Lab 1 - Execution Control
CEC 450 Spring 2020, 02/18/20
Cameron Stark (CS - starkc1@my.erau.edu)
Kevin Dumitrescu (KD - dumitrek@my.erau.edu)

Section 1: Effort

Planning and Preparation:
CS - 2 h 15 m
KD - 2 h 15 m
Experiment:
2.5 h
Report Writing:
2 h

Section 2: Objectives

This week lab experiments focused on task stacks as well as task controls and delays. The objective of this lab was figuring out how to control the state of a task, specify task parameters, obtain information about tasks running on the system, and identify whether a task has crashed or not.

Section 3: Procedures and Results

Part A: Task Stack

During the course of lab 1, the entirety of the lab was done using the VXWorks Simulator. To begin working on the lab, a new project needed to be created. By selecting the file button, going down to "new" create downloadable kernel project, we were able to then add the given lab files *recur.c* and *recurTwo.c* and then build the project (A1).

Once the project was built and we had the code up, we then proceeded to test function **recur()**. By calling **recur()** with a small value, we initially noticed that the value for **count** would not display, so in order to get the value of **count** to display and find out what it is, we would either need to change the function to were it won't be void, so that **count** can be returned, or call the **count** variable to have the **count** value be displayed (A2).

With the ability of now being able to see the value of **count**, we then went to execute the shell syntax **tid = repeat(0,repeat,6)**, which is a shell syntax that would cause our task to go on forever with a level of 6 and asked to monitor the values of **count**. In order to monitor the value of **count** as the task would run loop forever, we had the function return the **count** at the end of each repeated execution (A3).

After receiving the values of **count** as we had **recur()** loop forever, we then proceeded to examine the task using **ti(tid)**, where we received information such as task name: t1, entry point: repeatRun, identifier: 1223edb0, priority: 100, and stack size: 65536 (A4).

Once we obtained the information of the task ID, we then went and executed the **i()** command four times to see if the stack pointer changes. After running the **i()** command, we noticed that the stack pointer of the task does change very slightly, oscillating incrementally and decrementally. We presume that this oscillating incrementation and decrementation is due to other parts of the program needing more stack space at different times causing the task pointer to shift in the stack (A5).

After observing the changes in the stack pointer, we then traced the stack frames using the shell command **tt()** to first watch the progress of the program and then use **checkStack(tid)** to examine the stack usage. Upon examination of the stack, we found that the total size of the stack was 65536, the used size was 80, the largest used was 292, and the margin was 65244. This showcases that the stack size did not change from it's default, that very little of the stack size was used, and the largest did not take much stack space (A6).

When the stack was examined and we obtained our values, we then deleted the task started at A3 using the shell syntax **td(taskID)**, where the taskID was the ID from A3. After deleting that task, we then went to confirm if the

task was truly deleted by using the ti(taskID) command with the taskID being the one deleted. If we received a "task not found" error message, then we knew that we had successfully deleted the task (A7).

Part B: Task Controls and Delays
During the second portion of the lab, the second built and compiled with the two functions **reucr1()** and **recur2** was loaded. To begin a task was created that would call **recur1()** forever by using the function **repeat(0, recur1, 150)** with 150 being used as a large enough value to check enough times with overflowing the memory, with us checking the value of count1, but entering the variable into the terminal window three times (B1)

Next was to suspend task 1, check the value of count1 and resume task 1. This can be done with the commands **taskSuspend(taskId)** and **taskResume(taskId)** (B2)

Task 2 was created with **repeat(0, recur2, 150)** that would run the function recur2 for an extended period of time allowing the observation of the value of count2 and count1, during the execution to perceive how the simulation handles the two running tasks. (B3)

To understand the priority of the tasks on the CPU, the priority level of task1 is reduced using **taskPrioritySet(taskId,priorityValue)** which changes the priority of the associated task, which as a result causes the task2 to get more access to the CPU because of its now higher priority (B4)

After suspending task2 and checking the values of count1 and count2 it is found that the values of count1 count upwards more often now that it is now the only task on queue and it has the highest priority on thak queue. After resuming task2 and returning task1 to the same priority, the tasks begin to alternate more frequently. (B5)

In the attempt to have both tasks share the CPU equally with the need of priority scheduling or value queue, the implementation of round-robin can be used, which is a system of time-slices to divide access to the CPU in specific times. The time slice is set using **kernelTimeSlice(timeSlice)** which starts the process with the round-robin scheduling (B6)

After reverting the priority back to normal and removing the time slicing. Delaying the shell can be done with the function **taskDelay(time)**, which causes the tasks to be paused while the time remains on the task delay. If a value for delay time of 6000 is inputted the shell and program will pause for a very long time and potentially crash, to fix this without doing a reset, is pushing ctrl + c (B7)

After suspending task 1 and task 2, the task creates a delay with a time value of 100000. To check the information for this task, the command **ti(taskId**) is used which displays all information in a table format. After running this command several times, we can see that the value is going down showing that the delay is counting down. (B9)

Section 4: Observation, Comments, and Lessons Learned

Observed that the documentation for vxWorks is rather lacking, for specifically the terminal functions and in the event that a problem does occur with the program, board or simulated board so for future labs we can ask for more help specifically on the parts that currently being worked on.

We will begin writing the paper much sooner than last time.

Appendix

Answers to questions:

A1: To create the project, file > new create downloadable kernel project. To add the files right click on the project file new, name file with .c as the file extension. Once the file code has been written, right click on the project and build the project.

A2: Change the function not a void so that the count can be returned, or call the count variable to have the count value displayed

A3: Have the function return the count at the end of each repeated execution

A4: The values for task 1 are as follows:
Task Name: t1
Entry Point: repeatRun
Identifier: 1223edb0
Priority: 100
Stack Size: 65536

A5: After running the i() command the SP of T1, the SP changes very smaller oscillating by incrementing and decrementing, because other parts of the program might need more stack space at different times causing the task pointer to shift in the stack.

A6: Total Size: 65536
Used Size: 80
Largest Used: 292
Margin: 65244

A7: td(taskId) -> to delete inputted task, To check if task is successfully deleted run the ti(taskId) command which will return with task not found if deleted successfully

B1: repeat(0, recur1, 150) -> leads to infinitely running until the condition is met in the function

B2: taskSuspend(id) -> suspends the inputted task and prevents the values inside from changing

B3: The same shell command is used as above to start tid2, and to resume tid1 taskResume is used to resume the task. Because a non preemptive system is being used task1 is going to continue running because it is a lower number, but same priority

B4: Tid2 is now running and count2 is now incrementing because of the new difference in the priority, it is now running in preemptive scheduling with tid2 interrupting tid1

B5: After suspending tid2, tid1 starts running so count1 starts incrementing, then after resume and the priorities are back to the same value, the scheduling is returned to non-preemptive.

B6: To implement a round-robin style scheduling system, a time slice needs to be added, with the command kernelTimeSlice(value) with a value of 10 ticks. The counts are now incrementing in turns with both incrementing because both are given time on the cpu in equal amounts

B7: taskDelay just delays the use of the shell by blocking the access to the cpu and thereby preventing the tid1 and tid2 from running. Means of recovering from an accidental running of this is ctrl + c in the shell will close and restart the shell or a disconnection of the remote connection

B9: To get the information on the tid use the ti(id), repeat this command to view the delay counter going down

Source for functions:
https://www.ee.ryerson.ca/~courses/ee8205/Data-Sheets/Tornado-VxWorks/vxworks/ref/taskLib.html#taskPrioritySet