1 Introduction

Read the entire document before starting. There are critical pieces of information and hints along the way.

In this project, you will be implementing a virtual memory system simulator. You have been given a simulator which missing some critical parts. You will be responsible for implementing these parts. Detailed instructions are in the files to guide you along the way. If you are having trouble, we **strongly suggest** that you take the time to read about the material from the book and class notes.

There are 10 problems in the files that you will complete. The files that you will be changing are the following:

- `page_splitting.h` - Break down a virtual address into its components.
- `paging.c` - Initialize any necessary bookkeeping and implement address translation.
- `page_fault.c` - Implement the page fault handler.
- `page_replacement.c` - Write frame eviction and the clock sweep algoritm.
- `stats.c` - Calculate the Average Access Time (AAT)

You will fill out the functions in these files, and then validate your output against the given outputs. If you are struggling with writing the code, then step back and review the concepts. Be sure to start early, ask Piazza questions, and visit us in office hours for extra help!

2 Page Splitting

In most modern operating systems, user programs access memory using virtual addresses The hardware and the operating system work together to turn the virtual address into a physical address, which can then be used to address into physical memory. The first step of this process is to translate the virtual address into two parts: The higher order bits for the VPN, and the lower bits for the page offset.

In `page_splitting.h`, complete the `vaddr_vpr` and `vaddr_offset` functions. These will be used to split a virtual address into its corresponding page number and page offset. You will need to use the parameters for the virtual memory system defined in `pagesim.h` (`PAGE_SIZE`, `MEM_SIZE`, etc.).

3 Memory Organization

The simulator simulates a system with 64KB of physical memory. Throughout the simulator, you can access physical memory through the global variable `uint8_t mem[]` (an array of bytes called "mem"). You have access to, and will manage, the entirety of physical memory.

The system has a 20-bit virtual address space and memory is divided into 4KB pages.

Like a real computer, your page tables and data structures live in physical memory too! Both the page table and the frame table fit in a single page in memory, and you'll be responsible for placing these structures into memory.

Note: Since user data and operating system structures (such as the frame table and page tables), coexist in the same physical memory, we must have some way to differentiate between the two, and keep user pages from taking over system pages.

Modern day operating systems often solve this problem by dividing physical memory up into a "kernal space" and a "user space", where kernal space typically lies below a certain address and user space above. For this

project, we'll take a simpler approach: Every frame has a "protected" bit, which we'll set to "1" for system frames and "0" for user frames.

4 Initialization

Before we can begin accessing pages, we will need to set up the frame table (sometimes known as a "reverse lookup table"). After that, for every process that starts, you'll need to give it a page table.

For simplicity, we always place the frame table in physical frame 0 (don't forget to mark this frame as "protected"). Since this frame table belongs in a frame, we want to make sure that we will **never evict the frame table**. To do this, we set a protected bit. During your page replacement, you will need to make sure that you never choose a protected frame as your victim.

Since processes can start and stop any time during your computer's lifetime, we must be a little more sophisticated in choosing which frames to place their page tables in. For now, we won't worry about the logistics of choosing a frame–just call the `free_frame` function you'll write later in `page_replacement.c`. (Do we ever want to evict the frame containing the page table while the process is running?)

Your task is to fill out the following functions in `paging.c`:

1. `system_init()`
2. `proc_init()`

Each function listed above has helpful comments in the file. You may add any global variables or helper functions you deem necessary.

5 Context Switches and the Page Table Base Register

As you know, every process has its own page table. When the processor needs to perform a page lookup, it must know which page table to look in. This is where the *page table base register* (PTBR) comes in.

In the simulator, you can access the page table base register through the global variable `pfn_t PTBR`.

Implement the `context_switch` function in `paging.c`. Your job is to update the PTBR to refer to the new process's page table. This function will be very simple.

Gonig forward, pay close attention to the type of the PTBR. The PTBR holds a physical frame number (PFN), not a virtual address. Think about why this must be.

Each frame contains `PAGE_SIZE` bytes of data, therefore to access the start of the i-th frame in memory, you can use `mem + (i * PAGE_SIZE)`.

6 Page Replacement

Recollect that when a CPU encounters an invalid VPN to PFN mapping in the page table, the OS allocates a new frame for the page by either finding an empty frame or evicting a page from a frame that is in use. In this section, you will be implementing an efficient page replacement algorithm.

Implement the function `page_fault()` in `page_fault.c`. A page fault occurs when the CPU attempts to translate a virtual address, but finds no valid entry in the page table for the VPN. To handle the page fault, you must find a frame to hold the page (call `free_frame()`, then update the page table and frame table to reference that frame.

At last, you will also need to implement the free frame selection algorithm in `page_replacement.c`.

To get the free frame, implement and call `select_victim_frame()`.

If necessary, you will need to evict a current page mapping. Implement this logic in `free_frame()`. You must update the mappings from VPN to PFN in the current process' page table as well as invalidate the mapping the evicted process' page table to resolve the page fault.

If the evicted page is dirty, we will need to swap it out and write its contents to disk. To do so, we provide a method called `swap_write()`, where you will pass in a pointer to the victim's pagetable entry, and a pointer to the frame in memory. Similarly, after you map a new frame to a faulting page, you should check if it has a swap entry assigned, and call `swap_read()` if so.

Remember again that if the protected bit is set, it should never be chosen as a victim frame.

7 Finishing a Process

If a process finishes, we don't want it to hold onto any of the frames that it was using. We should release any frames so that other processes can use them. Also: If the process is no longer executing, can we release the page table?

As part of cleaning up a process, you will need to also free any swap entries that have been mapped to pages.

You can use `swap_free()` to accomplish this. Implement the function `proc_cleanup()` in `paging.c`.

8 Computing AAT

In the final section of this project, you will be computing some statistics.

1. `writes` - The total number of accesses that were writes
2. `reads` - The total number of accesses that were reads
3. `accesses` - The total number of accesses to the memory system
4. `page_faults` - Accesses that resulted in a page fault
5. `writes_to_disk` - How many times you wrote to disk
6. `reads_from_disk` - How many times you read data from the disk
7. `aat` - The average access time of the memory system

We will give you some numbers that are necessary to caluclate the AAT:

1. `MEMORY_READ_TIME` - The time taken to access memory **SET BY SIMULATOR**
2. `DISK_PAGE_READ_TIME` - The time taken to read a page from the disk **SET BY SIMULATOR**
3. `DISK_PAGE_WRITE_TIME` - The time taken to write to disk **SET BY SIMULATOR**

You will need to implement the `compute_stats()` function in `stats.c`

9 How to Run / Debug Your Code

9.1 Environment

Your code will need to compile under Ubuntu 16.04 LTS. You can develop on whatever environment you prefer, so long as your code also works in Ubuntu 16.04 (which we will use to grade your projects). **Non-**

compiling solutions will receive a 0! Make sure your code compiles with no warnings. We recommend downloading VirtualBox and setting up a new virtual machine with Ubuntu 16.04 if you are on Mac OS or Windows. If you are having trouble with this, come to office hours.

9.2 Compiling and Running

We have provided a Makefile that will run gcc for you. To compile your code with no optimizations (which you should do while developing, it will make debugging easier), run

```
$ make
$./vm-sim -i traces/<trace>.trace
```

9.3 Debugging Tips

You can use valgrind and gdb (GNU Debugger) to help debug your project. Valgrind is a program that helps detect memory management bugs, and GDB is a command line interface that will allow you to set breakpoints, step through your code, see variable values, and identify segfaults. There are tons of online guides, click here (http://condor.depaul.edu/glancast/373class/docs/gdb.html) for one.

To use valgrind, run:

```
$ valgrind ./vm-sim -i traces/<trace>.trace
```

To start your program in gdb, run:

```
$ gdb ./vm-sim
```

Within gdb, you can run your program with the `run` command, see below for an example:

```
$ (gdb) r -i traces/<trace>.trace
```

Feel free to ask about valgrind or gdb and how to use it in office hours and on Piazza. Do not ask a TA or post on Piazza about a segfault without first running your program through both.

9.4 Verifying Your Solution

On execution, the simulator will output data read/write values. To check against our solutions, run

```
$ ./vm-sim -i traces/<trace>.trace > my_output.log
$ diff my_output.log outputs/<trace>.log
```

NOTE: To get full credit you must completely match the TA generated outputs for each trace.

10 How to Submit

Run `make submit` to automatically package your project for submission. Submit the resulting tar.gz zip on T-square.

Always re-download your assignment from T-Square after submitting to ensure that all necessary files were properly uploaded. If what we download does not work, you will get a 0 regardless of what is on your machine.

This project will be demoed. In order to receive full credit, you must sign up for a demo slot and complete the demo. We will announce when demo times are released.