Brian Reber
CprE 308, Section D
Lab 05

Lab 05a: Process Scheduling

From my experimentation, I have found that the scheduler that produces the smallest average time from arrival to finish is Shortest Remaining time First. This is pretty logical, because we allow the processes with the shortest remaining time to finish first. If a process doesn't have much left, it makes sense to just let it finish. Round Robin with priority is often the next scheduler algorithm with the shortest average time from arrival to finish. This is a more fair scheduler, and it also gives processes higher priority if priority matters in a system (which it often does). First come, first served is often after Round Robin with priority, as it takes care of processes that arrived earlier first. Round robin without priority was often the one that completed processes the slowest. This is likely because it gives any process that is ready an equal share of time running.

See my source code for these algorithms in **scheduling.c**. The output when srand is seeded with 0xC0FFEE is provided in the submission as **output.txt**.

Lab 05b: Analysis of a Round Robin Scheduler

2.1 Preprocessor Directives

- The #define on line 26 defines a string representing the scheduler version. This is used when registering/unregistering the scheduler.
- We will include lines 44 and 45 if we are using kernel 2.4.06.
- The declaration on line 51 is a sched_policy. The declaration on line 50 is an array of sched_policy pointers. What I am thinking will happen is that line 51 is the actual instance of the sched_policy, and the declaration on line 50 will allow us to set/modify the sched_policy per CPU. So each CPU will be able to have its own sched_policy.

2.3 Initialization

- Line 158 sets the new variable to the address of round_robin. This means that any changes to "new" will modify the round_robin variable.
- Lines 161 and 162 set the function pointers sp_preemptability and sp_choose_task to the functions in this module that have the implementation.
- Lines 165-166 assign the policy to each element in the array that is used to define the scheduler per CPU (the array we talked about in the previous section).
- Lines 182-186 are the lines that handle the unregistering of the module.

2.4 Scheduling

- The function ctrr_choose_task is the function that gets called when the kernel asks for the next process to run. It is essentially the main logic of the scheduler. It chooses what process is to run next.
- Lines 67-70: If a process is running and its policy does not allow it to yield running time and it is not idling on the current CPU (the one this is running on), and this process still

has time left to run, we mark it as keep.

- The counter is replenished on line 93 (p->counter = NICE_TO_TICKS(p->nice);). This doesn't necessarily happen for each process each call. It happens for only the currently running process on the CPU.
- The idle_task is returned if we didn't choose to "keep" the process on lines 67-70 (as talked about above), and the current task is running and is able to yield running time (according to its policy), and the init_task is the same as the init_task.next_run.
- The purpose of the cttr_preemptability function is to check to see if the task is preemptable or not. To do this, this algorithm checks to see if the task trying to preempt the current process (the thief task) has more ticks left than the current task.

3 pset.c

- Goodness is used on the following lines:
    - Line 132: goodness(curr_task, this_cpu, curr_task->active_mm);
    - Line 142: goodness(p, this_cpu, curr_task->active_mm);

  The difference between these two uses is the first parameter. In the first instance, we are checking the goodness of the current running task. On line 142, we are checking the goodness for other tasks. This second one is in a loop, so it ends up checking the goodness of all available tasks.
- IDLE_WEIGHT is defined as -1000 (found in sched.h). It is likely set as this because we don't want to run a process that is waiting for something to finish. And as described in the lab description, a goodness value of -1000 means that it will never be selected.
- can_choose is defined on line 97. is_visible is defined on line 85.
- The initial value of high is IDLE_WEIGHT, and the initial value of choice is the return value of idle_task(this_cpu).
- choice is set to curr_task when the current task is running, and it is visible, and the goodness of the current task is not negative.
- The code on lines 138-148 chooses the task that is visible, and has the highest goodness value.
- If we have tasks that are runnable but have no ticks left, the value of high would be 0, therefore making !high be TRUE, causing the body of the if statement on line 151 to be executed.
- Line 158 goes to the label recheck, which is on line 126.
- I don't think this section should run twice in a row. What it does is replenish the counters, therefore making the counters not equal zero anymore. Since they aren't zero, the high value should be able to be set, therefore not going into this section again. If all tasks are in the IDLE_WEIGHT category, this section will run twice in a row, and the algorithm will choose the current task.
- If we were to remove lines 130-135, we wouldn't necessarily get the same result. Since on line 143, we just check if the task's goodness is greater than high, we will choose the task that comes first in the list. So if the current task has the same goodness as another task in the list, and that other task is closer to the front of the list, we will choose that one over the current task, which isn't what the algorithm does now.
- We want to include this check because we want to choose the current task if it has the highest goodness (even if that goodness is the same as another task in the list). The

reason we want to choose the current task over the other task with the same goodness is because if costs a lot to switch processes. So if we have two processes with the same goodness and one of them is already running, we should keep running it so that we don’t incur the cost of switching processes.