CSci 5103: Operating Systems

# Project 1: Process Implementation and Scheduling

Due: 10:00 PM February 21, 2012

## 1. Objective

In this project you will be exposed to three aspects of Operating Systems. The first relates to the implementation issues that surround processes and the asynchrony that must be managed to execute and schedule them. The second is implementing and evaluating new research ideas from systems papers in the context of Operating Systems. For this task, you will experiment with scheduling policies including Lottery Scheduling. The third is the provision of useful tools needed by end-users. For this you will implement a simple version of top. We will define a minimum feature level and fidelity for your "OS" and you are welcome (and encouraged) to do more if you are willing. This project should be done in groups of 2 or 3.

## 2. Design Overview

You will simulate four process management functions: process creation, process state transition, process scheduling, and context switching. You will also simulate OS interactions with the CPU hardware and external devices. Your CPU model will be very simple - it will support the execution of a reduced instruction set including system calls and privileged instructions, two registers, PC, VC, and a mode bit. You will also provide three I/O device types that generate interrupts - a system clock timer, a character device, and a block device. For devices, the only aspect you will model is the rate at which interrupts and signals are generated by each device class.

You must use C/C++ and the code must run in a Unix environment. You will implement your OS as a Unix process that contains basic code for process management, process switching, process scheduling, and interaction with simulated hardware. We strongly recommend the use of a modular structure with classes (if you use C++) for devices, CPU, scheduler, and so on. The modularity of your code will be a factor in the grading. For example, the use of queues, PCBs, the ability to add new devices, new scheduling algorithms, etc will be assessed.

## 3. Project Details

When your OS boots up, it establishes a timer and two I/O devices. The timer sends interrupts every T real-time time-units, and the two devices are programmed to send UNIX signals (CHAR_SIGNAL, BLOCK_SIGNAL, SIGUSR1, or SIGUSR2) at the "completion" of their I/Os. The amount of time each device I/O takes is a constant, CHAR_DELAY, and BLOCK_DELAY. These will be a multiple of your basic clock timer interval. An I/O is scheduled for completion {CHAR|BLOCK}_DELAY time-units in the future. Your OS also reads a trace file when it boots up that contains an initial process to create and run. Your OS then goes into a main loop looking for simulated processes to run. It waits for a timer interrupt to indicate the passage of time. You may want to block all others signals until the timer interrupt is received. Thus, each simulated CPU instruction takes 1 time-unit. When there is no more work to be done (all processes are finished and no I/Os are pending), your OS may exit. Each simulated process contains a simple VAS with just a text segment and associated some kernel state (PCB). Each simulated process is comprised of a program that manipulates (sets/updates) the value of a single integer variable/register. Thus the state of a simulated process at any instant is comprised of the value of its integer variable/register and the value of its program counter. A simulated process' program consists of a sequence of instructions. There are seven types of instructions:

S x : **store x to VC**
A x : **add x to VC**
D x : **decrement x from VC**

C <priority> <file-name> : **create a process with the given priority using file-name (a code file)**
E : **terminate the current process**
I <dev-class> (B/C) : **IO syscall req to given device class**
P : **privileged instruction (exception if mode not supervisor)**

An example of a program is as follows (the P instruction would cause an exception):

```
S 1000
A 19
A 20
D 53
A 55
C 5 proc1.trace
C 10 proc2.trace
P
E
```

The code for a program such as this would be kept in file (see main.trace for an example). The program counter starts at 0 for a VAS text segment (e.g. the S 1000 instruction is at 0). Your OS keeps track of process state transitions, queues, and status (ready, blocked, terminated, and running), the progress of time, and schedules processes to be run using different scheduling algorithms. A process is blocked awaiting completion of an I/O that it has initiated. When a process is picked to run, the OS will switch out the current process, switch in the new one, and the CPU will execute its instructions. A process has a priority that may be used for various scheduling algorithms. You must implement basic context-switching logic and store sufficient information into the kernel PCB for a process. Think about what information this will need to contain. Among other items, your bookkeeping data-structures should contain process id (you will generate this), parent id, start CPU time, amount of accumulated CPU time, memory size (just text size), priority, program counter, etc. CPU time is in units of timer interrupts. So when the first interrupt is received, the CPU time is 1, the next interrupt, the CPU time is 2, and so on. You are welcome to store the integer process value (this is the value in the VC register) in kernel memory or allocate a data area as part of your VAS (the latter gives a bit more reality). You will also need a device table/data-structure to determine the status of devices and which processes are waiting on them. Your OS should create an initial process (contained in a file called main.trace). The initial process has a parent process id = 0.

During CPU execution, control returns to the "kernel" for system-calls (C, I, E), interrupts, and for exceptional conditions (P). In a real system, the OS kernel instructions would then be executed by the CPU, including any bookkeeping code. To give a flavor of this, one OS system call (I) call begins with the simulated CPU execution of a fixed set of privileged instructions (a constant number of P instructions - these represent privileged I/O device register instructions); in our case, they are no-ops, but are allowed by the CPU (unlike in user mode). Following this set of P instructions, the actual kernel code (i.e. in C/C++) is executed. The I instruction schedules an I/O completion event in the future (that will arrive via a signal); until this time, the executing (simulated) process is blocked. When the I/O completes (via the signal), the process can be unblocked.

Your OS should keep track of the passage of time (via timer interrupts), do CPU accounting, and produce a trace file of CPU activity (see trace.log). When a process terminates, all kernel state and memory (i.e. the VAS) for it is removed from the system.

B) Scheduling:

You will implement two scheduling policies as part of this assignment: round-robin with a fixed quantum size (ignore the priority in this case) and lottery scheduling. For the latter, implement basic lotteries - assign a number of tickets equal to the priority of the process. Are you able to observe the desired effects in lottery scheduling?

Finished early?
You may experiment with a scheduling policy of multiple queues with priority classes. In this policy, the first simulated process starts with priority 0 (treat this as the highest priority). There are a maximum of four priority classes. Time slice (quantum size) for priority class 0 is 1 unit of time; time slice for priority class 1 is 2 units of time; time slice for priority class 2 is 4 units of time; and time slice for priority class 3 is 8 units of time.

If a running process uses its time slice completely, it is preempted and its priority is lowered. If a running process blocks before its allocated quantum expires, its priority is raised. Are you able to observe the desired effects of this type of scheduling?

C) Top:

You will implement a simple version of the Unix top program which can be run in another shell concurrent with your OS. Top probes the OS tables to print out process status including: pid, memory used, CPU used, status, etc. You may run our top to get an idea of what it can do, but feel free to add more bells-and-whistles if you like. Your implementation should fetch OS process information *periodically* (the interval is up to you). Since top is a separate Unix process how does it interact with your OS? This is also up to you - some possibilities include pipes and shared memory.

# 4. Important Notes on Implementation

You will need to be clear on the difference between the "Unix OS process" and "top process" (these are what you are coding for the assignment) and the simulated processes within your OS. We have deliberately left things a bit open ended to allow you some implementation flexibility.

Some of the system calls which may be of use in your implementation are listed below. However, feel free to explore the use of other system calls if you feel they're better suited to the task at hand. Most often the best place to look for details on system calls (such as their signatures, which headers to include, compile time flags, platform specific behavior etc.) are their man pages.

**Interrupts and Signals:**
It is recommended that you use POSIX timers since they allow you to set multiple timers per signal at a fine time resolution.

timer_create - To create a timer and specify which signal is generated.
timer_settime - To set when it goes off.

See:
http://www.kernel.org/doc/man-pages/online/pages/man2/timer_create.2.html and
http://www.kernel.org/doc/man-pages/online/pages/man2/timer_settime.2.html for more details.

**For signal handling you could use the calls:**
sigaction - To specify a signal handler
sigsuspend - To selectively wait for a signal

See:
http://www.linuxprogrammingblog.com/all-about-linux-signals?page=show and
http://www.enderunix.org/docs/signals.pdf for more details.

**Inter Process Communication:**
You could handle IPC using shared memory or pipes (or other techniques).

**Shared memory:**
shmget - Allocate a shared memory segment
shmat - To attach a shared memory segment
shmdt - To detach a shared memory segment

shmctl - Various forms of control (e.g. destroying shared segments)

**Pipes:**
pipe - Used to create a pipe. You can use the read and write system calls after this on the two file descriptors.

A nice article on Unix IPC can be found at:http://www.ecst.csuchico.edu/~beej/guide/ipc/

**Output File Format:**

To aid us in evaluating your projects, we require you to print the following items to a trace log file:

1) The clock tick number every time a clock tick interrupt is received.
2) When instructions are executed, you should print the instruction and the originating process' ID.
3) When a {CHAR|BLOCK} device interrupt is received, print out which process has been unblocked (moved to the ready queue).

.
.
.
Clocktick: 48
Executing: A 10 for pid= 1

Clocktick: 49
Executing: A 80 for pid= 2

Clocktick: 50
Unblocking process: 3
Executing: D 20 for pid= 1

Clocktick: 51
Executing: A 10 for pid= 3
.
.
.

4) For Lottery scheduling, you also need to print the ticket allocations to various processes.
5) Whenever a new process is being scheduled you also need to print the lottery ticket number used for this decision.

A sample of your log file might look like:
.
.
Clocktick 52
Lottery allocations: pid=1 tickets=10, pid=2 tickets=30,pid=3 tickets=10
Scheduler: lottery ticket= 45 chose process with pid=3
.
.

# 5. Submission and Grading

**Testing:**

- There is no restriction on machines and OS for developing your code, but it must compile and run

correctly on the virtual machine linked on the course's moodle site. Vmware Player and VirtualBox are free cross platform applications for running and interacting with virtual machines. Both are well documented online and readily downloaded. If any student has questions or concerns about using the virtual machine should contact the TA during office hours.

- We will provide trace files for execution.

**Submission:**

- Zip/Tar your code and submit it via moodle.
- Submission must include a makefile which builds your solution via “make all” and executes your solution via “make run”.
- Submission must include a readme file which has the following information:
  - Names of all group members
  - x500 login names of all group members
  - Student IDs of all group members
- No need to submit a paper copy. Save trees instead.

**Grading:**

- Code design and documentation: 10%
- Asynchrony - interrupts/signals: 20%
- Context switching: 10%
- Device Handling: 10%
- Kernel operations (bookkeeping, exceptions, syscalls, etc): 20%
- Scheduling: 20%
- Top: 10%
- Extra credit: 5% (multi-level priority queues)