helper system calls that allocate and free pages. These system calls will return physical addresses so that you can use them to add page table pages to the page table as necessary.

The exact behavior of the system calls is as follows:

1. long readMMU (unsigned long vaddr, unsigned long * pml4e, unsigned long * pdpte, unsigned long * pde, unsigned long * pte);
   Store the value of the page table entries for virtual address *vaddr* into the memory pointed to by *pml4e*, *pdpte*, *pde*, and *pte*. Your code should walk the page table and fill in the values that are present in each level of the page table e.g., the first level entry should be stored in the memory pointed to by *pml4e*, the second level entry should be stored into the memory pointed to by *pdpte*, etc. If an entry does not exist because that level of the page table is missing, a 0 should be written into the memory location. The system call should return 0 on success and -1 on error.

2. long writeMMU (unsigned long vaddr, unsigned long pml4e, unsigned long pdpte, unsigned long pde, unsigned long pte);
   Modify the page table entries that map *vaddr* to the values specified by *pml4e*, *pdpte*, *pde*, and *pte*. The system call should return 0 on success and -1 on error.

3. long allocPage (void);
   Allocate a frame and return the physical address of the first byte of the frame. Your test programs can use this function to allocate additional pages to add to the page table. The system call should return the physical address of the first byte of the frame allocated on success and -1 on error.

4. long freePage (long physaddr);
   Free a frame that was allocated by allocPage(). This system call should be given the physical address of the first byte of the frame. It should return 0 on success and -1 on error. When an error occurs, the system call should set errno to EINVAL.

Each system call should have the system call number specified in Table 1.

5 Test Programs

You must write some user-level programs that use your new system calls to query information about the MMU page tables.

181	long readMMU()
182	long writeMMU()
183	long allocPage()
184	long freePage(long physaddr)

Table 1: System Call Numbers

Remember that the OpenWRT disk image does not have all the dynamic libraries that you will need. You must therefore compile your code with the `-static` option to generate statically linked binaries. For example,

```
gcc -static -o test test.c
```

... will generate a statically linked program named test.

You should write the following programs and answer the following questions in your README file:

1. Write a program that reads the page table entry for virtual address zero. Is virtual address zero readable? Is it writeable? Is it even present? Why do you think virtual address zero is mapped as it is?

2. Create an 8 KB file. Write a program that

   (a) Uses `mmap()` to map the file into memory with read/write permissions

   (b) Calls `readMMU()` to find the page table entries for the virtual address to which the file is mapped and prints these values to standard output

   (c) Reads the first four bytes of the memory to which the file is mapped and prints the value of these bytes to standard output

   (d) Calls `readMMU()` again to find the page table entries for the virtual address to which the file is mapped and prints these values to standard output

   Is there anything mapped at that virtual address aftet the first call to `readMMU()`? If so, what permissions does the memory have? Are the page table entries different after the second call to `readMMU()`? If so, why are they different?

3. Create two 8 KB files. Each file should have different contents in the first 4 KB of the file. Write a program that uses `mmap()` to map these two files into two different virtual addresses; call these virtual addresses A and B. This program should then use `readMMU()` and `writeMMU()` to map the physical page mapped to virtual address A to virtual address B. Your program should write into virtual address A and be able to see the same value written into virtual address B. It will help if the files are mapped with read and write access. Report whether you are able to make this work with your system calls.

4. Create a program that allocates a 4 KB memory buffer. Use `readMMU()` to read the page table entry for the buffer. After that, create a child process with `fork()` and have both the parent and child process report the result of using `readMMU()` to read the page table entries of the allocated buffer. Do the entries from the parent and child process have the same physical address? What are their page permissions? Why do you think they are configured the way they are?

6 Tips

To read page table entries, you will need to convert physical addresses into virtual addresses so that you can read the next page table layer. The kernel has a range of virtual memory that is mapped one-to-one with physical memory so that it can quickly find a virtual address for a given physical address (this range of memory is sometimes referred to as the *direct map*). The `__va()` and `__pa()` functions in arch/x86/include/asm/page.h can do this for you.

7 Administrative Policies

7.1 Turn-in

You are to electronically turn in the following via Blackboard:

1. A patch containing your changes to the Linux kernel source code. The patch should be named `<netid>-patch` where `<netid>` is replaced with your network ID name. Please use the `mkpatch.sh` script to create your patch.

2. The source code for the test programs your wrote for Section 5.

3. A README file, in text format, that contains a clear description of files you modified or added to the kernel source code to implement your system calls.

4. A PDF file named `answers.pdf` which contains your answers to the questions in Section 5.

You should submit your files as a single GNU zipped tar archive; when unpacked, it should generate a directory named `<netid>-mp3` that contains your files (where `<netid>` is replaced with your network ID name). Your files should all be located within the top-level `<netid>-mp3` directory.

If you are unfamiliar with the tar and gzip/gunzip programs, please consult their man pages.

7.2 Code Comments

As you will learn when you start reading operating system kernel code, *too many* programmers write too few comments. It helped them code faster at the expense of your time and the time of all future kernel programmers to come. Therefore, to help end this cycle of misery, you are required to transcend current industry norms and comment your code.

Any non-trivial function should have a comment before the function describing its purpose, the meaning of its inputs, the meaning of its outputs, and the interpretation of its return value. Even simple functions that return `true` and `false` may need explanation of what those two values mean. Additional comments should exist throughout the code as necessary to explain what the code is doing and why.

In short, if the graders have difficulty understanding your code, then your commenting does not suffice.

7.3 Code Format

In addition to comments, your code should also be formatted well. While there are numerous coding standards, the most important factor in coding standards is *consistency*. This means that:

1. Your code should use a consistent formatting style, including brace placement style, indentation style, and commenting style.
2. Your code should use the same formatting style as other code in the kernel. If the kernel puts the curly brace at the end of the line, put your curly braces at the end of the line. If the kernel indents using tabs, indent your code using tabs.

Additionally, the instructor requires that your code be line-wrapped at 80 columns.

7.4 Grading Scheme

- *20 points:* your readMMU function works
- *20 points:* your writeMMU function works
- *20 points:* the results you report for Section 5 are correct
- *20 points:* your code does not make use of existing kernel functions for reading and writing page table entries. Using existing kernel functions to flush the TLB or to read the `CR3` register is okay; you may reuse existing kernel code to implement the `allocPage()` and `freePage()` system calls.
- *10 points:* your system calls properly handle errors in pointer values because they correctly use `copy_from_user()`, `copy_to_user()`, and other related functions

- *5 points:* software engineering: your code is properly commented and formatted and is well-written
- *5 points:* you turn in all requested files in the requested format