ECEN5623, Real-Time Embedded Systems:

Exercise #1 – Invariant LCM Schedules

DUE: As Indicated on Canvas and in class

Please thoroughly read Chapters 1 & 2 in the text

Please see example code provided on Canvas

This lab is written to be completed with embedded Linux running on the Jetson Nano, R-Pi3b+ or 4b, or other approved development kit

Note: You must run any code that uses SCHED_FIFO, real-time priority, using "sudo"

Exercise #1 Requirements:

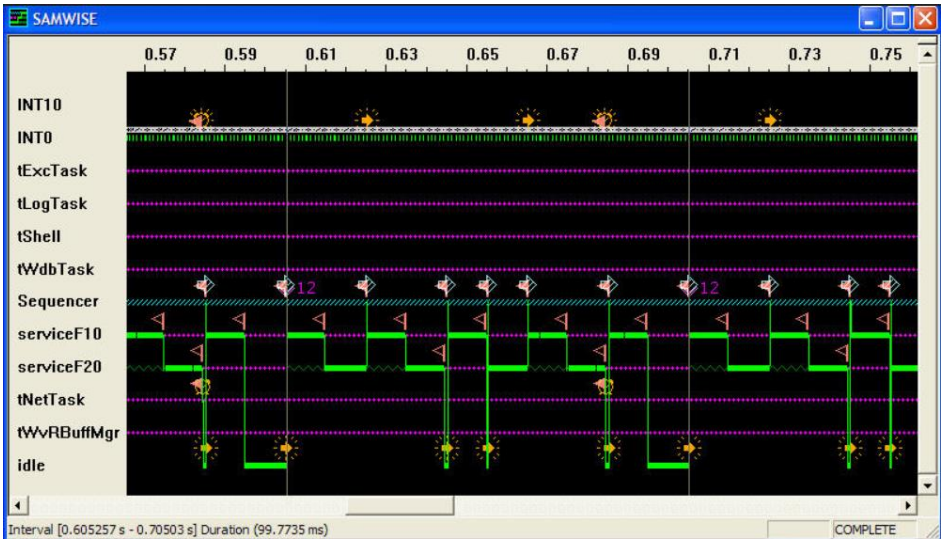
1) [20 points] The Rate Monotonic Policy states that services which share a CPU core should multiplex it (with context switches that preempt and dispatch tasks) based on priority, where highest priority is assigned to the most frequently requested service and lowest priority is assigned to the least frequently requested AND total shared CPU core utilization must preserve some margin (not be fully utilized or overloaded).
   a) Draw a timing diagram for three services S_1, S_2, and S_3 with T_1=3, C_1=1, T_2=5, C_2=2, T_3=15, C_3=3 where all times are in milliseconds. [Note that you can find examples of timing diagrams in Lecture and here and in Canvas – note that we have not yet covered dynamic priorities, just RM fixed policy described here, so ignore EDF and LLF for now].
   b) Label your diagram carefully and describe whether you think the schedule is feasible (mathematically repeatable as an invariant indefinitely) and safe (unlikely to ever miss a deadline).
   c) What is the total CPU utilization by the three services?

2) [20 points] Read through the Apollo 11 Lunar lander computer overload story as reported in RTECS Notes, based on this NASA account, and the descriptions of the 1201/1202 events described by chief software engineer Margaret Hamilton as recounted by Dylan Matthews. Summarize the story.
   a) What was the root cause of the overload and why did it violate Rate Monotonic policy?
   b) Now, read Liu and Layland's paper which describes Rate Monotonic policy and the Least Upper Bound – they derive an equation which advises margin of approximately 30% of the total CPU as the number of services sharing a single CPU core increases.
   c) Plot this Least Upper bound as a function of number of services.

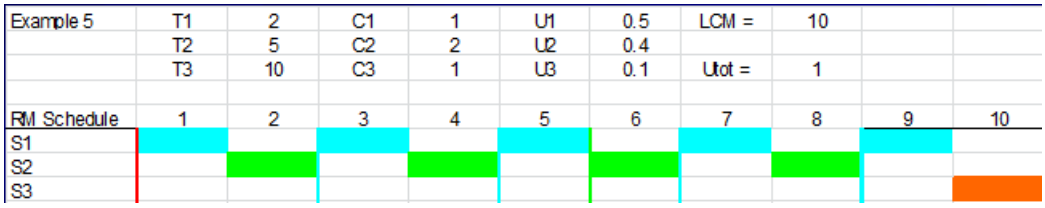
d) Describe 3 key assumptions they make and document 3 or more aspects of their fixed priority LUB derivation that you don't understand.
e) Would RM analysis have prevented the Apollo 11 1201/1202 errors and potential mission abort? Why or why not?

3) [20 points] Download the RT-Clock code from http://mercury.pr.erau.edu/~siewerts/cec450/code/RT-Clock/ or from Canvas (Modules ->Exercises->Exercise 1->Code) and build it on a Jetson board, Raspberry Pi, or Altera DE1-SOC board and execute the code.
   a) Describe what the code is doing and make sure you understand clock_gettime and how to use it to time code execution (print or log timestamps between two points in your code).
   b) Most RTOS vendors brag about three things: 1) Low Interrupt handler latency, 2) Low Context switch time and 3) Stable timer services where interval timer interrupts, timeouts, and knowledge of relative time has low jitter and drift. Why are each important?
   c) Do you believe the accuracy provided by the example RT-Clock code? Why or why not?

4) [40 points] This is a challenging problem that requires you to learn a bit about Pthreads in Linux and to implement a schedule that is predictable.
   a) Download, build and run code in the following three example programs: 1) simplethread, 2) rt_simplethread, and 3) rt_thread_improved and briefly describe each and output produced. (These example programs can also be found on Canvas) [Note that for real-time scheduling, you must run any SCHED_FIFO policy threaded application with "sudo" – do man sudo if you don't know what this is].
   b) Based on the examples for creation of 2 threads provided by incdecthread/pthread.c. Describe the POSIX API functions used by reading POSIX manual pages as needed and commenting your version of this code. Note that this starter example code - testdigest.c is an example that makes use of and sem_post and sem_wait and you can use semaphores to synchronize the increment/decrement and other concurrent threading code. Try to make the increment/decrement deterministic (always in the same order). You can make thread execution deterministic two ways – by using SCHED_FIFO priorities or by using semaphores. Try both and compare methods to make the order deterministic and compare your results.
   c) Describe how you would attempt to implement Linux code to replicate the LCM invariant schedule implemented in the VxWorks RTOS (sequencers/lab1.c) which produces the schedule measured using event analysis shown below:



The observed timing above fits our theory for RM policy on a priority preemptive scheduling system as shown by the timing diagram below:



Your description should outline how you would implement code equivalent to the VxWorks synthetic load generation and schedule emulator.

d) Code the Fib10 and Fib20 synthetic load generation and work to adjust iterations to see if you can at least produce a reliable 10 millisecond and 20 millisecond load on a Virtual Machine, or on a Jetson, Altera or Raspberry Pi system (they are preferred and should result in more reliable results). Based upon POSIX Pthread code and examples of the use of interval timers and semaphores, ***please attempt to implement two services using POSIX threading that compute a sequence (synthetic load) and match the original VxWorks diagram*** with: S1=f10, C1=10 msec, T1=20 msec, D1=T1 and S2=f20, C1=20 msec, T2=50 msec, D2=T2 as is diagrammed above in the timing diagram and shown with the VxWorks trace. You may want to review example code for help (sequencer, sequencer_generic) and look at a more complete example in this contributed Linux code from one of our student assistants in Report.pdf. Recall that U=90%, and the two services f10 and f20 simply burn CPU cycles computing a sequence and run over the LCM of their periods – 100 msec. The trace above was based on this original VxWorks code and your code should match this schedule and timing as best you can.

e) Describe whether you are able to achieve predictable reliable results when you implemented the equivalent code using Linux and pthreads to replicate the LCM

invariant schedule. Provide a trace using syslog events and timestamps (Example syslog) and capture your trace to see if it matches VxWorks and the ideal expectation. Explain whether it does and any difference you can note.

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've done. Include any C/C++ source code you write (or modify) and Makefiles needed to build your code. I and the TAs 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 from the TAs if you get stuck.

Upload all code and your report completed using MS Word or as a PDF to Canvas and include in an appendix all source code with example output integrated into the report directly. ***Your code must include a Makefile so the TAs or Instructor can build your solution on Ubuntu VB-Linux, or a DE1-SoC or Jetson or Raspberry Pi. Please zip or tar.gz your solution with your first and last name embedded in the directory name.***

Hints

You will find the LLNL (Lawrence Livermore National Labs) pages on pthreads to be quite helpful. Another great resource is Gallmeister's POSIX.4 book which can also be found on Canvas.
If you really get stuck, a detailed solution and analysis can be found on Canvas in Report.pdf, but if you use it, be sure to cite it and make sure you understand it and can describe it well. If you use this resource, note how similar or dissimilar it is to the original VxWorks code and how predictable it is by comparison.

Grading Rubric

[20 points] Rate Monotonic Analysis and Timing Diagrams:

Section Evaluation	Points Possible	Score	Comments
Correct diagram	10		
Feasibility and safety issues articulated	5		
Utility and method description	5		
TOTAL	20		

[20 points] Shared CPU system overload:

Section Evaluation	Points Possible	Score	Comments
Apollo 11 reading and summary	4		
Root cause analysis and description	4		
RM LUB plot and description of margin	4		
3 Key assumptions made by Liu and Layland and three doubts	4		
Would RM analysis have prevented the Apollo 11 errors (at least 2 main arguments)	4		
TOTAL	20		

[20 points] POSIX Real-Time Clock:

Section Evaluation	Points Possible	Score	Comments
Download, build, run as is and explain what the code does	5		
Description of code as is	5		
What is value of each RTOS bragging point?	5		

Determination and argument for or against accuracy of RT clock on system tested	5		
TOTAL	20		

[40 points] Pthread download, build, and analysis of features:

Section Evaluation	Points Possible	Score	Comments
Download, build, run simple thread code – show screen shot test.	5		
Download, build, run incdecthread code showing you did this and made modifications requested	5		
Description of key RTOS / Linux OS porting requirements including use of threads and semaphores for schedule emulation and Synthetic workload generation	15		
Synthetic workload analysis and adjustment on test system – using example code and adapting to match 90% utility two service timing example	5		
Implementation of the LCM invariant schedule.	5		
Description of challenges and test/prototype work and to trace it – does result match timing expected?	5		
TOTAL	40		