Designing a Thread Scheduler for a Prototype OS

Derek Ziemba

Brady Sullivan

CSCE351 SEC 001

11/15/2015

# Introduction

This document is prepared for the second CSCE351 project – designing a thread scheduler for a PrototypeOS. The goal of this project is to is to create, schedule, maintain, terminate and monitor thread statistics for several threads running on a NIOS II cpu. In this document we will cover the timeline, build tools and working environment, summary of tasks, design details, and improvements.

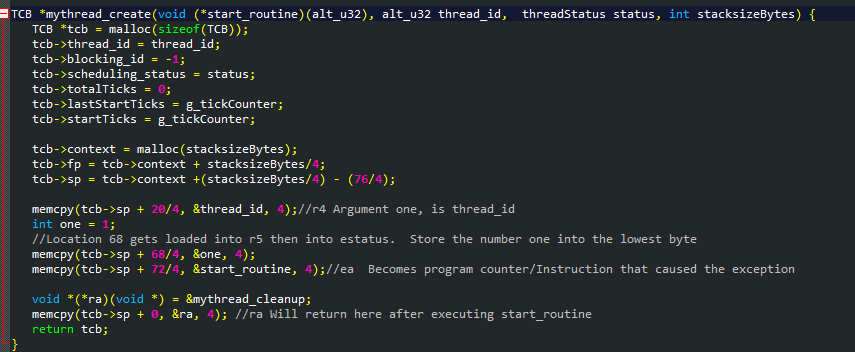
* For our project we started working on our project on the 9th due to conflicting schedules but this left us little time for error in our implementation of the prototype OS. This was the largest risk going into the project and one we traced the line of carefully. We were also worried about the scope of the project but we soon realized that we had done quite a bit of the heavy lifting in our implementation of the second homework and project 1.
* To accomplish this project we used Visual Studio with GCC for its IntelliSense, superior debugging, and error checking to create and implement the basics of this project. Once it was working in Visual Studio it was ported over to the NIOS II Eclipse environment for the main coding portion of the project. Otherwise, to keep up to date with each other, we used Dropbox and GitHub.
* Our main source for resources to get through issues we faced on the project was google. This lead to various stack overflow links and a few git repositories that helped which I will link at the end of this report.
* Working on this project with Derek has been great and I have learned a lot from him. Coding together down in the lab and preparing our report has been seamless. Our work ethic is good and both members are willing to put in the time and effort to get this project done on time and of great quality.

# Summary of Tasks

## Thread Control Block

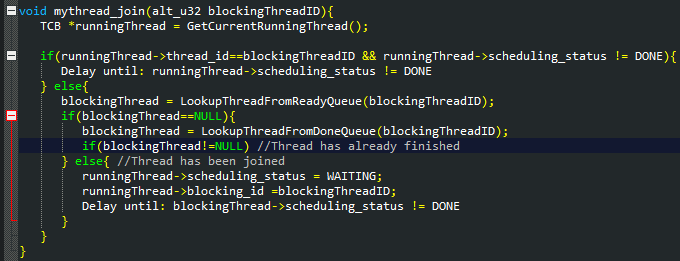
Implementing the TCB was fairly easy given the initial variables in the project description. A few additional fields such as the time the thread was: created, last run, and total runtime were added for extra credit. It is for this reason that the TCB is never destroyed and added to a DONE queue. This is discussed in greater detail later. Properly creating the TCB was the most difficult challenge of the project. We had an issue were we could not get threads to properly clean up after themselves and execute properly, the issue was eventually traced back to improperly setting the stackpointer and the address of the cleanup function in the TCB.

Below is a code excerpt from the create thread function. The TCB is initialized to 32-bit unsigned values because this is what the native register size is on the NIOS CPU. It contains everything from the context, stack pointer, frame pointer, etc. Create is where these values are actually set when we make the thread. Malloc is used to assign the memory to the context and associated pointers. You may notice that memcpy is used to place the values into the specific location instead of using array pointers. This is due to our limited understanding of array pointers in C. We had tried to use array pointers, but couldn’t get the threads to enter the mythread\_cleanup function. We also had this issue using memcpy, but were able to fix it using the weird void \*(\*ra)(void \*) cast.



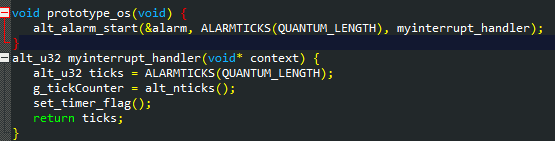
## Thread Join

Our creation of the threads is mostly described in the prior section with our thread control block structure. This is due to the nature of the two and how they work so closely together in the program. Since the TCB is the meat and bones of every thread it’s easy to show the implementation of the TCB and create simultaneously. Join, however is slightly different. The implementation, shown below, is just what you would expect: blocks the calling thread until it terminates while continuing to perform the other threads actions. Our Join is an attempt at this synchronization method that blocks the running thread (that is, the thread that calls the method) until the thread that was Joined has completed its course. The code below is also shortened and in a semi-psuedo form so if you would like to see the various prints in between or have any other questions, look at the source.

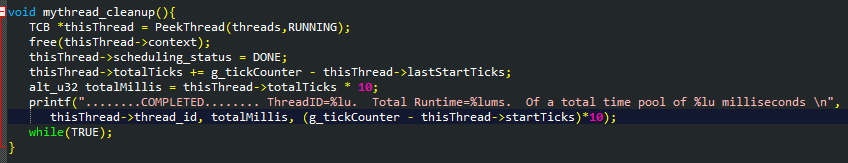


## Prototype\_OS

The Prototype\_OS is executing its main loop, every 1000ms myinterrupt\_handler() is called. If the main program loop gets interrupted in the middle of sending its message, the interrupt will send its message right in the middle of the main loop’s message. This illustrates how interrupts must be used with caution. To prevent this from happening interrupts can be disabled by setting a flag. But disabling interrupts in an interrupt driven system could cause “hickups” and lag. For instance when a keyboard keypress isn’t registered by a PC. Another option is to have the interrupt evaluate the context it is given and to decide its behavior based on that context. For instance, in this case the interrupt knows the value of that static global variable MAX and the inner busy loop index through the context. Based on these two variables it can decide if it would be wise to also send a message. If sending a message at that instant would cause corruption, it can return a lower AlarmTick count and be called again soon. Having the interrupt evaluate the context is just one option, but ultimately interrupts are slow due to having to push registers on the stack so it would have been better for interrupts to have been disabled.



## Thread Cleanup

Normally thread cleanup is where you would destroy the TCB be we don’t actually do this. Instead of destroying the TCB, we place the thread it into a “Done” queue so that we can keep track of stats. This allowed us to show the print line for the extra credit on this project. Regardless, cleanup of the threads context runs normally. First, it peeks at the running thread and then deallocates the memory associated with the thread. Then we place it into the “done” queue and gather the total time that it ran for stat tracking. This runs while true then continues the next thread. Final note, the other reason we kept the done queue was so that we could output a final summary of all the threads and the usage statistics.

# Design

Several design decisions were made in the name of readability, extensibility, and future reusability. We wanted to be able to use this project as a basis for future projects. The P2PrototypeOS.h file contains many variables that you can use to easily tweak different aspects of the PrototypeOS.

## Generic Linked List

A generic Linked List was created and verified for correctness using Visual Studio unit tests. The purpose of this linked list is to provide the foundation for the ThreadQueue. A linked list was chosen so that any number of threads could be suspended to a list and we were not restricted by the constraints of an array. However, there was a downsides to having a generic linked list. Using the NIOS II debugger we were not able to inspect the TCB contained in the nodes. For this reason we gave in and decided to make it non-generic for this project.

## Thread Queue

 Four different thread queues, index-able by thread status were used. The possible thread statuses include: ready, running, waiting, and done queue. It was the responsibility of the thread\_scheduler to maintain the placement of threads in these queues based on the TCB’s scheduled\_status field.

* Ready Queue - Contains the threads that have been recently created that are currently not being blocked by a thread\_join.
* Running Queue – Is where threads that are currently running are placed. Since the NIOS II is single core processor, this will only ever contain a single thread.
* Waiting Queue – Is where the main thread is placed when thread\_join is called.  In this implementation, it will only ever contain the main thread.
* Done Queue – Contains the TCB’s of threads that have already been cleaned up by having their context freed.  They are placed here in order to keep track of their execution time and times started/stopped so we can obtain the extra credit.

## Thread Scheduling

### Assembly

In order to schedule a new thread we inject code into exception.exit.user. When an exception occurs such as a timer interrupt, this code checks if it was indeed a timer interrupt that triggered the exception by checking a timer interrupt flag. If the exception is a timer interrupt this code:

* Prevents any other interrupts from occurring by setting the status bit in the control register to 0.
* Moves the stack pointer and frame pointer into registers r4 and r5 respectively so that they may be passed into the thread scheduler.
* Calls thread\_scheduler. The thread scheduler returns the new stack and frame pointer in registers r2 and r3 respectively.
* The assembly then updated the stack pointer and frame pointer to the values stored in r2 and r3.
* Interrupts are re-enabled.

### Thread\_scheduler

The thread scheduler is designed to only be called by assembly that already manages that status of interrupts. The thread scheduler starts by retrieving the stack and frame pointers. After which it checks if a TCB has already been created to track the statistics of the main thread. If one has not been created, a custom one is generated that does not have a context and is then queued to the RUNNING queue. The code following this is oblivious and treats all TCBs the same, hence why the thread was set in the RUNNING queue. The remaining code:

* Pulls the first thread from the RUNNING queue then recalculates its statistics, sets the TCBs’ stack and frame pointers, then enqueues the thread in the proper thread queue based on the threads scheduling\_status that has possibly been set to WAITING by thread\_join or DONE by thread\_cleanup.
* Checks if there are any threads that are ready or waiting, in that order, then dequeues one, sets its status to RUNNING, then enqueues it in the running queue before returning the stack and frame pointers of that thread.

# Conclusion

Overall, we learned quite abit about creating a protoype operating system. Learning how to control the memory and making sure that the system was being efficient was quite the undertaking. Managing memory in general, is a good skill to learn when working in the future on new projects. The problems faced in this project were difficult when it came down figuring out the small details. For example, earlier we mentioned our improper use of the stackpointer. After debugging however, our prototype os came together. We look forward to the next project and final addition to the prototype os.

# References

|  |  |
| --- | --- |
| [1] | "Nios II Software Developer's Handbook," [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en\_US/pdfs/literature/hb/nios2/n2sw\_nii52017.pdf. |
| [2] | Altera, "Application Binary Interface," 2 April 2015. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en\_US/pdfs/literature/hb/nios2/n2cpu\_nii51016.pdf. |
| [3] | Altera, "Nios II Core Implementation Details," [Online]. Available: https://www.altera.com.cn/zh\_CN/pdfs/literature/hb/nios2/n2cpu\_nii5v1\_02.pdf. |
| [4] | C. Crockrom and S. Fanning, "Camtendo/oskernelsproject," 21 Feb 2013. [Online]. Available: https://github.com/Camtendo/oskernelsproject. [Accessed 10 Nov 2015]. |