CSCI4730/6730 – Operating Systems

Project #3

Due date: 11:59pm, 4/12/2017

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 – (Grad: 35%, Undergrad: 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 (physical memory) and the size of virtual memory are configurable in "vm.h" (#define MAX_FRAME, #define MAX_PAGE). The machine you are simulating supports up to 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– (Grad: 35%, Undergrad: 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 Second-chance (Clock) algorithms.

- LRU, Clock 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 : Clock Algorithm

Part 3: Translation Look-aside Buffer (TLB) – (Grad: 30%, Undergrad: extra 30%)

Page table (part 1) is located in the main memory. If we want to access location *i*, we must first index into the page table and it requires a memory access. It provides us with the frame number, which is combined with the page offset to produce the actual address. We can then access the desired place in memory. With this scheme, two memory accesses are needed to access a byte (one for the page-table entry, one for the byte). Thus, memory access is slowed by a factor of 2.
To address this problem, we use translation look-aside buffer (TLB). It is associative, high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value. When the associative memory is presented with an item, the item is compared with all keys simultaneously. If the item is found, the corresponding value field is returned. The search is fast; a TLB lookup in modern hardware is part of the instruction pipeline, essentially adding no performance penalty.
In this part, you will implement **two-way associative TLB** system. The number of TLB entry is configurable in “vm.h” (#define TLB_ENTRY). The system supports up to 16 TLB entries and the replacement algorithm of our TLB is **least-recently-used (LRU).**
For the simplicity, we assume that each process has own TLB.
Note that the *TLB set = (PAGE_NUMBER % (#_of_TLB_ENTRY / 2)).*

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. It reads number of process, number of virtual pages, and number of physical frames from "vm.h".

The output of your simulation should be the result of each memory request: *"[PID, R/W] TLB:hit/miss, Page:hit/miss, original_request -> physical_address".*
At the end of a request sequence, the total number of request, the total numbers of hit and miss in pagetable and TLB, 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, miss, disk read and disk write count.

Example of output (4 TLB entries, 8 virtual pages, 8 physical frames):

Replacement Policy: 1 - FIFO
[pid 0, W] TLB:miss, Page:miss, 0x4a8 -> 0xa8
[pid 0, W] TLB:miss, Page:miss, 0x7ae -> 0x1ae
[pid 1, W] TLB:miss, Page:miss, 0x303 -> 0x203
[pid 1, W] TLB:miss, Page:miss, 0x42e -> 0x32e
[pid 1, R] TLB:miss, Page:miss, 0x6ec -> 0x4ec
[pid 1, R] TLB:hit, Page:hit, 0x439 -> 0x339
[pid 1, W] TLB:miss, Page:miss, 0x7d0 -> 0x5d0
[pid 1, W] TLB:miss, Page:miss, 0x16 -> 0x616
[pid 0, R] TLB:hit, Page:hit, 0x742 -> 0x142
[pid 0, W] TLB:miss, Page:miss, 0x6e3 -> 0x7e3
[pid 1, W] TLB:miss, Page:miss, 0x5eb -> 0xeb
[pid 1, R] TLB:miss, Page:hit, 0x38e -> 0x28e
[pid 1, R] TLB:miss, Page:hit, 0x7bc -> 0x5bc
[pid 1, R] TLB:miss, Page:miss, 0x11e -> 0x11e
[pid 0, W] TLB:miss, Page:miss, 0x3e5 -> 0x2e5
[pid 1, W] TLB:hit, Page:hit, 0xac -> 0x6ac
[pid 1, W] TLB:miss, Page:hit, 0x52e -> 0x2e
[pid 0, W] TLB:miss, Page:miss, 0x471 -> 0x371
[pid 1, W] TLB:miss, Page:hit, 0x6a6 -> 0x4a6
[pid 1, R] TLB:hit, Page:hit, 0xd9 -> 0x6d9
==========
Request: 20
Page Hit: 8 (40.00%)
Page Miss: 12 (60.00%)
TLB Hit: 4 (20.00%)
TLB Miss: 16 (80.00%)
Disk read: 12
Disk write: 4

Submission

Submit a tarball file using the following command
%tar czvf p3.tar.gz README Makefile *.c *.h

1. README file with:
    a. Your name
    b. List what you have done and how tested 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.