Programming Assignment #3 (Lab 3): Virtual Memory Management **Professor Hubertus Franke**
Class CSCI-GA.2250-001 Fall 2013

Due November 11th 2013 (3 weeks)

You are to implement a Virtual Memory Manager in C or C++ and submit the **source** code, which we will compile and run. Your email should describe how to compile your code.

In this lab/programming assignment you will implement/simulate the virtual memory operations that map a virtual address space comprised of virtual pages onto physical page frames. As the size of the virtual address space may exceed the number of physical pages of the simulated system (#frames will be an input parameter) paging needs to be implemented. Only one virtual address space will be considered in this assignment. The size of the virtual space is 64 virtual pages. The number of physical page frames varies and is specified by an option.

The input to your program will be a sequence of "instructions" and optional comment line (shown in Listing 1). A comment line anywhere in the input file starts with '#' and should be completely ignored by your program and not count towards the instruction count. An instruction line is comprised of two integers, where the first number indicates whether the instruction is a read (value=0) or a write (value=1) operation and which virtual page is touched by that operation. You can assume that the input files are well formed as shown below, so fancy parsing is not required (but verify the virtual page index).

```
#page reference generator
#inst=1000000 pages=64 %read=75.000000 lambda=2.000000
1 13
0 34
0 18
```

Listing 1: Input File Example

You need to represent the virtual address space as a single level pagetable.
Each page table entry (pte) is comprised of the PRESENT, MODIFIED, REFERENCED and PAGEDOUT bits and an index to the physical frame (in case the pte is present). This information can and should be implemented as a single 32-bit value (and not as a structure of multiple integer values, unless you know how to do bit structures).
(see http://www.cs.cf.ac.uk/Dave/C/node13.html (BitFields) or http://www.catonmat.net/blog/bit-hacks-header-file/ as an example).

During each instruction you simulate the behavior of the hardware and hence you must check that the page is present and if not locate a free frame, page-in the page if the PAGEDOUT flag is set to the frame or zero the frame. Then the virtual page of the instruction needs to be mapped to the frame. You must maintain the PRESENT, MODIFIED, REFERENCED and PAGEDOUT bits and the frame index in the pagetable's pte.
The frame table is NOT updated by the simulated hardware.

Initially all page frames are maintained in a free list with frame-0 being the first frame in the list and all pte's of the pagetable are zero'd (i.e. NOT PRESENT and NOT PAGEDOUT).

You need to implement page replacement to handle the cases where there are no free page frames available.
The following page replacement algorithms are to be implemented (letter indicates option (see below)):

Algorithm	Based on Physical Frames	Based on Virtual Pages
NRU (Not Recently Used)		N
LRU (Least Recently Used)	l	
Random	r	
FIFO	f	
Second Chance (derivative of FIFO)	s	
Clock	c	C
Aging	a	A

The page replacement should be a generic and the algorithms should be special instances of the page replacement class to avoid "switch/case statements" in the simulation of instructions.

When a virtual page is replaced it must be unmapped and paged out. While you are not effectively implementing these operations you still need to track them and create entries in the output (see below).

In "Physical" you base your decision on the status of the physical frames. Since you need access to the pte-s (access to the ref and modify bits which are only maintained in the pagetable) you should keep track of the inverse mapping (frame -> pte). In "Virtual", you base the replacement decision on the virtual pages.
You are to create the following output

```
==> inst: 0 2
78: UNMAP  42  26
78: OUT    42  26
78: IN      2  26
78: MAP     2  26

Output 1
```

```
==> inst: 0 37
69: UNMAP  35  18
69: IN     37  18
69: MAP    37  18

Output 2
```

```
==> inst: 0 57
75: UNMAP  58  17
75: ZERO       17
75: MAP    57  17

Output 3
```

For instance in Output 1 Instruction 78 is a read operation on virtual page 2. The replacement algorithms selected virtual page 42 or physical frame 26 (dependent on whether a physical or virtual replacement type was used) and hence first has to unmap the virtual page 42 to avoid further access, then because the page was dirty (modified) (this would have been tracked in the PTE) it pages the page out to a swap device. Then it pages in the virtual page 2 into the physical frame 26, finally maps it which makes the PTE_2 a valid/present entry and allows the access. Similarly in output 2 a read operation is performed on virtual page 37. The replacement selects 35/18 as the virtual / physical page to be replaced. The page is not paged out, which indicates that it was not dirty/modified since the last mapping. The virtual page 37 is then paged in to physical frame 18 and finally mapped. In output 3 you see that after unmapping page 58/17, the frame is zeroed (hence no virtual page number in the output), which indicates that the page was never paged out (though it might have been unmapped previously see output 2). An operating system must zero pages on first access (unless filemapped) to guarantee consistent behavior.

In addition your program needs to compute and print the summary statistics related to the VMM. This means it needs to track the number of unmap, map, pageins, pageouts and zero operations. In addition you should compute the overall execution time in cycles, where maps and unmaps each cost 400 cycles, page-in/outs each cost 3000 cycles, zeroing a page costs 150 cycles and each access (read or write) costs 1 cycles.

```
printf("SUM %d U=%d M=%d I=%d O=%d Z=%d ===> %llu\n",
    inst_count, stats.unmaps, stats.maps, stats.ins, stats.outs, stats.zeros, totalcost);
```

Note, the cost calculation can overrun 2^32 and you must account for that, so use 64-bit counters. We will test your program with 1 Million instructions.

Execution and Invocation Format:

Your program **must** follow the following invocation:
./mmu [-a<algo>] [-o<options>] [–f<num_frames>] inputfile randomfile (optional arguments in any order).
e.g. ./mmu –aC –o[OPFS] infile rfile selects the Clock Algorithm for Virtual Pages and creates output for operations, final page table content and final frame table content and summary line (see above). The outputs should be generated in that order if specified in the option string regardless how the order appears in the option string. Default values are equivalent to "-al –f32" for options. We will for sure test the program with –oOPFS options (see below), change the page frame numbers and "diff" compare it to the expected output.

The test input files and the sample file with random numbers are on the website. The random file is required for the Random algorithm and the Virtual NRU algorithm. Please reuse the code you have written for lab2, but note

the difference in the modulo function which now indexes into [0, size) vs previously (0, size].

In the Random you compute the frame selected as with size==num_ frames) and in the virtual NRU algorithm, you use the random function to select a random page from the lowest class identified, in particular you should create the class by creating an array and then index into that array via the index = array[class][rands[rofs] % num_pages_in_array[class]]. As in the lab1 case, you increase the *rofs* and wrap around from the input file.

- The 'O' (ohhh) option shall generate the required output as shown in output-1/3.
- The 'P' (pagetable option) should print after the execution of all instructions the state of the pagetable: As a single line, you print the content of the pagetable pte entries as follows:

  0:RMS 1:RMS 2:RMS 3:R-S 4:R-S 5:RMS 6:R-S 7:R-S 8:RMS 9:R-S 10:RMS 11:R-S 12:R-- 13:RM- # # 16:R-- 17:R-S # # 20:R-- # 22:R-S 23:RM- 24:RMS # # 27:R-S 28:RMS # # # # # 34:R-S 35:R-S # 37:RM- 38:R-S * # 41:R-- # 43:RMS 44:RMS # 46:R-S * * # * * * # 54:R-S # * * 58:RM- * * # * *

  R (referenced), M (modified), S (swapped out).

  Pages that are not valid are represented by a '#' if they have been swapped out (note you don't have to swap out a page if it was only referenced but not modified), or a '*' if it does not have a swap area associated with. Otherwise (valid) indicates the virtual page index and RMS bits with '-' indicated that that bit is not set.

  Note a virtual page, that was once referenced, but was not modified and then is selected by the replacement algorithm, does not have to be paged out (by definition all content must still be ZERO) and can transition to '*'.

- The 'F' (frame table option) should print after the execution and the frame table and should show which frame is mapped at the end to which virtual page index or '*' if not currently mapped by any virtual page.

  30 * 1 12 6 13 * 19 40 55 14 47 32 43 11 3 5 63 24 9 0 61 23 36 27 60 57 25 4 48 2 52

- The 'S' option prints the summary line ("SUM …") described above.
- The 'p' options prints the page table after each instructions (see example outputs) and this should help you significantly to track down bugs and transitions
- The 'f' option prints the frame table after each instruction.

We will not test or use the '-f' and the '-p' option during the grading. It is purely for your benefit to add these.

All scanning (virtual or physical) algorithms (with exception of PHYS_LRU, clock algorithms, random) should start their scanning at page-0 or frame-0, respectively.

More details and examples are on the website.

FAQ:

NRU requires that the REFERENCED-bit be periodically reset for all valid page table entries. The book suggest on every clock cycle. Since we don't implement a clock interrupt, we shall reset the ref bits every 10^{th} page replacement request before you implement the replacement operation.

AGING requires to maintain the age-bit-vector. In this assignment please assume a 32-bit counter (vector). It is implemented on every page replacement request.