Operating Systems Lab

Dept of CSE, LNMIIT, Jaipur

15 March **2021**

Lab Session 07

This lab session is sourced from University of Notre Dame student assignment.

Virtual Memory

The goals of this project are:

- To demonstrate mastery of the virtual memory concept.
- To learn the code mechanics of operating system fault handlers.
- To develop skills in quantitative system evaluation.
- To gain experience in writing short technical reports.

Work Planned for this week

This lab session is to help the students learn about Linux Signals and Call back functions. As they develop the solutions for the simpler cases of the problem, students should learn about Signals.

This week the students are only required to implement FIFO policy for page replacement.

Students can find discussions on Linux signals (also called software interrupts) at
http://www.cs.kent.edu/~ruttan/sysprog/lectures/signals.html

an alternate source for information on signals that looks good is:
https://www.tutorialspoint.com/unix/unix-signals-traps.htm

Lab Session 09

We will leave the actual implementation and demonstration of other page replacement policies for Lab Session 09. In Lab session 09, you are to implement either LRU or NRU.

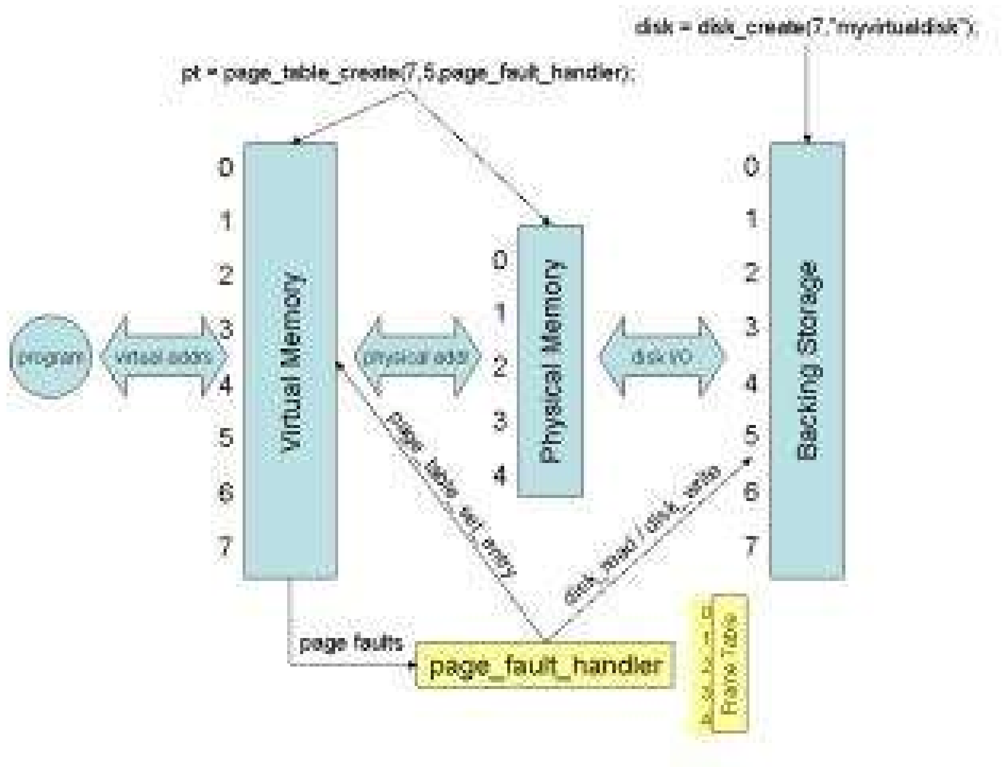
Optional: Work out a way to test optimal policy by perhaps running the program twice. The first run will be used to generate a page reference sequence. This run may be organised by assuming very large memory and using protection bit settings that allows us to capture all page accesses.

Once the page reference sequence is available, it can be used to guide optimal replacement policy with the realistic memory size parameters.

What follows below is the problem description imported from another university.

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:



You are provided 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 behaviour of each algorithm.

Getting Started

Begin by building the source code. Look through `main.c` and notice that the program simply creates a virtual disk and page table, and then attempts to run one of 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 is not 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 do not 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:

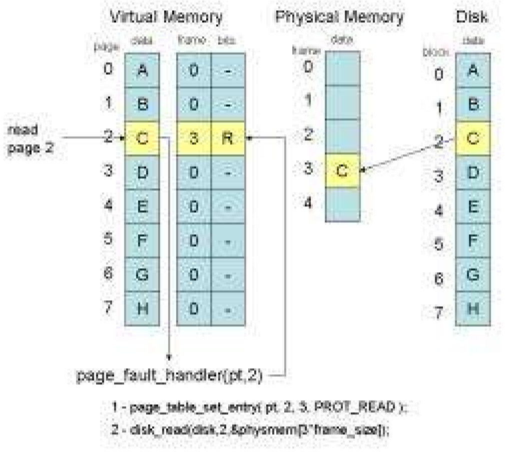
```
page_table_set_entry(pt,page,page,PROT_READ|PROT_WRITE);
```

With that page fault handler, all 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.

Example Operation

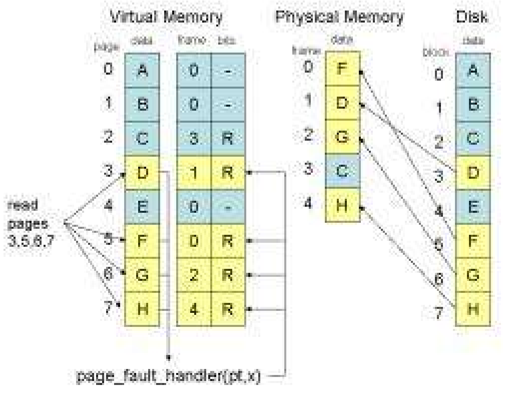
The virtual page table is very similar to what we have discussed in class, except that it does not have a reference or dirty bit for each page. The system supports a read bit (PROT_READ), a write bit (PROT_WRITE), and an execute bit (PROT_EXEC), which is enough to make it work.

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 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 page 3. When the page fault handler completes, the read operation is re-attempted, and succeeds.

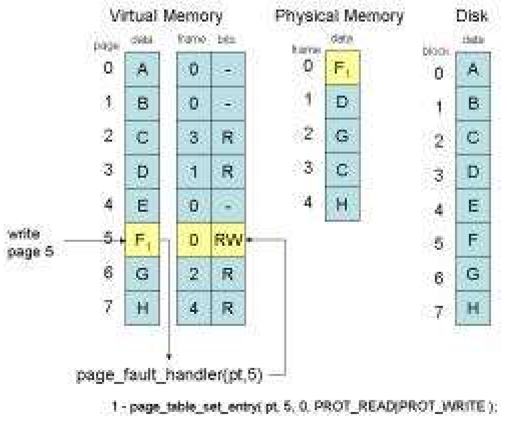


```
1 - page_table_set_entry( pt, 2, 3, PROT_READ );
2 - disk_read(disk,2,&physmem[3*frame_size]);
```

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

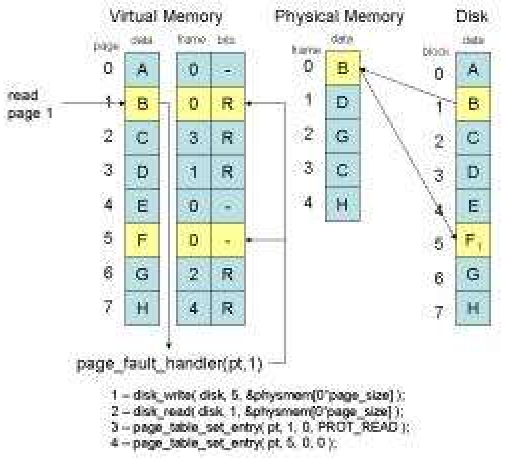


Now suppose that the application attempts to write to page 5. Because this page only has the R bit set, a page fault will result. The page fault handler looks at the current page bits, and upon seeing that it already has the PROT_READ bit set, adds the PROT_WRITE bit. The page fault handler returns,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



```
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 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.



```
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 );
```

Essential 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**, 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 three page-replacement algorithms. Besides the random and FIFO techniques, you should invent a third algorithm, **custom** that does a better job than **rand** or **fifo**. (Better means results in fewer disk reads and writes.).

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 `printf()` while testing and debugging your program, but the final version should not have any extraneous output.

You will also turn in a lab report that discusses the experimental comparison of the three replacement strategies for the different application scenarios and that has the following elements:

- In your own words, briefly explain the purpose of the experiments and the experimental setup, i.e., what do you expect to see from the comparison of the different strategies and how you perform this comparison. Be sure to clearly state on which machine you ran the experiments, and exactly what your command line arguments were, so that we can reproduce your work in case of any confusion.
- Very carefully describe the custom page replacement algorithm that you have invented. Make sure to give enough detail that someone else could reproduce your algorithm, even without your code.
- Measure and graph the number of page faults, disk reads, and disk writes for each program and each page replacement algorithm using 100 pages and a varying number of frames between 2 and 100. Spend some time to make sure that your graphs are nicely laid out, correctly labelled, and easy to read. Do not use coloured backgrounds. You may connect data points with straight lines, but not with splines or curves.
- Explain the nature of the results. If one algorithm performs better than another under certain conditions, then point that out, explain the conditions, and explain *why* it performs better.
- Explain how the code implements demand paging.

Hints

- To see if a given page is written on disk or in memory, you can use `page_table_get_entry(page, &frame, &bits)`. If the given page has non-zero permission bits, then it resides in the indicated frame number. If the permission bits are zero, then it is not in memory, and the frame number is irrelevant.
- Create a frame table that keeps track of the state of each frame. That will make it easy to find a free frame or a frame for replacement.
- Use `lrand48()` to generate random numbers for your page fault handler (do not use `srand` or `rand`!).

Test your program for correct implementation of demand paging with any arbitrary access pattern and amount of virtual and physical memory.