

CS 385 –Operating Systems – Fall 2011

Homework Assignment 4

Simulation of a Memory Paging System

Due: Tuesday November 15th. Electronic copy due at 2:30, optional paper copy at the beginning of class.

Overall Assignment

For this assignment, you are to write a simulation to analyze different virtual memory paging algorithms. A given number of simultaneous processes will be simulated, each making a certain number of memory accesses during each of their time slices, and the total number of page faults (and time to handle them) will be analyzed.

Command Line Arguments

The command line for this program will be as follows:

```
vmpager inputDataFile [ #processes MB_RAM maxReads fractionWrite . . . ]
```

- inputDataFile is a data file to be read and interpreted as page requests. See below for details.
- #processes is the number of simultaneous processes sharing the available virtual memory. Default value if not provided = 64.
- MB_RAM is the amount of memory available to the virtual memory manager. (I.e. the amount of memory available on the system, minus that memory not managed as virtual memory.) A value of zero indicates an infinite amount of memory, or at least enough that no page ever needs to be paged out. (See below.) Default value should be 16 MB.
- maxReads is the maximum number of memory accesses a process may make during a time slice, equal to 512 if not provided.
- fractionWrite is the fraction of memory accesses that are writes, equal to 2 if not provided. (This is the denominator of a fraction, so 2 implies one-half, 3 implies one-third, etc.)
- You may have additional parameters that follow the above parameters, such as which paging algorithm to use, how many reference bits, frequency of updates, etc. This is up to you, but must be well-documented in your readme file.

Virtual Memory Configuration

For this assignment the system hardware is assumed to have the following properties:

- Page tables are of size 256, because the page number is an 8-bit unsigned number.
- The offset is 12 bits, yielding 4kB pages and a maximum addressable space of 1MB per process. (20 bit virtual addresses.)
- Frame numbers are 12 bits, yielding 24-bit physical addresses. This results in 4096 total frames of 4kB each for a maximum addressable physical memory size of 16 MB. (Note: Ignore this restriction when processing the infinite memory case. All you really need to do for that case is to count the first reference to any page as a page miss and all subsequent references to the same page as page misses.)

Data File Interpretation

The data file is to be read as a series of binary numbers, in the following format:

1. Skip over the first 5000 bytes of the file by using `fseek` to position 5000 relative to the beginning of the file.
2. An unsigned short is read, and modded with the number of processes to determine the PID of the process which gets the next time slice on the CPU.
3. A second unsigned short is read, and modded with the maximum number of memory accesses (`maxReads`), to determine how many memory accesses this process makes on this time slice.
4. That process will then make that many page accesses, by reading that many unsigned chars, (integers in the range 0 to 255 inclusive), from the file and processing each as a page number in that process's virtual address space.
 - The first access made by any particular process (on that process's first turn) is always for reading.
 - After that, each page number read from the file is modded with `fractionWrite`, and if the result is zero, then the **next** page accessed is for writing, otherwise the next access is for reading.
5. Steps 2 to 4 are repeated until the end of the file is reached. Note that the last process may or may not complete their selected number of accesses when the end of the file is reached.

Algorithms to Explore:

Your program should examine the following paging algorithms:

1. Infinite Memory. Or at least enough that no page ever needs to be paged out, which will be 1 MB times the number of running processes, (or more if you implement an algorithm / policy that calls for a minimum number of free pages at all times.) This should generate the minimum number of possible page faults, because once a page is loaded the first time, it never needs to be paged out and loaded again. (Although the number of page faults generated will still be a function of the number of processes running, because page 42 for process X is not the same page as page 42 for process Y, and pages still need to be loaded the first time they are accessed no matter how much memory is available.)
2. FIFO. This is the most basic simple algorithm, and can easily be implemented by placing frame numbers in a queue when they are loaded and selecting victim frames from the other end of the queue when needed.
3. [**Optional Enhancement in Fall 2011**] Your best algorithm. Using the techniques covered in class, develop the best algorithm you can, (other than OPT), and compare it against the other two. This can be either one of the book's algorithms used without any modification, or a combination of ideas fine-tuned to optimize performance. Some of the issues you may want to explore are local versus global page replacement policies, the use of a free list, working sets, etc.

Constraints and Timing Parameters

- Assume that the hardware provides support for a single reference bit, and a dirty bit. Up to 8 additional bits can be implemented in software, i.e. as a field in the page table or frame table.
- Each process must be guaranteed a minimum of 32 pages that it can keep in memory without losing them to other processes. (This could possibly be a variable parameter to explore.)

- Analysis can be done in terms of the number of page faults and disk accesses made, using the infinite memory case as a benchmark. (Note that if a dirty page is selected as a victim page, then an additional disk access is required.)
- Alternatively timing could be considered by counting a page access as 1 time unit and a disk operation as 40,000 time units. Under this scheme a page hit requires 1 time unit, a page miss when no dirty pages are involved requires 40,001 time units, and replacing a dirty page requires 80,001 time units. If you incorporate an algorithm that periodically copies dirty pages to disk then that can be done as a "free" background activity, though the number of such writes that can be performed per time slice should be limited.
- The evaluation criteria is the ADDITIONAL number of page faults and disk accesses required for your algorithms as compared to infinite memory, expressed as a percentage. So for example, if infinite memory requires a million time units for a particular input data file and FIFO requires 1.2 million, then the additional time is 20%.

Simplifying Assumptions

- Time spent in queues is ignored. It is assumed that the I/O device is always available when needed, and that I/O does not delay the rest of the process.
- Process context switch time is ignored.
- Processes do not get swapped out of the CPU when a page fault occurs. They continue to keep the CPU until they have made their allotment of memory accesses. (This is a really BAD, FALSE assumption, but it is necessary to produce comparable usage patterns.)
- Time spent looking things up and/or storing things in page tables and/or frame tables is ignored. I.e. the virtual memory manager gets to use "free" CPU cycles to do its job. (Another BAD assumption, in that there is no penalty for complex algorithms unsupported by hardware.)
- Assume that all page tables and the frame table can be kept in memory at all times, and that they do not consume virtual memory. (I.e. you do not need to page the page tables, and the storage allocation for these data structures lies outside the domain of the virtual memory space.)
- The time spent flushing dirty pages out to disk after a process exits is ignored, (as is the entire concept of processes finishing and freeing up their pages.)

Required Output

- All programs should print your name and CS account ID as a minimum when they first start.
- Beyond that, this program should print run statistics suitable for algorithm analysis. Some of the data you may want to collect include the number of processes, frame table size, number of page hits & misses, time lost due to paging, number of pages swapped out (and then swapped back in again later), etc. Your ultimate goal is to produce an analysis comparing the performance of different paging algorithms under different load conditions. (At one extreme there is enough memory that pages never need to be swapped out once they are swapped in. At the other extreme, almost every page access causes a page miss.)
- In addition to a program printout, a short memo / report shall be written with the results of the algorithm analysis. Ideally you would like to determine the conditions under which one algorithm is better or worse than another, with statistical data to back up your conclusion. (You should test your program using at least two different data files, hopefully showing different usage patterns.)

Other Details:

- A makefile is required, that will allow the TA to easily build your program. As always, you are free to develop wherever you wish, but the TA will be grading the programs based on their performance on the CS department Linux machines.

What to Hand In:

1. Your code, **including a makefile if needed, and a readme file**, should be handed in electronically using Blackboard.
2. The purpose of the readme file is to make it as easy as possible for the grader to understand your program. If the readme file is too terse, then (s)he can't understand your code; If it is overly verbose, then it is extra work to read the readme file. It is up to you to provide the most effective level of documentation.
3. The readme file must specifically provide direction on how to run the program, i.e. what command line arguments are required and whether or not input needs to be redirected. It must also specifically document any optional information or command line arguments your program takes, as well as any differences between the printed and electronic version of your program.
4. If there are problems that you know your program cannot handle, it is best to document them in the readme file, rather than have the TA wonder what is wrong with your program.
5. A printed copy of your program, along with any supporting documents you wish to provide, (such as hand-drawn sketches or diagrams) should be handed in **at the beginning of class** on the date specified above.
6. Make sure that your **name and your CS account name** appear at the beginning of each of your files. Your program should also print this information when it runs.

Optional Enhancements:

- More than three paging algorithms. OPT is a particularly good choice for an extra algorithm, as it yields the minimum number of page faults for a limited-memory system.
- Explore parameter adjustments within algorithms, and how adjusting those parameters affects paging performance under different load conditions. (Partially included in finding your "best" algorithm. This enhancement would require a more thorough analysis of parameter effects.)
- Adjust the balance of reading versus writing accesses, and see how it affects each algorithm.
- It is intended that the scheme of reading data from files (and the particular data files provided) will produce some locality of reference, reasonably representative of real process performance. It would be interesting to analyze the data files, and to determine which (if any) exhibit good locality of reference properties.
- Can you identify the threshold conditions under which thrashing begins to occur?
- Elimination of one or more of the simplifying assumptions.
- Explore shared memory. Mark a certain number of each process's pages as shared, either with a specific subset of other processes, or with all other processes. The shared pages should be randomly determined based on the data read, such as:
 - If a page number divisible by fractionShared is read, then the following page is a shared page, up to a maximum of 64 shared pages per process.
 - The first shared page of process X is the same as the first shared page of all other processes, and likewise for the second pages, third pages, etc. So for example, if the first shared page for process X is 42, and the first shared page for process Y is 13, then page 42 of process X occupies the same frame as page 13 of Y.

- Copy-on-write would be a separate optional enhancement.
- Calculate the predicted hit ratios and timings by hand, and see how well your simulation matches your predictions.
- Can you express the results of your simulation in a (simple) formula? I.e. for a given number of processes, memory sizes, etc, can you predict the page hit ratios and program timings?