EECS 678:
Intro to Operating Systems

Programming Project 3:
Virtual Memory in Nachos

Nathan Disidore &
Parker Roth

December 1st, 2011

IMPLEMENTATION DESCRIPTION

To implement our solution we started by modifying the TranslationEntry class to track the data necessary to implement the various page replacement policies (PRP). This included adding a reference history which was represented by a bit buffer. This buffer was modified by setting the most significant bit to 1 after a translation, and shifted to the right after a wsRefresh. These two methods were necessary for the least recently used (LRU) PRP.

In addition to the reference history, the load time of a page was added to the translation entry object. This load time was set with the current CPU time at the time the page was paged into memory. This information was necessary to implement the first in first out (FIFO) and enhanced second chance (SC) PRPs.

Modifications were also made to the implementation of the working set (WS). The initial size was set to the minimum value of four pages to prevent a process from having more pages than required upon initialization. The refresh method was modified to complete the following tasks: First, it calculated a new WS size by analyzing the number of pages currently being accessed by a process and choosing a size accordingly. Next, the frames were compacted by using the chosen PRP. Finally, the translation entry's history information was shifted to the left using the according to the chosen delta value.

Three PRPs were implemented during this project. The first utilized the enhanced second chance algorithm. The purpose of this algorithm is to favor paging out pages that have not been reference and have not been written to (are clean). We implemented this functionality by looping over every page in the address space and made a decision based on the status of their reference bit, dirty bit, and load time. The best case was a clean and accessed page. Next, a dirty but still accessed page. Finally a clean accessed page and then a dirty accessed page. If an accessed page was analyzed its reference bit was cleared to improve the next page replacement request. Within each group only a page that had an older load time was stored. Upon completion of the loop, the PRP returned the best page by looking through the choices (the int array) in order of choice quality and selecting the first option it found. The wost case scenario would look similar to FIFO.

The goal of the next PRP, least recently used (LRU), is to select the page that was least recently used for replacement. We implemented this algorithm by first looping over all the pages in the address space. During each iteration, the history buffer is compared to the previously best choice. If the history is less than that of the previous best, the page was not accessed as recently because it has fewer high value bits set. In this case the page is designated as the new best choice. In the event two pages have equivalent history buffers, the best choice is chosen by the one with the older load time. After the loop the choice is returned.
The final PRP, first in first out (FIFO), simply chooses the oldest page in the working set as the victim. We implemented this by looping over all the pages in the address space to find the page with the lowest load time value. This page was then returned as the chosen victim.

TESTING AND DEBUGGING

Other than simple syntax errors caught at compilation time, most of the debugging for this project was done in response to results in the table. Once a PRP's output matched the expected value we were assured that it was working correctly and required no further attention.

We tested our implementation by running the regression tests and analyzing the resulting table. This test covered all the implemented PRPs and every delta value for each. To speed up the process of making nachos and the test results we modified the nachos Makefile to accept a make-table command that cleaned/compiled nachos, cleaned/compiled regressions tests, processed the resulting data, and created the table output.

MAKE TABLE RESULTS

The table outputs test results for each program/PRP/delta combination as a tuple of (page faults / page outs). The first program accesses a 1024 by 32 array of data in row major order. Since it accesses the data in the same manner it is stored it has fewer page faults and page outs than the results for access2. The reason access2 has higher rates is that it accesses the same 1024 by 32 array in column major order. This results in a page fault for every access (1024 x 32) instead of a page fault for every row change (1024).

The third program accesses a 10 x 32 array in row major order. It therefore generates one page fault for every row resulting in ideally that may rows. The fourth and final program simply runs program three 10 times and therefore has roughly 10 times the number of page faults and page

QUESTIONS

Question 1:

What is the absolute fewest possible number of page faults for access1 in Nachos? Note that the compiler stores its arrays in row-major order (see the note in access1.c) and that Nachos employs demand paging. What about for access3?

In the lab description, the maximum working set size is defined to be 32 pages. For the access tests, the column dimensions are defined such that each page holds exactly one row. Using a constant row dimension of 1024, the matrix is accessed 1024 x pageSize/sizeof(int) times. With a pagesize of 128 (as defined in phys.h) and an int being 4-bytes in length, the resulting value is that the matrix is accesses 32,768 times. However, because pages are stored in row-major order, a page only needs to be faulted in once the working set size has been reached, so the number of faults is reduced to 32,768/32 = 1,024 page faults for access1.

Access3 follows a very similar pattern but the row dimension is given to be 10. As such, memory is accessed 10 x 32 times for a total of 320 times. With the working set size again the maximum 32 pages, this results in 320/32=10 page faults.

Question 2:

What is the absolute fewest possible number of page faults for access2 in Nachos? Note that the difference between access1 and access2 is the pattern in which they traverse the array.

Question two is set up very similarly to the way access1 was set up in question one except for the column-major matrix access method. Again, with a row dimension of 1024 and column dimension of pageSize/sizeof(int), the data is accessed 32,768 times. Where it differs is because of the conflict between row-major storage and column-major access method, a page must be faulted in each time it's accessed. As such, we get a maximum number of page faults equal to the number of times the data is accessed: 32,768.

Question 3:

Why does delta affect access1 page fault count for FIFO and DUMB but not for LRU and Enhanced Second Chance?

All page replacement policies replace pages based on their sense of history. In the case of FIFO and dumb, this history is linear throughout program execution. As such, the time these pages are kept is dependent upon the size of the history window.

Conversely, LRU and Enhanced Second Change have the ability change a page's history to reflect usage during program execution. Because recently used pages are not needlessly paged out as is the case with FIFO and dumb, fewer page faults results. In the specific case of access1, since the data pages are not needlessly paged-out, there is the minimum number of page faults necessary to get the data in.

Question 4:

Look at the inter-page fault histogram for access1 with a delta of 8 for all the PRPs. All of the generated histograms are located in dsui-vm/page_replacement_policy/pdfs directory (the .histo files are located in dsui-vm/page_replacement_policy/histos). See the Testing and Validation section for how to create the histograms. There ought to be a main spike which all PRPs share. Explain the existence of that spike: to what does it correspond?

All page replacement policies for the access1 program share a spike in in the range of duration falling in the bucket 735-750. The value of this spike is 1024. Since for access1, a page maps to a row and the data is accessed in row-major order, the duration represents the amount of ticks that it takes the outer loop to run once, including the writing a value, increment the loop counter, check the loop condition, compute the next element's address, and then write to it.

Question 5:

Now consider page-outs and three major influences: 1) PRP, 2) delta size => working set size, and 3) the program's memory access pattern. Consider 3) as you answer these two questions:

- *For 1), in which programs did the PRP not really affect the page-outs, and give a reasonable reason for that.*
- *For 2), in which programs did the delta size have an effect? Where did it not? Why?*

1. Access3: the row dimension for access3 is only 10. Because data is being accessed in row-major order, the size of delta (history size) is much closer (comparatively) than the other programs.
2. Access2: access2 didn't care about the delta size because it's already running at it's worst case. As such, even 8 bits of history isn't particularly useful.