**CS 418 – Operating Systems**

Lab #3: Virtual Memory Management, Demand Paging & Delayed Writes

**Due:** (**L01**) Tue, 1 Dec 2015 (before Lesson #12)

**Directions**

Please submit your answers via the course website – alternate submissions via e-mail, etc. should be absolutely last resort. If you have to attach any files, then name them using the following format:

* CS418\_<your section>\_<your first initial + last name>\_<description of contents>.<file type>

Upload your solutions before 11:00 PM (2300 hours) EST on the given due date. The lateness penalties are listed in the Course Syllabus. Also, remember that a partial solution, even for reduced points, helps your overall course grade much more than no submission at all.

**Textbook Exercises**

Silberschatz, Galvin & Gagne Essentials: 7.28 & 8.3 **[**10 pts each**]**; 7.11 & 8.30 **[**15 pts each**]**

**Programming Project – Virtual Memory Management [**100 points**]**

We’ve discussed the basic paging mechanisms, where the operating system translates logical address requests into actual physical addresses for multiple concurrent processes. ***Your task is to implement the mechanisms needed to keep track of the locations of pages between frames in memory; swap space on the backing store; and, in the original files on disk.***  This project is similar to Programming Problem 9.40, and the Chapter 9 Programming Project: Designing a Virtual Memory Manager.

**Supporting Multiple Processes, Pages and Frames**

Your system must keep track of an arbitrary number of processes, pages and frames. You will be provided a shell program that provides a basic interactive loop that accepts and interprets commands from the user. Also, you will be given certain information to define the memory space, including the size of the address space (ADDRESS\_SPACE\_SIZE) and of an individual page (PAGE\_SIZE); maximum number of processes (NUM\_PROCESSES); and, the number of frames in memory (NUM\_FRAMES). Each active process can have as many pages as can fit into the address space. The swap space and disk are arbitrarily large and can hold as many pages as needed, while each frame can hold one page at a time.

**Memory Management Operations**

Your system must support five basic commands: ***begin()***, ***read()***, ***write()***, ***end()*** and ***view()***. Also, the results shown below are from a correctly implemented solution – the shell you have been given doesn’t implement any of these functions yet. The commands needed to compile and execute the shell program are:

**capitol@ubuntu:~$ gcc -o frog cs418\_lab3\_virtualmem\_shell\_v4.c**

**capitol@ubuntu:~$ ./frog**

**sys !> main: memory manager started**

**request: >**

For the following examples, the address space size is 100, page size is 10, and there are 4 memory frames. The ***begin()*** command is used to activate a process. A process might be activated before it can read or write pages. Examples of this command:

**request: > begin p3**

**request: > begin p5**

The ***read()*** command is used to access addresses within an active process’ address space. A page must be loaded into a valid memory frame before it can be accessed. Examples of this command:

**request: > read p3 7**

**move p3, page(0) from disk to frame(0)**

**reading p3, addr(7) from frame(0) in memory**

**request: > read p5 8**

**move p5, page(0) from disk to frame(1)**

**reading p5, addr(8) from frame(1) in memory**

**request: > read p2 33**

**sys !> unable to access active process**

**sys !> unable to read the requested page**

**request: > read p3 27**

**move p3, page(2) from disk to frame(2)**

**reading p3, addr(27) from frame(2) in memory**

The ***write()*** command is used to modify the content/data at addresses within an active process’ address space. A page must be loaded into a valid memory frame before it can be modified. Also, there are two techniques for handling writes: immediate or delayed writes. With the immediate write technique, changes to a page are written back to the disk immediately. While this makes management simpler in some ways, it can also be inefficient when there are multiple writes to the same page. A potential (much) more efficient technique is to delay writing modified pages back to disk until the process is ended. An example of the immediate write technique:

**request: > write p5 91**

**move p5, page(9) from disk to frame(3)**

**writing p5, addr(91) to frame(3) in memory**

**move p5, page(9) from frame(3) to disk**

The ***page(9)*** of process ***p5*** is moved back to disk immediately after writing the modified data. This immediate move is not required if you chose to use the delayed write technique, which will be demonstrated more clearly below when ending a process.

The ***end()*** command is used to inactivate a currently active process. Resources in use should be recycled, including memory frames. Also, any pages that have been written must be moved back to disk if the delayed write technique is being used. An example of this command, along with the delayed write technique, is shown in this (restarted) session, where process ***p4*** has accessed 7 different pages, and written to five of them: namely, ***pages(0, 1, 2, 3, 5)***:

**capitol@ubuntu:~$ ./frog**

**sys !> main: memory manager started**

**request: > begin p4**

**request: > write p4 3**

**move p4, page(0) from disk to frame(0)**

**writing p4, addr(3) to frame(0) in memory**

**request: > write p4 13**

**move p4, page(1) from disk to frame(1)**

**writing p4, addr(13) to frame(1) in memory**

**request: > write p4 23**

**move p4, page(2) from disk to frame(2)**

**writing p4, addr(23) to frame(2) in memory**

**request: > write p4 35**

**move p4, page(3) from disk to frame(3)**

**writing p4, addr(35) to frame(3) in memory**

**request: > write p4 56**

**move p4, page(0) from frame(0) to swap space**

**move p4, page(5) from disk to frame(0)**

**writing p4, addr(56) to frame(0) in memory**

**request: > read p4 61**

**move p4, page(1) from frame(1) to swap space**

**move p4, page(6) from disk to frame(1)**

**reading p4, addr(61) from frame(1) in memory**

**request: > read p4 77**

**move p4, page(2) from frame(2) to swap space**

**move p4, page(7) from disk to frame(2)**

**reading p4, addr(77) from frame(2) in memory**

**request: > view**

**--- active processes ---**

**[3]: pid: 4**

**--- active pages ---**

**[30]: loc: swap, dirty [31]: loc: swap, dirty [32]: loc: swap, dirty [33]: loc: frame(3), dirty [35]: loc: frame(0), dirty [36]: loc: frame(1) [37]: loc: frame(2)**

**--- active frames ---**

**[0]: page\_ref: 35 [1]: page\_ref: 36 [2]: page\_ref: 37, cbs [3]: page\_ref: 33**

The ***view()*** command displays the internal data that is being used to track the memory metadata, including status information such as the dirty tag for pages that have been modified. It also shows that ***pages(0, 1, 2)*** are being stored in swap space, and will need to be reloaded into memory and moved back to disk when process ***p4*** is ended. There is no specifically-required format for the ***view()*** output: it will depend on how you decide to implement your internal data structures.

**request: > end p4**

**move p4, page(3) from frame(3) to swap space**

**move p4, page(0) from swap space to frame(3)**

**move p4, page(0) from frame(3) to disk**

**move p4, page(1) from swap space to frame(3)**

**move p4, page(1) from frame(3) to disk**

**move p4, page(2) from swap space to frame(3)**

**move p4, page(2) from frame(3) to disk**

**move p4, page(3) from swap space to frame(3)**

**move p4, page(3) from frame(3) to disk**

**move p4, page(5) from frame(0) to disk**

As shown above, the system writes all of the dirty pages back to disk once the process is ended. Also, since all frames were in use, the system had to select a victim page to be moved to swap space to be able to move ***pages(0, 1, 2)*** back to disk. It selected ***page(3)*** in ***frame(3)*** as the victim page, which was also dirty, and then later reloaded and flushed ***page(3)*** back to disk as well.

This system is employing the Second-Chance (Clock) page replacement algorithm as given in Chapter 9.4 of the text. Your system is permitted to implement any of the other valid page replacement algorithms from the textbook to manage page faults and victim selection.

The ***view()*** command is really reserved for your use: you should use this command to display the underlying data structures that you have selected for your implementation. This will help you when troubleshooting your project. I won’t grade your ***view()*** command for any specific format or requirements.

**Grading and Other Helpful Suggestions**

These are my guidelines for grading your memory management program:

|  |  |
| --- | --- |
| Grade | Capability of the Shell Program |
| **A** | • Implement ***write()*** requests with the delayed write protocol  • Design your program to work correctly even if the ADDRESS\_SPACE\_SIZE, PAGE\_SIZE, NUM\_PROCESSES and/or NUM\_FRAMES values are modified |
| **B** | • Implement ***read()*** and ***write()*** requests, and ensure pages are loaded into an available frame with the immediate or delayed write protocol  • Implement an approved page replacement algorithm from the text |
| **C** | • Implement ***read()*** requests and ensure pages are loaded into an available frame |
| **D** | • Translate memory addresses into page numbers  • Implement the ***view()*** command |

**Testing Your Program**

Your main goal is for your program to maintain a consistent state for paging tables, and process and frame information. I will test your system using various sequences of ***read()*** and ***write()*** requests intended to force page faults, and ensure that modified pages are written back to disk. Good luck, and don’t hesitate to seek assistance as needed!