Programming Assignment #3 (Lab 3): Virtual Memory Management **Professor Hubertus Franke**
Class CSCI-GA.2250-001 Summer2018

In this lab/programming assignment you will implement/simulate the operation of a Virtual Memory Manager that maps the virtual address spaces of multiple processes onto physical frames. The assignment will assume multiple processes, each with its own virtual address space of 64 virtual pages. As the sum of all virtual pages in all virtual address space may exceed the number of physical pages of the simulated system paging needs to be implemented. The number of physical page frames varies and is specified by a program option, but assume you have to support 128 frames (No more, but certainly less). Implementation is to be done in C/C++. Please submit your source only including a Makefile as a single compressed file via NYU classes.

INPUT SPECIFICATION:

The input to your program will be a comprised of:
1. the number of processes (processes are numbered starting from 0)
2. a specification for each process' address space is comprised of
   a. the number of virtual memory areas (aka VMAs)
   b. specification for each said VMA comprises of 4 numbers:
      "starting_virtual_page ending_virtual_page write_protected[0/1] filemapped[0/1]"

Following is a sample input with two processes. First line not starting with a '#' is the number of processes. Each process in this sample has two VMAs. **Note: ALL** lines starting with '#' must be ignored and are provided simply for documentation and readability. In particular, the first few lines are references that *document* how the input was created, though they are irrelevant to you.

```
#process/vma/page reference generator
#       procs=2 #vmas=2 #inst=100 pages=64 %read=75.000000 lambda=1.000000
#       holes=1 wprot=1 mmap=1 seed=19200
2
#### process 0
#
2
0 42 0 0
43 63 1 0
#### process 1
#
2
0 17 0 1
20 63 1 0
```

Since it is required for VMAs of single address space to not overlap, this property is **guaranteed** for all provided input files. However, there can potentially be holes between VMAs, which means that not all virtual pages of an address space are valid (i.e. assigned to a VMA). Each VMA is comprised of 4 numbers.
start_vpage:
end_vpage: (note the VMA has (end_page – start_vpage + 1) virtual pages
write_protected: binary whether the VMA is write protected or not
file_mapped: binary to indicated whether the VMA is mapped to a file or not

The process specification is followed by a sequence of "instructions" and optional comment lines (see following example).
An instruction line is comprised of a character ('c', 'r', or 'w') followed by a number.
"c <procid>" specifies that a context switch to process #<procid> is to be performed. It is guaranteed that the first instruction will always be a context switch instruction, since you must have an active pagetable in the MMU (in real systems).
"r <vpage>" implies that a load/read operation is performed on said virtual page.
"w <vpage>" implies that a store/write operation is performed on said virtual page.

```
# #### instruction simulation ######
c 0
r 32
w 9
r 0
r 20
r 12
```

You can assume that the input files are well formed as shown above, so fancy parsing is not required. Just make sure you take care of the '#' comment lines.

DATA STRUCTURES:

To approach this assignment, read in the specification and create processes, each with its list of *vma*s and a *page_table* that represents the translations from virtual pages to physical frames for that process.

In addition, you must define a global *frame_table* that each operating system maintains to describe the usage of each of its physical frames and where you maintain backward mappings to the address space(s) and the vpage that maps a particular frame (In this assignment a frame can only be mapped by at most one PTE at a time).

A page table naturally must contain exactly 64 page table entries (PTE). A PTE is comprised of the PRESENT/VALID, WRITE_PROTECT, 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 or as a bit structures (likely easier). It cannot be a structure of multiple integer values that collectively are larger than 32-bits. (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, I highly encourage you to use this technique, let the compiler do the hard work for you).
Assuming that the maximum number of frames is 128, which equals 7 bits and the mentioned 5 bits above, you effectively have 32-12 = 20 bits for your own usage in the pagetable. You can use these bits at will (e.g. remembering whether a PTE is file mapped or not).

SIMULATION and IMPLEMENTATION:

During each instruction you simulate the behavior of the hardware and hence you must check that the page is present. A special case is the ‘c’ (context switch) instruction which simply changes the current process and current page table pointer.

Structure of the simulation

The structure of the simulation should be something like the following:

```
typedef struct { … } pte_t;          // can only be total of 32-bit size !!!!
typedef struct { … } frame_t;

class Pager {
       virtual frame_t* select_victim_frame();
};

frame_t *get_frame() {
       frame_t *frame = allocate_frame_from_free_list();
       if(frame == NULL) frame = THE_PAGER->determine_victim_frame();
        return frame;
}

while (get_next_instruction(&operation, &vpage)) {
        pte_t *pte = &current_process.page_table[vpage];   / done by hardware in reality
        if ( ! pte->present) {
            frame_t *newframe = get_frame();
            //-> figure out if/what to do with old frame if it was mapped
            //   see general outline in MM-slides under Lab3 header
            //   see whether and how to bring in the content of the access page.
         }
        // simulate instruction execution by hardware by updating the PTE bits
        update_pte(read/modify/write) bits based on operations.
}
```

If the page is not present, as indicated by the associated PTE’s valid/reference bit, the hardware would raise a page fault exception. Here you just simulate this by calling your (operating system’s) pagefault handler. In the pgfault handler you first determine that the *vpage* can be accessed, i.e. it is part of one of the VMAs. If not, a SEGV output line must be created and you move on to the next instruction. If it is part of a VMA then the page must be instantiated, i.e. a frame must be allocated, assigned to PTE belonging to the vpage of this instruction (i.e. *currentproc->pagetable[vpage].frame = allocated_frame*) and then populated with the proper content. The population depends whether this page was previously paged out (in which case the page must be brought back from the swap space ("IN") or ("FIN" in case it is a memory mapped file). If the vpage was never swapped out and is not file mapped, then by definition it still has a zero filled content and you issue the "ZERO" output.

That leaves the allocation of frames. All frames initially are in a free "list", which you can implement as an index to the frame_table (starting with 0) [note once all frames are used in this assignment they are never returned to the free list]. Once you run out of free frames, you must implement paging. We explore the implementation of several page replacement algorithms. Page replacement implies the identification of a victim page according to the algorithm's policy. This should be implemented as a subclass of a general *Pager* class with one virtual function "`frame_t* (*select_victim_frame)();`" that returns a victim frame. Once a victim frame has been determined, the victim frame must be unmapped from its user (<address space:vpage>), i.e. its entry in the owning process's page_table must be removed ("UNMAP"), however you must remember the state of the bits. If the page was modified, then the page frame must be paged out to the swap device ("OUT") or in case it was file mapped written back to the file ("FOUT"). Now the frame can be reused for the faulting instruction. First the PTE must be reset (note once the PAGEDOUT flag is set it will never be reset as it indicates there is content on the swap device) and then the PTE's frame must be set.

Classes for every paging algorithm should derive from a base Pager class and there should be no replication of the simulation code. If you need a special functions (e.g. HW LRU), create an empty base function and overwrite in the specific pager subclass. This will allow adding new classes.

At this point it is guaranteed, that the vpage is backed by a frame and the instruction can proceed in hardware (with the exception of the SEGV case above) and you have to set the REFERENCED and MODIFIED bits based on the operation. In case the instruction is a write operation and the PTE's write protect bit is set (which it inherited from the VMA it belongs to) then a SEGPROT output line is to be generated. The page is considered referenced but not modified in this case.

Your code must actively maintain the PRESENT (aka valid), MODIFIED, REFERENCED, and PAGEDOUT bits and the frame index in the pagetable's pte. The frame table is NOT updated by the simulated hardware as hardware has no access to the frame table. Only the pagetable entry (pte) is updated just as in real operating systems and hardware.

The following page replacement algorithms are to be implemented (letter indicates option (see below)):

Algorithm	Based on Physical Frames
FIFO	f
Second-Chance (derivative of FIFO)	s
Random	r
NRU (Not Recently Used)	n
Clock	c
Aging	a

The page replacement should be generic and the algorithms should be special instances of the page replacement class to **avoid "switch/case statements" in the simulation of instructions**. Use object oriented programming and inheritance.

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

Since all replacement algorithms are based on frames and the reference and modified bits are only maintained in the page tables of processes you need access to the PTEs. To be able to do that you should keep track of the inverse mapping: (frame --> <proc-id,vpage>) inside each frame's frame table entry.

Note: you MUST NOT set any bits in the PTE. The pte should be initialized to "0" before the instruction simulation starts. This is also true for assigning FILE or WRITEPROTECT bits from the VMA. This is to ensure that in real OSs the full page table (hierarchical) is created on demand. Instead on the first page fault on a particular pte, you have to search the *vaddr* in the VMA list. At that point you can store bits in the pte based on what you found in the VMA.

You are to create the following output if requested by an option (see at options description):

```
78: ==> r 2                    69: ==> r 37                    75: ==> w 57
 UNMAP 1:42                     UNMAP 0:35                      UNMAP 2:58
 OUT                            IN                              ZERO
 IN                             MAP 18                          MAP 17
 MAP 26

Output 1                       Output 2                        Output 3
```

For instance in Output 1 Instruction 78 is a read operation on virtual page 2 of the current process. The replacement algorithms selected physical frame 26 that was used by virtual page 42 of process 1 (1:42) and hence first has to **UNMAP** the virtual page 42 of process 1 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 with the (1:26) tag so the Operating system can find it later when process 1 references vpage 42 again (note you don't implement the lookup). Then it pages **IN** the previously swapped out content of virtual page 2 of the current process (note this is where the OS would use <current-process-id : vpage> tag to find the swapped out page) into the physical frame 26, and 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 frame 18 that was mapped by process_0's vpage=35. 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 frame 17 was selected forcing the unmapping of its current user process_2, page 58, the frame is zeroed (hence no virtual page number in the output), which indicates that the page was never paged out or written back to file (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 the case of filemapped virtual pages (i.e. part of filemapped VMA) even the initial content must be loaded from file.

In addition your program needs to compute and print the summary statistics related to the VMM if requested by an option. This means it needs to track the number of instructions, segv, segprot, unmap, map, pageins (IN, FIN), pageouts (OUT, FOUT), and zero operations for each process. 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, file in/outs cost 2500 cycles, zeroing a page costs 150 cycles, a segv costs 240 cycles, a segprot costs 300 cycles and each access (read or write) costs 1 cycles and a context switch costs 121 cycles.

Per process:

```
     printf("PROC[%d]: U=%lu M=%lu I=%lu O=%lu FI=%lu FO=%lu Z=%lu SV=%lu SP=%lu\n",
                     proc->pid,
                     pstats->unmaps, pstats->maps, pstats->ins, pstats->outs,
                     pstats->fins,          pstats->fouts,          pstats->zeros,
                     pstats->segv, pstats->segprot);
```

All:

```
          printf("TOTALCOST %lu %lu %llu\n", ctx_switches, inst_count, cost);
```

If requested by an option you have to print the relevant content of the page table of each process and the frame table.

```
PT[0]: * 1:RM- * * * 5:-M- * * 8:--- * * # * * * * * * * * # * * * 24:--- * * * # *
* * * * * * * * # * * * * * * * * * * # * * * # * * * * # * * * * * * * * *
FT: 0:1 0:5 0:24 0:8
PROC[0]: U=25 M=29 I=1 O=8 FI=0 FO=0 Z=28 SV=0 SP=0
TOTALCOST 1 31 52951
```

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. Also the end calculations are tricky, so do them incrementally. Don't add up 32-bit numbers and then assign to 64-bit. Add 32-bit numbers incrementally to the 64-bit counters, if you use 32-bit.

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 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. **We will grade 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 supplied. The random file is required for the Random algorithm and the 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 replacement algorithm you compute the frame selected as with (size==num_frames) and in the NRU algorithm, you use the random function to select a random frame 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 lab2 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 for each process, you print the content of the pagetable pte entries as follows:

  ```
  PT[0]: 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) (note we don't show the write protection bit as it is implied/inherited from the specified VMA.
  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 should show which frame is mapped at the end to which <pid:virtual page> or '*' if not currently mapped by any virtual page.

  ```
  FT: 0:32 0:42 0:4 1:8 * 0:39 0:3 0:44 1:19 0:29 1:61 * 1:58 0:6 0:27 1:34
  ```

- The 'S' option prints the summary line ("SUM …") described above.
- The 'x' options prints the current page table after each instructions (see example outputs) and this should help you significantly to track down bugs and transitions
- The 'y' option is like 'x' but prints the page table of all processes instead.
- The 'f' option prints the frame table after each instruction.
- The 'a' option prints the "aging" information after each instruction for complex algorithms.

We will not test or use the '-f' ,'-a' or the '-x,-y' option during the grading. **It is purely for your benefit to add these and compare with the reference program under ~frankeh/Public/mmu on any assigned cims machines.**

All scanning replacement algorithm, should start at frame[0].

Please note you have to provide a modular design which separates the simulation (instruction by instruction) from the replacement policies. Use OO style of programming and think about what operations you need from a generic page replacement (which will define the API). A lack of doing a modular design with separated replacement policies and simulation will lead to a deduction of 5pts.

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 10th page replacement request before you implement the replacement operation (note these don't include the initial page faults, only invocation for selecting a victim page).

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.

The *pagetable* and *frametable* output is generated AFTER the instruction is executed, so the output becomes the state seen prior to executing the next instruction.

How do I provide **optional argument**s in arbitrary order ?
Please read: http://www.gnu.org/software/libc/manual/html_node/Example-of-Getopt.html (very useful and trivial to use)

What data structures do I maintain:
You must maintain a single level pagetable (struct PTE page_table[64]) per process and a global frametable[num_frames] Note that reference and modified etc can ONLY be maintained in the pagetables as hardware will only update those in real life. **The frametable is an OS structure with one entry to represent/describe per physical frame metadata**. PTE bits (Reference, Modified, et. al. are **NOT** maintained in the frametable !!!)

Provided Inputs

The submission comes with a few generated inputs that you can use to address implementation in an incremental step. Each input is characterized by how many processes, how many maximum vmas per process, how many maximum write protected vmas per process, whether there is a filemapped vma and maximum numbers of vma holes might exist for a process.

Input File	#inst	#procs	#vmas (max)	Write-protect	File-mapped	Holes
in1	30	1	1	0		
in2	30	1	1	0		
in3	100	1	4	0		
in4	100	1	5	2		
in5	100	1	5	0	1	
in6	100	1	5	2		2
in7	200	2	2			
in8	200	2	3	1	1	
in9	1000	3	4	1	1	2
in10	10000	4	6	2	1	2

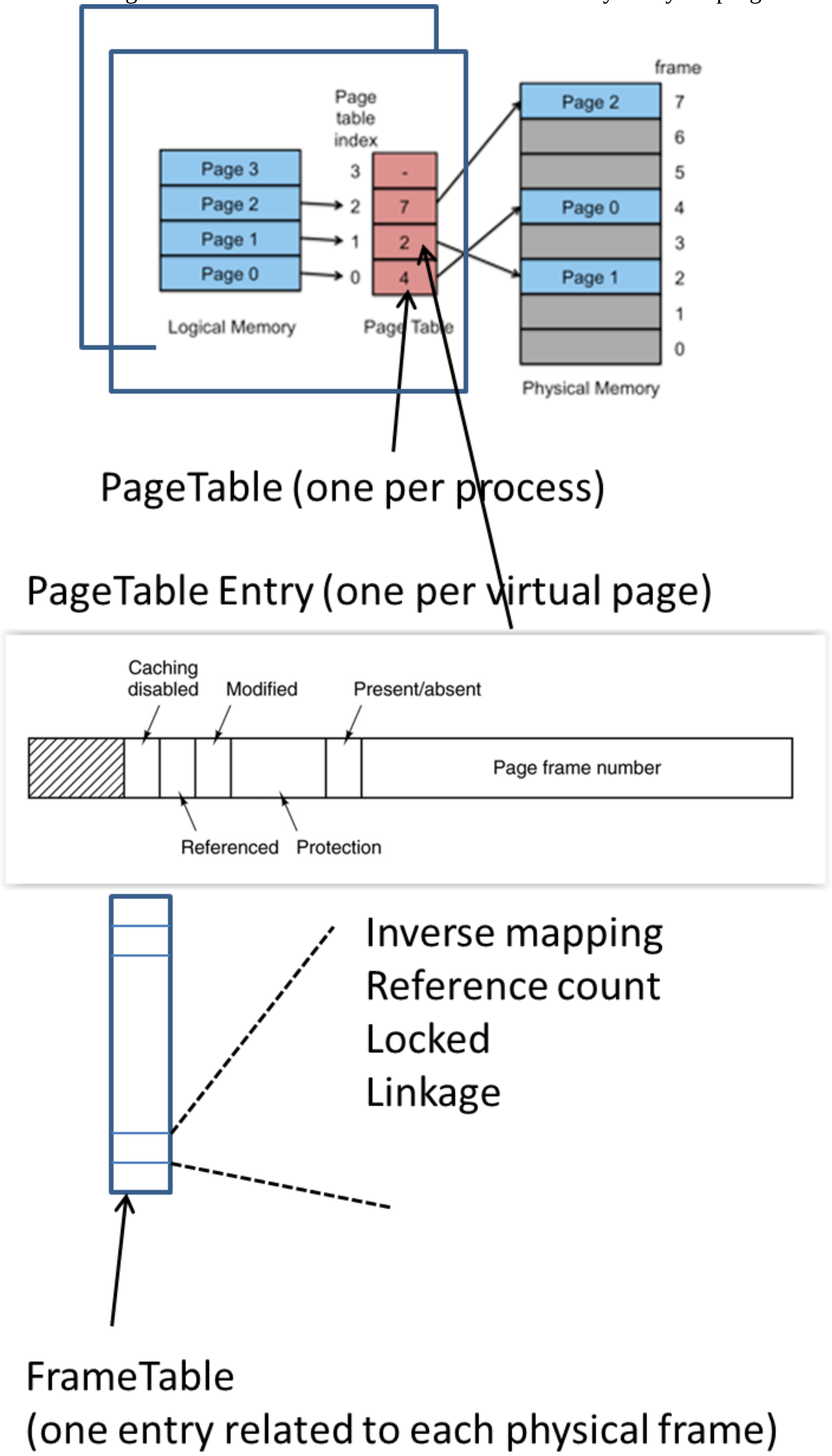
It is suggested that you follow this approach
a) read the input creating your processes and their respective vmas and print them out again, to ensure you read properly.
b) implement the features for a single process first (in1..in6) and implement one algorithm (e.g. fifo).
c) add the basic features for context switching (in7..in8) and then expand to other algorithms
d) Finally, try the more complex input files (in9..in10).

Sample output files are provided as a *.zip for each input and each algorithm for two frame number scenarios –f16 and –f32 → 10 * 2 * 6 → 120 files. If you need more you can generate your own outputs using the reference program on the cims account (under *~frankeh/Public/mmu)* for different frame numbers.

I also provide a ./runit.sh and a ./gradeit.sh. (change the INPUTS and ALGOS in both to limit what you want to test)
./runit.sh <inputs_dir> <output_dir> <executable>
./gradeit.sh <refout_dir> <output_dir> # will generate a summary statement and <output_dir>/LOG file for more details
Look at the LOG file and it shows the "diff command" for those files that failed and even if the TOTALCOST are the same there are differences elsewhere. Remove the '-q' flag in the diff command of the LOG and run manually.

<u>OTHER STUFF:</u>

The following data structures should somewhat find their way into your program:


frame
Page table index
Page 3
Page 2
Page 1
Page 0
3
2
1
0
-
7
2
4
Logical Memory
Page Table
Page 2
Page 0
Page 1
7
6
5
4
3
2
1
0
Physical Memory
PageTable (one per process)
PageTable Entry (one per virtual page)
Caching disabled
Modified
Present/absent
Page frame number
Referenced
Protection
Inverse mapping
Reference count
Locked
Linkage
FrameTable
(one entry related to each physical frame)