Virtual Memory Simluation Assignment

Virtual memory allows a process of P pages to run in F frames, even if F < P. This is achieved by use of a page table, which records which pages are in RAM in which frames, and a page fault mechanism by which the memory management unit (MMU) can ask the operating system (OS) to bring in a page from disk. The page table must be accessible by both the MMU and the OS, and an IPC facility is needed for communication between the MMU and OS. Given the MMU simulator below, write a program that does the work of the OS to maintain a process' use of RAM and the page table. The page table is held in shared memory, and signals are used for IPC. You do not need to simulate the RAM, only the page table maintenance. However, to make things realistic, your OS process must `sleep(1)` whenever a disk access (write out page to disk, read in page from disk) would be necessary in a real implementation.

The Page Table

The page table has four fields in each page table entry:

- A Boolean `Valid` indicating if the page of that index is in RAM.
- An integer `Frame` giving the frame number of the page in RAM.
- A Boolean `Dirty` indicating if the page has been written to.
- An integer `Requested` which is non-zero only if that page is not in RAM and has been requested by the MMU. In this case it's value is the PID of the MMU.

The OS process must create the page table in shared memory, and initialize it to indicate that no pages are loaded (all `Valid` fields set to `0`). You may need to add more fields for your OS, with corresponding changes in the MMU.

The MMU

This memory management unit simulator takes several arguments:

- The number of pages in the process.
- A reference string of memory accesses, each of the form <mode><page>, e.g., `W3` is a write to page 3.
- The PID of the OS process.

The MMU attaches to the shared memory (using the OS PID on the command line as the key), then runs through the reference string. For each memory access, the MMU:

1. Checks if the page is in RAM.
2. If not in RAM, writes its PID into the `Requested` field for that page.
3. Simluates a page fault by signalling the OS with a `SIGUSR1`.
4. Blocks until it receives a `SIGCONT` signal from the OS to indicate that the page has been loaded (well, as mentioned above, the load is not done in this project, just simulated by `sleep(1)` delays).
5. If the access is a write access, sets the `Dirty` bit.
6. Prints the updated page table.

When all memory accesses have been processed, the MMU detaches from the shared memory and signals the OS one last time, but without placing its PID in any `Requested` field. That must be

detected by the OS, at which point it can destroy the shared memory and exit.

The OS

The OS simulator must take two arguments:

- The number of pages in the process.
- The number of frames allocated to the process.

For simplicity, assume that the pages and frames are numbered 0, 1, 2, ...

After creating and initializing the page table in the shared memory, the OS must sit in a loop waiting for a `SIGUSR1` signal from the MMU. When it receives a signal, it must:

1. Scan through the page table looking for a non-zero value in the `Requested` field.
2. If a non-zero value is found, it's the PID of the MMU, and indicates that the MMU wants the page at that index loaded.
3. If there is a free frame allocate the next one to the page.
4. If there are no free frames, choose a victim page.
    - If the victim page is dirty, simulate writing the page to disk by `sleep(1)` and increment the counter of disk accesses.
    - Update the page table to indicate that the victim page is no longer `Valid`.
5. Simulate the page load by `sleep(1)` and increment the counter of disk accesses.
6. Update the page table to indicate that the page is `Valid`, in the allocated `Frame`, not `Dirty`, and clear the `Requested` field.
7. Print the updated page table.
8. Send a `SIGCONT` signal to the MMU to indicate that the page is now loaded.
9. If no non-zero `Requested` field was found, the OS exits the loop.

You can use whatever algorithm you want for choosing a victim page. You may need to add extra fields to the page table, and make *minor* modifications to the MMU code (ask me if you are unsure if what you are doing is *minor*). Before terminating the OS must print out the total number of disk accesses, and destroy the shared memory.

Sample Run

Here's a sample run. I have spaced things out so you can see the sequence of events in time.

```
> OS 5 2
The shared memory key (PID) is 78801
Initialized page table
                                          > MMU 5 W2 R3 W3 R4 78801
                                          Initialized page table:
                                           0: Valid=0 Frame=-1 Dirty=0 Requested=0
                                           1: Valid=0 Frame=-1 Dirty=0 Requested=0
                                           2: Valid=0 Frame=-1 Dirty=0 Requested=0
                                           3: Valid=0 Frame=-1 Dirty=0 Requested=0
                                           4: Valid=0 Frame=-1 Dirty=0 Requested=0

                                          Request for page 2 in W mode
                                          It's not in RAM - page fault
Process 78803 has requested page 2
Put it in free frame 0
Unblock MMU
                                          Set the dirty bit for page 2
                                           0: Valid=0 Frame=-1 Dirty=0 Requested=0
                                           1: Valid=0 Frame=-1 Dirty=0 Requested=0
                                           2: Valid=1 Frame= 0 Dirty=1 Requested=0
```

```
                                        3: Valid=0 Frame=-1 Dirty=0 Requested=0
                                        4: Valid=0 Frame=-1 Dirty=0 Requested=0

                                       Request for page 3 in R mode
                                       It's not in RAM - page fault
Process 78803 has requested page 3
Put it in free frame 1
Unblock MMU
                                        0: Valid=0 Frame=-1 Dirty=0 Requested=0
                                        1: Valid=0 Frame=-1 Dirty=0 Requested=0
                                        2: Valid=1 Frame= 0 Dirty=1 Requested=0
                                        3: Valid=1 Frame= 1 Dirty=0 Requested=0
                                        4: Valid=0 Frame=-1 Dirty=0 Requested=0

                                       Request for page 3 in W mode
                                       It's in RAM
                                       Set the dirty bit for page 3
                                        0: Valid=0 Frame=-1 Dirty=0 Requested=0
                                        1: Valid=0 Frame=-1 Dirty=0 Requested=0
                                        2: Valid=1 Frame= 0 Dirty=1 Requested=0
                                        3: Valid=1 Frame= 1 Dirty=1 Requested=0
                                        4: Valid=0 Frame=-1 Dirty=0 Requested=0

                                       Request for page 4 in R mode
                                       It's not in RAM - page fault

Process 78803 has requested page 4
Chose a victim page 2
Victim is dirty, write out
Put in victim's frame 0
Unblock MMU
                                        0: Valid=0 Frame=-1 Dirty=0 Requested=0
                                        1: Valid=0 Frame=-1 Dirty=0 Requested=0
                                        2: Valid=0 Frame=-1 Dirty=0 Requested=0
                                        3: Valid=1 Frame= 1 Dirty=1 Requested=0
                                        4: Valid=1 Frame= 0 Dirty=0 Requested=0
                                       Tell OS that I'm finished
The MMU has finished
 0: Valid=0 Frame=-1 Dirty=0 Requested=0
 1: Valid=0 Frame=-1 Dirty=0 Requested=0
 2: Valid=0 Frame=-1 Dirty=0 Requested=0
 3: Valid=1 Frame= 1 Dirty=1 Requested=0
 4: Valid=1 Frame= 0 Dirty=0 Requested=0
4 disk accesses required
```

Some Test Cases

1. 5 pages, 3 frames
   `R0 R2 R1 W3 R0 R2 R1 W4 R0 R2 R1 W3 R0 R2 R1 W4 R0 R2 R1 W3 R0 R2 R1 W4`
   My program does 20 disk accesses.
2. 4 pages, 3 frames
   `R0 R0 R1 R0 R1 R2 R0 R1 R2 R3 R0 R1 R2 R0 R1 R0 R0 R0 R1 R0 R1 R2 R0 R1 R2 R3 R0 R1 R2 R0 R1 R0`
   My program does 10 disk accesses.
3. 4 pages, 2 frames
   `R0 R1 W1 R0 R2 W2 R0 R3 W3 R0 R1 W1 R0 R2 W2 R0 R3 W3 R0 R1 W1 R0 R2 W2 R0 R3 W3`
   My program does 19 disk accesses.
4. 5 pages, 3 frames
   `R0 R1 R1 W3 R0 R2 R2 W4 R0 R2 R2 W4 R0 R2 R2 W4 R0 R1 R1 W3 R0 R1 R1 W3`
   My program does 12 disk accesses.

5. 4 pages, 3 frames
   `R0 R1 R2 R0 R1 R2 R3 R1 R2 R3 R1 R2 R3 R0 R2 R3 R0 R2 R3 R0 R1 R3 R0 R1 R3 R0 R1 R2 R0 R1 R2 R0 R1 R2`
   My program does 7 disk accesses.
6. 5 pages, 3 frames
   `W0 R1 R2 R0 R3 W1 R2 R3 R4 R1 R2 W4 R1 R2 R0 R1 R2 R0 R1 R2 W0 R1 R3 W1`
   My program does 13 disk accesses.
7. 5 pages 4 frames
   `W0 R1 R2 R0 R3 W1 R2 R3 R4 R1 R2 W4 R1 R2 R0 R1 R2 R0 R1 R2 W0 R1 R3 W1`
   My program does 12 disk accesses.
8. 5 pages 3 frames
   `R0 R1 R0 W1 R0 R1 R0 W1 R0 R2 R0 W2 R0 R2 R0 W2 R0 R3 R0 W3 R0 R3 R0 W3 R0 R4 R0 W4 R0 R4 R0 W4`
   My program does 8 disk accesses

Submission

You must submit the source code of your OS and MMU programs using the `submit2` program, by 6pm on 5th December. To submit files do:

```
~csc521/bin/submit2 csc521 VirtualMemory <your file names>
```

Your program will be assessed on:

- Successful virtual memory management - 50%
- Efficient virtual memory management, measured by the total number of disk accesses - 25%
- Programming style - 25%

It is worth 15% of the subject's assessment. Please review the policies on assessment in the administration document.

Solution

- An executable of a rather stupid OS is available in the lab as `~geoff/CSC521/VirtualMemory/OS`.
- OS.c - Source code

Students' Test Strings
