University of Colorado Boulder

Harsimransingh Bindra
Vidur Sarin

Exercise 1 - Time Invariant LCM Schedules

Table of Contents

University of Colorado Boulder

Harsimransingh Bindra
Vidur Sarin

Question 1

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). Draw a timing diagram for three services S1, S2, and S3 with T1=3, C1=1, T2=5, C2=2, T3=15, C3=3 where all times are in milliseconds. [Note that you can find examples of timing diagrams in Lecture and here and in D2l – note that we have not yet covered dynamic priorities, just RM fixed policy described here, so ignore EDF and LLF for now]. 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). What is the total CPU utilization by the three services?

Solution:

LCM of all the periods: lcm(3, 5, 15) = 15

Timing Diagram

Feasibility and Safety Issues

According to the Lehoczky, Sha, and Ding theorem, if the proposed system can be shown to be feasible so that it can be scheduled with the RM policy over the LCM period derived from the periods of the individual services then it is real-time safe.
Now, we should utilize the mathematical expression for RM LUB(Least Upper Bound) to check for the feasibility of the service set. The expression is a follows:

$$U = \sum_{i=1}^{m} (C_i/T_i) \leq m(2^{1/m} - 1)$$

Here, $C_i \rightarrow$ Execution time of the Service S_i

$T_i \rightarrow$ Release period of the Service S_i
m → net number of services in the system sharing common CPU resources.

Evaluating,
LHS:
U = {(1/3) + (2/5) + (3/15) } = **0.933**

RHS:
$3.(2^{(1/3)} - 1)$ = **0.7798**

The above values imply that the RM LUB feasibility test failed. However, because the RM LUB is a pessimistic feasibility test i.e service sets that fail the test may still be viable but not service sets that pass the test. Hence, because the 3 services described are still schedulable within the LCM of the periods, the given service set is still schedulable.
% CPU utilization for the service set is: 0.933 * 100 = **93.3%**
Since the CPU utilization is very high the system **cannot be considered safe** but the system is definitely feasible.

Utility and method description
The timing diagram is drawn with considering priorities and deadlines for each task, the set of services were found to feasible by rate monotonic policy.
The timing diagram also gives information about the CPU idle time. The following steps are followed to draw the timing diagram:

- First determine the order of which is based on the periods of the task.Period and priority are inversely related, the task with lower period will have higher priority.
- Next step is to calculate the LCM of all the periods to find if the service is invariant indefinitely. The LCM of the given three task periods(3,5,15) is 15.
- Next step is to draw the timing diagram based on the computation time(C1,C2,C3) of the three tasks given.
- The last step is check the feasibility of the services based on the timing diagram and RM LUB expression.

Question 2

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?

Solution:

Apollo 11 reading and summary

- There was a memory constraint on the system. There was only 2048 words of erasable memory and because of this, the programmers were forced to reuse the same memory address for different purposes at different times. They tested to make sure that the same memory location was not being used by different programs at the same time.
- The only time that they executed programs out of the erasable memory location was when they had to patch the program post release.
- The embedded system on the lander was:
    A. interrupt driven
    B. time dependant
    C. priority ordered jobs that dealt with less time-critical things
- Each scheduled job had some erasable memory that it could use for intermediate computational results. This was done so that certain jobs could concurrently access that region in the memory to share the computed data. Each job had a core set of 12 erasable memory locations.
- There existed the concept of a Vector Accumulator(44 erasable words). 7sets * 5 VAC areas in erasable memory. This was only when a job required more temporary storage.
- **The Problem:-->** repeated jobs to rendezvous radar data were scheduled because of a misconfiguration of the radar switches. That implied that there was

an overflow of the core sets in the process, generating a 1202 alarm. The 1201 alarm that came later was because of the scheduling request that caused the overflow was the one that had requested a VAC area.

- This caused a reboot of the system that would essentially make sure that all the essential tasks like steering the engine and maintaining flight path would regain its state but would make sure that all the erroneously scheduled rendezvous radar jobs would not restart and overflow that VAC area again.

Root cause analysis and description:

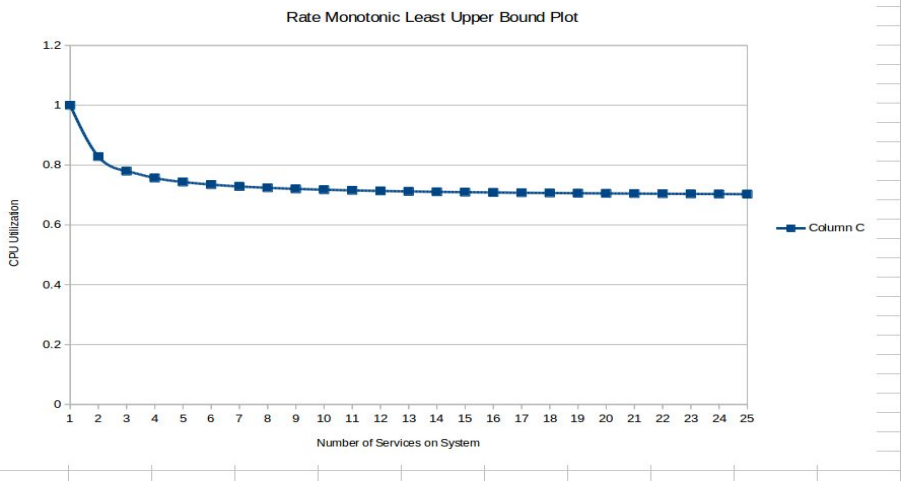
The problem on Apollo 11 was root caused to an overflow of erroneously scheduled jobs to rendezvous radar data. This lead to an overflow of the core sets of the Vector Accumulator(in erasable memory) that lead to the 1202 alarm. The 1201 was due to another scheduling request form a process that requested more VAC area while there was already an overflow of the VAC. A core requirement of the Rate Monotonic Policy is that the job occuring the with the highest frequency/period would have the highest priority. But that wasn't the case aboard Apollo 11. Clearly, the radar rendezvous jobs wasn't more important than something that controlled the engines and the flight path. However, because of the erroneous configuration, it caused an overflow of the scheduling queue and became the most frequently occuring job on the system. Hence, it violated the RM policy.

CPU Utilization Formula:

$U=m.(2^{(1/m)} - 1)$

RM LUB plot

m	$m(2^{(1/m)} - 1)$
1	1
2	0.8284271247
3	0.7797631497
4	0.75682846
5	0.743491775
6	0.7347722899
7	0.7286265957
8	0.7240618613
9	0.72053765
10	0.7177346254
11	0.7154519838
12	0.7135571323
13	0.7119589943
14	0.7105929411
15	0.7094118423
16	0.7083805188
17	0.7074721811
18	0.7066660686
19	0.7059458445
20	0.7052984768
21	0.7047134431
22	0.7041821542
23	0.7036975293
24	0.7032536794
25	0.7028456664



Description of margin:
From the LUB plot above it can be observed that as the number of services increases the CPU utilization reduces. This is to guarantee the feasibility and safety of the system. The formula for CPU utilization derived in Liu Layland paper is given below

$$U = \sum_{i=1}^{m} (C_i/T_i) \leq m(2^{1/m} - 1)$$

Where,
U-->CPU Utilization factor
Ci-->Computation time(run-times)
Ti-->Request periods
m-->Number of services
From the formula above we can derive the CPU utilization LUB value. As the value of m tends to infinity the equations narrows down to U=ln(2), where ln is the natural log. The value for **U~=0.69314718** which is the least upper bound. Therefore, CPU utilization of **69%** would mean that the system can meet all the deadlines.

Some assumptions made by Liu Layland in deriving an expression for the Least Upper Bound are [1]:

1. The request for all tasks for which hard deadlines exit are periodic, with constant interval between requests.
2. Deadlines consist of run-ability constraints only - i.e each task must be completed before the next request for it occurs.
3. The tasks are independent in that request for a certain task do not depend on the initiation or the completion of requests for other tasks.
4. Run time for each tasks 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.
5. 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.

3 points of the Fixed priority LUB theorem which we weren't able to comprehend are:

- In theorem 4, It is stated that “Clearly, C', C2', • • • , Cm' also fully utilize the processor”. The processor is fully utilized is directly stated without any explanation.
- In Theorem 3, It is stated that ”The run-time C1 is short enough that all requests for τ1 within the critical time zone of T2 are completed before the second τ2 request.” Here there is no explanation for why the assumption is made for further derivation.
- In Theorem 5, it is stated that “letT~ = qT,+r, q> landr > 0. Let us replace the task r~ by a task r(such that T; = qT~ and C~' = C~, and increase C~ by the amount needed to again fully utilize the processor”. How was the increment in Computation time calculated to fully utilize the processor again.

Arguments for RM policy are:

RM is a static, deterministic scheduling policy. This means that given a set of services that are to run on a system, you can use the feasibility test to confirm if the service set would be schedulable on the system or not. This could be successfully extended to the Apollo 11 system. The system designers could check if the given services were feasible or not and additionally assign static priorities to the tasks thereby making sure that the erroneous rendezvous radar data would not hog the system and overflow the VAC.

Arguments against RM policy:

The feasibility test for RM policy is intrinsically flawed. This is it is not a necessary and sufficient feasibility test. This implies that a feasibility test failure does not mean that the service set is not schedulable. For a hard real time system, it is necessary to have exacting qualitative analysis reports and the RM LUB is not exacting enough.

Question 3

Download http://mercury.pr.erau.edu/~siewerts/cec450/code/RT-Clock/ and build it on the Altera DE1-SOC, TIVA or Jetson board 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?

Solution:

Description of the code:

- The code is controlled by a compile time switch.
  - One part of the compile time switch does not spawn a new thread and runs the delay test for 3 seconds.
  - The other part of the switch spawns a thread after setting thread specific parameters in *main_sched_arrr* and it inherits the SCHED_FIFO property. Moreover, the thread gets the maximum priority for SCHED_FIFO assigned to it.
- delay_test(): its main function is to find out the delta in the clock when run for 3 secs for different scheduling policies and differing priorities. It uses the *clock_gettime()* to start and stop the realtime clock and record the time of start and stop, in seconds and nanoseconds.
- The CLOCK_REALTIME property provides a best effort estimate of the UTC that is backwards compatible with existing practice. This is a best guess clock that can jump forwards and backwards as the system time changes, including by NTP.[2]. Its resolution is **20ms(upper limit)**. For the given example the resolution is assumed to be **1 ns**.[3]

Importance of 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 are as follows:

1. **Low Interrupt handler latency:** Interrupt latency is the time that elapses from when an interrupt is generated to when the source of the interrupt is serviced. A particular service or task is performed by entering a particular interrupt handler when an interrupt is generated. Meeting deadlines is one of the most important task in Real-time applications. Hence if interrupt latency is high there would be delay in servicing the task and the OS may

not meet the deadline which can be fatal in hard real-time applications. Hence lower latency means there will be lesser overhead in servicing the interrupt and deadlines would not be missed.[4]

2. **Low Context switch time:** A context switch is the process of storing the state of a process or of a thread, so that it can be restored and execution resumed from the same point later. There may be more context switching if high priority tasks keep on appearing and if it is a preemptive scheduler. If the the CPU utilization is high then margin would be quite less then high context switch latency may result in missing deadlines in case of real-time systems which may be catastrophic in case of hard real-time applications. Hence low context switch time is an important aspect in real-time applications.

3. **Stable timer services where interval timer interrupts, timeouts, and knowledge of relative time has low jitter and drift:** Timer interrupts, timeouts and other timer services need to arrive at the exact time expected. The main causes of delay in timer services are jitter and drift. If there is delay in timer services then the services would not start at the expected time and there would be delay in finishing the scheduled services. If the the CPU utilization is high then margin will be quite less, in this case the system would miss the required deadlines. It may result in total system failure for Hard real-time systems.

Here are the results for the program running with SCHED_FIFO and SCHED_OTHER:
SCHED_OTHER:

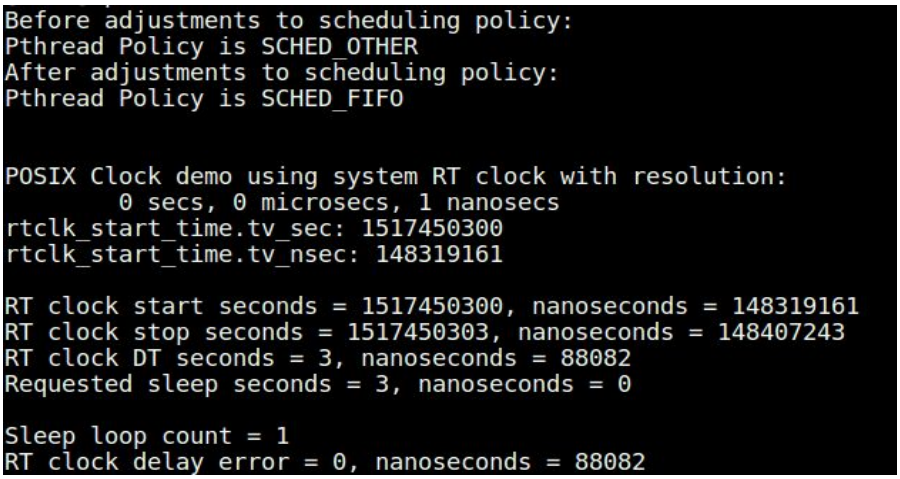
```
Before adjustments to scheduling policy:
Pthread Policy is SCHED_OTHER


POSIX Clock demo using system RT clock with resolution:
        0 secs, 0 microsecs, 1 nanosecs
rtclk_start_time.tv_sec: 1517447228
rtclk_start_time.tv_nsec: 405674270

RT clock start seconds = 1517447228, nanoseconds = 405674270
RT clock stop seconds = 1517447231, nanoseconds = 406265847
RT clock DT seconds = 3, nanoseconds = 591577
Requested sleep seconds = 3, nanoseconds = 0

Sleep loop count = 1
RT clock delay error = 0, nanoseconds = 591577
```

SCHED_FIFO:



As can be observed from the above screenshots, the jitter in nanoseconds is lesser when we run the thread with SCHED_FIFO and max priority than when the thread is not called and running other SCHED_OTHER.

Argument against the accuracy RT clock

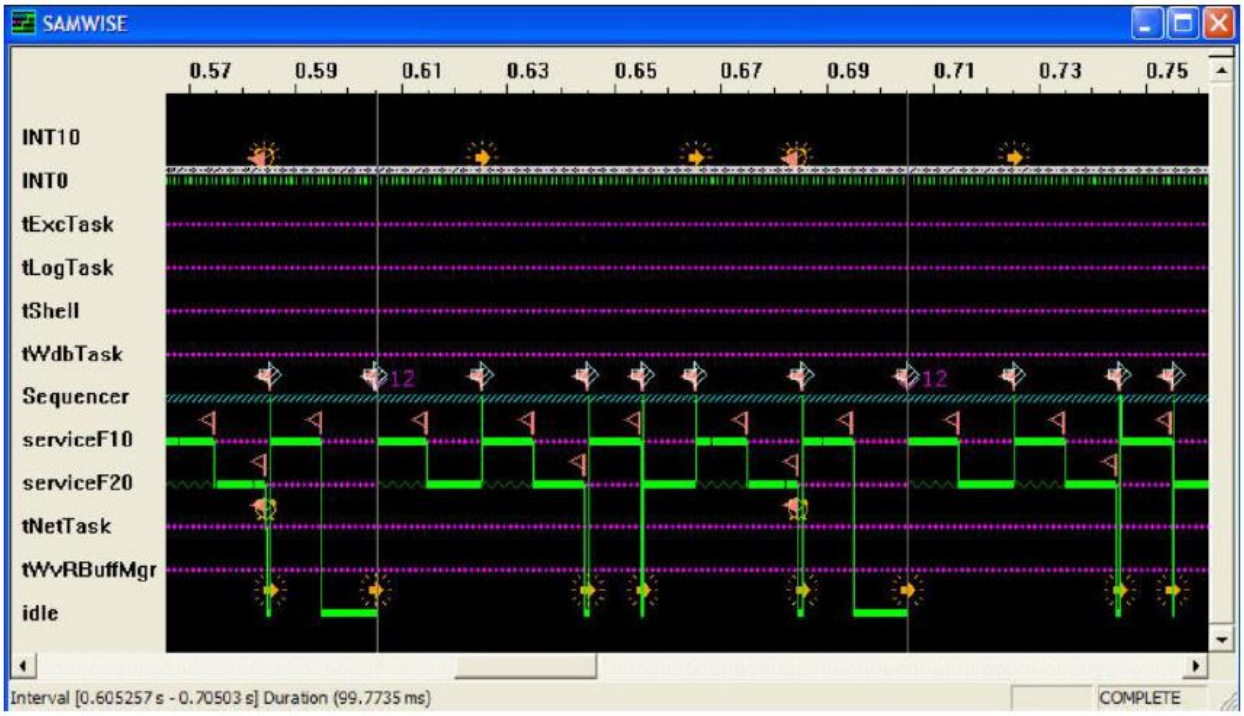
The accuracy provided by the CLOCK_REALTIME is not accurate enough for real-time applications. This is because of the following reasons

1. The jitter values for the same test are consistently very high. For example, in the SCHED_FIFO policy, we obtain an additional **88,082ns** delay.
2. The values are consistently varying each time the code is executed in the same environment. A real-time clock is expected to be far more accurate.

The above 2 reasons prove that CLOCK_REALTIME is **not accurate enough** for real-time applications.

Question 4

This is a challenging problem that requires you to learn a bit about pthreads in Linux and to implement a schedule that is predictable. 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. [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]. Based on the examples 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	Utot =	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 a Virtual Machine, or on the Jetson, Altera or TIVA system (they are 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.

Hints – You will find the LLNL (Lawrence Livermore National Labs) pages on pthreads to be quite helpful.

If you really get stuck, a detailed solution and analysis can be found here and on D2L, 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.

Solution:

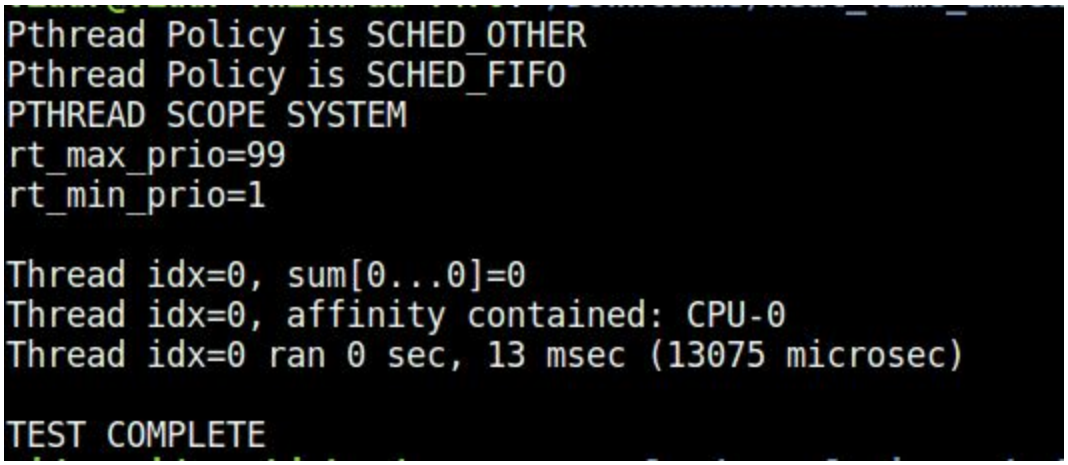
simplethread:

This program is creating 12 threads one after the other in a loop. Each thread's job is to print the sum from all numbers to its thread id. For example, if the thread number is 5 then, when thread 5 is spawned, it will run a loop within its execution context to sum up all integers starting from 0 to 5.

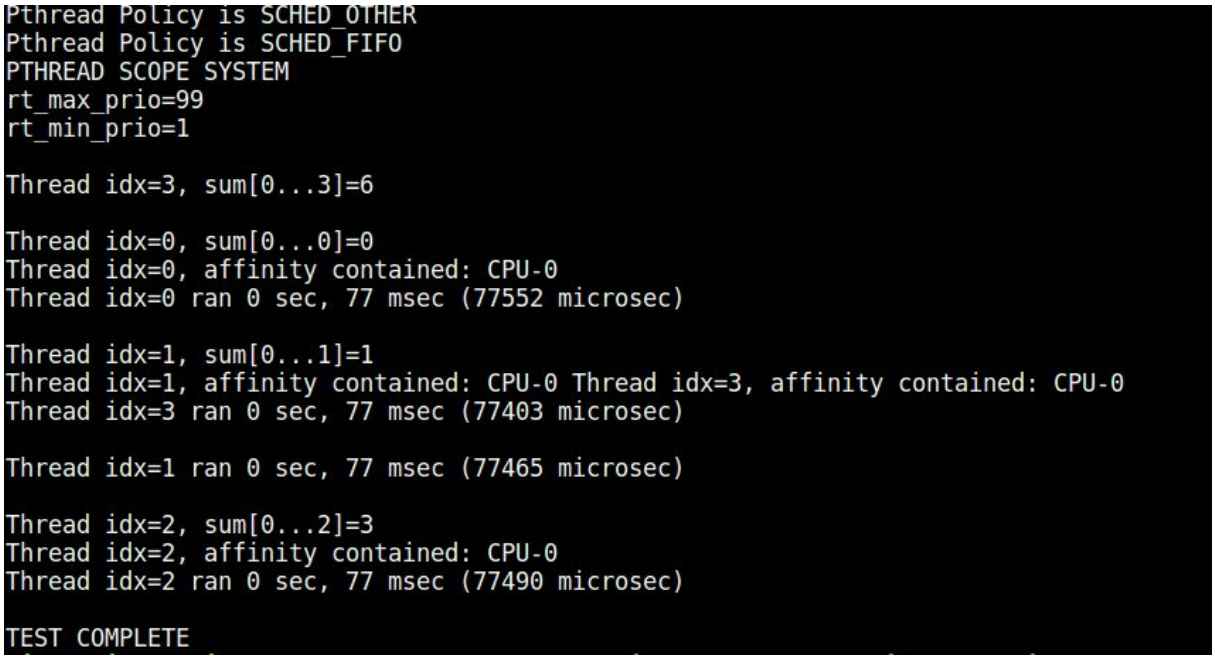
```
Thread idx=1, sum[0...1]=1
Thread idx=4, sum[0...4]=10
Thread idx=3, sum[0...3]=6
Thread idx=0, sum[0...0]=0
Thread idx=2, sum[0...2]=3
Thread idx=5, sum[0...5]=15
Thread idx=7, sum[0...7]=28
Thread idx=6, sum[0...6]=21
Thread idx=9, sum[0...9]=45
Thread idx=10, sum[0...10]=55
Thread idx=11, sum[0...11]=66
Thread idx=8, sum[0...8]=36
TEST COMPLETE
```

But everytime the code is run, the sequence in which the threads produce an output changes. The reason for this is the default priority is SCHED_OTHER which is a dynamic scheduling policy that is changed by the system based on the characteristics of the thread and the nice value for the system (0 is the default nice value). Changing this value will change the way the threads are handled[5]. (Please note that the nice value concept is only valid for systems that are based on the linux kernel).

rt_simplethread:



Single Thread Run



Running 4 Threads

1. Creates the number of threads as specified by the NUM_THREADS macro calls the callback function counterThread and passes the threadIdx to it.
2. The function counterThread does the following:
    - Get the clock time for CLOCK_REALTIME
    - Compute the sum starting from 0 to threadIdx(0, 1, 2, and so on).
    - It waits for a predetermined period doing some work, counting the fibonacci numbers. Next, it computes the sum value starting from 1 to (threadIdx+1).
    - Next, it gets and prints the CPU affinity, gets the clock time and calculates a delta of time, printing how long the computation took for the particular thread.
    - If you increase the NUM_THREADS macro and re-run the program, the prints come is different orders everytime the code is run. The reason for that is the way that the threads are scheduled by the processors.

rt_thread_improved

```
This system has 1 processors configured and 1 processors available.
number of CPU cores=1
Using sysconf number of CPUS=1, count in set=1
Pthread Policy is SCHED_OTHER
main_param: Operation not permitted
Pthread Policy is SCHED_OTHER
PTHREAD SCOPE SYSTEM
rt_max_prio=99
rt_min_prio=1
Setting thread 0 to core 0
 CPU-0
Launching thread 0
Setting thread 1 to core 0
 CPU-0
Launching thread 1
Setting thread 2 to core 0
 CPU-0
Launching thread 2
Setting thread 3 to core 0
 CPU-0
Launching thread 3

Thread idx=0, sum[0...100]=4950
Thread idx=0 ran on core=0, affinity contained: CPU-0

Thread idx=0 ran 0 sec, 51 msec (51207 microsec)

Thread idx=3, sum[0...400]=79800
Thread idx=3 ran on core=0, affinity contained: CPU-0

Thread idx=3 ran 0 sec, 89 msec (89473 microsec)

Thread idx=2, sum[0...300]=44850
Thread idx=2 ran on core=0, affinity contained: CPU-0

Thread idx=2 ran 0 sec, 87 msec (87702 microsec)

Thread idx=1, sum[0...200]=19900
Thread idx=1 ran on core=0, affinity contained: CPU-0

Thread idx=1 ran 0 sec, 83 msec (83792 microsec)

TEST COMPLETE
```

In the above rt_thread_improved code,there are 4 pthreads created and each thread is set an affinity to the corresponding CPU core. rt_thread_improved creates threads depending on the number of threads specified in the macro which runs the *counterThread(). The threads are initialized and spawned in the main function. This function is the callback of the threads and calculate the sum upto the no of thread specified*100 individually. This also binds the thread affinity to different CPU cores ((0-3)%NUM_THREADS) so threads are executed parallely. They also calculate the time between start and stop of individual threads.

Threading vs Tasking

Threads: Threads are basically "light weight processes". There are five fundamental parts of a process: code ("text"), data (VM), stack, file I/O, and signal tables. There is a significant amount of overhead associated with Heavy-weight processes such as all the tables have to be flushed from the processor for each task switch. Moreover, the only way to achieve shared information between HWPs is through shared memory and pipes . If a HWP spawns a child HWP using fork(), the only part that is shared is the text. Threads reduce overhead by sharing fundamental parts. By sharing these parts, switching happens much more frequently and efficiently. Basically it can be simply defined as threads are a part of a process and they have a shared memory space.[6]

Creating a pthread in Linux

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,\
        void *(*start_routine) (void *), void *arg);
```

Tasking:A task is similar to a process. A real-time system that uses an RTOS can be structured as a set of independent tasks. There is no coincidental dependency between tasks. Each task executes within its own context within the system or the RTOS scheduler itself. Simply we can say that it is a thread with normal thread state including stack, registers, PC, but it also includes signal handlers, identification, priority, entry point, and a number of state and inter-task communication data contained in a task context which is not present in a thread.

Creating a task in RTOS(free RTOS)

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,
                        const char * const pcName,
                        unsigned short usStackDepth,
                        void *pvParameters,
```

```
    UBaseType_t uxPriority,
    TaskHandle_t *pxCreatedTask
);
```

Semaphores:A semaphore is a value in a designated place in operating system(kernel) storage that each process can check and then change. Depending on the value that is found, the process can use the resource or will find that it is already in use and must wait for some period before trying again. Semaphores are in essence a signalling mechanism that can be incremented or decremented(signalled) by any thread within the process. This signalling capability helps synchronize different execution contexts. Semaphores can be binary(0 or 1) or can have additional values.[7]

Semaphore Creation:

Linux:

*int sem_init(sem_t *sem, int pshared, unsigned int value);*

RTOS:

SemaphoreHandle_t xSemaphoreCreateBinary(void);

Wait: There are two methods in semaphores Wait and sync. There are two possibilities when wait is executed

1. If the resources are available the semaphore value is reduced by 1. Basically it blocks the semaphore(binary semaphores).
2. If the semaphore value is 0 then the thread is put in a wait state until the semaphore is available again.

Linux:

*int sem_wait(sem_t *sem);*

RTOS:

xSemaphoreTake(SemaphoreHandle_t xSemaphore,TickType_t xTicksToWait);

Sync: This is one of the methods in semaphores. There are 2 cases

1. If there is no thread waiting the value of semaphore is increased by one and the thread resumes its execution.
2. If the thread is waiting for the counter the value of the semaphore must be zero (see the discussion of **Wait** above). One of the waiting threads will be allowed to leave the queue and resume its execution. The thread that executes **Signal** also continues.

Linux:

*int sem_post(sem_t *sem);*

RTOS:
xSemaphoreGive(SemaphoreHandle_t xSemaphore);

NOTE: All the code was run on a **Virtual machine(Linux OS)**.

```
Scheduling Policy: Pthread Policy is SCHED_FIFO
System Scope Type: PTHREAD SCOPE SYSTEM

---------------------- Start Scheduler Test -----------------------------
Iteration Settings:
reqIterations: 5500000
seqIterations: 110

(fib10):Thread Id=1 | Priority: 98 | run time: 0 sec, 9.605063 msec
(fib10):Thread Id=1 | Priority: 98 | run time: 0 sec, 28.975673 msec
(fib20):Thread Id=2 | Priority: 97 | run time: 0 sec, 37.268872 msec
(fib10):Thread Id=1 | Priority: 98 | run time: 0 sec, 49.364574 msec
(fib10):Thread Id=1 | Priority: 98 | run time: 0 sec, 70.636502 msec
(fib20):Thread Id=2 | Priority: 97 | run time: 0 sec, 79.899400 msec
(fib10):Thread Id=1 | Priority: 98 | run time: 0 sec, 91.615485 msec

Thread Id=0 | Priority: 99 | run time: 0 sec, 101.375315 msec, 101375.000000 usec
------------------------ End of Scheduler Test -----------------------------
```

Synthetic workload analysis and adjustment on test system:

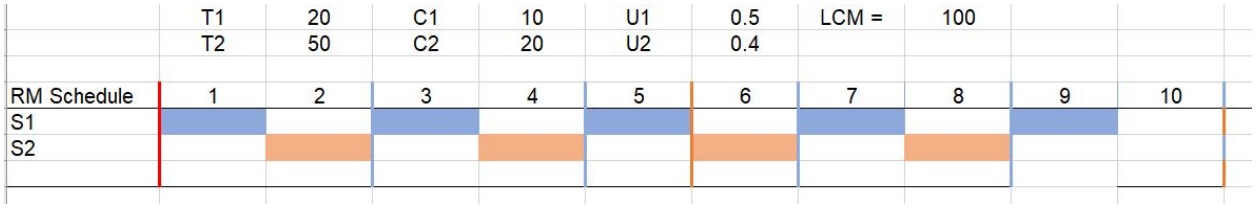
In the main function the scheduler thread is created and highest priority(99-Linux) was assigned to it. In the scheduler thread, 2 new threads(Fib10 and Fib20) were created. Fib10 was assigned priority 98 and and Fib20 was assigned priority 97. The FIB_TEST() function has 2 arguments namely, seqIterations and reqIterations. The FIB_TEST() function generates the required synthetic workload. The value of seqIterations was changed from “47” to “110” and the value of reqIterations was changed to “5500000”. These values gave the results closest to the desired value. The thread creation and priority assignment code was picked up from Dr Sam Siewert’s code. There are some parts of the code which are picked up from the independent study code provided in the link.

The code generated the desired output with respect to the timing diagram in terms of workload prioritization. The Scheduler is assigned the highest priority(99) and the workload is closest to the required values of 10ms (Fib10) and 20ms (Fib20) with some error as RT_clock in Linux is not very accurate as observed in the code provided for Q3.

Here service 1(Fib10) has computation time of 10ms and period of 20ms is assigned higher priority of 98 than service 2(Fib20) thread which has the computation time of 20ms and period of 50ms. The priority assignment is done according to the RM policy which says that the priority and period of a service is inversely related. The services

with more period are given less priority. The LCM of both the periods(20, 50) is 100 hence the total run time for the testing the code is 100ms.

Timing Diagram:



$$U = \sum_{i=1}^{m} (C_i/T_i) \leq m(2^{1/m} - 1)$$

Here, $C_i \rightarrow$ Execution time of the Service S_i
$T_i \rightarrow$ Release period of the Service S_i
$m \rightarrow$ net number of services in the system sharing common CPU resources.

Evaluating,
LHS:

U = { (1/2) + (2/5) } = **0.90**

RHS:

$3.(2^{(1/2)} - 1)$ = **0.828427**

Observations:

1. It fails the RM LUB feasibility test however, as it is a pessimistic test, not all failed results would imply that the service set would be unschedulable.
2. We observe that the services are schedulable according to the timing diagram.
3. The code was executed on the virtual machine and it gave the desired result.
4. But the point to be noted is that if the values of iteration variables(reqIterations and seqIterations) are changed, after a certain value the systems fails to give the desired output.

Challenges:

1. Get the signalling(semaphores - sem_wait() & sem_post()) to work properly in order to get the required execution sequences.
2. Setting the right reqIterations and seqIterations values to get the correct execution sequences.

References

- *https://stackoverflow.com/questions/3523442/difference-between-clock-realtime-and-clock-monotonic*
- *http://www.cl.cam.ac.uk/~mgk25/posix-clocks.html*
- *https://en.wikipedia.org/wiki/Interrupt_latency*
- *https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_MRG/2/html/Realtime_Reference_Guide/chap-Priorities_and_policies.html*
- *http://ecee.colorado.edu/~ecen5623/ecen/labs/Linux/IS/Report.pdf*
- *https://www.cs.ru.nl/~hooman/DES/liu-layland.pdf*

Exercise Staff

Harsimransingh Bindra	Vidur Sarin
Graduate Student Embedded Systems Engineering University Of Colorado Boulder Harsimransingh.Bindra@colorado.edu	Graduate Student Electrical Engineering University Of Colorado Boulder Vidur.Sarin@colorado.edu