Project 2 Design Document

Team BestOS2014:
Simon Ho
Christian Ambriz
Jim York
Eric Kaneshige
Josh Nussbaum

May 2, 2014

Instructions on how to 'make' is found in README.txt.

Design:

Implemented a lottery scheduler for user processes only. Processes run by the user are given tickets, a lottery is held to determine which process is allowed to run during the current time slice. A process which holds more tickets would be much more likely to have the opportunity to run.

We chose not to modify the queues in ~/minix/config.h. All processes start in Queue 15, and if a winner is chosen, it moves to Queue 14 where it has higher priority, and is ran. After the quantum has passed, the running process stops and a new lottery is held again. The process that was running moves back down to the 'loser' queue, Queue 15. If a process finishes running, it is taken out of the queues and its respective tickets will be removed from the lottery's ticket pool. This system loops until all processes have finished executing. The lottery scheduler has two different ways of functioning, depending on which schedule.c was used during the make:

1) **static:** The user specifies the number of tickets a process has using the 'nice' command. For example, 'nice –n 1 ./longrun 33 1000' would give the longrun program 21 tickets (33 and 1000 are just inputs for the longrun program).

2) **dynamic:** The scheduler decides for itself the allocation of tickets to processes. Each process starts with 20 tickets. If a process wins the lottery, it is moved to the higher priority queue and is ran for a given time slice. After the process finishes running, it is moved back down to the lower priority queue and its ticket count is decremented by 1. All of the processes that had not been chosen have their ticket count incremented by 1. If the chosen process blocked, when the quantum is finished, a process in the lower priority queue would return instead of the 'winner' process. In this case, the chosen process would be moved back down and have its ticket count incremented. We do not increment or decrement the tickets of the 'loser' process that ran when the 'winner' blocked.

Implementation:

The implementation is divided into 3 files:

- /usr/src/servers/sched/schedproc.h – defines the schedproc struct to hold tickets.
- /usr/src/servers/sched/schedule.c – defines the main scheduler.
- /usr/src/servers/pm/schedule.c – defines what 'nice' will do.

Overview of important issues:
Prioritizing IO over CPU processes – To make sure that IO processes are able to run when the user needs them to, we increase the ticket counts of blocking processes, as these are most likely IO tasks. When a process wins the lottery but blocks during its quantum, a process from the 'loser' queue runs in its place. After that, the blocking process receives an extra ticket, such that it has a higher likelihood of being drawn when it is needed to be ran.

config.h: no changes were made to the original config.h file.

schedproc.h: created the new field 'tickets' for the schedproc struct for use in schedule.c.

~/pm/schedule.c: commented out 'maxprio' and changed m.SCHEDULING_MAXPRIO to = (int) nice. Originally, when the user called 'nice,' the input would be translated into a priority when passed to do_nice. Now, the input after calling 'nice' instead of changing the user input into a priority number, the input number is added to the number of tickets for the process. (Ex: 'nice –n 5' adds 5 tickets to the process' default tickets; 20 + 5 = 25 tickets for that process.)

~/sched/schedule.c:

PRIVATE void play_lottery(void)
This function runs the lottery function which selects (at random) a ticket from the pool of process tickets. When the winner is drawn, it calls schedule_process() on it to schedule the process to be run.

PUBLIC int do_noquantum(message *m_ptr)
In STATIC: called when the quantum runs out. Here, we check if the winner is blocked by looking if it executes inside of the LOSER_QUEUE. If that happens, we want to put the winner back into the LOSER_QUEUE, and then reschedule it. If the winner is not blocked, then do_noquantum is called in the winner queue. if that is the case, place that process into LOSER, and then reschedule. Once done, play the lottery again. Calls play_lottery();
In DYNAMIC: Does the same thing as in the static case, with the addition that the function will increment 'loser' and blocked process tickets, as well as decrement winning processes that finished running their quantum.

PUBLIC int do_stop_scheduling(message *m_ptr)
This function is executed when a running process terminates. In such a case, the terminating process will be removed from the queues, as it is finished running, and its respective tickets decremented from the lottery's ticket pool.

PUBLIC int do_start_scheduling(message *m_ptr)

Handles requests, populates process slots, and assigns queue priorities for processes, including system processes. Initializes all processes to the 'losers' queue. Processes stay in this queue until the lottery draws a winning process. This function was also edited to initialize processes (rmp) with 20 tickets. The total number of tickets in the lottery's ticket pool is also initialized here, to be the sum of all the tickets of processes that need to be run. This function calls 'schedule_process' to schedule a given process and give it quantum.

PUBLIC int do_nice(message *m_ptr)

Checks for valid endpoint, and then makes an adjustment to the number of tickets the process to be ran holds. See changes made to nice in ~/pm/schedule.c for information on how 'nice' functions with the changes made. If an error occurs (process is not OK), the ticket count reverts back to default (20).

do_nice will not be called in DYNAMIC mode, as the program takes care of adjusting the number of tickets a process holds.

PRIVATE int schedule_process(struct schedproc * rmp)

Schedules the given process (rmp), and prints an error if it is not OK.

PUBLIC void init_scheduling(void)

Initializes and sets the timer.

PRIVATE void balance_queues(struct timer *tp)

Resets the timer.

Testing files:

longrun.c: a test program that loops forever (unless given a maximum), adjusted so that a compiler would not be able to tell offhand that it is an infinite loop and thus optimizing it.

Usage: %s <id> <loop count> [max loops]

It takes in 2 (up to 3 arguments), an id string, and a loop count. If given max loops, it will stop looping when the number of iterations has reached the max loop count. Longrun prints the id string and the iteration during every loop.

increments.c: Simulates CPU bound process.
a test script that runs x number of forks that do an incremention process from an initial value y to (y + 100) with an incrementation of 0.000001.
The initial value is an irrelavent value that is equal to it's fork ID number multiplied by 2.
The x variable is determined by the parameters given when the program is run.
The program either accepts either none or one double parameter.
If the program is given a parameter, x is equal to that parameter.
If there are no parameters given, then x is equal to one.
Each fork records how long it took to compelete its incrementation operation.
Once each fork is created, then the time for when the fork was created is recorded.
After the process of the fork is finished, then the fork will print its ID and finished run time.
The run time is calculated by finding the difference between its end time and the time that the fork was created. In MINIX, the time accuracy is incorrect, but the difference between the various forks is still consistent.
Checksum.c: Simulates IO bound process with little CPU usage.

Main: reads in a file and copies its contents to a new file named “copyfile”. It then copies the file to a character array and saves the number of bytes read from the file. The function checksum is then called and the results are printed.

checkSum: adds up the integer value of all of the characters in the file and returns the result.

Testing Procedure:

Static: Open two terminals in host machine, and ssh into MINIX with both. Make and run the longrun program on both, using different nice values to give them different priorities.

Example:

```
Terminal1$  nice –n 0 ./longrun 33 100
Terminal2$  nice –n 20 ./longrun 35 100
```

After these commands are executed (at around the same time), we let the programs run and observe the number of iterations they both have gone through. The longrun process in terminal 2 should be running more often than the process in terminal one. We can observe this by looking at the outputs of the longrun processes, specifically at the number of iterations there have been. The iteration number in terminal 2 should be significantly higher as time goes on, tending towards having 100% more iterations than the process in terminal 1, as it would have 20+20 = 40 tickets as opposed to terminal 1’s 20+0 = 20 tickets – a ratio of 2:1, thus having 100% more iterations.

When given the same nice values, both longrun programs should output iteration numbers that are very close to being equal to one another, as both processes should be drawn by the lottery at the same rate.

If we open 3 terminals and run tests using nice values of 25, 50, and 100, respectively, as time goes on, we can observe that the amount of times that each process is run will tend toward the ratio of 1:2:4.

1:100 ticket test – when 2 terminals are used and nice values of 1 and 100 are given respectively, we will observe that the process holding only 1 ticket will still run, although at a much slower rate due to the lottery not drawing the ticket anywhere near as often as the process with 100 tickets.

Dynamic: Open multiple terminals in the host machine, and ssh into MINIX. Make both increments and checksum. Then run variations of both programs on their respective terminals, and observe/compare the ticket counter, winners, and losers of the lottery system. It should be seen that the checksum processes would end up holding higher ticket counts and thus be run more often than the increments processes, due to being IO bound. IO bound processes block more often, and thus will have their ticket counts incremented more often.