ECEE 5623, Real-Time Systems:

Exercise #3 – Threading/Tasking and Real-Time Synchronization

DUE: As Indicated on Canvas

Please thoroughly read Chapters 3 & 4 in RTECS with Linux and RTOS

Please see example code provided - Linux, FreeRTOS, and VxWorks.

This lab is written **to be completed with embedded Linux running on the R-Pi3b or 4b (order or borrow and install R-Pi Linux). It is possible to also use a Jetson Nano (Getting started), however this is recommended for 5763 and not required for 5623**. The Raspberry Pi is preferred (RTES-Home-Lab-Set-Up-Notes.pdf). **The standard final project requires use of a camera, which is simple with embedded Linux using the UVC driver**.

Linux start code and example code can be found here - Linux/code/

Exercise #3 Goals and Objectives:

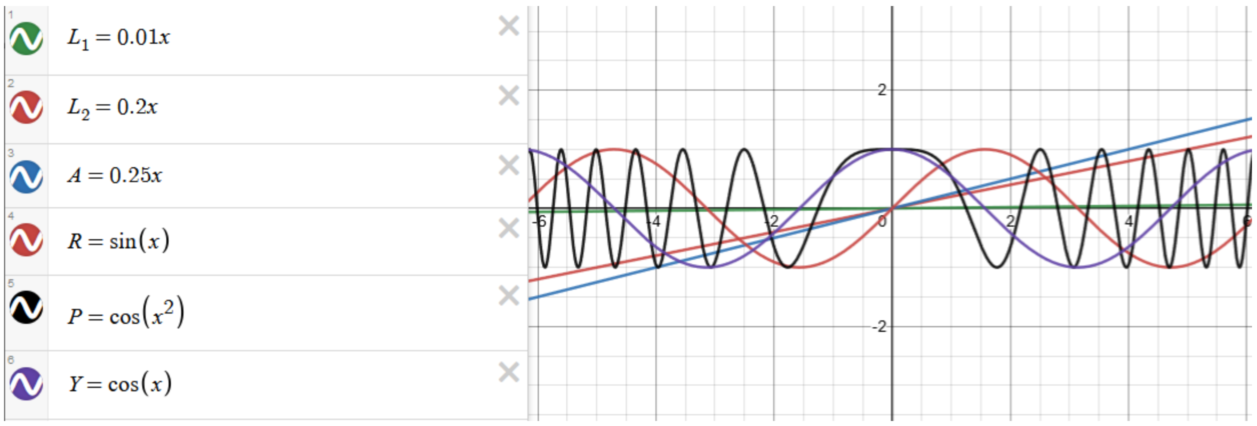
1) [15 points] Read Sha, Rajkumar, et al paper, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization".
   a) Summarize 3 main key points the paper makes. Read my summary paper on the topic as well, which might be easier to understand.
   b) Read the historical positions of Linux Torvalds as described by Jonathan Corbet and Ingo Molnar and Thomas Gleixner on this topic as well. Note that more recently Red Hat has added support for priority ceiling emulation and priority inheritance and has a great overview on use of these POSIX real-time extension features here – general real-time overview. The key systems calls are pthread_mutexattr_setprotocol, pthread_mutexattr_getprotocol as well as pthread_mutex_setpriorityceiling and pthread_mutex_getpriorityceiling. Why did some of the Linux community resist addition of priority ceiling and/or priority inheritance until more recently (Linux has been distributed since the early 1990's and the problem and solutions were known before this).
   c) Given that there are two solutions to unbounded priority inversion, priority ceiling emulation and priority inheritance, based upon your reading on support for both by POSIX real-time extensions, which do you think is best to use and why?

2) [30 points] Review the terminology guide (glossary in the textbook).
   a) Describe clearly what it means to write "thread safe" functions that are "re-entrant". There are generally three main ways to do this: 1) pure functions that use only stack and have no global memory, 2) functions which use thread indexed global data, and 3)

functions which use shared memory global data, but synchronize access to it using a MUTEX semaphore critical section wrapper.

b) Describe each method and how you would code it and how it would impact real-time threads/tasks.

c) Now, using a MUTEX, provide an example using RT-Linux Pthreads that does a thread safe update of a complex state (3 or more numbers – e.g., Latitude, Longitude and Altitude of a location) with a timestamp (pthread_mutex_lock). Your code should include two threads and one should update a timespec structure contained in a structure that includes a double precision position and attitude state of {Lat, Long, Altitude and Roll, Pitch, Yaw at Sample_Time} and the other should read it and never disagree on the values as function of time. You can just make up values for the navigational state using math library function generators (e.g., use simple periodic functions for Roll, Pitch, Yaw sin(x), cos(x^2), and cos(x), where x=time and linear functions for Lat, Long, Alt) and see http://linux.die.net/man/3/clock_gettime for how to add a precision timestamp. The second thread should read the times-stamped state without the possibility of data corruption (partial update of one of the 6 floating point values). There should be no disagreement between the functions and the state reader for any point in time. Run this for 180 seconds with a 1 Hz update rate and a 0.1 Hz read rate. Make sure the 18 values read are correct.


$L_1 = 0.01x$
$L_2 = 0.2x$
$A = 0.25x$
$R = \sin(x)$
$P = \cos(x^2)$
$Y = \cos(x)$

E.g., Function Generators - https://www.desmos.com/calculator

3) [20 points] Download example-sync-updated-2/ and review, build and run it.

a) Describe both the issues of deadlock and unbounded priority inversion and the root cause for both in the example code.

b) Fix the deadlock so that it does not occur by using a random back-off scheme to resolve. For the unbounded inversion, is there a real fix in Linux – if not, why not?

c) What about a patch for the Linux kernel? For example, Linux Kernel.org recommends the RT_PREEMPT Patch, also discussed by the Linux Foundation Realtime Start and this blog, but would this really help?

   d) Read about the patch and describe why think it would or would not help with unbounded priority inversion. Based on inversion, does it make sense to simply switch to an RTOS and not use Linux at all for both HRT and SRT services?

4) [15 points] Review POSIX-examples and POSIX_MQ_LOOP and build the code related to POSIX message queues and run them to learn about basic use.
   a) First, re-write the simple message queue demonstration code so that it uses RT-Linux Pthreads (FIFO) instead of SCHED_OTHER, and then write a brief paragraph describing how the use of a heap message queue and a simple message queue for applications are similar and how they are different.
   b) Message queues are often suggested as a better way to avoid MUTEX priority inversion. Would you agree that this really circumvents the problem of unbounded inversion? If so why, if not, why not?

5) [20 points] For this problem, consider the Linux manual page on the watchdog daemon - https://linux.die.net/man/8/watchdog [you can read more about Linux watchdog timers, timeouts and timer services in this overview of the Linux Watchdog Daemon].
   a) Describe how it might be used if software caused an indefinite deadlock.
   b) Next, to explore timeouts, use your code from #2 and create a thread that waits on a MUTEX semaphore for up to 10 seconds and then un-blocks and prints out "No new data available at <time>" and then loops back to wait for a data update again. Use a variant of the pthread_mutex_lock called pthread_mutex_timedlock to solve this programming problem.

Overall, provide a well-documented professional report of your findings, output, and tests so that it is easy for a colleague (or instructor) to understand what you have done. Include any C/C++ source code you write (or modify) and Makefiles needed to build your code and make sure your code is well commented, and documented. I will look at your report first, so it must be well written and clearly address each problem providing clear and concise responses to receive credit.

Note: Linux manual pages can be found for all system calls (e.g. fork()) on the web at http://linux.die.net/man/ - e.g. http://linux.die.net/man/2/fork

In this class, you'll be expected to consult the Linux manual pages and to do some reading and research on your own, so practice this in this first lab and try to answer as many of your own questions as possible, but do come to office hours and ask for help if you get stuck.

Upload all code and your report completed using MS Word or as a PDF to Canvas and include all source code (ideally example output should be integrated into the report directly, but if not, clearly label in the report and by filename if test and example output is not pasted directly into the report). ***Your code must include a Makefile so I can build your solution on an embedded Linux system (R-Pi 3b+ or Jetson). Please zip or tar.gz your solution with your first and last***

name embedded in the directory name and/or provide a GitHub public or private repository link. Note that I may ask you or SA graders may ask you to walk-through and explain your code. Any code that you present as your own that is "re-used" and not cited with the original source is plagiarism. So, be sure to cite code you did not author and be sure you can explain it in good detail if you do re-use, you must provide a proper citation and prove that you understand the code you are using.

Grading Rubric

[15 points] Priority inheritance paper review and unbounded priority inversion issues:

Problem	Points Possible	Score	Comments
Three Priority inheritance key points made in paper	5		
Why the Linux position makes sense or not historically	5		
Position on which solution is best for use with Linux	5		
TOTAL	15		

[30 points] Thread safety with global data and issues for real-time services:

Problem	Points Possible	Score	Comments
Description of methods	10		
Impact of each to real-time services	10		
Correct shared state update code	10		
TOTAL	30		

[20 points] Example synchronization code using POSIX threads:

Problem	Points Possible	Score	Comments
Demonstration and description of deadlock with threads	5		
Demonstration and description of priority inversion with threads	5		
Description of RT_PREEMPT_PATCH and assessment of	5		

whether Linux can be made real-time safe			
RT PREEMPT patch would or would not help with unbounded priority inversion position	5		
TOTAL	20		

[15 points] Example synchronization code using POSIX message queues and threads:

Problem	Points Possible	Score	Comments
Demonstration of message queues on Jetson adapted from examples	10		
Description of how message queues would or would not solve issues associated with global memory sharing	5		
TOTAL	15		

[20 points] Watchdog timers (for the system) and timeouts (for API calls):

Problem	Points Possible	Score	Comments
Description of how the Linux WD timer can help with recovery from a total loss of software sanity (E.g., system deadlock)	10		
Adaptation of code from #2 MUTEX sharing to handle timeouts for shared state	10		
TOTAL	20		

Report file MUST be separate from the ZIP file with code and other supporting materials.

<u>Rubric for Scoring for scale 0…10</u>

Score	**Description of reporting and code quality**
0	No answer, no work done
1	Attempted and some work provided, incomplete, does not build, no Makefile
2	Attempted and partial work provided, but unclear, Makefile, but builds and runs with errors
3	Attempted and some work provided, but unclear, build warnings, runs with no apparent error, but not correct or does not terminate
4	Attempted and more work provided, but unclear, build warnings, runs with no apparent error, but not correct or does not terminate
5	Attempted and most work provided, but unclear, build warnings, runs with no apparent error, but not correct or does not terminate
6	Complete answer, but does not answer question well and code build and run has warnings and does not provide expected results
7	Complete, mostly correct, average answer to questions, with code that builds and runs with average code quality and overall answer clarity
8	Good, easy to understand and clear answer to questions, with easy-to-read code that builds and runs with no warnings (or errors), completes without error, and provides a credible result
9	Great, easy to understand and insightful answer to questions, with easy-to-read code that builds and runs cleanly, completes without error, and provides an excellent result
10	Most complete and correct - best answer and code given in the current class