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 is 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:

- `va_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 algorithm.
- `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 Address 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 `va_splitting.h`, complete the `vaddr_vpn` 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

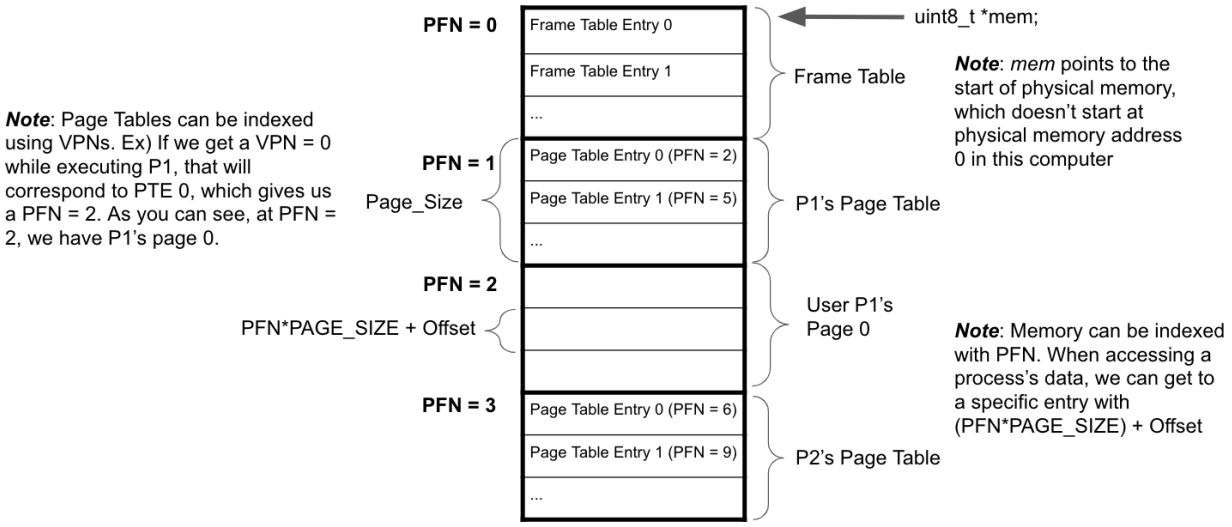

Figure 1: Organization of physical memory showing frames and frame entries.

The simulator simulates a system with 1MB 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 24-bit virtual address space and memory is divided into 16KB pages.

Like a real computer, your page tables and paging data structures live in physical memory too! Conveniently, both a page table and the frame table fit in a single page frame in memory and so youll want to dedicate some page frames for storing this data. You are responsible for placing and initializing these structures in memory.

Note: Since user data page frames and operating system page frames 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 replacing system pages.

For this project, we will take a simpler approach: Every page frame has a protected bit in the frame table, which well set to 1 for system frames and 0 for user frames. In other words, well set the protected bit to 1 for the frames holding paging system meta-data and 0 for the page frames holding user pages.

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. To set up the frame table, you need to initialize the pointer to the start of the frame table. Remember that this frame table belongs in a frame in memory - the first frame, so the pointer to the start of the frame table is just a frame table entry pointer. Don't forget to mark this first frame as "protected". 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?)

Furthermore, when we initialize a process, we want to initialize that process diskmap. A diskmap is an OS data structure that maps virtual page numbers to page-sized blocks in the memory image file we want to use to initialize the virtual address space. This is used by the page replacement code to obtain the disk location from which to initialize the page the first time it is referenced. This is how we arrange for an object program file (containing a memory image of the program) to be loaded into memory and run by the process. In our case, for simplicity, we are going to presume the disk maps point to files of all-zero pages which will be used to initialize the virtual address space. In your project, when you initialize a process, you can use the diskmap_init function to initialize the diskmap for the process.

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.

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)`.

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.

Going 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.

6 Reading and Writing Memory

The ability to allocate physical frames is useless if we cannot read or write to them. In this section, you will add functionality to the simulator to allow it to make read and write memory references on behalf of the simulated processes. The simulator will use pre-recorded lists of memory references captured (traced) from the execution of real processes to simulate the memory references in each process. Because processes operate on a virtual memory space, it is necessary to first translate a virtual address supplied by a process into its corresponding physical address, which then will be used access the location in physical memory. This is accomplished using the page table, which contains all of a process's mappings from virtual addresses to physical addresses.

As was said, when running a user process, all addresses from the CPU are virtual and must be translated. Do note that when the operating system is running (i.e. the CPU is in system mode), address translation is disabled and all memory addresses referenced by the CPU will be treated as physical addresses. This is why the operating system itself has no page table!

Implement the `mem_access` function in `paging.c`. You will need to use the passed-in virtual address to find the correct page table entry and the offset within the corresponding page. **HINT:** Use the virtual address splitting functions that you wrote earlier in the project.

Once you have identified the correct page table entry, you must use this to find the corresponding physical frame and its physical address in memory, and then perform the read or write at the proper location within the page. (Remember that the simulator's physical memory is represented by the `mem` array and its subscripts are the physical memory addresses).

Keep in mind that not all entries in a process's page table have necessarily been mapped. Entries not yet mapped are marked as invalid, and an attempt to access an invalid address should generate a page fault. You will write the `page_fault()` function in the next section, so for now just assume that it has successfully allocated a page for that address after it returns.

Make sure to mark the containing page as "dirty" in the process's page table on a write. These bits will be used later when deciding on what pages should be evicted first, and if an evicted page needs to be written to the disk to preserve its content.

7 Eviction and Replacement

Recall 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 a page fault and replacement mechanism.

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.)

Next, we will turn our attention to the eviction process in `page_replacement.c`.

If you ask the system for a free frame when all the frames are in use, the operating system must select an in-use frame and re-use it, "evicting" any existing page that was previously using the frame. Implement this logic in `free_frame()`. To resolve the page fault, you must do the following:

1. Update the mapping from VPN to PFN in the current process' page table.
2. Invalidate the evicted process' page table mapping.
3. Unmap the corresponding frame table entry.

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

If you page fault on an address and it does not have an associated swap entry yet, then you should use the process' diskmap to read in a page from disk. To help with this, we provide a `diskmap_read` function.

Swap space effectively extends the memory of your system. If physical memory is full, the operating system kicks some frames to the hard disk to accommodate others. When the "swapped" frames are needed again, they are restored from the disk into physical memory.

8 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`.

9 Better Victim Selection

In section 7, we relied on the `select_victim_frame()` function to tell us which frame to choose as our "victim".

We have provided you with a default, inefficient page replacement algorithm that randomly selects a page to be evicted. The simulator can run this replacement strategy out-of-the-box so that you can test the other parts of your code without having to write a page replacement algorithm. Run the simulator with `-rrandom` to use the random algorithm.

Of course, we can do better than random replacement. Implement the **Clock Sweep algorithm**. When implementing the Clock Sweep algorithm, every frame table entry has a reference bit which is set once the page has been accessed. When looking for a victim, the Clock Sweep algorithm will choose the first page that does not have its reference bit set to 1. If all of the frame table entries have their reference bit set then this will become FCFS. Look at section 8.3.5 in the textbook for more information on this.

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

Once you have implemented the Clock Sweep algorithm, you will be able to run the simulator with the `-rclocksweep` argument to use the algorithm as your page replacement strategy.

After implementing the clocksweep algorithm, implement the **first-in, first-out replacement algorithm**. Your FIFO algorithm should choose the least recently mapped frame table entry based on a timestamp, which represents the time that the entry was mapped. Each frame table entry contains a timestamp field, which you will update whenever you map the frame to a new virtual page. To get the current time of the simulator, you may use the provided `get_current_timestamp()` function.

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

Once you have implemented the FIFO algorithm, you will be able to run the simulator with the `-rfifo` argument to use the algorithm as your page replacement strategy.

Once you write your stats function in section 10, compare the performance of the three algorithms. What do you observe?

10 Computing AAT

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

- `writes` - The total number of accesses that were writes
- `reads` - The total number of accesses that were reads
- `accesses` - The total number of accesses to the memory system
- `page_faults` - Accesses that resulted in a page fault
- `writebacks` - How many times you wrote to disk
- `aat` - The average access time of the memory system

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

- `MEMORY_READ_TIME` - The time taken to access memory **SET BY SIMULATOR**
- `DISK_PAGE_READ_TIME` - The time taken to read a page from the disk **SET BY SIMULATOR**
- `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`

11 Simulator Process Diagram

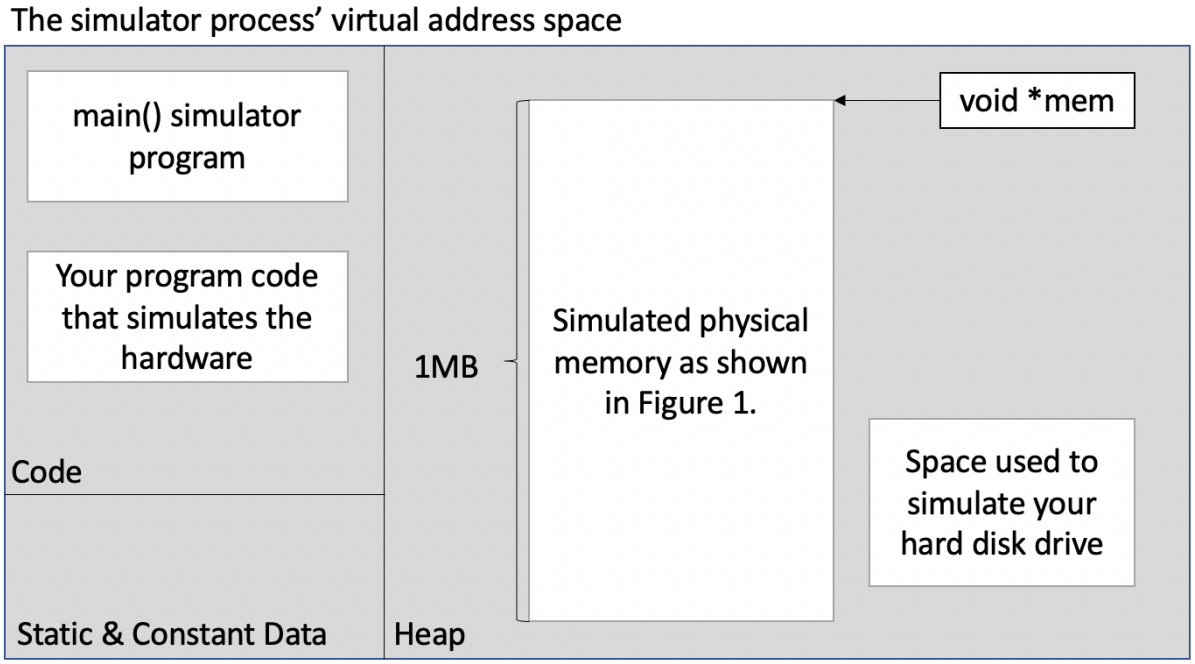

Figure 2: This diagram gives a general overview of how the simulator works.

12 How to Run / Debug Your Code

12.1 Environment

Your code will need to compile under Ubuntu 20.04 LTS. You can develop on whatever environment you prefer, so long as your code also works in Ubuntu 20.04 (which we will use to grade your projects). **Non-compiling solutions will receive a 0! Make sure your code compiles with no warnings.** We highly recommend using the setup available on Canvas. If you are having trouble with this, ask a TA about it in office hours.

12.2 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) and test with the "random" algorithm, run:

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

Once your Clock Sweep algorithm has been implemented, you can run the program with the `-rclocksweep` argument in order to test. For example, you should run:

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

Once your FIFO algorithm has been implemented, you can run the program with the `-rfifo` argument in order to test. For example, you should run:

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

We highly recommend starting with "`simple.trace`." This will allow you to test the core functionality of your virtual memory simulator without worrying about context switches or write backs, as this trace contains neither.

12.3 Corruption Checker

One challenge of working with any memory-management system is that your system can easily corrupt its own data structures if it misbehaves! Such corruption issues can easily hide until many cycles later, when they manifest as seemingly unrelated crashes later.

To help with detecting these issues, we've included a "corruption check" mode that aggressively verifies your data structures after every cycle. To use the corruption checker, run the simulator with the `-c` argument:

```
$./vm-sim -c -i traces/<trace>.trace -r<algorithm>
```

12.4 Debugging Tips

If your program is crashing or misbehaving, you can use GDB to locate the bug. 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 compile with debugging information, you must build the program with `make debug`:

```
$ make clean
$ make debug
```

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 -r<algorithm>
```

You may find it useful to set a breakpoint inside the main loop of the simulator to debug specific simulator commands in your implementation. You can do this either by finding the line number inside `pagesim.c` and breaking there:

```
$ (gdb) break pagesim.c:53 ! set breakpoint at call to system_init
$ (gdb) r -i traces/<trace>.trace -r<algorithm>
! (wait for breakpoint)
$ (gdb) s   ! step into the function call
```

or by using the actual function name being called from the main loop:

```
$ (gdb) break sim_cmd      ! set breakpoint at call to sim_cmd
$ (gdb) r -i traces/<trace>.trace -r<algorithm>
! (wait for breakpoint)
$ (gdb) s   ! step into the function call
```

Sometimes, you may want to examine a large area of memory. To do this in GDB, you can use the `x` command (short for examine). For example, to examine the first 24 bytes of the frame table, we could do the following:

```
$ (gdb) x/24xb frame_table
0x1004000aa: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1004000b2: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x1004000ba: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```

The format of this command is `x/nfu [memory location]`, where **n** is the number of items to print, **f** is a formatting identifier (for example, **x** for hexadecimal), and **u** is the specifier for the units you would like to print. **b** specifies units of 1 byte, **h** specifies 2 bytes, **w** specifies 4 bytes, and **g** specifies 8 bytes. So, our above command showed us 24 bytes of memory starting at `frame_table` in hexadecimal form.

If you use the corruption checker, you can set a breakpoint on `panic()` and use a backtrace to discover the context in which the panic occurred:

```
$ (gdb) break panic
$ (gdb) r -i traces/<trace>.trace -r<algorithm>
! (wait for GDB to stop at the breakpoint)
$ (gdb) backtrace
$ (gdb) frame N    ! where N is the frame number you want to examine
```

Feel free to ask about 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 GDB.

12.5 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 -r<algorithm> > my_output.log
$ diff my_output.log outputs/<trace>.log
```

The second half of the output file name includes the type of replacement algorithm that should be run when comparing the output. Ex. astar-random.log should be compared with the output from using random replacement algorithm ($-rrandom$) as shown below.

```
$ ./vm-sim -i traces/astar-random.trace -rrandom > my_output.log
$ diff my_output.log outputs/astar-random.log
```

You **MUST** implement the Clock Sweep algorithm in order to test against all the *-clocksweep.log output files.

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

13 How to Submit

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

Always re-download your assignment from Canvas 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.