Virtual Memory

Spring 2021 CSci 5103 Project 3

Final Submission Due Monday, April 12
Intermediate Submission Due Friday, April 2

1. Overview

In this project, you are asked to implement a simple, yet fully functional demand paged virtual memory. You will also learn the closely related topic of memory mapped files. Although virtual memory is normally implemented at the kernel level, it can also be implemented at the user level, which is a technique used by modern virtual machines. Thus, you will learn an advanced technique without having the headache of writing kernel-level code. The following figure gives an overview of the components:

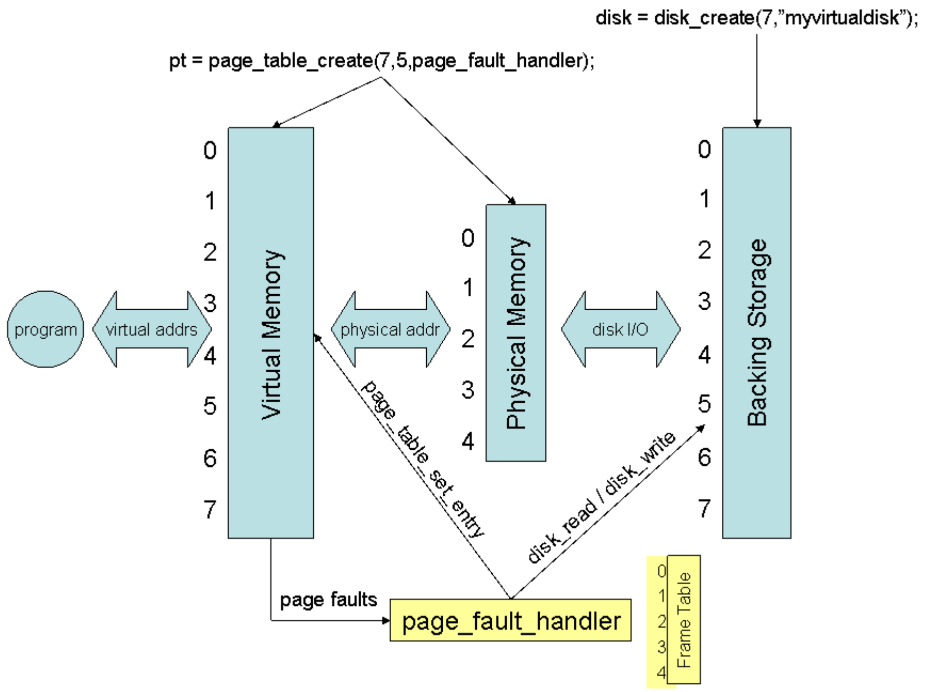


Fig. 1. Overview of sub-components

We will provide you with code that implements a virtual page table and a virtual disk. The virtual page table will create a small virtual and a small physical memory, along with the methods for updating the page table entries and protection bits. When an application uses the virtual memory, it can result in a page fault that calls a custom handler. Your job is to implement a page fault handler that traps page faults and triggers a series of actions, including updating the page table and moving data back and forth between disk and physical memory. You will implement multiple different page replacement algorithms to be used in your handler. After implementing the code, you will evaluate the performance of the different page replacement algorithms using a selection of simple benchmark programs across a range of memory sizes. You will write a short report that describes your implementation, explains the results, and analyzes the performance of each algorithm.

2. Getting Started

Download the base code and build it. The provided code uses system calls that are only available on Linux so you will not be able to build or run this project on other POSIX compliant operating systems.

The following source files are provided:

- *main.cpp*: This is the only file you should need to modify. You should implement your page fault hander and page replacement algorithms here. The provided code includes a main driver that will run the test program specified on the command line.
- *program.cpp/h*: A set of test programs for testing your virtual memory implementation. These functions are called from the main driver.
- *page_table.cpp/h*: Page table management functions implemented for you. You will want to use the functions provided in the header file when implementing your page fault handler.
- *disk.cpp/h*: Disk management functions implemented for you. You will want to use the functions provided in the header file when implementing your page fault handler.

2.1 Virtual Memory Emulation

Read *page_table.cpp/h* and *disk.cpp/h* to learn how the various virtual memory library calls should be used. You should also look through this code to get an understanding of how virtual memory is emulated in our user-space program. To accomplish this, `page_table_create()` creates a file that will act as physical memory. We then use `mmap` to create an address space in the program that maps to that file (refer to the man page for more details).

Memory-mapped files are used to provide a virtual memory address space (`virtmem`) and physical memory address space (`physmem`) to the physical memory file. The physical memory address space always maps directly to the physical memory file. The virtual memory address space will have indirect mappings from pages in the virtually memory address space to frames in the physical memory file. This is achieved using the `remap_file_pages` system call (refer to the man page for more details). This function allows us to create a non-sequential mapping of the memory-mapped address space to the underlying file. By using `remap_file_pages` with a page size granularity, we can create a virtual memory address space that indirectly maps pages to different frames in the physical memory file.

Access to memory-mapped files can also be protected using the `mprotect` system call (refer to the man page for more details). This will allow us to set read and write permissions on regions of the virtual memory address space. That way, for example, we can set up a non-resident page in the virtual address space to have no permissions so that if the user attempts to access that memory region it will result in a page fault. Page faults are caught in our user space library by setting up a signal handler for the segmentation fault signal (SIGSEGV). When user code attempts to use a portion of the virtual address space that it does not have permission to use, a signal will be sent by the operating system and caught by `internal_fault_hanlder()` within *page_fault.cpp*. This function will then call the page fault handlers you will write for this project.

2.2 Page Faults

Go through *main.cpp* and notice that the program simply creates a virtual disk and a page table and then attempts to run one of the three benchmark programs from *program.cpp/h* using the virtual memory.

Try running the sort test program with 4 pages and 2 frames using the rand page replacement algorithm:

```
% ./virtmem 4 2 rand sort
```

Since no mapping has been made between virtual and physical memory, a page fault happens immediately:

```
page fault on page #0
```

The program exits because the page fault handler hasn't been written yet. That is your job! As a starting point, if you run the program with an equal number of pages and frames, then you don't need the disk. You can simply map page N directly to frame N. Specifically, call the following in the page fault handler:

```
page_table_set_entry(pt, page, page, PROT_READ | PROT_WRITE);
```

With this naive page fault handler, all of the benchmark programs should be able to run, while causing some number of page faults, as long as the number of frames specified is greater than or equal to the number of pages.
Test your direct mapping handler out (assumes the rand handler is being used for direct mapping for now):

```
% ./virtmem 2 2 rand sort
```

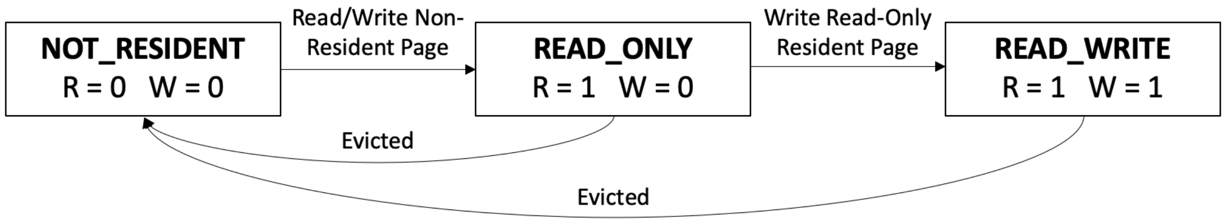
The test program should run and exit successfully. Congratulations! You have implemented your first fault handler. Of course, when there are fewer frames than pages, this naive scheme will not work. In that situation, you will need to keep recently used pages in the physical memory, place other pages on disk, and update the page table appropriately as pages are moved back and forth.

3. Page Fault Handling

The virtual page table is very similar to what we have discussed in class, except that it does not have a referenced or dirty bit for each page. The system supports a read bit (`PROT_READ`) and a write bit

(`PROT_WRITE`). When neither the read bit nor the write bit is set, the page should not be resident in physical memory.

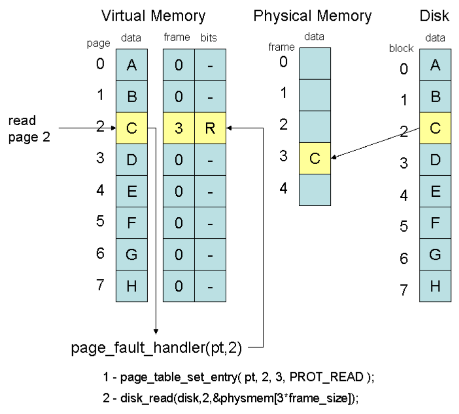
The following state machine should be used to handle page faults and page table management.



Before exiting the main function, you should print the number of page faults, disk reads, and disk writes over the course of the program. You can print this in whatever format you would like. For the number of page faults, you should only consider the transition from NOT_RESIDENT to READ_ONLY as a page fault. The transition from READ_ONLY to READ_WRITE is technically resulting in a page fault in our user-space program, but it used to emulate a dirty bit that would usually be handled by hardware and not require a page fault. For this reason, only count the number of page faults due to not resident pages.

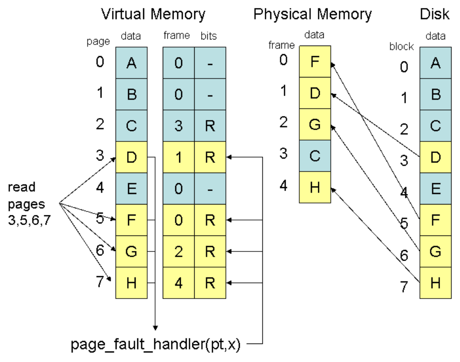
4. Example Operations

Let's work through a concrete example, starting with the figure on the right side. Suppose we begin with nothing in physical memory. If the application begins by trying to read page 2, this will result in a page fault. The page fault handler chooses a free frame, say frame 3. It then adjusts the page table to map page 2 to frame 3, with read permissions. Then, it loads page 2 from disk into frame 3. On the first page fault, you can assume the disk block corresponding to this page is appropriately zeroed out. When the page fault handler completes, the read operation is automatically re-attempted by the system and succeeds.

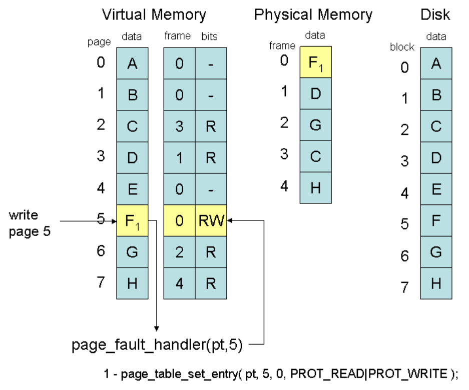


```
1 - page_table_set_entry( pt, 2, 3, PROT_READ );
2 - disk_read(disk,2,&physmem[3*frame_size]);
```

The application continues to perform read operations. Suppose that it reads pages 3, 5, 6, and 7. Each read operation results in a page fault, which triggers a memory loading as in the previous step. After this step physical memory is full.

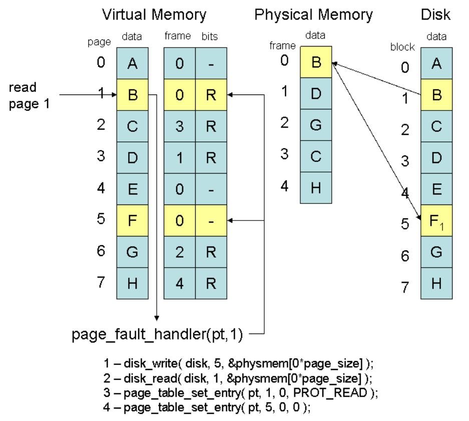


Now suppose that the application attempts to write to page 5. Because this page only has the PORT_READ bit set, a page fault will occur. The page fault handler checks page 5's current page bits and adds the PROT_WRITE bit. When the page fault handler returns, the write operation is automatically re-attempted by the system and succeeds. Page 5, frame 0 is modified.



```
1 - page_table_set_entry( pt, 5, 0, PROT_READ|PROT_WRITE );
```

Now suppose that the application reads page 1. Page 1 is not currently paged into physical memory. The page fault handler must decide which frame to evict. Suppose that it picks page 5, frame 0. Because page 5 has the PROT_WRITE bit set, it is dirty. The page fault handler writes page 5 back to the disk and reads page 1 to frame 0. Two entries in the page table are updated to reflect the new state.



```
1 – disk_write( disk, 5, &physmem[0*page_size] );
2 – disk_read( disk, 1, &physmem[0*page_size] );
3 – page_table_set_entry( pt, 1, 0, PROT_READ );
4 – page_table_set_entry( pt, 5, 0, 0 );
```

5. Requirements

Your program must be invoked as follows:

```
% ./virtmem npages nframes rand|fifo|custom scan|sort|focus
```

`npages` is the number of pages and `nframes` is the number of frames to create in the system. The third argument is the page replacement algorithm. You must implement `rand` (random replacement), `fifo` (first-in-first-out), and `custom`, which is an algorithm of your own design and should perform better than `rand` (meaning causing fewer disk reads and writes and/or fewer page faults in the common case). When you implement your own `custom` algorithm, try to make it simple and realistic. The last argument specifies which benchmark program to run: `scan`, `sort`, or `focus`.

Each test program accesses memory using a slightly different pattern. You are welcome (and encouraged) to implement additional benchmark programs with richer patterns. If you implement any new programs, indicate how to run them in the README. At a minimum, you should test your virtual memory implementation on each provided test case with varying numbers of pages and frames. More specifically, it is a good idea to test with more pages than frames. This will result in more page faults and evictions.

A complete and correct program will run each of the benchmark programs to completion with only the following outputs:

- A line of output from the test program indicating success
- A line of output from your implementation with the number of page faults, disk reads, and disk writes over the course of the program

You can add debug message during testing, but the final version should not have any extraneous output.

You will also turn in a concise lab report (report.pdf) that includes:

- In your own words, briefly explain the purpose of the experiments and the experimental setup. Be sure to clearly state on which machine(s) you ran the experiments, and what your command line arguments were. So that we can reproduce your work in case of any confusion.
- Describe the `custom` page replacement algorithm that you have implemented. Make sure to give enough details that someone else could reimplement your algorithm without your code.
- Measure and draw graphs of the number of page faults, disk reads, and disk writes for each program and each page replacement algorithm using 100 pages and varying numbers of frames between 1 and 100. Spend some time to polish your graphs such that they are nicely laid out, correctly labelled, and easy to read.
- Analyze your results and describe when one algorithm outperforms the others, and why.

6. Deliverables

You should submit all source code, the Makefile, README, report, and any other additional files necessary to run or describe your implementation.

All files should be submitted in a single tarball (.tar.gz). The following command can be used to generate the file:

```
$ tar -cvzf submission.tar.gz project_folder
```

7. Grading

Tentative grade breakdown:

- (+10) Intermediate submission
- (+50) Correct implementation of demand paging supporting arbitrary access patterns and varying sizes of virtual and physical memory
- (+30) A clearly written lab report which contains appropriate descriptions of the experiments, well-presented results, and reasonable analysis
- (+10) Good coding style, including error handling, formatting, and useful comments