Project 2 Design Doc

Richard Nicholson, James Shedel, James Kuch, Mark Serrano, Tyler Smith

May 2, 2014

1 Design

1.1 Overview

In this project we were to design a userspace lottery scheduler for Minix. Since it runs in userspace, it only adjusts parameters to the kernel's own round-robin scheduler. Here, we describe the correct behaviour for the userspace scheduler.

1.2 Starting scheduling for a process

When the scheduler starts scheduling a process, it checks to see whether the process is a new process or if it is being inherited from another process. If it is inherited, it gets the same priority and quantum as the parent. If it is a new process however, we set the priority and quantum to be our default values. We also assign it a certain number of tickets (the default being 20) and update the total number of tickets.

1.3 Stopping scheduling for a process

When the scheduler stops scheduling a process, it updates the total number of tickets. If something goes wrong, it returns an error.

1.4 Adjusting tickets for a process

The number of tickets for a process can be adjusted using the `nice` command as described in 2.2.3.

1.5 Lottery

The lottery should pick a random number and then find the process that has the winning tickets. The process that has the winning ticket should be elevated in priority for the next quantum. The more tickets a process has, the more likely it will win the lottery, and therefore the more likely that it will execute with a higher priority.

2 Implementation

2.1 Modified

Implementation edited the following files of Minix:

- /usr/src/servers/sched/schedproc.h
- /usr/src/servers/sched/schedule.c
- /usr/src/servers/pm/schedule.c

New functions defined:

- int do_lottery()
    - runs the lottery, selects a winner, elevates and de-elevates priorities, and schedules
    - returns status of schedule_process_local

Functions modified:

- pm/schedule.c
    - int sched_nice(struct mproc * rmp, int nice) - pm/schedule.c
        - modified to help change the format in the nice program to affect tickets
- sched/schedule.c
    - int do_noquantum(message *m_ptr)
        - calls do_lottery() at the end
    - int do_stop_scheduling(message *m_ptr)
        - modifies total ticket value and calls do_lottery()
    - int do_start_scheduling(message *m_ptr)
        - modifies total ticket value and gives 20 tickets to each process
    - int do_nice(message *m_ptr)
        - changes a process' ticket value
    - void init_scheduling(void)
        - initializes total_tickets to 0

2.2 Overview of Important Issues

2.2.1 Selecting a Winner

1. The total number of tickets would be kept track of in schedule.c as the global variable total_tickets. This value would be increased and decreased in do_start_scheduling(), do_stop_scheduling(), or whenever the ticket number was changed as part of the dynamic scheduler.
2. Whenever do_lottery() is called, a winner is chosen by means of random() % total_tickets.
3. The process table is then iterated through linearly, and the process that won has its priority elevated, as well as the previous winner having its priority lowered.

2.2.2 Rescheduling Processes

1. A lottery is held at the end of every call to do_noquantum() or do_stop_scheduling()
2. If a process wins the lottery, it's priority is increased by one and the previous winner's priority is decreased by one.

2.2.3 Tickets

We implemented tickets using the `nice` command. The `nice` command accepts a value between -20 and 20, and converts this value to a number between 1 and 100, with -20 corresponding to 1 and 20 corresponding to 100, intermediate values being evenly distributed across the range. The converted value is added to the number of tickets the target process holds.

3 Design Commentary

3.1 Linear winner-choosing

We decided that an O(n) implementation of the winner-choosing algorithm would be acceptable because the userspace process table only has 256 entries by default and, in practice, not all of these would be filled anyway. In the end we opted for this solution due to simplicity and the fact that it introduced no additional space complexity.

3.2 What went wrong

Our scheduler doesn't work. We ran into an issue where, after elevating the winning process' priority and lowering the previous winner, execution would just kind of hang on everything until balance_queues() is called after 5 seconds, at which point everything would go back to normal for roughly a second before hanging again. Even in its base state though, we couldn't get do_noquantum to be called more than once a second, meaning that time-slicing doesn't work the way we believed. The code as it currently is represents us struggling with this problem for a long period of time and thus, in some ways, does not accurately represent our initial design principles or even basic logical reasoning.

We didn't proceed enough in the project to start to implement dynamic scheduling and so chose not to include it in our design document.

3.3 But what we learned

An operating system is like a 18th century Parisian courtesan- complex and interesting, but high maintenance. We should’ve started this project earlier.