EXERCISE 1

ECEN 5623:

Real-Time Embedded Systems

PRESENTED BY:

Gaurav Gandhi

Sarang Kulkarni

February 4, 2017

INDEX

Solution 1

Diagram:

Following are the simulation of the given data for S1, S2 and S3 using the Cheddar Real-time Scheduling Simulator tool.

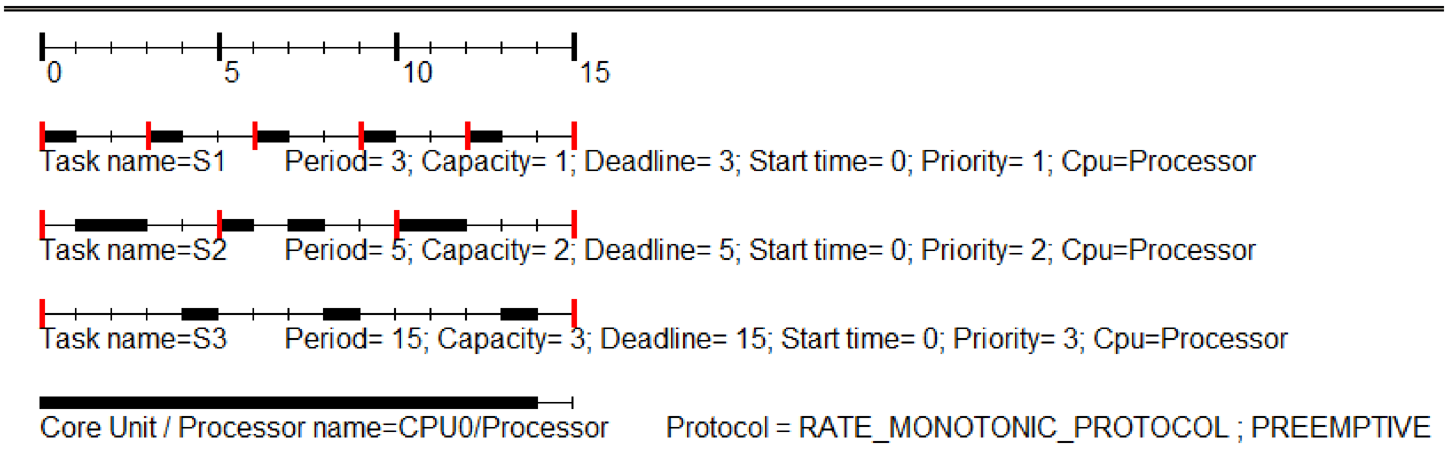
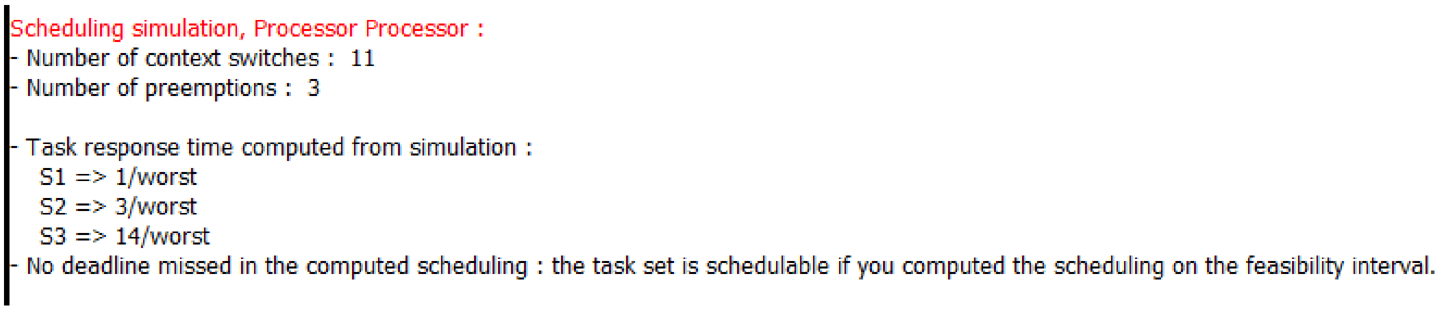
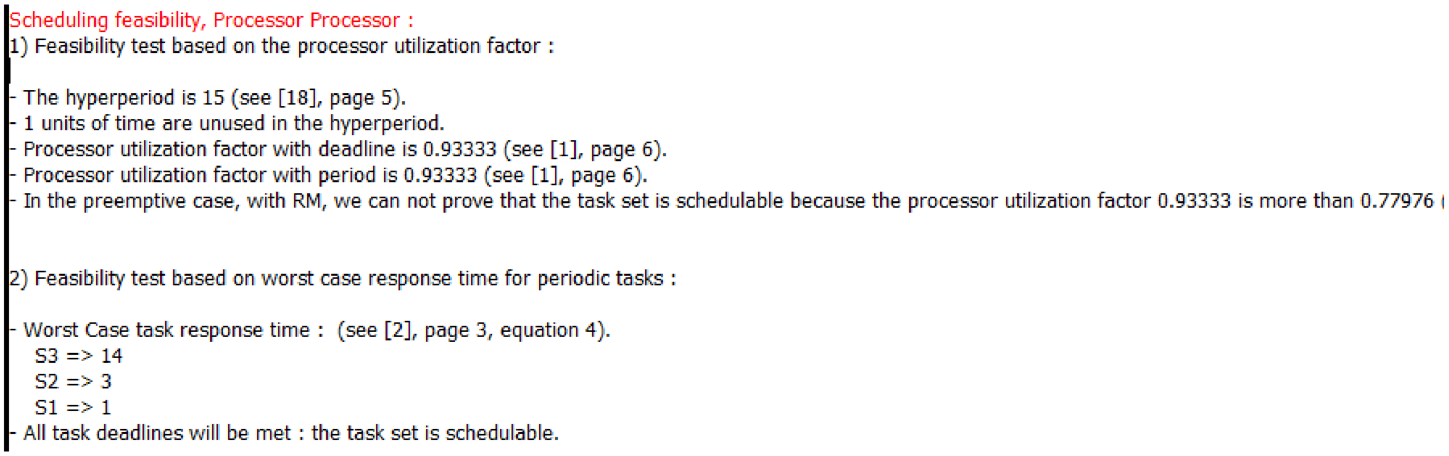

0
5
10
15
Task name=S1 Period= 3; Capacity= 1; Deadline= 3; Start time= 0; Priority= 1; Cpu=Processor
Task name=S2 Period= 5; Capacity= 2; Deadline= 5; Start time= 0; Priority= 2; Cpu=Processor
Task name=S3 Period= 15; Capacity= 3; Deadline= 15; Start time= 0; Priority= 3; Cpu=Processor
Core Unit / Processor name=CPU0/Processor Protocol = RATE_MONOTONIC_PROTOCOL ; PREEMPTIVE

Fig 1. RM Policy Scheduling diagram



```
Scheduling simulation, Processor Processor :
- Number of context switches :  11
- Number of preemptions :  3

- Task response time computed from simulation :
    S1 => 1/worst
    S2 => 3/worst
    S3 => 14/worst
- No deadline missed in the computed scheduling : the task set is schedulable if you computed the scheduling on the feasibility interval.
```

Fig 2. Simulation details for the given data



```
Scheduling feasibility, Processor Processor :
1) Feasibility test based on the processor utilization factor :

- The hyperperiod is 15 (see [18], page 5).
- 1 units of time are unused in the hyperperiod.
- Processor utilization factor with deadline is 0.93333 (see [1], page 6).
- Processor utilization factor with period is 0.93333 (see [1], page 6).
- In the preemptive case, with RM, we can not prove that the task set is schedulable because the processor utilization factor 0.93333 is more than 0.77976 (


2) Feasibility test based on worst case response time for periodic tasks :

- Worst Case task response time :  (see [2], page 3, equation 4).
    S3 => 14
    S2 => 3
    S1 => 1
- All task deadlines will be met : the task set is schedulable.
```

Fig 3. Feasibility analysis for the given processes

Feasibility and safety issues:

As is evident from the above figure the processor utilization is more than the least upper bound for the processor utilization still the processes are schedulable because the periods of the processes are multiples of each other. Though these processes are schedulable and the schedule is mathematically repeatable, it is not safe to use this schedule as according to Least upper bound theorem the CPU utilization should be less than the Rate Monotonic Least Upper Bound for CPU utilization. There is no guarantee that this schedule will never miss a deadline.

The CPU utilization can be calculated as follows:

$U = C1/T1 + C2/T2 + C3/T3$
$= 1/3 + 2/5 + 3/15$
$= 14/15 = 0.9333333$
$= 93.333\%$

Solution 2

Apollo 11 reading summary:
Computer in Apollo 11 (AGC) was Apollo Guidance Computer which had tasks such as Lunar Descent and Lunar Ascent. This computer had huge constraints like 36,864 - 15 it words ROM and 2048 words of RAM. So that 2kb RAM was shared between different tasks at different times. Some of the memory locations were shared 7 ways. The operating system which was controlling all those resources was real time and multitasking. It was interrupt driven, time independent and priority based.

Each task had some part of Ram while executing. There were 12 core sets of erasable memory locations and 5 VAC (vector accumulator) areas. System call were made to schedule a job. If scheduled job needed VAC area, then OS would find available VAC first then core sets. In any case when VAC is not available then 1201 alarm was set. Similarly, if no core sets were available then 1202 alarm would sound.

Root cause analysis:
Root cause of alarm 1202 misconfiguration of RADAR switches. Because of that repeated jobs to process rendezvous radar data were scheduled which was actually not present and it filled core sets and gave 1202 alarm. The 1201 alarm was caused by scheduling request that cased the actual overflow. That scheduling request requested VAC a VAC area causing overload.

It violated rate monotonic policy because fixed priority was not assigned to the tasks. As the RADAR’s expected task frequency was low it should have been assigned low priority. So that other important task would have been able to preempt that task avoiding overload. But this did not happen, in this way it violated rate monotonic policy.

RM LUB plot and description of margin:

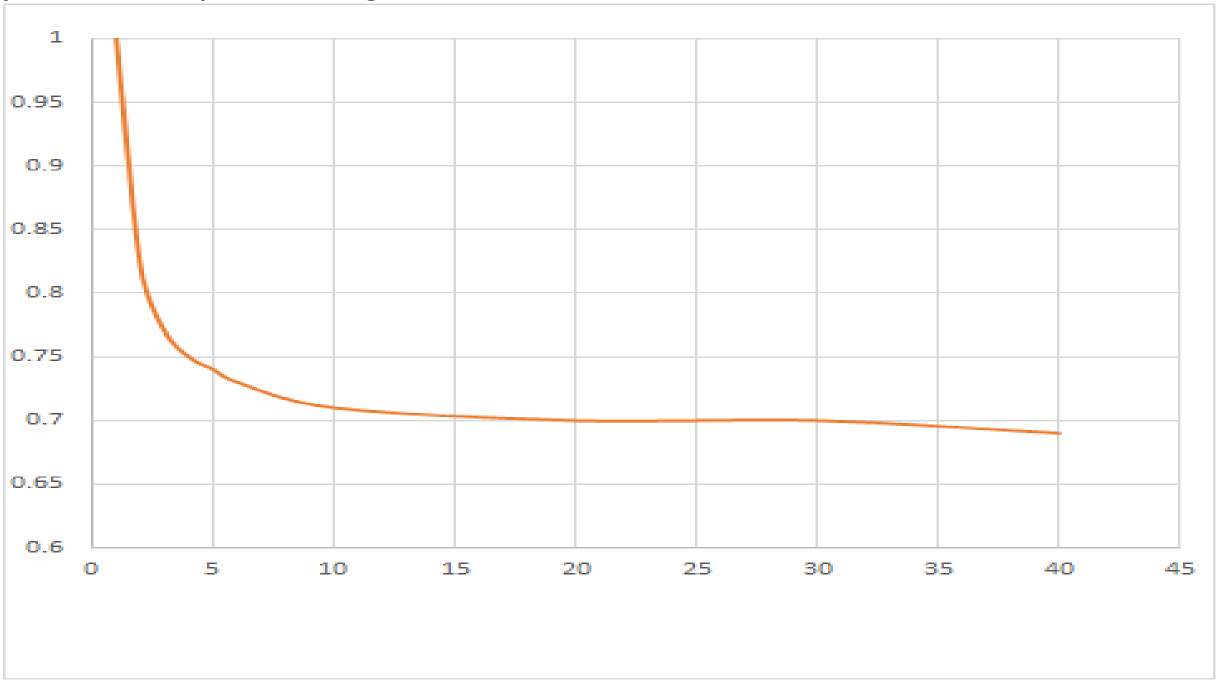


Fig 3. RM LUB plot for the Equation on CPU utility is U = m (2^(1/m) - 1), where m = no. of services

As we can see in the plots the number of processes increases we get the utility of CPU around 70%. So we have the margin of around 30%. This margin is there to avoid CPU overloading in real life scenarios.

Concepts which we didn't understand in Liu &Leyland's Paper are:

1. In theorem 3, there are two cases. They have used some equations for C1 and C2. We could not understand how did they derived those equations
2. Mathematical Derivation of equation U = m(2^(1/m) - 1) in theorem 4.
3. Mathematical Derivation of Theorem 5.

3 Key assumptions in Liu & Layland's paper –

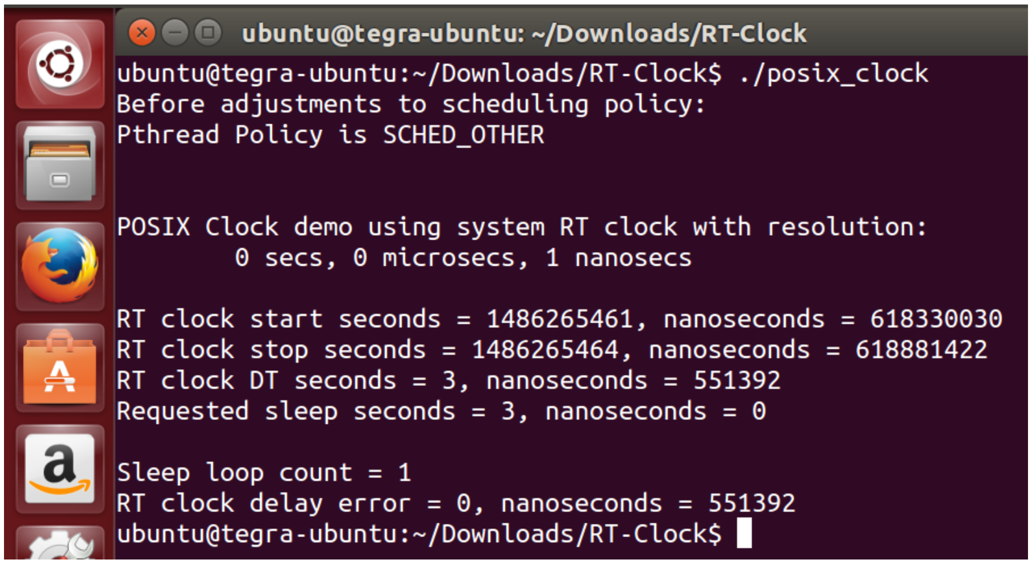
1. The requests for all tasks for which hard deadlines exist are periodic, with constant interval between requests.
2. Deadlines consist of run-ability constraints only--i.e., each task must be completed before the next request for it occurs.
3. Run-time for each task is constant for that task and does not vary with time. Run-time here refers to the time which is taken by a processor to execute the task without interruption.

Arguments for RM Analysis - RM Analysis would have prevented the alarm, if frequency of the radar task occurrence was calculated to be low then according to RM policy it would have been given lower priority. So it would have been pre-empted by other important tasks and thus avoiding overloading.

Argument against RM analysis - RM analysis is based on certain assumptions such as events are periodic and they have fixed execution time. But this is not the case with real world scenarios such as Apollo 11 AGC. So even after doing RM analysis, the alarm 1201/2 still would have been caused.

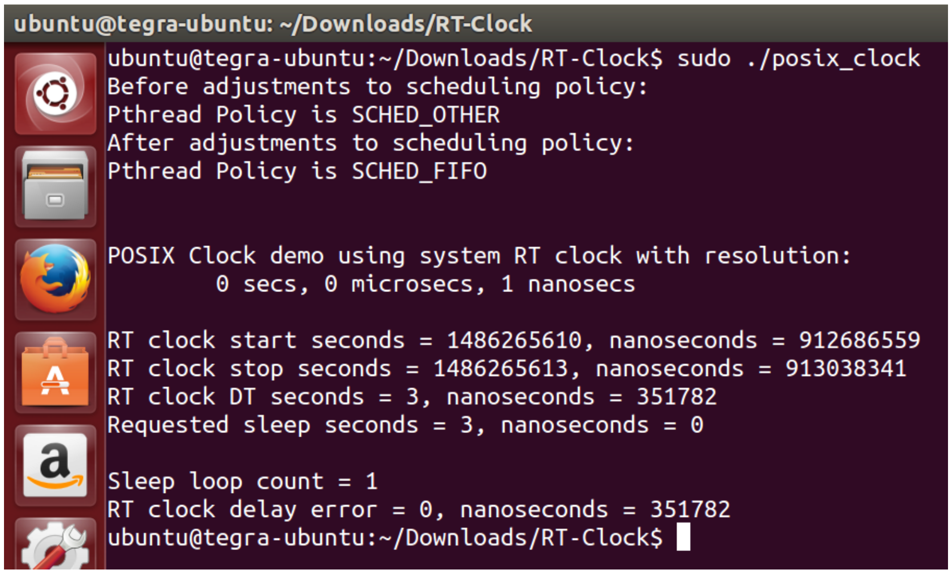
Solution 3

Execution of POSIX RT-Clock code:



```
ubuntu@tegra-ubuntu: ~/Downloads/RT-Clock
ubuntu@tegra-ubuntu:~/Downloads/RT-Clock$ ./posix_clock
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 = 1486265461, nanoseconds = 618330030
RT clock stop seconds = 1486265464, nanoseconds = 618881422
RT clock DT seconds = 3, nanoseconds = 551392
Requested sleep seconds = 3, nanoseconds = 0

Sleep loop count = 1
RT clock delay error = 0, nanoseconds = 551392
ubuntu@tegra-ubuntu:~/Downloads/RT-Clock$
```

Fig 4: Execution of RT-clock code with SCHED_OTHER policy



```
ubuntu@tegra-ubuntu: ~/Downloads/RT-Clock
ubuntu@tegra-ubuntu:~/Downloads/RT-Clock$ sudo ./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 = 1486265610, nanoseconds = 912686559
RT clock stop seconds = 1486265613, nanoseconds = 913038341
RT clock DT seconds = 3, nanoseconds = 351782
Requested sleep seconds = 3, nanoseconds = 0

Sleep loop count = 1
RT clock delay error = 0, nanoseconds = 351782
ubuntu@tegra-ubuntu:~/Downloads/RT-Clock$
```

Fig 5: Execution of RT-clock code with SCHED_OTHER policy

RTOS Bragging points:

1. Low interrupt handler latency - Interrupt latency primarily refers to the amount of time that elapses from the time an external interrupt arrives at the processor and till the time the processing of this interrupt begins (i.e. ISR begins). One of the most crucial ability of a real-time operating system is to service these interrupts within a specified amount of time. Low interrupt latency helps the processor to meet the deadlines in case of a heavy workload. So, the ability of an RTOS to handle interrupts in less amount of time is crucial to meet the deadlines and also for deadlines to be deterministic for any hard real-time system.

2. Low context switch time - The process of context switch refers to the storing of the state of a process or thread in order to resume the execution from the same point at a later time. This process allows for multiple threads to run on a single CPU. Context switch is an overhead for the processor as the processing power is utilized in irrelevant task of storing data. For a real-time system low context switch time is important in order to meet the deadlines effectively. Low context switch time also indicates that most of the processing power is utilized for the tasks rather than context switching. In some systems context switching is done using dedicated hardware like a DMA controller to offload some work from the processor.
3. Low jitter for timer services: Real time systems depend upon clock ticks provided by a timer to schedule event. So timer is a crucial part of the real time system. In such system most of the services are scheduled for accurate timing intervals, jitter in the timing mechanism can overthrow these services and can cause the system to miss deadlines. So, jitter and drift in timer interrupts, timeouts, and relative time can compromise deterministic completion of deadlines and repeatability of the tasks.

Description of Code:
The RT-Clock code exploits the pthread and timing libraries provided by the Linux operating system to measure the time of execution of the threads to nanosecond precision. In the given code the delay_test (void *threadID) function uses the function nanosleep() to start a 3 second sleep cycle. The accuracy of this delay is profiled using the function clock_gettime(), this function return the value of system clock. This function is called at the start and at the end of the sleep cycle in order to measure the exact time the sleep was performed.

The clock_gettime() function gets the current time of the clock specified by clock_id, and puts it into the structure pointed to by tp. The only supported clock ID is CLOCK_REALTIME. The argument tp is a timespec struct.

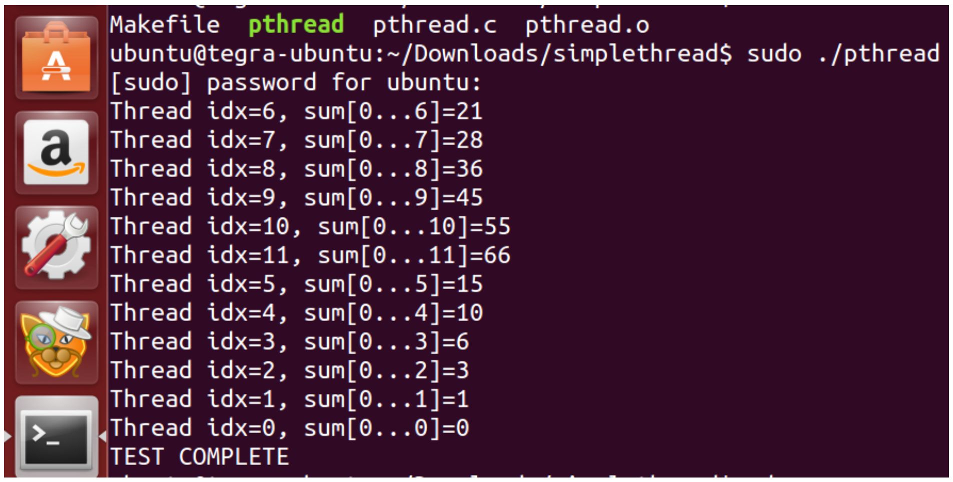
```
struct timespec {
time_t tv_sec;          /* seconds */
long tv_nsec;           /* nanoseconds */
};
```

Code gives option of executing single thread using 2 scheduling policies by using compile time switch. 1.SCHED_OTHER and 2. SCHED_FIFO. In SCHED_OTHER policy, the scheduling of thread is decided by the OS. So for that no other attributes such as priority are passed to the thread. While in case of SCHED_FIFO, thread is created using pthread_create function with assigning attributes. Pthread join function is used to keep main process alive till the execution of the thread is complete.

Accuracy of RT-clock:
The delay in RT_Clock code is up to 0.0183% for SCHED_OTHER and It is up to 0.0117% for SCHED_FIFO for given 3 sec sleep code. If our tolerance for delay is within this range, then the accuracy provided by RT_CLOCK code can be trusted. If some Hard real time systems need less delay than that then this RT_CLOCK cannot be trusted.

Solution 4

Analysis of simplethread code:

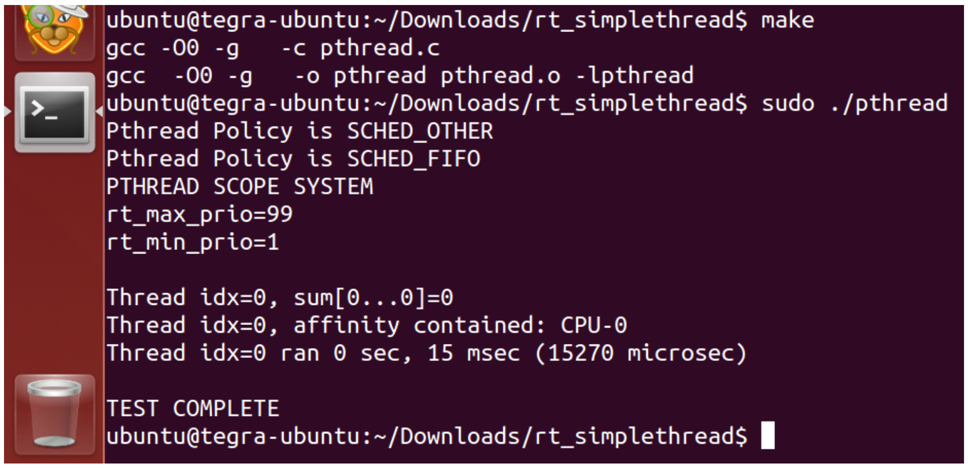


```
Makefile  pthread  pthread.c  pthread.o
ubuntu@tegra-ubuntu:~/Downloads/simplethread$ sudo ./pthread
[sudo] password for ubuntu:
Thread idx=6, sum[0...6]=21
Thread idx=7, sum[0...7]=28
Thread idx=8, sum[0...8]=36
Thread idx=9, sum[0...9]=45
Thread idx=10, sum[0...10]=55
Thread idx=11, sum[0...11]=66
Thread idx=5, sum[0...5]=15
Thread idx=4, sum[0...4]=10
Thread idx=3, sum[0...3]=6
Thread idx=2, sum[0...2]=3
Thread idx=1, sum[0...1]=1
Thread idx=0, sum[0...0]=0
TEST COMPLETE
```

Fig 5. Output of Simplethread code

- This code generated 12 threads using pthread_create function.
- No scheduling policy is used. Default scheduling policy SCHED_OTHER is used.
- Each thread is just a sum of all the numbers till thread index.
- No attributes are passed to the to the threads.
- Pthread_join function is used in the end.
- Because of pthread_join function main process wont exit unless execution of all the threads is not complete.

Analysis of rt_simplethread:

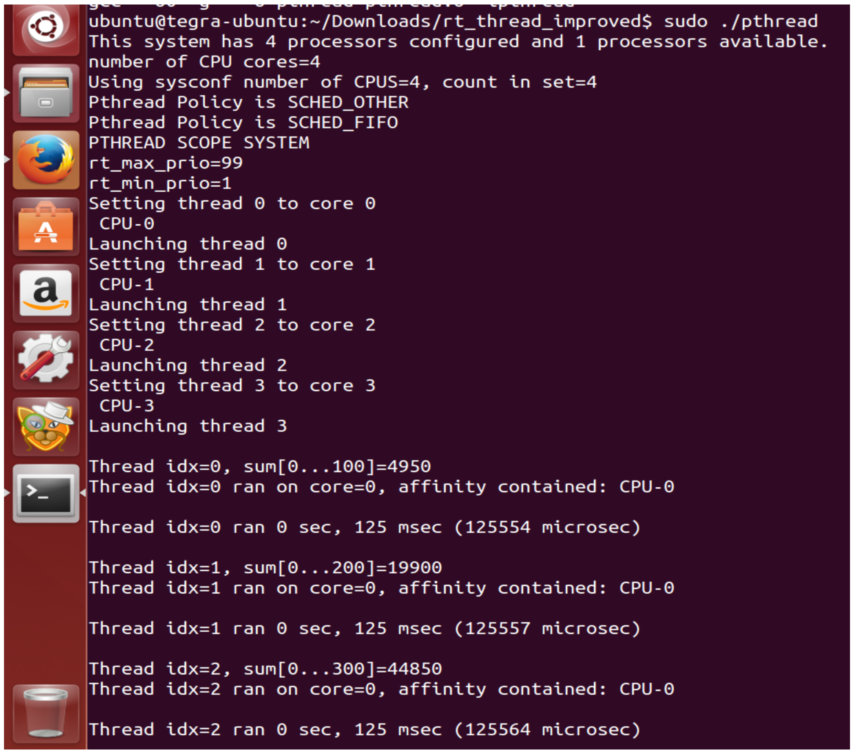


```
ubuntu@tegra-ubuntu:~/Downloads/rt_simplethread$ make
gcc -O0 -g   -c pthread.c
gcc  -O0 -g   -o pthread pthread.o -lpthread
ubuntu@tegra-ubuntu:~/Downloads/rt_simplethread$ sudo ./pthread
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_FIFO
PTHREAD SCOPE SYSTEM
rt_max_prio=99
rt_min_prio=1

Thread idx=0, sum[0...0]=0
Thread idx=0, affinity contained: CPU-0
Thread idx=0 ran 0 sec, 15 msec (15270 microsec)

TEST COMPLETE
ubuntu@tegra-ubuntu:~/Downloads/rt_simplethread$
```

Fig 6. Output of Simplethread code

- This code contains a single thread which is using SCHED_FIFO Scheduler
- This thread is running on CPU core 0.
- Maximum priority that can be assigned using SCHED_FIFO is 99 while minimum is 1.
- Execution time of the thread is profiled using delay_test function which uses function gettime.
- Execution time is 15270 microseconds.

Analysis of rt_thread_improved:



```
ubuntu@tegra-ubuntu:~/Downloads/rt_thread_improved$ sudo ./pthread
This system has 4 processors configured and 1 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=0, sum[0...100]=4950
Thread idx=0 ran on core=0, affinity contained: CPU-0

Thread idx=0 ran 0 sec, 125 msec (125554 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, 125 msec (125557 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, 125 msec (125564 microsec)
```

Fig 7. Output of the rt_thread_improved

- This code creates 4 threads which run on 4 different CPU cores.
- Attributes are used to assign every thread to different CPU core.
- Execution time of each thread is
    - Thread 0: 125554 microsecond
    - Thread 1: 125557 microsecond
    - Thread 2: 125564 microsecond

Threading vs Tasking: A thread has its own stack and kernel resources. Thread allows the highest degree of control. Thread can be aborted, suspended or resumed. Threading adds additional CPU overhead as the processor adds additional CPU overhead as the processor context-switch between threads.
A task is a set of instructions loaded in the memory. A task does not create its own OS thread. Instead, tasks are executed by a Task Scheduler. A task represents some asynchronous operation. It's a set of APIs for running tasks asynchronously and in parallel.

Semaphores wait and sync:
A semaphore is a variable or abstract data type that is used to control access to a common resource by multiple processes in a concurrent system such as a multiprogramming operating system

sem_wait(sem_t *sem) : It locks the semaphore pointed to by *sem*. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement.

sem_post(): It unlocks the semaphore pointed to by sem. If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a sem_wait(3) call will be woken up and proceed to lock the semaphore.

Using sem_wait() and sem_post() we can control allocation of semaphores to any thread and thus achive synchronization of those threads.

Synthetic workload generation: It is generating fixed load on the operating system. In this technique a particular thread is kept in execution for a certain period of time. In the VxWorks code example; FIB_TEST (seqCnt, iterCnt) is used for load generation.

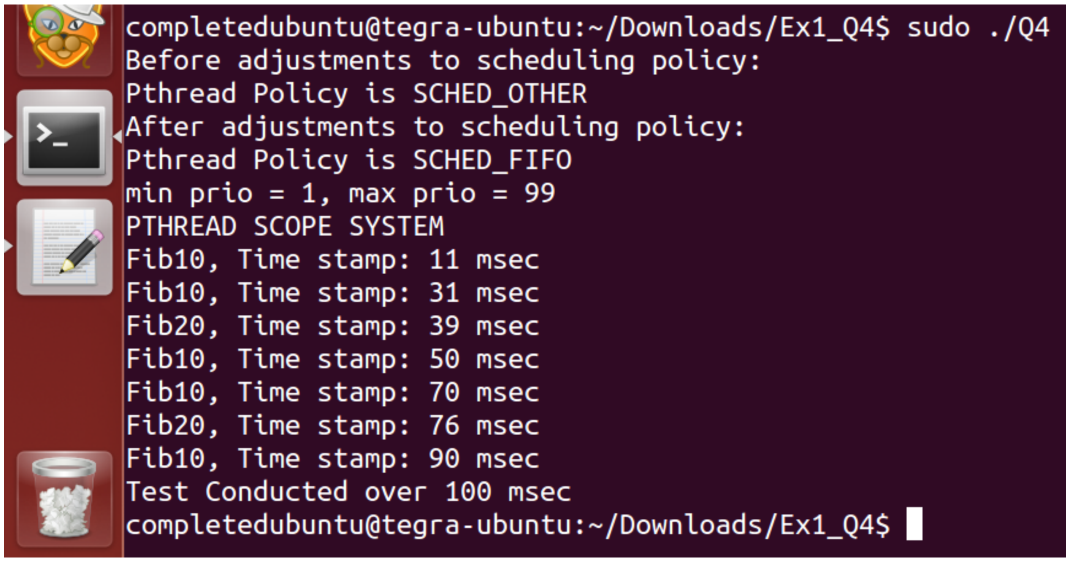
Synthetic workload analysis:

In our code we used FIB_TEST (seqCnt, iterCnt) function from the VxWorks code.That function is defined as follows.

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

In the main process two threads, fib10 and fib20 were created. These threads are supposed to have an execution timing of 10 milliseconds and 20 milliseconds respectively. Attributes for these threads like scheduler, scheduling policy, priority and inheritance were set using the functions:
Enter code here

So for generating particular amount of delay we have 2 variables. By adjusting iterCnt and seqCnt we can change load produced. To achieve the execution timing of 10 milliseconds and 20 milliseconds for the threads we changed the variable iterCnt to 1304000 and 2208000 respectively for fib10 and fib20 threads. To measure the execution timings of the threads we used the function provided in the RT-Clock example namely, clock_getime() function to get the current value of time and delta_t() function to calculate the time elapsed since the start.



```
completedubuntu@tegra-ubuntu:~/Downloads/Ex1_Q4$ sudo ./Q4
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
Fib10, Time stamp: 11 msec
Fib10, Time stamp: 31 msec
Fib20, Time stamp: 39 msec
Fib10, Time stamp: 50 msec
Fib10, Time stamp: 70 msec
Fib20, Time stamp: 76 msec
Fib10, Time stamp: 90 msec
Test Conducted over 100 msec
completedubuntu@tegra-ubuntu:~/Downloads/Ex1_Q4$
```

Fig 8: Output of the code for synthetic load

Challenges:

- Understanding and implementation of semaphores and pthreads.
- Porting code from VxWorks to our Linux Environment
- Getting the correct seqCnt and iterCnt values used in FIB_TEST function to get 10ms and 20ms load for Jetson board

Observations: While executing the code for synthetic load we found that the execution time for load varies with every execution. CPU utilization for this code is 90% and LUB is 82%, So deadlines might not meet for every execution.

Appendix A: References

1. **Nvidia Jetson TK1 user's guide: http://developer.download.nvidia.com/embedded/jetson/TK1/docs/2_GetStart/Jeston_TK1_User_Guide.pdf**
2. **Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment - C. L. Liu and James W. Layland**
3. **VxWorks code by Dr Sam Siewarts: http://mercury.pr.erau.edu/~siewerts/cec450/code/VxWorks-sequencers/lab1.c**
4. **Independent study by Nisheeth Bhat: 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**
5. **Lawrence Livermore National Labs reference on Linux pthreads.**
6. **Linux man pages**

Appendix B: Group Members

Gaurav Gandhi

Graduate Student [ECEE]

University of Colorado, Boulder

Gaurav.gandhi@colorado.edu

+1-720-755-8812

Sarang Kulkarni

Graduate Student [ECEE]

University of Colorado, Boulder

sarang.kulkarni@colorado.edu

+1-720-755-8823