HomeWork 3, Virtual Memory Manager (VMM)

Due Date: Wednesday 10/27/2021
HW Delivery: submit on **Canvas** by the due date, before midnight
Total Points: 60

General rules: Create homework, compose specifications or any text by using a common *document-creation* tool, such as Microsoft® Word. Program in C, C++ or Java. Refer to the wwweb or lecture notes for this class to design, implement, and debug solid SW solutions. Be complete, and precise.

Summary: Design, implement, test, debug and document a simplified simulator for a demand-paged ***Virtual Memory Manager*** named ***Vmm***. Vmm assumes a 32-bit, byte-addressable architecture with 4 kB sized, and 4 kB aligned page frames. Run your program with your own, well thought-out inputs. Also *use the inputs* mentioned here. Only submit inputs and outputs of your test runs, not your program source, and not the traces you used for your own debug!

Abstract: Simulate a Virtual Memory Manager named ***Vmm*** that reads in textual form memory requests (AKA loads and stores) from stdin and simulates them. *Vmm* measures the number of CPU cycles of any simulated action. To simplify, the fixed cycles for memory accesses are defined via constants; they do not vary dynamically as they generally will in a real run-time system. Implement three levels of 32-bit VMM address mapping, using 10, 10 and 12 bits of any logical address. Except for the Page Directory (***PD***) which is implemented as a dedicated 4 kB-size cache, *Vmm* operates without caches. Accesses through the PD cost 1 cycle each.

Track the exact number of cycles ***for each*** and ***for all*** memory accesses; also compute the number of cycles if no *virtual memory* mapping had been available. Track the number of page *swap-ins, swap-outs*, and all interesting *demand paging related* activities. Define the number of cycles for any swap operation to be fixed at 5000 cycles per swap. In reality this will be higher and does generally vary from one disk access to another.

Output some text file to help you debug your code, showing the state of the memory subsystem at each swap-in and swap-out. This file can become large when the system is thrashing. Make sure you do run some thrashing case. Do not turn in these files.

Only turn in the actual output of short simulations, do not turn in your debug information. For each simulation run show (and hand in) the actual input, including the initial **p xx**, with **xx** specifying the number of page frames available. Each time a new Page Table (PT) or user page is allocated, output the current status of the Page Directory (PD) and PT. For brevity it is OK only to dump valid table entries (with their respective index), not the complete table with more than 1000 possible entries; many of those may be NULL.

Detailed Requirements: Simulate Vmm for a byte-addressable, *4-bytes per word* architecture with 32 bit addresses, using ***demand paging***. A page is 4 kB long, and 4 kB aligned. Write also a separate address-generating program to be used for your simulation inputs. Load- and store requests (in ASCII text form) are read by the simulator's front-end ***(FE)*** and "executed" by the simulator.

To start a simulation run, the first input tuple **p** of information is the number of page frames available; e.g. input **p 9** indicates 9 physical page frames are available for this run; no more. Whenever the data space needed to execute your simulated program is larger than the available number of physical frames (like in case of **p 9**), some resident page will have to be swapped out. This is called the victim page. Careful, if the victim page is unmodified and

already on disk, it can simply be overridden; no need to swap back (AKA write back) onto disk. Yet any page swapped out must be marked as being "no present" in its VMM data structure. Only user pages are swapped out, not Page Tables. The *PD* being implemented as a HW cache can never be swapped out.

Use three levels of address mapping. Each logical 32-bit address is interpreted as a 10+10+12 bit string as follows:

PD: The leftmost 10 bits of a logical address hold the index into the Page Directory ***PD***. The *PD* is a 4 kB HW cache, having space for 1024 pointers to some **PT**. An entry in the PD is 4 bytes long, or 32 bits, sufficient to hold the frame-size aligned address of a **PT.** The remaining 12 bits in a PD entry are usable for administrative detail about the PT. You use some of these bits, to track present, modified, etc.

PT: The next 10 bits of a logical address are the index into the Page Table ***PT***. An entry in the PT is 4 bytes long, or 32 bits, each sufficient to hold the address of a **user page.** The remaining 12 bits are usable for further detail about the user page, such a present, modified, etc.

User Page: Finally, the rightmost 12 bits of any logical address are the byte-offset into a user page. The start address of a user page is pointed to by the appropriate *PT* entry.

Replacement: If a page must be swapped out, use a ***Random Replacement Policy***. Define some fixed, arbitrary, small number of (simulated) machine cycles for selecting the victim page; e.g. 10 cycles would be an acceptable constant. This constitutes a simplification compared with a real demand paged system that must identify a victim page, sometimes at great cost.

Note that only 20 address bits need to be stored in each of the 1024 *PD* and *PT* entries. While logical addresses are 32 bits long, all frames are aligned on 4 Kb boundaries, thus the rightmost 12 bits of any page address always happen to be 0. No need to store them! Instead, the remaining 12 bits can be used for: the **P** bit (present); **D** bit (dirty, set if written, clear initially; AKA **M** bit for modified); **R** bit for read-only pages; **S** bit for a shared page; and other data interesting to an Operating System about a page. Perhaps a few bits even remain unused.

Summary Output after completion of a simulation:

```
* * * Paging Activity Statistics * * *
 number of memory accesses       = xx
 number of triples (1 + access)  = xx
 number of swap ins (faults)     = xx
 number of swap outs             = xx
 total number of pages malloced  = xx
 number of pages for Page Tables = xx
 number of page frames for user  = xx
 total memory cycles             = xx
 cycles w/o Vmm                  = xx
 cycles per swap_in              = 5000
 cycles per swap_out             = 5000
 last working set size           = xx
 max working set size ever       = xx
 max physical pages              = xx
 page size                       = 4096
 replacement algorithm           = random
```

Required Input and resulting output: P x at the beginning of the input specifies the maximum number **x** of physical page frames. Example above showed **p 9**. Other input samples to the simulator: **w x y** define that at address **x** the value **y** will be stored, i.e. **w**ritten; in your actual simulation you may ignore the **y** value actually being stored. Tuple **r x** defines that memory address **x** is loaded (AKA **r**ead) into some register. Simulate and track the timing for each memory access.

Sample Input, Required to be Tested with 9 Page Frames:

```
p     9
w  1254    0  w  1250    4  w  2500    8  w  1252    7  w  2600    3
w  2650    2  w  1260    0  w  2800    0  w   1268   0  w  2700    8
w     0    1  r     0       r  1250       r  2500       w     0    0
r     0       r  1252       r  2550       w     0    0  r     0
r  1254       r  2600       w     0    0  r     0       r  1256
r  2650       w     0    0  r     0       r  1258       r  2700
w     0    0  r     0       r  1260       r  2750       w     0    0
r     0       r  1262       r  2800       w     0    0  r     0
r  1264       r  2850       w     0    0  r     0       r  1266
r  2900       w     0    0  r     0       r  1268       r  2950
r  3298       w     0 6048  r     0       r  2482       r  3348
w     0 6528  r     0       r  2484       r  3398       w     0 7020
r     0       r  2486       r  3448       w     0 7524  r     0
r  2488       r  3498       w     0 8040  r     0       r  2490
r  3548       w     0 8568  r     0       r  2492       r  3598
w     0 9108  r     0       r  2494       r  3648       w     0 9660
r     0       r  2496       r  3698       w     0 10224 r     0
r  2498       r  3748       w     0 10800 r     0       w  4998 10800
```

Final Output for Some Simulation Run:

```
 * * * Paging Activity Statistics * * *
 number of memory accesses       = 64376
 number of triples (1 + access)  = 64377
 number of swap ins (faults)     = 0
 number of swap outs             = 2
 total number of pages malloced  = 10
 number of pages for Page Tables = 1
 number of page frames for user  = 9
 total memory cycles             = 1,287,500
 cycles w/o Vmm                  =   643,750
 cycles per swap_in              = 5000
 cycles per swap_out             = 5000
 last working set size           = 8
 max working set size ever       = 8
 max physical pages              = 8
 page size                       = 4096
 replacement algorithm           = random       -- or your choice
 Address range                   = . . .
```

Sample Swap-Activity; Leading 0s not printed:

```
  * * *  Done mallocing a new Page Table * * *
Dumping Page Directory:
pd[0].v   = 0xdb40
```

```
pd[0].lru = 2

Dumping Page Tables:
Page Dir entry 0:

  * * *  Done mallocing a new page Malloc User Page * * *
Dumping Page Directory:
pd[0].v   = 0xdb40
pd[0].lru = 2

Dumping Page Tables:
Page Dir entry 0:
pt[0].v = 0x000eb48, lru = 2

  * * *  Done mallocing a new page Malloc User Page * * *
Dumping Page Directory:
pd[0].v   = 0xdb40
pd[0].lru = 2

Dumping Page Tables:
Page Dir entry 0:
pt[0].v = 0x000eb48, lru = 3, written
pt[2].v = 0x000fb50, lru = 4

  * * *  Done mallocing a new page Malloc User Page * * *
Dumping Page Directory:
pd[0].v   = 0xdb40
pd[0].lru = 2

Dumping Page Tables:
Page Dir entry 0:
pt[0].v = 0x000eb48, lru = 3, written
pt[2].v = 0x000fb50, lru = 4
pt[4].v = 0x0010b58, lru = 5

  * * *  Done mallocing a new page Malloc User Page * * *
Dumping Page Directory:
pd[0].v   = 0xdb40
pd[0].lru = 2

Dumping Page Tables:
Page Dir entry 0:
pt[0].v = 0x000eb48, lru = 11, written
pt[2].v = 0x000fb50, lru = 12
pt[4].v = 0x0010b58, lru = 9
pt[5].v = 0x0011b60, lru = 13
```

Sample Execution and Output: 1 Single Memory Access

```
p 9                          -- required input: # page frames, here 9
w 0x1008  0                  -- 1 memory access example, store 0 at hex 1008
pd[0] is NIL: 1 cycle, pd[0]=NIL
addr = get_new_page_table(), costs 5,000 cycles -- start output for 1 store
pd[0] = addr, 1 cycle
pt[1] is NIL, 10 cycles
```

```
addr = get_new_user_page(), costs 5,000 cycles
pt[1] = addr, 10 cycles
st addr+8   10 cycles
Total cost: 10,032 cycles                        -- end output for 1 store
```