CSC501 Fall 2017

PA 1: Process Scheduling

Due: September 26 2017, 4:00 AM

1. Objective

The objective of this assignment is to get familiar with the concepts of process management, including process priorities, scheduling, and context switching.

2. Readings

The Xinu source code for functions in sys/, especially those related to process creation (create.c), scheduling (resched.c, resume.c suspend.c), termination (kill.c), changing priority (chprio.c), system initialization (initialize.c), and other related utilities (e.g., ready.c), etc.

3. What To Do

The default scheduler in Xinu schedules processes based on the highest priority. Starvation occurs when two or more processes that are eligible for execution have different priorities. The process with the higher priority gets to execute first, resulting in processes with lower priorities never getting any CPU time unless process with the higher priority ends.

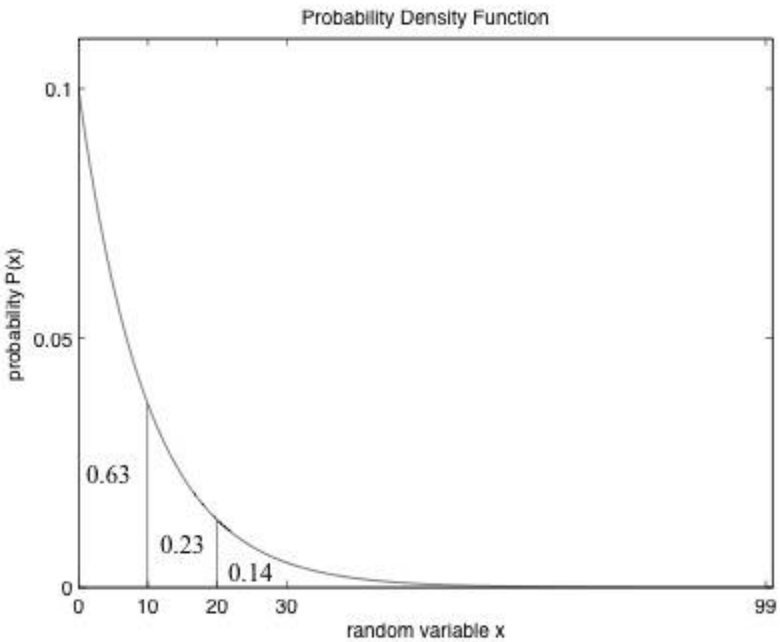
The two scheduling policies to implement, as described as follows, address this problem.

For Linux-like scheduling policies, the value of a valid process priority is an integer between 0 to 99, where 99 is the highest priority.

1) Exponential Distribution Scheduler

The first scheduling policy is the **exponential distribution scheduler**. This scheduler chooses the next process based on a random value that follows the exponential distribution. When a rescheduling occurs, the scheduler generates a random number with the exponential distribution and chooses a process with the lowest priority that is greater than the random number. If the random value is less than the lowest priority in the ready queue, the process with the lowest priority is chosen. If the random value is no less than the highest priority in the ready queue, the process with the largest priority is chosen. When there are processes having the same priority, they should be scheduled in a round-robin way.

For example, let us assume that the scheduler uses a random value with the exponential distribution of $\lambda = 0.1$, and there are three processes A, B, and C, whose priorities are 10, 20, and 30, respectively. When rescheduling happens, if a random value is less than 10, process A will be chosen by the scheduler. If the random value is between 10 and 20, the process B will be chosen. If the random value is no less than 20, process C will be chosen. The probability that a process is chosen by the scheduler follows the exponential distribution. As shown in the figure below, the ratio of processes A (priority 10), B (priority 20), and C (priority 30) to be chosen is 0.63 : 0.23 : 0.14. It can be mathematically calculated by the cumulative distribution function $F(x; \lambda) = 1 - e^{-\lambda x}$.



In order to implement an exponential distribution scheduler, you need to implement expdev(), which generates random numbers that follow the exponential distribution. The generator can be implemented by $-1/\lambda * \log(1 - y)$, where y is a random number following the uniform distribution in [0, 1].

2) Linux-like Scheduler (based loosely on the Linux kernel 2.2)

This scheduling algorithm loosely emulates the Linux scheduler in the 2.2 kernel. We consider all the processes "conventional processes" and use the policies of the `SCHED_OTHER` scheduling class within the 2.2 kernel. In this algorithm, the scheduler divides CPU time into continuous *epochs*. In each epoch, every existing process is allowed to execute up to a given *time quantum*, which specifies the maximum allowed time for a process within the current epoch. The time quantum for a process is computed and refreshed at the beginning of each epoch. If a process has used up its quantum during an epoch, it cannot be scheduled until another epoch starts. For example, a time quantum of 10 allows a process to only execute for 10 ticks (10 timer interrupts) within an epoch. On the other hand, a process can be scheduled many times as long as it has not used up its time quantum. For example, a process may invoke `sleep` before using up its quantum, but still be scheduled after waking up within an epoch. The scheduler ends an epoch and starts a new one when all the **runnable** processes (i.e., processes in the ready queue) have used up their time.

The rules for computing the time quantum for a process are as follows. At the beginning of a new epoch, the scheduler computes the time quantum for **every** existing processes, including the blocked ones. As a result, a blocked process can start in the new epoch when it becomes runnable. For a process that has never executed or used up its time quantum in the previous epoch, its new quantum value is set to its process priority (i.e., `quantum = priority`). For a process that has not used up its quantum in the previous epoch, the scheduler allows half of the unused quantum to be carried over to the new epoch. Suppose for each process there is a variable `counter` describing how many ticks are left from its previous quantum, then the new quantum value is set to `floor(counter / 2) + priority`. For example, a counter of 5 and a priority of 10 produce a new quantum value of 12. Any new processes created in the middle of an epoch have to wait until the next epoch to be scheduled. Any priority changes in the middle of an epoch, e.g., through `create()` and `chprio()`, only take effect in the next epoch.

To schedule processes during each epoch, the scheduler introduces a *goodness value* for each process. This goodness value is essentially a dynamic priority that is updated whenever the scheduler is invoked. For a process that has used up its quantum, its goodness value is 0. For a runnable process, the scheduler considers both the priority and the remaining quantum (recorded by `counter`) in computing the goodness value: `goodness = counter + priority`. The scheduler always schedules a runnable process that has the highest *goodness* value (it uses the round-robin strategy if there are processes with the same goodness value). The scheduled process will keep running without being preempted until it yields or uses up its time quantum.

Examples of how processes should be scheduled under this scheduler are as follows:
If there are processes P1, P2, P3 with priority 10, 20, 15, the initial time quantum for each process would be P1 = 10, P2 = 20, and P3 = 15, so the maximum CPU time for an epoch would be 10 + 20 + 15 = 45. The possible schedule for two epochs would be P2, P3, P1, P2, P3, P1, but not: P2, P3, P2, P1, P3, P1.
If P1 yields in the middle of its execution (e.g., by invoking `sleep`) in the first epoch, the time quantum for each process in the second epoch could be P1 = 12, P2 = 20, and P3 = 15, thereby the maximum CPU time would be 12 + 20 + 15 = 47.

3) Other Implementation Details

1. `void setschedclass (int sched_class)`
   This function should change the scheduling type to either `EXPDISTSCHED` or `LINUXSCHED`.
2. `int getschedclass()`
   This function should return the scheduling class, which should be either `EXPDISTSCHED` or `LINUXSCHED`.
3. Each of the scheduling class should be defined as a constant:

```
#define EXPDISTSCHED 1
#define LINUXSCHED 2
```

4. Some of the source files of interest are: create.c, resched.c, resume.c, suspend.c, ready.c, proc.h, kernel.h, etc.