Programming Assignment 4: Virtual Memory Paging Strategies

CSCI 3753 - Operating Systems
University of Colorado at Boulder
Spring 2012

By Andy Sayler and Junho Ahn and Richard Han
Adopted from assignment by Dr. Alva Couch, Tufts University[1]

Due Date: Wednesday, April 18th, 2012 11:55pm

1 Assignment Introduction

All modern operating systems use virtual memory and paging in order to effectively utilize the computer's memory hierarchy. Paging is an effective means of providing memory space protection to processes, of enabling the system to utilize secondary storage systems fro addition memory space, and of avoiding the need to allocate memory sequentially.

In CS2400, we studied how virtual memory systems are structured, and how the MMU converts virtual memory address to physical memory addressees by means of a page table and a translation lookaside buffer (TLB). When a page has a valid mapping from VM address to physical address, we say the page is "swapped in". When no valid mapping is available, the page is either invalid (a segmentation fault), or more likely, "swapped out". When the MMU determines that memory request requires access to a page that is current swapped out, it calls the operating system's page-fault handler. This handler must swap-in the necessary page, possible evicting another page to secondary memory in the process, and then retry the offending memory access and hand control back to the MMU.

As you might imagine, how the OS chooses which page to evict when it has reached the limit of available physical memory can have a major effect on the performance of the programs running under the OS. In this assignment, we will look at various strategies for managing the system page table and controlling when pages are paged in and when they are paged out.

2 Your Task

The goal of this assignment is to implement a paging strategy that maximizes the performance of program memory access across a set of predefined processes. You will accomplish this task by using a paging simulator that has already been created for you. Your job is to write the paging strategy that the simulator utilizes. You will be graded on the throughput of your paging strategy (the ratio of time spent doing useful work vs time spent paging).

2.1 The Simulator Environment

The simulator has been provided for you. You have access to the course code if you wish to review it (`simulator.c` and `simulator.h`), but you should not need to modify this code for the sake of this assignment. You will be graded using the stock simulator, so any enhancements to the simulator program made with the intention of improving your performance will be for nigh.

The simulator runs a random collection of processes utilizing a limited number of shared physical pages. Each process has a fixed number of virtual pages that it might try to access. For the purpose of this simulation, all memory access is due to the need to load program code. Thus, the simulated program counter (PC) for each process dictates which memory location that process currently requires access to, and thus which virtual process page must be swapped-in for the process to successfully continue.

The constants discussed above that define the simulated environment are available in the `simulator.h` file. For the purposes of grading your assignment, the default constants will be used:

- 20 virtual pages per process (`MAXPROCPAGES`)
- 100 physical pages total (`PHYSICALPAGES`)
- 20 simultaneous processes competing for pages (`MAXPROCESSES`)
- 128 memory unit page size (`PAGESIZE`)
- 100 tick delay to swap page in our out (`PAGEWAIT`)

As you can see, you are working in a very resource constrained environment. You will have to deal with attempts to access up to 400 virtual pages (20 processes times 20 virtual pages per process), but may only have at most 100 physical pages swapped on at any given time. Thus, at best, three-quarters of the potentially required pages will be swapped-out at any given moment.

In addition, swapping a page in or out is an expensive operation, requiring 100 ticks to complete. A tick is the minimum time measurement unit in the simulator. Each instruction or step in the simulated programs requires 1 tick to complete. Thus, in the worst case where every instruction is a page miss (requiring a swap-in), you will spend 100 ticks of paging overhead for every 1 tick of useful work. This leads to a overhead to useful work ration of 100 to 1: very, very, poor performance.

2.2 The Simulator Interface

The simulator exports three functions through which you will interact with it. The first function is called `pageit`. This is the core paging function. It is equivalent to the page-fault handler in your Operating System. The simulator calls `pageit` anytime something interesting happens (memory access, page fault, process completion, etc). It passes the function a page map for each process, as well as the current value of the program counter for each process. See `simulator.h` for details. You will implement your paging strategy in the body of this function.

The simulator also exports a function called `pagein` and a function called `pageout`. These functions request that a specific page for a specific project be swapped-in or swapped-out, respectively. You will use these function to control the allocation of virtual and physical pages when writing your paging strategy. Each of these functions return `1` if they succeed in *starting* a paging operation, or if a paging operation is *already in progress*. 100 ticks after starting a paging operation, the operation will complete, and the page maps passed to `pageit` will reflect the new state of the simulator. These function return `0` if the paging request can not be processed (due to exceeding the limit of physical pages or because another paging operation is currently in process on the requested page) or if the request is invalid (paging operation requests non-existent page, etc). You can use these function to test whether or not a page is available to be paged in. If `1` is returned, the page is available and paging has begun. If `0` is returned, the page is not available.

2.3 The Simulated Programs

The simulator populate sits 20 processes by randomly selecting processes from a collection of 5 simulated "programs".

3 Some Implementation Ideas

4 What You Must Provide

When you submit your assignment, you must provide the following as a single archive file:

- A copy of all your code
- A makefile that builds any necessary code
- A README explaining how to build and run your code

5 What's Included

We provide some code to help get you started. Feel free to use it as a jumping off point (appropriately cited).

1. **Makefile** A GNU Make makefile to build all the code listed here.
2. **README** As the title so eloquently instructs: read it.

6 Extra Credit

7 Grading

40% of you grade will be based on the submission you provide. To received full credit your submission must:

- Meet all requirements elicited in this document
- Code must build with "-Wall" and "-Wextra" enabled, producing no errors or warnings.
- Code must adhere to good coding practices.

The other 60% of your grade will be determined via your grading interview where you will be expected to explain your results and answer questions regarding them and any concepts related to this assignment.

8 Obtaining Code

The starting code for this assignment is available on the Moodle and on github. If you would like practice using a version control system, consider forking the code from github. Using the github code is not a requirement, but it will help to insure that you stay up to date with any updates or changes to the supplied codebase. It is also good practice for the kind of development one might expect to do in a professional environment. And since your github code can be easily shared, it can be a good way to show off your coding skills to potential employers and other programmers.

Github code may be forked from the project page here:
`https://github.com/asayler/CU-CS3753-2012-PA4`.

9 Resources

Refer to your textbook and class notes on the Moodle for an overview of OS paging policies and implementations.

If you require a good C language reference, consult K&R[2].

The Internet[3] is also a good resource for finding information related to solving this assignment.

The most recent version of the assignment from which this assignment was adopted is available at [1].

You may wish to consult the man pages for the following items, as they will be useful and/or required to complete this assignment. Note that the first argument to the "man" command is the chapter, insuring that you access the appropriate version of each man page. See `man man` for more information.

- `man 1 make`

References

[1] Couch, Alva. *Comp111 - A5*. Tufts University: Fall 2011. `http://www.cs.tufts.edu/comp/111/assignments/a5.html`.

[2] Kernighan, Brian and Dennis, Ritchie. *The C Programming Language*. Second Edition: 2009. Prentice Hall: New Jersey.

[3] Stevens, Ted. *Speech on Net Neutrality Bill*. 2006. `http://youtu.be/f99PcP0aFNE`.