**CS 354 Spring 2018**

**Lab 3: Monitoring and Scheduling of XINU Processes (280 pts)**

**Due: 02/27/18 (Tue.), 11:59 PM**

**1. Objectives**

The objective of this lab is to monitor the CPU usage of XINU processes and affect "fair" sharing of CPU cycles for CPU- and I/O-bound processes.

**2. Readings**

Read Chapter 5 of the XINU textbook.

**3. Monitoring CPU usage of processes [100 pts]**

A basic kernel facility is to monitor CPU usage of processes. The resultant measurements may be used for scheduling, accounting, and other purposes. One approach to monitoring is to use the kernel's event tracking timer, called system timer, to measure the cumulative CPU time that has been allocated to a process since its creation. We will discuss clock management, including use of special clock hardware in x86, later in the course. For present purposes, it suffices to note that one of the hardware clocks of our backend machines is programmed in XINU to interrupt every 1 msec which invokes clkhandler() (see clkdisp.S and clkhandler.c in system/). This fixed and repeating (i.e., periodic) time interval is called a tick.

(a) XINU uses a global variable, clktime of type uint32, to keep track of the total number of seconds elapsed since bootstrapping a backend. Define another global variable, clkmilli of the same type, to keep track of the total time elapsed at the granularity of millisecond. clkmilli faces wrap-around when the counter eventually reaches its maximum and resets to counting from 0. It does so a thousand times sooner than clktime. However, unless you hog a XINU backend -- which you should not -- the wrap-around problem will not matter for this lab and we will ignore it. Define clkmilli in the same header file where clktime is defined.

(b) Define a new field as a process table entry, called *prcputot*, of type uint32, that is used by XINU to keep track of the CPU time used by a process. This field is initialized to 1 when a process is created. To keep track of CPU usage, when allocating CPU to a process by context switching it in, the current value of clkmilli is remembered, and when it is context switched out in the future, this value is subtracted from clkmilli to yield the CPU time used by the process. Add this value to prcputot to update the process's CPU usage. Use a field, prctxswbeg, to remember the time when a process is context switched in.

Modify the internal kernel functions clkhandler() and resched() of our legacy XINU code so that the aforementioned CPU usage accounting operations are implemented. Make sure to adequately comment your code so that it shows where changes in source code have been introduced by whom (your name), when, and for what purpose. As mentioned in the TA Notes, maintain backup copies of intermediate versions of kernel functions that you are modifying such as by using recommended version control tools so that you can readily revert to an earlier versions when necessary.

(c) Run tests by spawning several XINU processes and checking their monitored CPU usage. When designing test cases, consider what outcomes you expect in the measurements based on the priorities of processes when they were created. For example, if four processes of equal priority are run where each process is a for-loop with an upper bound UPBOUND (use #define to set its value in a header file test.h in include/), they should receive approximately equal share (a little less than 25%) of CPU time. Make this your baseline test case. Set UPBOUND to be sufficiently large so that transient effects are averaged out.

(d) Each test process (created by the process running main()) prints prcputot just before terminating. To do so, create a new system call, syscall getcputot(pid32 pid), stored in file getcpuused.c in system/, that takes the process ID of a process as argument (in our case we will call getcputot() with the global variable currpid which contains the PID of the current process) and returns a 32-bit integer (check how the type syscall is defined) that specifies total CPU usage of the process. The code of getcputot() looks up the prcputot field in the process table entry of the process. If PID is that of the current process, getcputot() returns the value of prcputot adjusted by time slice consumed by the current process. When implementing a system call, it is critical to perform sufficient sanity checks such as determining if the PID passed in the argument is valid. Look at existing system calls in system/ that take PID as argument and follow the sanity checks that they perform. Also, keep in mind that in XINU system calls first disable interrupts before doing anything else, and they reenable interrupts before returning. Make sure to abide by this convention.

(e) For debugging purposes, you will need to print system variables during the lifetime of a process. Use #ifdef DEBUG conditional compilation to turn off debugging print messages when submitting your work. Do not clean up your code by removing debug kprintf() calls that output clkmilli, prctxswbeg, and prcputot, among other values. They need to be present in your code so that the TAs can enable them when evaluating your code. They will also convey how you went about determining whether implementation works correctly or not.

(f) A process that sleeps intermittently will receive less compared to ones that never block. If processes of different priority levels are running concurrently, a process of lower priority may potentially never receive CPU cycles, i.e., starve. Create additional test cases to evaluate CPU sharing and usage accounting under nonuniform process priorities. Discuss your findings in Lab3Answers.pdf.

**4. Dynamic priority scheduling [180 pts]**

**4.1 General background**

The default XINU scheduler is a fixed priority scheduler that uses priorities specified in the create() system call and never changes them. XINU's process scheduler, resched(), when invoked, picks a highest priority ready process to run next. If there is a tie (i.e., multiple processes at the same priority level) round-robin scheduling is implemented.

As discussed in class, a dominant paradigm of "fair" scheduling in operating systems such as UNIX and Windows (and until recently Linux) has been classifying processes into CPU- and I/O-bound processes based on their run-time behavior, which is then used to adjust their priorities and time slices dynamically. For example, a CPU-bound process tends to hog the CPU and not relinquish it voluntarily through blocking system calls. This requires preemption of the process by the scheduler so that the shared CPU is not monopolized. Preemption is effected by the clock interrupt handler (see clkdisp.S and clkhandler.c) which keeps track of the time slice consumed by a process -- part of the clock interrupt handler's bookkeeping activity -- and calls the scheduler (see resched.c) when the time budget maintained in global variable *preempt* is depleted (i.e., reaches zero).

In contrast, an I/O-bound process tends to make system calls to invoke kernel I/O services. In the case of a read() system call that reads from an I/O device, if data at an I/O device buffer (e.g., web client waiting on server response packets on an Ethernet card) is not available, the default agreement between a process making the read() system call and the kernel is that the process in question is put in a blocking state and context-switched out so that a highest priority ready process may utilize the CPU. When an event (e.g., packet arrival) that the process is blocking on eventually occurs, the kernel will unblock the process and put it in the ready state so that the scheduler, when invoked in the future, will include the unblocked process in its scheduling decision.

Since I/O-bound processes often relinquish the CPU voluntarily through blocking system calls, whereas CPU-bound processes hog the CPU and must be "forcefully" preempted, it makes sense to assign I/O-bound processes higher priorities relative to CPU-bound processes. That is, when a blocked I/O-bound process becomes unblocked (such unblocking events are processed by the kernel's interrupt service routines), it is desirable that the kernel's scheduler is invoked which then preempts a CPU-bound process that is currently occupying the CPU and context-switches in the unblocked I/O-bound process. This is conducive to promoting "fair" sharing of CPU cycles between CPU- and I/O-bound processes. Since I/O-bound processes, by their nature, consume less CPU cycles than CPU-bound processes, they are rewarded by increased responsiveness compared to CPU-bound processes when they become ready to run.

**4.2 Dynamic priority scheduling to achieve "fairness"**

Modify XINU so that it implements dynamic priority scheduling where process priorities are continually changed to achieve equitable sharing of CPU cycles by I/O- and CPU-bound processes. At any given time, the priority of a process is defined as 1 / prcputot (a real number) where prcputot is the total (i.e., cumulative) CPU usage of a process since creation as implemented in Problem 3. Hence the smaller the amount of CPU time a process has received, the higher its priority relative to processes that have received a larger share. Instead of maintaining 1 / prcputot as a floating point number, we will use MAXPRIO - prcputot as the priority of a process where MAXPRIO is the largest priority value possible of XINU processes which is determined by type pri16. Define the constant MAXPRIO in include/process.h. This simplification is not necessary but avoids using floating point numbers and related complications not essential for this lab. During testing you need to ensure that processes do not receive more than MAXPRIO milliseconds of CPU time which is sufficient for our purposes.

With MAXPRIO - prcputot defined as the priority of a process, we can keep using XINU's ready list which is a priority queue sorted in decreasing (more precisely, nonincreasing) order of process priority from which the scheduler, resched(), picks the first element. That is, a process with the smallest total CPU time received is at the front of the list. Since prcputot changes (i.e., increases) as a process receives CPU cycles, the priority of a process prprio stored in the process table also changes. Thus, unlike in legacy XINU where the priority of a process specified in create() is never altered, in our modified XINU priorities evolve and the scheduler performs dynamic priority scheduling as modern kernels such as Linux/UNIX and Windows do. Note, the ready list only holds ready processes, i.e., processes that are ready to use the CPU. Since we won't be discussing I/O until later in the semester, we will consider I/O-bound processes as those who call sleepms() and thereby voluntarily relinquish the CPU. What matters is whether processes voluntarily block or not. A process that has woken up from sleepms() moves from a sleep queue to the ready list. This is taken care of by legacy XINU. Lastly, set the global variable preempt that represents time slice to 20 msec through the constant QUANTUM so that the scheduler gets a chance to preempt CPU-bound processes every 20 msec.

As in Problem 3, document your code changes and preserve conditional debugging printouts which will help gauge correctness of your implementation. Maintain backup copies of working versions of code using version control tools for your own benefit. It is easy to make mistakes and it is best to avoid starting from scratch.

**4.3 Performance evaluation**

To evaluate how well the dynamic fair scheduling implementation in XINU balances fairness and performance of CPU- and I/O-bound processes, consider the following three test case scenarios.

1. *All processes are CPU-bound*. In the first test case, create 5 processes that run the same program *cpubound()*. Put the code of *cpubound()* in a separate file *cpubound.c*. Assign the same initial priority value 50 when calling *create()* from *main()*, and set *QUANTUM* in *kernel.h* to 20 msec. The code structure of *cpubound()* follows:
2. for (i=0; i<UP1; i++) {
3. for (j=0; j<UP2; j++) {
4. // Insert code that performs memory copy (slow) and/or
5. // ALU operations (fast).
6. // Note: this loop consumes significant CPU cycles.
7. // Using kprintf, print the pid, outer loop count i, and prcputot
8. }
9. }
10. // Print the pid followed by prcputot, prprio, and preempt before terminating.

XINU's *null process* should only run if there are no ready processes in the system. Discuss your findings in Lab3Answers.pdf and give your evaluation if fair scheduling of CPU-bound processes is being achieved by your dynamic priority scheduler. You will need to experiment with different values of *UP1* and *UP2* to induce CPU sharing whose output can be used for gauging fair scheduling performance.

1. *All processes are I/O-bound*. Follow the same steps as above but replace *cpubound()* by *iobound()* (in *iobound.c*) which has the same code structure as *cpubound()* except that the code in the inner loop (for ALU and memory copy) is replaced by a single call to *sleepms()*. By varying the argument of *sleepms()* you can modulate the degree to which *iobound()* is prone to block and voluntarily relinquish the CPU. When testing, use the same sleep time as argument to sleepms() for all 5 instances of I/O-bound processes. In Lab3Answers.pdf, include a discussion of the results observed and your assessment of fairness by the dynamic priority scheduler.
2. *Half-and-half*. Create 6 processes where half execute *cpubound()* and the other half execute *iobound()*. In the first part of the evaluation, under this mixed workload of CPU- and I/O-bound processes, determine if the three CPU-bound processes -- among themselves -- achieve equal sharing of CPU cycles as indicated by the output. Do the same for the three I/O-bound processes with their sleep time arguments to sleepms() fixed to the same value. Evaluate CPU sharing between the two groups of processes -- CPU- and I/O-bound -- and discuss your findings in Lab3Answers.pdf.

When generating the above workload, we want to let the main() process create and resume all test processes but not get in the way (i.e., exert significant influence on scheduling and CPU cycles received) subsequently. It does so by terminating after creating and resuming the last test process.

We also need to ensure that the idle process running nulluser(), which is always ready, has the least priority so that it only runs when there are no other ready processes in the system. Since the idle process is initialized with priority 0 by sysinit(), XINU's scheduler needs to handle the idle process as a special case so that its priority always remains 0. We achieve this by keeping track of the CPU time received by the idle process (which is useful information since it conveys to what extent the CPU is being utilized) but not updating its prprio with MAXPRIO - prcputot. Perform update of prprio inside kernel functions where appropriate. Using comments, clearly indicate where in XINU kernel code updates are being done. Note that in our modified XINU implementing dynamic priority scheduling, the initial process priority given as argument to create() is effectively being ignored.

**Bonus problem [15 pts]**

Modern operating systems such as Linux/UNIX and Windows support real-time (RT) processes whose priority exceeds that of all time-share (TS) processes. Consequently if a RT process never voluntarily relinquishes the CPU, TS processes never get to run. Due to this, only a superuser or root is allowed to create RT processes. Describe a modification of XINU that supports RT processes which is compatible with the fair scheduler implemented in Problem 4. That is, they may co-exist. The modification encompasses how RT processes are created and how the XINU scheduler manages to give higher priority to RT processes over TS processes whose priorities dynamically change. You don't need to implement your design but it should be detailed enough so that someone familiar with XINU can understand and code it in a straightforward manner. Describe your design in Lab3Answers.pdf.

**Turn-in Instructions**

Before submitting your work, make sure to double-check the [TA Notes](http://www.cs.purdue.edu/homes/ogg/cs354_spring2018.html) to ensure that additional requirements and instructions have been followed.

*Electronic turn-in instructions:*

        i) Go to the xinu-spring2018/compile directory and do "make clean".

        ii) Go to the directory of which your xinu-spring2018 directory is a subdirectory. (NOTE: please do not rename xinu-spring2018, or any of its subdirectories.)

                e.g., if /homes/joe/xinu-spring2018 is your directory structure, go to /homes/joe

        iii) Type the following command

                turnin -c cs354 -p lab3 xinu-spring2018

You can check/list the submitted files using   
  
turnin -c cs354 -p lab3 -v

***Important: Please provide comments inside your code so that its function and flow can be conveyed to the reader. Turn off all debugging output before you submit your code.***