CS 4760 **Operating Systems**
Assignment # 6 Due Date: Dec 7th, 2023

Memory Management

In this project we will be designing and implementing a memory management module for our Operating System Simulator `oss`. This project should share a lot of code with project 5, as we are essentially swapping out resource management and deadlock detection and instead doing paging and page replacement. The overall structure and messaging is mostly the same, except for the content of the messages.

In particular, we will be implementing the FIFO (CLOCK) page replacement algorithms. 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 victim frame based on our FIFO replacement policy. This treats the frames as one large circular queue.

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 these operations take. The dirty bit will be implemented as a part of the page table.

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 one main process who will create new children exactly like project 5. We still have as our clock the 2 integers in shared memory.

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 32k of memory and so requiring 32 entries as the pagesize is 1k. The page table should have all the required fields that may be implemented by bits or character data types.

Assume that your system has a total memory of 256K. You will require a frame table of 256 structures, with any data required such as reference byte and dirty bit contained in that structure.

After the resources have been set up, allow new processes in your system just like project 5. Make sure that you never have more than 18 user processes in the system. If you already have 18 processes, do not create any more until some process terminates (this is probably specified by your -s parameter, but never let that get above 18). Thus, if a user specifies an actual number of processes as 30, your hard limit will still limit it to no more than 18 processes at any time in the system. Your user processes execute concurrently and there is no scheduling performed. They run in a loop constantly till they have to terminate.

`oss` will monitor all memory references from user processes (ie: messages sent to it from user processes) and 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 4 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.

While 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. You can display unallocated frames by a period and allocated frame by a `+`.

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  HeadOfFIFO
Frame 0: No         0
Frame 1: Yes        1
Frame 2: Yes        0         *
Frame 3: Yes        1
```

where `Occupied` indicates if we have a page in that frame, the * indicates that the next page will be replaced there, and the dirty bit indicates if the frame has been written to.

0.1 Overview of the procedure

```
while (stillChildrenToLaunch or childrenInSystem) {
       do a nonblocking waitpid to see if a
       child process has terminated
        if so, free up its resources

       determine if we should launch a child

       check to see if event wait for a child is
       now finished and it gets granted its request
       ie: Its page is swapped in

       check if we have a message from a child

       if (msg from child) {
         if (not a pagefault)
          send a message back
         if (pagefault)
          set up its waiting for an event
       }

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

```
}
```

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 (32).

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.

At random times, say every 1000 ± 100 memory references, the user process will check whether it should terminate. If so, all its memory should be returned to `oss` and `oss` should be informed of its termination.

The statistics of interest are:

- Number of memory accesses per second overall in the system
- Number of page faults per memory access

You should terminate after more than 100 processes have gotten into your system, or if more than 5 real life seconds have passed. Make sure that you have signal handling to terminate all processes, if needed. In case of abnormal termination, make sure to remove shared memory, queues and semaphores.

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 FIFO scheme

Deliverables

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
```

Do not forget `Makefile` (with suffix rules), version control, and `README` for the assignment. If you do not use version control, you will lose 10 points. Omission of a `Makefile` (with suffix rules) will result in a loss of another 10 points, while `README` will cost you 5 points.