CSC369 Assignment 2 - Page Tables and Replacement Algorithms

June 3, 2020

1 Introduction

For this assignment, we're going back to the realm of user mode programming. Specifically, you will have to simulate the operation of page tables and page replacement. As I keep saying: the way to gain a solid understanding of the theory is by applying it in practice.

You have two tasks in this assignment, which will be based on a virtual memory simulator. The first task is to implement virtual-to-physical address translation and hldemand paging using a two-level page table. The second task is to implement four different page replacement algorithms: FIFO, Clock, exact LRU, and OPT.

Before you start work, you should complete the set of readings about memory, if you haven't done so already: link

2 Requirements

2.1 Setup

You will find the starter code HERE. It is your responsibility this time to add the code in your repository and make sure that you submit all the necessary files!

Note that you may be generating some large trace files and must NOT commit any of the trace files that you generate to your repository or you will run into serious problems with disk quota. Most of the trace programs should be familiar to you from the exercise in week 6. We have added a blocked version of matrix multiply, *blocked.c* which should exhibit fewer page faults under at least some of the page replacement algorithms. The Makefile shows you exactly how to compile and run the traces. Note that it takes quite a while to run the trace collection.

Compile the trace programs and generate the traces.

You may have noticed while doing the Exercise that the traces generated by Valgrind are enormous since they contain every memory reference from the entire execution. We have provided a program, *fastslim.py* to reduce the traces by removing repeated references to the same page that occur within a small window of each other while preserving the important characteristics for virtual memory simulation. (For example, a sequence of references to pages A and B such as "ABABABABAB...AB" are reduced to just "AB".) The runit script pipes the output of valgrind through this program to create the reduced trace. If you wish, you can experiment with fastslim.py to try omitting the instruction references from the trace or using a smaller or larger window (*fastslim.py –help*). You may also want to create traces from other programs, and you will definitely want to create small manual traces for testing.

2.2 Task 1 - Address Translation and Paging

Implement virtual-to-physical address translation and demand paging using a two-level pagetable.

The main driver for the memory simulator, *sim.c*, reads memory reference traces in the format produced by the fastslim.py tool from valgrind memory traces. For each line in the trace, the program asks for the simulated physical address that corresponds to the given virtual address by calling find_physpage, and then reads from that location. If the access type is a write ("M" for modify or "S" for store), it will also write to the location. You should read sim.c so that you understand how it works but you should not have to modify it..

The simulator is executed as *./sim -f <tracefile>-m <memory size>-s <swapfile size>-a <replacement algorithm>* where memory size and swapfile size are the number of frames of simulated physical memory and the number of pages that can be stored in the swapfile, respectively. The swapfile size should be as large as the number of unique virtual pages in the trace, which you should be able to determine easily.

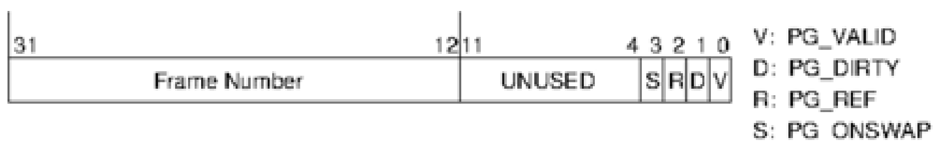
There are four main data structures that are used:

1. *char *physmem*: This is the space for our simulated physical memory. We define a simulated page size (and hence frame size) of *SIMPAGESIZE* and allocate *SIMPAGESIZE * "memory size"* bytes for *physmem*.

2. *struct frame *coremap*: The coremap array represents the state of (simulated) physical memory. Each element of the array represents a physical page frame. It records if the physical frame is in use and, if so, a pointer to the page table entry for the virtual page that is using it.

3. *pgdir_entry_t pgdir[PTRS_PER_PGDIR]*: We are using a two-level page table design; the top-level is referred to as the page directory, which is represented by this array. Each page directory entry (pde_t) holds a pointer to a second-level page table (which we refer to simply as page tables, for short). We use the low-order bit in this pointer to record whether the entry is valid or not. The page tables are arrays of page table entries (pte_t), which consist of a frame number if the page is in (simulated) physical memory and an offset into the swap file if the page has been written out to swap. The format of a page table entry is shown here: image of pte format

   Note that the frame number and status bits share a word, with the low-order *PAGE_SHIFT* bits (12 in our implementation) used for status (we only have 4 status bits, but you can add more if you find it useful). Thus, for a given physical frame number (e.g. 7), remember to shift it over to leave room for the status bits (e.g., 7 <<*PAGE_SHIFT*) when storing into the pte and to shift it back when retrieving a frame number from a pte (e.g., *p->frame* >>*PAGE_SHIFT*).



4. *swap.c*: The swapfile functions are all implemented in this file, along with bitmap functions to track free and used space in the swap file, and to move virtual pages between the swapfile and (simulated) physical memory. The *swap_pagein* and *swap_pageout* functions take a frame number and a swap offset as arguments. Be careful not to pass the frame field from a page table entry (pte_t) directly, since that would include the extra status bits. The simulator code creates a temporary file in the current directory where it is executed to use as the swapfile, and removes this file as part of the cleanup when it completes. It does not, however, remove the temporary file if the simulator crashes or exits early due to a detected error. You must manually remove the swapfile.XXXXXX files in this case.

To complete this task, you will have to write code in *pagetable.c*. Read the code and comments in this file – it should be clear where implementation work is needed and what it needs to do. The rand replacement algorithm is already implemented for you, so you can test your translation and paging functionality independently of implementing the replacement algorithms.

2.3 Task 2

Using the starter code, implement each of the four different page replacement algorithms: FIFO, exact LRU, CLOCK (with one ref-bit), OPT.

You will find that you want to add fields to the struct frame for the different page replacement algorithms. You can add them in *pagetable.h*, but please label them clearly.

Once you're done implementing the algorithms, run all three programs from the provided traceprogs, plus a fourth program of your choosing with interesting memory reference behaviour, using each of your algorithms (include rand as well). For each algorithm, run the programs on memory sizes 50, 100, 150, and 200. Use the data from these runs to create a set of tables that include the following columns. (Please label your columns in the following order,)

- Hit rate
- Hit count
- Miss count
- Overall eviction count
- Clean eviction count
- Dirty eviction count

2.4 Write up

Include a file called README.pdf that includes the following information.

- The tables prepared in Task 2
- One paragraph comparing the various algorithms in terms of the results you see in the tables.
- A second paragraph explaining the data you obtained for LRU as the size of memory increases.

Notes:

- First-In First-Out (FIFO)
  - Is the simplest page replacement algorithm [1]
  - FIFO suffers from **Belady's Anomaly**

Refernces:

1) Geeks for Geeks: Page Replacement Algorithm in Operating Systems, link