

EXERCISE 1

BY

Amreeta Sengupta and Ridhi Shah

02/05/2019

QUESTION 1

Rate Monotonic Policy tells us that the various tasks are scheduled according to their frequency of occurrence. The task that occurs most frequently gets the highest priority and the task that occurs the least frequently gets the lowest priority. The figure below shows three tasks that are scheduled using Rate Monotonic Policy:

Q1.	T1	3	C1	1	U1	0.34	LCM =	15							
	T2	5	C2	2	U2	0.4									
	T3	15	C3	3	U3	0.2	U _{tot} =	0.94							
RM Schedule	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S1															
S2															
S3															

According to Rate Monotonic Policy,

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

where U = CPU Utility

m = Number of services

C = Computation Time

T = Time period

$$\text{Therefore, Total CPU Utilization} = \frac{1}{3} + \frac{2}{5} + \frac{3}{15} = \frac{4}{15} = 0.94$$

Feasibility and safety

This schedule is feasible as all the three services meet the required deadlines, as shown above. Since CPU Utilization is not less than 30% (it is 94% in this case), this schedule cannot be called safe. Therefore the given schedule is feasible but it is not safe.

Utility and Method

Service S1 comes every 3ms and its computation time is 1ms. So we allot S1 every 3ms. Now service S2 comes every 5ms and its computation time is 2ms. After scheduling S1, we allot first available time slots

to S2 such that it meets its deadline of 5ms. Now service S3 comes every 15ms and its computation time is 3ms. After scheduling S1 and S2, we allot first available time slots to S3 such that it meets its deadline of 15ms.

QUESTION 2

Apollo 11 Summary and Root cause Analysis

The NASA_Apollo_11 Paper tells us about the issues faced during the lunar landing and the constraints the engineers had to work with. The Lunar Guidance computer was equipped with 36,864 15-bit words of fixed memory (ROM) which was used to store most of the executable code and 2048 words of erasable memory (RAM) which was used to store the variable data, counters etc. Since erasable memory was limited, some memory locations were shared several ways which made testing quite challenging. The Lunar Module Descent Engine was developed as a real-time multi-tasking operating system with tasks with interrupts and real-time deadlines to meet. Erasable memory was assigned to each of these tasks during execution, which was used to store the intermediate result. A core set of 12 erasable memory locations were available for each of the tasks and for additional temporary storage, the task had the facility to request for a VAC (vector accumulator) with 44 erasable words. A total of seven core sets and five VAC areas were available for use by the various tasks. If a job needed a VAC area, the operating system was responsible for scanning and finding the one which was ready for use, post which the core sets would be scanned for the same. If the task specified a "NOVAC", the scanning for memory would be skipped. During the descent of Apollo 11 towards the lunar surface, due to a misconfiguration of the radar switches, repeated tasks of processing the random incorrect data lead to an overflow and all the core sets were completely occupied which triggered the 1202 alarm. The task also requested VAC area which resulted in 1201 alarm getting triggered due to overflow of available VAC areas. These two alarms in turn caused a software reboot and reinitialization of the computer which restarted the programs from where they had previously stopped. Every time the alarm appeared, the important tasks like steering the descent engine etc. were restarted, but it did not restart all the incorrectly scheduled radar tasks. Due to extensive testing of the restart capability by NASA, the mission could be carried on.

So, to summarize, the root cause of the problems faced was the erroneous data that was being provided by the faulty radar switches that lead to filling up of the core set and it triggered the 1202 alarm. 1201 alarm was also triggered due to overflow of available VAC areas. This also resulted in a software reboot and all the essential programs again restarted from where they had stopped. Thus, the alarms were basically getting triggered due to non-critical tasks. Hence, they went ahead with the Lunar Descent.

The key assumptions made are:

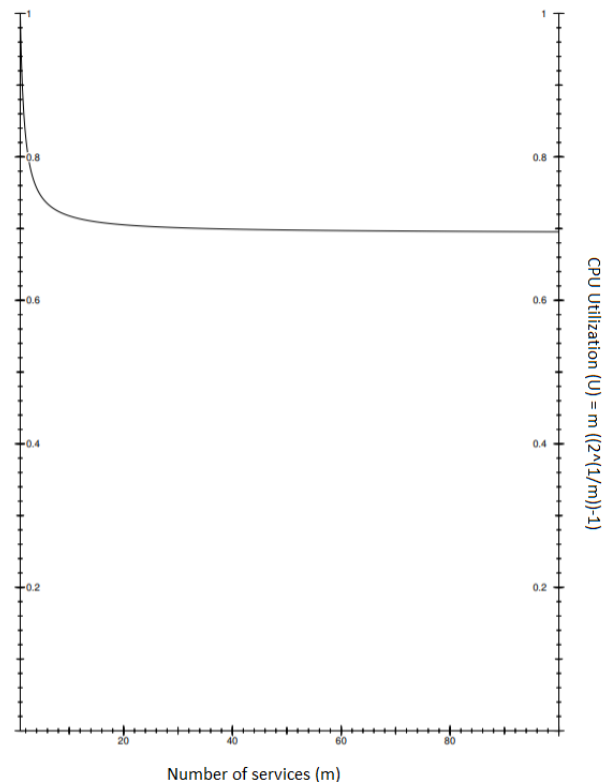
- The requests for all tasks for which hard deadlines exist are periodic, with constant interval between requests.
- Deadlines consist of run-ability constraints only--i.e, each task must be completed before the next request for it occurs.
- The tasks are independent in that requests for a certain task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

The aspects of the LUB derivation that we didn't understand are:

- In the derivation of Theorem 4, we didn't understand the assumptions for computing U' for both the scenarios of $C1 = T2 - T1 + \Delta$ and $C1 = T2 - T1 - \Delta$.
- To find the least bound, equation 4 in Theorem 4 has to be minimized by setting the first derivative to zero. We did not understand the mathematical calculations involved in solving the differential equation.
- We were unable to understand how $U = m(2^{1/m} - 1)$ is obtained after re-substitution of equation 6 in equation 4 in Theorem 4.

Rate Monotonic Least Upper Bound Plot

The figure below shows the plot of CPU Utilization vs the number of services:



This curve exists only when the ratio of request times of any two task from the given task set is less than two. This plot shows the variation of LUB with respect to number of services.

Use of RM Policy in Apollo 11

- According to RM Policy, the task with highest frequency gets the highest priority.
- Use of RM policy in Apollo 11 would not have prevented the overload scenario as it is a fixed priority scheduling policy and it doesn't take into account the variations in execution time of radar calculations.
- These variations in time work because of resource constraint and preemption by error.
- The frequency of occurrence error was so high that RM scheduling policy would not have been a feasible solution.

QUESTION 3

Description of code

The main aim of the code is to determine the time taken to execute a thread / task.

The function `clock_gettime` is used to get the time during which the task was started and the time during which it was completed. Before this, another function called `clock_getres` is used to find the resolution of the clock. This resolution is stored in the `timespec` structure. 4 types of clocks can be implemented in the linux kernel which are `CLOCK_REALTIME`, `CLOCK_MONOTONIC`, `CLOCK_PROCESS_CPUTIME_ID`, `CLOCK_THREAD_CPUTIME_ID`. In the given code, `CLOCK_REALTIME`, which is a system-wide real time clock, is used. The time given by this clock represents seconds and nanoseconds in real time. In the given code, two parameters are passed in the `clock_getres` function. These parameters are `CLOCK_REALTIME` and the address of `rtclk_resolution`, which is a `timespec` structure that stores the resolution of the clock. By observing the output of the code, we can see that the resolution here is 1 nanosecond. After getting the resolution of the real time clock, we execute a task where the kernel has to sleep for 3 seconds, and we have to find the time difference between the starting and the ending of the task. For this, `clock_gettime` function is used. `clock_gettime` collects timestamp when it is called. So before starting the task, we call the function to get the time stamps. Two parameters are passed in this function, the 1st parameter is the type of clock and the second one is a `timespec` structure used to store the seconds and the nanoseconds of the timestamp.

After this, the execution of the task starts. Since the task is to make the kernel sleep for 3 seconds, a function called `nanosleep` is used. This function contains two parameters, the first parameter is a `timespec` structure defined to store the value of the sleep time requested. In this case, the sleep time is initialized to 3 seconds. If an interrupt occurs before the requested sleep time is completed, the kernel wakes up from sleep handles the interrupt and goes back to sleep for the remaining time. This remaining time is the second parameter of the `nanosleep` function, it stores the time remaining in seconds and nanoseconds. The `nanosleep` function executes until the remaining time becomes zero or the sleep count (the number of times the kernel has entered into sleep) is less than the maximum number of sleep calls defined earlier (3 in this case).

Right after the execution of the task, the function `clock_gettime` is again called to get the timestamp. The timestamp is stored in the structure `rtclk_stoptime`, which stores the timestamp in seconds and nanoseconds. The difference between the 2 timestamps stored in `rtclk_stop_time` and `rtclk_start_time` gives the time taken to execute the task. To compute this difference a function called `delta_t` is defined. The start time and stop time are passed in this function and the difference is calculated.

On observing the output of the code, we can see that the scheduling policy of the pthread is displayed. This is done in the main function. A function called `print_scheduler` is called inside the main function. The function is defined such that it gets the policy type by calling the `sched_getscheduler` function from `sched.h` library and prints the policy returned by that function. We can see that when no adjustments are made to the policy, the pthread policy is `SCHED_OTHER` which is the standard round robin time sharing policy. There are other policies like `SCHED_BATCH`, `SCHED_IDLE`, `SCHED_RR` that can be used. Later in the main function we can see that adjustments are made to the scheduling policy and the policy is set to `FIFO` by using the `sched_setscheduler` function. Also, the attributes for the pthread are initialised and the scheduling policy is set. The scheduling policy can be inherited or can be explicitly set. To specify if the policy has to be inherited or will be explicitly set, the function `pthread_attr_setinheritsched` is used. In this case we set the policy as `FIFO` explicitly. Post this, the scheduling policy after adjustments, is displayed. The thread is then created using `pthread_create` where the thread, thread attributes, the function to be executed and the arguments to the function are passed. In this case we pass the `delay_test` function wherein the task to make the kernel sleep is executed and the time difference is computed. We then join the threads after the termination of the execution and then destroy the threads.

Low Interrupt handler latency

Interrupt handler latency is an important factor in Real time embedded systems because it can determine whether the task can be completed before the given deadline. If the interrupt handler latency is very high i.e. if the time spent in the handler is too high due to certain reasons like fetching instructions from different types of memories etc., it slows down the execution of the task and the task will be likely to miss a deadline. Hence this is a critical factor that determines whether the real time system is accurate enough.

Low Context Switch Time

Context switching is the process of saving the state of a process or a thread before switching to another process / thread. The state is saved so that it can be restored while resuming the execution of that thread. Context switching can occur as a result of interrupts or switching of CPU modes. Context switching time varies, depending on whether the context is being switched between 2 threads in a same process or different processes. Usually the time is higher if it is between 2 processes. If this time is too high it affects the performance of the system and the systems becomes slower which can be crucial in real time systems. Hence context switching time is a critical factor in determining the performance of a real time embedded system

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

In real time systems whether the timer services are stable or not is a deterministic factor. The unstable timer services can occur due to variations in the interval between the release times of periodic tasks called

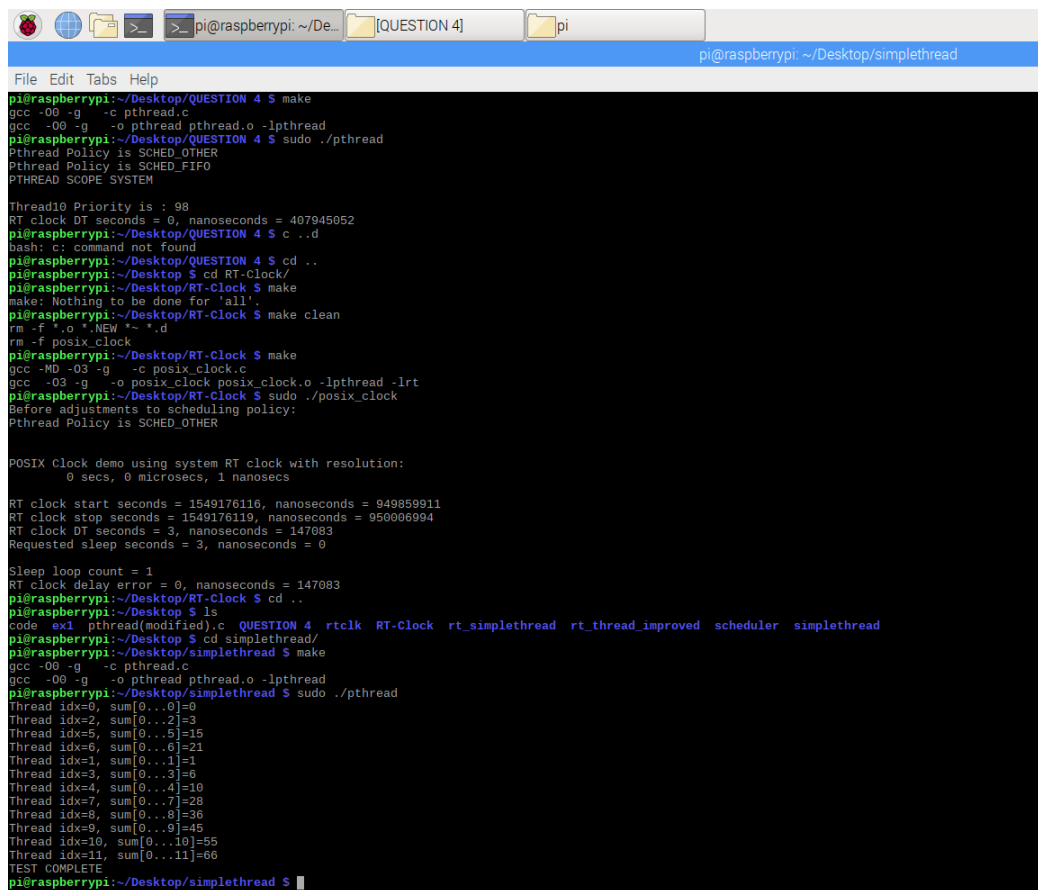
jitter and also due to the amount of drift in the clock over time. These conditions are likely to occur during the interrupts produced by the timers and during timeouts. If jitter and drift are significant then the performance of the real time systems will be affected. Hence low jitter and drift are important characteristics to be considered in real time systems

Accuracy provided by the RT Clock

The RT Clock used in the system is relatively less accurate. The `getclocktime()` function uses `REALTIME_CLOCK` argument which measures time since EPOCH and its accuracy is 1 nanosecond. To measure time between two points in code, most accurate method will be to measure the CPU cycles between these two points. Since we know the frequency of this processor, we can calculate time from these two parameters. Hence, time measurement using `REALTIME_CLOCK` argument is relatively less accurate compared to clock cycle methods.

QUESTION 4

Description of simplethread



```

pi@raspberrypi: ~/Desktop/QUESTION 4 $ make
gcc -O0 -g -c pthread.c
gcc -O0 -g -o pthread pthread.o -lpthread
pi@raspberrypi: ~/Desktop/QUESTION 4 $ sudo ./pthread
Pthread Policy is SCHED_OTHER
Pthread Scope SYSTEM
Thread10 Priority is : 98
RT clock DT seconds = 0, nanoseconds = 407945052
pi@raspberrypi: ~/Desktop/QUESTION 4 $ c ..d
bash: c: command not found
pi@raspberrypi: ~/Desktop/QUESTION 4 $ cd ..
pi@raspberrypi: ~/Desktop $ cd RT-Clock/
pi@raspberrypi: ~/Desktop/RT-Clock $ make
make: Nothing to be done for 'all'.
pi@raspberrypi: ~/Desktop/RT-Clock $ make clean
rm -f *.o *.NEW ~*.d
rm -f posix_clock
pi@raspberrypi: ~/Desktop/RT-Clock $ make
gcc -MD -O3 -g -c posix_clock.c
gcc -O3 -g -o posix_clock posix_clock.o -lpthread -lrt
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

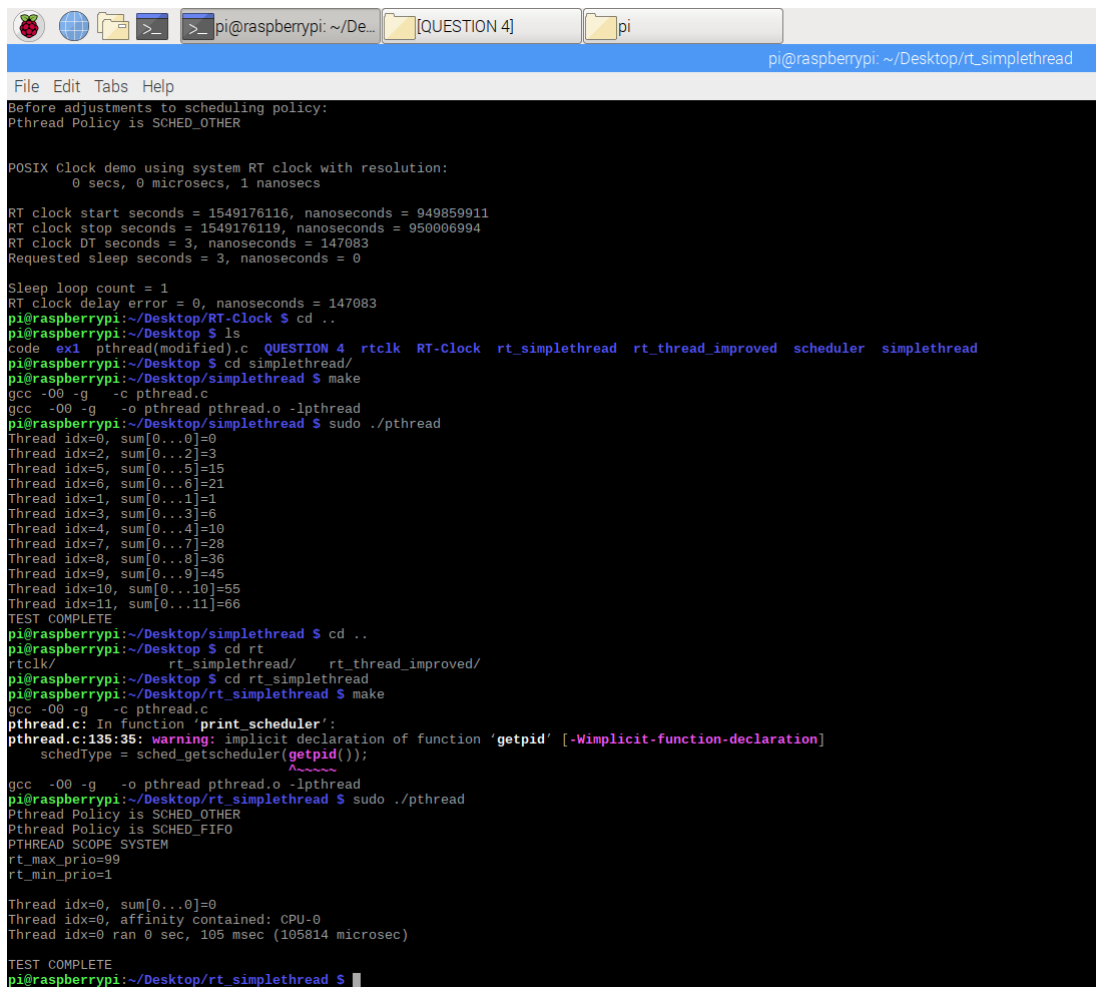
RT clock start seconds = 1549176116, nanoseconds = 949859911
RT clock stop seconds = 1549176119, nanoseconds = 950069994
RT clock DT seconds = 3, nanoseconds = 147083
Requested sleep seconds = 3, nanoseconds = 0

sleep loop count = 1
RT clock delay error = 0, nanoseconds = 147083
pi@raspberrypi: ~/Desktop/RT-Clock $ cd ..
pi@raspberrypi: ~/Desktop $ ls
code  ex1  pthread(modified).c  QUESTION 4  rtclk  RT-Clock  rt_simplethread  rt_thread_improved  scheduler  simplethread
pi@raspberrypi: ~/Desktop $ cd simplethread/
pi@raspberrypi: ~/Desktop/simplethread $ make
gcc -O0 -g -c pthread.c
gcc -O0 -g -o pthread pthread.o -lpthread
pi@raspberrypi: ~/Desktop/simplethread $ sudo ./pthread
Thread idx=0, sum[0...0]=0
Thread idx=2, sum[0...2]=3
Thread idx=5, sum[0...5]=15
Thread idx=6, sum[0...6]=21
Thread idx=1, sum[0...1]=1
Thread idx=3, sum[0...3]=6
Thread idx=4, sum[0...4]=10
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
pi@raspberrypi: ~/Desktop/simplethread $

```

The main aim of this code is to create multiple threads. Here number of threads is defined as 12 and a loop is created to assign the Thread ID. `pthread_create()` function is used in a loop to create 12 threads, each of which execute the `counterThread` function. The following arguments are given to `pthread_create()` : `pthread ID`, the attributes of the thread (here default attributes are used), the function to be executed (`counterThread`) and the parameter to pass in the function. The `counterThread` function executes a task in which the sum is calculated till the `threadID` number which is basically used to kill the CPU time. Next, a print statement is used to display the `threadID` number and the sum for each thread created. `pthread_join` function is used 12 times to stall the program till the respective threads are executed. This completes the test.

Description of RT_simplethread



```

pi@raspberrypi: ~/Desktop/rt_simplethread
File Edit Tabs Help
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 = 1549176116, nanoseconds = 949859911
RT clock stop seconds = 1549176119, nanoseconds = 95006994
RT clock DT seconds = 3, nanoseconds = 147083
Requested sleep seconds = 3, nanoseconds = 0

Sleep loop count = 1
RT clock delay error = 0, nanoseconds = 147083
pi@raspberrypi:~/Desktop/RT-Clock $ cd ..
pi@raspberrypi:~/Desktop $ ls
code  ex1  pthread(modified).c  QUESTION 4  rtclk  RT-Clock  rt_simplethread  rt_thread_improved  scheduler  simplethread
pi@raspberrypi:~/Desktop $ cd simplethread/
pi@raspberrypi:~/Desktop/simplethread $ make
gcc -O0 -g -c pthread.c
gcc -O0 -g -o pthread pthread.o -lpthread
pi@raspberrypi:~/Desktop/simplethread $ sudo ./pthread
Thread idx=0, sum[0...0]=0
Thread idx=2, sum[0...2]=3
Thread idx=5, sum[0...5]=15
Thread idx=6, sum[0...6]=21
Thread idx=1, sum[0...1]=1
Thread idx=3, sum[0...3]=6
Thread idx=4, sum[0...4]=10
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
pi@raspberrypi:~/Desktop/simplethread $ cd ..
pi@raspberrypi:~/Desktop $ cd rt
rtclk/      rt_simplethread/  rt_thread_improved/
pi@raspberrypi:~/Desktop $ cd rt_simplethread
pi@raspberrypi:~/Desktop/rt_simplethread $ make
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
pi@raspberrypi:~/Desktop/rt_simplethread $ sudo ./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, 105 msec (105814 microsec)
TEST COMPLETE
pi@raspberrypi:~/Desktop/rt_simplethread $

```

The aim of this code is to create a thread, configure and set its attributes (priority, number of CPU cores to be used), to execute tasks for computing fibonacci series and lastly to determine the start and end time of the task. Initially the number of threads and the number of CPUs is set to 1.

In the main() function, initially the cpu_set_t object is cleared then the number of CPUs that are specified are added by using CPU_SET. Then the priority of the main thread is set to max and the policy is set to be FIFO. Post this, we check the scope of the attributes of the main thread. The scope can be limited to the process or the entire system. After this, the attributes like priority, scheduling policy and affinity of the CPU are specified for the number of threads defined earlier and then the threads are created using pthread_create. The function to be executed is specified while creating the thread. Here the function is Counterthread(). After the execution of all the threads the pthread_join is called, which waits for all the threads to get terminated and then returns a value 0 on success.

The Countthreads function executes a simple loop that computes the sum by taking the task ID. before the execution. The clock_gettime is called to get the timestamp before execution of the loop. The CPU object is cleared, and the loop starts execution, also another macro to find fibonacci is executed for the number of iterations specified earlier. The computation ends and the values and the number of CPUs are printed. The number CPUs set is determined by using the CPU_ISSET function that gives the number of bits set in the CPU mask. After this, again the timestamp is taken by using clock_gettime. Then, to compute the time difference between the start and stop time, the delta_t function is called.

On observing the output of this code, we see that the policy is SCHED_OTHER. After adjustments to the policy is made, SCHED_FIFO is printed. The scope in this code is limited to the entire system and the max and min priorities of the FIFO policy are 99 and 0. The CPU used is 0 and the time taken to execute the task is 5ns.

Description of RT_thread_improved

```
rm -f pthread
pi@raspberrypi:~/Desktop/rt_thread_improved $ make
gcc -O0 -g -c pthread.c
pthread.c: In function 'print_scheduler':
pthread.c:154:95: 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
pi@raspberrypi:~/Desktop/rt_thread_improved $ sudo ./pthread
This system has 4 processors configured and 4 processors available.
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=1, sum[0...200]=19900
Thread idx=1 ran on core=1, affinity contained: CPU-0 CPU-1 CPU-2 CPU-3
Thread idx=3, sum[0...400]=79800

Thread idx=2, sum[0...300]=44850
Thread idx=2 ran on core=2, affinity contained: CPU-0 CPU-1 CPU-2 CPU-3
Thread idx=2 ran 1 sec, 232 msec (232538 microsec)

Thread idx=0, sum[0...100]=4950
Thread idx=0 ran on core=0, affinity contained: CPU-0 CPU-1 CPU-2 Thread idx=3 ran on core=3, affinity contained:
Thread idx=1 ran 1 sec, 232 msec (232613 microsec)
CPU-0 CPU-1 CPU-2 CPU-3

Thread idx=3 ran 1 sec, 232 msec (232456 microsec)
CPU-3
Thread idx=0 ran 1 sec, 232 msec (232977 microsec)
CPU-0

TEST COMPLETE
pi@raspberrypi:~/Desktop/rt_thread_improved $
```


The main aim of this code is to execute multiple threads by setting different attributes such as scheduling policy, priority etc and configuring the CPU cores. Initially the number of threads is set to 4 and the number of CPU cores is 1.

In the main() function we initially print the number of processors configured by the operating system by using `get_nprocs_conf()` function and the number of available processors by using the `get_nprocs()` function. the `cpu_set_t` object is cleared then the number of CPUs that are specified are added by using `CPU_SET`. Then the priority of the main thread is set to max and the policy is set to be FIFO. After this, we check the scope of the attributes of the main thread. The scope can be limited to the process or the entire system. After this, attributes like priority, scheduling policy and affinity of the CPU are specified for the number of threads defined earlier and then the threads are created using `pthread_create`. Since more than 1 thread has to be executed a for loop is used to set all the attributes, CPU affinity, Scheduling policy, priority and also for the creation of multiple threads. The counterthreads function is passed in the `pthread_create` function and thus all the threads have to execute the counterthreads function.

The countthreads function executes a simple loop that computes the sum by taking the task ID. before the execution. The `clock_gettime` is called to get the timestamp before execution of the loop. The CPU object is cleared, and the loop starts execution, also another macro to find fibonacci is executed for the number of iterations specified earlier. The computation ends and the values and the number of CPUs is printed. The number CPUs set is determined by using the `CPU_ISSET` function that gives the number of bits set in the `cpu_mask`. After this again the timestamp is taken by using `clock_gettime`. Then, to compute the time difference between the start and stop time, the `delta_t` function is called.

On observing the output, we can see that 4 threads with thread IDs 0,1,2,3 are executed, the CPU affinity for each thread is displayed and the core on which the thread ran is displayed. And the time taken to execute each thread is displayed.

Scheduling of two tasks using semaphore

```
ubuntu@tegra-ubuntu: ~/Downloads
1
ubuntu@tegra-ubuntu:~$ ls
1
3.1.0.tar.gz
Desktop
Documents
Downloads
ex5_rtes_vb
Exercise1
Exercise1_Sched.c
EXERCISE3
exercise3_04
Exercise-5-Requirements2018Sp.docx
EXERCISE5_Vtshal
DetPack-L4T-3.2-linux-x64_b196.run
Linux_for_Tegra
master.zip
Music
new_code
object-tracker-master
opencv-3.1.0
ubuntu@tegra-ubuntu:~$ cd Downloads/
ubuntu@tegra-ubuntu:~/Downloads$ ls
Makefile pthread.c Untitled Folder
ubuntu@tegra-ubuntu:~/Downloads$ make
gcc -O0 -g -c pthread.c
gcc -O0 -g -o pthread pthread.o -lpthread
ubuntu@tegra-ubuntu:~/Downloads$ sudo ./pthread
[sudo] password for ubuntu:
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_FIFO
PTHREAD_SCOPE_SYSTEM

Thread0 Priority is : 98
Timestamp = 10 ms ( 10156 us)

Thread0 Priority is : 98
Timestamp = 30 ms ( 30132 us)

Thread20 Priority is : 97
Timestamp = 40 ms ( 40052 us)

Thread0 Priority is : 98
Timestamp = 50 ms ( 50187 us)

Thread0 Priority is : 98
Timestamp = 70 ms ( 70301 us)

Thread0 Priority is : 98
Timestamp = 90 ms ( 90239 us)

Thread20 Priority is : 97
Timestamp = 100 ms ( 100392 us)
ubuntu@tegra-ubuntu:~/Downloads$
```

The main aim of this code is to schedule 2 tasks with periods $T_1=20$ msec and $T_2=50$ msec and with execution times $C_1=10$ msec and $C_2=20$ msec.

To achieve the computation times C_1 and C_2 a fibonacci loop is used. We adjust the number of iterations such that the timing of C_1 and C_2 is achieved.

In the main() function we first initialize the semaphore, semaphore is a variable used to control the access to a common resource shared by multiples threads or processes. The sem_init function present in the semaphore.h file is used for this. This initializes the semaphore at the address pointed by the pointer passed. Another argument is passed to this function which indicates whether the semaphore is to be shared between the threads of a process or between process. After initialisation we set various attributes, policies and priority to the 2 threads. The policies are set as FIFO and the priority for first thread is 98 and for the second thread is 97. After this we configure and set the affinity for the CPU. only one core is used here. We then take the timestamp by using clock_gettime function , 2 threads are then created and a sequencer is created. The 2 threads are created and these threads wait for the semaphore to get incremented. As soon as it gets the semaphore it starts the execution of the fibonacci loop. The time stamp is taken at the end of this to and the delta function is called to find the time taken to execute the thread. The threads run until they are aborted. Then , in the sequencer first the semaphore is given to both tasks and then the kernel is put to sleep for 20 milliseconds. After this the semaphore is given to the 1st task by using sem_post . this increments the semaphore. If the value is greater than 0 then it unlocks the other thread which was in a blocked state because of sem_wait call. After this the kernel again is put to sleep for 20 milliseconds. The task is executed and again the semaphore is incremented and the task1 is started. After this the second task is aborted. Thus the sequencer is accordingly written so that it runs over the period equal to the LCM. then the threads are terminated and pthread_join is called. The semaphores are then destroyed for both the tasks.

Description of key RTOS / Linux OS porting requirements

Threading vs. tasking

The concept of threading is used in Linux OS. This includes processes that contain several threads and there can be multiple number of processes. Thus this model is used in linux , but in Vxworks RTOS there is no such model that has process which contains threads. Everything runs in the task context except the interrupt handler. In tasking multiple tasks can be executed concurrently but while multithreading the CPU switches between tasks. In conclusion , the linux OS has a single memory space unlike Vxworks. In linux we use pthread_create to create threads whereas thread spawn is used in Vxworks. Also, certain calls related to semaphores like sem_post , same_wait are similar to sem_give and same_take in Vxworks.

Semaphores

A semaphore is used to control access to a common resource by multiple processes in a concurrent system. While using semaphores we first initialize it using the sem_init call. Once the semaphores are initialised they have to be given to the tasks to begin the execution. Until then the tasks are waiting for the semaphore. This is done by using semaphore_wait. Sem wait is used to lock the semaphore. if the semaphore has a non-zero value, the value gets decremented to indicate the locking. Thus, by using this sem_wait and sem_post we synchronize the thread.

Synthetic workload generation

Synthetic workload generation refers to the generating of a simple task or loop to achieve the execution time of the tasks. Here we have computed the fibonacci series and executed multiple iterations of it to achieve the required time.

The following code snippet is used to generate workload:

```
#define FIB_TEST(seqCnt, iterCnt) \
    for(idx=0; idx < iterCnt; idx++) \
    { \
        fib = fib0 + fib1; \
        while(jdx < seqCnt) \
        { \
            fib0 = fib1; \
            fib1 = fib; \
            fib = fib0 + fib1; \
            jdx++; \
        } \
    }
```

Synthetic workload analysis and adjustment on test system

As described earlier we have used multiple iterations of fibonacci series to generate the synthetic workload. Here to achieve the computation times of 10msec and 20msec we initially used 1000000 iterations and then determined the time taken for its execution. and then accordingly the number of iterations were increased.

Overall good description of challenges and test/prototype work

RM Schedule	1	2	3	4	5	6	7	8	9	10
S1										
S2										

Here $T1 = 20$ ms and $T2 = 50$ ms and $C1 = 10$ ms and $C2 = 20$ ms

We schedule this task using Rate Monotonic fixed priority policy. Service S1 comes every 20ms and its computation time is 10ms. So we allot S1 every 20ms. Now service S2 comes every 50ms and its computation time is 20ms. After scheduling S1, we allot first available time slots to S2 such that it meets its deadline of 50ms. The challenges faced by us included:

- Understanding the code without prior knowledge of semaphores.
- Trial and testing the iteration count to have an accurate execution time of 10ms and 20ms.

REFERENCES

- <http://ecee.colorado.edu/~ecen5623/ecen/labs/Linux/IS/Report.pdf>
- [liu_layland.pdf](#)
- VxWorks code from Sam Siewert
- NASA_Apollo_11 Paper
- [https://en.wikipedia.org/wiki/Semaphore_\(programming\)](https://en.wikipedia.org/wiki/Semaphore_(programming))