

ECEN 5623
REAL-TIME EMBEDDED
SYSTEMS

EXERCISE-1

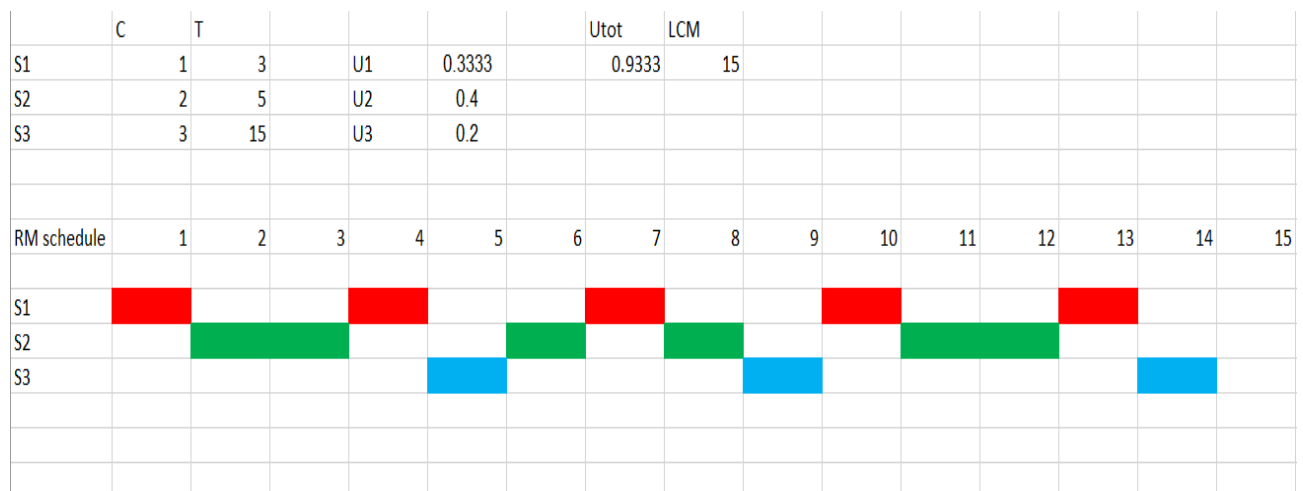
SUBMITTED BY:
ARUNSUNDAR KANNAN
SOWMYA
RAMAKRISHNAN

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 $T_1=3$, $C_1=1$, $T_2=5$, $C_2=2$, $T_3=15$, $C_3=3$ where all times are in milliseconds. [Note that you can find examples of timing diagrams in Lecture and [here](#) and in D21 – 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?

Answer:

- TIMING DIAGRAM**



- CALCULATION OF CPU UTILIZATION**

$$U = C_1/T_1 + C_2/T_2 + C_3/T_3$$

$$U = 1/3 + 2/5 + 3/15$$

$$U = 1/3 + 2/5 + 1/5$$

$$U = 1/3 + 3/5$$

$$U = 14/15 = 93.33\%$$

CPU utilization is 93.33%

- **CALCULATION OF LEAST UPPER BOUND**

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

$$\begin{aligned} \text{Least upper bound} &= 3(2^{1/3} - 1) \\ &= 3(0.2599) \\ &= 0.7797 \\ &= 77.97 \% \end{aligned}$$

- **PROCESS DESCRIPTION**

- At T1, since S1 holds the highest priority, S1 is executed first.
- S2 needs processing twice every 5ms. Thus to minimize context switching, at T2 and T3, S2 is executed and its deadline is met satisfactorily.
- Since S3 is at lowest priority, S1 is executed again since it needs frequent CPU service.
- Now at T5, S3 is executed. Thus all three deadlines are met up until T5.
- At T6, S2 is processed.
- Now at T7, since S1 needs processing once every 3 cycles, it is executed at the start of a new three set period because of higher priority.
- S2 is executed at T8 because its priority is lower than S1 and higher than S3
- Next S3 is executed.
- Again at T10, a new set of three time periods begin and S1 is serviced
- S2 is serviced at T11 and T12 to avoid context switching and to meet the deadline
- S1 is serviced at T13 and finally S3 is serviced at T14
- The CPU is left idle for T15 to ensure that the processor is not used up for all the time.

- **SAFETY AND FEASIBILITY**

When CPU utilization is less than LUB, we can guarantee that schedule is feasible. However, if it exceeds LUB, we cannot with assert with absolute certainty that schedule is feasible. In this case, the CPU utilization is more than LUB, thus require an alternate method to come to a conclusion.

According to Lehockzy, Shah, and Ding Theorem if a set of services can be shown to be feasible so that it can be scheduled with the RM policy over the LCM (least common multiple) period derived from all proposed service periods, then the Lehoczky, Shah, and Ding theorem guarantees it real-time-safe. Considering the timing diagram above, the task can be scheduled properly without missing a

deadline as the tasks are scheduled properly when we use LCM number of cycles. This can guarantee that the tasks can be scheduled without missing any deadline.

Even though this hypothesis proves that the task can be scheduled, a big hole is left behind as we assume that all the requests to the tasks are periodic. If the requests are aperiodic, then the scheduling algorithm may fail as we are not dynamically assessing the situation and scheduling. This may not be an ideal algorithm to satisfy hard real time services.

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?

Answer:

APOLLO 11 LUNAR LANDER COMPUTER OVERLOAD STORY-SUMMARY

“In the early days, women were often assigned software tasks because software just wasn't viewed as very important.”

[Source: <https://www.vox.com/2015/5/30/8689481/margaret-hamilton-apollo-software>]

Margaret Hamilton, a (revolutionary) woman, changed that. From her work being described as “the foundation for ultra-reliable software design” [Source: https://en.wikipedia.org/wiki/Margaret_Hamilton_%28scientist%29] to coining the term “Software Engineering” [Source: <https://www.vox.com/2015/5/30/8689481/margaret-hamilton-apollo-software>] which is perhaps the most widely used term in today's technology and software-driven world, Hamilton has been responsible for the growth of software and its importance, and most importantly, its role in saving the Apollo 11 mission from being aborted.

Margaret's team was responsible for designing the Apollo Guidance Computer and its on-board flight software, which was placed in both the command module and the lunar module for spacecraft control and navigation assistance. There were different modules for specific functions of the spacecraft (takeoff, landing etc.) and because of a lack of enough erasable memory (only 2048 words were available), they all ended up using the same memory locations, albeit at different times for different purposes, and were tested against interfering with each other. The problem was that, the radar system to be used when leaving the moon and reconnecting with the control module and

the computer-aided guidance system in the lunar module used incompatible power supplies. The radar, which didn't really have a purpose in the landing portion of the mission, started sending the computer lots and lots of data based on random electrical noise. These interrupts overloaded the computer and threatened to leave no room for the computational tasks necessary for landing. The alarms meant that the computer was not able to complete all its tasks in real time and had to postpone some of them. But Margaret had anticipated this kind of a problem beforehand and had the solution ready, in the form of an 'asynchronous executive'. Essentially, this meant that when overloads came up, the computer would automatically drop low-priority tasks. It was programmed to automatically and (almost) instantaneously reboot, flushing out the unimportant tasks (like the radar data). a unique priority was assigned to every process in the software to ensure that all its events would take place in the correct order and at the right time relative to everything else that was going on. This was known to the astronauts. Hence, when there was an interruption in the astronauts' displays during the alarms, they had to take a "go/no go" decision since they knew what the alarms were for and whether they were important to the landing sequence. They decided to "Go" and saved the mission from being aborted due to insignificant reasons/emergency alarms. The computer had a complete set of recovery programs incorporated in its software and it could eliminate lower-priority tasks and re-establish the important ones.

ROOT CAUSE OF THE OVERLOAD AND WHY IT VIOLATED RATE MONOTONIC POLICY

The issue with the Apollo 11 software system was that there was very little erasable memory available and hence, memory had to be managed in such a way that there were no scheduling conflicts. Great care was taken to ensure that there were no conflicts when it came to memory management. In the multi-tasking OS, there were interrupt driven tasks as well as low priority schedule tasks. In the Apollo 11, repeated jobs to process rendezvous radar data were scheduled because of a misconfiguration of the radar switches.

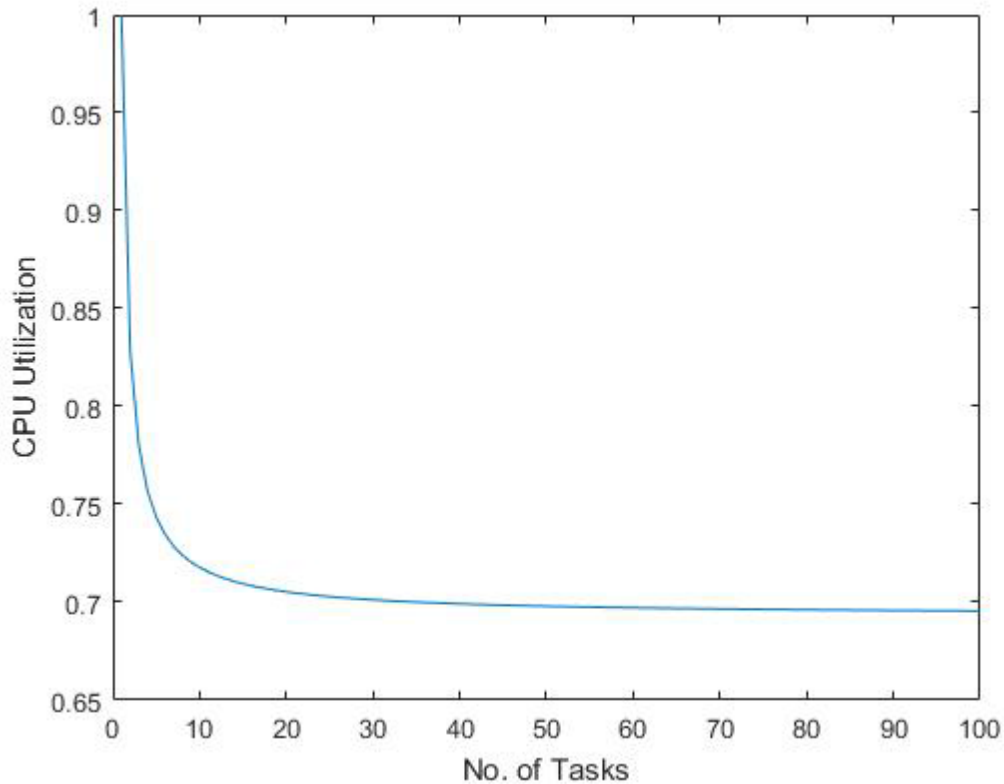
The root cause for the overload were that there was a priority inversion and the deadlines for the priorities weren't met. These triggered the alarms 1201/1202 in the Apollo 11 spacecraft, forcing the engineers to re-think whether to go ahead with the mission or abort it. The priority inversion happened because of an unexpected restart of the system causing it to mess up the deadlines. The fact that the rendezvous radar data of ascent was still being picked up during descent (where it was not needed and was unnecessary data that proved to be liability), was the main cause that triggered the 1201/1202 alarms. When the descent data was requesting a VAC (Vector Accumulator), it was not free because it was filled up with the ascent data. The CPU was programmed to ignore all these secondary triggers because of the repeated testing done at the MIT Instrumentation Lab. Thus, whenever the alarms were triggered, the system rebooted itself, focusing only on the important tasks.

The Apollo 11 scheduling violates the Rate Monotonic Scheduling Policy because of the priority inversion that happened and hence the messing up with the scheduling of the tasks. This meant that unless a system reboot occurred, the priorities were not set straight and constant triggering of 1201/1202 would occur.

RATE MONOTONIC LEAST UPPER BOUND (RM LUB) PLOT

The equation which advises margin of approximately 30% of total CPU as number of services sharing a single CPU core increases [Source: “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment – Liu and Layland”] is derived as follows.

Assume m to be the number of tasks/services. From the Source cited above (the Liu & Layland Paper), we can calculate the least upper bound on CPU utilization as $m \cdot (2^{1/m} - 1)$. Ratio between any two-request period is assumed to be less than 2. Now, there can be indefinite schedules for hard real-time constraints for all the tasks where CPU utilization falls below a Definite Least Upper Bound. Other tasks must be analyzed to determine if they can happen or not. From the graph as plotted, one can deduce that as the number of services increases, the CPU utilization saturates.



ASSUMPTIONS MADE

- Each task should be completed before the next task request occurs. This is termed as “Run-ability constraints for deadlines”.
- The requests for all the tasks meant for hard deadlines are periodic and there is a constant interval between each request.
- All tasks are independent and no task depends on the request/completion of other tasks.

APOLLO 11 AND RATE MONOTONIC POLICY

RM analysis would definitely have prevented the Apollo 11 1201/1202 errors and the potential mission abort. The RM policy would have helped schedule the tasks with deadlines in consideration. The rendezvous radar tasks would have had low priorities compared to the high priority tasks during descent. There would not have been any load on the VAC if there was Rate Monotonic policy and hence the trigger of the 1201/1202 alarms would have been avoided. The priority would have been assigned at the beginning itself, thus eliminating the tedious and time-consuming process of rebooting the system.

Why RM policy is preferred:

- Provides an optimal scheduling policy for all hard real-time systems where there is a fate constraint.
- Proper scheduling of the tasks based on their deadlines and their completion time gives each task its chance to be performed.

Aspects that are not comprehensible/against RM policy:

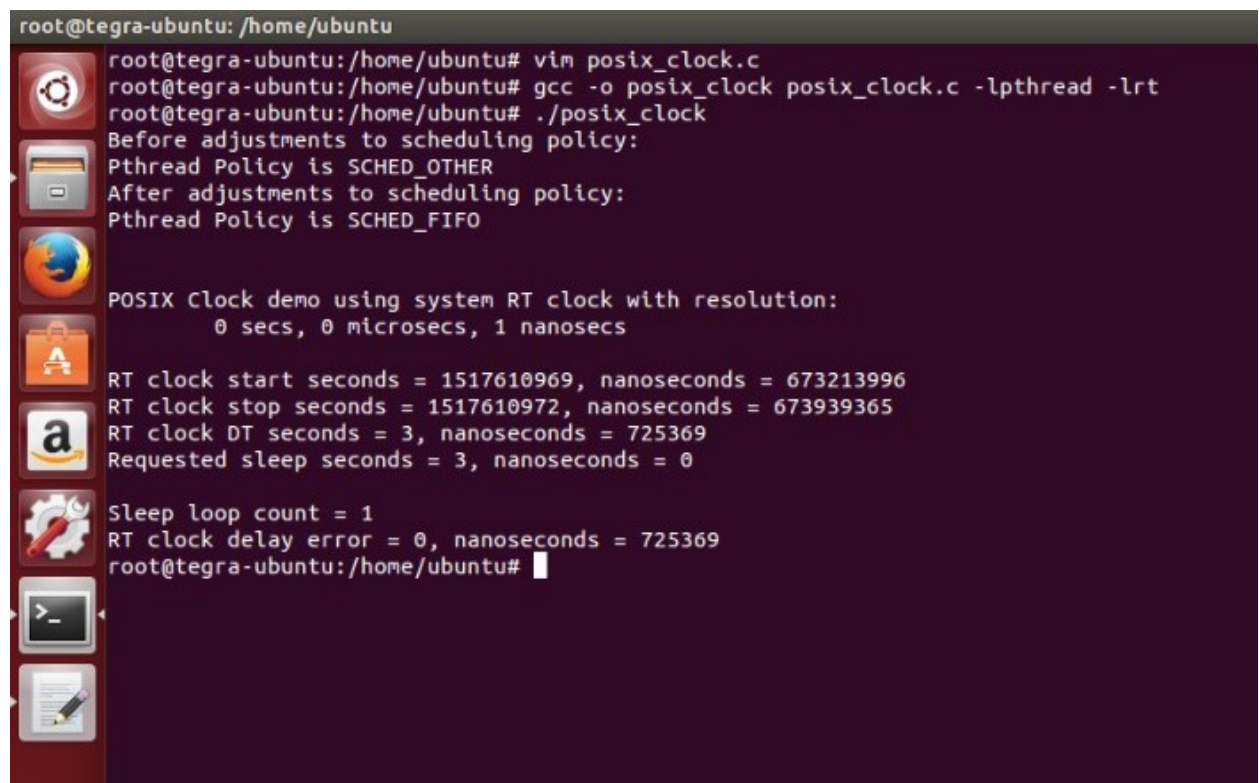
- The assumption that the ratio between periods should be less than 2 actually presents the argument of the previously stated fact.
- The least upper bound for any Rate Monotonic Policy increases for a harmonic service utilizing the CPU. Thus, there is no set limits to determine the scheduling for those services.
- Practical latency of the tasks in CPU is not considered under Rate Monotonic Policy.

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?

Answer:

- **Download, build and run:** The code is built and run in the Jetson TK-1 with root access and the output of the code is shown below in the picture.



```
root@tegra-ubuntu: /home/ubuntu
root@tegra-ubuntu:/home/ubuntu# vim posix_clock.c
root@tegra-ubuntu:/home/ubuntu# gcc -o posix_clock posix_clock.c -lpthread -lrt
root@tegra-ubuntu:/home/ubuntu# ./posix_clock
Before adjustments to scheduling policy:
Pthread Policy is SCHED_OTHER
After adjustments to scheduling policy:
Pthread Policy is SCHED_FIFO

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

RT clock start seconds = 1517610969, nanoseconds = 673213996
RT clock stop seconds = 1517610972, nanoseconds = 673939365
RT clock DT seconds = 3, nanoseconds = 725369
Requested sleep seconds = 3, nanoseconds = 0

Sleep loop count = 1
RT clock delay error = 0, nanoseconds = 725369
root@tegra-ubuntu:/home/ubuntu#
```



```
root@tegraubuntu:/home/ubuntu#
printf("requested sleep seconds = %ld, nanoseconds = %ld\n",
sleep_requested.tv_sec, sleep_requested.tv_nsec);

printf("\n");
printf("sleep loop count = %d\n", sleep_count);
printf("rt sleep delay error = %ld, nanoseconds = %ld\n",
delay_error.tv_sec, delay_error.tv_nsec);

exit(0);
}

#define RUN_RT_THREAD

void main(void)
{
    int rc, scope;

    printf("before adjustments to scheduling policy\n");
    print_scheduler();

#ifdef RUN_RT_THREAD
    pthread_attr_t attr(&main_sched_attr);
    pthread_attr_t attr_inherited(&main_sched_attr, PTHREAD_EXPLICIT_SCHED);
    pthread_attr_t attr_setschedpolicy(&main_sched_attr, SCHED_FIFO);

    rt_max_prio = sched_get_priority_max(SCHED_FIFO);
    rt_min_prio = sched_get_priority_min(SCHED_FIFO);

    main_param.sched_priority = rt_max_prio;
    rc=sched_setscheduler(getpid(), SCHED_FIFO, &main_param);

    if (rc)
    {
        printf("ERROR: sched_setscheduler to %d failed\n", rc);
        perror("sched_setscheduler"); exit(-1);
    }

    printf("after adjustments to scheduling policy\n");
    print_scheduler();

    main_param.sched_priority = rt_max_prio;
    pthread_attr_t attr_setschedparam(&main_sched_attr, &main_param);

    rc = pthread_create(&main_thread, &main_sched_attr, delay_test, (void *)0);

    if (rc)
    {
        printf("ERROR: pthread_create() to %d failed\n", rc);
        perror("pthread_create");
        exit(-1);
    }

    pthread_join(main_thread, NULL);

    if(pthread_attr_destroy(&main_sched_attr) != 0)
        perror("main_thread");

#ifdef delay_test
    delay_test((void *)0);
#endif
    printf("TEST COMPLETE\n");
}
```

- void print_scheduler(void)
 - ✓ This routine is to print the type of scheduling policy which is currently used by the operating system.
- int delta_t(struct timespec *stop, struct timespec *start, struct timespec *delta_t)
 - ✓ This routine gets three struct pointers as its input. The start pointer contains the second, nanosecond value from the clock_gettime() function before sleeping for 3 seconds.
 - ✓ The stop pointer contains the second and nanosecond value obtained from clock_gettime() function after the completion of the sleep operation.
 - ✓ The delta_t pointer subtracts the second and nanosecond data members of the start and stop struct and serves as pointer to store error caused by the delay.
- void end_delay_test(void)
 - ✓ This routine prints the results of the delta_t() function
- void *delay_test(void *threadID)
 - ✓ This routine serves as a thread which performs like a driver function to compute the delay error of the RT clock.
- Global declarations
 - ✓ The program contains global declaration of a pthread and pthread attribute variable
 - ✓ The program contains the global object declaration for the sched_param structure and the timespec structure
- Main function
 - ✓ The main function initially changes the scheduling algorithm from sched_other to sched_fifo.
 - ✓ It then initializes the pthread attribute variable and calls the pthread_attr_setinheritsched() function to set the inherit scheduler attribute which specifies that the thread will take its scheduling attributes from the values specified by the attributes object.
 - ✓ It then gets the maximum available priority from the sched_get_priority_max() function and sets the sched_priority data member of sched_param struct to the maximum available priority using sched_setscheduler() method.

- ✓ It then joins the main_thread (similar to a call to delay_test method).
- ✓ The delay_test() method gets the time before and after sleeping and subtracts the values and computes and prints the delay
- ✓ An #else statement is inserted just in case if the thread fails.

➤ **Clock_gettime()**

- ✓ This function gets a clock id and a pointer to timespec structure as inputs and sets the two data members of the timespec structure (tv_sec and tv_nsec) with second and the nano second values since the linux epoch.

- **Value of each RTOS bragging point**

➤ **Low Interrupt handler latency**

- ✓ Interrupt handler latency (or interrupt response time) is the amount of time that an RTOS takes to respond to an external event or on generation of an interrupt. It is one of the most important bragged about factor in any real time embedded system to ensure that the deadlines are met properly, in a faithful manner. In simple real time systems, the software flow is only dependent on a main configuration with functions calls based on interrupts (ISRs). In such cases, if the interrupt latency is too high, the services might miss deadlines and fail to serve their purpose due to latency. On the contrary, low latency enables faster process execution because the RTOS recognizes and processes the interrupt faster to complete a given task. It also helps to keep the WCET (worst case execution time) to a minimum low which allows higher utilization of CPU resources. Thus, low interrupt handler latency is an important parameter in RTOS selection.

➤ **Low Context Switching time**

- ✓ Context switching is the process of storing and restoring the state or the execution context (the program counter, thread or process ID, register values etc.) of a process or a thread so that its execution can be resumed from the same point at a later time. This process allows an OS to virtually multitask by dividing its processing time amongst several processes that need CPU time. It is important that process data be saved

before preempting it to resume its execution from the same point at a later stage. Since this switching occurs frequently, it is important that it takes minimum time for data push and data pop. A low latency time is critical and desirable for any real time system. Thus, several RTOS sellers pay attention to optimize this parameter for better RTOS performance.

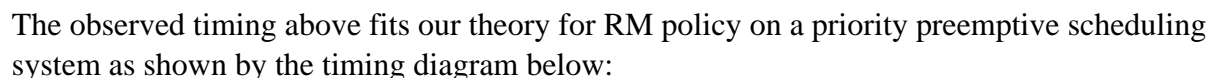
➤ **Stable Timer Services:**

- ✓ Clocks and time services are some of the basic facilities provided to programmers by every real-time operating system. The time services provided by an operating system are based on a software clock called the system clock maintained by the operating system. The system clock is maintained by the kernel based on the interrupts received from the hardware clock. Jitter plays a major role in degrading the resolution of the system clock. Jitter is caused on account of interrupts having higher priority than system calls. When an interrupt occurs, the processing of a system call is stalled. Also, the preemption time of system calls can vary because many operating systems disable interrupts while processing a system call. The variation in the response time (jitter) introduces an error in the accuracy of the time value that the calling thread gets from the kernel. In commercially available operating systems, jitters associated with system calls can be several milliseconds. Thus, it is important to keep jitter to a minimum low and is an important parameter to decide upon an RTOS usability.

- **Arguments on accuracy of the RT clock on system tested**

The start and stop variables of the RT-CLOCK show the start and end time of the 'clock_gettime' and 'nanosleep' functions. Difference between these two values is the clock delay error. The accuracy of the RT-CLOCK code (the 'nanosleep' function) varies in each run by some value. There are several reasons for this difference. This variation introduces jitter in the system clock which can prove fatal to the RTOS performance as discussed above (Stable Timer Services). Since the difference isn't constant each time, the randomness of the error does not allow us to add a correcting factor to get rid of the deviation. Thus, it is not recommended if the application requires very high accuracy since even minute deviation in timekeeping caused by the RT-CLOCK code could cause system failure. On the flip side, if the application does not require a very high accuracy in terms of resolution of nanoseconds, that is, if a jitter of a few milliseconds can be tolerated, the clock can be reliably used.

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:



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										

Your 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.

Answer:

SIMPLETHREAD

```
root@sowmya-HP-Pavilion-Notebook: /home/sowmya/Downloads/simplethread
root@sowmya-HP-Pavilion-Notebook:/home/sowmya/Downloads# cd simplethread
root@sowmya-HP-Pavilion-Notebook:/home/sowmya/Downloads/simplethread# make pthread
gcc -O0 -g -c pthread.c
gcc -O0 -g -o pthread pthread.o -lpthread
root@sowmya-HP-Pavilion-Notebook:/home/sowmya/Downloads/simplethread# ./pthread
Thread idx=0, sum[0...0]=0
Thread idx=1, sum[0...1]=1
Thread idx=5, sum[0...5]=15
Thread idx=2, sum[0...2]=3
Thread idx=3, sum[0...3]=6
Thread idx=4, sum[0...4]=10
Thread idx=6, sum[0...6]=21
Thread idx=7, sum[0...7]=28
Thread idx=8, sum[0...8]=36
Thread idx=9, sum[0...9]=45
Thread idx=10, sum[0...10]=55
Thread idx=11, sum[0...11]=66
TEST COMPLETE
root@sowmya-HP-Pavilion-Notebook:/home/sowmya/Downloads/simplethread#
```

In this program, 12 threads are created and numbered from thread0 to thread11. For each thread created, the sum is iterated from 0 to the number of the thread i.e. for example, $\text{sum} = 0 + 1 + 2 + 3 = 6$ for thread3, and so on. It is a simple program to understand the creation of threads and their working.

RT_SIMPLETHREAD

```
root@sowmya-HP-Pavilion-Notebook: /home/sowmya/Downloads/rt_simplethread
root@sowmya-HP-Pavilion-Notebook:/home/sowmya/Downloads# cd rt_simplethread
root@sowmya-HP-Pavilion-Notebook:/home/sowmya/Downloads/rt_simplethread# make pthread
gcc -O0 -g -c pthread.c
pthread.c: In function 'print_scheduler':
pthread.c:135:35: warning: implicit declaration of function 'getpid' [-Wimplicit-function-declaration]
    schedType = sched_getscheduler(getpid());
                                   ^
gcc -O0 -g -o pthread pthread.o -lpthread
root@sowmya-HP-Pavilion-Notebook:/home/sowmya/Downloads/rt_simplethread# ./pthread
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_FIFO
PTHREAD_SCOPE_SYSTEM
rt_max_prio=99
rt_min_prio=1
Thread idx=0, sum[0...0]=0
Thread idx=0, affinity contained: CPU-0
Thread idx=0 ran 0 sec, 20 msec (20583 microsec)
TEST COMPLETE
root@sowmya-HP-Pavilion-Notebook:/home/sowmya/Downloads/rt_simplethread#
```

In this program, initially, the priority levels, scheduling type and inheritance type parameters are all set and then a thread is created. A print function is used to verify the working of the schedule policy that is set. Inside the 'counterthread' function, a clock_gettime() call is used to get the system time and calculate the execution time of that particular thread. A Fibonacci series is used to provide the delay for the thread.

RT_THREAD_IMPROVED

```
root@sowmya-HP-Pavillon-Notebook: /home/sowmya/Downloads/rt_thread_improved
root@sowmya-HP-Pavillon-Notebook: /home/sowmya/Downloads# cd rt_thread_improved
root@sowmya-HP-Pavillon-Notebook: /home/sowmya/Downloads/rt_thread_improved# make pthread
gcc -O0 -g -c pthread.c
pthread.c: In function 'print_scheduler':
pthread.c:154:35: warning: implicit declaration of function 'getpid' [-Wimplicit-function-declaration]
    schedType = sched_getscheduler(getpid());
                                ^
pthread.c: In function 'main':
pthread.c:183:86: warning: implicit declaration of function 'get_nprocs_conf' [-Wimplicit-function-declaration]
    printf("This system has %d processors configured and %d processors available.\n", get_nprocs_conf(), get_nprocs());
                                                                    ^
pthread.c:183:105: warning: implicit declaration of function 'get_nprocs' [-Wimplicit-function-declaration]
    printf("This system has %d processors configured and %d processors available.\n", get_nprocs_conf(), get_nprocs());
                                                                    ^
gcc -O0 -g -o pthread pthread.o -lpthread
root@sowmya-HP-Pavillon-Notebook: /home/sowmya/Downloads/rt_thread_improved# ./pthread
This system has 4 processors configured and 4 processors available.
number of CPU cores=4
Using sysconf number of CPUS=4, count in set=4
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_FIFO
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 1
CPU-1
Launching thread 1
Setting thread 2 to core 2
CPU-2
Launching thread 2
Setting thread 3 to core 3
CPU-3
Launching thread 3
Thread idx=3, sum[0...400]=79800
Thread idx=3 ran on core=1, affinity contained: CPU-0 CPU-1 CPU-2 CPU-3
Thread idx=3 ran 0 sec, 444 msec (444219 microsec)
Thread idx=2, sum[0...300]=44850
```

```
root@sowmya-HP-Pavillon-Notebook: /home/sowmya/Downloads/rt_thread_improved
This system has 4 processors configured and 4 processors available.
number of CPU cores=4
Using sysconf number of CPUS=4, count in set=4
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_FIFO
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 1
CPU-1
Launching thread 1
Setting thread 2 to core 2
CPU-2
Launching thread 2
Setting thread 3 to core 3
CPU-3
Launching thread 3
Thread idx=3, sum[0...400]=79800
Thread idx=3 ran on core=1, affinity contained: CPU-0 CPU-1 CPU-2 CPU-3
Thread idx=3 ran 0 sec, 444 msec (444219 microsec)
Thread idx=2, sum[0...300]=44850
Thread idx=2 ran on core=2, affinity contained: CPU-0 CPU-1
Thread idx=1, sum[0...200]=19900
Thread idx=1 ran on core=0, affinity contained: CPU-0 CPU-1 CPU-2 CPU-3
Thread idx=1 ran 0 sec, 444 msec (444393 microsec)
Thread idx=0, sum[0...100]=4950
Thread idx=0 ran on core=3, affinity contained: CPU-0 CPU-1 CPU-2 CPU-3
Thread idx=0 ran 0 sec, 444 msec (444503 microsec)
CPU-2 CPU-3
Thread idx=2 ran 0 sec, 444 msec (444346 microsec)
TEST COMPLETE
root@sowmya-HP-Pavillon-Notebook: /home/sowmya/Downloads/rt_thread_improved#
```


In this program, initially, the priority levels, scheduling type and inheritance type parameters are all set and then a thread is created. A print function is used to verify the working of the schedule policy that is set. Inside the 'counterthread' function a clock_gettime() call is used to get the system time and calculate the execution time of that particular thread. A Fibonacci series is used to provide the delay for the thread. Each thread is dedicated to a particular CPU core during its execution. Since the VM had 4 cores dedicated, all threads were run on the 4 cores simultaneously.

EXAMPLE-SYNC

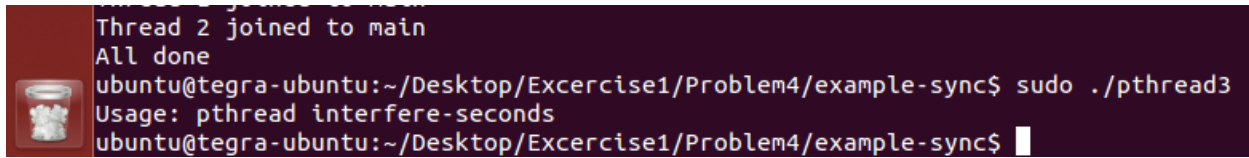
DEADLOCK:

```
ubuntu@tegra-ubuntu: ~/Desktop/Excercise1/Problem4/example-sync
ubuntu@tegra-ubuntu:~/Desktop/Excercise1/Problem4/example-sync$ sudo ./deadlock
[sudo] password for ubuntu:
Will set up unsafe deadlock scenario
Creating thread 1
Thread 1 spawned
Creating thread 2
Thread 2 spawned
rsrcACnt=0, rsrcBCnt=0
will try to join CS threads unless they deadlock
THREAD 2 grabbing resources
THREAD 1 grabbing resources
THREAD 1 got B, trying for A
THREAD 1 got A, trying for B
```

DEADLOCK_TIMEOUT:

```
ubuntu@tegra-ubuntu:~/Desktop/Excercise1/Problem4/example-sync$ sudo ./deadlock_timeout
Will set up unsafe deadlock scenario
Creating thread 1
Creating thread 2
will try to join both CS threads unless they deadlock
Thread 1 started
THREAD 1 grabbing resource A @ 1518060939 sec and 669867033 nsec
Thread 1 GOT A
rsrcACnt=1, rsrcBCnt=0
Thread 2 started
THREAD 2 grabbing resource B @ 1518060939 sec and 670069948 nsec
Thread 2 GOT B
rsrcACnt=1, rsrcBCnt=1
THREAD 1 got A, trying for B @ 1518060940 sec and 670142672 nsec
THREAD 2 got B, trying for A @ 1518060940 sec and 670405920 nsec
Thread 2 TIMEOUT ERROR
Thread 1 GOT B @ 1518060942 sec and 670965116 nsec with rc=0
rsrcACnt=1, rsrcBCnt=1
THREAD 1 got A and B
THREAD 1 done
Thread 1 joined to main
Thread 2 joined to main
All done
```

PTHREAD3:

A terminal window with a dark background. The text shows a program execution where 'Thread 2 joined to main' and 'All done' are printed. The user then runs 'sudo ./pthread3' which outputs 'Usage: pthread interfere-seconds'.

```
Thread 2 joined to main
All done
ubuntu@tegra-ubuntu:~/Desktop/Excercise1/Problem4/example-sync$ sudo ./pthread3
Usage: pthread interfere-seconds
ubuntu@tegra-ubuntu:~/Desktop/Excercise1/Problem4/example-sync$
```

Threading VS Tasking

THREADING	TASKING
In multi-threading, different parts of a program get executed simultaneously	In multi-tasking, different programs get executed simultaneously
All the threads in a process share the same address space	Tasks do not share the same address space. Each tasks run in their own address space
Communication between two threads involve very less overhead	Communication between two tasks has more overheads as it involves system call s and copying data
Errors are shared in threads. Error occurring due to one thread may have its impact on other threads	Errors are not impact full within different task as they do not share the same memory space

- **Semaphores**

Semaphore is an integer variable which is used to manage critical sections and to synchronizes different processes in a multi-processing environment. The value of the semaphore can be modified by two routines in the POSIX standard called `sem_wait` and `sem_post`. Sempahores are of two types, they are binary and counting semaphores. Binary semaphores can have only 1 and 0. Counting semaphores can have non negative values.

- **Sem_wait()**

This function takes the pointer to the semaphores as its argument. The `sem_wait` function decrements the value pointed to by the input argument. It basically acts as a lock to the shared resource. The function decrements the value until its becomes zero. After that it waits for the value to become more than zero or for a signal handler to interrupt.

- **Sem_post()**

This function takes the pointer to the semaphores as its argument. It increments (unlocks) the semaphore pointed to by the argument. If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a `sem_wait()` call will be woken up and proceed to lock the semaphore.

- **Synthetic Workload Generation**

An executable workload model which consists of a set of artificial programs which produce controllable demands on the resources of the system. Synthetic workload generation is a method that confirms the schedulability of a given task in a practical environment by creating a particular sequence of requests based on a workload table specified by the user. The synthetic task set can then be executed by a scheduling simulator to test the observance of the hard timing constraints.

- **Synthetic workload analysis and adjustment on test system**

The aim of such synthetic workload analysis is to test if the systems can be repeatedly run for the given execution time with stability. To achieve such execution times, we are using a Fibonacci computing loop. The Fibonacci loop is fine tuned to achieve execution times of 10ms and 20 ms by changing the sequence iterations in the source code.

```
FIB_TEST(seqCnt, iterCnt)
```

```
for(idx=0; idx < iterCnt; idx++)
```

```
{
```

```
while(jdx < seqCnt)
```

```
{
```

```
if (jdx == 0)
```

```
{
```

```
fib = 1;
```

```
}
```

```
else
```

```
{
```

```
fib0 = fib1;
```

```
fib1 = fib;
```

```
fib = fib0 + fib1;
```

```
}
```

```
jdx++;
```

```
}
```

```
}
```

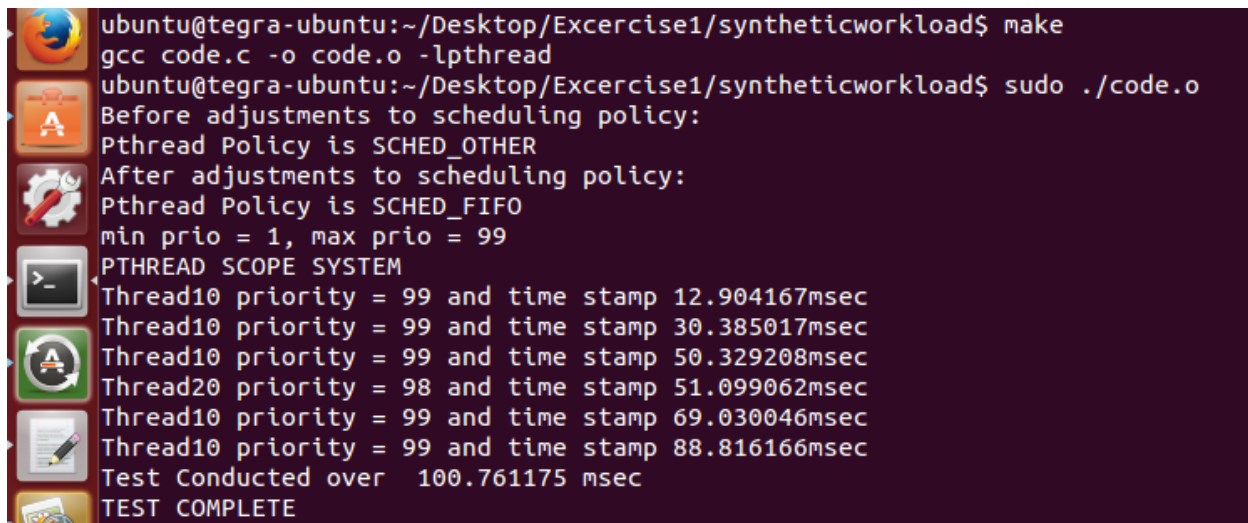
```

usleep(t_20);
sem_post(&sem_t10);
usleep(t_20);
sem_post(&sem_t10);
usleep(t_10);
abortTest_20 = 1;
sem_post(&sem_t20);
usleep(t_10);
sem_post(&sem_t10);
usleep(t_20);
abortTest_10 = 1;
sem_post(&sem_t10);
usleep(t_20);

```

The thread10 is given highest priority and thread20 is given the next highest priority.

	T(ms)	C(ms)	lcm=10							
S1	2	1								
S2	5	2								
RM Schedule	1	2	3	4	5	6	7	8	9	10
S1										
S2										



```
ubuntu@tegra-ubuntu:~/Desktop/Excercise1/syntheticworkload$ make
gcc code.c -o code.o -lpthread
ubuntu@tegra-ubuntu:~/Desktop/Excercise1/syntheticworkload$ sudo ./code.o
Before adjustments to scheduling policy:
Pthread Policy is SCHED_OTHER
After adjustments to scheduling policy:
Pthread Policy is SCHED_FIFO
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM
Thread10 priority = 99 and time stamp 12.904167msec
Thread10 priority = 99 and time stamp 30.385017msec
Thread10 priority = 99 and time stamp 50.329208msec
Thread20 priority = 98 and time stamp 51.099062msec
Thread10 priority = 99 and time stamp 69.030046msec
Thread10 priority = 99 and time stamp 88.816166msec
Test Conducted over 100.761175 msec
TEST COMPLETE
```

The code is attached as a .zip file.

- **Challenges Faced**

A number of challenges were faced, first was understanding how the various function calls, the different types of scheduling policies, priority levels, inheritance concepts etc. worked, and then the second major hurdle was assigning the parameters to the threads while creating them. Each and every line of code was understood, the syntaxes looked up (Linux manual pages) and proper values were passed to ensure full functionality. Selecting the sequence iteration value for the Fibonacci function posed challenges the system produced different output on every execution. The test system works perfectly fine as per scheduling is concerned but it tends to miss one deadline. In the above code, thread20 missed one deadline after 50ms.

REFERENCES

1. <https://www.vox.com/2015/5/30/8689481/margaret-hamilton-apollo-software>
2. https://en.wikipedia.org/wiki/Margaret_Hamilton_%28scientist%29
3. <https://www.hq.nasa.gov/alsj/a11/a11.1201-pa.html>
4. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment – Liu and Layland
5. Real-Time Embedded Components and Systems – Sam Siewart
6. Independent Study - RM Scheduling Feasibility Tests conducted on TI DM3730 Processor - 1 GHz ARM Cortex-A8 core with Angstrom and TimeSys Linux ported on to BeagleBoard xM - Nisheeth Bhat
7. <https://computing.llnl.gov/tutorials/pthreads/>
8. The Rate Monotonic Scheduling Algorithm: Exact Characterization And Average Case Behavior - John Lehoczky, Lui Sha and Ye Ding
9. <http://www.electro.fisica.unlp.edu.ar/temas/p7/HRT/Chapter11.pdf>
10. Operating Systems : Three Easy Pieces - Remzi Arpaci-Dusseau & Andrea Arpaci-Dusseau

