IE 411 (Winter 2020-21), Lab 5: Virtual Memory Simulation

This objective of this lab is to implement a simulation of virtual memory page replacement policy.

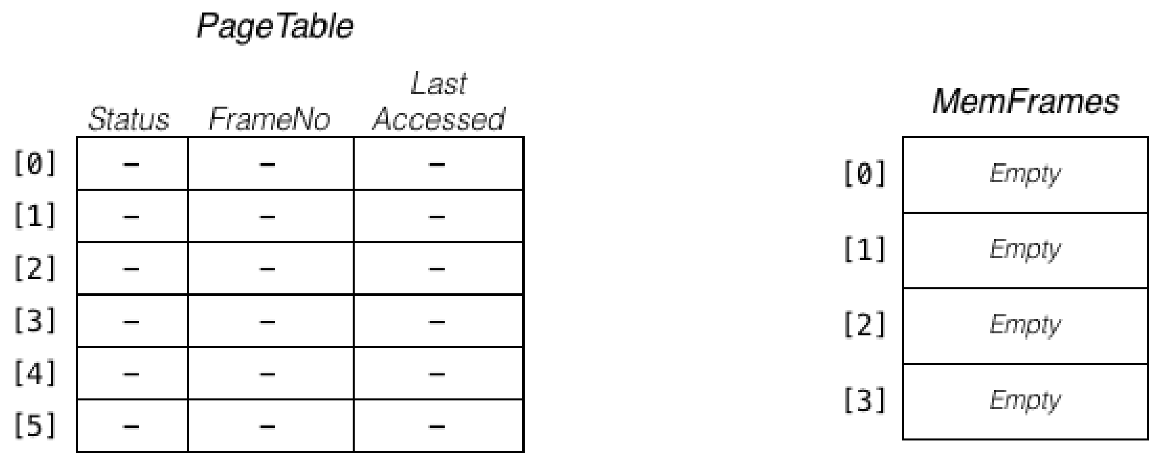
Getting checked off:

- You need to submit vmsim.c via classroom. This is due 19th April, 11:59 pm. Late submissions will not be accepted.
- After submitting the code, demonstrate your code to a TA in the next week's lab session, 20th April.

Background

Virtual memory is a technique for managing process memory and physical memory by (a) partitioning process memory and physical memory into fixed-size *pages*, (b) loading process pages into memory on demand, and (c) recording where they are loaded in a *page table* for each process. Pages in virtual memory are backed by *frames* of physical memory. A page table is an array of *page table entries*, or PTEs: each entry provides information on (a) where the process page occurs in the physical memory, (b) whether it has been modified since it was loaded, and (c) when it was last accessed (either read or written).

Consider the following simple virtual memory setup, where the process has 6 pages and the physical memory has 4 frames. When the process starts, all PTEs are empty as no process pages are loaded into memory. The following diagram shows the state at the start of the process:



PageTable

	Status	FrameNo	Last Accessed
[0]	–	–	–
[1]	–	–	–
[2]	–	–	–
[3]	–	–	–
[4]	–	–	–
[5]	–	–	–

MemFrames

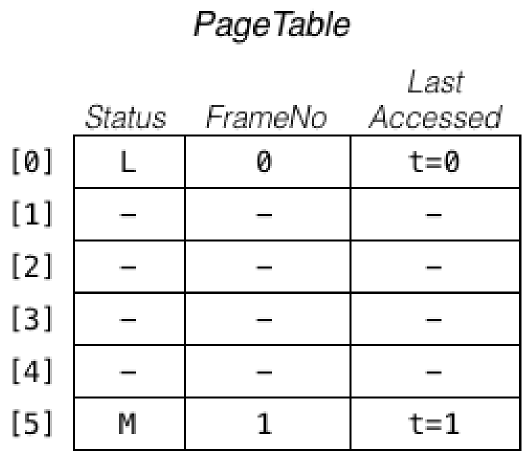
[0]	Empty
[1]	Empty
[2]	Empty
[3]	Empty

The first column in the page table gives the status of each page, which, here, is one of:

- `'-'` if the page is not currently loaded
- `'L'` if the page is loaded into memory but not modified
- `'M'` if the page is loaded and has been modified since loading

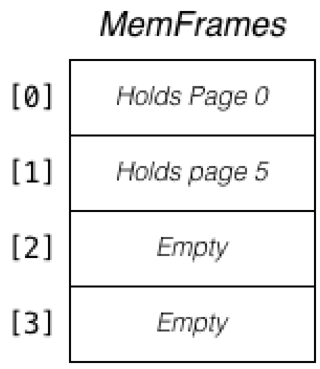
The second column in the page table is either `'-'` (if the page is not loaded), or is the number of the frame where the page is loaded. The third column is a timestamp indicating when the page was last read or written. Timestamps are simply integers starting from 0 and increasing by one each time an instruction is executed.

Consider the scenario where the first thing that the process does is to access its stack. It needs to have the startup code loaded (assume it's in page 0), and the stack data loaded (assume it's in page 5). The next diagram shows the state after this has happened. We assume that the code is loaded first, because it contains the instructions to access the stack.



PageTable

	Status	FrameNo	Last Accessed
[0]	L	0	t=0
[1]	-	-	-
[2]	-	-	-
[3]	-	-	-
[4]	-	-	-
[5]	M	1	t=1

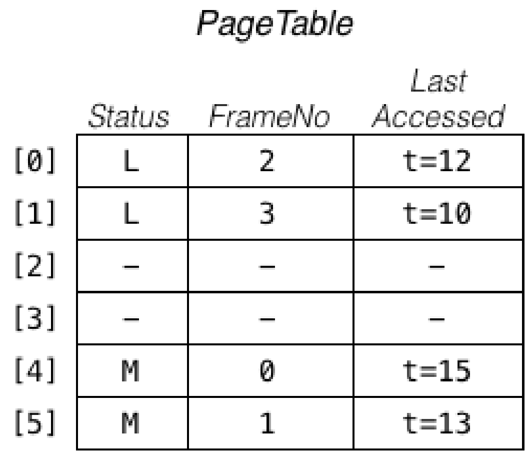


MemFrames

[0]	Holds Page 0
[1]	Holds page 5
[2]	Empty
[3]	Empty

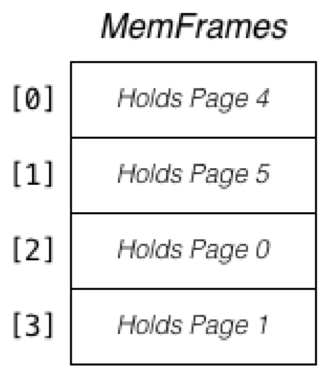
Note that the first page loaded is placed in the first free memory frame; the second page loaded is placed in the second memory frame; and so on. If the memory was large enough to hold all of the process's pages, they would be loaded one-by-one as required, and then stay in the memory until the process was finished.

Now consider the state later on in the computation, where the process has accessed a number of its pages and they are loaded in memory as follows:



PageTable

	Status	FrameNo	Last Accessed
[0]	L	2	t=12
[1]	L	3	t=10
[2]	–	–	–
[3]	–	–	–
[4]	M	0	t=15
[5]	M	1	t=13

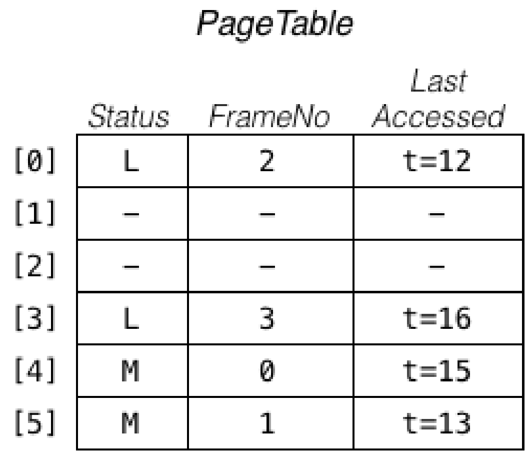


MemFrames

[0]	Holds Page 4
[1]	Holds Page 5
[2]	Holds Page 0
[3]	Holds Page 1

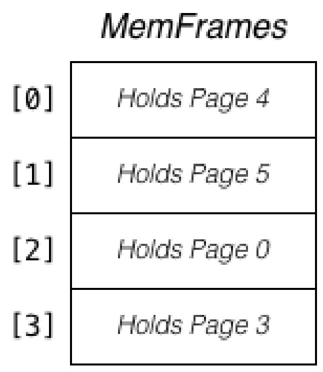
If the next instruction references an address from page 3, then we have a problem. All of the memory frames are in use, so we need to replace the contents of one of the frames to handle the new address reference (i.e., we need to evict a page from a frame, and replace the contents of the frame by the page which has just been referenced). If the page being evicted has been modified, it will need to be written to disk before being replaced in memory.

The following diagram shows the scenario after page 3 was loaded. It replaced page 1 in frame 3.



PageTable

	Status	FrameNo	Last Accessed
[0]	L	2	t=12
[1]	–	–	–
[2]	–	–	–
[3]	L	3	t=16
[4]	M	0	t=15
[5]	M	1	t=13



MemFrames

[0]	Holds Page 4
[1]	Holds Page 5
[2]	Holds Page 0
[3]	Holds Page 3

How did we decide to replace page 1 rather than, say, page 5? Here, we've used the *Least Recently Used* strategy: this involves scanning the page table and finding the loaded page with the oldest access timestamp. In the above example, this was page 1, last accessed at time t=10. All of the other pages were more recently accessed than this.

Setting Up

Download `lab.zip` from the link above, and extract it using unzip. You should now find the following files in the directory:

- Makefile – a set of dependencies used to control compilation
- vmsim.c – a partially-complete virtual memory simulator
- tests/ - a directory containing simple test cases

Note that, as supplied, the program will compile, but will *not* behave as we want. It will simply complain that any address you supply is invalid.

Exercise

The simulation program involves two core data structures:

- `PageTable`, an array of `nPages` page table entries, and
- `MemFrames`, an array of `nFrames` integers, where each entry is -1 (nothing loaded) or a page number

The simulation assumes that there is a single process whose page table entries are maintained in `PageTable`, and a memory with `nFrames` frames.

What the `main()` function does is roughly as follows:

```
main:
    set nPages from argv[1]
    set nFrames from argv[2]
    initialise the clock, PageTable and MemFrames
    for each line of input in stdin:
        extract the operation and virtual address
        convert the virtual address to a physical address
        display the new state of all of the data structures
        tick the clock
```

It performs the above until it either runs out of input or until it encounters an invalid address (outside the range of addresses [0 … MAX), where the maximum address MAX = nPages x PAGESIZE.

You should read the definitions at the start of vmsim.c, then the main() function (whose operation is described above). After that, look at the helper functions initPageTable(), initMemFrames, and showState(), which should give you a clearer idea of how the data structures work. Once you've done that, your goal for this lab is to complete the physicalAddress() function, which should behave as follows:

```
physicalAddress (vAddr, action):
    extract page# and offset from vAddr
    if the page# is not valid, return -1

    if the page is already loaded:
        set the Modified flag if action is a write,
        update the access time to the current clock tick, and
        use the frame number and offset to compute a physical address

    otherwise:
        look for an unused frame;
        if we find one, use that,
        otherwise:
            // we need to replace a currently loaded frame, so
            find the Least Recently Used loaded page,
            set its PageTable entry to indicate "no longer loaded",
            increment the nReplaces counter,
            increment the nSaves counter if modified, and
            use the frame that backed that page.

        // we should now have a frame# to use, so:
        increment the nLoads counter,
        set PageTable entry for the new page
            (flags, frame#, accesstime=current clock tick), and
        use the frame number and offset to compute a physical address

    return the physical address
```

The input file for the simulator is a trace of memory accesses, one access per line, where each line contains an action (`'R'` or `'W'`), and a virtual (process) address (in decimal). For example:

```
R 1000
W 2000
R 1004
W 2004
...
```

which indicates a program that, first, reads a value from virtual address 1000, then writes a value to virtual address 2000, then reads a value from virtual address 1004, then writes a value to virtual address 2004.

The output of the simulator is a sequence of states, where each state is preceded by the address that was last processed. For example:

```
@ t=7, read from pA=2 (vA=12290)

PageTable (Stat,Acc,Frame)
[ 0] --, -1, -1
[ 1] LM,  3,  1 @ 4096
[ 2] L-,  5,  2 @ 8192
[ 3] L-,  7,  0 @ 0
[ 4] --, -1, -1
MemFrames
[ 0]  3 @ 0
[ 1]  1 @ 4096
[ 2]  2 @ 8192
```

… shows that, at time t=7, the virtual address 12290 was accessed. After this, the page table shows that pages 1, 2, 3 are loaded in memory. Page 1 has been modified since it was loaded, while pages 2 and 3 are unmodified. The second column in the page table shows the last accessed time of each loaded page (or -1 if it is not loaded). The third column shows which frame the page is contained in; while the value for the corresponding frame shows which page is loaded in that frame.

The final line of the output shows the values of the three counters after the program has finished accessing addresses.

You can see several examples of the expected output from the program in the `tests/` directory. Tests are in pairs, e.g., the file `05.sh` shows the command line parameters that were used to run the test, while `05.exp` shows the expected output.

Notes:

- pages are 4 KiB = 2^{12} bytes
- all address values are in decimal (not hexadecimal)
- no replacements ever occur if nFrames ≥ nPages
- if the Modified bit is set, the Loaded bit will also always be set
- if the Loaded bit is set, the Modified bit not necessarily be set
- you can force different behaviours by changing the command-line parameters