Project 2: Lottery Scheduler

Design Document

Matt Caruano, Taylor Caswell, Eric Evans, Mesuilame Mataitoga, Radhika Mitra

Purpose

To implement a dynamic and static lottery scheduler in Minix

Available Resources:

We used the following files and made changes to them:

sched/schedule.c

pm/schedule.c

sched/schedproc.h

The following is structure from schedproc.h is the most important as far as our assignment goals are concerned. More specifically, the member variables *priority* and *num_tickets* are used to track the queue that a process are currently in and how often it gets to run.

```
EXTERN struct schedproc {
	endpoint_t endpoint;  /* process endpoint id */
	endpoint_t parent;    /* parent endpoint id */
	unsigned flags;              /* flag bits */

	/* User space scheduling */
	unsigned max_priority;       /* this process' highest allowed priority */
	unsigned priority;           /* the process' current priority */
	unsigned time_slice;         /* this process's time slice */
	unsigned num_tickets;    /* Number of lottery tickets */
} schedproc[NR_PROCS];
```

Design Plans

- Switching between static lottery and dynamic lottery:
  Made a #define in sched/schedule.c : # define STATIC_LOTTERY (if 0 then static lottery is off and dynamic lottery is on, if 1 then static lottery is on and dynamic lottery is off
- Initializing tickets
  Added a variable in sched/schedproc.h: int num_tickets
  It is initialized to 20 tickets and the bound is from 1 to 100

- The nice() function can only lower the priority of processes. This follows the philosophy that user space processes should not be allowed to increase their own priority, as this creates a vulnerability that can be abused by malicious programs.
- Made changes to the following functions:
    - ➔ do_start_scheduling ()
      - In the case of SCHEDULING_INHERIT change the priority of the process to LOSER_QUEUE which is #defined as value 14
    - ➔ do_noquantum () found in sched/schedule.c
      - *Designed a way to assign tickets and retract tickets to processes:*
      Assign a ticket: If a process is in the LOSER_QUEUE, and if there is a process in the WINNER_QUEUE, give the process in the winning queue a ticket
      Retract a ticket: If a process used up its quantum 5 times in the WINNER QUEUE, take away a ticket
      - *Implement the lottery scheduler*
      Iterate through all processes in loser queue, sum up all of their *num_tickets* and assign it to variable *ticket_sum*
      Generate a random number from 1 - *ticket_sum*
      Iterate through all processes again in the user queue, and subtract all of their *num_tickets* from the random number
      When the random number hits 0 or less, add the process to the WINNER_QUEUE
    - ➔ sched_nice() found in pm/schedule.c
      Reassigned variable *m.SCHEDULING_MAXPRIO* to the actual value of variable *nice*
      Purpose of this being done was to access the actual value of nice from the do_nice() function called in sched/schedule.c
    - ➔ do_nice() found in sched/schedule.c
      Uses the field *SCHEDULING_MAXPRIO* from the passed in message to increment or decrement the tickets of the process it was called on
- Determining CPU bound vs. IO bound processes
  We will default each process with 20 tickets.
  We will be using existing queues in sched/schedproc.h
  The WINNER_QUEUE define # 13 will contain the active process and LOSER_QUEUE define # 14 will carry all of the waiting processes.

Initially all processes are in the wait queue (14) except the current process that is running will be in the active queue(13).

CPU Bound: If a process calls do_noquantum() from the WINNER_QUEUE then we assume its CPU heavy and we take a ticket away.

IO Bound: If a process calls do_noquantum() from the LOSER_QUEUE then we know a process blocked in the WINNER_QUEUE and give every process in the WINNER_QUEUE a ticket. Although IO bound process will call do_noquantum () from the WINNER_QUEUE occasionally, they will be dominated by gaining tickets from waiting in the winner queue.

Testing:

cpu bound process:

crunch()

```
start_time = current time
loop a simple math operation that runs roughly 10 seconds
end_time = current time
run_time = end_time - start_time
print run_time
```

fast()

```
print “fast() completed”
```

Our makefile executes the following tests:

1. Show that running two equal CPU bound tasks with equal tickets lets them run at about the same speed
   -> Run crunch() twice simultaneously both with 20 tickets.
   -> Output: The output from crunch () for both processes should be approximately the same.
2. Show that running two CPU tasks where 1 has twice the tickets, the task with more tickets finishes in 1/2 the time
   -> Run crunch() twice simultaneously , one with 10 tickets and another with 20 tickets.
   -> Output: Of the two values printed to the screen, the first value printed should be roughly half the value of the second.
3. Show that running three CPU tasks with 25, 50, and 100 tickets runs the tasks in the right ratio
   -> Run crunch() three times simultaneously, one with 5, another with 10, and the third with 20 tickets. (This follows the same ticket ratio as 25:50:100).
   -> Output: The Three different outputs should hold roughly the same ratio 25:50:100
4. Show that running several CPU tasks with 100 tickets doesn't completely starve another task with just 1 ticket

-> Run crunch() five times simultaneously all with 20 tickets, along with fast() once with 5 tickets.
-> Output: Among the values that are output for crunch, you should also see a message for fast() printed. The fast() message should print somewhere between the other 5 messages from crunch. This will indicates that a process with 1 ticket is not getting starved buy the 100 ticket processes.

NOTE: Tests 5 and 6 will be conducted under one test

5. Show that your dynamic scheduler actually does dynamically adjust tickets of processes
6. Show that your dynamic scheduler improves performance when you mix CPU and IO bound tasks compared to keeping a fixed number of tickets for each process

-> The file IObound.c creates an executable IObound that simulates an IO bound process. The main loop(infinite) of the process writes data (~30MB) to a file, then it rewind()s back to the top of the file for the next loop.
To show that tests 5 and 6 succeed, run IObound alongside two instances of crunch and view their queue placements with the top command. IObound should be predominantly in the winner queue, queue 13, while the two crunches will take turns between queue 14 and 13.