Project - Phase III: Memory Management

CSE 460: Operating Systems
Spring 2016, Zemoudeh
School of Computer Science and Engineering
California State University, San Bernardino

In this phase we will add memory management layer to our toy OS. We will implement demand-paging. Let us fix page size to be 8 words, therefore there are 32 frames in our 256 word memory. Each entry in page table consists of the frame number, the valid/invalid-bit, and the modify-bit. We will use two page replacement algorithms, FIFO and LRU. When a page-fault occurs, the offending process is placed in the wait queue with the trap completion time set to 35 clock ticks later. After a page fault is serviced, that is 35 or more clock ticks have passed, the process is moved to ready queue and the new page is loaded into memory. In addition to the information gathered in Phase II, for each process compute:

Number of Page-faults
and
Hit Ratio

Hit Ratio is the percentage of non-page fault memory references.

You will need to add a Translation Look-aside Buffer (TLB) to the Virtual Machine. Without a TLB, and assuming a PTBR, every logical memory reference results in two physical memory references and hence a slower system.

Inclusion of TLB is not only more realistic, it also simplifies the OS! If a memory reference is found in TLB it is handled in the hardware and it takes 4 time units; otherwise, it results in a trap to the OS. Every time a new process takes over the Virtual Machine, it copies its page table content onto the TLB as part of the context switch process.

Along the same line of reasoning, the OS doesn't want to check the validity of `pc` every time it is incremented. At all times, `base` and `limit` registers point to the currently executing page, and the hardware makes sure that the value of `pc` is bound by these two registers. Whenever `pc` goes out of bound and the new page is not already in TLB, only then an OS trap is generated. Otherwise, if the page is already in TLB, values of `base` and `limit` registers are updated to point to the new page and the process will continue without a page fault. Of course, all of this happens within the Virtual Machine.

To support LRU, add 32 registers to the Virtual Machine, that is one register per frame. These "frame registers" can be represented by a vector of integers. Each time a frame is accessed, the hardware saves the current time in its corresponding register. OS accesses frame registers to perform the LRU page replacement algorithm.

Note that FIFO doesn't use frame registers. The OS maintains its own vector of frame registers, and FIFO is entirely done by the OS without hardware help. When a page is brought into memory, the OS records the current time in its corresponding (software) frame register. Now the page-fault handling section of OS consults its own frame registers, and not those in the hardware, to select the victim page.

Run your OS twice, each time with a different page replacement algorithm but for the same set of programs:

```
$ os -fifo
$ os -lru
```

In this way the merits of the two algorithms can be compared. Use `seekp()` to replace a page in `.o` file which serves as the simulated disk. Note that you don't need to replace a page if its corresponding frame has not been modified.

As far as stack is concerned, it takes up as many pages as necessary in high memory. Memory locations taken up by stack will not be available for paging purposes. If the stack grows too large trying to overwrite an allocated frame, that frame has to move out (and replaced on disk if needed) before the stack can grow any larger.

Let us fix degree of multiprogramming at 5; only 5 PCBs are in either ready queue, wait queue, or CPU at the same time. When the OS starts,
it assembles 5 `.s` files to their corresponding `.o` files,
loads in the first page of each `.o` file into memory,
sets up their page tables,
sets up their PCBs in ready queue,
and starts executing the first process.

As the processes are executed, more pages are brought in based on the availability of frames and requests of the processes. Only when a process exits, a new process is added to the system to maintain a degree of multiprogramming equal to 5. Note that in this phase we need to add "Page Fault" as a new condition for context switch in addition to the conditions enumerated in Phase II. Therefore we use bit 10 of status register in conjunction with bits 5-7 to represent page fault: when bits 5-7 are all zeros, if bit 10 was 0 then there was a time-slice interrupt; otherwise, (bits 5-7 were all zeros and) bit 10 was 1 then there was a page fault.

Add four more `.s` programs: `addVector.s`, `subVector.s`, `simple1.s`, and `simple2.s` to the list of programs from Phase II for a total of 10 programs. The input files for the vector programs are `addVector.in` and `subVector.in`.

Since the `ls` command lists files in alphabetical order, the order in which the programs are brought in as a result of

```
system("ls *.s > progs");
```

is as follows:

```
addVector.s
fact1.s
fact2.s
io.s
simple1.s
simple2.s
sub.s
subVector.s
sum1.s
sum2.s
```

To incrementally develop your OS, first make sure that your program runs correctly for just simple1.s and simple2.s. That is the OS has only the above two programs to run. A quick inspection reveals that each program generates one page fault and 28 frames remain free. This implies there is no need for a page replacement algorithm in this first incremental step. After these two programs run correctly, then try running more programs at the same time.

Demonstrate your program and hand in printouts of your source code and all `.out` files for each page replacement algorithm. The same grading criteria as Phase I and II holds.