In this lab/programming assignment you will implement/simulate the operation of an Operating System's 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 principle 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 your source and Makefile 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 comprised 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. Processes in this sample have 2 and 3 VMAs, respectively. **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. All provided inputs follow 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
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 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 indicate 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.

```
##### example of an instruction sequence  ######
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 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). Please use constants rather than hardcoding "64". 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, you effectively have 32-12 = 20 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 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. 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 the 'c' (context switch) instruction which simply changes the current process and current page table pointer and the 'e' (process exit) instruction which exits a process.

Structure of the simulation

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

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

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 {
        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  generates  the  page  fault  exception  and  now  you  execute
           // verify this is actually a valid page in a vma if not raise error and next inst
           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.
        }
       // check write protection
       // simulate instruction execution by hardware by updating the R/M PTE bits
       update_pte(read/modify) bits based on operations.
}
```

When accessing a page (“r” or “w”) and the page is not present, as indicated by the associated PTE’s valid/present 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, 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 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 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 according to the algorithm’s policy. This should be implemented as a derived class of a general *Pager* class with at least one virtual function “`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>), i.e. its entry in the owning process’s page_table must be removed (“UNMAP”), however you must inspect the state of the R and M 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 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 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 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 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-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; 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 (remember you have ~20 bits free here).

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

```
49: ==> r 4              69: ==> r 37              75: ==> w 57
 UNMAP 1:42               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 49 is a read operation on virtual page 4 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 4 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_4 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**) 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 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 segv, segprot, unmap, map, pageins (IN, FIN), pageouts (OUT, FOUT), and zero operations for each process and instructions, process exits, context switches globally. You have to compute the overall execution time in cycles, where the cost of operations (in terms of cycles) are as follows: read/write (load/store) instructions count as 1, context_switches instructions=130, process exits instructions=1250. In addition if the following operations counts as follows:
maps=300, unmaps=400, ins=3100, outs=2700, fins=2800, fouts=2400, zeros=140, segv=340, segprot=420

Per process output:

```
    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);
```

Summary output:

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

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 4
```

Note, the total 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. If you use individual 32-bit counters, don't add up 32-bit numbers all at once and then assign to 64-bit, this will result in overflows. Add 32-bit numbers incrementally to the 64-bit counters.

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 page frame numbers and "diff" compare it to the expected output.

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.

- The 'O' (ooooh nooooh) 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 (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), M (modified), S (swapped out) (note we don't show the write protection bit as it is implied/inherited from the specified VMA.
  PTEs 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' option prints the 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)
- 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_selecton 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' options 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.

<u>**Please note again:**</u> 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**, because it is so fundamentally different then OSs work. **Another 3pts** for not printing the real size of pte_t from your program which must show "4" indicating your pte indeed is 32bit.

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 simply use a "hand" (read index into the frame table) that is incremented each time (don't forget wraparound) you pick 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. The option "-oa" will produce `ASELECT: 5` where the number indicates where the hands starts.

CLOCK See above: implement as the derivative of the FIFO implementation.
The option "-oa" will produce `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 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 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.

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 unsigned counter treated as a bit vector. Note: "`age = age >> 1`" shifts the age to the right and "`age = (age | 0x80000000)`" sets the leading bit to 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.

The "-oa" option will produce the following output: (<before> | <during> | <after> scan): (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 as we don't have anything else in this simulation. 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).
The "-oa" option will produce the following output: (<before> | <during> | <after> scan)

```
  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.
```

if no frame matches the time condition the one with the oldest "time_last_used" will be 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 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/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> <yourexecutable>`
`./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
```

How to approach this lab

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 the page table and frame table output, you need these for debugging and final printout anyway.

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.

You can modify “runit.sh” and “gradeit.sh” so to only run specific input files and algorithms.

```
INPUTS=`seq 1 11`
ALGOS="f r c e a w"
FRAMES="16 32"
```

you run the program slightly different then lab1/lab2

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

Programming Assignment #4 (Lab 4): IO Scheduling **Professor Hubertus Franke**
Class CSCI-GA.2250-001/002 Spring2021

In this lab you implement/simulate the scheduling and optimization of I/O operations. Applications submit their IO requests to the IO subsystem (potentially via the filesystem and page-cache), where they are maintained in an IO-queue until the disk device is ready for servicing another request. The IO-scheduler then selects a request from the IO-queue and submits it to the disk device. This selection is commonly known as the `strategy()` routine in operating systems. On completion, another request can be taken from the IO-queue and submitted to the disk. The scheduling policies will allow for some optimization as to reduce disk head movement or overall wait time in the system. The schedulers to be implemented are FIFO (i), SSTF (j), LOOK (s), CLOOK (c), and FLOOK (f) (the letters in bracket define which parameter must be given in the –s program flag).

You are to implement these different IO-schedulers in C or C++ and submit the **source** code, which we will compile and run. Your submission must contain a Makefile so we can run on **linserv*.cims.nyu.edu** (and please at least test there as well).

Invocation is a follows:

```
./iosched –s<schedalgo>  [options] <inputfile>
```

// options are -v -q -f (see below)

The input file is structured as follows: Lines starting with '#' are comment lines and should be ignored.
Any other line describes an IO operation where the 1st integer is the time step at which the IO operation is issued and the 2nd integer is the track that is accesses. Since IO operation latencies are largely dictated by seek delay (i.e. moving the head to the correct track), we ignore rotational and transfer delays for simplicity. The inputs are well formed.

```
#io generator
#numio=32 maxtracks=512 lambda=10.000000
1 430
129 400
:
```

We assume that moving the head by one track will cost one time unit. As a result, your simulation can/should be done using integers. The disk can only consume/process one IO request at a time. Everything else must be maintained in an IO queue and managed according to the scheduling policy. The initial direction of the LOOK algorithms is from 0-tracks to higher tracks. The head is initially positioned at track=0 at time=0. Note that you do not have to know the maxtrack (think SCAN vs. LOOK).

Each simulation should print information on individual IO requests followed by a SUM line that has computed some statistics of the overall run. (see reference outputs).

For each IO request create an info line (5 requests shown).

```
    0:      1      1    431
    1:     87    467    533
    2:    280    431    467
    3:    321    533    762
    4:    505    762    791
```

Created by

```
printf("%5d: %5d %5d %5d\n",i, req->arrival_time, r->start_time, r->end_time);
```

- IO-op#,
- its arrival to the system (same as inputfile)
- its disk service start time
- its disk service end time

Please remember " %5d" is not "%6d" !!!

and for the complete run a SUM line:

total_time: total simulated time, i.e. until the last I/O request has completed.
tot_movement: total number of tracks the head had to be moved
avg_turnaround: average turnaround time per operation from time of submission to time of completion
avg_waittime: average wait time per operation (time from submission to issue of IO request to start disk operation)
max_waittime: maximum wait time for any IO operation.

Created by:
```
printf("SUM: %d %d %.2lf %.2lf %d\n",
        total_time, tot_movement, avg_turnaround, avg_waittime, max_waittime);
```

Various sample inputs and outputs are provided with the assignments on NYU classes.
Please look at the sum results and identify what different characteristics the schedulers exhibit.

You can make the following assumptions:
- at most 10000 IO operations will be tested, so its OK (recommended) to first read all requests from file before processing.
- all io-requests are provided in increasing time order (no sort needed)
- you never have two IO requests arrive at the same time (so input is monotonically increasing)

I strongly suggest, you do not use discrete event simulation this time. You can write a loop that increments simulation time by one and checks whether any action is to be taken. In that case you have to check in the following order.

1) If a new I/O arrived to the system at this current time → add request to IO-queue ?
2) If an IO is active and completed at this time → Compute relevant info and store in IO request for final summary
3) If an IO is active but did not yet complete → Move the head by one sector/track/unit in the direction it is going (to simulate seek)
4) If no IO request active now (after (2)) but IO requests are pending → Fetch the next request and start the new IO.
5) If no IO request is active now and no IO requests pending → exit simulation
6) Increment time by 1

When switching queues in FLOOK you always continue in the direction you were going from the current position, until the queue is empty. Then you switch direction until empty and then switch the queues continuing into that direction and so forth. While other variants are possible, I simply chose this one this time though other variants make also perfect sense.

Additional Information:

As usual, I provide some more detailed tracing information to help you overcome problems. Note your code only needs to provide the result line per IO request and the ‘SUM line’.

The reference program under ~frankeh/Public/iosched on the cims machine implements three options –v, -q, -f to debug deeper into IO tracing and IO queues.

The **–v** execution trace contains 3 different operations (***add*** a request to the IO-queue, ***issue*** an operation to the disk and ***finish*** a disk operation). Following is an example of tracking IO-op 18 through the times 1139..1295 from submission to completion.

```
1139:   18 add 211            //  18 is the IO-op # (starting with 0) and 211 is the track# requested
1127:   18 issue 211 279      //  18 is the IO-op #, 211 is the track# requested, 279 is the current track#
1195:   18 finish 68          //  18 is the IO-op #, 68 is total length/time of the io from request to completion
```

-q shows the details of the IO queue and direction of movement (1==up , -1==down) and
–f shows additional queue information during the FLOOK.

> Here Queue entries are tuples during add [ior# : #io-track] or triplets during get [ior# : io-track# : distance], where distance is negative if it goes into the opposite direction (where applicable).

Please use these debug flags and the reference program to get more insights on debugging the ins and outs (no punt intended) of this assignment and answering certain “why” questions.

Generating your own input for further testing:

A generator program is available under ~frankeh/Public/iomake and can be used to create additional inputs if you like to expand your testing. You will have to run this against the reference program ~frankeh/Public/iosched yourself.

Usage: iomake [-v] [-t maxtracks] [-i num_ios] [-L lambda] [-f interarrival_factor]

maxtracks is the tracks the disks will have, default is 512
num_ios is the number of ios to generate, default is 32
lambda is parameter to create poisson distribution, default is 1.0 (consider ranges from 0.1 .. 10.0)
interarrival_factor is time factor how rapidly io will arrive, default is 1.0 (consider values 0.5 .. 1.5), to small and the system will be overloaded and to large it will be underloaded and scheduling is mute as often only one i/o is outstanding.

Below are the parameters for the 15 inputs files provide in the assignment:

```
#numio=10   maxtracks=128  lambda=0.100000
#numio=20   maxtracks=512  lambda=0.500000
#numio=50   maxtracks=128  lambda=0.500000
#numio=100  maxtracks=512  lambda=0.500000
#numio=50   maxtracks=256  lambda=5.000000
#numio=20   maxtracks=256  lambda=2.000000
#numio=100  maxtracks=512  lambda=9.000000
#numio=80   maxtracks=300  lambda=3.500000
#numio=80   maxtracks=1000 lambda=3.500000
#numio=500  maxtracks=512  lambda=2.500000
#numio=51   maxtracks=300  lambda=4.000000
#numio=99   maxtracks=312  lambda=2.200000
#numio=30   maxtracks=10   lambda=0.300000
#numio=1000 maxtracks=648  lambda=2.500000
#numio=999  maxtracks=999  lambda=2.500000
#numio=200  maxtracks=512  lambda=2.500000
```