CS423: MP2

Brad Misik and Ivan Hu

September 30, 2011

Abstract

Implementation, development, and challenges of writing a rate monotonic scheduler for Linux.

1 Introduction

Although we read through and implemented the MP2 spec step-by-step, there were quite a few places where our implementation deviated. Those deviations, as well as a brief overview of the project and the important observations we had, will be outlined in this document.

1.1 Environment Setup

I first mirrored the recommended virtual machine setup locally, because the remote machines were painfully slow to work on. Then, I downloaded the solution to MP1 and used it as a base for MP2. I created a git repo of the root folder and uploaded it to share with the team.

1.2 Initial Development

The majority of MP2 was just refactoring the solution from MP1. The struct and variable names were modified, some functions were deleted, but the majority of both the structure, variables in use, and functions remained in the code.

1.3 Miscellaneous

Our test application has a period of 500ms and a computation time of 10ms. It prints out the time difference between every yield, which averages out to near 500ms. We also included a test.sh script which attempts to run and register one more task than is schedulable according to the schedulability test (35). 34 * (10 / 500) = 0.68, 35 * (10 / 500) = 0.7

2 Implementation Deviations

2.1 Timing

Since this is a rate monotonic scheduler, every task timer signals a deadline for when a job must be completed. When that timer goes off, it is intuitively reset, because we are scheduling periodic tasks. That intuition is why our solution deviates from the spec. Instead of calculating the correct deadline time based on a current time and the last firing time during a call to yield(), we reset the task timer in the timer interrupt handler with the task's period. This task period is retrieved from the argument to the interrupt handler, which is set in the timer struct's data member. We do not believe this extra code violates the two halves approach, because the interrupt is still overall very short and resetting a timer here is not unreasonable.

2.2 Task Selection

Instead of selecting only Ready tasks, we select Ready or Running tasks to be run next, because there are instances where our dispatching thread is requested to run for rescheduling, and one can imagine that the currently running task may have a shorter period than other Ready tasks, in which case it makes sense for it to continue running.

2.3 First Scheduling

For each newly registered task, we add it to our list and assign it the Registering state. Upon its first yield, its state is set to Ready, the task is put to sleep, and only then is its first timer set.

3 Challenges

3.1 Mutual Exclusion

The obvious problem in this exercise is guaranteeing mutual exclusion between our two threads (and the interrupt context of the timer handlers). Initially, we used one lock to isolate the critical sections of our list data structure that included: adding to the list in our main kernel thread, removing from the list in our main kernel thread, and reading through the list in our dispatch thread. Since the dispatch thread doesn't insert and remove from the list, this is all that is necessary to avoid collisions. However, these are not the only problems we can incur with regards to mutual exclusion. We also added a lock around part of our yield and deregister functions, with the same lock wrapping most of the context switch of our dispatch thread, and here is why: if we deregister a task that is the current task, we may end up freeing its memory before the dispatch thread attempts to read or modify it, resulting in a use-after-free. Similarly,

when yielding, we want to ensure that we set the state of our task to Sleeping and that it doesn't get clobbered in the middle of a context switch.