Virtual Memory

Fall 2019 CSCi 5103 Project 2, Due at midnight Nov. 10

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:

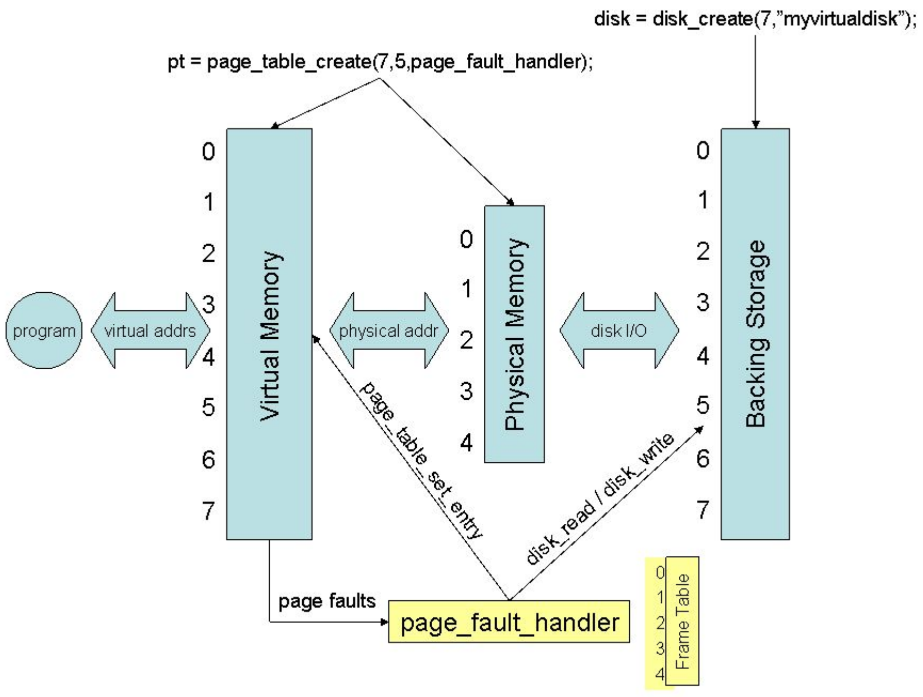


Fig. 1. Overview of sub-components

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 memory and a small physical memory, along with the methods for updating the page table entries and protection bits. When an application uses virtual memory, it will result in a page fault that calls a custom handler. Your job is to implement a page fault handler that handles page faults and takes a series of action, including updating the page table, and moving data back and forth between disk and physical memory. After implementing the code, you will evaluate the performance of 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. You are **FORBIDDEN** to hunt down any online code including in web sites and git repositories.

2. Getting Started

Download the base code and build it. Go through main.c and notice that the program simply creates a virtual disk and and a page table, and then attempts to run one of the three benchmark programs using the virtual memory. Read *page_table.c* and learn how various system calls should be used. Then run the base code:

```
% ./virtmem 100 10 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 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 this in the page fault handler:

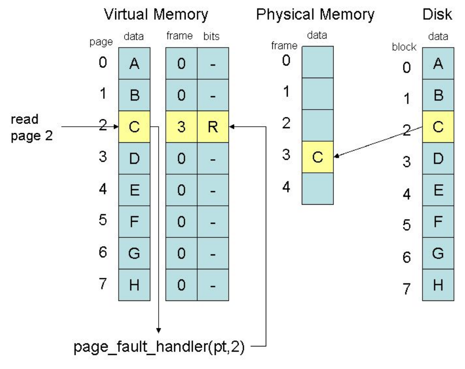
```
page_table_set_entry(pt,page,page,PROT_READ|PROT_WRITE);
```

With this naive page fault handler, all of the benchmark programs should be able to run, while causing some number of page faults. 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. Example Operations

The virtual page table is very similar to what we have discussed in class, except that it does not have a reference or dirty bit for each page. The system supports a read bit (`PROT_READ`), a write bit (`PROT_WRITE`), and an execute bit (`PROT_EXEC`), which is enough to make it work.

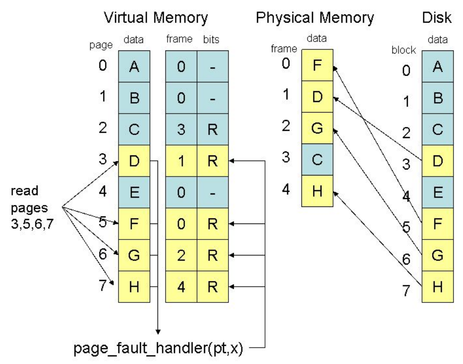
Let's work through a concrete example, starting with the figure on the right side. Suppose we begin with nothing in physical memory. If the application begins by trying to read page 2, this will result in a page fault. The page fault handler choose a free frame, say frame 3. It then adjusts the page table to map page 2 to frame 3, with read permissions. Then, it loads page 2 from disk into page 3. When the page fault handler



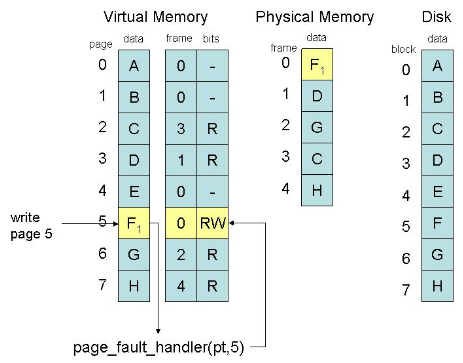
```
1 - page_table_set_entry( pt, 2, 3, PROT_READ );
2 - disk_read(disk,2,&physmem[3*frame_size]);
```

completes, the read operation is re-attempted, and succeeds.

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.

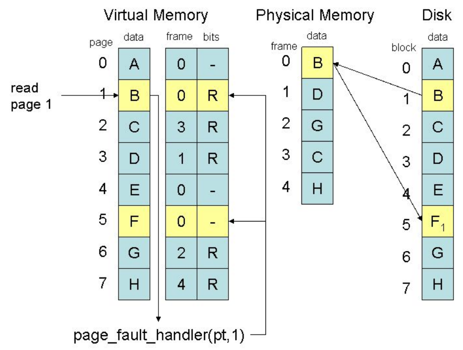


Now suppose that the application attempts to write to page 5. Because this page only has the `PORT_READ` bit set, a page fault will occur. The page fault handler checks page 5's current page bits and adds the `PROT_WRITE` bit. When the page fault handler returns, the application will continue. Page 5, frame 1 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 remove. 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 );
```

4. 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). 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 accesses memory using a slightly different pattern. You are welcome (and encouraged) to implement additional benchmark programs with richer patterns.

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

- A single line of output from scan, sort, or focus.
- A summary of 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 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) did you run the experiments, and what your command line arguments were. So that we could 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.
- Analysis of the results, such as under what conditions does one algorithm outperform the others, and why.

5. Project Groups

All students should work in groups of size 2 or 3. Only ONE submission is needed for each group. Include all group members' names in the document and the source code.

6. Deliverables

- A report
- Source code, including makefile, (optional) a script to run the tests

7. Grading (tentative)

- Correct implementation of demand paging supporting arbitrary access patterns and varying sizes of virtual and physical memory. (60%)
- A clearly-written lab report which contains appropriate descriptions of the experiments, well-presented results, and reasonable analysis. (30%)
- Good coding style, including error handling, formatting, and useful comments. (10%)