Com S 352 Fall 2016 Project 2:

A Demand-Paging Virtual Memory Simulator

Due: Friday, December 2, 11:59pm

Objective

In this project, you are required to develop a multi-threaded C/C++ or Java program to simulate the working of a demand paging-based virtual memory system. In this multi-thread program, multiple concurrent ``user processes'' (simulated by multiple user threads) will access addresses in their own ``logical address spaces'', which will be dynamically mapped to a ``physical main memory'' according to the demand paging strategy and the LRU page replacement algorithm that we learnt from the classes.

Behaviors of the Program

The inputs to the program will include:

- S (the size of a page/frame in the unit of Byte),
- P (the maximum number of pages in the logical memory space of each process),
- F (the number of frames in the physical memory),
- n (the number of ``user processes'' to be simulated), and
- n plaintext files (named trace_1.txt, trace_2.txt, ..., trace_n.txt), where the ith (i= 1, ..., n) file trace_i.txt stores a list of logical addresses (in decimal) that are to be accessed by the ith ``user process'' in sequence.

The program will simulate that it has a physical main memory of F frames for user processes, and the size of each frame is S. Initially all frames will be empty.

Once the program is started, the main thread will create a memory manager thread (simulating the behavior of paging hardware), a page fault handler thread (simulating the procedure of handling a page fault triggered by the memory manager thread), and additional n threads that each simulates a user process.

Each simulated user process will pretend to have its logical address space of P pages. It issues a sequence of logical addresses, pass each of the addresses to the memory manager thread, which will check if the address is valid and if page fault should be triggered. If an address is invalid, access to the address should be denied and an error message should be printed out. If a page fault should be triggered, the page fault handler thread should be notified to start page fault handling. Otherwise (the address is valid and does not trigger page fault), the memory manager thread should translate the logical address into physical address, and print out the physical address.

The simulated user processes run concurrently. For simplicity, no TLB will be simulated.

Formats of Outputs

The simulated ith user process will read the logical addresses in the ith file one by one, and for each address, it will simulate that it is accessing the address. That is, it submits the local address to the memory manger thread, who needs to extract the page number of the address. Then:

• If the page is in the simulated physical main memory, a message as follows will be output:

[Process i] accesses address x (page number = p, page offset=d) in main memory (frame number = f).

• If the page is not found in the simulated physical main memory, triggering a page fault will be simulated. Specifically, the page fault handler thread should be notified to execute the demand paging strategy:

· Firstly, it tries to find a free frame in the main memory. A message as follows should be output:

[Process i] accesses address x (page number = p, page offset = d) not in main memory.

· If there is no free frame found, the LRU algorithm should be used to determine which page should be replaced. One of the following messages should be output:

o When a free frame is found:

[Process i] finds a free frame in main memory (frame number = f).

o Otherwise:

[Process i] replaces a frame (frame number = f) from the main memory.

· The page fault handling thread should sleep for 1 millisecond (i.e., calling usleep(1000) in C) to simulate that the time needed to swap the demanded page into the main memory. (Note that, the underlying OS may swap the sleeping thread out and switches to run another thread; this does not matter to our simulation.) The following message should be output before the thread goes to sleep:

[Process i] issues an I/O operation to swap in demanded page (page number = p).

· When the thread becomes active from the sleep, the page table of the user process that it is simulating should be updated accordingly. Also, the following message should be output:

[Process i] demanded page (page number =p) has been swapped in main memory (frame number = f).

· Finally, the interrupted address access should be resumed and completed. [Process i] accesses address x (page number = p, page offset =d) in main memory (frame number = f). After all addresses in the ith file have been accessed, the ith thread (a.k.a., the ith user process simulated by it) will terminate. Before termination, it should output the

following message: [Process i] ends.

Notes

- The actual execution order of the concurrent threads (a. k. a., their simulated user processes) is determined by the underlying Linux OS that the simulator is running on. So, you do not need to schedule their executions.

- If a data structure is shared by multiple threads, mutual exclusion mechanisms (e.g., POSIX semaphores, pthread mutexs, etc.) should be used. You have the freedom to choose the mechanisms that you want to use.

- For simplicity, let's assume there is only one page fault handler. Before one page fault has been completely handled, no other page fault can be handled. However, while one page fault is being handled, accesses made by other user processes can be handled by the memory manager thread as long as these accesses do not cause page fault.

User Interface

- Name your main program as VMsim.c, VMsim.cpp, or VMsim.java.

- Parameters S, P, F and n should be provided to your program as command-line arguments. For example, if your code is compiled to VMsim, then it can be launched on pyrite as:

  >VMsim 1024 16 32 10

  to simulate the running of 10 user processes, where the page/frame size is 1024 bytes, the number of pages of each process is at most 16 and the number of frames in the main memory is 32.

- In each address file trace_i.txt, each line contains only one address. For example, the content of file trace_1.txt may be as follows:

  1024
  2048
  3072
  4079

Other Instructions

- Your code should be compiled successfully in pyrite. Otherwise, you may receive no points. Your program can be written in C, C++, or Java.
- Write comments in your code to help testing and grading.
- Well test your code before submission.
- Your code should be put into a folder (named as <your_isu_email_address>352Proj2; for example, wzhang352Proj2), and then zipped into a single file (named as <your_isu_email_address>352Proj2.zip; for example, wzhang352Proj2.zip). After being unzipped, there should be a makefile or a script file that can be run by the TA to compile your code.

Start as early as possible!