CSCI4730/6730 – Operating Systems

Project #2

Due date: 11:59pm, 10/31/2015

Description

In this project, you will implement a virtual memory simulator in order to understand the behavior of a page table and the page replacement algorithms. Virtual memory allows the execution of processes that are not completely in memory. This is achieved by page table and page replacement techniques. A page table stores the mapping between virtual addresses and physical addresses. A page fault mechanism by which the memory management unit (MMU) can ask the operating system (OS) to bring in a page from the disk. The system we are simulating is 16-bit machine (8 bits for index and 8 bits for offset).

Part 1: Page Table - 50%

The page table structure that you will simulate is a linear page table. Each page table entry should contain a valid bit, a physical page number, and a dirty bit. The size of main memory on the machine you are simulating is 256 pages.

Initially, physical memory is empty and a free-frame list contains every physical page. Your page allocation policy should simply hand out the physical pages in order of increasing page number, until the free-frame list is empty. From that point on, all physical pages will be obtained by page replacements algorithm. In our simulation, pages will never be added back to the free-frame list (processes never terminate).

We do not model the actual placement of virtual pages on the disk. Instead, we simply keep track of the number of disk read and write operations that are needed to handle the page faults.

- Data structure for the page table entry and statistics (hit and miss count) are defined in pagetable.h
- You will need to implement “hit_test()” and “pagefault_handler()” functions in pagetable.c file
- You will need to call “disk_read” to read a page from the disk into the main memory, and “disk_write” to write out a dirty page to the disk.
- If the physical memory is full, you will need to call “page_replacement()” to find a victim page.

Part 2: Page Replacement Algorithm – 50%

The current simulator only supports random page replacement algorithm that randomly choose a victim page.
In this part, you will implement three page replacement algorithms, First-in-first-out (FIFO), Least-recently-used (LRU) and Least-frequently-used (LFU).

- LRU, LFU and FIFO functions are declared in "replacement.c". You will need to fill out the body of each function.
- The simulator requires a command-line argument to specify a replacement policy to use.
    - ./vm 0 : random
    - ./vm 1 : FIFO
    - ./vm 2 : LRU
    - ./vm 3 : LFU

Input and Output

The input format for your simulator will be in the following format:

pid R/W virtual_address

You may use a provided input generation tool "input_gen" to generate random inputs.

The output of your simulation should be the result of each memory request: "Hit/Miss: original_request -> physical_address".
At the end of a request sequence, the total number of request, the total numbers ofhit and miss in pagetable, and the total numbers of disk read and write should be printed. A code for the output is already in the simulator and you will need to maintain hit and miss count.

Example of output:
./vm 0 < input.txt
Replacement Policy: 0 - RANDOM
Miss: [2] R 0x107 -> 0x7
Hit: [2] W 0x129 -> 0x29
Miss: [1] W 0x256 -> 0x156
Miss: [0] W 0x19c -> 0x29c
Hit: [0] W 0x1e8 -> 0x2e8
Hit: [2] R 0x123 -> 0x23
Miss: [2] W 0x7a -> 0x37a
Miss: [3] W 0xcb -> 0xcb
Miss: [2] R 0x1aa -> 0x3aa
Hit: [2] W 0x19f -> 0x39f
Miss: [0] R 0xcd -> 0x1cd

Miss: [1] W 0x207 -> 0x7
Miss: [3] W 0x3cd -> 0xcd
Miss: [1] W 0x30e -> 0x20e
Miss: [3] R 0x2f1 -> 0x1f1
Hit: [3] W 0x260 -> 0x160
==========
Request: 16
Hit: 5 (31.25%)
Miss: 11 (68.75%)
Disk read: 11
Disk write: 6

Submission

Submit a tarball file using the following command
%tar czvf p2.tar.gz README Makefile *.c *.h

1. README file with:
    a. Your name
    b. List what you have done and how to test them. So that you can be sure to receive credit for the parts you've done.
    c. Explain your design of data structures.
2. All source files needed to compile, run and test your code
    a. Makefile
    b. All source files
    c. Do not submit object or executable files
3. Your code should be compiled in cf0-cf11 machine (cf0.cs.uga.edu – cf11.cs.uga.edu)
4. Submit a tarball through ELC.