CSci 460: Operating Systems **Spring 2001, Zemoudeh**
Handout 3: Phase II – Process Management **Due Wednesday May 17**

California State University, San Bernardino
Computer Science Department

In this phase we will add the process management layer converting the single-user OS of phase I to a batch-time-sharing OS. Under your home directory create a `460` directory, under that create a `phase2` directory, copy everything from `phase1` to this directory. Modify, implement, and run the new version of the OS in this directory.

Every program consists of 5 files with the same name but 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.` The `.s` and `.in` files must exist before starting the OS. They hold the assembly program and its input respectively. The `.o` file has to be generated by the OS for each `.s` file through a call to the assembler. The `.out` file is created by the OS and contains the output of the program. The `.st` file is an input/output file and contains the stack when the process is not running; only one stack (of the running program) at a time resides in memory. When VM is allocated to a process, its stack is read into high memory from the `.st` file; and when a process relinquishes VM, its stack is written onto the `.st` file. It is easy to decide if the `.st` file has to be read/written by examining the value of the `sp`. Remove this file when its corresponding process halts.

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

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

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

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

In this phase of the project the degree of multiprogramming is the same as the number of `.s` files in the current directory (in `progs`). The processes are resident in memory until the OS halts. The processes (PCBs) are either in the ready, waiting, or running state. Maintain two queues of processes: `readyQ` and `waitQ`. The queues are of type pointer to PCB:

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

So is the running process:

```
PCB* running;
```

Each pointer in the queues and in `running` point to a PCB in the linked-list of PCBs established earlier. Initially all processes are pushed on the `readyQ`. The processes are run from the `readyQ` using a Round-Robin Scheduling algorithm with a time slice of 40 time units. After a process is assigned to the VM (it starts running), it either completes its time slice, when it will be added to the end of `readyQ`; or it executes an I/O operation (`read` or `write` instruction), when it will be added to the end of `waitQ`.

Every time a time slice is over or there is an I/O request, a context switch takes place and the scheduler reorganizes the queues. Context switching takes 10 time units (all CPU time). During this time **first,** all the processes in `waitQ` whose I/O operation has been completed are placed in `readyQ`, **second** the running process is placed in the proper queue, and **third** the next process from `readyQ` is run (assigned to VM).

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 the PCB, move the PCB to `waitQ`, and set the interrupt (I/O completion) time to `clock + 64`. During the next context switch, if 64 or more time units has passed (the I/O interrupt has arrived), the PCB is moved to the `readyQ`. If all processes are waiting on I/O (both `readyQ` is empty and VM is idle), you must add as many time units to the clock to match the time of completion of the earliest I/O request, at which point that process will be ready for execution. This decreases CPU utilization, see below. If the time slice of a process is over in the middle of `load` or `store` instructions, finish the instruction first and then perform the context switch. Any time this occurs, it effectively extends the time slice of a process by at most 3 time units.

All memory references have to be checked against the `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 has to be offset from `base`; at run time add the `base` to the addresses for `load`, `store`, `call`, or the `jump` instructions.

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

`read` and `write` each take 1 CPU time unit and 63 I/O time units.

Waiting Time: number of time units spent in `readyQ`.

Turnaround Time: time up to and including the `halt` instruction execution.

I/O Time: number of time units spent in `waitQ`.

Response Time: time up to and including the first `write` instruction execution.

Largest Stack Size: largest number of memory locations allocated to the process stack.

System Information:

CPU Utilization: percent of the time CPU is busy.

Throughput: number of processes completed per second,

assume 1 second = 1000 time units.

Your `OS` class could be a `friend` of the `VirtualMachine` class so that for each process the state of the VM can be loaded from or stored to its PCB by the OS. The VM returns either after 40 time units (time slice is over), an I/O instruction is encountered, a `halt` instruction is encountered, or an out-of-bound reference is made.

Run your OS for 6 programs as follows:

(From phase I) `fact1.s` with `fact1.in` content being 6

(From phase I) `fact2.s` with `fact2.in` content being 3

(From phase I) `sub.s` the subtract 2 program

`sum1.s` with `sum1.in` content being 50

`sum2.s` with `sum2.in` content being 101

`io.s` with `io.in` content being 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 the OS to run (only 2) programs without any I/O (just compute intensive `.s` programs). Second, modify the OS to handle programs with I/O. Third, try several compute and I/O intensive programs. Fourth, modify the OS to gather accounting information. Fifth, modify the OS to handle programs with subroutine calls (grow stack).

Demonstrate your program and hand in printouts of your source code including the OS, the new VM, and the assembler, and all `.s`, `.o`, `.in`, and `.out` files. The same grading criteria as phase I holds.