CPSC 545 Computing Systems

Project 4: Virtual Memory Simulation

Due by 6:25 PM, June 6, 2011

1. Goals

In this project, you will apply fundamental concepts such as page tables, page faults, demand paging, page replacement to strengthen understanding of how virtual memory works. In addition, you will use shared memory and signals for process communication.

2. Description

Virtual memory allows a process of ***P*** pages to run in ***F*** frames, even if ***F < P***. This is achieved by use of a page table, which records which pages are in RAM in which frames, and a page fault mechanism by which the memory management unit (MMU) can ask the operating system (OS) to bring in a page from disk. The page table must be accessible by both the MMU and the OS, and an IPC facility is needed for communication between the MMU and OS. Given the MMU simulator described below, the work of the OS is to maintain a process' use of RAM and the page table. The page table is held in shared memory, and signals are used for IPC. You do not need to simulate the RAM, only the page table maintenance. However, to make things realistic, your OS process must `sleep(1)` whenever a disk access (write out page to disk, read in page from disk) would be necessary in a real implementation.

2.1 Page table

The page table has four fields in each page table entry:

```
struct page_table_entry {
    bool Valid;
    int Frame;
    bool Dirty;
    int Requested;
};
```

- A Boolean Valid indicating if the page of that index is in RAM.
- An integer Frame giving the frame number of the page in RAM.
- A Boolean Dirty indicating if the page has been written to.
- An integer Requested which is non-zero only if that page is not in RAM and has been requested by the MMU. In this case its value is the PID of the MMU.

The OS process must create the page table in shared memory, and initialize it to indicate that no pages are loaded (all Valid fields set to 0). You may need to add more fields for your OS, with corresponding changes in the MMU.

2.2 MMU

The MMU executes as a separate process, taking several arguments:

- The number of pages in the process.
- A reference string of memory accesses, each of the form <mode><page>, e.g., W3 is a write to page 3.
- The PID of the OS process.

The MMU attaches to the shared memory (using the OS PID on the command line as the key), then runs through the reference string. For each memory access, the MMU:

1. Checks if the page is in RAM.
2. If not in RAM, writes its PID into the Requested field for that page.
3. Simluates a page fault by signalling the OS with a SIGUSR1.
4. Blocks until it receives a SIGCONT signal from the OS to indicate that the page has been loaded (well, as mentioned above, the load is not done in this project, just simulated by sleep(1) delays).
5. If the access is a write access, sets the Dirty bit.
6. Prints the updated page table.

When all memory accesses have been processed, the MMU detaches from the shared memory and signals the OS one last time, but without placing its PID in any Requested field. That must be detected by the OS, at which point it can destroy the shared memory and exit.

2.3 OS

The OS simulator must take two arguments:

- The number of pages in the process.
- The number of frames allocated to the process.

For simplicity, assume that the pages and frames are numbered 0, 1, 2, ...

The OS manages free frames.

After creating and initializing the page table in the shared memory, the OS must sit in a loop waiting for a SIGUSR1 signal from the MMU. When it receives a signal, it must:

1. Scan through the page table looking for a non-zero value in the Requested field.
2. If a non-zero value is found, it's the PID of the MMU, and indicates that the MMU wants the page at that index loaded.
3. If there is a free frame allocate the next one to the page.
4. If there are no free frames, choose a victim page using LRU replacement.
    - If the victim page is dirty, simulate writing the page to disk by sleep(1) and increment the counter of disk accesses.
    - Update the page table to indicate that the victim page is no longer Valid.
5. Simulate the page load by sleep(1) and increment the counter of disk accesses.
6. Update the page table to indicate that the page is Valid, in the allocated Frame, not Dirty, and clear the Requested field.

7. Print the updated page table.
8. Send a SIGCONT signal to the MMU to indicate that the page is now loaded.
9. If no non-zero Requested field was found, the OS exits the loop.

You must use LRU algorithm for choosing a victim page. You may need to add extra fields to the page table. Before terminating the OS must print out the total number of disk accesses, and destroy the shared memory.

2.4 Requirements

You are required to complete the two programs MMU (e.g., mymmu.cpp) and OS (e.g., myos.cpp). Here's a sample run (Note for correctness purpose, just for illustration purpose). Your program execution outcome should be like this. I have spaced things out so you can see the sequence of events in time.

```
> OS 5 2
The shared memory key (PID) is 78801
Initialized page table
                                    > MMU 5 W2 R3 W3 R4 78801
                                    Initialized page table:
                                     0: Valid=0 Frame=-1 Dirty=0 Requested=0
                                     1: Valid=0 Frame=-1 Dirty=0 Requested=0
                                     2: Valid=0 Frame=-1 Dirty=0 Requested=0
                                     3: Valid=0 Frame=-1 Dirty=0 Requested=0
                                     4: Valid=0 Frame=-1 Dirty=0 Requested=0

                                    Request for page 2 in W mode
                                    It's not in RAM - page fault
Process 78803 has requested page 2
Put it in free frame 0
Unblock MMU
                                    Set the dirty bit for page 2
                                     0: Valid=0 Frame=-1 Dirty=0 Requested=0
                                     1: Valid=0 Frame=-1 Dirty=0 Requested=0
                                     2: Valid=1 Frame= 0 Dirty=1 Requested=0
                                     3: Valid=0 Frame=-1 Dirty=0 Requested=0
                                     4: Valid=0 Frame=-1 Dirty=0 Requested=0

                                    Request for page 3 in R mode
                                    It's not in RAM - page fault
Process 78803 has requested page 3
Put it in free frame 1
Unblock MMU
                                     0: Valid=0 Frame=-1 Dirty=0 Requested=0
                                     1: Valid=0 Frame=-1 Dirty=0 Requested=0
                                     2: Valid=1 Frame= 0 Dirty=1 Requested=0
                                     3: Valid=1 Frame= 1 Dirty=0 Requested=0
                                     4: Valid=0 Frame=-1 Dirty=0 Requested=0
```

Request for page 3 in W mode
It's in RAM
Set the dirty bit for page 3
0: Valid=0 Frame=-1 Dirty=0 Requested=0
1: Valid=0 Frame=-1 Dirty=0 Requested=0
2: Valid=1 Frame= 0 Dirty=1 Requested=0
3: Valid=1 Frame= 1 Dirty=1 Requested=0
4: Valid=0 Frame=-1 Dirty=0 Requested=0

Request for page 4 in R mode
It's not in RAM - page fault

Process 78803 has requested page 4
Chose a victim page 2
Victim is dirty, write out
Put in victim's frame 0
Unblock MMU
0: Valid=0 Frame=-1 Dirty=0 Requested=0
1: Valid=0 Frame=-1 Dirty=0 Requested=0
2: Valid=0 Frame=-1 Dirty=0 Requested=0
3: Valid=1 Frame= 1 Dirty=1 Requested=0
4: Valid=1 Frame= 0 Dirty=0 Requested=0
Tell OS that I'm finished
The MMU has finished
0: Valid=0 Frame=-1 Dirty=0 Requested=0
1: Valid=0 Frame=-1 Dirty=0 Requested=0
2: Valid=0 Frame=-1 Dirty=0 Requested=0
3: Valid=1 Frame= 1 Dirty=1 Requested=0
4: Valid=1 Frame= 0 Dirty=0 Requested=0
4 disk accesses required

2.5 Some test cases

- 5 pages, 3 frames
  R0 R2 R1 W3 R0 R2 R1 W4 R0 R2 R1 W3 R0 R2 R1 W4 R0 R2 R1 W3 R0 R2 R1 W4
- 4 pages, 3 frames
  R0 R0 R1 R0 R1 R2 R0 R1 R2 R3 R0 R1 R2 R0 R1 R0 R0 R0 R1 R0 R1 R2 R0 R1 R2 R3 R0 R1 R2 R0 R1 R0
- 4 pages, 2 frames
  R0 R1 W1 R0 R2 W2 R0 R3 W3 R0 R1 W1 R0 R2 W2 R0 R3 W3 R0 R1 W1 R0 R2 W2 R0 R3 W3
- 5 pages, 3 frames
  R0 R1 R1 W3 R0 R2 R2 W4 R0 R2 R2 W4 R0 R2 R2 W4 R0 R1 R1 W3 R0 R1 R1 W3
- 4 pages, 3 frames
  R0 R1 R2 R0 R1 R2 R3 R1 R2 R3 R1 R2 R3 R0 R2 R3 R0 R2 R3 R0 R1 R3 R0 R1 R3 R0 R1 R2 R0 R1 R2 R0 R1 R2

- 5 pages, 3 frames
  W0 R1 R2 R0 R3 W1 R2 R3 R4 R1 R2 W4 R1 R2 R0 R1 R2 R0 R1 R2 W0 R1 R3 W1
- 5 pages 4 frames
  W0 R1 R2 R0 R3 W1 R2 R3 R4 R1 R2 W4 R1 R2 R0 R1 R2 R0 R1 R2 W0 R1 R3 W1
- 5 pages 3 frames
  R0 R1 R0 W1 R0 R1 R0 W1 R0 R2 R0 W2 R0 R2 R0 W2 R0 R3 R0 W3 R0 R3 R0 W3 R0 R4 R0 W4 R0 R4 R0 W4

2.6 Some system calls

In this project, you need to be familiar with some signal system calls, such as kill(), sigaction(), pause(), sleep(). Differentiate pause() and sleep() when using them.

3. Documentation

You *must* include a README file which describes your program. It needs to contain the following:

- The purpose of your program
- How to compile the program
- How to use the program from the shell (syntax)
- What exactly your program does

The README file does not have to be very long, as long as it properly describes the above points. Proper in this case means that a first-time user will be able to answer the above questions without any confusion.

Within your code you should use one or two sentences to describe each function that you write. You do not need to comment every line of your code. However, you might want to comment portions of your code to increase readability.

At the top of your README file and main C source file please include the following comment:

```
/* CPSC545  Spring2011 Programming Assignment 1
* login: cs1_login_name (login used to submit)
* Linux
* date: mm/dd/yy
* name: full_name1
* emails: your emails */
```

4. Grading

5% README file
20% Documentation within code, coding, and style (indentations, readability of code, use of defined constants rather than numbers)
75% Test cases (correctness, error handling, meeting the specifications)

- Please make sure to pay attention to documentation and coding style. A perfectly working program will not receive full credit if it is undocumented and very difficult to read.
- The test cases will not be given to you upfront. They will be designed to test how well your program adheres to the specifications. So make sure that you read the specifications very carefully. If there is anything that is not clear to you, you should ask for a clarification.
- Make sure your code compiles and run on cs1.seattleu.edu. Failures to compile and execute you program result in zero in test cases.

5. Deliverables

The following deliverables must be submitted:

- Source code files
- A README file
- A Makefile that will compile your code and produce two program called myos and mymmu Note: this Makefile will be used by us to compile your program with the make utility.

All files should be made a package by tar and submitted using the SUBMIT utility. This is your official submission that we will grade. Please note that future submissions under the same homework title OVERWRITE previous submissions; we can only grade the most recent submission. You can NOT submit your code after the deadline is passed.

6. Submission

You should connect to cs1.seattleu.edu using your account to submit your project deliverables. You can submit multiple times before the deadline. The system keeps the most recent submission. You should follow the steps to submit your project:

- Make a tar package of your deliverables (one example below)
  tar -cvf p4.tar README Makefile myos.cpp mymmu.cpp myos.h mymmu.h
- Submit your tar package
  /home/fac/zhuy/CPSC545/submit p4 4.tar