CSci 5103: Operating Systems

Project 2: Virtual Memory

Due: 10 PM March 20, 2012

1. Objective

In this project you will design and evaluate the components of a virtual memory system, explore policy options, assess their behavior, and analyze and present results.

2. Design Overview

You will build and simulate a virtual memory system. The components include a simple per process paging mechanism, simple address translation, a MMU/TLB, and various page replacement policies. You will evaluate various choices for TLB replacement, page replacement, and page size.

You must use C/C++ and the code must run on the virtual machine distributed on from the course's Moodle site. You will implement your VM simulator as a Linux process that contains basic code for virtual memory management. As before, we strongly recommend the use of a modular structure.

3. Project Details

Inputs: The input to your simulator will be a list of per-process memory access traces that contain instructions of the form:

R <VA> /* read VA */

W <VA> /* write VA */

You will assume a virtual address space of 32 bits, with 20 for the page, and 12 for the offset initially (PS = 2^12). Your system should allow a variable number of processes as input each with different memory traces. In this simulation, each instruction takes 1 unit of time (pure CPU time), a context-switch 5 units (pure CPU time), and a page-fault incurs 10 time units of delay (to bring in a page). To issue the I/O for the page-fault takes 1 time unit (pure CPU time) in addition to the 10 unit delay. Keep a virtual clock that starts at time 0. The I/O cost of 10 time units is the delay incurred to wait for an I/O completion, thus the CPU can process other instructions in parallel with the I/O delay. All processes are assumed to arrive to the system at time 0.

Parameters/Metrics: Your memory system will consist of M frames each of size PS. Your TLB with have sufficient memory to store at most K translations. You must keep track of TLB hits/misses, context-switches, page in, page outs, and so on.

Address translation: You will implement a *very simple* MMU that knows about the TLB and expects the current page table address to be in a register variable PTBR. You do not need to worry about protection, illegal addresses, and so forth. During a context-switch, you put the current page table address into the PTBR.

MMU/TLB: during address translation, you will check the TLB “in parallel” with the page-table. Since the TLB is of finite-size, you may have to do TLB replacement. You will implement a simple TLB that performs FIFO replacement. The TLB must also be flushed when processes are switched.

OS/Paging: You will implement pure demand-paging using single-level page tables for all processes. Each process will have its own page table. Your PTEs will contain reference, dirty bits, a time field (set only by the OS) and the frame #. The first access to a new page causes a page fault. You resolve the page fault by locating a free frame, loading the page into the frame (simulated, of course), and updating the TLB. You will also keep track of free and allocated frames. When a process terminates (its trace ends), all of its pages are reclaimed. Processes will be scheduled round-robin with a quanta of P instructions (e.g. R and W above). You may try different values of P. You will also switch to the next process if the current process page-faults during its quanta. If there are no other eligible processes, do not context-switch. This process is eligible to run again only when the page-in has completed. Assume that the context-switch and page-in can be overlapped in virtual time.

Page replacement: You will implement the following **GLOBAL** page replacement policies:

1. FIFO
2. Pure LRU – use the current virtual time for each access
3. LRU approx – when control returns to the OS (think about when this will happen), set the access time for each page with a reference bit of 1 to be the virtual time; then reset all reference bits to 0

d) Extra-credit: implement a version of working set

As in many systems, e.g. Linux, you will implement a page cleaning "daemon". The job of this daemon is to ensure that at least F frames are available at any time. When the system falls below F free frames, then it will "clean" (i.e. free-up) G frames. Be careful to lock/pin frames for any pending page-ins from prior page faults, i.e. do not clean frames that are expecting pages. Assume there is sufficient bandwidth for any number of concurrent I/Os and the I/O delay is 10 time units. Assume however it takes one time unit (pure CPU time) to issue a page-out I/O per frame. So to clean G pages, the cost would be G time units (pure CPU time) +10 time units of delay. **However, to reclaim a non-dirty page costs nothing.**

The daemon will adopt a page replacement strategy as dictated by a-d above. You may elect to do this synchronously (i.e. have your OS simply invoke the page cleaning after a page-fault is signaled and the threshold has been crossed) or you can try to use a separate paging daemon process and coordinate with the OS process. Warning: if you use a separate process, you will have to "slow down" your simulation as it will run *very* fast. Set F and G appropriately to avoid continuous cleaning.

C) Evaluation.

You will evaluate the effect of changing page size (PS) and changing page replacement policy separately. Hold one constant and vary the other. You are to create traces that SHOW sensitivity to these parameter choices. We may provide a few traces as well. Explain the high-level characteristics of each trace (e.g. it has a certain type of locality, etc.) and why a particular trace performed a certain way (see D for performance metrics). A few data points that illustrate any trends are adequate. Why do the results look as they do? Explain any patterns – both expected and unexpected.

D) Data collection/Report. You will submit a simple report that summarizes the results of your simulations. You will record the number of page faults, page-ins/page-outs, TLB hits/misses, context switches, and average trace completion time, on separate graphs/tables with a brief analysis.

4. Submission and Grading

Testing:

- There is no restriction on machines and OS for developing your code, but it should be compiled and run correctly on the course virtual machine.
- We will provide trace files for execution.

Submission:

- Zip up/Tar your code and submit it online.
- Projects **MUST** include a readme.txt file in your assignment which has the following information:
  - Your names;
  - Your student IDs;
  - How to build your program (ideally via "make" or "make all")
  - How to run your program;
- No need to submit a paper copy. Save trees instead.

Grading:

- Code modularity and documentation: 10%
- TLB and address translation: 10%
- Page replacement algorithms: 40%
- Page fault handling: 10%
- Paging Daemon: 10%
- Data Analysis: 20%
- Working set: (extra credit) 10%