

In this lab/programming assignment you will implement/simulate the operation of an Operating System's Virtual Memory Manager that maps the virtual address spaces of multiple processes onto physical frames using page table translation. 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, you have to support up to 128 frames, tests will only use 128 or 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
 - i. the number of virtual memory areas / segments (aka VMAs)
 - ii. 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. The input always follows the format below, though number and location of lines with '#' might vary.

```
#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 that the VMAs of a 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_vpage - 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', 'w' or 'e') 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 virtual page <vpage> of the currently running process.

"w <vpage>": implies that a store/write operation is performed on virtual page <vpage> of the currently running process.

"e <procid>": current process exits, we guarantee that <procid> is the current running proc, so you can ignore it.

```
# ##### 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. E.g. you can use `sscanf(buf, "%d %d %d %d",...) OR stream >> var1 >> var2 >> var3.`

DATA STRUCTURES:

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

A page table naturally must contain exactly 64 page table entries (PTE). Please use constants rather than hardcoding "64". 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 (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). **What you can NOT do** is run at the beginning of the program through the page table and mark each PTE with bits based on filemap or writeprotect. This is NOT how OSes do this due to hierarchical pagetable structures (not implemented in this lab though). You can only set those bits on the first page fault to that virtual page.

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 reverse mappings to the process 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).

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 and the 'e' (process exit) instruction.

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() = 0;    // virtual base class
};

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

while (get_next_instruction(&operation, &vpage)) {
    // handle special case of "c" and "e" instruction
    // now the real instructions for read and write
    pte_t *pte = &current_process.page_table[vpage]; // in reality this is done by hardware
    if (!pte->present) {
        // this in reality will generate the page fault exception and now you
        // execute
        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 R/M PTE bits
    update_pte(read/modify/write) bits based on operations.
}
```

Classes for every paging algorithm should derive from a base Pager class and there should be no replication of the simulation code. This will allow adding new classes.

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, maybe you can find a faster way then searching each time the VMA list as long as it does not involve doing that before the instruction simulation (see above). 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 the 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" (you can choose a different FIFO data structure of course). 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 frame 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.

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 SEGV 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 frame table can only be accessed as part of the "simulated page fault handler" (see code above).

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

Algorithm	Based on Physical Frames
FIFO	f
Random	r
Clock	c
Enhanced Second Chance / NRU	e
Aging	a
Working Set	w

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, i.e. you are looping through the entire or parts of the frame table, 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 reverse mapping from frame to PTE that is using it. Provide this reverse mapping (frame \Rightarrow `<proc-id,vpage>`) inside each frame's frame table entry.

Note (again): you **MUST NOT** set any bits in the PTE before instruction simulation start, i.e. the pte (i.e. all bits) 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 and what bits are not occupied by the mandatory bits.

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

```
78: ==> r 2
UNMAP 1:42
OUT
IN
MAP 26
```

Output 1

```
69: ==> r 37
UNMAP 0:35
FIN
MAP 18
```

Output 2

```
75: ==> w 57
UNMAP 2:58
ZERO
MAP 17
```

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 <curprocid : 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 read from file (**FIN**) in to physical frame 18 (implies it is file mapped) and finally mapped (**MAP**). 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, 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. For 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 and process exit costs 175 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 %lu %llu\n", inst_count, ctx_switches, process_exits, 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 31 1 0 52951
```

Note, the cost calculation can overrun 2^{32} and you must account for that, so use 64-bit counters (unsigned long long). 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. 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)` 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 additional “aging” information during victim_select and after each instruction for complex algorithms (not all algorithms have the details described in more detail below)

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.**

(Note only a max of 10 processes and 8 VMAs per process are supported in the reference program which means that is the max we test with).

All scanning replacement algorithm typically continue with the `frame_index + 1` of the last selected victim frame.

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. Initializing the PTE before instruction simulation to anything but “0” will cost another 5 pts.

FAQ: (and lots of debugging help)

ESC/NRU requires that the REFERENCED-bit be periodically reset for all valid page table entries. The book suggests on every timer cycle which is way too often. Typically this is done at periodic times using a daemon. We simulate the periodic time inside the `select_victim_frame` function. If 50 or more instructions have passed since the last time the reference bits were reset, then the reference bit of the pte's reached by the frame traversal should be cleared after you considered the class of that frame/page. Also in the algorithm you only have to remember the first frame that falls into each class as it would be the one picked for that class. Naturally, when a frame for class-0 (R=0,M=0) is encountered, you should stop the scan, unless the reference bits need to be reset, at which point you scan all frames. Once a victim frame is determined, the *hand* is set to the next position after the victim frame for the next `select_victim_frame()` invocation.

The "-oa" option will produce the following output: (<before> | <after> scan):

```
ASELECT: 5 1 | 3 5 9
          ^ hand at beginning of select function
          ^ Boolean whether reference bit has to be reset (>= 50th instructions since last time)
            ^ Lowest class found [0-3]
              ^ Victim frame selected
                ^ Number of frames scanned
```

AGING requires to maintain the age-bit-vector. In this assignment please assume a 32-bit counter treated as a bit vector. Aging is implemented on every page replacement request. Since only active pages can have an age, it is fine (read recommended) to stick the age into the frame table entry. Once the victim frame is determined, the *hand* is set to the next position after the victim frame for the next `select_victim` invocation. Note the age has to be reset to 0 on each MAP operation.

The "-oa" option will produce the following output: (<before> | <during> | <after> scan):

```
ASELECT 2-1 | 2:40000000 3:10000000 0:20000000 1:80000000 | 3
          ^ scan from frame 2 to 1 (so there is a wrap around )
            ^ frame# and age (hexadecimal) after shift and ref-bit merge (while scanning the frames)
              ^ frame selected (3)
```

WORKING-SET: In the case of working set we also need to deal with time (tau). Again we use the execution of a single instruction as a time unit. If more than 50 time units (instructions) have passed since the value of tau recorded in the frame and the reference bit is not set then this frame will be selected (see algo). Note when you map a frame, you must set its tau value to the current time (instruction count).

The "-oa" option will produce the following output: (<before> | <during> | <after> scan)

```
ASELECT 3-2 | 3(0 3:7 5196) 0(0 3:31 5198) 1(0 0:34 5199) 2(1 0:10 0) | 3
          ^ scan from frame 3 to 2 (so there is a wrap around )
            ^ frame# (refbit, proc:vpage, tau) (while scanning the frames)
              frame selected (3) ^
```

```
ASELECT 44-43 | 44(1 2:18 0) 45(0 1:33 5140) STOP(2) | 45
          the scan stops if a frame with bigger tau is found ^
          and that frame will be selected, (2) is number of frames scanned and 45 is the frame selected
```

What to do on process exit:

On process exit (instruction), you have to traverse the active process's page table starting from 0..63 and for each valid entry UNMAP the page, FOUT modified filemapped pages. Note that dirty non-fmapped (anonymous) pages are not written back (OUT) as the process exits. The used frame has to be returned to the free pool and made available to the `get_frame()` function again. The frames then should be used again in the order they were released.

OTHER STUFF

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.

Optional arguments in arbitrary order ? This was shown in the sample programs `~frankeh/Public/ProgExamples.tz`

Please read: http://www.gnu.org/software/libc/manual/html_node/Example-of-Getopt.html (very useful and trivial to use)

Provided Inputs

The submission comes with a few generated inputs that you can use to address the 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	Proc Exits
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	5000	4	6	2	1	2	
in11	5000	4	6	3	1	2	2

It is suggested that you follow this approach:

- read the input creating your processes and their respective vmas and print them out again, to ensure you read properly.
- implement the features for a single process first (in1..in6) and implement one algorithm (e.g. fifo).
- add the basic features for context switching (in7..in8) and then expand to other algorithms
- Finally, try the more complex input files (in9..in11).

Sample output files are provided as a *.tar.Z for each input and each algorithm for two frame number scenarios –f16 and –f32 → 11 * 2 * 6 → 132 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 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.

If you runit and then gradeit you will get a matrix as follows .. any failing test will be marked as ‘x’.

```
$ ./gradeit.sh ../refout ../studentout/
input  frames  f  r  c  e  a  w
1      16      .  .  .  .  .  .
1      32      .  .  .  .  .  .
2      16      .  .  .  .  .  .
2      32      .  .  .  .  .  .
3      16      .  .  .  .  .  .
3      32      .  .  .  .  .  .
4      16      .  .  .  x  .  .
4      32      .  .  .  .  .  .
5      16      .  .  .  .  .  .
5      32      .  .  .  .  .  .
6      16      .  .  .  .  .  .
6      32      .  .  .  .  .  .
7      16      .  .  .  .  .  .
7      32      .  .  .  .  .  .
8      16      .  .  .  .  .  x
8      32      .  .  .  .  .  .
9      16      .  .  .  .  .  .
9      32      .  .  .  .  .  .
10     16      .  .  .  .  .  .
10     32      .  .  .  .  .  .
SUM                20 20 20 19 20 19
```