CS 503 - Spring 2014

Lab 1: Process Scheduling (100 pts)

Due Date: Monday 02/17/2014 11:59PM

In this lab you will implement a two-level process scheduler. And all scheduling policies in XINU that will avoid *starvation* of processes and improve CPU load balancing among I/O- and CPU-bound processes. At the end of this lab you will be able to explain the advantages and disadvantages of the scheduling policies implemented and evaluated in XINU.

Starvation is produced in XINU when we have two or more processes eligible for execution that have different priorities. The scheduling invariant in XINU assumes that, at any time, the highest priority process eligible for CPU service is executing, with round-robin scheduling for processes of equal priority. Under this scheduling policy, processes with the highest priority, if ready, will always be executing. As a result, processes with lower priority may never receive CPU time. For ease of discussion we call the set of processes in the ready list and the current process as *eligible* processes.

All processes in XINU will be put into one of the two scheduling group: proportional group and TS group. Within each group, processes will be scheduled via different scheduling policy. When ***resched()*** is invoked, it should decide which group should occupy the CPU at first. Then it picks up a process from this group to run. Here it should apply **Aging Scheduler** to pick the group. After the group is decided, it should pick up one process in this group via group-specific scheduling policy. Proportinal group should apply **Proportional Share Scheduler**; TS group should apply **TS Scheduler**. In the following we will introduce the three scheduling policies.

---

1. Aging Scheduler

The scheduling policy for process group scheduling is **aging scheduler**. On each rescheduling operation, the scheduler should count the number of processes from different groups in ready queue (e.g. there are 3 processes from proportional group and 4 processes from TS group in ready queue). It should increases the priority of the groups by their processes number (priority of proportional group increases by 3 and priority of TS group increases by 4). Then it should pick up the group with highest priority. If there are only processes from one group, just picking up this group directly.

Implementation sketch:

Each group has an *initial priority* . By default the priority of those two groups are both 10. And it could be changed via the call to *chgprio()*. Every time the scheduler is called it takes the following steps.

- If the *current* process belongs to group A, the priority of group A is set to the *initial priority* assigned to it.
- The priorities of all groups are incremented by the number of proccesses (in that group) in the ready

queue. Here you don't count the *current* process

Note that during each call to the scheduler, the complete ready list has to be traversed.

---

2. Proportional Share Scheduler

Every process in the proportional scheduling group has a scheduling parameter called *rate*. In the following, we will assume that all the processes in question belong to the proportional scheduling group. For a process *i* let us denote the *rate* as *Ri*. Every process *i* has a *priority value Pi*. Initially, all the processes start with a *priority value 0* (*Pi* = 0). Whenever a rescheduling occurs, the *priority value Pi* of the **currently** running process is updated as follows

$$Pi := Pi + (t * 100 / Ri)$$

where ***t*** is the CPU ticks consumed by the process since it was last scheduled. *Ri* is a percentage value and takes values between 0 and 100.

Now the scheduler schedules an eligible process with the ***smallest*** *Pi*. Whenever a process is scheduled the first time or is scheduled after blocking, its *Pi* value is updated as

$$Pi := max (Pi, T)$$

where *T* is the number of CPU ticks that have elapsed since the start of the system.

Intuitively, you can think of this policy as one that gives the processes some guarantees about their CPU share. If a process has a rate *Ri*, then it is guaranteed at least *Ri* percent of CPU time, provided the sum of all *Ri* values is less than 100. As the CPU usage of a process increases, its *Pi* value also increases depending on its *Ri*. If you have a large *Ri*, then your *Pi* increases more slowly and hence giving you a larger share of the CPU. Thus, *Ri* can be considered as a *share* of the CPU for the process *i*.

The second formula can be intuitively understood from the following example: Consider two processes A, B starting at time 0 and running continuously with rate 50 and 40 respectively. Let us say that another process C is created after 100,000 ticks with rate 10. C will start executing with a *priority value* of 0 (if the second formula were not to be applied) and hence will hog the CPU for a very long time and processes A, B have to wait for long to get the CPU back and would not enjoy their share of the CPU till all the *Pi* values level off. On the other hand, if the process C starts with a *priority value* 100,000 instead of 0, then it will run only for a short amount of time before yielding the CPU back to A and B.

Implementation sketch:

According to this policy, processes are scheduled in increasing order of their priority, i.e., a process with the lowest *priority value Pi* will be scheduled first. However, note XINU works in the opposite way, i.e., a process with the highest priority is scheduled first. In order to overcome this mismatch, we can maintain an internal variable *Pi* for every process which will contain the *priority value*. Let us denote the XINU process priority of a process *i* to be *PRIOi*. Then *PRIOi* can be calculated as ***PRIOi = MAXINT - Pi.*** As *Pi* increases, *PRIOi* decreases and the process will get lesser share of CPU.

To summarize, at every reschedule operation the proportional share scheduler does the following:

- The ***Pi*** value of the current process is modified and its ***PRIOi*** value updated.
- The scheduler chooses for execution the process with the highest priority, choosing from the processes in the ready list *and* the current process.

Also, whenever a process is scheduled for the first time or immediately after blocking, then the ***Pi*** value is changed as indicated above and the ***PRIOi*** is updated to reflect the change. You need to identify when and how to change ***Pi***.

---

3. TS Scheduler

A TS (timeshare) scheduler attempts to classify processes into CPU-bound and I/O-bound categories such that I/O-bound processes can be assigned higher priority for increased responsiveness without sacrificing fairness (i.e., starving CPU-bound processes). This is so since I/O-bound processes tend to voluntarily relinquish the CPU by issuing blocking system calls before their time quanta has expired.

Implementation sketch:

Implementing a TS scheduler entails two aspects: first, identification as I/O- or CPU-bound, and second, adapting process priorities. For the first step, we will use a simple (but also efficient) criterion, namely, whether the current process (if it belongs to the TSSCHED class), at the moment ***resched()*** is called, voluntarily relinquished the CPU or was forcibly interrupted by a clock interrupt handler because its time quantum expired. This may be determined inside ***resched()*** by checking the global variable *preempt* which counts down from the symbolic constant QUANTUM each time the clock interrupt handler is called (the internal details of the clock interrupt handler does not concern us here). A TSSCHED class process that, by this criterion, is considered I/O-bound (note that this only takes into account the most recent history) has its priority increased by 1. Of course, the maximum allowed priority (confer the range of the *pprio* field in the *pentry* structure) must not be exceeded. Note that we are not adapting the time quantum in this version of TS.

---

4. System call implemetation

create(void *funcaddr, uint32 ssize, int group, pri16 priority, ...)

Please add a new argument, *group* to this function (before *priority*) to this function. *group* should be either PROPORTIONALSHARE or TSSCHED. And for processes of proportional group, *priority* is used to define *Ri*. For processes which created by XINU by default (e.g. main, null), you could put them into any group of those two.

chgprio(int group, pri16 newprio)

You should add this new system call to change the initial priority of groups. Here you could have a look at system call*chprio* which changes processes' priority. The new group priority will be effective from the next scheduling.

resched()

This system call will be invoked for scheduling a process to run. Most of your work will be done here. So

please understand each line of code of this system call before you start to implement.

---

5. Turnin Instructions

Turnin instructions for Lab1 code (electronic):

1. Make sure to turn off debugging output.
2. Make sure that most of the code changes/addition you make are in the files *resched.c, create.c, chgprio.c.* If you need to declare new variables, functions etc., declare them in a file *lab1.h* inside the *xinu-14spring-linksys/include/* directory.
3. Go to *xinu-14spring-linksys/compile* directory and do "make clean".
4. Go to the directory of which your *xinu-14spring-linksys* directory is a subdirectory. (eg) if */homes/rsanjel/xinu-14spring-linksys* is your directory structure, goto */homes/rsanjel*
5. Submit using the following command:

turnin -c cs503 -p lab1 xinu-14spring-linksys

You can write code in main.c to test your procedures, but please note that when we test your programs we will replace the main.c file. Therefore, do not put any functionality in the main.c file.
