Project 2 Design Document

Craig Collins Ryan Gliever Mat Fukano
Leland Miller

April 2014

ABSTRACT

This project implements a lottery scheduler for prioritizing processes in the MINIX operating system. Each process has some n number of tickets. All processes' tickets are put into a pool of tickets. A random selection is made, and the process corresponding to the winning ticket drawn will be prioritized for some amount of time, before being deprioritized again.

1 INTRO

NOTE: This program has 2 versions: Dynamic Lottery Scheduler and Static Lottery Scheduler. Edit the schedule.c file in path /usr/src/servers/sched, and change the macro DYNAMIC to 1 to enable Dynamic and 0 to enable Static.

In MINIX, scheduling is done partly in the user-level process "sched". We will be making most of our changes in schedule.c, located in path /usr/src/servers/sched/. There are four notable functions in this file:

`do_noquantum`

called "on behalf of process' that run out of quantum", i.e. whenever a process uses the entirety of its time slice, this function is called. The function lowers the priority of the process it was called on behalf on, and reschedules that process.

`do_start_scheduling`

sets priority, time slice, and endpoint for whatever process it was called for, and then schedules the process.

`do_stop_scheduling`

opposite of `do_start_scheduling`; stops scheduling for processes that have completed or run out of quantum.

`do_nice`

sets the "nice" level of a process. The "more nice" a process is, the higher its priority.

MINIX uses these functions for its regular scheduler. We will be manipulating these functions to implement our lottery scheduler. We will also be editing the file schedproc.h, located in the same path as schedule.c. The schedproc.h file contains the struct schedproc, which refers to a process' endpoint id, max priority, priority, and time slice. We will manipulate this file in implementing our lottery scheduler.

2 DESIGN

Now that we have a basic understanding of the functions in schedule.c and schedproc.h, we can start integrating lottery scheduling into the MINIX scheduler.

2.1 Changes to user/src/servers/sched/schedproc.h

In schedproc.h, we will be making some minor changes to the struct schedproc. We will add two more fields, `unsigned num_tickets` and `unsigned nice` to the existing fields. `num_tickets` will hold a value equal to the number of tickets that a process has. `nice` will hold a value that corresponds to the behavior of the process; the nice value will be used in the `do_nice` function to adjust the number of tickets a process has.

2.2 Changes to user/src/servers/sched/schedule.c

`lottery`

Here is where the main code for our lottery scheduler is located. The main logic of our code will grab every process in the 15th queue and add its number of tickets into an `unsigned total_tickets`. Next we randomly generate a winning number between 1 and `total_tickets`, inclusive. We go through a second loop and subtract each processes ticket count from the winning number. When the winning number becomes less than or equal to zero, the process thats tickets we just subtracted is the winner. It then gets moved to queue 14.

`do_noquantum`

Here is where we will be calling `lottery`. When the quantum runs out for a winner process (a process in the winner's queue), we will move the process down to the loser's queue and hold a new lottery. If the quantum runs out for a process in the loser's queue, then we know that the winner process has blocked for some reason. In this case, we will let the blocked winner process stay in the winner's queue, hold a new lottery, and bring another process from the loser's queue to the winner's queue.

`do_start_scheduling`

The code here will be mostly unedited from the original. However, we will give the processes an initial number of tickets here for usage in the lottery later.

`do_stop_scheduling`

Here is where we will call lottery as well. The process here will be similar to that of do_noquantum. If do_stop_scheduling is called for a process in the winner's queue, then we will move the winner to the loser's queue and hold a new lottery. If do_stop_scheduling is called for a loser process, then we will keep the blocked winner process in the winner's queue, hold a new lottery, and bring a process from the loser's to the winner's queue.

`do_nice`

Here is where we adjusted the do_nice function which works with the "nice" command. We created a way to to adjust the ticket count based on the 'nice' of a process. the syntax is : "nice -n [number of tickets] [process]". Keep in mind that all processes start with 20 tickets so the number of tickets here will just be added to 20. **Our code is optimized when the number of tickets added is between 20 and -19.**

2.3 Changes to user/src/include/minix/config.h

`#define MAX_USER_Q`

Here, we changed the macro MAX_USER_Q from 0 to 14. We used this value as the "winner's queue," where the winner of the lottery will be placed. MIN_USER_Q is set at 15 by default, and we used that value as the "loser's queue."

2.4 Changes to user/src/servers/pm/schedule.c

We made a small change in this file. All we did was change the m.SCHEDULING_MAXPRIO to be equal to 'nice'. This would then affect the same message pointer passed into the do_nice function.

3 LOTTERY ALGORITHM

Our lottery algorithm looks through the buffer of processes and checks each one to make sure it's in use, not a system process, and in the "loser" queue, or the lowest user-space priority queue. If it is, it adds the number of tickets that process holds to the total number.

```
for(process in processes){
  if(proc->flags & IN_USE && proc->is_sys_proc != 1 &&
    proc->priority == loser_q) {
      tickets += proc->tickets;
  }
}
```

User space process check

After this, we have the total number of tickets for all processes in user-space. We then iterate through, finding the winner of the lottery by generating a random seed, modding that number by the total number of tickets (incremented by 1, so we'll always have 1 ticket), and checking cases therein. If the process follows the conditions of the user-space process check, then we look to see if we have more than zero tickets. If so, we decrement the number of tickets a particular process has from the modded total from before. If that total is now less than, or equal to, zero, and is in the loser queue, we decrement the current process's priority by one, moving it into the winner's queue, and schedule another process.

```
if (tickets > 0){
  winner = random % tickets + 1;
  for (process in processes) {
    if (user-space process check) {
      if (winner > 0){
        winner -= proc->tickets;
        if (winner <= 0 && proc->priority == loser_q){
          proc->priority -= 1;
          schedule_process();
        }
      }
    }
  }
}
```

Main lottery loop

4 TESTING

Our testing system consists at its core of two programs that simulate real-life processes. One of these simulates an CPU bound process and the other simulates a IO bound process. These programs are called `cpu_sim` and `io_sim` respectively.

`cpu_sim` repeatedly runs a remainder operation, as this was found to take up a good amount of resources. To use it takes a command line argument that represents an amount of work for the program to do. A value of 10 took approximately 10 seconds in testing, though this value should not be thought of as a time value (in reality a value of one represents 250 million remainder operations). The program outputs three messages while its running and one when it is finished.

`io_sim` takes an input file and computes a checksum while copying the file to a new destination as the instructions suggested. This allows for simulation of an IO bound process.

The quiet versions of these programs minimize the output, but also print a start and finish time to stdout. These are what the script uses to provide useful output data. Using these programs, the testing script, and top to monitor priority levels was the core of our testing strategy. The testing script (`run_tests.sh`) runs the programs in different configurations and consolidates the output into something useful for analyzing.

Usage

To run the testing script simply go into the testing directory and run make. This will create the file test_results.md in the directory with the results of our tests. make clean can then be used to clear the binary files out of the directory. The script will create a subdirectory tmp and then remove it at the end. It uses about 100 megabytes for the test though this could be easily changed. This is just to run IO tests. The make target just builds the programs and runs the script run_tests.sh.