Project - Phase III: Memory Management

CSE 460: Operating Systems
Winter 2019, Zemoudeh
School of Computer Science and Engineering
California State University, San Bernardino

In this phase we will add the Memory Management layer to our OS by implementing demand-paging.

We set page size to 8 words which implies there are 32 frames in our 256 word memory. Each entry in page table consists of the frame number and the valid/invalid-bit. Every process will have an instance of page table in its PCB.

PT.h is an implementation of page table class.

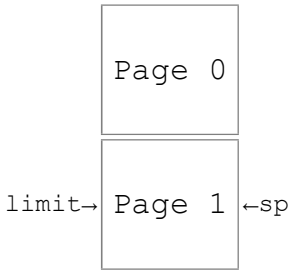
We will use two page replacement algorithms, FIFO and LRU. When a page-fault occurs, the offending process is placed in the wait queue with the trap completion time set to 27 clock ticks later--same as I/O operations. After a page fault is serviced, that is 27 or more clock ticks have passed, the process is moved to ready queue. Once a page-fault occurs, you may load the new page into memory immediately. This way when page-fault has been serviced (its time is reached or has passed) the page is already in memory.

In addition to the information gathered in Phase II, for each process and the system compute:

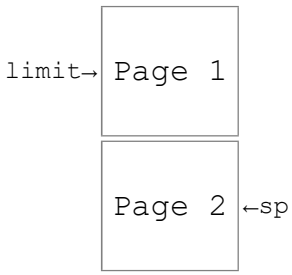
Number of Page-faults
and
Hit Ratio

Hit Ratio is the percentage of non-page-fault memory references. Clearly, Number of Page-faults for the system is the sum of page-faults for all the programs.

We remove the artificial requirement that the running program loads its stack in high memory from its `*.st` file. In this phase each program's stack grows from the bottom of the program downward. This implies initially `sp` is set to `limit`. For example, if a program is 12 words long, initially the logical address space of the program looks like:



And after the program runs for a number of clock ticks, the stack might grow into logical page 2 as follows:



Now execution of call instruction increments `sp`, and execution of return instruction decrements it. And there is no need for `*.st` file; if a stack page is swapped out, it's stored in `*.o` file. We will set the maximum size of the stack to 8 pages (64 words).

Add a Translation Look-aside Buffer (TLB) to the Virtual Machine (VM). Without a TLB, and assuming a PTBR, every logical memory reference results in two physical memory references and hence a slower system. TLB will be an instance of page table (PT.h) in Virtual Machine. As part of the context switch, every time a process wants to run on Virtual Machine, OS copies the process' page table onto TLB.

Inclusion of TLB is both more realistic and simplifies the OS! If a memory reference is found in TLB it is handled in the hardware/VM and it takes 4 time units; otherwise, it results in a page-fault (trap to the OS.)

All logical addresses in a program are translated at run time (by VM) to physical memory addresses through TLB. Refer to `log_to_phys()` in PT.h. This includes references to memory through `pc` and `sp` as well. Every time a program refers to a logical/virtual address or VM increments `pc`, VM first compares the address with `limit` to make sure an out of bound reference has not occurred, then it computes its equivalent physical address by mapping it through TLB. If the physical address is not found, because its containing page is not in memory (it's invalid), then a page-fault results and VM returns. Of course, all of this happens within the VM.

Sections of code in VM that can potentially cause page-fault are:

- load instruction (r[0] = mem[addr])
- store instruction (mem[addr] = r[0])
- instruction fetch (ir = mem[pc++])
- call instruction (mem[sp++] = r[0])
- return instruction (r[0] = mem[--sp])

It's important to distinguish between logical/virtual address and physical address. All addresses (addresses in load and store instructions, pc, and sp) are kept and used as logical addresses, only when they are supplied to memory they are converted to physical address through TLB.

Add a vector of 32 bools called "modified" to VM--one per frame. As VM runs, if a frame is modified (for example, store instruction writes on one of its words), its corresponding modified bit (bool) is set to true. These bits are used to write a victim frame on `*.o` file if necessary. 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.

To support LRU, add 32 registers to the VM--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.

FIFO doesn't need these frame registers. The OS maintains its own vector of (software) 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. The page-fault handling section of OS consults

software frame registers to select the victim frame.

OS uses an inverted page table to
1. find victim process and page number
2. maintain list of free frames
3. record time a frame was brought in for FIFO (above)

```
class IRow
{
    string process;
    int page_num;
    bool valid;
    int time_in;
    ...
};

class InvertedPT
{
    vector<IRow> frames;
    ...
};
```

Run your OS twice, once with LRU and once with FIFO page replacement algorithm for the same set of programs.

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

This way the merits of the two algorithms can be compared.

As in phase II, when the OS starts, assemble all `.s` programs to their corresponding `.o` files, sets up their PCBs in ready queue, and starts executing the first process.

As processes executes, pages are brought into memory on demand. Note that in this phase we need to add "Page Fault" as a new condition for context switch in addition to the conditions in Phase II. We use bit 10 of status register (SR) in conjunction with bits 5-7 to represent page-fault.

When bits 7:5 are all zeros and bit 10 is 0, we have a time-slice interrupt. (Phase 2)
When bits 7:5 are all zeros and bit 10 is 1, we have a page-fault interrupt.

In case of page-fault, VM uses bits 11 through 15 to let OS know which page to bring in.

Here is the new configuration of SR:

Page#	PF	I/O Reg	VM Ret Status	V	L	E	G	C
15:11	10	9:8	7:5	4	3	2	1	0

Add seven more `.s` programs: one more version of factorial program `fact3.s`, two versions of `addVector.s`, two versions of `subVector.s`, `simple1.s`, and `simple2.s` to the list of programs from Phase II for a total of 13 programs.

`simple` programs don't have input.

The input file for `fact3.s` is `fact3.in`.

The input files for the `vector` programs are `addVector1.in`, `addVector2.in`, `subVector1.in`, and

subVector2.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:

```
addVector1.s
addVector2.s
fact1.s
fact2.s
fact3.s
io.s
simple1.s
simple2.s
sub.s
subVector1.s
subVector2.s
sum1.s
sum2.s
```

Use the `touch` command to create any empty input files `*.in` that these programs might need to run.

To incrementally develop your OS, first make sure that your OS runs correctly for just `simple1.s` and `simple2.s`. A quick inspection of these two programs reveals that each one generates only two page-faults 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, implement your page replacement algorithms and run more programs to force page replacement. Here, it helps to run multiple copies of subVector.s and addVector.s in addition to other programs.

Finally, implement stack handling and add factorial programs to the mix of programs.

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.