Project - Phase II: Process 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 process management layer to our OS. We will convert the single-user OS of phase I to a time-sharing OS.

Under your home directory create `cse460` directory, under that directory create directories `phase1` and `phase2`. Copy all your current code and data to both `phase1` and `phase2`. Modify, implement, and run the new version of the OS in `phase2` directory without changing your old code in `phase2`. This way you would have an untouched version of phase I to run and test against the new version as it's being developed.

In this phase every program consists of 5 files with the same name but different suffixes: `.s`, `.o`, `.in`, `.out`, and `.st`. For example, the factorial program consists of `fact.s`, `fact.o`, `fact.in`, `fact.out`, and `fact.st`.

`*.s` and `*.in` files, which contain the assembly program and its input respectively, must exist before starting the OS.

As in phase I, the OS generates a `*.o` file for each `*.s` file through a call to the assembler.

`*.out` file is created by the OS and contains output of the program.

`*.st` file is an input/output file and contains the stack of a program.

Design a Process Control Block (PCB) to represent a process. Refer to OS.h for an example of PCB. And here is an example of os_main.cpp.

Only one stack at a time resides in memory. When a process is running on VM, its stack is read into high memory from its `*.st` file; and when the process relinquishes VM, its stack is written onto its `*.st` file. OS examines `sp` value to tell whether stack content in memory needs to be saved in `*.st` file. When a process relinquishes VM and `sp = 256`, the stack is empty and therefore there is nothing to save. Otherwise, when `sp < 256`, there is a stack and its content must be saved for a future restart. Analogously, when a process is assigned to the VM, if `sp` in its PCB is less than 256 then it has a stack and it needs to be loaded from its `*.st` file into memory.

When the OS comes up it looks in current directory and gathers all `*.s` files:

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

It then opens file `progs` and reads in file names. Each file is assembled, its object code loaded in memory, and a pointer to its PCB is stored in a linked-list:

```
list<PCB *> jobs;
PCB * p = new PCB;
jobs.push_back(p);
```

The degree of multiprogramming is the same as number of `*.s` files in the current directory (in `progs`). The processes are resident in memory until OS halts. The processes (their PCBs) are either in ready, waiting, or running state. Maintain two queue of processes, Ready Queue and Wait Queue, of type pointer to PCB:

```
queue<PCB *> readyQ, waitQ;
```

We also keep track of the running process by a pointer to its PCB:

```
PCB * running;
```

Pointers in `readyQ, waitQ`, and `running` point to a PCB in the linked-list of PCBs (`jobs`). Initially all processes are pushed on `readyQ`. To execute the very first process, the pointer to the process in front of `readyQ` is popped and assigned to `running` and the process is assigned to VM and starts running.

Usually two conditions force a running process to relinquish VM:

Either
the process completes its time slice, when it will be added to end of `readyQ`;
or
it executes an I/O operation (`read` or `write` instruction), when it will be added to end of `waitQ`.

There are also other conditions that cause VM to return. As a result, the VM returns to the OS with a return status indicating which condition occurred. The complete return-status list is:

a. time slice
b. `halt` instruction
c. out-of-bound reference
d. stack overflow
e. stack underflow
f. invalid opcode
g. I/O operation

VM sets the status register based on the above conditions and OS examines it to know how the previous process relinquished VM. As far as overflow bit (V in `sr`) is concerned, it's the assembly programmer's resposibility to check for V in their program and take appropriate action. In other words, although VM sets the value of V, it does not return to OS if there was an overflow.

In this phase, `sr` format is extended to include VM Return-status encoded in 3 bits:

d	...	d	I/O Register	VM Return-status	V	L	E	G	C
15		10	9:8	7:5	4	3	2	1	0

Meaning of VM return status in `sr` (contained in bits 7:5) is summarized in the following table:

VM Return-status	Meaning
000	Time slice
001	Halt Instruction
010	Out-of-bound Reference
011	Stack Overflow
100	Stack Underflow
101	Invalid Opcode
110	Read Operation
111	Write Operation

In case of Read/Write (I/O) operations, the destination register is specified in bits 9:8. The OS needs to know which register was the target of the I/O operation. For example, if the instruction was

```
read 3
```

the VM passes 3 = 11_2 in bits 9:8 (of `sr`) to the OS. The OS performs the I/O operation (possibly through DMA) and sets content of register 3 (from the `.in` file) into the PCB. When the process is ready to resume, content of register 3 is ready and will be transferred to the VM.

Any time the VM returns (one of the above eight conditions has occurred) a context switch happens and the scheduler reorganizes the queues. Context switch takes 5 clock ticks (all CPU time). During this time
first, all processes in `waitQ` whose I/O operation has been completed are placed in `readyQ`,
second, the running process is placed in the proper queue or terminated, and
third, the next process from `readyQ` is assigned to VM (CPU).

I/O requests could immediately occur in the PCB: when an I/O operations is encountered, immediately perform the I/O (`read` or `write` instruction) in PCB, move the PCB to `waitQ`, and set the interrupt (I/O completion) time to `clock + 27` (1 tick already taken by VM to decode the instruction). During the next context switch, if the I/O completion time of a process in `waitQ` is less than or equal to the current time (the I/O interrupt has arrived), its PCB is moved to `readyQ`.

If all processes are waiting on I/O (`readyQ` is empty), you must add as many clock ticks to the clock to match the completion time of the earliest I/O request, at which point that process will be ready for execution and is moved to `readyQ` and then to running state. This is counted as idle time and decreases CPU utilization, see below.

If time slice of a process is over in the middle of `load`, `store`, `call`, and `return` instructions, finish the instruction first and then perform context switch. Any time this occurs, the time slice of the process is effectively extended by at most 3 clock ticks.

All memory references made by a process have to be checked against its `base` and `limit` values. If an out-of-bound reference is made, the program is terminated and an appropriate message must appear in the `.out` file. Note all addresses are offset from `base`; at run time add `base` value to the addresses in `load`, `store`, `call`, and `jump` instructions.

Each PCB should at least include `pc`, `r[0]-r[3]`, `sr`, `sp`, `base`, `limit`, process name, `fstream`s associated with `*.o`, `*.in`, `*.out`, and `*.st` files, and the following accounting information: VM (CPU) Time, Waiting Time, Turnaround Time, and I/O Time. The accounting information for each process must appear at end of the `*.out` file. Also VM Utilization and Throughput must appear at end of EACH `*.out` file after the process specific accounting information.

The definitions of the accounting information as they pertain to this phase are:

Process Specific:
CPU Time: number of clock ticks a process executes in CPU. (`read` and `write` each take 1 CPU clock tick and 27 I/O clock ticks.)
Waiting Time: number of clock ticks spent in `readyQ`.
Turnaround Time: time up to and including the `halt` instruction execution.
I/O Time: number of clock ticks spent in `waitQ`.

System Information:
System Time = sum of all Context Switch Times and Idle Times
System CPU Utilization: percent of time CPU is busy = (final clock - sum of all Idle Times) / final clock
User CPU Utilization: percent of the time CPU executes user jobs = (sum of all jobs' CPU time) / final clock

Throughput: number of processes completed per second. Assume 1 second = 1000 clock ticks.

The following table summarizes all times.

load/store instr	**call/return instr**	**read/write instr**	**all other instr**	**time slice**	**context switch**	**1 second**
4 clock ticks	4 clock ticks	28 clock ticks	1 clock tick	15 clock ticks	5 clock ticks	1000 ticks

Make `OS` class a `friend` of `VirtualMachine` class so that for each process the state of the VM can be loaded from or stored to its PCB by the OS.

Run your OS for 6 programs as follows:
(From phase I) `fact1.s` with input 6 (`fact1.in` contains 6)
(From phase I) `fact2.s` with input 8 (`fact2.in` contains 8)
(From phase I) `sub.s` (subtract 2 program)
`sum1.s` with input 50 (`sum1.in` contains 50)
`sum2.s` with input 101 (`sum2.in` contains 101)
`io.s` where `io.in` contains 0 1 2 3 4 5 6 7 8 9 10 11

The `sum` program is as follows:

```
    loadi  0 1   ! i = 1
    loadi  1 0   ! sum = 0
    read   2
    compr  0 2
    jumpe  8     ! done
    add    1 0   ! sum += i
    addi   0 1   ! i++
    jump   3     ! loop again
    write  1
    halt
```

The `io.s` program is as follows:

```
    loadi  0 0   ! i = 0
    compri 0 6   ! 6 pairs to read
    jumpe  9     ! i == 6 done
    read   1
    read   2
    add    1 2
    write  1
    addi   0 1   ! i++
    jump   1     ! loop again
    halt
```

Implement your program incrementally.
First, modify your OS to run only two programs without any I/O (just compute intensive `.s` programs).
Second, modify your OS to handle programs with I/O.
Third, try several compute and I/O intensive programs.
Fourth, modify your OS to gather accounting information.
Fifth, modify your OS to handle programs with subroutine calls (which grow stack).

Demonstrate your program and hand in printouts of your source code including OS, new VM and assembler, and all `*.s`, `*.o`, `*.in`, and `*.out` files.

Same grading criteria as phase I holds.