PA4

Start Assignment

Due Sunday by 11:59pm **Points** 50 **Submitting** a file upload **File Types** txt **Available** until Apr 18 at 11:59pm

Introduction

Modern operating systems use virtual memory and paging in order to effectively utilize the computer's memory hierarchy. Paging provides memory space protection to processes, enables the use of secondary storage, and also removes the need to allocate memory sequentially for each process.

We have studied how virtual memory systems are structured and how the MMU converts virtual memory addresses to physical memory addresses by means of a page table and a Translation Lookaside Buffer (TLB). When a page has a valid mapping from a virtual memory address to a 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 a memory request requires access to a page that is currently swapped out, it calls the operating system's page-fault handler. This handler must swap-in the necessary page, possibly evicting another page to secondary memory in the process. It then retries the offending memory access and hands 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 pages (sometimes called frames) can have a major effect on the performance of the memory access on a given system. **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.**

Your Task

The goal of this assignment is to implement a paging strategy that maximizes the performance of the memory access in a set of predefined programs. You will accomplish this task by using a paging simulator that has been created for you. Your job is to write the paging strategy that the simulator utilizes (roughly equivalent to the role the page fault handler plays in a real OS).

Your initial goal will be to create a Least Recently Used (LRU) paging implementation. You will then need to implement some form of predictive page algorithm to increase the performance of your solution. You will be graded on the throughput of your solution (the ratio of time spent doing useful work vs time spent waiting on the necessary paging to occur).

The Paging Simulator Environment

The paging simulator has been provided for you in PA4.zip. You have access to the source code if you wish to review it (simulator.c and simulator.h), but you should not need to modify this code. You will be graded using the unmodified simulator, so any enhancements to the simulator program made with the intention of improving your performance will be for naught.

The simulator runs a random set of programs utilizing a limited number of shared physical pages. Each process has a fixed number of virtual pages (that compose the process's virtual memory space) that it might try to access. For the purpose of this simulation, all memory access is due to the need to load program code.

The simulated program counter (PC) for each process dictates which memory location that process currently requires access to, and thus which virtual page must be swapped-in for the process to successfully continue.

The values of the constants mentioned above are available in the simulator.h file. For the purposes of grading your assignment, the default values will be used:

- 20 virtual pages per process (MAXPROCPAGES)
- 20 simultaneous processes competing for pages (MAXPROCESSES)
- 128 memory unit page size (PAGESIZE)
- 100 tick delay to swap a page in or out (PAGEWAIT)
- 100 physical pages (frames) total (PHYSICALPAGES)
- 40 processes run in total (QUEUESIZE)

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 in at any given time.

In addition, swapping a page in or out is an expensive operation, requiring 100 ticks to complete. A tick is the minimum time unit used 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! If all physical pages are in use, this turns into 200 ticks per page miss since you must also spend 100 ticks swapping a page out in order to make room for the required page to be swapped in. This leads to an "overhead to useful work" ratio of 200 to 1, which is very, very, poor performance. Your goal is to implement a system that does much better than this worst case scenario.

The Simulator Interface

The simulator exports three functions which you will use to interact with it: pageit(), pagein() and pageout().

The first function, pageit(), is the core paging function. It is roughly 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.) or basically every CPU cycle, which we'll refer to as a tick. 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 pageit() function is passed an array of pentry structs, one per process. This struct contains a copy of all of the necessary memory information that the simulator maintains for each process. You will most likely need the information contained in this struct to make intelligent paging decisions. You can read these fields as necessary, but **you should not write to them**, as any changes you make will be lost when you return from pageit(). The struct contains:

long active	A flag indicating whether or not the process has been completed. 1 running, 0 exited.
long pc	The value of the program counter for the process. The current page can be calculated as page = pc/PAGE_SIZE .
long npages	The number of pages in the processes memory space. If the process is active (running), this will be equal to MAX_PROC_PAGES . If the process has exited, this will be 0.
long pages[MAX_PROC_PAGES]	An array representing the page map for a given process. If pages[X] is 0, page X is swapped out, swapping out, or swapping in. If pages[X] is 1, page X is currently swapped in.

The simulator also exports the functions pagein() and pageout() , which are used to request that a specific page for a specific process be paged in or out. You will use these functions to control the allocation of virtual and physical pages when writing your paging strategy. Note that a page will be marked as swapped out as soon as the pageout() request is made, but is not recognized as swapped in until after the pagein() request completes 100 ticks later.

Pagein() and pageout() will return a 1 if they succeed in starting a paging operation, if the requested paging operation is already in progress, or if the requested state already exists. 100 ticks after requesting a paging operation, the operation will complete.

These functions 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). See Figure 1 below for more details on the behavior of pagein() and pageout().

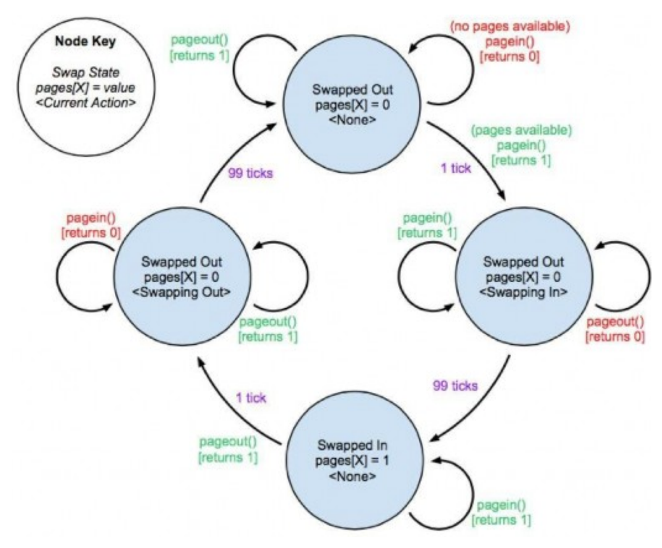

Figure 1: Possible Page States and Transitions

Figure 1 shows the possible states that a virtual page can occupy, as well as the possible transitions between these states. Note that the page map values alone do not define all possible page states. We must also account for the possible operations currently underway on a page to fully define its state. While the page map for each process can be obtained from the pageit() input array of structs, there is no interface to directly reveal any operations underway on a given page. If knowing whether or not a paging operation is underway on a given page (and thus knowing the full state of a page) is necessary for your pageit() implementation, you must maintain this data yourself.

The Simulated Programs

The simulator populates its 20 processes by randomly creating them from a collection of 5 simulated "programs". Pseudo code for each of the possible 5 programs is provided below:

Program 1 - A loop with an inner branch

```
# loop with inner branch
for 10 30
  run 500
  if .4
    run 900
  else
    run 131
  endif
end
exit
```

Program 2 - Single loop

```
# one loop
for 20 50
 run 1129
end
exit
```

Program 3 - Double nested loop

```
#doubly-nested loop
for 10 20
  run 1166
  for 10 20
    run 516
  end
end
exit
```

Program 4 - Linear

```
#entirely linear
run 1911
exit
```

Program 5 - Probabilistic backward branch

```
# probabilistic backward branch
for 10 20
  label :
    run 500
    if .5
      goto label
    endif
end
exit
```

This simple pseudocode notation shows you what will happen in each process:

- for X Y : A "for" loop with between X and Y iterations (chosen randomly)
- run Z : Run Z (unspecified) instructions in sequence
- if P : Run next clause with probability P, run else clause (if any) with probability (1-P)
- goto label : Jump to "label"

As we discuss in the next section, you may wish to use this knowledge about the possible programs to:

- Profile processes and know which kind of programs each is an instance of.
- Use this knowledge to predict what pages a process will need in the future with rather high accuracy.

Note that while you know the structure of these programs, any program's flow is still probabilistic in nature. Which branch a specific process takes or how many loop iterations occur will be dependent upon the random seed generated by the simulator. Thus, you may never be able to perfectly predict the execution of a process, only the probabilistic likelihood of a collection of possible execution paths.

Some Implementation Ideas

In general, your pageit() implementation will need to follow the basic flow presented in Figure 2. You will probably spend most of your time deciding how to implement the "Select a Page to Evict" element.

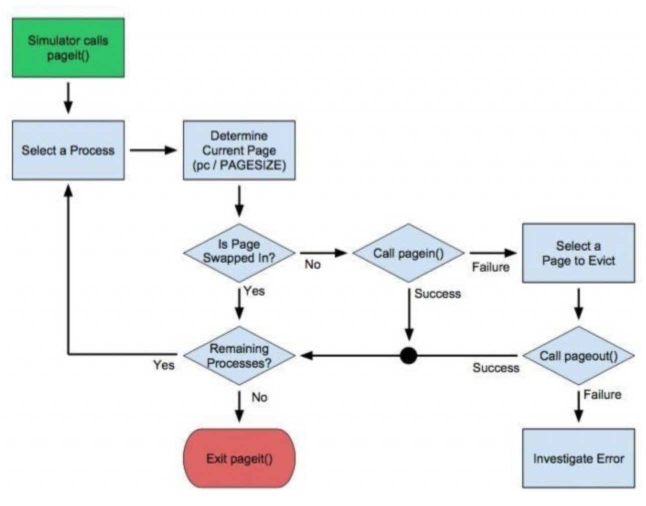

Figure 2: Basic Reactive pageit() flow

A basic "one-process-at-a-time" implementation is provided for you. This implementation never actually ends up having to swap out any pages. Since only one process is allocated pages at a time, no more than 20 pages are ever in use. When each process completes, it releases all of its pages and the next process is allowed to allocate pages and run. This is a very simple solution, and as you might expect, does not provide very good performance. Still, it provides a demonstration of the simulator API. See pager-basic.c for more information.

To start, create some form of "Least Recently Used" (LRU) paging algorithm. An LRU algorithm selects a page that has not been accessed for some time when it must swap a page out to make room for a new page to be swapped in. An LRU algorithm can either operate globally across all processes, or locally to a given process. In the latter case, you may wish to pre-reserve a number of physical pages for each process and only allow each process to compete for pages from this subset. A stub for implementing an LRU version of pageit() has been created for you in the pager-lru.c file. Note the use of static variables in order to preserve local state between calls to pageit() . Your LRU algorithm should perform much better than the trivial solution discussed above, but will still suffer from performance issues. We can do better.

To really do well on this assignment, you must create some form of predictive paging algorithm. A predictive algorithm attempts to predict what pages each process will require in the future and then swaps these pages in before they are needed. Thus, when these pages are needed, they are already swapped in and ready to go. The process need not block to wait for the required pages to be swapped in, greatly increasing performance. Figure 3 shows a modified version of the Figure 2 flowchart for a predictive implementation of pageit() . A simple predictive stub has been created for you in the pager-predict.c file.

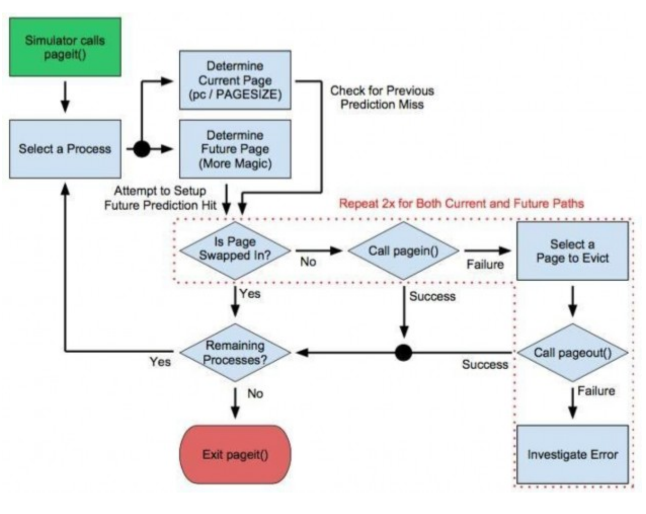

Figure 3: Basic Predictive pageit() Flow

There are effectively two approaches to the predictive algorithms. The first approach is to leverage your knowledge of the possible program types (see previous section). In this approach, one generally attempts to heuristically determine which program each process is an instance of by tracking the movement of the process' program counter (PC). Once each process is classified, you can use further PC heuristics to determine where in its execution the process is, and then implement a paging strategy that attempts to swap in pages required by upcoming program actions before they occur. Since the programs all have probabilistic elements, this approach will never be perfect, but it can work quite well.

The second approach to predictive algorithms is to ignore the knowledge you have been given regarding the various program types. Instead, you might track each process' program counter to try to detect various common patterns (loops, jumps to specific locations, etc). If you detect a pattern, you assume that the pattern will continue to repeat and attempt to swap in the necessary pages touched during the execution of the pattern before the process needs them. Working set algorithms are a subset of this approach.

Note that in any predictive operation, you ideally wish to stay 100-200 ticks ahead of the execution of each process. This is the necessary predictive lead time in which you must make paging decisions in order to insure that the necessary pages are available when the process reaches them and that no blocking time is required. As Figure 3 shows, in addition to swapping in pages predicatively, you must still handle the case where your prediction has failed and are thus forced to re-actively swap in the necessary page. This is referred to as a predictive miss. A good predictive algorithm will minimize misses, but still must handle them when they occur. In other words, you can not assume that your predictions always work and that every currently needed page is already available.

There are a number of additional predictive notions that might prove useful involving state-space analysis, Markov chains, and similar techniques. We will leave such solutions to the student to investigate if they wish.

What's Included

We provide code in PA4.zip to get you started.

- Makefile : GNU Make makefile to build all the code listed here.
- README : As the title instructs: please read it.
- simulator.c : Core simulator source code, for reference only.
- simulator.h : Simulator header file including the simulator API.
- programs.c : Struct representing simulated programs. For use by simulator code only.
- pager-basic.c : Basic paging implementation that only runs one process at a time.
- pager-lru.c : Stub for your LRU paging implementation.
- pager-predict.c : Stub for your predictive paging implementation.
- api-test.c : pageit() implementation that detects and prints simulator state changes. May be useful if you want to confirm the behavior of the simulator API. Builds to test-api .
- test-* : Executable test programs. Runs the simulator using your pager-*.c strategy. Built using the Makefile . The simulator provides a lot of tools to help you analyze your program. Run ./test-*
- -help for information on available options. It also responds to various signals by printing the current page table and process execution state to the screen (try ctrl-c while simulator is executing).
- test-api : API test program. See api-test.c .
- see.R : R script for displaying a visualization of the process run/block activity in a simulation. You must first run ./test-* -csv to generate the necessary trace files. To run a visualization, launch R in windowed graphics mode (in Linux: R -g Tk & at the command prompt) from the directory containing the trace files (or use setwd to set your working directory to the directory containing the trace files). Then run source("see.R") at the R command prompt to launch the visualization.

What You Must Submit

When you submit your assignment, you must provide the following:

- The pager-lru.c of your LRU paging implementation
- The pager-predict.c of your best predictive paging implementation
- Any additional .c and .h files you might have created to support your pager implementations

If the only files you modified are pager-lru.c and pager-predict.c, simply type '**make submit**' inside your working directory. Enter your Identikey username when prompted, make will generate a <username>.txt file for submission to Canvas.

Grading

As with PA3, this assignment will have both an interview score and code score, each worth 50% of your overall PA4 grade.

Your code will be subjected to an automated grading script which will evaluate your paging algorithm as measured by the overhead to useful work score (blocked/compute cycles):

- score >= 12.0: 0 Points
- 1.28 <= score < 10.0: 5 Points
- 0.64 <= score < 1.28: 10 Points
- 0.32 <= score < 0.64: 15 Points (Basic LRU implementation)
- 0.16 <= score < 0.32: 20 Points
- 0.08 <= score < 0.16: 25 Points
- 0.04 <= score < 0.08: 30 Points
- 0.02 <= score < 0.04: 35 Points
- 0.01 <= score < 0.02: 40 Points (Good predictive implementation)
- 0.005 <= score < 0.01: 45 Points
- 0.000 <= score < 0.005: 50 Points (Excellent predictive implementation)

We will run your code using several random seeds and will use the average of these runs as your coding score. Thus, if your program's performance varies widely from run-to-run, you may get bitten by our automated grader.

If your code generates warnings when building on the standard VM using -Wall and -Wextra you will be penalized 2 points per warning. In addition, to receive full credit your submission must:

- compile and run to completion in the standard VM using the provided Makefile
- meet all requirements elicited in this document
- adhere to good coding practices
- be submitted to Canvas prior to the due date