In this lab/programming assignment you will implement/simulate the operation of a **Virtual Memory Manager** which 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 exactly 64 virtual pages (yes this is small compared to the 1M entries for a full 32-address architecture), but the principal counts. As the sum of all virtual pages in all virtual address spaces may exceed the number of physical frames 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 only source code, makefile and make.log/grade.log files via NYU Brightspace as single a zip/tar/tar.Z file to avoid resubmission and declared penalties.

INPUT SPECIFICATION:

The input to your program will be a comprised of:

1. the number of processes (processes are numbered starting from 0)
2. is comprised of
   i. the number of virtual memory areas / segments (aka VMAs)
   ii. specification for each said VMA comprised of 4 numbers:
   ending_virtual_page write_protected[0/1]

Following is a sample input with two processes. **Note: ALL** 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. Processes in this sample have 2 and 3 VMAs, respectively. All provided inputs vary. Parsing should be straight forward with nested loops.

```
#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 41 0 0
42 63 1 0
#### process 1
3
0 17 0 1
20 60 1 0
62 63 0 0
```

Since it is required that the VMAs of a single address space do 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 properties:

;
; (note the VMA has (`end_vpage - start_vpage + 1`) virtual pages)
; VMA is write protected or not
; VMA is mapped to a file or not

The process specification is followed by a and optional comment lines (see following example). An instruction line is comprised of a character (or

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 nd no further context switches will be made to this process

```
##### example of an instruction sequence ######
c 0
r 22
w 19
r 11
r 57
```

You can assume that the input files are well formed as shown above, so fancy parsing is not required. Just make sure you take 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 input and create process objects, each with its array/vector/list of *vma*s 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). A PTE is comprised of the PRESENT=VALID, REFERENCED, MODIFIED, WRITE_PROTECT, and PAGEDOUT bits and the number of the physical frame (in case the pte is present). This information can and **must** be implemented as a single 32-bit value or as a bit structure (easier). It cannot be a structure of multiple integer values that collectively is 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 the first 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, plus 14 mandatory reserved bits, you effectively have 32-12-14 = 6 bits for your own usage in the pagetable entry. 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 will lead to deductions. This is NOT how OSes do this due to hierarchical pagetable structures (not implemented in this lab though). The pagetable is initialized to all ZERO. You can only set those bits on the first page fault for that faulting virtual page (vpage) and that vpage only.

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. Note, that in this assignment a frame can only be mapped by at most one PTE at a time, which simplifies things significantly.

SIMULATION and IMPLEMENTATION:

During each instruction you simulate the behavior of the hardware (shown below in **blue**) and hence you must check that the page is present. A special case are
page table pointer which exits a process.

General structure of the simulation

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

```
typedef struct { … } frame_t;
typedef struct { … } pte_t;          // can only be total of 32-bit size and we will check on this

frame_t frame_table[MAX_FRAMES];
pte_t   page_table[MAX_VPAGES];  // a per process array of fixed size=64 of pte_t  not pte_t pointers !

class Pager {  // this is the base class from which all others derive
       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();
```

the page is not present, as indicated by the associated present bit, the hardware would raise a page fault exception. Here you just simulate this by calling your ( ) 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, hint you have free bits in the PTE). 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

never swapped out and is not file mapped

That leaves the allocation of frames. All frames initially are in a free pool (use deque to get desired semantics). 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
derived class of a general *Pager* class with at least one `frame_t* select_victim_frame();` that returns a victim frame (or returns int for the frame number). Once a victim frame has been determined, the victim frame must be unmapped from its user (<address_space:vpage>)
inspect the state of the R and M bits. If the page was modified, then the page frame must be paged out to in case it was file mapped it has to be written back to
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 the valid bit can 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 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 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

The following page replacement algorithms are to be implemented (letter indicates program 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 code 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.

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 <proc- Each time you do a MAP from vpage->frame also create the reverse mapping from frame->vpage and similar break them when you do an UNMAP.

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 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; 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 faulting pte (and faulting

pte only) based on what you found in the VMA and what bits are not occupied by the mandatory bits (remember you have ~20 bits free here). To enforce this and to avoid misuse some of the 20 free bits for wrong things, **you must reserve 14 bits in the PTE for the OS in general, which leaves you 6 bits to play with.**

You are to create the following output if requested by an option (see at options description and set of options we grade with):

```
39: ==> r 15                69: ==> r 37                75: ==> w 57
 UNMAP 1:41                  UNMAP 0:35                  UNMAP 2:58
 OUT                         FIN                         ZERO
 IN                          MAP 18                      MAP 17
 MAP 26

Output 1                    Output 2                    Output 3
```

For instance, in Output 1 instruction 39 is a read operation on virtual page 15 of the current process. The replacement algorithms selected physical frame 26 that was used by virtual page 41 of process 1 (1:41) and hence we first have to **UNMAP** the virtual page 41 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 (1:41) tag so the Operating system can find it later when process 1 references vpage 41 . Then it pages **IN** the previously swapped out content of virtual page 15 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_15 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 . 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**) into physical frame 18 (implies it is file mapped) and finally mapped (**MA**P). In output 3 you see that frame 17 was selected forcing the unmapping of its current user process_2, vpage 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 virt 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 segv, segprot, unmap, map, pageins (IN, FIN), pageouts

Execution and Invocation Format:

Your program **must** follow the following invocation:
./mmu f<num_frames> -a<algo> [-o<options>] inputfile randomfile (arguments can be in any order → `getopt()`).
e.g. `./mmu -f4 -ac -oOPFS 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), run with varying

Test input files and the file with random numbers are supplied (same as lab2). 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 on overflow.

- ooooh nooooh) option shall generate the required output as shown in output-1/3.
-
  As a single line for each process, you print the content of the pagetable pte entries as follows (shown for process 0).

  ```
  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)
  from the specified VMA.
  PTEs that are not valid are represented by

  (valid) indicates -
  Note a virtual page, that was once referenced, but was not modified and then is selected by the replacement algorithm,

- after the execution and should show which frame is mapped at the end to which <pid: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
  ```

- TOTALCOST
- current page table after each instructions (see example outputs) and this should help you significantly to track down bugs and transitions (remember you write the print function only once)
- processes instead.
-
- additional aging information during victim_selection and after each instruction for complex algorithms (not all algorithms have the details described in more detail below)

- - -x,-y s during the grading. **It is purely for your benefit to add these and compare with the reference program under ~frankeh/Public/lab3/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.

DEDUCTIONS: 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**. **will cost another 5 pts**, because it is so fundamentally different from how OSs work. The right way is to set required flags on the pte during the page fault and then only on that faulting page. **Another 2pts** for not printing the real size of pte_t from your **Another 2 pts** if you do not reserve 14 bits in the PTE.

Another 2 pts each for wrong pagetable and frame table organizations. They are both arrays/vectors of PTEs and FRAMES, not of pointers to PTE or FRAMES. Finally, the same deductions for not providing the make.log and grade.log files as in previous labs.

FAQ: (and lots of debugging help by the reference program)

FIFO we are not implementing a strict FIFO due to the intricacies of the free pool (see get_next_frame()) where on process exit frames will be returned and have to be used first again before calling the select_victim_frame() function. So please

another victim_frame. Then this can be also expanded to do the clock algorithm by simply dealing with referenced frames and resetting the R bit. - `ASELECT: 5` where the number indicates where the hands starts.

CLOCK See above: implement as the derivative of the FIFO implementation.
- `ASELECT: 5 15` where the first number indicates where the hands starts and the 2nd number indicates how many frames where inspected to finally find one where the reference bit = 0 .

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 48 or more instructions have passed since the last time the reference bits were reset, then the reference bit 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 continue to 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. Try to walk the frametable only once using Booleans.

- (<before> | <after> scan):

```
ASELECT: 5 1 | 3  5  9
          ^ hand at beginning of select function
            ^ Boolean whether reference bit has to be reset (>= 48th 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 unsigned counter treated as a bit vector. `age = age >> 1` `age = (age | 0x80000000)` one. 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. Use a virtual pager function at the top to reset its value, where others algos simply implement a default noop for that function.

- : (age is hexadecimal)

```
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. Again we use the execution of a single instruction as a time unit . We assume TAU=49, so if 50 or more instructions have passed since the time of last use was 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 time of last use to the current time (instruction count).

-

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

```
ASELECT 44-43 | 44(1 2:18 2340) 45(0 1:33 5140) STOP(2) | 45
```

the scan stops if a frame is found with condition ^
and that frame will be selected, (2) is number of frames scanned and 45 is the frame selected.
_last_used

What to do on Process Exit:

On process exit (instruction), you have to
UNMAP the page and 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

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/lab3/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 <output_dir> <yourexecutable>`
`./gradeit.sh <refout_dir> <output_dir>` # will generate a summary statement and <output_dir>/LOG.txt file for more details.
Look at the LOG file and it shows the

- .txt and run manually.

and you can run a manual diff on that particular case. See the LOG.txt file in your output directory for the two file names involved.

```
$ ./gradeit.sh ./refout ../studentout/
input  frames    f  r  c  e  a  w
1       16       .  .  .  .  .  .
1       31       .  .  .  .  .  .
2       16       .  .  .  .  .  .
2       31       .  .  .  .  .  .
3       16       .  .  .  .  .  .
3       31       .  .  .  .  .  .
4       16       .  .  .  x  .  .
4       31       .  .  .  .  .  .
5       16       .  .  .  .  .  .
5       31       .  .  .  .  .  .
6       16       .  .  .  .  .  .
6       31       .  .  .  .  .  .
7       16       .  .  x  .  .  .
7       31       .  .  .  .  .  .
8       16       .  .  .  .  .  x
8       31       .  .  .  .  .  .
9       16       .  .  .  .  .  .
9       31       .  .  .  .  .  .
10      16       .  .  .  .  .  .
10      31       .  .  .  .  .  .
SUM             20 20 19 19 20 19
```

I suggest 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 and generate the desired output.

b) implement PTE, FTE, pagetables and frametable implement the page table and frame table output, you need these for debugging and final printout anyway. Not there should be no trailing <space> after a PTE/FTE entry, the <space> is before (common error that rises when we use `cmp` instead of `diff` for grading.

c) implement the features for a single process first (in1..in6) and implement one algorithm (e.g. fifo).

d) add the basic features for context switching (in7..in8) and then expand to other algorithms.

e) try the more complex input files (in9..in10).

f) finally handle the process exit instructions (in11) as discussed above.

g) be ambitious, generate your own inputs with many more instructions and run against your code and reference program.

so to only run specific input files and algorithms during development.

```
INPUTS=`seq 1 11`
ALGOS="f r c e a w"
FRAMES="16 31"
```

you run the program slightly different then lab1/lab2

```
cd scripts
./runit.sh   <youroutputdir> <yourprogram>
./gradeit.sh ./refout <youroutputdir>
```

Or change them as command prefixes to overwrite the defaults in the script

```
INPUTS=”2 4” ALGOS=”f r” FRAMES=”17” ./runit.sh   <youroutputdir> <yourprogram>
INPUTS=”2 4” ALGOS=”f r” FRAMES=”17” ./gradeit.sh ./refout <youroutputdir>
```

Generating your own sample outputs.

There is a simple generator under ~/Public/lab3/mmu_generator
Type `mmu_generator - -p` argument (only for professor).
Take the input table as a guidance as those inputs were generated with this tool as well. Note, however, it is not the most robust tool.

Program Accuracy and Efficiency

crunchy machines
cmp does a byte by byte comparison This is often most apparent in the printing of the page/frame table, where you might have trailing space while as the reference implements the spaces before the token.

USEDIFF=0 USEDIFF= ,

If your program runs too long, we have to kill it. There are 360 grading test cases and for the test cases provided I would expect each of them to take no longer than a second (many simple test cases take 50-100msec). While we set the limit higher (**30seconds**) to catch the 1M+ instruction runaway case, the burden is on you to create a somewhat performant program and you can extrapolate how long 1M+ instructions will run from the 5000 instructions use case provided in in10/in11 inputs or even better create a file with large instructions.

You can time your individual run as follows

```
time   mmu -f24 -aw -oOPFS ../inputs/in11 ../inputs/rfile > /tmp/out
```

For reference, the (what I think) most time consuming test case (1M , a or w algo, large number of frames) is run by the reference program in 1.25sec.

real 0m1.264s
user 0m1.161s
sys 0m0.094s

To enable whether process time is monitored/killed or not (default is not), you can change the parameter TIMELIMIT in the runit.sh script similarly to the USEDIFF.

You can also run these programs as follows without having to change the scripts.

```
TIMELIMIT=10 ./runit.sh ../myoutdir myschedulerexe  # kills programs after 10 secs
USEDIFF=1 ./gradeit.sh ./refout ../myoutdir  # use diff instead of cmp
```

Good Luck