CS 4760 **Operating Systems**
Assignment # 6 Due Date: Dec 10th, 2025

Memory Management

Purpose

The goal of this homework is to learn about page replacement inside an operating system. You will work on the specified strategy to manage page tables and memory requests and take care of any possible page misses and page replacements

This project will strip the request for resouces out of the previous project and instead replace it with memory requests. User processes will instead send a message requesting a particular memory address (and whether it is a read or write). This project should share a lot of code with project 5, as we are essentially swapping out resource management and deadlock prevention and instead doing paging and page replacement. The overall structure and messaging is mostly the same, except for the content of the messages.

Task

In particular, we will be implementing the (FIFO)) page replacement algorithm. When a page-fault occurs, it will be necessary to swap in that page. If there are no empty frames, your algorithm will select the frame that was first filled and replace it. This will require that you keep a queue of which frames were filled.

Each frame should also have an additional dirty bit, which is set on writing to the frame. This bit is necessary to consider dirty bit optimization when determining how much time is taken when we have to swap a page out. The dirty bit is set everytime a page in that frame is written to, but is initially set to 0 when a page gets put in the frame.

Operating System Simulator

This will be your main program and serve as the master process. You will start the operating system simulator (call the executable `oss`) as the main process who will fork multiple children at specific times as in previous projects. You will take in the same command line options as the previous project.

oss should be called as follows:

```
oss [-h] [-n proc] [-s simul] [-t timeLimitForChildren]
    [-i factionOfSecondToLaunchChildren] [-f logfile]
```

The -i is as previous, indicating how often we decide to launch another one, as long as it does not violate our limits. The -t parameter serves the same function as in project 5.

The -f parameter is for a log file, where you should write the output of oss (NOT the workers). The output of oss should both go to this log file, as well as the screen. This is for ease of use for this project, so yes I realize your output will be done in two places. Again, THE WORKER OUTPUT SHOULD NOT GO TO THIS FILE.

In the beginning, `oss` will allocate memory for system data structures, including page table. You will also need to create a fixed size array of structures for a page table for each process, with each process having 16k of memory and so requiring 16 entries as the pagesize is 1k. The page table can be implemented as a separate data structure or each process in your process table can have its page table as part of that structure.

Assume that your system has a total memory of 64K. You will require a frame table of 64 structures, with any data required such as the dirty bit contained in that structure. These bits can be implemented using whatever type you want (int, char, bool).

After the resources have been set up, allow new processes in your system just like project 5. Your user processes execute concurrently and there is no scheduling performed. They run in a loop constantly till they have to terminate.

`oss` will do a non-blocking message receive and if it gets one, look at the memory requested. If the reference results in a page fault, the process will be *blocked* till the page has been brought in. If there is no page fault, `oss` just increments the clock by 100 nanoseconds (just so if the user process looks at the clock, it is different than when it sent the request) and sends a message back. In case of page fault, `oss` queues the request to the device (ie: we simulate it being blocked). Each request for disk read/write takes about 14ms to be fulfilled. In case of page fault, the request is queued for the device and the process is suspended as no message is sent back and so the user process is just waiting on a msgrcv (ie: similar to project 5 wait time, where it has to wait for that event to happen). The request at the head of the queue is *fulfilled* once the clock has advanced by disk read/write time since the time the request was found at the head of the queue. The fulfillment of request is indicated by showing the page in memory in the page table. `oss` should periodically check if all the processes are queued for device and if so, advance the clock to fulfill the request at the head. We need to do this to resolve any possible soft deadlock (not an actual deadlock, we aren't doing deadlock detection for this here!) in case memory is low and all processes end up waiting.

When a page is referenced, `oss` performs other tasks on the page table as well such as updating the page reference, setting up dirty bit, checking if the memory reference is valid (all will be valid for this project) and whether the process has appropriate permissions on the frame (we aren't doing frame locking, so all should be valid here), and so on.

When a process terminates, `oss` should log its termination in the log file and also indicate its effective memory access time. `oss` should also print its memory map every second showing the allocation of frames and each processes page table. You can display unallocated frames by a period and allocated frame by a + or in some other shorthand but I must be able to see the frames and page tables.

For example at least something like...

```
oss: P2 requesting read of address 25237 at time xxx:xxx
oss: Address 25237 in frame 13, giving data to P2 at time xxx:xxx
oss: P5 requesting write of address 12345 at time xxx:xxx
oss: Address 12345 in frame 203, writing data to frame at time xxx:xxx
oss: P2 requesting write of address 03456 at time xxx:xxx
oss: Address 12345 is not in a frame, pagefault
oss: Clearing frame 107 and swapping in p2 page 3
oss: Dirty bit of frame 107 set, adding additional time to the clock
oss: Indicating to P2 that write has happened to address 03456

Current memory layout at time xxx:xxx is:
         Occupied   DirtyBit  Process   Page
Frame 0: No         0         -1        -1
Frame 1: Yes        1         0         2
Frame 2: Yes        1         1         4
Frame 3: Yes        1         1         5
...

P0 page table: [ -1 -1 1 17 13 78 85 ... ]
P1 page table: [ 85 102 37 18 2 3 110 ... ]
...
```

where `Process` and `Page` indicates what processes page is in that frame and the dirty bit indicates if the frame has been written to.

Make sure to release any resources claimed by a process when it is terminated.

Important: oss should be incrementing the clock each time before sending a message to a user process or each iteration of the loop. You want to make sure that time keeps advancing in your system.

Overview of the procedure

```
while (stillChildrenToLaunch or childrenInSystem) {

        increment the clock by some amount

        determine if we should launch a child

        check to see if a blocked process should be unblocked
        If so, grant the process the request

        do a non-blocking message receive from any children

        if (msg from child) {
           if (request)
            if (its a pagefault) {
                 block it for 14 ms
            }
             else {
                  send it a message back indicating
                  it got its request
             }
           if (terminate)
             free up its page table and all its frames
        }

        Every half a second, output the frame table,
        page table and process table to the logfile and screen

}

Output total number of pagefaults, total number of reads,
total number of writes and the percentage of pagefaults
```

User Processes

The user processes will go in a loop, sending messages to oss indicating they want to make a memory request.

Each user process generates memory references to one of its locations. When a process needs to generate an address to request, it simply generates a random value from 0 to the limit of the pages that process would have access to (16).

Now you have the page of the request, but you need the offset still. Multiply that page number by 1024 and then add a random offset of from 0 to 1023 to get the actual memory address requested. Note that we are only simulating this and actually do not have anything to read or write.

Once this is done, you now have a memory address, but we still must determine if it is a read or write. Do this with randomness, but bias it towards reads. This information (the address requested and whether it is a read or write) should be conveyed to `oss`. The user process will do a msgrcv waiting on a message back from `oss`. `oss` checks the page reference by extracting the page number from the address, increments the clock as specified above, and sends a message back.

User processes should terminate in the same fashion that we used for project 5.

I suggest you implement these requirements in the following order:

1. Get a makefile that compiles two source files, have oss allocate shared memory, use it, then deallocate it. Make sure to

check all possible error returns.

2. Get oss to fork off and exec one child and have that child attach to shared memory and check the clock and verify it has correct resource limit. Then test having child and oss communicate through message queues. Set up PCB and frame table/page tables
3. Have child request a read/write of a memory address (just using the first scheme) and have oss always grant it and log it.
4. Set up more than one process going through your system, still granting all requests.
5. Now start filling out your page table and frame table; if a frame is full, just empty it (indicating in the process that you took it from that it is gone) and grant the request.
6. Implement a wait queue for I/O delay on needing to swap a process out.
7. Do not forget that swapping out a process with a dirty bit should take more time on your device
8. Implement the LRU scheme

Termination Criterion

`oss` should stop generating processes if it has already generated up to n processes, or if more than 5 real-time seconds have passed. If you stop adding new processes, the system should eventually have no children and then, it should terminate. Tune your parameters so that the system is able to encounter a deadlock state and that is shown in log file.

Criteria for Success

Make sure that the code adheres to specifications. Document the code appropriately to show where the specs are implemented. *You must clean up after yourself.* That is, after the program terminates, whether normally or by force, there should be no shared memory, semaphore, or message queue that is left allocated to you.

Grading

1. *Overall submission: 30pts*. Program compiles and upon reading, seems to solve the assigned problem in the specified manner.
2. *README/Makefile: 10pts*. Ensure that they are present and work appropriately.
3. *Code readability: 10pts*. Code should be readable with appropriate comments. Author and date should be identified.
4. *Conformance to specifications: 50pts*. Algorithm is properly implemented and documented.

Submission

Handin an electronic copy of all the sources, `README`, Makefile(s), and results. Create your programs in a directory called *username*.6 where *username* is your login name on opsys. Once you are done with everything, *remove the executables and object files*, and issue the following commands:

```
chmod 700 username.6

cp -p -r username.6 /home/hauschildm/cs4760/assignment6
```