MP3: Nachos Virtual Memory
CSCI 40300/ECE 40800 - Operating Systems
Due date: Friday, Apr 20, 2012, 11:55pm

1 Overview

This machine problem involves implementation of part of a virtual memory subsystem for Nachos. This is the first MP that the virtual machine of Nachos will be used. The code for the test case is a set of array operations that span large portions of the virtual memory space. This virtual memory space is mapped into 20 physical frames of memory. Address translation and page table structures are given by the machine implementation (in `machine` directory). The purpose of this MP is to implement the routines that handle page faults. This will involve swapping pages to and from the backing store when appropriate. In an effort to minimize the number of page faults and I/O swaps from memory to disk, you will be implementing a page buffering scheme and three page replacement algorithms, First In First Out (FIFO), Least Recently Used (LRU - Approximation using Additional-Reference bits), and Enhanced Second Chance (ESC).

2 Preparation

NOTE: THIS MP IS LONG AND WILL REQUIRE SUBSTANTIALLY MORE TIME TO COMPLETE THAN PREVIOUS ASSIGNMENTS. START IT NOW!

Materials: The given code for this MP can be extracted with the usual tar command with the appropriate command line arguments as in the previous projects. Once you unzip and untar the archive the source code will be in the "mp3given" directory. For this MP, you will be working in the "userprog" subdirectory. Look at the files filesys/OpenFile.java, filesys/OpenFileStub.java, machine/TranslationEntry.java, machine/Timer.java, userprog/NoffHeader.java, and userprog/BitMap.java. It will be helpful to understand the additional functions added to the List class in threads/List.java as well (note that these functions are not optimized but are given for your convenience in programming). In this assignment you will need to modify only MemManager.java (in userprog directory).

Structures: As you know, each address space has a page table associated with it. Each entry in the page table is an instance of a TranslationEntry. The three primary fields that we will be working with in this MP are the USE, DIRTY, and VALID fields. These are described thoroughly in machine/TranslationEntry.java.

The structure AddrSpace consists of all the information about the address space of a nachos process - the page table, its length, etc, and functions to manipulate the address space. When a new process is created this structure is instantiated, initialized and associated with the process. The code supplied implements demand paging. In this paging scheme a page is read into the physical memory from the source (executable) file only if the process accesses it during its execution. So initially when the process is created all the entries in the page table are initialized to invalid (the valid bit of the all page table entries is set to false). However, in general it may not be legal to access certain pages (the ones that do not belong to any program segment). While initializing the table a note is made about which pages are legal to access and which are not (the legal field). Every executable file starts with a header which specifies the virtual address range of all the segments of the program. Read AddrSpace.AddrSpace() and the comments in NoffHeader.java for further details.

Beginnings: The AddrSpace.readSourcePage() function is called to read a single page from the executable file into the physical memory. Note that this function is called only when a virtual page is accessed for the first time by the process. For subsequent accesses, if the page is not in physical memory

the swap file rather than the source file is searched. The swap file is a single file, consisting of frames of memory just like MainMemory, except that they are stored on a disk. An array of TranslationEntry pointers named "swapOwners" is used to keep track of the page table entries that refer to the page stored there in the swap file. A similar structure, "coreOwners", exists for pages in main memory. Read AddrSpace.readSourcePage() and MemManager.pageIn() for further details.

Execution: During execution, the machine emulation will handle setting the appropriate bits in the TranslationEntry such as DIRTY=TRUE when a write occurs or USE=TRUE when a read or write occurs. If the machine emulator processes a memory request on a TranslationEntry that has a LEGAL=TRUE, but VALID=FALSE, it will trap to MemManager.faultIn(). This is the entry point for your implementation. Given the faulting TranslationEntry, use pageIn(), pageOut(), and methods that you will implement to clear the exception by loading the appropriate buffer into a frame of memory, updating the TranslationEntry and returning control to the virtual machine.

Read the code for MemManager.pageFaultExceptionHandler(). This is called by Nachos.exceptionHandler() when it receives a page fault exception. The details about how to implement MemManager.faultIn along with a few hints are given as comments in MemManager.pageFaultExceptionHandler(). Note that you should not increment the program counter in this function. After the exception is handled and the control is returned back to the faulting process the instruction which caused the page fault should be re-executed.

Anatomy: While "faultIn" is the "handler", various other methods encapsulate important functionality. pageOut is responsible for writing a page to the backing store. It is assumed that only dirty pages are given to pageOut because the others can be overwritten, since they are already on the backing store or are unchanged from their original state. pageIn is responsible for reading a page into a specified frame. If the page is not in the backing store, it is loaded from the original file. MemManager.makeFreeFrame is responsible for choosing a victim with the appropriate page replacement algorithm when no free frames are available. MemManager.recordHistory is a timer interrupt handler that updates history information from the appropriate bits in the TranslationEntry.

3 Reading Guide

Read this handout, then read userprog/AddrSpace.java, userprog/MemManager.java, and threads/Nachos.java (method exceptionHandler). Look in machine/TranslationEntry.java for the declaration of class TranslationEntry. You may also want to look at userprog/Bitmap.java. Look at MemManager.pageIn and MemManager.pageOut also. Make a paper design before starting to code.

4 Problem: Various Page Replacement Algorithms.

4.1 Prerequisite: RecordHistory

Implement the Interrupt Service Routing (MemManager.recordHistory) to record the state of the USE bit from each TranslationEntry that has a frame "in core". This is the algorithm that your textbook calls aging and uses n-bits that are updated periodically (Sec tion 3.4.7). Also, read the section 9.4.5.1 "Additional-Reference-Bits Algorithm" from the Siberschatz book. The relevant pages are attached to this assignment.

4.2 FIFO replacement (Sec 9.4.2 if textbook)

Recommend the selection of a frame in-core using the FIFO algorithm. Basically, you can use a FIFO queue to record the order of frames. In userprog/MemManager.java, the queue is given as an array of integers. queueCounter is used to track the head position of the queue.

4.3 LRU from N bit history (Sec 9.4.5.1)

From the history stored by RecordHistory(), recommend the selection of a frame in-core using the Least Recently Used algorithm. The number of bits in the history will vary based on the command line parameters. Please consider the bits in the history AND the current state of the USE bit. Search the victim page in coreOwners array. Use MemManager.counter to track the start position and increase the counter to search. MemManager.counter is initialized to 0. If there is a tie for the value, use the first frame encountered. Future requests should start considering frames starting directly after the frame recommended (i.e., returned from MakeFreeFrame). Do not forget to reset the history where appropriate (hint: when a page is brought into memory, it has yet to be used)

4.4 Enhanced Second Chance (ESC) (Sec 9.4.5.2 and 9.4.5.3)

From the USE and DIRTY bits of each frame in-core, recommend the selection of a frame using the Enhanced Second Chance algorithm. See the textbook for details. Implement the queue for second chance algorithm by array MemManager.queue and MemManager.queueCounter. (See the sections 9.4.5.2 and 9.4.5.3 and Figure 9.17 of your textbook.) At the beginning, the frames in MemManager.queue are in FIFO order. If there is a tie for the value, use the first frame encountered. Future requests should start considering frames directly after the frame recommended. Do not forget to reset the USE bit where appropriate. In the enhanced second chance (ESC) algorithm, all physical frames are divided into four classes. The frame in the lowest category and closest to the counter (mentioned above) is chosen as the page to be replaced. After finding the victim page, set MemManager.queue to the position directly after the victim page.

Note that, to find the victim frame we need to cycle through all the physical frames at most once. In fact, we need to go through the entire cycle only if there is no frame in the lowest category- (0,0). (If you give USE a value of 2 and DIRTY a value of 1, this problem is equivalent to finding the first least element in a list of numbers ranging from 0 to 3.)

5 Adding Page Buffering (Sec 9.4.7 of textbook)

Extend faultIn to implement page buffering. This function is called by MemManager.pageFaultExceptionHandler(). Make use of pageIn and pageOut functions described above and as seen in the given code.

A page buffer contains a set of physical frames. Note that the frames in the buffer need not be contiguous. These frames may be distributed randomly across the main memory. The number of frames in the buffer will vary based on command line arguments and will be available to the constructor. Please store this in the variable MemManager.buffersize. The initialization of page buffer has been done in the constructor. Each entry in the page buffer contains the physical frame number of the buffer page in main memory. Replacement of frames in the buffer is dependent on the following 2 cases:

1. If there is a free frame in memory, then the faulting page should be brought into that frame. No changes are necessary to the page buffer.
2. If there is no free frame in memory, then select a frame from main memory to be replaced. Search the page buffer for the faulting page (be careful with null pointers). Consider the following 2 cases
    a. If the page is in the page buffer, remove it from the buffer and add it to main memory. Add the frame selected from memory to the buffer. This amounts to manipulating the VALID “bit”.

b. If the page is not in the page buffer, replace the first page in the page buffer (FIFO order). Swap it out if necessary and page the faulting page into the selected buffer frame. Add the page selected from memory to the buffer.

6 Compiling Testing

In order to compile your code type: **make** from userprog directory. There is one test program: tc2. The source for this program is tc2.c in the test directory. The parameters of the test cases, however, are quite varied, because of the command line arguments. Test your program by typing:

./nachos -M <num_pages> <num_bits> -x ../test/tc2

from userprog directory where <num_pages> is the number of pages in the page buffer, <num_bits> is either zero for "enhanced second chance", greater than zero for "n bit LRU", or less than zero for "FIFO." Specifying zero <num_pages> disables page buffering. The test program uses a new "test case" system call (the code for which is in userprog/VMTest.java) to test and to otherwise display kernel internals. The standard outputs are given in the mp3given/std directory. Use the **test0** through **test9** batch programs to test your code and compare your output to the test.x.std files in the std directory. If you implement everything as specified, you should be able to match the standard outputs. testall is a batch program that runs all of the test cases and compares them to the standard test files.

The test cases will test pages ranging from 0 - 10 and bits from 1 to 10. Of course they will also test ESC and FIFO.

7 Submission / Grading Criteria

Submit the tar'ed and gzipped mp3given directory after appropriate running the "make cleanclass" command first. Submit this mp3given.tgz in the "Assignments 2" tab, MP3 page on Oncourse.

Your grade will be based on whether or not your program works correctly for the various page replacement algorithms and for the page buffering scheme (In order to get full credit for page buffering, the page replacement algorithms must still work correctly).

FIFO	25%
LRU	25%
ESC	25%
PageBuffering	25%
Total	100.00%