Sorabh Gandhi
Roberto Baquerizo
ECEN 5623
January 27, 2020

Exercise 1

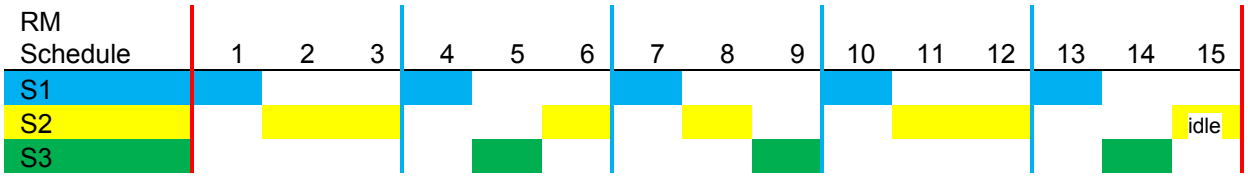
Problem 1

According to the Rate Monotonic Policy, we must assign a priority to each task according to its period. The shorter the period of the task, the higher the priority assigned.
We are given three tasks S1, S2, and S3 with computation times and request periods:

- For S_1: C_1=1 and T_1=3.
- For S_2: C_2=2 and T_2=5.
- For S_3: C_3=3 and T_3=15.

Given the periods of the three tasks (S_1, S_2, and S_3), we determine the order of priority prio(S_1) > prio(S_2) > prio(S_3), since $T_1 < T_2 < T_3$. After assigning priorities, we must determine the least common multiple of the periods. Given that T_1=3, T_2=5 and T_3=15, the LCM of these periods is 15. Therefore, we the timing diagram for the RM Schedule must span 15 milliseconds.

The timing diagram below shows the resulting RM schedule. Each task must complete, i.e. run for their computation times, inside one period in order to be feasible. The colored blocks indicate the currently executing service, and only one service runs at a time. For example, at the critical instant (time 0), we schedule S_1 since it is the highest priority task and we fill in the time block with light blue. Because its computation time (C_1) is 1 millisecond, the service completes, and we can schedule the next lowest priority task S_2. S_2 has a computation time of 2 milliseconds, so it must run for two time blocks. We fill in the blocks from 2 to 3 in yellow. Next, at time 3 milliseconds, S_1 is due to run again, because its period is 3 milliseconds. It is the highest priority task, so it must be scheduled again. Finally, we can schedule S_3 at time 4 milliseconds, because S_2 already completed execution for its time period and is not due again yet. We continue this until time 15 milliseconds.



Because we were able to schedule the three services over the LCM time period without missing a deadline, the schedule is feasible (mathematically repeatable as an invariant indefinitely). To determine if it is safe, we calculate the Rate Monotonic Least Upper Bound from Liu and Layland:

Sorabh Gandhi
Roberto Baquerizo
ECEN 5623
January 27, 2020

$$U = \sum_{i=1}^{m} \frac{c_i}{T_i} \leq m\left(2^{(\frac{1}{m})} - 1\right)$$

So, plugging in for the number of services m = 3, we for the least upper bound CPU utilization: $3\left(2^{(\frac{1}{3})} - 1\right) = 0.7797$, or 77.97%.

CPU utilization for each service is defined as the computation time divided by the request period.

- For S_1, the utilization is $\left(\frac{C_1}{T_1}\right) = 1/3 = 0.333$.
- For S_2, the utilization is $\left(\frac{C_2}{T_2}\right) = 2/5 = 0.4$.
- For S_3, the utilization is $\left(\frac{C_3}{T_3}\right) = 3/15 = 0.2$.

And for the total CPU utilization, we sum up the individual utilizations computed above:

$$U = \left(\frac{C_1}{T_1}\right) + \left(\frac{C_2}{T_2}\right) + \left(\frac{C_3}{T_3}\right) = \left(\frac{1}{3}\right) + \left(\frac{2}{5}\right) + \left(\frac{3}{15}\right) = 0.9333$$

The total CPU utilization by the three services is then 93.3%. Because this value is larger than the RM LUB, the schedule fails the sufficient RM LUB feasibility test. However, we showed by the timing diagram that the schedule is feasible, since no deadlines were missed over the LCM request period. Although feasible, the schedule may not be safe, i.e. unlikely to ever miss a deadline, because the CPU utilization is so high at 93.3%. Any overloading, or period jitter during actual operation may cause some deadlines to be missed.

Sorabh Gandhi
Roberto Baquerizo
ECEN 5623
January 27, 2020

PROBLEM 2

In the NASA account by Peter Adler, he recounts his work on the LEM guidance computer. He describes how the more experienced people concentrated on the Command Module Computer software, while he and his friend, Don Eyles, were responsible for programming the LM powered-flight routines. He concentrated on what he identified as the LM P40's, which were all the flight routines aside from Don Eyles' P60's (Lunar Descent) and P12 (Lunar Ascent). Mr. Adler describes the technological constraints they had to work with when programming the LGC. What we now refer to as ROM was known as "Fixed" memory, and it had a capacity for 36,864 15-bit words (about 72 KB) As for RAM, they called that "Erasable" memory and they only had 2,048 words (less than 4 KB) to work with. Under these conditions, they had to use the same memory address for different purposes at different times. They had to perform extensive testing in order to prove to themselves that no memory location was being used by more than one program at the same time. The memory was allocated to each job in terms of core sets and vector accumulators (VAC). A "core set" was a set of 12 erasable memory locations, and there was a total of seven core sets. If a job required more temporary storage, it could request a VAC, which was an area of 44 erasable words. There was a total of five VAC areas. When a task to be scheduled required a VAC area, the operating system scanned the five VAC areas to find an available one. Once a VAC had been reserved for the job, the operating system would then scan the seven core set areas to find an available one. Scanning of the VAC areas could be skipped if the scheduling request specified "NOVAC". The two 1201 and 1202 alarms were related to the availability of VAC areas and core sets: if no VAC areas were available, the system would raise a 1201 alarm; if no core sets were available, the system would raise a 1202 alarm.

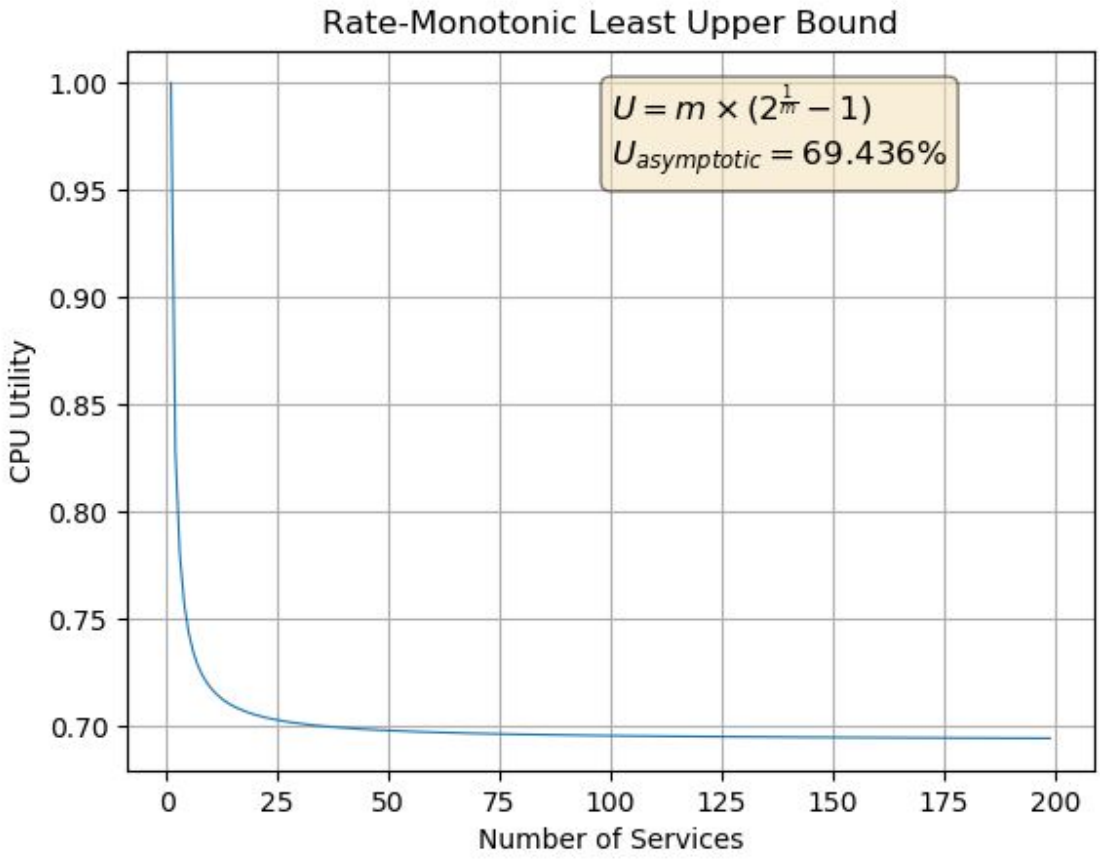
The reason the alarms were raised during the Apollo 11 flight was due to an overload in the system due to faulty configuration of radar switches: repeated jobs to process rendezvous radar data were scheduled as a result of the misconfiguration. Due to these repeated requests, the core sets were filled up, and the software raised the 1202 alarm. The scheduling request that caused the overflow was one that had requested a VAC area. And this request led to the 1201 alarm being raised during the landing. This failure mode as accounted for in software: the system was designed such that it would prioritize more important tasks (such as steering) over the less-important rendezvous radar data. In order to achieve this, however, the computer had to reboot. So, each time the alarms 1201 or 1202 were raised, the computer would reboot, and restart all the important tasks, but not the rendezvous radar jobs that were causing the alarms. Because this restart capability had been tested extensively, the team was confident that the mission could proceed safely.

The rendezvous radar jobs were scheduled due to misconfiguration of the radar switches. The radar jobs were lower priority, but frequent, and the radar sent

a lot of data to the computer based on random electrical noise. This noise was due to an incompatibility in the power supplies for the rendezvous radar and the computer-aided guidance system. The data from the radar jobs eventually overloaded the CPU; there were no resources available for other tasks. Because the system was able to detect the overload on the CPU and reboot itself, and then restarting only the highest-priority and necessary tasks, the mission completed successfully.

According to Rate Monotonic policy, priorities are assigned according to service request periods: the less frequent a task, the lower the priority it is assigned, and the highest priority is assigned to the task that occurs most frequently. The Apollo 11 mission violated the policy by assigning a lower priority to the frequent tasks of the rendezvous radar calculations.

The figure below shows a plot of the Rate Monotonic Least Upper Bound, as derived by Liu and Layland, as a function of the number of services, m. As the number of services increases, we see that the RM LUB CPU utilization tends towards about 70%.



Sorabh Gandhi
Roberto Baquerizo
ECEN 5623
January 27, 2020

Three key assumptions made by Liu and Layland in their paper are:

- The requests for all tasks for which hard deadlines exist are periodic, with constant intervals between requests.
- Deadlines consist of run-ability constraints only. Each task must be completed before the next request for that task occurs again.
- Any nonperiodic tasks in the system are special; they are initialization or failure-recovery routines; they displace periodic tasks while they themselves are being run, and do not themselves have hard, critical deadlines.

Throughout the course of Liu and Layland's Fixed Priority Least Upper Bound derivation, there were three aspects that I did not understand:

- I did not understand how equation $C_1 + C_2 \leq T_1$ (2) implies $\lfloor T_2/T_1 \rfloor \times C_1 + C_2 \leq T_2$ (1) after the proof of Theorem 1, prior to stating Theorem 2.
- In the proof of Theorem 3, Liu and Layland consider two cases for the relationship between the run-times and periods for two tasks. I did not understand how the relation for Case 2 arose $C_1 \geq T_2 - T_1 \times \lfloor T_2/T_1 \rfloor$.
- In the proof of Theorem 4, Liu and Layland write down an equation for the run-time of the m^{th} task: $C_m = T_m - 2(C_1 + C_2 + \ldots + C_{m-1})$. I did not understand where that relation was used in the subsequent derivation steps.

As Rate Monotonic policy assigns priority according to the frequency of a task, with higher frequency tasks being higher priority, RM Analysis would have resulted in the less-important radar jobs being assigned a very high priority. The erroneously-scheduled radar jobs which were occurring at very high rates would have been given a higher priority than the important, but less frequent task of engine steering. For that reason, RM Analysis would not have prevented the Apollo 11 1201 and 1202 alarms and a potential mission abort.

Sorabh Gandhi
Roberto Baquerizo
ECEN 5623
January 27, 2020

PROBLEM 3

The RT-Clock program provided attempts to run a timer for a specified amount of time. The time is set in the code, and it is 3 seconds. An explanation of the code follows:

The main execution first prints the scheduling policy being used by the process. The print_scheduler() function accomplishes this by calling the C-function sched_getscheduler(). The sched_scheduler() function determines the scheduling policy applied to the process identified by the process ID that is passed in. The PID is retrieved by a call to getpid() that returns the PID of the current process. In this case, the scheduling policy associated with our process is SCHED_OTHER, which is the default Linux time-sharing scheduler. From the Linux man pages: "**SCHED_OTHER** is the standard Linux time-sharing scheduler that is intended for all threads that do not require the special real-time mechanisms."

After the call to print_scheduler(), there is some code that is conditionally compiled by the preprocessor directive #ifdef. If the macro RUN_RT_THREAD is defined during compilation, then the program would execute the code in between the #ifdef and #else. However, the code was not compiled with that macro defined, so the code executes delay_test() instead.

The delay_test() function performs the timer test. It first determines the resolution of the clock by calling clock_getres(), which takes a clock id of type clockid_t. In this program, clock_getres() is called with CLOCK_REALTIME to get the resolution of the Linux real-time clock. If successful, the resolution is printed. It turns out that the resolution of the RT clock is 1 nanosecond. After finding and printing the resolution of the clock, the program sets the amount it should delay execution for, which is 3 seconds. Before pausing execution by calling nanosleep(), the program gets the current time on the RT clock by calling clock_gettime(). Right after execution is resumed, the program calls clock_gettime() again to get the stop time. From these two time values, the program calculated the actual time elapsed by the thread in sleep. These calculations are performed in the delta_t() function. Next, the accuracy of the RT clock is calculated by comparing the actual delta-time (stored in rtclk_dt of type struct timespec) and the sleep time requested (stored in sleep_requested also of type struct timespec). The results are then printed to the terminal by the end_delay_test() function.

The clock_gettime() function retrieves the time of a clock identified by the clk_id. The clk_id identifies a particular clock on which to act. A clock may be system-wide, or per-process. In our case, we were interested in CLOCK_REALTIME which is a system-wide real-time clock. The value returned by clock_gettime() represents the time in seconds and nanoseconds since the Epoch. Another example of a system-wide clock is CLOCK_MONOTONIC, which

measures the number of seconds that system has been running since it was booted.

We can use clock_gettime() to determine the execution time of a particular section of code by following these general steps:

1. Define and declare two variables of type struct timespec. One will hold the start time, and the other the stop time. Make sure the members of both structures are initialized to 0.
2. Store the time of the clock at the start of execution by calling clock_gettime(CLOCK_REALTIME, &start_time).
3. At the end of the code section you are interested in, call clock_gettime(CLOCK_REALTIME, &stop_time). This will store the time of the clock at the end of execution.
4. Calculated the elapsed time by comparing the values in start_time and stop_time.

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, timouts, and knowledge of relative time has low jitter and drift.

The **interrupt handler latency** is the time that elapses from when an interrupt is generated to when the interrupt service routine (ISR) corresponding to that interrupt begins executing. A low interrupt handler latency helps in reducing the response time of a system, and can also help achieve determinism in system response time. When working with hard real-time systems, a short response time can help in ensuring critical deadlines are met. Because a low interrupt latency can improve system performance, it is something that RTOS vendors brag about.

A context switch is the process of storing the state of a process, so that it be restored and resume execution at a later point, and allowing another process that is ready to run to execute. The time that it takes for this to occur (save the state of the currently executing process, restoring or initializing the state of the next ready process, and transferring control to the new process) is the **context switch time**. Minimizing the time for context switching reduces the overhead on a system running multiple processes, i.e. more time is spent doing useful work. Thus, reducing the context switch time can help improve the response time of a system and ensure critical deadlines are met. For a real time system, a development team that is using an OS with slow context switching can be missing real-time deadlines, but when they switch to an RTOS with a low context switch time, they could start meeting those deadlines. RTOSes are designed for a low context switch time, since real-time systems demand very fast responses to events (that could trigger context switches). For that reason, RTOS vendors brag about the low context switch time of their RTOS.

Sorabh Gandhi
Roberto Baquerizo
ECEN 5623
January 27, 2020

Timers are used in embedded systems (real-time or not) in order to keep track of when things should be happening, i.e. toggle an LED at 1Hz or check if processes are alive at a regular interval (watchdog timer). For real-time systems, timers must provide an accurate measurement of time in order to meet critical deadlines. The timers must have low jitter and drift, which are detrimental to the deterministic goals of a real-time system and can prevent the system from meeting its real-time deadlines. The timer services provided by an RTOS have much higher resolution than those provided by Embedded Linux (single-digit nanosecond resolution vs microsecond or microsecond resolution). Because an RTOS can provide a stable timer service and this is crucial for the performance of a real-time system, RTOS vendors brag about their timer services having low jitter and drift.

The figure below shows the output when the RT-Clock code is executed a couple of times. The program reports that the clock resolution is 1 nanosecond, and that the time measured by the timer is close to 3 seconds, although not exactly 3 seconds due to the error in the delta-t calculation. The program reported, on the first execution, that the error was 132030 nanoseconds. The second time it was 130780 nanoseconds, and the third time, 129634 nanoseconds. This error in the timer for 3 seconds is likely unacceptable for a real-time system, especially so because the error is not consistent or deterministic. The RT-Clock is not very accurate.

```
./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 = 1580440978, nanoseconds = 208422458
RT clock stop seconds = 1580440981, nanoseconds = 208554488
RT clock DT seconds = 3, nanoseconds = 132030
Requested sleep seconds = 3, nanoseconds = 0

Sleep loop count = 1
RT clock delay error = 0, nanoseconds = 132030

[Thu Jan 30 10:23:01]
[baquerrj@jetsonnano] [exercise-1]
>> ~/boulder/ECEN5623/exercise-1/exercise1_3
./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 = 1580444808, nanoseconds = 571655320
RT clock stop seconds = 1580444811, nanoseconds = 571786100
RT clock DT seconds = 3, nanoseconds = 130780
Requested sleep seconds = 3, nanoseconds = 0

Sleep loop count = 1
RT clock delay error = 0, nanoseconds = 130780

[Thu Jan 30 11:26:51]
[baquerrj@jetsonnano] [exercise-1]
>> ~/boulder/ECEN5623/exercise-1/exercise1_3
./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 = 1580444815, nanoseconds = 806182348
RT clock stop seconds = 1580444818, nanoseconds = 806311982
RT clock DT seconds = 3, nanoseconds = 129634
Requested sleep seconds = 3, nanoseconds = 0

Sleep loop count = 1
RT clock delay error = 0, nanoseconds = 129634
```

Sorabh Gandhi
Roberto Baquerizo
ECEN 5623
January 27, 2020

Problem 4

Simplethread Program

Build and Run Steps:

1.)Download the project folder from canvas - (https://canvas.colorado.edu/courses/58104/files/12570965?module_item_id=1730299) and extract the files.
2.) Build the simplethread project using “make” command
3.) Execute the program using ./pthread command

The observed output is shown below,

```
Thread idx=0, sum[0...0]=0
Thread idx=1, sum[0...1]=1
Thread idx=2, sum[0...2]=3
Thread idx=3, sum[0...3]=6
Thread idx=4, sum[0...4]=10
Thread idx=5, sum[0...5]=15
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
```

Program Explanation

1.)In this program ‘n’ number of threads are created which is specified using the macro - “NUM_THREADS”
2.)The thread handler function - “counterthread” calculates the sum from 1 to the threadIdx which is passed as an argument to the thread
3.)The main thread creates all the thread and waits for all the thread to run to completion and then it exits.

RT_Simplethread Program

Build and Run Steps:

1.)Download the project folder from canvas - (https://canvas.colorado.edu/courses/58104/files/12570965?module_item_id=1730299) and extract the files.
2.)Build the rt_simplethread project using “make” command
3.)Execute the program using sudo ./pthread command

The observed output is shown below,

```
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, 5 msec (5941 microsec)

TEST COMPLETE
```

Program Explanation

1.) This program sets the SCHED_FIFO scheduling policy for the main thread and assigns it a highest priority.
2.) Further the main thread creates a 'n' number of threads and assigns everyone the SCHED_FIFO scheduling policy. It then assigns priority in descending order based on the thread created.
3.) The thread handler function - "counterThread" executes a dummy - fibtest function (for synthetic load) and computes the execution time
4.) Also the main thread binds all the threads in the program to a single core using the function - pthread_attr_setaffinity_np
5.) The main thread waits for all the running thread to run to completion and then exits

RT_thread_improved Program

Build and Run Steps:

1.)Download the project folder from canvas - (https://canvas.colorado.edu/courses/58104/files/12570965?module_item_id=1730299) and extract the files.
2.)Build the rt_thread_improved project using "make" command
3.)Execute the program using sudo ./pthread command

The observed output is shown below,

```
This system has 8 processors configured and 8 processors available.
number of CPU cores=8
Using sysconf number of CPUS=8, count in set=8
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=3, affinity contained: CPU-0  CPU-1  CPU-2  CPU-3  CPU-4  CPU-5  CPU-6  CPU-7

Thread idx=3 ran 0 sec, 196 msec (196858 microsec)

Thread idx=0, sum[0...100]=4950
Thread idx=0 ran on core=2, affinity contained: CPU-0  CPU-1  CPU-2  CPU-3  CPU-4  CPU-5  CPU-6  CPU-7

Thread idx=0 ran 0 sec, 197 msec (197061 microsec)

Thread idx=2, sum[0...300]=44850
Thread idx=2 ran on core=1, affinity contained: CPU-0  CPU-1  CPU-2  CPU-3  CPU-4  CPU-5  CPU-6  CPU-7

Thread idx=2 ran 0 sec, 197 msec (197010 microsec)

Thread idx=1, sum[0...200]=19900
Thread idx=1 ran on core=0, affinity contained: CPU-0  CPU-1  CPU-2  CPU-3  CPU-4  CPU-5  CPU-6  CPU-7

Thread idx=1 ran 0 sec, 197 msec (197088 microsec)

TEST COMPLETE
```

Program Explanation

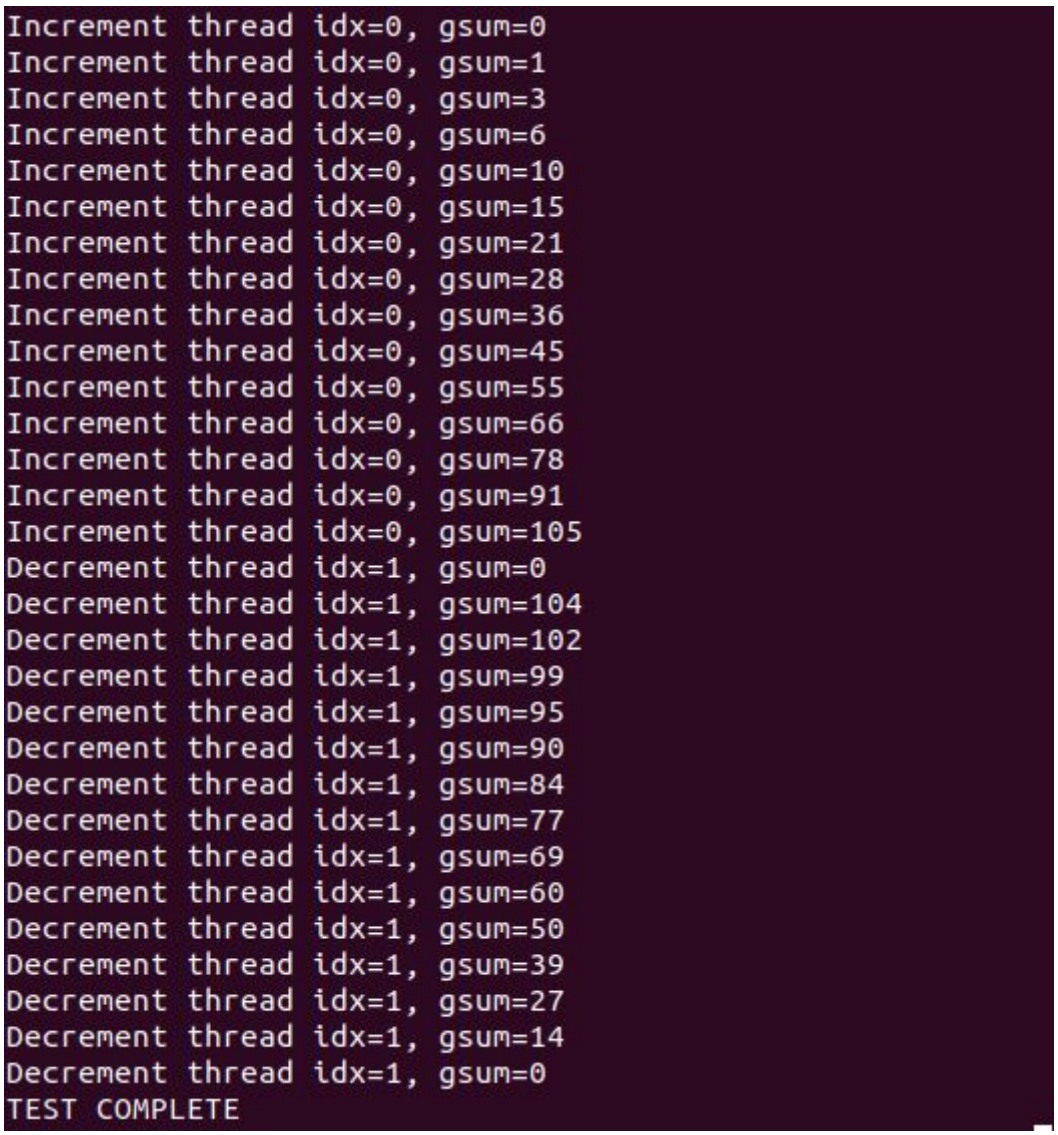
1.)This program is to ensure that all CPU cores are in a normal state.
2.)The thread handler runs the assigned task on each and every CPU core and verifies the CPU state
3.)The main program creates n threads based on the number of threads specified in the macro “NUM_THREADS”. It then waits for all the threads to complete its operation and joins it to the main thread.

Incdecthread Program

Build and Run Steps:

1.)Download the project folder from canvas - (https://canvas.colorado.edu/courses/58104/files/12570965?module_item_id=1730299) and extract the files.
2.)Build the incdecthread project using “make” command
3.)Execute the program using sudo ./pthread command

Sorabh Gandhi
Roberto Baquerizo
ECEN 5623
January 27, 2020
The observed output is shown below,



Program Explanation

1.)The main function creates two threads - incThread and decThread. Both of them updates the same variable ‘gsum’
2.)The incthread increments the value of gsum variable by the value of ith iteration and similarly the decthread decrements the value of gsum variable by the value of ith iteration

Sorabh Gandhi
Roberto Baquerizo
ECEN 5623
January 27, 2020

VxWorks Code Explanation

This Vxworks codepiece implements a two-task scheduler to replicate the LCM invariant schedule. The main function (start()) spawns a sequencer task which is responsible for replicating the timing diagram. The sequencer task spawns two task - fib10 and fib 20. After creating both the task, the sequencer schedules the task according to the timing diagram, making the design precise and predictable. Binary semaphores are used to emulate the running sequence of each task. Using the semaphore API, the sequencer posts to f1b 10 and fib 20 task in the sequence as shown below,

```
/* Sequencing loop for LCM phasing of S1, S2
 */
while(!abortTest)
{

        /* Basic sequence of releases after CI */
    taskDelay(20); semGive(semF10);
    taskDelay(20); semGive(semF10);
    taskDelay(10); semGive(semF20);
    taskDelay(10); semGive(semF10);
    taskDelay(20); semGive(semF10);
    taskDelay(20);
```

The thread handler waits until it receives the appropriate semaphore and then executes the FIB_TEST function with a set number of iterations based on the assumed completion time C1 and C2.

When both the task get semaphore initially, fib10 runs first as it is of higher priority. In the meantime, sequencer task is asleep for 20ms. First the fib 10 executes for 10ms and then fib 20 executes for 10ms as its next in the queue. After that when the sequencer wakes up, it then repeats the same cycle again. Similarly, the entire sequence of releases during the runtime of fib 10 and fib 20 is achieved.

Linux Code implementation

Build and Run Steps:

1.) Download the submitted exercise_1 folder from canvas and extract the files.
2.) Build the problem4 project using “make” command
3.) Execute the program using sudo ./pthread command

The observed output is shown below,

```
Pthread Policy is SCHED_FIFO
max prio = 99
PTHREAD SCOPE SYSTEM
Starting Sequencer
Fib10 priority = 98 and time stamp = 1.975060 msec
Fib20 priority = 97 and time stamp = 5.809069 msec
Fib10 priority = 98 and time stamp = 24.585009 msec
Fib10 priority = 98 and time stamp = 45.141935 msec
Fib10 priority = 98 and time stamp = 65.499067 msec
Fib20 priority = 97 and time stamp = 66.205978 msec
Fib10 priority = 98 and time stamp = 87.471962 msec
Test Conducted over 100.569010 msec
test complete
```

Linux POSIX threads:

In Linux implementation, I have used POSIX API to create threads and schedule using semaphores. This is similar to the taskSpawn API used in VxWorks implementation. Also in Linux, the scheduling policy and priority of the task is configured using pthread attributes and is passed as an argument while creating the thread.

The snapshot of code configuration in linux version is shown below,

```
pthread_attr_init(&fib10_sched_attr);
pthread_attr_init(&fib20_sched_attr);
pthread_attr_init(&main_sched_attr);

pthread_attr_setinheritsched(&fib10_sched_attr, PTHREAD_EXPLICIT_SCHED);
pthread_attr_setschedpolicy(&fib10_sched_attr, SCHED_FIFO);

pthread_attr_setinheritsched(&fib20_sched_attr, PTHREAD_EXPLICIT_SCHED);
pthread_attr_setschedpolicy(&fib20_sched_attr, SCHED_FIFO);

pthread_attr_setinheritsched(&main_sched_attr, PTHREAD_EXPLICIT_SCHED);
pthread_attr_setschedpolicy(&main_sched_attr, SCHED_FIFO);

max_priority = sched_get_priority_max(SCHED_FIFO);
rc=sched_getparam(getpid(), &nrt_param);
main_param.sched_priority = max_priority;
fib10_param.sched_priority = (max_priority - 1);
fib20_param.sched_priority = (max_priority - 2);

rc = sched_setscheduler(getpid(), SCHED_FIFO, &main_param);
if (rc) {
    printf("ERROR; sched_setscheduler rc is %d\n", rc); perror(NULL);
    exit(-1);
}


pthread_attr_setschedparam(&fib10_sched_attr, &fib10_param);
pthread_attr_setschedparam(&fib20_sched_attr, &fib20_param);
pthread_attr_setschedparam(&main_sched_attr, &main_param);

start = get_curr_time();
rc = pthread_create(&fib10_thread_id, &fib10_sched_attr, fib10, NULL);
if (rc != 0) {
    handle_error_en(s, "Error Creating fib10 thread\n");
}

if (pthread_create(&fib20_thread_id, &fib20_sched_attr, fib20, NULL) != 0) {
    handle_error_en(s, "Error Creating fib20 thread\n");
}

if (pthread_create(&main_thread_id, &main_sched_attr, sequencer, NULL) != 0) {
    handle_error_en(s, "Error Creating Sequencer thread\n");
}
```

Sorabh Gandhi
Roberto Baquerizo
ECEN 5623
January 27, 2020
Program Explanation:

The implementation of the Linux version is pretty similar to VxWorks codepeice. In this codepiece, three threads are created - fib10, fib 20 and sequencer. In all the three threads scheduling policy is set as SCHED_FIFO and the highest priority is assigned to sequencer, then followed by fib10 and fib20. Even in this implementation, sequencer is responsible for scheduling fib10 and fib20 task based on the timing diagram to achieve a reliable and predictable response.

Finally after the entire execution cycle, all the threads are joined and the semaphores and attributes are destroyed.

Synthetic Workload generation:

The FIB_TEST macro calculates the fibonacci series based on the sequence count and the iteration count specified by the thread calling this function. This macro was used in VxWorks to generate the load on the CPU and we have also used the same macro in the linux implementation. The number of CPU cycles for fibonacci series has to be calculated based on the measured time. Using this approach we achieve a load generation of 10ms for fib 10 and 20ms for fib 20.

The obtained result using linux version is similar to the given timing diagram that was replicated in VxWorks codepiece authored by Dr.Sam Siewert.

Reference:

1.) Independent study paper on “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” by Nisheeth Bhat.
2.)http://mercury.pr.erau.edu/~siewerts/cec450/code/VxWorks-sequencers/lab1.c
3.)https://www.electronicsweekly.com/market-sectors/embedded-systems/analysis-linux-versus-rtos-2008-08/