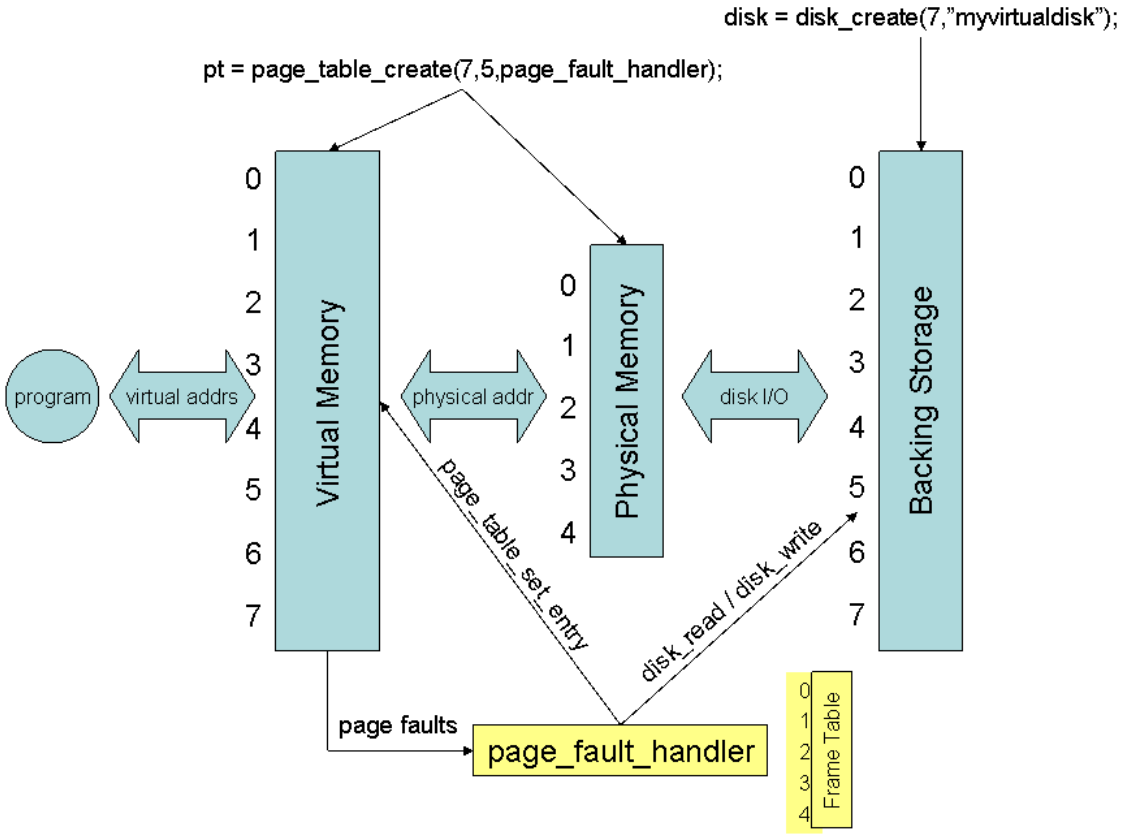
CS 537 - Introduction to Operating Systems

Project 3: Virtual Memory

Aman Rakesh Chadha
(9068354597)

1 Project Overview

In this project, you will build a simple but fully functional demand paged virtual memory. Although virtual memory is normally implemented in the operating system kernel, it can also be implemented at the user level. This is exactly the technique used by modern virtual machines, so you will be learning an advanced technique without having the headache of writing kernel-level code. The following figure gives an overview of the 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 physical memory, along with methods for updating the page table entries and protection bits. When an application uses the virtual memory, it will result in page faults that call a custom handler. Most of your job is to implement a page fault handler that will trap page faults and identify the correct course of action, which generally means updating the page table, and moving data back and forth between the disk and physical memory.

Once your system is working correctly, you will evaluate the performance of several page replacement algorithms on a selection of simple programs across a range of memory sizes. You will write a short lab report that explains the experiments, describes your results, and draws conclusions about the behavior of each algorithm.

2 Getting Started

Begin by downloading the source code and building it. Look through main.c and notice that the program simply creates a virtual disk and page table, and then attempts to run one of our three "programs" using the virtual memory. Because no mapping has been made between virtual and physical memory, a page fault happens immediately:

```
% ./virtmem 100 10 rand sort
page fault on page #0
```

The program exits because the page fault handler isn't written yet. That is your job!

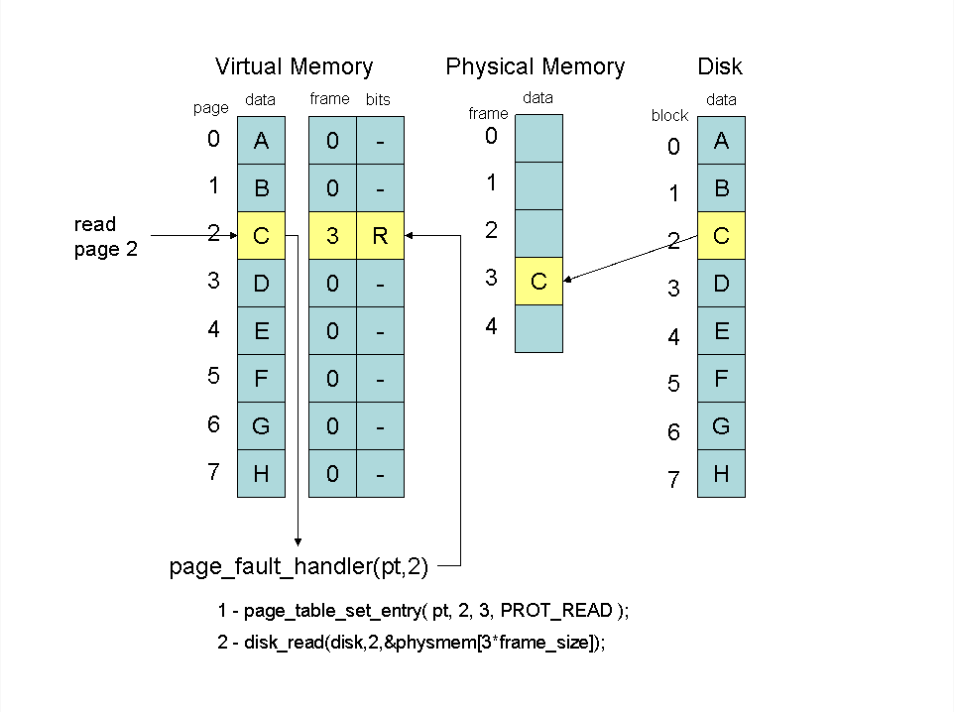
Try this as a getting started exercise. If you run the program with an equal number of pages and frames, then we don't actually need a disk. Instead, you can simply make page N map directly to frame N, and do nothing else. So, modify the page fault handler to do exactly that:

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

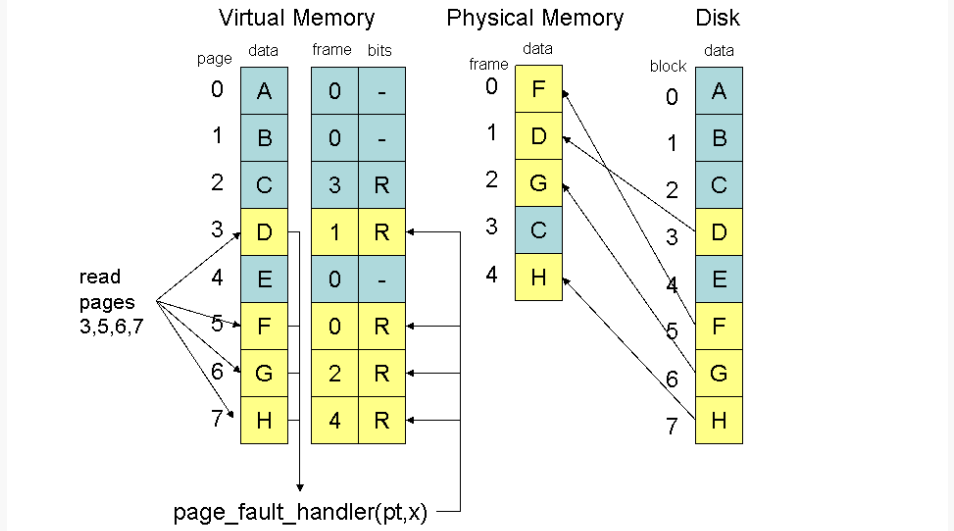
With that page fault handler, all of the example programs will run, cause a number of page faults, but then run to completion. Congratulations, you have written your first fault handler. Of course, when there are fewer frames than pages, then this simple scheme will not do. Instead, we must keep recently used pages in the physical memory, other pages on disk, and update the page table appropriately as they move back and forth.

3 Example Operation

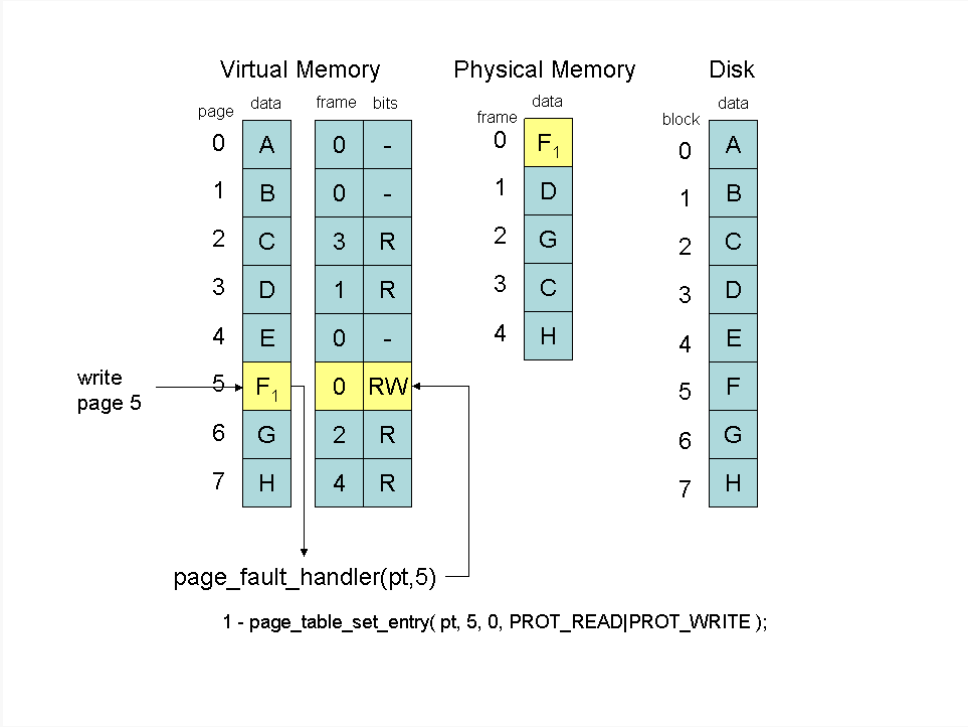
The virtual page table is very similar to what we have discussed in class, except that it does not have a reference or dirty bit for each page. The system supports a read bit (PROT_READ), a write bit (PROT_WRITE), and an execute bit (PROT_EXEC), which is enough to make it work.


Virtual Memory
Physical Memory
Disk
page
data
frame
bits
block
read
page 2
page_fault_handler(pt,2)
1 - page_table_set_entry(pt, 2, 3, PROT_READ);
2 - disk_read(disk,2,&physmem[3*frame_size]);

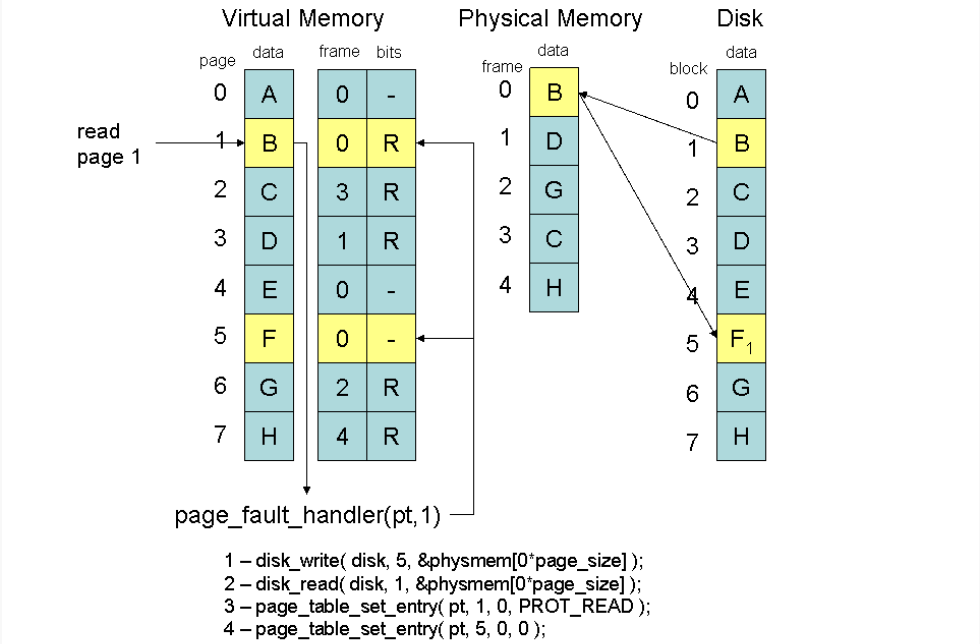
Step 1


Virtual Memory
Physical Memory
Disk
page
data
frame
bits
block
read
pages
3,5,6,7
page_fault_handler(pt,x)

Step 2


Virtual Memory
Physical Memory
Disk
page
data
frame
bits
block
write
page 5
page_fault_handler(pt,5)
1 - page_table_set_entry(pt, 5, 0, PROT_READ|PROT_WRITE);

Step 3


Virtual Memory
Physical Memory
Disk
page
data
frame
bits
block
read
page 1
page_fault_handler(pt,1)
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);

Step 4

Step 1. Let's work through an example, starting with the figure at the right. Suppose that 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 completes, the read operation is re-attempted, and succeeds.

Step 2: The application continues to run, reading various pages. Suppose that it reads pages 3, 5, 6, and 7, each of which result in a page fault, and must be loaded into memory as before. Now physical memory is fully in use.

Step 3. Now suppose that the application attempts to write to page 5. Because this page only has the R bit set, a page fault will result. The page fault handler looks at the current page bits, and upon seeing that it already has the PROT_READ bit set, adds the PROT_WRITE bit. The page fault handler returns, and the application can continue. Page 5, frame 1 is modified.

Step 4. 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 at random. Because page 5 has the PROT_WRITE bit set, we know that it is dirty. So, the page fault handler writes page 5 back to the disk, and reads page 1 in its place. Two entries in the page table are updated to reflect the new situation.

4 Essential Requirements

Your program must be invoked as follows:

```
./virtmem npages nframes rand|fifo|2fifo|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), **2fifo** (second-chance fifo) and custom, an algorithm of your own invention. The final argument specifies which built-in program to run: **scan**, **sort**, or **focus**. Each accesses memory using a slightly different pattern.

You may only modify the file main.c. Your job is to implement four page replacement algorithms. rand, fifo and 2fifo should be implemented as discussed in class. For 2fifo, you should use 25% of memory (rounded down) for the second chance list.

Your fourth algorithm targets flash drives. They have the property that if you write to the device too many times, it wears out. Thus, a paging system for flash drives should avoid writing, at the expense of a few more reads. Your task is to invent a paging mechanism with no more than **10%** (on average across all three programs) more reads than second-chance fifo that reduces the number of writes as much as possible.

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

- The 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 may certainly add some printfs while testing and debugging your program, but the final version should not have any extraneous output.