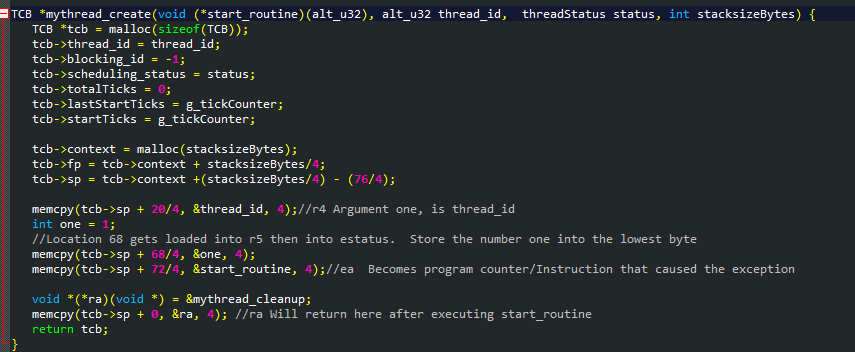
Project 2, CSCE 351

11/15/2015

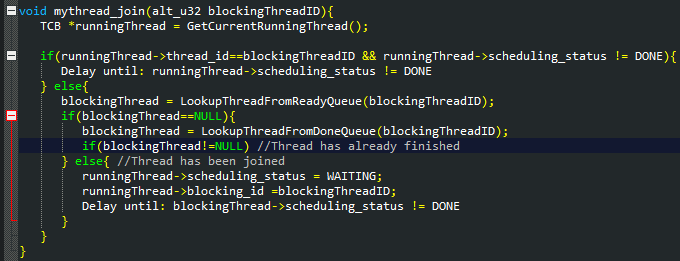
Derek Ziemba

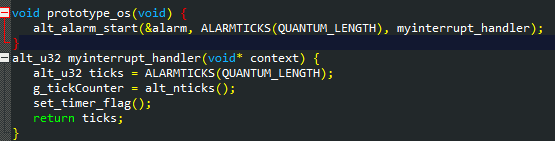
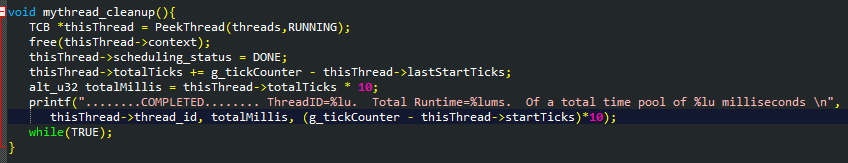
Brady Sullivan

1. Introduction
   1. For the second project in this course, we were instructed to design and implement a thread scheduling subsystem that would allow our prototype OS to create, schedule, maintain, and terminate multiple threads on our Altera board. The main issue with this being that the board we are given only supports single-thread execution. We also provide basic execution information and statistics via standard out.
   2. 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.
   3. To accomplish this project we used Visual Studio with GCC for debugging and 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 IntelliSense.
   4. 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.
   5. 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 good quality.
2. Summary of our tasks
   1. Implementing the TCB was fairly easy given the initial variables in the project description while adding a few for the tick counter. It wasn’t too hard to create but we found joining and cleaning up can be a bit tricky. Regardless, below you will find the code from our create thread function. According to the tasks, we initialized the tcb values to 32-bit unsigned values. 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. We implemented memcpy and used it to put the values into the specific location due to poor pointer arithmetic and our limited knowledge of C. We had many troubles at this spot due to reversing our stack pointer. Eventually we realized we needed to put it 76 bytes down to avoid the r2 and r4 registers.



* 1. 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.



* 1. While 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.
  2. 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 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.

1. Criticism of Implemented methods
   1. Linked List: used to suspend threads and stored here.  This was chosen so that there was no limit to how many threads could be in a queue since a linked list doesn't need to be pre-allocated like an array.

* 1. Thread Queue:
     1. Create 4 different queues containing threads that are: ready, running, waiting, and done
        1. The ready queue contains the threads that have been recently created that are not blocked by mythread\_join
        2. Running is a queue consisting of all the threads that are currently running. Since the NIOS II is single core processor, this will only ever contain a single thread.
        3. Waiting queue is where the main thread is placed when mythread\_join is called.  In this implementation, it will only ever contain the main thread
        4. The done queue contains the 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.

* 1. Thread scheduling assembly code:
     1. The assembly passes the stack pointer and frame pointer to the scheduler in registers r4 and r5 respectively.
     2. The scheduler then passes the next thread’s stack pointer and frame pointer back to the assembly in registers r2 and r3 respectively.  The assembler then moves these values into the sp and fp before re-enabling interrupts.

References:

1. "Application Binary Interface." *Altera*. Altera Corporation, 2 Apr. 2015. Web. 16 Nov. 2015. <https://www.altera.com/content/dam/altera-www/global/en\_US/pdfs/literature/hb/nios2/n2cpu\_nii51016.pdf>.
2. "Camtendo/oskernelsproject." *GitHub*. Ed. Cameron Crockrom and Sarah Fanning. 21 Feb. 2013. Web. 10 Nov. 2015. <https://github.com/Camtendo/oskernelsproject>.
3. "Nios II Software Developer's Handbook," [Online]. Available:

<https://www.altera.com/content/dam/alterawww/>

1. "Nios II Core Implementation Details," [Online]. Available:

<https://www.altera.com.cn/zh_CN/pdfs/literature/hb/nios2/n2cpu_nii5v1_02.pdf>