ICS 53, Winter 2021

Assignment 3: Virtual Memory Simulator

You will implement a VM Simulator which will simulate the operation of a virtual memory system. To launch the system, your simulator will accept one optional command-line argument for the selection of the page replacement algorithm. The system will then accept commands to read/write from/to a virtual address space. Your system will need to correctly move pages between disk and main memory in order to satisfy access requests. Your system will need to properly maintain the page table as part of this process.

3.1 Parameters of the Virtual Memory System

Each address in memory stores a single integer. All memory locations are initialized to the value of -1. All virtual pages are initially on disk, so the valid bits of all page table entries are equal to 0.

Your virtual memory system will have 64 addresses (0 - 63) and main memory has 32 addresses (0 - 31). Each page contains 8 addresses, so your virtual memory system will have 8 pages and your main memory will have 4 pages. Pages are numbered sequentially starting at the lowest memory address. So page 0 contains addresses 0-7, page 1 contains addresses 8-15, and so on. And the disk page number of each virtual page is the same as the virtual page number.

3.2 User Interface of VM Simulator

The user will type in a sequence of commands and your program will perform the operation specified by each command. Your program will execute in a loop which starts with your program printing a "> " prompt at the beginning of the line in order to let the user know that your program is ready to receive a new command. The user will type in a command, followed by any necessary arguments, and then hit the 'enter' key to indicate that the command is complete. Your program will then execute the command, print data to the screen if necessary, print a "> " prompt on a new line to receive a new command. (Note: "> " is one > and one space.)

3 Commands

Your program must (and will only) process the following user commands.

1. **read <virtual_addr>**: This command prints the contents of a memory address. The command takes one argument which is the virtual address of the memory location to be read. When a page fault occurs, "A Page Fault Has Occurred\n" is printed to the screen before the contents of a memory address.

2. **write <virtual_addr> <num>**: The command writes data to a memory location. The command takes two arguments, the virtual address to write the data to, and an integer which will be written to the address. When a page fault occurs, “A Page Fault Has Occurred\n” is printed on the screen.

3. **showmain <ppn>**: This command prints the contents of a physical page in the main memory. The command takes one argument which is the number of the physical page to be printed. Since each page contains eight addresses, eight contents of addresses should be printed and together with their associated physical addresses. You can see the format from the example below which shows the contents of physical page 1 that includes the value 100 at address 8, the value 101 at address 9, the value 102 at address 10, and so on.

8:100
9:101
10:102
11:103
12: 105
13: 107
14: 101
15: 100

4. **showdisk <dpn>**: This command prints the contents of a page on disk. The command takes one argument which is the number of the disk page to be printed. Since each page contains eight addresses, eight contents of addresses should be printed and together with their associated addresses. The format for printing the contents of the disk page is the same as the **showmain** command.

5. **showptable**: This command prints the contents of the page table. Your virtual memory system contains 8 virtual pages, so this command will print 8 page table entries. Each page table entry contains three fields of information about a page.

- *Valid bit*: 1 indicates that the corresponding page is in main memory and 0 indicates that the page is on disk.

- *Dirty bit*: 1 indicates that the corresponding page has been written to while in main memory. 0 indicates that the page has not been written to since it has been in main memory. The dirty bit has no meaning if the page is not in main memory, i.e. if the Valid bit is 0.

- *Page Number (PN)*: This is an integer indicating the number of the page (in main memory or disk) where the virtual page can be found in memory. The PN refers to a main memory physical page when the Valid bit is equal to 1, and the PN refers to a disk page when the valid bit is 0.

The **showptable** command will print the contents of each page table entry on a separate line in the following format, where each line is divided by colons with four entries, virtual page number, *Valid bit*, *Dirty bit*, and *PN*. Below is an example output of a page table from the **showptable** command. For instance, the first line represents that "At virtual page number 0, *Valid bit* is 1 and *Dirty bit* is 0, and *Page Number* is 0."

0:1:0:0
1:1:1:3
2:0:0:2
...

Although the above example only shows 3 pages for brevity, your **showptable** command should print all 8 pages from the page table.

6. **quit**: This command quits the program. (Note: No need to print anything.)

3.4 Page Replacement Algorithm (Handling page faults)

When a page fault occurs, a disk page must be copied into the main memory. If there is one or more pages in main memory which is available (there is not a virtual page mapped to it) then the disk page should be copied into the available main memory page with the lowest page number. If all pages in main memory are in use, then a victim page must be chosen for eviction from main memory.
You will implement two page replacement algorithms, FIFO and LRU, for the virtual memory system. And the page replacement algorithm will be selected at startup.

- **FIFO**: The simplest algorithm is First-In-First-Out replacement. When a page fault occurs, the page that has been the longest in the main memory will be replaced.
- **LRU**: The near-optimal algorithm is Least Recently Used replacement. When a page fault occurs, the page that is the least recently used/accessed in the main memory will be replaced. *show* commands do not count as use.*

Remember that the victim page must be copied back to disk before it is evicted if its own dirty bit is 1. When a victim page is copied back to disk, always copy it back to the disk page whose number is the same as the number of the virtual page.

Your simulator should accept EITHER "**FIFO**" or "**LRU**" as the command-line argument OR no command-line argument, which by default adopts the FIFO algorithm. The following examples show how your simulator should launch the system. We assume that the linux prompt is the symbol "$ " and the executable is named "a.out".

$ ./a.out	The system, by default, adopts the FIFO replacement algorithm.

$ ./a.out FIFO	The system adopts the FIFO replacement algorithm.
$ ./a.out LRU	The system adopts the LRU replacement algorithm

Additional Note:

For the sake of simplicity, the page table will be assumed to be stored separately from the main memory. This means that when you issue the **showmain** command you will not see the page table contents. Also, when you issue the **showptable** command you will not see the main memory.

For keeping counters please do not use `time(NULL)` because it will cause issues when running with autoCoverage.py. Use a global counter as a clock instead which increments by 1 each time you issue any command. Assume maximum number of commands will be < 255.

3.5 Testing instructions:

For this assignment you will need to create a set of tests for your program which collectively achieve 100% line coverage, executing every line of code in your program. You will need to submit your test data so that we can verify that you have achieved 100% line coverage. We have created a Python script called ***autocov.py*** which you should download with the assignment and use before you submit your assignment in order to ensure that you have achieved 100% line coverage. The autocov.py script will compile and execute your code with a set of test cases which you define and it will report the line coverage achieved by the test case. Do not modify the autocov.py script which we provide.

Test Runs

Testing is performed by executing your program a number of times with different test data. Each test execution of your code is referred to as a test run ("run" for short) and the test data associated with each run must be stored in a .run file. A .run file will contain all of the test data which will be supplied to your program during a single test execution. There are two types of test data which can be supplied to your program, command-line inputs and interactive inputs. Command-line inputs are those supplied at the command line when the program is invoked, and interactive inputs are those supplied during the execution of the program through stdin, typically from the keyboard.

A .run file is a text file describing all of the inputs supplied to your program for a single execution. The name of each .run file must have the suffix ".run", but the prefix does not matter. Each line of a .run file can be no longer than 80 characters. The first line of the .run file is a series of command-line arguments passed to the program when it is invoked. Each command-line argument on the first line of the .run file must be separated by at least one space or tab character. The remaining lines of the

.run files contain a sequence of interactive inputs which will be supplied to your program through stdin, as if they were typed into the keyboard.

Input to autocov.py

The autocov.py script requires your access to the source code of your program and the set of .run files to be used during testing. Your source code file must be named “hw.c”. This is important; it is the only file which will be compiled by the script. Any number of .run files may be provided. All of these files, hw.c and all of your .run files, must be “zipped” together into a single zip file named “hw.zip”. You can create the zip file using the “zip” command on linux. For example, if you have two .run files called “t1.run” and “t2.run” then you could create the zip file with the following command: “zip hw.zip hw.c t1.run t2.run”. The hw.zip file is the only input to the autocov.py script.

Running autocov.py

In order to execute the script, the autocov.py file must be placed in the same directory as the hw.zip file and another directory called “hw”. The hw directory will be used by the script to compile and run your program, so it should be empty before running the script. The script needs to be run using a Python interpreter version 3.6.8 or later. On the openlab systems the command “python3” invokes the Python 3.x interpreter, so you can run the autocov.py script by entering the command “python3 autocov.py”. The script will print a number of lines to the screen, including program outputs, but the second-to-last line should look like this, “Lines executed: X% of Y”, where X is the line coverage and Y is the number of lines in your program.

3.6 Submission Instructions:

Your source code and .run files **must** be submitted as a single .zip file. Your source code file must be named “hw.c”. This is important; it is the only file which will be compiled by the script. Any number of .run files may be provided. All of these files, hw.c and all of your .run files, must be “zipped” together into a single zip file named “hw.zip”. You can create the zip file using the “zip” command on linux. For example, if you have two .run files called “t1.run” and “t2.run” then you could create the zip file with the following command: “zip hw.zip hw.c t1.run t2.run”. Be sure that your program compiles on openlab.ics.uci.edu using gcc version 4.8.5. Also, be sure that no compiler switches are used, other than -o and the switches required for gcov.

Submissions will be done through Gradescope. You can follow this video to create a group assignment on Gradescope. The first line of your submitted file should be a comment which includes the name and ID number of you and your partner (if you are working with a partner).

3.7 Example Usage

Following the above assumption, this is an example, launching the system with the FIFO replacement algorithm. The text typed by the user is **bold**.

$ **./a.out FIFO**
> **showptable**
0:0:0:0
1:0:0:1
2:0:0:2
3:0:0:3
4:0:0:4
5:0:0:5
6:0:0:6
7:0:0:7
> **read 9**
A Page Fault Has Occurred
-1
> **write 9 201**
> **read 9**
201
> **showmain 0**
0:-1
1:201
2:-1
3:-1
4:-1
5:-1
6:-1
7:-1
> **showptable**
0:0:0:0
1:1:1:0
2:0:0:2
3:0:0:3
4:0:0:4
5:0:0:5
6:0:0:6
7:0:0:7
> **quit**
$ ←*you are back to the linux prompt.*

Additional Example Usage (eviction from main memory)

This is another example, launching the system with the LRU replacement algorithm. The text typed by the user is **bold**.

$ **./a.out LRU**
> **write 10 202**
A Page Fault Has Occurred
> **write 31 403**
A Page Fault Has Occurred
> **read 19**
A Page Fault Has Occurred
-1
> **read 0**
A Page Fault Has Occurred
-1
> **read 12**
-1
> **write 9 300**
> **showmain 1**
8:-1
9:-1
10:-1
11:-1
12:-1
13:-1
14:-1
15:403
> **showdisk 3**
24:-1
25:-1
26:-1
27:-1
28:-1
29:-1
30:-1
31:-1
> **showptable**
0:1:0:3
1:1:1:0
2:1:0:2
3:1:1:1

4:0:0:4
5:0:0:5
6:0:0:6
7:0:0:7
> **read 32**
A Page Fault Has Occurred
-1
> **showmain 1**
8:-1
9:-1
10:-1
11:-1
12:-1
13:-1
14:-1
15:-1
> **showdisk 3**
24:-1
25:-1
26:-1
27:-1
28:-1
29:-1
30:-1
31:403
> **showptable**
0:1:0:3
1:1:1:0
2:1:0:2
3:0:0:3
4:1:0:1
5:0:0:5
6:0:0:6
7:0:0:7
> **quit**
$ ←*you are back to the linux prompt.*