

1 Introduction

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

In this assignment, 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 textbook and class notes.

There are 9 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.
- `stats.c` - Calculate the Average Access Time of the memory system (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_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

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 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 "kernel space" and a "user space", where kernel 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.

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.

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.

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 for reading and writing individual bytes to memory.

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.

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 page 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 address in memory, and then perform the read or write at the proper location within the page. (Remember that the simulator's memory is represented by the `mem` array).

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.

When performing a memory access to an address, you must also make sure to set the mark the containing frame as "referenced" in the appropriate frame table entry, as well as marking the containing page as "dirty" in the process's page table. 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()`. 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, you will need to swap it out and 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 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 the page has a swap entry assigned, and call `swap_read()` if so.

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 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. `aat` - The average access time of the memory system

We will give you some numbers that are necessary to calculate 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`

10 How to Run / Debug Your Code

10.1 Environment

Your code should compile on the ECE linux machines. You can develop on whatever environment you prefer, so long as your code also works in the ECE machines. **Non-compiling solutions will receive a 0!**

10.2 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
```

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 writebacks, as this trace contains neither.

10.3 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
```

10.4 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
```

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
! (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.

10.5 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 > my_output.log
$ diff my_output.log outputs/<trace>.log
```

Currently, we have released the expected outputs for `astar.trace` in `outputs/astar.log`. We will release a few more expected outputs closer to the deadline.

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

11 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.