Lab 1: Process Scheduling (100 pts)

- CS 503 Spring 2018
- Due: Feb 21 2018, 11:59 PM

Objectives

In this lab you will implement a two-level process scheduler and few scheduling policies in XINU that will avoid *starvation* of processes and improve load balancing between 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 groups: PS1 group and PS2 group. Within each group, processes will be scheduled via Proportional share 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. In this lab, scheduler should run **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. In the following we will introduce the scheduling policies.

1. Aging Scheduler [30pts]

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 PS1 group and 4 processes from PS2 group in ready queue). It should increases the priority of the groups by their number of processes (priority of PS1 group increases by 3 and priority of PS2 group increases by 4). Then it

should pick up the group with highest priority. If there are processes only from one group, it just picks 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 and *null* process

Note that during each call to the scheduler, the complete ready list has to be traversed. Also, when both groups have the same priority, choose PS1 group.

2. Proportional Share Scheduler [70pts]

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 1 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 and 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). Hence C will hog the CPU for a very long time, and processes A and 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 using the second formula, 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***. Other than using previous method, you can keep track of minimum of ***Pi***, which requires extra traversals in readylist.

Benchmark sketch:

To evaluate Proportional Share Scheduling, you will need to experiment with two situtations : (i) when *Ri* has the equal rate. (ii) when *Ri* has the different

rates. Also, you should consider when some processes in this group are executed late or are blocked and come back later.

3. System call implementation

```
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 PS1 or PS2. 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 can put them into either PS1 or PS2.

```
#define PS1 0
#define PS2 1
```

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. **One thing to note** is that this lab's focus is not in scheduling's efficiency, but its correctness. Therefore, you can share readylist for both scheduling policy.

5. Turnin Instructions

Turnin instructions for Lab1 code (electronic):

1. Make sure to turn off debugging output. Also, please do not rename xinu-spring2018, or any of its subdirectories.
2. Go to *xinu-spring2018/compile* directory and do "make clean".
3. Go to the directory of which your *xinu-spring2018* directory is a subdirectory. (eg) if */homes/XXX/xinu-spring2018* is your directory structure, goto */homes/XXX*
4. Submit using the following command:

   turnin -c cs503 -p lab1 xinu-spring2018

5. After submitting your files, you may want to check that it's indeed submitted using command:

turnin -v -c cs503 -p lab1

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.

FALL:
1. Please exclude NULL process from your calculations. It's priority should always remain 0.
2. For t: find out t from preempt; counter to count the times clkhandler().
3. Non-blocked process: a). in state PR_CURR or PR_READY; b). preempt <= 0.
4. First, update the priority of the current process. Then, pick a group using the aging scheduler.
5. Change the priority of process (first time, blocking cases) once it has been picked from ready queue to schedule.

SPRING:
1. Bump the priority of main to highest priority so main exit quite quickly (chgprio in main; TA will handle it)

EXTRA:
1. exclude ‘null’ process from Aging Scheduler?
2. exclude ‘null’ from priority change, PS schedular

CHECK:
1. In general, processes from higher group priority should finish early.
2. For simulating blocking process, call "sleepms" inside the loop.

Question:
1. would ‘Start up’ process’ priority also be handled?
2. since last scheduled, keep current???