Senior Project Final Report: Implementation and Theory of Real-Time Scheduling

Jacob Earle
New Haven, CT

Abstract

In the study of operating systems, real-time scheduling is the field concerned with scheduling algorithms for tasks that have a deadline before which the task must be completed in order for the order of scheduled tasks to be considered ”correct.” For my senior project for the 2021 fall semester, I have focused on gaining insights into both the theoretical and practical aspects of real-time scheduling. As a part of these ongoing efforts, I have researched theoretical results in the field, written a tool in R that implements feasibility tests on theoretical real-time task sets, and worked on implementing a real-time scheduling algorithm in Theseus, an experimental operating system written entirely in Rust and meant to push the boundaries of traditional operating systems design by leveraging intralingual design patterns. [1]

Operating Systems, Scheduling

Introduction

In the study of operating systems, real-time scheduling is the field concerned with scheduling algorithms for tasks that have a deadline before which the task must be completed in order for the order of scheduled tasks to be considered ”correct.”

I decided to undertake my senior project on this topic because I first encountered the concept of real-time scheduling through my research in the field of operating systems at the Yale Efficient Computing Lab and was intrigued to learn more about the theory and applications behind the topic. I believe that this project has been worthwhile to pursue because real-time scheduling algorithms allow for greater reliability for time-critical applications and provide a method for theoretically verifying that a set of tasks can be scheduled to execute successfully in a timely manner. Especially in embedded systems, where computing resources are meager but the cost of failure can be catastrophic, a real-time scheduler can enable developers to create efficient embedded applications with confidence. Thus, through my work this semester, I have gained much applicable knowledge that I can use to contribute to further work in the field, as well as begun a contribution to an open source project that I and others can continue developing.

In this paper, I will provide a report on the work that I have completed this semester, discussing the deliverables, the challenges I faced along the way, and the ways this project could be expanded in the future.

Project Overview

I have undertaken this project to fulfill my senior requirement according to the guidelines set out in the syllabus for CPSC 490. My advisor for this project was Professor Lin Zhong, and my research was conducted as part of a project at the Yale Efficient Computing Lab. Since I am currently pursuing a joint major in Mathematics and Computer Science, I will have to make a presentation on the mathematical aspects of my project to members of the Yale mathematics faculty.

In order to provide a more thorough picture of the goals behind my project, I will briefly discuss the deliverables associated with the project and the skills and knowledge I hoped to gain through this project.

Goals

As a Math and Computer Science major, I thought it would be beneficial to not only gain the practical skills necessary to work on a real-time operating system. I also wanted to gain insight into the theory behind the most common algorithms and tests used in real-time scheduling. This way, I will be able to better appreciate new theoretical results in the field and have the potential to someday contribute to the knowledge of the domain as well. Thus, I have made sure to include deliverables that tackle both the practical and theoretical/mathematical aspects in the field.

Deliverables

There are three main deliverables associated with this project:

1. **Implementation of a real-time Scheduling Algorithm:** A contribution to the open-source Theseus repository including a real-time scheduler for the ARM port of the operating system, as well as a system for developers of applications on top of Theseus to define periodic tasks with specified priorities and deadlines. As part of this development, I have also had to implement a sleep API to allow for the implementation of periodic tasks. This scheduling algorithm is built around the existing design patterns of the Theseus operating system and can be swapped in for any other scheduling algorithm implemented for the OS.

   This portion of the project serves as the main piece of work for the practical side of my project. The pull request containing the changes I have written as part of this delivery is located here: `https://github.com/theseus-os/Theseus/pull/467`.

2. **Command Line Tool for Analyzing feasibility of real-time Task Sets:** In order to put the theoretical knowledge I gained while researching the field of real-time scheduling, I decided to write a command line tool in the R language that can read in a description of a theoretical real-time task set and perform various tests that I have learned about to determine whether the task set is "feasible" under different scheduling algorithms, i.e. whether the tasks can be scheduled such that none of them will miss their deadlines. Using various arguments, the program allows for various parameters of the analysis to be altered, such as the number of cores on the system and whether the tasks have explicit or implied deadlines.

   The project can be found on Github at this link:
   `https://github.com/jacob-earle/RealtimeDeadlineAnalysis`.

3. **Presentation:** As explained earlier, I am pursuing a joint major in Mathematics and Computer science, so I am required to present an approximately 15 minute presentation on the mathematical aspects of my project. In this case, my presentation will pertain particularly to explaining the proofs and analysis of the algorithms my code relies on and the feasibility tests that I have researched.

Milestones

In order to stay on track for my project, I had weekly progress meetings to check in with my advisor and with the leader of the Theseus operating system project that I was contributing too. In order to meet the recommended work requirement for the class, I spent an average of approximately one hour per day working on my project, which would equate it to the workload of a typical Yale class. The following milestones summarize how I spent my time working on this project.

1. **Week 1:** Read and understood the most important works on real-time scheduling, such as those of Liu and Layland [3], to decide upon an algorithm to implement.

2. **Week 2:** Researched the implementation of periodic tasks in other embedded real-time operating systems, such as FreeRTOS.

3. **Weeks 3 - 5:** Implemented a simple real-time scheduling algorithm for Theseus. This involved implementing a system to allow tasks to sleep and until they are ready to complete some work at fixed periodic intervals. Additionally, implemented crates in Rust to implement the Rate Monotonic Scheduling (RMS) algorithm and its associated runqueue in Theseus.

4. **Weeks 6 - 9:** Tested and debugged implementation of simple scheduling algorithm. Devised system to let application developers specify custom periods for tasks and investigated better data structures/design patterns to help the code fit better into the open source project as a whole. Submitted a pull request to get these changes integrated into the codebase.

5. **Weeks 10 - 11:** Began theoretical portion of the project by researching feasibility tests in single-core, multi-core, and soft real-time systems. Wrote R command line tool to implement some of these feasibility tests.

6. **Weeks 11 - 12:** Finalized code with proper documentation and bug fixes. Created presentation for math faculty and wrote final report for the project.

Changes to Original Milestones

It is important to note that some of the milestones that I actually ended up completing above vary slightly from the original milestones I described in my project proposal at the beginning of the semester.

First, one small change that occurred was that some of the challenges I faced in implementing the simple real-time scheduling algorithm – which I will discuss when I go into detail on this part of the project – caused me to spend about a week more than I intended working on this segment of my project. Still, this is to be expected as things will not also go as smoothly as intended, and I believe the challenges have helped me learn more about proper design patterns and important concepts in operating systems.

The second, more major change that I have made from my original milestones is that my original milestones did not include the research on theoretical feasibility tests or the command line tool in R. Originally, I had planned to implement a more complicated scheduling algorithm on top of the RMS algorithm I wrote for Theseus, but I instead opted to focus that time on the theoretical side of real-time scheduling. I found this prudent because, as a Mathematics and Computer Science major, my project had to have a significant mathematical component, and thus I was worried just implementing a preexisting algorithm would not provide me with a sufficient amount of mathematical knowledge.

Overall, I do not feel that these changes have in any way lessened what I have gained from this project, and in fact I believe I am happy to learn even more about the theoretical side of real-time scheduling than I would have otherwise.

Reflections on Implementation of Scheduling Algorithm

Theseus is an experimental operating system written entirely in Rust and meant to push the boundaries of traditional operating systems design by utilizing features of the Rust language to offload tasks often handled by the operating system onto the compiler. Although originally built for x86-based computers, recent efforts

have been made to port Theseus to highly embedded systems such as ARM Cortex-M microcontrollers. As a part of these ongoing efforts, I wanted to contribute to further development on the project and apply my knowledge of real-time scheduling algorithms by working on implementing real-time scheduling for the ARM port of Theseus.[1] This section of the paper will describe the general design decisions I made and challenges I faced in the process, as well as including suggestions for future work that I could do on the project.

Design Decisions

The first design decision that I had to make was choosing which scheduling algorithm to implement. Within the field of real-time scheduling, there are two main subcategories of scheduling algorithm. The first of these subcategories is referred to as ”hard” real-time scheduling, in which all tasks must be completed before their deadlines with no tolerance for a late task. If any of the tasks cannot be scheduled to finish before its associated deadline, then the ordering of the scheduled tasks is considered to have failed. The second subcategory is referred to as ”soft” real-time scheduling, in which an ordering of tasks can result in some tasks completing after their deadlines. However, each task has a certain penalty associated with lateness, and as long as the total penalty across all tasks in the ordering is below a certain threshold, the ordering is considered to be correct. However, if the total penalty is above the threshold, then the ordering is considered to have failed. [3]

Whereas soft real-time scheduling algorithms often require complex statistical analyses to validate, many of the most common hard real-time algorithms are fairly simple to analyze and implement. Thus, I decided that it would be best to begin with a hard real-time scheduling, as a thorough exploration and implementation of soft real-time scheduling is outside the scope of a single semester-long undergraduate project. However, in my reflections on my theoretical/mathematical research that will be found at the end of this paper, I will include a brief description of some of the theory pertaining to soft real-time scheduling that I learned.

In the end, I chose to implement a version of the Rate Monotonic Scheduling algorithm (RMS) devised by Liu and Layland in their seminal work on real-time scheduling algorithms. The RMS algorithm is an example of a ”fixed priority” scheduling algorithm, in which all tasks are assumed to be periodical and the jobs associated with those tasks are assigned the same fixed priority when they are released. In the case of RMS, the priorities assigned to tasks are inversely proportional to their periods, i.e. the tasks with the shortest periods are assigned the highest priorities, and vice-versa. [2] I chose to implement this algorithm because, as a fixed-priority algorithm, it does not have much overhead associated with it, which was beneficial for the low-memory embedded devices that I was working on. In order to account for the fact that some tasks may not be periodic on a system, my chosen algorithm allows for ”background execution” of non-periodic tasks. This basically means that non-periodic tasks are simply given priorities lower than all of the periodic tasks, thus ensuring that no non-periodic tasks would be allowed CPU resources until all of the jobs of the higher-priority periodic tasks had finished running.

Aside from the need to choose which scheduling algorithm to use, I had to make many design decisions relating to how I would add the new features needed to implement a simple real-time scheduling algorithm into the Theseus operating system while conforming the project’s philosophy of combining Rust’s safety-conscious language features with independent, abstracted interfaces for different operating system functions. The main design concerns that I had to contend with include:

1. **How to support basic periodic tasks**
   Critical to the implement of any real-time operating system is the ability to support periodic tasks, which release jobs at fixed intervals that must be completed before a certain deadline relative to the job’s release. In the case of RMS, we assume that each job’s deadline is equal to the task’s period.

Thus, in order to support real-time scheduling, it was necessary to implement some kind of API that can be used by developers of the operating system to create periodic tasks. One simple conception that can be used to implement such periodic tasks is providing developers with a function that can be called at the end of the main task loop in order to put the task to sleep whenever it has finished an iteration of the loop (which is equivalent to a job released by the task completing) and set it to wake at fixed intervals of time. The inspiration for this design decision is Free-RTOS, a popular embedded real-time operating system written in C, which supports the creation of periodic tasks through the `vtaskdelayuntil` API (for more info on `vtaskdelayuntil`, you can find the documentation here).

In order to implement such an API for creating periodic tasks in Theseus, I realized that the desired behavior could be implemented as an extension of the common `sleep` API, which was not yet implemented in Theseus, which would allow the task to repeatedly call the `sleep` function to delay the task until the desired fixed interval is reached. However, as the `sleep` API had not yet been implemented in Theseus, I needed to implement that as well, in the process accomplishing both the goal of supporting periodic tasks and putting tasks to sleep for any period of time in general. This was the first main piece of code that I had to write for the code, and in order to do this, I chose to keep a list of all of the currently delayed tasks as well as the times (measured in system ticks) at which the tasks would need to be woken up and removed from the list. This list was implemented as a priority queue with the values being references to the delayed tasks and the keys being the times at which the tasks would need to wake. This way, the list would always be sorted in increasing order of wake-up time. Thus, every time a system tick was issued, the Rust crate responsible for managing the list of delayed tasks would look at the earliest wake-up time in the list and remove any tasks that had a wake time equivalent to the current time in ticks.

For the `sleep` API, I wrote 3 different functions that could be used to delay tasks: `sleep`, which takes a value in ticks and causes the task to delay for that number of ticks, `sleep_until`, which takes an argument representing an absolute moment of time and causes the task to delay until that time is reached, and finally, `sleep_periodic`, which achieves the desired behavior of allowing tasks to be put to sleep and woken up repeatedly at fixed intervals. This completed a basic implementation of periodic tasks.

2. **How to keep track of time for delayed tasks**
   Central to the implementation of any real-time system is a sensible representation of time elapsed. As mentioned in the section on implementing periodic tasks, tasks that are stored in the delayed task list are not meant to be woken up until a certain moment in the future. However, without the system somehow keeping track of time elapsed, these tasks would never be removed from the list, as the operating system would have no idea when the wake time had been reached. Thus, I needed to implement some way for the system to keep track of time. For an implementation with a single list of delayed task, I realized that it was sufficient to keep a count of the number of elapsed ticks since the system was initialized. I achieved this by keeping this counter in the Rust crate associated with the `sleep` API and adding a callback to the SysTick handler function that would alert the crate to update the counter. Now, whenever the counter is updated, the crate managing the delayed task list can check if any delayed tasks have reached their wake time, and if they have, remove those tasks from the list and wake them. Keeping track of time using a count of elapsed SysTicks requires some overhead, and thus future research could involve figuring out how to create individual timers that can wake a task without having to consult a list of delayed tasks.

3. **Implementing the logic and runqueue data structure associated with the RMS scheduling algorithm**
   Fortunately, the Theseus operating system provides abstractions for runqueues and scheduling algorithms that allow a new scheduling algorithm to be implemented fairly easy by conforming to a standard interface. This abstraction has already been used before to implement various scheduling algorithms like round-robin and priority scheduling. Thus, once I had figured out a way to support periodic tasks, the task of implementing the RMS algorithm into Theseus was fairly straightforward, as I just had to write Rust crates that implemented the logic for how RMS chooses the next task to be scheduled and an appropriate runqueue data structure.

   To begin with, I had to implement the runqueue data structure to conform to the standard runqueue interface. At its heart, my runqueue implementation consists of a list of entries, where each entry consists of a reference to a task on the system alongside some information useful for scheduling the tasks. In this case, the additional information we needed to store with each task reference was information about the periodicity of the task. Using Rust's `Option` data type, I was able to store this information in a single field of each entry, where a periodic task with a period of p would store a value of `Some(p)` in this field, and a non-periodic task would store a value of `None` in this field. Using this additional information, we can keep the entries in the runqueue sorted in increasing order of the period value of the tasks. We achieve this sorting by inserting a new entry for a periodic task after all the tasks with lower periods and before all the tasks with a higher period; this achieves the desired behavior of tasks with a lower period to have higher priority, as they will be closer to the front of the runqueue and thus selected sooner by the scheduler. For a non-periodic task, the entry associated with the task will be inserted at the end of the list and reinserted at the end of the list any time it is selected in round-robin fashion; this achieves "background execution" by allowing non-periodic tasks to be selected for scheduling only if no periodic task is ready to run. Additionally, when tasks are spawned, they are created as non-periodic tasks, but the runqueue data structure provides methods for setting the periodicity of a task. Although the runqueue data structure is currently implemented as a linked list that allows insertion at any index and is manually kept sorted, it may be beneficial in the future to implement the runqueue with a data structure that allows for task entries to be inserted quickly at the proper sorted location, such as a priority queue. However, for relatively small task sets, keeping the list sorted through iterating to find the proper index for a new entry does not incur a large runtime penalty.

   Using this runqueue implementation, the logic used to select the next task to run is fairly straightforward. The scheduler simply loops through all the entries in the runqueue until it finds the first runnable task. Since the tasks are sorted in order of increasing period, with non-periodic tasks being placed at the end, we achieve the priority of task selection demanded by the RMS algorithm, as tasks with lower periods will be given higher priority than tasks with higher periods and will be able to preempt those tasks. Every system tick, the scheduler selects a new task to run, and will idle if no runnable tasks are found. Thus, through the use of an appropriate runqueue data structure and task selection logic, we have achieved the desired behavior of the RMS algorithm, including tasks with fixed priorities assigned in increasing order of the tasks' periods and background execution for non-periodic tasks.

Putting these designs together, we can provide an example of how a new periodic task that prints something every 2 seconds can be created and added to the scheduler in Theseus (in an example where a tick occurs every 10 milliseconds):

```
fn task_delay_two_seconds(arg: usize) {
    let start_time : AtomicUsize = AtomicUsize::new(
        sleep::get_current_time_in_ticks()
    );
    loop {
        info!("I run every two seconds!");

        // Since we trigger a Tick every 10ms,
        // 2 seconds will be 200 ticks
        sleep::sleep_periodic(&start_time, 200);
    }
}

...

// build and spawn a real time periodic task
// we will start it as blocked
// in order to set the period before it runs
let periodic_tb = new_task_builder(task_delay_two_seconds, 2).block();
let periodic_task = periodic_tb.spawn().unwrap();

// setting the period of the task
// Since we trigger a Tick every 10 ms,
// the 2 second period represents 200 ticks
scheduler::set_periodicity(&periodic_task, 200).unwrap();

// starting the task
periodic_task.unblock();
```

Challenges Faced

Like any software development project, my implementation of a real-time scheduler in Theseus was not without challenges. However, I believe that these challenges have taught me a great deal about good code structure and helpful design paradigms. The biggest problem that I faced when first writing my code is that I had trouble isolating the behavior associated with each of the features to a specific Rust crate. In fact, at some point I had code for various features scattered throughout separate crates, for example placing the behavior for setting a task's periodicity in the crate responsible for scheduling tasks, instead of in the crate containing the runqueue, which is ultimately the crate that needs to use this information. Another example is that I had logic related to delaying tasks directly inside of the SysTick interrupt handler function instead of simply using callback functions that implement this logic inside of the crate responsible for the task delay/sleep API. Thankfully, I was able to overcome this major hurdle through a significant refactoring of my code that the leader of the Theseus project recommended in order to isolate behaviors only to the crates that need them. I believe that this is an important lesson for my future coding projects as I have learned that proper isolation makes the system more robust, debuggable, and easier to abstract.

Another challenge I faced was in learning how to properly share resources across Rust crates, specifically in the case of sharing task references properly between the runqueue, scheduler, and delayed task list. In

the process, I learned more about how resources are shared in operating systems and the synchronization primitives that make safe sharing possible, such as atomic reference counters and mutexes.

Potential for Future Research

As I have alluded to when I discussed the design decisions I faced in implementing the real-time scheduling algorithm, there are several potential ways to improve the project further. For example, I could investigate more efficient methods for supporting delayed tasks, such as using individual timers that will independently unblock a delayed task, removing the need to check a list of delayed tasks every time a system tick occurs. This will reduce system overhead and increase the amount of CPU time that tasks receive. Another potential development is finding a more efficient data structure for the runqueue, such as a priority queue, that allows for efficient sorting of the list of tasks in order of increasing priority. Another potential avenue to explore is to implement a more complex scheduling algorithm such as the Earliest Deadline First (EDF) algorithm, which dynamically assigns the highest priorities to the jobs that have the earliest deadlines. [2] In general though, I believe that my work this semester on implementing RMS in Theseus is a good start and has provided me with a lot of knowledge, not only on real-time scheduling algorithms, but also on good operating system design principles.

Reflections on Theoretical/Mathematical Research

As mentioned earlier, since I am a Mathematics and Computer Science major, my senior project had to have a significant mathematical component. In order to fulfill this requirement, I thought it would be interesting to research the topic of tests for feasibility of real-time task sets. In my research, I learned about the theory behind tests that can be used to analyze task sets in a number of contexts, such as in single- and multicore systems, implicit or explicit deadlines, and certain soft real-time task models. However, for the sake of brevity, instead of describing each of these tests here, on the web page for my senior project submission, I have attached a slide deck for a presentation that I gave to the math department describing the results I have learned. Anyone interested in learning about these results is encouraged to view that slide deck. Consequently, in this section, I will briefly focus on reflecting on the implementation of the R script I wrote to allow users to carry out these tests on theoretical task sets.

Design Decisions

The most important design decision that I had to make in implementing the analysis tool was choosing which language to use. In order to choose the best language for the task, I had to consider the simplicity of writing a command line script, the mathematical primitives available, and the ability to represent and work with data representing a theoretical task set. In the end I chose R because it fulfills all of these criteria adeptly. Writing an R script to run as a command line tool is straightforward and intuitive, and it has packages available that can easily parse command line arguments. Likewise, R has functions to perform all kinds of mathematical and statistical calculations quickly without the need for any outside dependencies, and the functional nature of the language makes it easy to program any mathematical functionalities not already provided. Finally, R allows data to be easily read from various sources and manipulated.

The next major design consideration I had to make was how to represent a task set within the program and how to input that representation into the script. Fortunately, R's first-class support for columned data sets allowed me to represent a task set as a data set, with each row representing a task and with columns representing for example a task's id, period, compute time, or deadline. The task of inputting these data sets

into the program was simplified by R's ability to generate data sets from a CSV file. Thus, a description the task set can be written into a CSV file, and the name of the file can be passed as a command line argument.

Potential for Future Research

One obvious way to expand upon the work I have done in this domain is to research more feasibility tests for real-time task sets and implement them into the command line tool. For example, I could add tests for analyzing a task set under soft deadline constraints or when aperiodic tasks are present. Another potential avenue for future research is adding the ability to simulate the scheduling of a task set under various scheduling algorithms in order to provide certain statistical information such as the frequency of tasks missing their deadlines or the CPU utilization rate achieved.

Conclusion

The work that I have done for my senior project this semester is far from exhaustive, and there is still much that I can learn about the topic of real-time scheduling. However, I believe that my work has provided me with a solid foundation to learn even more, both from a practical and theoretical standpoint. Thus, I have very much enjoyed working on my senior project and hope that the reader has also gained some appreciation for the topic as well.

Acknowledgments

This project builds on the prior work of Kevin Boos, Zhiyao Ma, Namitha Liyanage, Ramla Ijaz, and all the past contributors to Theseus. I would like to thank Prof. Lin Zhong and Kevin Boos for providing mentorship on this project.

References

[1] K. Boos, N. Liyanage, R. Ijaz, and L. Zhong. Theseus: an experiment in operating system structure and state management. *OSDI 2020*, pages 1–19, 2020.

[2] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. 1973.

[3] L. Sha, T. Abdelzaher, K.-E. Arzen, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. Mok. Real time scheduling theory: A historical perspective. *Real-Time Systems*, (28):101–155, 2004.