David Merrick

CS 411

13 April, 2013

Individual Writeup for Project 1

1. *What do you think the main point of this assignment is?*

   There were several points to this assignment. The first was to get a sense for how to do kernel development. In particular, the location of the kernel files, what their functions are, how to properly configure the kernel to match the desired features and hardware, how to compile the kernel, where to place the compiled kernel, booting into the new kernel, etc. The second was to learn new methods for debugging and testing, as our code was running at a much lower level than any most of us had previously written. The third was to learn how to use SVN for revision control. Most of us had only used Git, so this presented a challenge. The fourth was to learn to leverage some tools that are essential for kernel development, including diff for creating kernel patches and grep to search for files containing definitions and functions. Another point was to coordinate a group of randomly-assigned team members to accomplish these tasks. Overall, though, I think the main point of the assignment was to get an understanding of how process scheduling in Linux works at the kernel level.

2. *How did you approach the problem? Design decisions, algorithm, etc.*

   Background: Real-time scheduling means to schedule processes within timing deadlines. The Linux scheduler provides soft real-time behavior, meaning that it tries to schedule processes within timing deadlines but doesn't make guarantees to always achieve this. Linux provides two real-time scheduling policies, FIFO (SCHED_FIFO) and round-robin (SCHED_RR). I will be using SCHED_RR and round-robin interchangeably in this writeup, likewise for SCHED_FIFO and FIFO. Both round-robin and FIFO make use of runqueues, which are essentially queues containing process descriptors for runnable processes. Round-robin scheduling cycles through processes, running each for a pre-specified amount of time known as a timeslice. On a more technical level, this means placing the process descriptor at the end of the runqueue after its timeslice, then running all subsequent tasks until it reaches the end of the queue, at which time it will start the cycle again. FIFO (First In, First Out) scheduling runs each SCHED_FIFO process of the same priority until it is finished, then runs the next process in the runqueue. It does this in order of when processes were placed in the runqueue. On a technical level, FIFO scheduling behaves almost identically to round-robin scheduling but with infinite timeslices.

   Design decisions: We thought the most efficient approach to the design would be to leverage the data structures, functions, and system calls already in place in the project kernel provided for us, as opposed to reinventing the wheel by rewriting these ourselves. We decided to first track down the relevant files for real-time scheduling, then narrow these down to the functions we would need to modify or implement within these files. We knew that the files we needed to modify would be found under the "kernel" directory in the root of the kernel source tree. This folder contains core subsystems, including the scheduler. Within this directory, we used 'grep "SCHED_RR" ./sched*' and 'grep "SCHED_FIFO" ./sched*' to determine which of the scheduler files we needed to edit. The first file we edited was sched.c. This file contains the definitions for the base scheduler code, which iterates over each scheduler class in order of priority to select which process runs next. We diffed this file with its implementation in the actual 3.0.4 kernel to give us an idea of what changes we would need to make.

   Data structures and functions related to real-time scheduling in sched.c: rt_rq is the runqueue data structure for real-time tasks. enqueue_task(), dequeue_task(), and activate_task() are all functions related to adding and removing tasks from the runqueue. These had all been implemented for us in the project kernel. prepare_task_switch(), finish_task_switch() both pertain to switching between tasks. rt_policy(), task_has_rt_policy(), init_rt_bandwidth(), rt_prio(),
   rt_mutex_setprio(), rt_mutex_adjust_pi(), init_rt_rq(), init_tg_rt_entry(), normalize_rt_tasks(), free_rt_sched_group(), alloc_rt_sched_group(), free_rt_sched_group(), alloc_rt_sched_group(), tg_has_rt_tasks(), rt_bandwidth_enabled(), __rt_schedulable(), sched_group_set_rt_runtime(), sched_group_rt_runtime(), sched_group_set_rt_period(), sched_group_rt_period(), sched_rt_global_constraints(), sched_rt_can_attach(), sched_rt_global_constraints(), sched_rt_handler(), cpu_rt_runtime_write(), cpu_rt_runtime_read(), cpu_rt_period_write_uint(), cpu_rt_period_read_uint() are all functions in sched.c related to real-time tasks. Of these, we only had to modify rt_policy() to enable real-time scheduling. We did this by just having it return a 1 if the scheduling policy is real-time (meaning SCHED_FIFO or SCHED_RR).

Implementing round-robin scheduling: For round-robin scheduling, we knew that we needed processes to be run one after the next, repeating, with a pre-defined timeslice to run. We found this interval defined in the sched.c file in the constant, DEF_TIMESLICE. From the comments in the file, we knew that this constant was intended for SCHED_RR tasks (FIFO tasks do not use a timeslice in theory; In practice they essentially have an infinite timeslice). By default, this value was set to be 100ms. Without the knowledge of the overhead of switching cost of tasks on this particular architecture, we left it defined the way it was. This value is critical to the operation of the scheduler. If it is too high, the system may not be as responsive and processes will not appear to be executing concurrently. If it is too low, the system will be very inefficient. For example, if the cost of switching tasks is 10ms and the timeslice is 10ms, then 50% of CPU cycles will be dedicated to switching between tasks.

To implement round-robin and FIFO scheduling, we edited the sched_rt.c file. This file contains the definitions related to real-time scheduling policies. Like we did with sched.c, we diffed this sched_rt.c with its corresponding file in the actual 3.0.4 kernel to give us an idea of what changes needed to be made.

Data structures and functions related to SCHED_RR task scheduling in sched_rt.c: task_tick_rt() and get_rr_interval_rt(). We modified both of these functions to implement our round-robin scheduling algorithm. get_rr_interval_rt() returns the timeslice for round-robin tasks.

Data structures and functions related to SCHED_FIFO task scheduling in sched_rt.c: task_tick_rt() and get_rr_interval_rt(). These functions were critical to the implementation of both SCHED_RR and SCHED_FIFO policies. We modified task_tick_rt() to basically not do anything if the task policy is set to SCHED_FIFO; this function was meant to act on SCHED_RR tasks. We modified get_rr_interval_rt() so that if the task policy is set to FIFO, it returns 0 because FIFO tasks do not have timeslices (they run until completion or they block and yield the CPU). A value of zero is interpreted by sched_rr_get_interval() in sched.c to mean an infinite timeslice.

The development process: We went through several iterations of modifying, compiling, and testing. We tested our system calls and as soon as we found one that was broken we determined why, referencing the full 3.0.4 kernel when necessary. We then started the cycle over again.

3. *How did you ensure your solution was correct? Testing details, for instance.*

Testing was one of the most challenging aspects of the project.

We determined that the best way to test round-robin scheduling would be to write a C program that set itself to a SCHED_RR process using sched_setscheduler(). This program would then fork off 4 child programs. Each of these programs would print their PID, time this print operation, and then busy-wait for the rest of their 100ms timeslice. They would cycle through this four times. If we had implemented our round-robin algorithm successfully, we knew that the processes should print out their respective PIDs one at a time, repeating. One caveat to this was that we had to set the CPU affinity. We also had to make sure that the processes did not call sleep() for the duration of their timeslice, as this would cause the scheduler to run the next task and interfere with our results. We used clock() with polling in a while loop to count up to 100ms, and added assembly code to prevent the compiler from potentially optimizing out this while loop. Initially, we had not correctly implemented the sched_setscheduler() system call, causing our program to exit with an error. Once we fixed this and recompiled the kernel, our program outputted the expected results.

To test FIFO scheduling, we leveraged the program we had used to test round-robin scheduling. This time, however, the program would set itself to SCHED_FIFO using sched_setscheduler() prior to forking off the child programs. The expected output on success would be that the first child would print its PID four times, then the next would print all four, and so on. Even though FIFO scheduling does not use timeslices, we left these intervals in because it would produce the same results and allowed us to reuse code. When we tested this, it also performed as expected.

Test code:

4. *What did you learn?*

In completing this project, I gained a much greater understanding for how Linux operates "under the hood" in kernel space. I learned about the performance trade-offs with scheduling tasks in FIFO vs round-robin. I learned about task prioritization. I learned that my perception of a computer multitasking and process concurrency is really just an illusion; processes are run sequentially in a prioritized fashion by the scheduler, but all of this happens so quickly that it appears as though many processes are running on each core of the CPU simultaneously. I learned the purpose of the files within the kernel directory tree. I learned how to use diff to make patches for the kernel. I learned how to configure and compile a new kernel. I learned how to boot into a newly-compiled kernel and debug it.