PS03: Multi-Threaded Robot Programming in C

Due: March 26, 2014

Robot systems need to sense, process, and act on information in real-time, with strict timing guarantees. For this assignment, you will experiment with the multi-threaded kernel on the rone robots, and then build four fun behaviors in C: Avoid obstacles, wall-follow, orbit, and tree navigation.

1 Setup

We have created a VirtualBox image to save time and effort installing the C development environment on your computers. We will use Git for version control. You must learn some basics of Git to get started, but you do not need to become an expert.

1. Install VirtualBox - `https://www.virtualbox.org/wiki/Downloads`
2. Download the VirtualBox Appliance - `https://www.clear.rice.edu/comp551/resources/Ubuntu-14.04-COMP551.ova`
3. Import VirtualBox Image onto your computer
   Open VirtualBox; File->Import Appliance->Select File Location->Select Settings->Import
4. Fork the rice-comp551 Git repository - `https://github.com/mrsl/comp551`
5. Clone the Git repository onto your VirtualBox machine

You are now setup and can program the robots from Eclipse within the virtual box.

1.1 JTAG Programmer

The JTAG programer is used to flash your C programs on to the r-one. It also allows you to debug your programs by stepping through your program's execution and set breakpoints in your code.

Listing 1: jtag_test.c

```
#include "roneos.h"
#include "ronelib.h"

int main (void)
{
      systemInit();
      systemPrintStartup();

      ledsSetPattern(LED_ALL, LED_PATTERN_CIRCLE, LED_BRIGHTNESS_MED, LED_RATE_MED);

      osTaskStartScheduler();
      //Program should never get here
      return 0;
}
```

2 Tasks, Mutexes, and Messages

Tasks (Threads) are separate execution instances. They each have their own stack, but they share the general memory with each other. One reason why you would have multiple tasks running on an embedded system is to separate sensor processing, motor control, and other system tasks from your program. For example, the IR communication system tracks the robot's neighbors. It operates as a background task that updates a neighbor data structure periodically. Your program will read from this neighbor data structure to learn about neighboring robots. The writes to this neighbor data dnd the reads from this neighbor data must be *mutually exclusive*, or *mutex*, with each other, to ensure that the shared data does not get corrupted. Please review the FreeRTOS user guide in the docs directory of the Git repo for a good discussion of tasks and mutexes, we will present a brief summary here.

A mutex is an operating system object that allows multiple tasks to share a common resource. In our example, your program and the background neighbor task would like to use the neighbor data structure. Your task needs to lock the mutex before accessing the neighbor data structure to prevent conflicts with the other task. If the mutex is taken by another task, then your task would enter a blocked state and will wait until the mutex becomes available again. A task releases the mutex after utilizing the data structure so other tasks can continue executing.

A couple problems with using mutexes for concurrent programming is deadlock and priority inversion. Deadlock occurs when two tasks are waiting for some resource held by the other task and so cannot make progress. Priority inversion occurs when a higher priority task is delayed by some lower priority task. For example, suppose a low-priority task takes a mutex to perform some computation. Then, along comes a high-priority task that also wants to use the same mutex. Since the mutex is already in-use, the high-priority task enters a blocked state. During the low-priority task's execution, a medium-priority task preempts the low-priority task. Now, the low-priority task is no longer executing and the high-priority task is blocked, waiting for the low priority task to release the mutex. The medium-priority task is controlling the execution flow when the high-priority task is supposed to have the highest priority. A solution to priority inversion is priority inheritance where the low-priority task is raised to priority of the highest priority task waiting for a mutex. Thus, preventing another task from interrupting the execution of the low-priority task.

Listing 2: led.c

```
// Testing the colored LEDs.
void buttonColors(void* parameters)
{
      uint32 lastWakeTime = osTaskGetTickCount();
      uint8 buttonRedOld = 0;
      uint8 buttonGreenOld = 0;
      uint8 buttonBlueOld = 0;
      uint8 buttonRed, buttonGreen, buttonBlue;

      while(TRUE)
      {
            buttonRed = buttonsGet(BUTTON_RED);
            buttonGreen = buttonsGet(BUTTON_GREEN);
            buttonBlue = buttonsGet(BUTTON_BLUE);
            if (buttonRed & !buttonRedOld)
            {
```

```
                ledsSetPattern(LED_RED, LED_PATTERN_CIRCLE, LED_BRIGHTNESS_MED,
                    LED_RATE_MED);
            }
            else if (buttonGreen & !buttonGreenOld)
            {
                ledsSetPattern(LED_GREEN, LED_PATTERN_CIRCLE, LED_BRIGHTNESS_MED,
                    LED_RATE_MED);
            }
            else if (buttonBlue & !buttonBlueOld)
            {
                ledsSetPattern(LED_BLUE, LED_PATTERN_CIRCLE, LED_BRIGHTNESS_MED,
                    LED_RATE_MED);
            }
            buttonRedOld = buttonRed;
            buttonGreenOld = buttonGreen;
            buttonBlueOld = buttonBlue;

            osTaskDelayUntil(&lastWakeTime, BEHAVIOR_TASK_PERIOD*10);
        }
}

int main (void)
{
        systemInit();
        systemPrintStartup();

        behaviorSystemInit(buttonColors, 4096);

        osTaskStartScheduler();
        //Program should never get here
        return 0;
}
```

The git repository contains the FreeRTOS Documentation. The password for the pdf document is found the text document in the folder.

Make a two-thread program based on `led.c`. Each thread should print a unique string 10 times. Something like, "Hippopotamus" and "Platypus". Do not use any yield or delay functions from the FreeRTOS API. Use `cprintf()` to print over the serial port. Use `osTaskCreate()` to create the tasks in order to set the task priority. For the other tasks, use `behaviorSystemInit()` which is the default function in the behavior system api. Run ./rcs/build/RCSRIO from the terminal command line to read the print statements from the serial port.

1. Make thread 1 a higher priority than thread 2. Capture the output and hand in.
2. Make the two threads the same priority. Capture the output and hand in.
3. Make a new function, `serial_send_string_mutex()`. Use a mutex to ensure that only one thread can print at a time. Capture the output and hand in.

Make a three-thread program and a message queue. Thread 1 and 2 should put 10 total messages on the queue, one every 0.5 second. The messages should be pointers to the strings from above, and use a different string for each thread. Thread 3 should read the queue and print the message. You will need to read about how to implement periodic threads in the FreeRTOS book.

1. Make thread 1, 2, and 3 the same priority. Capture the output and hand in.

3 Obstacle Detection and Wall Following

Listing 3: basic_motion.c

```
#include "roneos.h"
#include "ronelib.h"

void behaviorTask(void* parameters) {
      uint32 lastWakeTime = osTaskGetTickCount();

      Beh behOutput;
      while (TRUE)
      {
            /* Initialize the output behavior to inactive */
            behOutput = behInactive;

            behSetTv(&behOutput, 100);
            motorSetBeh(&behOutput);
            osTaskDelayUntil(&lastWakeTime, BEHAVIOR_TASK_PERIOD*10);
      }
}

int main (void)
{
      systemInit();
      systemPrintStartup();

      behaviorSystemInit(behaviorTask, 4096);

      osTaskStartScheduler();
      //Program should never get here
      return 0;
}
```

Make a new program based on `basic_motion.c`. Use a background thread to read the obstacle detector with the `irObstaclesGetRangeBits()` function.

1. Make an `obstacleAngleCompute()` function that takes the obstacle bits and computes the direction of the obstacle. Refer to the `process_nbr_message()` function for inspiration on computing direction from bits. Note that you will potentially need to deal with obstacles on many different sides of the robot.
2. Make a `avoidObstacles()` function that takes the obstacle angle and steers the robot away from obstacles. Put this function into a program to make the robot wander around the environment.
3. Make a `followWall()` function that takes the obstacle angle and drives the robot along a wall. Print the turning angle around corners to the console.

3.1 API hints

Here are some of the functions you will need to implement this section. Full details are in the rone API web pages.

Behavior System

1. struct Beh - roneLib/src/Behaviors/behaviorSystem.h - the values that move the robot
2. motorSetBeh(behOutputPtr) - Move robot based on beh struct
3. irObstaclesGetBits - roneOS/src/irComms/neighbors.h - the directions the IR sensors detect obstacles
4. irObstaclesGetRangeBits - roneOS/src/irComms/neighbors.h - the estimated distance to the obstacles - IR range bits

4 Orbit

Make a new program that creates an orbit task. Select a leader out of the neighboring robots using the robot id. The leader stays in-place while the other robots rotate around the leader. The orbiting robot should follow a circle centered at the leader with the distance from the robot to the leader as the radius.

4.1 API hints

Here are some of the functions you will need to implement this section. Full details are in the rone API web pages.

Neighbor System - roneos/src/IRComms/neighbors.h

1. NbrList - Array of neighbor data
2. nbrListCreate(nbrListPtr)
3. nbrListClear(nbrListPtr)
4. nbrListPrint(nbrListPtr, string name)
5. nbrListGetSize(nbrListPtr)
6. nbrListGetNbr(nbrListPtr, index)
7. nbrListFindSource(nbrListPtr, broadcastMsgPtr) - find the neighbor that is the source of the broadcast
8. nbrListPrintHops(nbrListPtr, broadcastMsgPtr, string name) - print neighbors and hop count

Broadcast Communication - roneLib/src/NeighborListOps/BroadcastComms.c

1. BroadcastMessage struct
2. broadcastMsgCreate(broadcastMsgPtr, MAX_HOPS) - Create broadcast message
3. broadcastMsgIsSource - Check if this robot is the source of the broadcast message
4. broadcastMsgUpdateLeaderElection - Select leader by changing broadcast message to the robot with the lowest id and hop count
5. broadcastMsgUpdate - Update broadcast message; Allows for multiple sources but no leader election

6. broadcastMsgUpdateNbrData - Update local neighbor data with the incoming data from the sender of broadcast message

5 Tree Navigation

Create a self-stabilizing tree that allows the leaf nodes to navigate to the root node.

5.1 Write self_stabilizing_tree function

Select the robot with the lowest id as the root node. The remaining robots should select the neighbor with the smallest number of hops to the root as their parent. Break ties between each robot by choosing the robot with the lowest id.

Turn on all R,G,B leds for the root node. Use the circle red led pattern when the robot does not have any neighbors. Set the count on each color of leds with the function:
`ledsSetPattern(color, LED_PATTERN_COUNT, brightness, count)`

Where count ranges from 0-5. This will show colors on one led, and clear the rest. You can indicate more than 5 hops with different colors:

1. Red - 1-5 hop
2. Green - 6-10 hops
3. Blue - 11-15 hops

5.2 Add tree_navigation functionality

Select a robot in the tree using a button push. The robot will then use the tree to navigate to the root node. The robots should update their hop count and parent nodes, allowing the tree to self-stabilize.

5.3 Special Case

Consider if you select a robot in the tree that is not a leaf node. You could potentially separate some of the robots from the rest of the tree because they are no longer connected by the intermediate parent node. Implement some basic changes to your algorithm to improve its resilience to this scenario and summarize your design choices.

API Reference

1. ledsSetPattern(color, pattern, brightness, flash_rate) - roneos/src/InputOutput/leds.h
2. ledsClear(ledColor)
3. buttonsGet(buttonID) - roneos/src/InputOutput/buttons.h

6 Hand-In / Check-Off

1. **Hand-in:** Your traces from Section 2.
2. **Check-off:** Your `avoidObstacles()` and `followWall()` function in operation.
3. **Check-off:** Your `orbit()` function in operation

4. **Check-off:** Your `self_stabilizing_tree()` functions in operation
5. **Hand-in:** A summary of how your implementation handles the tree navigation special case