

(Please, zoom in this PDF document if required for a clear view of the embedded images and screenshots.)

Based on the RM fixed-priority scheduling policy, processes S1, S2, and S3 will have a priority in the order of: **Priority(S1) > Priority(S2) > Priority(S3)**. This is inherently based on the fundamental concept behind RM fixed policy that the most frequently requested service will be assigned the highest priority while the least frequently requested service will receive the lowest priority.

Reasons for the stand in the solution:

This RM schedule is feasible. This is essentially because the above timing diagram will be in a continuous repetition where at the end of the 15th millisecond time period, the scheduling will be repetitive. Given the repetition, it is certain and deterministic that the RM fixed policy schedule is feasible. However, it is unsafe as the CPU margin available in this case is just about 6.67% which is way less than the one suggested by Liu and Layland in their research. For a safe real-time system, the CPU margin available should be less than or close to the RM LUB required margin of approximately 30%. Thus, the scheduling is feasible but unsafe.

- 2) [15 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. What was the root cause of the overload and why did it violate Rate Monotonic policy? 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. Plot this Least Upper bound as a function of number of services and describe 3 key assumptions they make and document 3 or more aspects of their fixed priority LUB derivation that you don't understand. Would RM analysis have prevented the Apollo 11 1201/1202 errors and potential mission abort? Why or why not?

Sol.

Based on the RTECS Notes, the NASA Apollo 11 Lunar lander computer overload story revolves around the basic contention that the overloading was as a result of violation of the Rate Monotonic Policy. **The story can be briefly summarized as below:**

The Apollo-11 guidance computer system for the descent on the moon was a single computer system. It used the very concept of Rate Monotonic Scheduling with a specific priority order. The Apollo-11 guidance computer system had a 12-words data area called the "Core Set". It contained all the information to execute a given program. There were 6 of these Core Sets in the Command and 7 in the Lunar module. In each 12-word Core Set, 5-words were used for processing information and the 7 remaining words for extra storage (for variables essentially), whatever the need be. So, 5-words were used to store information to execute a program and the remaining 7-words were used to store temporary information like variables. Furthermore, the operating system was designed in a fashion that upon a request for execution of a program thread, based on the priorities assigned, it would search for a vacant space in the Core Set for executing the program. There were enough Core Sets to service the system requests efficiently. But, in reality, the computer did get overloaded. However, while performing the descent on the moon, the astronauts left the radar on slew mode which would continuously send the data points (in real-time) to the command module about the location of the lunar module and were essentially flooding the Apollo-11 guidance computer with counter interrupt signals. This was due to an oversight in the computer power supply design structure. The signals were taking up just a little bit of the computer processing time, and the spurious job kept running in the background. This ultimately prevented the computer from completing vital programs from completing and when a new task was requesting a space in the Core Set, there was no vacancy. This error would throw a 1201 error and the program would branch to alarm/abort routine. In addition to this, since no Core Sets were available because of the spurious task running in the back of the computer, the program would branch to 1202 error as per the RTECS Notes. However, this in turn triggered a software reboot. All the jobs were cancelled regardless of priority and started up again as per their priority, so quickly that no data were lost.

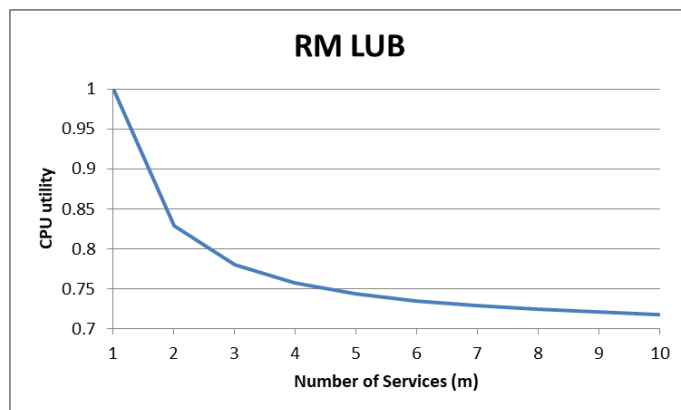
This clearly explains that the system was overloaded which in turn threw a 1201 error (for no VAC area was available in the Core Sets) and eventually 1202 because no Core Sets were available after which the system

rebooted quickly every time which allowed the system to complete the landing process whatsoever. Moreover, as explained in the notes, the processor overload indicated by 1202 was indeed as a result of more computing request than could be handled by the required deadlines forcing the system to reboot instantaneously.

CAUSES:

Based on the information provided above, root cause of the overload was mainly because of the data flooding that happened when the astronauts put the radar into the slew mode (which was unexpectedly flooding the computer with information). It did violate the rate monotonic policy because the information that was flooding the computer system prevented the scheduler to meet the required deadlines for other processes. It was ultimately not a case of priority inversion but potentially a deadlock – where the flooded information was being served continuously instead of the important services that were to be completed within the required deadlines.

RM LUB PLOT:



The above plot demonstrates the relationship between the CPU utility and the number of services that need to be serviced. This plot belongs to the course material and is duly credited as the work of Dr. Siewert (Course Instructor). The same can be accessed [here](#).

The margin is essentially the upper limit of CPU utilization that ensures that the system is safe. Using any CPU resource without a margin is very detrimental to a real-time system when safety is a high-priority and the results for missing a deadline can be catastrophic. The margin of a CPU utilization according to the RM LUB theorem should leave around 30% of CPU utility for safety and resource management in case of unanticipated circumstances. The potential disharmony in period is the reason that the RM LUB is less than full utility for two or more services.

ASSUMPTIONS MADE:

The assumptions made in the Liu Layland research work are as under which is solely their work and is duly credited:

- (A1) The requests for all tasks for which hard deadlines exist are periodic, with constant interval between requests.
- (A2) Deadlines consist of run-ability constraints only --i.e, each task must be completed before the next request for it occurs.
- (A3) The tasks are independent in that requests for a certain task do not depend on the initiation or the completion of requests for other tasks.
- (A4) Run-time for each task is constant for that task and does not vary with time. Run-time here refers to the time which is taken by a processor to execute the task without interruption.

(A5) Any non-periodic tasks in the system are special; they are initialization or failure-recovery routines; they displace periodic tasks while they themselves are being run, and do not themselves have hard, critical deadlines.

Arguments for and against RM policy and analysis prevention of Apollo 11 overload scenario:

The RM policy analysis could have prevented the Apollo 11 overload scenario. This is because of the following points:

1. If sufficient real-time simulations were performed, the flooding of radar data-points in the computer system of Apollo-11 while descending on the moon could have been spotted. The RM policy analysis would have made it possible to inspect in real-time if the Core Sets were full accidentally (error 1202) in case of an information bombardment (as it were unspotted in reality).
2. Secondly, the error 1201 (as a result of no VAC space) would have been debugged better with the RM policy as it would explain the logging of service requests and the computer system's response to it. At the core of the problem, the RM policy was violated as the Apollo-11 guidance computer was not going to meet the deadlines. However, thanks to the software that rebooted the system to fix the RM policy violation issues in real-time.

Credits:

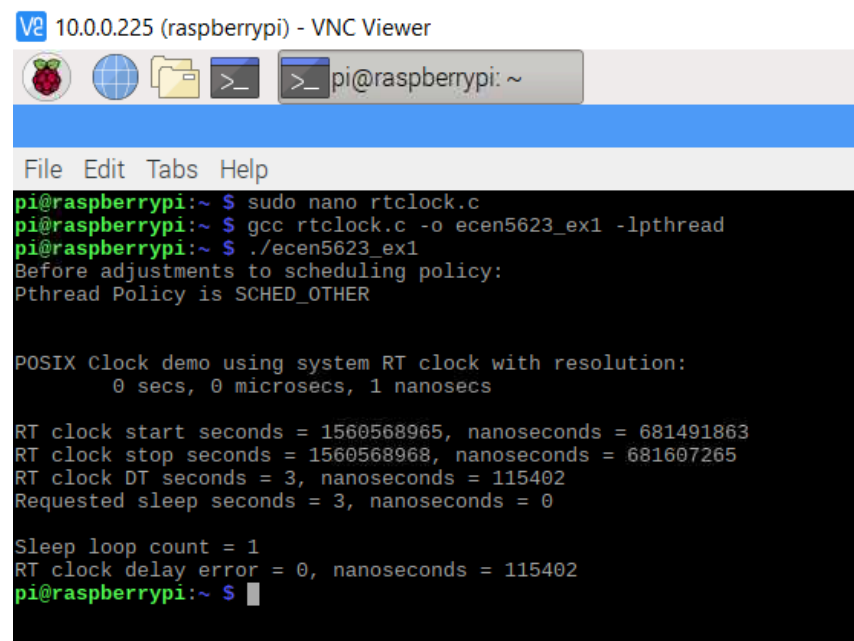
[The information provided in this answer has been sourced from a video which can be found [here](#).]

[Liu-Layland Research paper can be found [here](#).]

- 3) [20 points] Download [RT-Clock](#) and build it on an R-Pi3b+ or Jetson and execute the code. Describe what it's 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). 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? Do you believe the accuracy provided by the example RT-Clock code?

Sol.

The terminal output for a R-Pi3B+ environment for the RT-Clock POSIX code is as under:



```
10.0.0.225 (raspberrypi) - VNC Viewer
pi@raspberrypi:~ $ sudo nano rtclock.c
pi@raspberrypi:~ $ gcc rtclock.c -o ecen5623_ex1 -lpthread
pi@raspberrypi:~ $ ./ecen5623_ex1
Before adjustments to scheduling policy:
Pthread Policy is SCHED_OTHER

POSIX Clock demo using system RT clock with resolution:
    0 secs, 0 microsecs, 1 nanosecs

RT clock start seconds = 1560568965, nanoseconds = 681491863
RT clock stop seconds = 1560568968, nanoseconds = 681607265
RT clock DT seconds = 3, nanoseconds = 115402
Requested sleep seconds = 3, nanoseconds = 0

Sleep loop count = 1
RT clock delay error = 0, nanoseconds = 115402
pi@raspberrypi:~ $
```

The code used to deliver the above output can be found [here](#). The code presented there has been added in the form of `rtclock.c` file in the demonstration above. The output file was named `ecen5623_ex1` for simplicity.

The above code fundamentally attempts to calculate a time period of 3 seconds with a resolution of 1 nanosecond. That said, the computer (RPI-3B+) sets a timer that begins calculating a time period of 3 seconds with some error in the calculation that lies in the range of 1-2 milliseconds. While the timer is counting, the system goes to sleep. Apparently, the computer stops incrementing/decrementing the timer (as per the hardware and software configuration) and wakes up from sleep when it receives an interrupt that specifies that the timer count is associated with a 3 seconds time delay. At the very instance the timer counts a time period of 3 seconds, the computer is subjected to the below three constraints as interrupts occur that wakes up the computer from sleep and the timer is subjected to stop incrementing/decrementing:

1. Low Interrupt Handler Latency
2. Low Context Switch Time
3. Stable Timer Services (where interval timer interrupts, timeouts, and knowledge of relative time has low jitter and drift)

Based on the above constraints, although the computer essentially stops calculating time beyond the 3 seconds period, the interrupt handler latency and context switching both adds some delay to the computer response to the already calculated 3 seconds time period. Therefore, in actual real-time, the computer stops calculating the 3 second time period a little later after a certain fraction of time which is about 1-2 milliseconds has passed due to the system latencies. These latencies result into an error of about 1.15 milliseconds to be precise.

Detailed values behind each RTOS bragging point:

1. Low interrupt handler latency

Based on the theory provided in the RTECS notes, low interrupt handler latency is important for any real-time embedded system as it governs the amount of time that the system requires to “begin responding to an interrupt situation”. If the interrupt handler latency is high, the system interrupts will be serviced only after a significant amount of time from the moment the interrupt occurs. This not only undermines the effectiveness of the RM fixed-priority policy scheduler but also affects dynamic priority schedulers where tasks may be serviced, out of order, very late, or worse, would never be serviced due to an unanticipated timeout. In fact, it would be highly undesirable in isochronal real-time systems where an early or late system response is highly frowned upon. It is therefore important that the interrupt latency is as low as possible so that the processor OS can respond to the urgent system needs in real-time.

2. Low context switch time

Based on the theory provided in the RTECS notes, low context switching time is important as it governs the efficiency of the preemption and service dispatches performed by the RTOS scheduler. Context switch latency comes from the time it takes code to acknowledge an interrupt indicating data is available, to save register values and stack for whatever program may already be executing (preemption), and to restore state if needed for the service that will process the newly available data. Therefore, low context switch time is directly linked with how fast the scheduler can service the thread which is of higher-priority without missing a deadline and possibly a timeout in case of hard deadlines in real-time systems. Higher context switching time basically kills the very idea of real-time systems where a service may not be handled in real-time.

3. Stable timer services (where interval timer interrupts, timeouts, and knowledge of relative time has low jitter and drift)

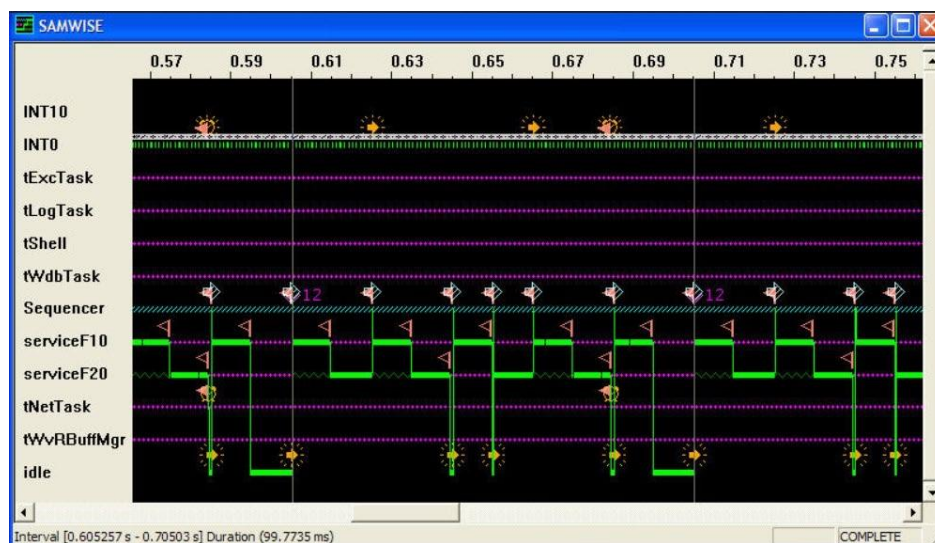
Based on the theory provided in the RTECS notes, stable timer services is probably the most important pillar for any real-time system. Stable timer services ensures that the system will provide services which meet deadline requirements with expected utility. Plus, stable timer services will ensure that the processor spends its resources only after threads that are supposed to run during the time allotted by the scheduler policy (e.g. RM fixed-priority scheduling policy) that has been implemented for a given system. Unstable timer services are not only prone to provide misleading or unreliable services but can often stall the system and can eventually result into a system crash (because of unexpected priority inversions and/or service/interrupt requests). Faulty and unexpected system behaviours (as was observed in the Apollo-11 computer system) can arise as a result of stable timer services that are out of reliability and order.

Accuracy of the example RT-Clock:

Picking up from above, the latencies offered by the computer in calculation of a 3 seconds time period results into an error of approximately 1.15 milliseconds. Although, this error is hardly of much consequence in most real-time systems but in highly sophisticated satellite systems (e.g. the CubeSats), it bears a lot of significance. Modern satellites have evolved with a task management software architecture that incorporates a circular service handling feature which simply repeats servicing the assigned and requested system tasks after a fixed certain period (usually around 15-20 milliseconds on the lower side). An error of 1-2 milliseconds in the system timing response for such sophisticated systems can result into about a 10% timing error which can turn out to be catastrophic when the system can miss certain deadlines every time as the error gradually adds up.

Courtesy: This answer contains some work that is directly taken from the RTECS reference book. The work is duly credited as it is originally authored by Dr. Sam Siewert and John Pratt.

- 4) [30 points] This is a challenging problem that requires your to learn quite a bit about pthreads in Linux and to implement a schedule that is predictable. Download, build and run code in <http://ecee.colorado.edu/~ecen5623/ecen/ex/Linux/simplethread/> and based on the example for creation of 2 threads provided by [incdecthread/pthread.c](#), as well as [testdigest.c](#) with use of SCHED_FIFO and sem_post and sem_wait as well as reading of POSIX manual pages as needed - describe how you would attempt to implement Linux code to replicate the LCM invariant schedule implemented in the [VxWorks RTOS](#) 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:

Example 5	T1	2	C1	1	U1	0.5	LCM =	10		
	T2	5	C2	2	U2	0.4				
	T3	10	C3	1	U3	0.1	U _{tot} =	1		
RM Schedule	1	2	3	4	5	6	7	8	9	10
S1										
S2										
S3										

You description should outline how you would implement code equivalent to the VxWorks synthetic load generation and schedule emulator. 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 ECES Linux or a Jetson system (Jetson is preferred and should result in more reliable results). Describe whether your able to

achieve predictable reliable results in terms of the C (CPU time) values alone and how you would sequence execution.

Sol.

The terminal output observed on the terminal for a R-Pi3B+ environment for the **Simplethread.c** code found [here](#) is as under:

```
pi@raspberrypi:~ $ sudo nano simplethread.c
pi@raspberrypi:~ $ gcc simplethread.c -o exefile -lpthread
pi@raspberrypi:~ $ ./exefile
Thread idx=1, sum[1...1]=1
Thread idx=2, sum[1...2]=3
Thread idx=3, sum[1...3]=6
Thread idx=5, sum[1...5]=15
Thread idx=4, sum[1...4]=10
Thread idx=6, sum[1...6]=21
Thread idx=7, sum[1...7]=28
Thread idx=8, sum[1...8]=36
Thread idx=9, sum[1...9]=45
Thread idx=10, sum[1...10]=55
Thread idx=11, sum[1...11]=66
Thread idx=12, sum[1...12]=78
Thread idx=13, sum[1...13]=91
Thread idx=14, sum[1...14]=105
Thread idx=15, sum[1...15]=120
Thread idx=16, sum[1...16]=136
Thread idx=17, sum[1...17]=153
Thread idx=18, sum[1...18]=171
Thread idx=19, sum[1...19]=190
Thread idx=20, sum[1...20]=210
Thread idx=21, sum[1...21]=231
Thread idx=23, sum[1...23]=276
Thread idx=24, sum[1...24]=300
Thread idx=22, sum[1...22]=253
Thread idx=25, sum[1...25]=325
Thread idx=26, sum[1...26]=351
Thread idx=27, sum[1...27]=378
Thread idx=28, sum[1...28]=406
Thread idx=29, sum[1...29]=435
Thread idx=30, sum[1...30]=465
Thread idx=31, sum[1...31]=496
Thread idx=32, sum[1...32]=528
Thread idx=33, sum[1...33]=561
Thread idx=34, sum[1...34]=595
Thread idx=35, sum[1...35]=630
Thread idx=36, sum[1...36]=666
Thread idx=37, sum[1...37]=703
Thread idx=38, sum[1...38]=741
Thread idx=39, sum[1...39]=780
Thread idx=40, sum[1...40]=820
Thread idx=41, sum[1...41]=861
Thread idx=42, sum[1...42]=903
Thread idx=43, sum[1...43]=946
Thread idx=44, sum[1...44]=990
Thread idx=45, sum[1...45]=1035
Thread idx=46, sum[1...46]=1081
Thread idx=47, sum[1...47]=1128
Thread idx=48, sum[1...48]=1176
Thread idx=50, sum[1...50]=1275
Thread idx=52, sum[1...52]=1378
Thread idx=53, sum[1...53]=1431
Thread idx=51, sum[1...51]=1326
Thread idx=54, sum[1...54]=1485
Thread idx=49, sum[1...49]=1225
Thread idx=55, sum[1...55]=1540
Thread idx=57, sum[1...57]=1653
Thread idx=58, sum[1...58]=1711
Thread idx=59, sum[1...59]=1770
Thread idx=61, sum[1...61]=1891
Thread idx=62, sum[1...62]=1953
Thread idx=56, sum[1...56]=1596
Thread idx=60, sum[1...60]=1830
Thread idx=63, sum[1...63]=2016
Thread idx=64, sum[1...64]=2080
TEST COMPLETE
pi@raspberrypi:~ $
```

In the above code for simple POSIX thread, the code creates a specific number of threads (64 in our case). These threads are attributed with default properties. The threads are assigned a task to calculate an addition of natural numbers from 1 to the number representing the thread ID. So, for an example, if the processor is servicing thread having an ID [3] then the thread will add the numbers 1, 2, and 3 and the result will be displayed as an output of the thread. The same procedure takes place for the entire set of 64 threads which are created, assigned with default properties and which later serves the function pointer `*CounterThread(*threadp)` that requires the thread to calculate the summation.

The output observed on the terminal for the **example-sync (deadlock.c)** code which can be found [here](#) code is as under:

```

pi@raspberrypi:~ $ sudo nano deadlock.c
pi@raspberrypi:~ $ gcc deadlock.c -o exefile -lpthread
deadlock.c: In function 'grabRsrcs':
deadlock.c:38:18: warning: implicit declaration of function 'usleep' [-Wimplicit-function-declaration]
    if(!nowait)    usleep(1000000);
                   ^~~~~~
deadlock.c: In function 'main':
deadlock.c:76:9: warning: implicit declaration of function 'strcmp' [-Wimplicit-function-declaration]
    if(strcmp("safe", argv[1], 4) == 0)
       ^~~~~~
pi@raspberrypi:~ $ ./exefile
Will set up unsafe deadlock scenario
Creating thread 1
Thread 1 spawned
Creating thread 2
THREAD 1 grabbing resources
Thread 2 spawned
rsrcACnt=1, rsrcBCnt=0
THREAD 2 grabbing resources
will try to join CS threads unless they deadlock
THREAD 1 got A, trying for B
^C
pi@raspberrypi:~ $ ./exefile race
Creating thread 1
Thread 1 spawned
Creating thread 2
THREAD 1 grabbing resources
THREAD 1 got A, trying for B
Thread 2 spawned
rsrcACnt=1, rsrcBCnt=0
will try to join CS threads unless they deadlock
THREAD 2 grabbing resources
^C
pi@raspberrypi:~ $
pi@raspberrypi:~ $ ./exefile safe
Creating thread 1
Thread 1 spawned
THREAD 1 grabbing resources
THREAD 1 got A, trying for B
THREAD 1 got A and B
THREAD 1 done
Thread 1: 76de2470 done
Creating thread 2
Thread 2 spawned
rsrcACnt=1, rsrcBCnt=1
will try to join CS threads unless they deadlock
THREAD 2 grabbing resources
^C
pi@raspberrypi:~ $

```

In the above output, a deadlock situation is created where two threads get control over certain portions of the critical section and further lock it (using mutexes) for preventing the other thread from modifying the data. However, both the threads 1 and 2 in the code obtain control over only a part of the critical section (as per the scheduling policy based on the *safe, race or unsafe deadlock scenarios*) and rely on each other for providing an access to the remaining part of the critical section which is not under control but deterministic. This creates a deadlock situation where both the threads wait for each other to complete a certain portion of the task and free the locked (mutexed) critical resources so that a complete access to the critical section data can be performed by the other task, which is ideally impossible given the deadlock.

The output observed on the terminal for the **example-sync (pthread3.c and pthread3ok.c)** codes which can be found [here](#) is as under:


```

root@raspberrypi:~# sudo nano pthread3.c
root@raspberrypi:~# gcc pthread3.c -o exefile -lpthread
pthread3.c: In function 'idleNoSem':
pthread3.c:72:3: warning: implicit declaration of function 'gettimeofday' [-Wimplicit-function-declaration]
  gettimeofday(&timeNow, (void *)0);
  ^~~~~~
pthread3.c: In function 'idle':
pthread3.c:98:3: warning: implicit declaration of function 'sleep' [-Wimplicit-function-declaration]
  sleep(2);
  ^~~~~
pthread3.c: In function 'print_scheduler':
pthread3.c:121:35: warning: implicit declaration of function 'getpid' [-Wimplicit-function-declaration]
  schedType = sched_getscheduler(getpid());
                                ^~~~~~
root@raspberrypi:~# ./exefile 1
interference time = 1 secs
unsafe mutex will be created
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_OTHER
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM
Creating thread 0
Start services thread spawned
will join service threads
Creating thread 3
Low prio 3 thread spawned at 1560734945 sec, 965603 nsec
Creating thread 2
Middle prio 2 thread spawned at 1560734946 sec, 965884 nsec
Creating thread 1, CScnt=1
High prio 1 thread spawned at 1560734946 sec, 966007 nsec
**** 2 idle NO SEM stopping at 1560734946 sec, 966498 nsec
**** 3 idle stopping at 1560734947 sec, 966869 nsec
LOW PRIO done
MID PRIO done
**** 1 idle stopping at 1560734949 sec, 968135 nsec
HIGH PRIO done
START SERVICE done
All done
root@raspberrypi:~# ./exefile 2
interference time = 2 secs
unsafe mutex will be created
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_OTHER
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM
Creating thread 0
Start services thread spawned
will join service threads
Creating thread 3
Low prio 3 thread spawned at 1560734959 sec, 677141 nsec
Creating thread 2
Middle prio 2 thread spawned at 1560734960 sec, 677399 nsec
Creating thread 1, CScnt=1
High prio 1 thread spawned at 1560734960 sec, 677484 nsec
**** 2 idle NO SEM stopping at 1560734960 sec, 678642 nsec
**** 3 idle stopping at 1560734961 sec, 679633 nsec
LOW PRIO done
MID PRIO done
**** 1 idle stopping at 1560734963 sec, 682186 nsec
HIGH PRIO done
START SERVICE done
All done
root@raspberrypi:~# exit
logout
pi@raspberrypi:~$

```

```

pi@raspberrypi:~ $ sudo -i
root@raspberrypi:~# sudo nano pthread3ok.c
root@raspberrypi:~# gcc pthread3ok.c -o exefile -lpthread
pthread3ok.c: In function 'idle':
pthread3ok.c:69:3: warning: implicit declaration of function 'gettimeofday' [-Wimplicit-function-declaration]
  gettimeofday(&timeNow, (void *)0);
  ^~~~~~
pthread3ok.c: In function 'print_scheduler':
pthread3ok.c:80:35: warning: implicit declaration of function 'getpid' [-Wimplicit-function-declaration]
  schedType = sched_getscheduler(getpid());
                                ^~~~~~
root@raspberrypi:~# ./exefile 1
interference time = 1 secs
unsafe mutex will be created
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_OTHER
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM
Creating thread 0
Start services thread spawned
will join service threads
Creating thread 1
High prio 1 thread spawned at 1560735138 sec, 623924 nsec
Creating thread 2
**** 0 idle stopping at 1560735138 sec, 623926 nsec
Middle prio 2 thread spawned at 1560735138 sec, 623996 nsec
Creating thread 3
**** 0 idle stopping at 1560735138 sec, 624021 nsec
Low prio 3 thread spawned at 1560735138 sec, 624088 nsec
**** 0 idle stopping at 1560735138 sec, 624110 nsec
LOW PRIO done
MID PRIO done
HIGH PRIO done
START SERVICE done
All done
root@raspberrypi:~# ./exefile 2
interference time = 2 secs
unsafe mutex will be created
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_OTHER
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM
Creating thread 0
Start services thread spawned
will join service threads
Creating thread 1
High prio 1 thread spawned at 1560735145 sec, 192070 nsec
Creating thread 2
**** 0 idle stopping at 1560735145 sec, 192075 nsec
Middle prio 2 thread spawned at 1560735145 sec, 192196 nsec
Creating thread 3
**** 0 idle stopping at 1560735145 sec, 192219 nsec
Low prio 3 thread spawned at 1560735145 sec, 192387 nsec
**** 0 idle stopping at 1560735145 sec, 192409 nsec
LOW PRIO done
MID PRIO done
HIGH PRIO done
START SERVICE done
All done
root@raspberrypi:~# exit
logout
pi@raspberrypi:~ $

```

In the output for pthread3.c code and pthread3ok.c (UPDATED) codes, the root configuration will be required on the terminal to run the code. This is because, the codes contain a scheduler FIFO configuration which requires higher order system permissions. The codes are built to deal with 1 high, 2 medium and 3 low priority tasks. The codes share light on the mutex configurations in the codes.

In the pthread3.c code, an unsafe mutex situation arises where the high priority task gets preempted and the medium and low priority tasks are completed before the high priority task gets serviced completely. This is visible from the output where the low and medium threads are spawned before the high priority thread and further the medium and high priority are serviced (removed from the idle state) before servicing (removing from the idle state) the high-priority thread.

In the pthread3ok.c code, the task threads are spawned in the order of their priority from high to low. However, the presence of an unsafe mutex in this code does creates an unsafe priority inversion. The threads are serviced in the reverse order (low to high priority).

Description of key RTOS / Linux porting OS requirements:

An RTOS (a real-time operating system) is an operating system that is intended to serve real-time applications which care about the response deadlines of the system. An RTOS with hard deadlines can have catastrophic results if the system response misses the required deadlines. On the contrary, missing soft RTOS deadlines do not yield

catastrophic results but certainly provided a low-fidelity application services which may or may not be useful at all.

The key porting OS requirements revolve around the following topics:

1. Threading vs tasking:

Threads of execution are fundamentally a fraction of a process that an OS runs on a system. A process could be anything like serving a user with the application of text editing (e.g. Microsoft Word). Threads of execution are basically formed by breaking a process into many different set of tasks with a group of tasks collectively called a thread of execution and a group of threads of execution forming one process. So, for an example, spelling-check and auto-saving of the text in the Microsoft Word are basically two separate threads of execution with two different purposes serving one process of providing the user with a Microsoft Word application. On the other hand, each thread of execution (e.g. spelling-check) will have a set of low-level instructions that the OS will service and these set of instructions form a task and tasks collectively form a thread of execution.

2. Semaphores, wait and sync:

Semaphores are a signaling mechanism that play a role with the mutexes in a RTOS. Mutexes act as a locking mechanism for when critical sections are being handled by different threads. Mutexes protect the data in the critical section from any accidental contamination. Semaphores signal whether a critical section is requiring a locked state or an unlocked state. An interrupt service routine (ISR) can indicate device data availability by setting a semaphore (flag), which allows the RTOS to transition the thread waiting for data to process from pending back to ready state.

On the other hand, thread synchronization is the concurrent execution of two or more threads that share critical resources. Threads should be synchronized to avoid critical resource use conflicts. Otherwise, conflicts may arise when parallel-running threads attempt to modify a common variable at the same time.

In addition to thread synchronization, message queues source the very concept of thread wait. The idea is that the OS stalls the execution of a thread until necessary or requisite data is available or has been processed by other concurrently running threads so that the critical sections are serviced flawlessly. Messages or in other words, data/information is stored in queues (stored in buffers) that wait for the system to respond (based on other threads of execution). A brief example of this would be the communication systems where the reliable baud rate ensures that the faster side communicates only at a speed lesser or equal to the speed that the slower side can handle for data exchange.

3. Synthetic work load generation

Synthetic work load generation is focused on testing the RTOS with certain known tasks under a set of threads of execution (e.g. read, write, list requests in a Hadoop File System which is essentially a soft RT system). A synthetic load generation framework is developed to test the partitioning algorithm which creates the tasks, executes them and measures the real-time characteristics. The user can control the intensity of the load by adjusting parameters for the threads in execution, the delay and the priorities between executions. Users can profile and monitor the system response while the synthetic load generator is running. When a load generation ends, it provides a set of log values that provide an insight into the kind of operation and resource utilization that the process undergoes.

Synthetic Work Load Analysis:

The code for this work has been completely reused which is authored by Dr. Siewert and can be found [here](#). The output is as under:

File Edit Tabs Help

```
pi@raspberrypi:~ $ sudo -i
root@raspberrypi:~# sudo nano code.c
root@raspberrypi:~# gcc code.c -o exefile -lpthread
code.c: In function 'main':
code.c:286:70: warning: implicit declaration of function 'get_nprocs_conf' [-Wimplicit-function-declaration]
    printf("System has %d processors configured and %d available.\n", get_nprocs_conf(), get_nprocs());
                                                                    ^~~~~~
code.c:286:89: warning: implicit declaration of function 'get_nprocs' [-Wimplicit-function-declaration]
    printf("System has %d processors configured and %d available.\n", get_nprocs_conf(), get_nprocs());
                                                                    ^~~~~~

root@raspberrypi:~# ./exefile
System has 4 processors configured and 4 available.
Using CPUS=1 from total available.
Pthread Policy is SCHED_FIFO
PTHREAD SCOPE SYSTEM
rt_max_prio=99
rt_min_prio=1
Service threads will run on 1 CPU cores
F10 runtime calibration 0.261873 msec per 635 test cycles, so 38 required
F20 runtime calibration 0.262186 msec per 635 test cycles, so 76 required
Start sequencer
Starting Sequencer: [S1, T1=20, C1=10], [S2, T2=50, C2=20], U=0.9, LCM=100

**** CI t=0.001198
F10 start 1 @ 0.000072 on core 3
F10 complete 1 @ 10.011034, 38 loops
F20 start 1 @ 10.039524 on core 3
t=20.067797
F10 start 2 @ 20.001651 on core 3
F10 complete 2 @ 30.032113, 38 loops
F20 complete 1 @ 39.793305, 76 loops
t=40.120231
F10 start 3 @ 40.159762 on core 3
F10 complete 3 @ 50.145431, 38 loops
t=50.185848
F20 start 2 @ 50.284962 on core 3
t=60.328756
F10 start 4 @ 60.438964 on core 3
F10 complete 4 @ 70.383176, 38 loops
F20 complete 2 @ 80.073952, 76 loops
t=80.443481
F10 start 5 @ 80.471710 on core 3
F10 complete 5 @ 90.396495, 38 loops

**** CI t=100.501487
F10 start 6 @ 100.607736 on core 3
F10 complete 6 @ 110.590437, 38 loops
F20 start 3 @ 110.659864 on core 3
t=120.617045
F10 start 7 @ 120.681055 on core 3
F10 complete 7 @ 130.613912, 38 loops
F20 complete 3 @ 140.337813, 76 loops
t=140.705728
F10 start 8 @ 140.753010 on core 3
F10 complete 8 @ 150.735303, 38 loops
t=150.773272
F20 start 4 @ 150.898375 on core 3
t=160.957326
F10 start 9 @ 161.121492 on core 3
F10 complete 9 @ 171.281796, 38 loops
t=181.140019
F10 start 10 @ 181.250039 on core 3
F10 complete 10 @ 191.244594, 38 loops
F20 complete 4 @ 191.630842, 76 loops

**** CI t=201.323910
F10 start 11 @ 201.513856 on core 3
F10 complete 11 @ 211.595828, 38 loops
F20 start 5 @ 211.818587 on core 3
t=221.535196
F10 start 12 @ 221.624362 on core 3
F10 complete 12 @ 231.599928, 38 loops
t=241.665545
F10 start 13 @ 241.787732 on core 3
t=251.837776
F10 complete 13 @ 252.026317, 38 loops
F20 complete 5 @ 252.525793, 76 loops
F20 start 6 @ 252.615064 on core 3
t=262.050093
F10 start 14 @ 262.185520 on core 3
F10 complete 14 @ 272.312134, 38 loops
t=282.218065
F10 start 15 @ 282.350140 on core 3
F10 complete 15 @ 292.560629, 38 loops
F20 complete 6 @ 293.234458, 76 loops

TEST COMPLETE
root@raspberrypi:~#
```

The Synthetic Load Generation code is based on the fundamental idea of sequencing threads such that each thread service period meets a particular value. To serve this purpose, a rate monotonic (RM) policy is applied in the code.

A detailed explanation on the code is as follows:

- The code contains two tasks (threads) that ideally run for 10 and 20 milliseconds (execution times) with a periods of 20 and 50 milliseconds respectively.
- A separate third thread is utilized to sequence the two threads and its execution such that it meets the RM policy.
- According to Liu Layland's work, the RM policy, the parameters for the code are S_1 and S_2 (where S_i is a service), $T_1 = 20$ milliseconds, $C_1 = 10$ milliseconds, $T_2 = 50$ milliseconds, $C_2 = 20$ milliseconds. (T_i is the period of the service and C_i is the execution time of the service).
- As per the RM policy, in this code, a thread with high-frequency of execution request is served with higher priority and the task with low-frequency of execution request is served with lower-priority.
- To serve the above purposes, we require the two threads to calculate Fibonacci Numbers such that they individually take 10 and 20 milliseconds (ideally) every time when they are executed to complete the assigned task.
- The code uses two parameters that are used for tuning and sequencing the thread executions: Number of CPU cores and the number of test cycles used for the threads. In addition to this, the Fib(10) and Fib(20) functions are built in a way that they count Fibonacci Numbers such that their execution on a single CPU core should take ideally 10 milliseconds and 20 milliseconds.
- The sequencer (thread) has the highest priority that will regularly call the two threads pertaining to Fib(10) and Fib(20) in a way that they meet the following conditions:
 - a. The threads pertaining to Fib(10) and Fib(20) functions will have a runtime calibration such that they take x milliseconds to be executed completely (since the functions are static and do not change in runtime).
 - b. The runtime calibration values for Fib(10) and Fib(20) will decide the number of times these functions will have to be run by the sequencer in order to **generate a synthetic load on the CPU** for 10 or 20 milliseconds. So suppose, it takes 1 milliseconds for Fib(10) for its complete execution, then the sequencer will run this thread for 10 times such that it consumes 10 milliseconds of time of CPU execution and thereby it generates a synthetic load of 10 milliseconds on the CPU.
 - c. The sequencer provides a time stamp. It provides a timestamp every 20, 20, 10, 10, and 20 milliseconds. This can be observed in the *Sequencer(*threadp) function.
 - d. The cyclic executive loop is set such that the Fib(20) thread will be preempted by the Fib(10) because of its high-frequency execution requests and therefore, the Fib(20) will report for a 30 milliseconds time period while it runs only for 20 milliseconds apart from the 10 milliseconds pre-emption.
- The program execution will take place as below (as can be learnt from the main loop):
 1. The sequencer semaphores are initialized.
 2. The FIFO Scheduler Policy is activated.
 3. The Real-Time Max and Min priorities are defined.
 4. The threads are initialized (i.e. their schedule policy is assigned along with CPU resources (which is not entirely deterministic))
 5. The threads are run for once and the ideal thread runtime (calibration) is noted for one execution of F10 and F20.
 6. The number of loop executions for F10 and F20 is calculated by the scheduler to provide a 10 and 20 milliseconds timing.
 7. The sequencer begins the execution by providing the threads with their assigned attributes for the number of cycles and loops defined.

The output for this problem shows how the processor load management take place.

MODIFICATIONS:

For my code, I modified the FIB_TEST_CYCLES to 635 which actually allows for a more calibrated F10 and F20 runtime outputs which ultimately allows for a more refined scheduling. With the 635 value instead of 100, I was able to run the threads with an average error of **0.1 to 0.2 milliseconds** on the high side. I do not completely understand it but due to some disconnect, the average error rate is a little bit higher for the F20 thread than for the F10 but is still in the range of 0.1 milliseconds.

Courtesy: This answer contains some work that is directly taken from the RTECS reference book. The work is duly credited as it is originally authored by Dr. Sam Siewert and John Pratt. Credits also to external works (which can be found [here](#), [here](#) and [here](#)) for the help in this question.