#### CS 354 Spring 2018

#### Lab 4: Fast TS Process Scheduling (250 pts)

**Due: 03/20/2018 (Tue), 11:59PM**

## 1. Objectives

In this lab, you will implement UNIX Solaris' TS scheduler in XINU. By using a multilevel feedback control queue, you will achieve constant overhead process scheduling independent of the number of processes in the system.

## 2. Readings

Read Chapters 6 and 7 of the XINU textbook.

## 3. Implementation of UNIX Solaris TS scheduling

### 3.1 Process classification for priority increase/decrease

Classification of processes based on their observed run-time behavior must be done by the kernel efficiently so that the scheduler's footprint is kept small. As discussed in class, a simple strategy used by UNIX Solaris is to classify a process based on its most recent scheduling related activity: (i) if a process consumed all of its time slice and needed to be preempted by the clock interrupt handler then the process is viewed as CPU-intensive; or (ii) a process voluntarily relinquishes the CPU by making a blocking system call (e.g., sleepms()) in which case the process is viewed as I/O-intensive. In both instances, the scheduler will change the process' priority value. In case (i), the process is demoted by reducing its priority; in (ii), the process's priority is bumped up.

After this change of priority, the scheduler picks a highest priority process from ready processes to run, round-robin if there are two or more highest priority processes. As to the time slice, in the case of (i), the quantum is increased to reflect the CPU-intensive behavior of the process. In case (ii), the opposite occurs. The exact quantitative change in priority and quantum values is determined by a system configuration file which contains calibrations to suit a particular operating environment. The generic, default configuration file is given by:

* Solaris UNIX dispatch table: [TS](http://www.cs.purdue.edu/homes/cs354/disp-table-solaris.txt)

Ignore columns 4 and 5 which are failsafe mechanisms to prevent starvation if TS scheduling goes awry. Use a data structure xts\_tab defined in kernel header file xts\_init.h that you will create and place in include/

struct xts\_tab {

int xts\_quantum; // new time slice

int xts\_tqexp; // new priority: CPU-intensive (time quantum expired)

int xts\_slpret; // new priority: I/O-intensive (sleep return)

};

extern struct xts\_tab xts\_conf[];

which is read to initialize a kernel data structure, struct xts\_tab xts\_conf[60], in sysinit() in initialize.c. Since XINU does not have a file system, the content of the Solaris dispatch must be hardcoded when xts\_tab xts\_conf[] is initialized. Replace the constant "60" using #define in xts\_init.h. Choose your own name for the constant definition. Note that in Solaris, the TS scheduling table is exported as a configurable kernel data structure (requires superuser status). Include xts\_init.h in xinu.h.

### 3.2 XINU kernel modification

Use the XINU kernel from lab3 with CPU usage monitoring code but without fair scheduling modifications to implement UNIX Solaris TS scheduling. A new data structure, xts\_multifb, defined in xts\_init.h is used to achieve constant overhead TS scheduling

struct xts\_multifb {

int status; // binary flag: 0 if queue is empty, 1 otherwise

qid16 queue\_head; // index to head of queue

qid16 queue\_tail; // index to tail of queue

};

extern struct xts\_multifb xts\_ready[];

where the size of the array xts\_ready[] is the same as xts\_conf[]. xts\_ready[] of type struct xts\_multifb replaces XINU's ready list to go from linear scheduling overhead to constant overhead. Thus we will ignore XINU's ready list but continue to use, struct qentry queuetab[NQENT], as the data structure used to maintain queues in XINU. This XINU specific approach to supporting queues works since a XINU process can only be in one queue (e.g., a process is ready or sleeping but not both).

Implement an internal kernel function, int xts\_enqueue(pid32 pid, pri16 prio), in xts\_enqueue.c that enqueues the process given by pid and priority prio into the multilevel feedback queue at the appropriate place. xts\_enqueue() return 0 if successful, -1 otherwise. Implement a function, pid32 xts\_dequeue(void), in xts\_dequeue.c that returns the process ID of a highest priority ready process from xts\_ready[]. If there are no ready processes outside of the kernel's NULL process, its PID 0 is returned. Implement a function, pri16 xts\_priochk(void), in xts\_priochk.c that returns the highest priority of (ready) processes in xts\_ready[]. xts\_priochk() is used by the scheduler resched() when determining if the current process should continue to run or not.

When making code changes to XINU, please follow the same guidelines as in lab3: clearly indicate where in the source code changes are being made and for what purpose, DEBUG printouts remain conditional.

### 3.3 Performance evaluation

Carry out the same performance evaluation as in Problem 4.3 of lab3. Use the same benchmark applications and calls to getcputot() to monitor CPU usage and gauge if your implementation of UNIX Solaris TS scheduling in XINU is achieving "fair" sharing of CPU cycles. Discuss your findings in Lab4Answers.pdf. Describe in your write-up how you are handling the NULL process under TS scheduling so that it is only executed if there are no other ready processes.

**Important: Before performing any kernel modification, please back up the files (even intermediate files to guard against accidental deletions) so that you can go back to a previous code base and recover more easily from human errors in the development process.**

## Bonus problem (20 pts)

The 60 priority levels used in UNIX Solaris has proven to be effective in real-world deployments, however, that is not to say that fewer priority levels will not work (almost) equally well. As mentioned in class, versions of BSD UNIX and Windows use significantly fewer priority levels and are known to operate well. Modify the UNIX Solaris TS implementation of Problem 3 so that priority levels are reduced from 60 to 20. The code should be the same except for reading 20 rows of values instead of 60 and initializing xts\_conf[] with different values. Create a new calibration table that follows the same format as [TS](http://www.cs.purdue.edu/homes/cs354/disp-table-solaris.txt) but has only 20 priority levels. Change the values of the entries so that the TS scheduling approach we discussed (higher relative priority to I/O-bound processes over CPU-bound processes, opposite for time slice) is followed. Set the values as you see fit so as to achieve fair sharing of CPU cycles. As in Problem 3, Benchmark the scheduler with the modified scheduling calibration table on the "half-and-half" test case only. Discuss your findings in Lab4Answers.pdf.

**Important: Please comment your code changes in Xinu such that (a) *where* changes are made is highlighted, and (b) *what* changes are made is conveyed.**

## 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 lab4 xinu-spring2018

You can check/list the submitted files using   
  
turnin -c cs354 -p lab4 -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.***