

EXERCISE 1 REPORT  
ECEN 5623  
SUMMER 2021  
UNIVERSITY OF COLORADO BOULDER

WRITTEN BY: SALONI SHAH

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).

Rate Monotonic (RM) Policy is a static priority scheduling algorithm. In this algorithm, highest priority is assigned to the service with highest frequency or shortest release periods. Moreover, rate monotonic policy is preemptive in nature i.e., while executing a task, another task with shorter period can overrun the running task. RM policy is used in aircraft monitoring, satellite systems and other hard real time systems where missing a deadline can cause significant damage of life and assets. In such systems, where meeting deadlines is critical, RM policy is an obvious choice. Also, with rate monotonic policy, we can predict exactly which tasks are likely to miss deadlines.

- a. 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 here – note that we have not yet covered dynamic priorities, just RM fixed policy described here, so ignore EDF and LLF for now].

In the given three services examples,

For service S1:  $T_1=3$ ,  $C_1=1$ ;

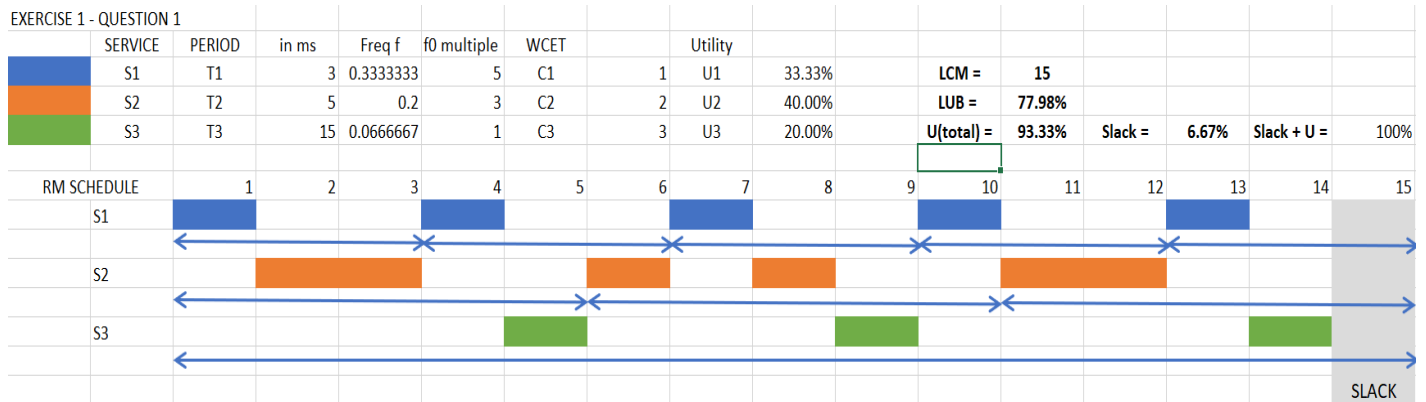
For service S2:  $T_2=5$ ,  $C_2=2$ ;

For service S3:  $T_3=15$ ,  $C_3=3$ .

Here  $T_1$ ,  $T_2$  and  $T_3$  are periods of the services and  $C_1$ ,  $C_2$  and  $C_3$  is the run-time of the services.

Here, according to Rate Monotonic Policy, the priorities of these services can be set as  $S_1 > S_2 > S_3$  as  $S_1$  has highest frequency and  $S_3$  has lowest frequency.

The timing diagram for the three services is drawn in the excel file [exercise1\\_1\\_timing\\_diagram](#) [here](#).



The above diagram shows the timing diagram for given services. The diagram shows calculated values for frequency, utility factor, LCM (Least Common Multiple), RM LUB (Least Upper Bound) and total CPU utilization by the services.

- Frequency  $f$  of the services:

For S1:  $f = 1/T1$   
 $= 1/3$   
 $= 0.3333333$

For S2:  $f = 1/T2$   
 $= 1/5$   
 $= 0.2$

For S3:  $f = 1/T3$   
 $= 1/15$   
 $= 0.0666667$

- Utility  $U$  of the services:

For S1:  $U1 = C1/T1$   
 $= 1/3$   
 $= 0.3333333 \approx 33.33\%$

For S2:  $U2 = C2/T2$   
 $= 2/5$   
 $= 0.4 = 40\%$

$$\begin{aligned}\text{For S3: } U_3 &= C_3/T_3 \\ &= 3/15 \\ &= 0.2 = 20\%\end{aligned}$$

- LCM of the services:

$$\begin{aligned}\text{LCM} &= \text{lcm}(3, 5, 15) \\ &= 15\end{aligned}$$

- LUB of the Services:

$$\begin{aligned}\text{LUB} &= m \left( 2^{\left(\frac{1}{m}\right)} - 1 \right) \\ &= 3 \left( 2^{\left(\frac{1}{3}\right)} - 1 \right) \\ &= 0.7797631 \approx 77.98\%\end{aligned}$$

Where  $m$  = number of services

- Total utility of the services:

$$\begin{aligned}U &= U_1 + U_2 + U_3 \\ &= 33.33\% + 40\% + 20\% = 93.33\%\end{aligned}$$

- Processor idle time for the services:

$$\text{Slack} = 1/15 = 0.0666667 \approx 6.67\%$$

- 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).

As mentioned by Liu and Layland in their paper 'Scheduling Algorithms for Multiprogramming', a scheduling algorithm is feasible if an overflow never occurs by the tasks scheduled. A simple calculation can determine whether the algorithm is feasible. Primarily according to Liu and Layland, if all requests are fulfilled for the tasks before their deadline, then the algorithm is feasible scheduling algorithm. Based on that statement, the RM algorithm for given three services is feasible.

Along with meeting deadlines, reliability of the system is also vital. The service execution must be repeatable, for the system to execute services and meet deadlines. If meeting these deadlines is guaranteed, then the system is

considered safe. For the system to be safe it must have deterministic timings. Whether a schedule is safe or not cannot be determined based on any calculations but is more of an arbitrary topic. For systems to get executed safely, there should be some margin in CPU utilization. This schedule utilizes 93% of processor to meet deadlines of all services. It is highly likely that the scheduling policy will miss some deadline if any of the services is interrupted.

As a result, the rate monotonic policy for given services is feasible but probably unsafe.

- c. What is the total CPU utilization by the three services?

The total CPU utilization by the three services can be calculated as follows.

$$\begin{aligned}
 U &= \frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{C_3}{T_3} \\
 &= 1/3 + 2/5 + 3/15 \\
 &= 0.933333
 \end{aligned}$$

$$= 93.33\%$$

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.

Apollo 11 was the first mission to make a crewed lunar landing and return to earth, undertaken by NASA in 1969. Here, we are discussing about the resource overload error that occurred 3 minutes before lunar landing on Apollo 11. This program alarm could have caused system failure and thus aborting the first ever lunar descent mission. However, because of the robust programming of the lunar module done by Margaret Hamilton, this crisis was averted. Below I have discussed the overload story and how it was solved in detail.

While entering the final stage of lunar descent, the Apollo 11 guidance computer generated alarm 1202 indicating missed deadlines. This alarm was noticed by Buzz Aldrin, Lunar Module Pilot for the mission. Eugene Kranz further explains the catastrophic outcome of the alarm. If the alarm continued, it would make the guidance, navigation, and crew display updates unreliable. Further, if the alarm were sustained, it would cause the computer to come to a halt and result in mission abortion. It was discovered that the continuous generation of tasks by rendezvous radar data caused the CPU resources to overload and the system started missing deadlines for lunar descent program. Peter Adler in his account, Apollo 11 Program Alarms explains in detail the root cause of these resource overflows in detail.

Peter Adler was a young expert at MIT Instrumentation Lab who worked on LM flight routines. In his account, Peter discusses about the memory constraints of the system and memory allocation to various tasks. In Apollo 11 program, one memory location was shared by various jobs. Extensive testing was undertaken to ensure that the same memory location was not shared by programs running at the same time. Fundamentally, each job was allocated a core set of 12 erasable locations. Further, if any job required extra temporary storage, VAC (Vector Accumulator) was assigned with 44 erasable words. Apollo 11 had 7 such core sets and 5 VACs. While executing a job, the system would first scan for any available VACs and then allocate core sets to the job. Alarm 1201 would be generated if no VACs are available and alarm 1202 is generated if no core set is available. As a result of insignificant radar data occupying the core sets and VACs the Apollo 11 system resources were quickly exhausted and the guidance computer generated alarm 1202 and later alarm 1201 while landing. This overload situation was anticipated while simulation and the program was made robust to overcome the system failure. One key feature of Apollo 11 guidance computer was that the system would reboot instantaneously upon generation of any such alarms. On system restart, the computer would automatically return to executing higher priority tasks like lunar descent program and abort any other low priority tasks which would be running and causing error like analysis of rendezvous radar data. This robust programming of the module made mankind's first lunar landing mission a success. The key person behind this

priority display innovation in  
Apollo 11 is Margaret Hamilton, head of Apollo 11 software design team.

Margaret Hamilton is an American software and systems design engineer. She is the brains behind the design of on-board flight software for Apollo 11. She also designed the “priority display” for Apollo 11 guidance computer. Regarding the Apollo 11 incident, Hamilton reported that the Apollo 11 software recognized that it had to perform more tasks in real-time than it could. Fortunately, the software was programmed to tackle any such situations and it was smart enough to recover from the alarms, reboot the system and run only higher priority tasks. Hamilton further writes that if the computer had not recognized this error, Apollo 11 would not have been able to perform lunar landing.

To summarize, Apollo 11 guidance computer suffered resource overload due to rendezvous radar data. This data completely occupied the memory core sets as well as vector accumulators (VAC sets). It was believed that the low priority radar data analysis was executed due to a radar switch which was kept on by mistake. However, the system recovered from the overload alarms due to robust programming of the computer by Margaret Hamilton and her team. The program immediately caught the error and aborted all low priority tasks. This way the first crewed lunar landing mission Apollo 11 undertaken by NASA was successful.

- a. What was the root cause of the overload and why did it violate Rate Monotonic policy?

The root cause of the 1201/1202 alarms was system overload. Because a wrong radar switch was left on (according to some reports) and the radar system which was to be used while leaving the moon and the computer guidance program, used incompatible power supplies. This caused the radar to send data continuously to the computer which had nothing to do with lunar landing. Due to this, the system started missing deadlines for the descent program and caused resource overload. According to the system program, the rendezvous radar data was supposed to be a low priority task and the lunar landing program

a higher priority task. Although, the system executed low priority tasks that resulted in missed deadlines for higher priority task. Rate Monotonic Policy states that the system will always execute higher priority task without missing any deadlines, which was not the case here. Hence, it can be considered as the violation of Rate Monotonic Policy.

- b. 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 increase.
- c. Plot this Least Upper bound as a function of number of services.

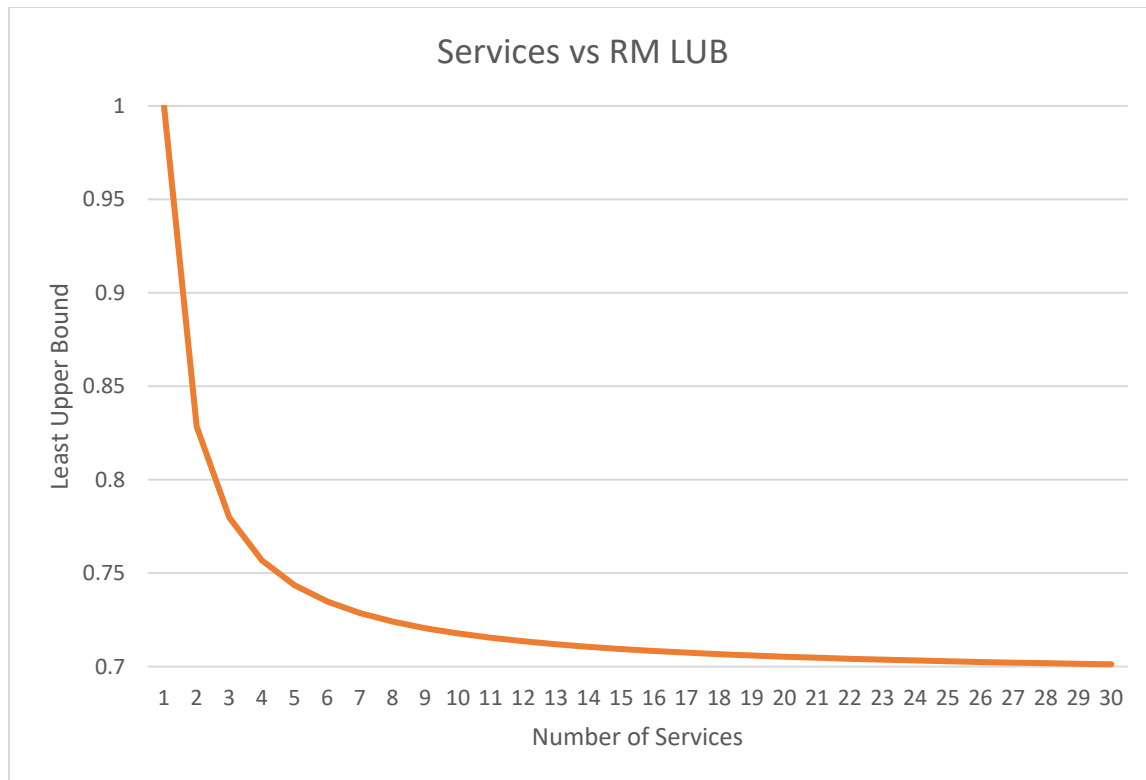
According to Liu and Layland's paper, RM Least Upper Bound can be calculated using the formula.

$$\text{LUB} = m \left( 2^{\left(\frac{1}{m}\right)} - 1 \right)$$

Here m = Number of services

Based on this formula, we can calculate RM LUB for any number of services. For 1 service, LUB is 1 i.e., 100%. With increase in number of services, RM LUB decreases. After about 10 services, value of LUB remains relatively constant. For 100 services, the value of LUB is 69.555506%. Below graph plot the value of RM LUB with respect to number of services.





- d. Describe 3 key assumptions they make and document 3 or more aspects of their fixed priority LUB derivation that you do not understand.

The most important assumptions made by Liu and Layland in their paper is assumption A1 and A4. They described these two assumptions as most important and least defensible and that these assumptions should be made design goals for real-time systems to meet hard deadlines.

- Assumption (A1) - The requests for all tasks for which hard deadlines exist are periodic, with constant interval between requests.
- Assumption (A4) - 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.

According to Liu and Layland, if these two assumptions do not hold true, we would need detailed knowledge of run-time periods and request periods of the

services. Unless these values are known, other analysis would also not work for scheduling.

- Assumption (A2) - Deadlines consist of run-ability constraints only--i.e., each task must be completed before the next request for it occurs.

This assumption would eliminate all queuing problems but for it to work, we would need buffering hardware for each function.

**Aspects I did not understand about Liu and Layland's fixed priority LUB derivation:**

- In theorem 5, the use of polynomial equation and variables  $q$  and  $r$  was unclear to me. I could not relate the use of this change with equations with how we can derive LUB from there.
- Similarly, for theorem 4 I was not able to understand the significance of  $C'$  and  $C''$  in deriving LUB. Consequently, the derivation of equation for  $C_m$  from  $C'$ ,  $C''$ ,  $T$  and  $U$  was very unclear to me. A proper explanation and detailed derivation with steps for these equations would have made the understanding procedure easier.
- Moreover, in theorem 3, I was able to understand the procedure and explanation for two tasks. I also understood the requirement of analysis of two different cases. But the subsequent derivation of utilization factor in both cases was very vague for me. I had no idea where the terms were calculated from.
- Overall, theoretically the paper was quite informative and gave good insight on designing real-time systems and scheduling algorithms. But the mathematical derivations seemed random and obscure. A proper reference for these mathematical calculations or detailed explanation of the steps would have made the paper more understandable to even a reader having very basic knowledge in the field.

- e. Would RM analysis have prevented the Apollo 11 1201/1202 errors and potential mission abort? Why or why not?

Alarm 1201 and 1202 in the Apollo 11 guidance computer indicated no available core sets or VAC sets for memory storage. These sets were completely utilized by data coming from the radar system which was a low priority task and hence the system could not meet deadline for higher priority task of lunar descent. Due to less knowledge of overall working of Apollo 11 and regarding the system failure, we need to make some assumptions to determine whether RM policy would have avoided the issue. I have explained my assumptions and conclusions below:

- If we assume that the rendezvous radar data task would be running faster than lunar landing data calculation, then RM policy would have given radar data service higher priority, overflowing available system resources and missing deadlines for lunar landing. In this case, if RM policy were applied in Apollo 11 it would have been a failed mission.
- An important point to note here is that Apollo 11 mission was launched in 1969 while the Rate Monotonic Policy was formalized in 1972. Even if the scientists working on Apollo 11 had the knowledge of this policy, there would be a major risk factor involved with utilizing this policy. In absence of any formal research on the scheduling policy, it is challenging to implement it successfully on such an important mission.
- Moreover, according to Liu and Layland's paper, a feasible and safe RM policy can be executed only when there is a margin between available processor and utilized CPU processing. In other words, CPU utilization should be less than RM LUB. In Apollo11 mission, where already we are short of processor power, leaving so much margin would require us to use another processor and hence increase the bulkiness of the shuttle. Also, on a heavy-duty space shuttle like this, with a plethora of services running it is almost impossible to optimize processor utilization.
- Hence in my opinion, it would not have been practical or safe to implement RM policy. There could have been some way to use the policy, but to determine that we would need more intel on Apollo 11 and the system.

3. Download RT-Clock and build it on an R-Pi3b+ or newer and execute the code.

- a. Describe what the code is 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).

**Code Gist:** This code demonstrates the working and accuracy of a real-time clock running as POSIX-thread or pthread. The real-time clock calculates the time spent during system sleep time which is pre-defined. A few numbers of test iterations are run to acquire samples of real-time clock and test its precision. This service is executed by the system as a thread.

#### Main functions for RT Clock:

- **`void print_scheduler(void):`**

This function checks the scheduling policy of the thread and prints the currently used scheduling policy. To get the scheduling policy it uses inbuilt function `sched_getscheduler` which returns the type of scheduling policy implemented. This function takes the thread identity as a parameter to check policy. `getpid()` function returns the process ID of the process.

Linux supports certain non-real-time and real-time policies which are as follows:

Non-Real-time Policies	Real-time Policies
<code>SCHED_OTHER</code> – standard round-robin time-sharing policy	<code>SCHED_FIFO</code> – first in, first out policy
<code>SCHED_BATCH</code> – batch style execution of policies	<code>SCHED_RR</code> – round-robin policy
<code>SCHED_IDLE</code> – running very low priority background jobs	

```

/*****
/* Function to print the current scheduling policy */
/* Parameters: none */
/* Return type: void */
*****/
void print_scheduler(void)
{
    int schedType;

    schedType = sched_getscheduler(getpid()); //returns the current scheduling policy of the thread i
identified by getpid(getpid() returns the process ID (
PID) of the calling process)

    switch(schedType) //schedtype is an integer whose value we get from scheduler info
    {
        case SCHED_FIFO: //first in first out policy
            printf("Pthread Policy is SCHED_FIFO\n");
            break;
        case SCHED_OTHER: //standard round-robin time-sharing policy
            printf("Pthread Policy is SCHED_OTHER\n");
            break;
        case SCHED_RR: //round-robin policy
            printf("Pthread Policy is SCHED_RR\n");
            break;
        default:
            printf("Pthread Policy is UNKNOWN\n");
    }
}

```

- **int delta\_t(timespec \*stop, timespec \*start, timespec \*delta\_t):**

This function calculates the time difference between the start time and stop time based on value of parameters passed. These parameters are pointers to structures which are in the form of time values in seconds and nanoseconds.

```

struct timespec {
    time_t tv_sec;
    long tv_nsec;
};

```

The time difference calculated is stored in delta\_t. The following code snippets show how the code handles various cases for the time difference in seconds and handles roll-over and overflow situation for time difference in nanoseconds.

```

int delta_t(struct timespec *stop, struct timespec *start, struct timespec *delta_t)
{
    //calculate difference of time in sec and nanosec
    int dt_sec=stop->tv_sec - start->tv_sec;
    int dt_nsec=stop->tv_nsec - start->tv_nsec;

    //printf("\ndt calculation\n");

    // case 1 - less than a second of change
    if(dt_sec == 0) //if the time difference in seconds is 0
    {
        //printf("dt less than 1 second\n");

        if(dt_nsec >= 0 && dt_nsec < NSEC_PER_SEC) //if dt_nsec is greater than or equal to 0 AND less
than 1e9
        {
            //printf("nanosec greater at stop than start\n");
            delta_t->tv_sec = 0;
            delta_t->tv_nsec = dt_nsec; //time difference in nanosec is same
        }

        else if(dt_nsec > NSEC_PER_SEC) //if dt_nsec is greater than 1e9 then increment sec and update
time diff in nsec to handle rollover
        {
            //printf("nanosec overflow\n");
            delta_t->tv_sec = 1;
            delta_t->tv_nsec = dt_nsec-NSEC_PER_SEC;
        }

        else // dt_nsec < 0 means stop is earlier than start
        {
            printf("stop is earlier than start\n"); //show error message for invalid time diffe
rence
            return(ERROR);
        }
    }

    // case 2 - more than a second of change, check for roll-over
    else if(dt_sec > 0) //now check if dt_sec is greater than 0
    {
        //printf("dt more than 1 second\n");

        if(dt_nsec >= 0 && dt_nsec < NSEC_PER_SEC) //if dt_nsec is greater than or equal to 0 AND less
than 1e9
        {
            //printf("nanosec greater at stop than start\n");
            delta_t->tv_sec = dt_sec; //time diff in sec is unchanged
            delta_t->tv_nsec = dt_nsec; //time diff in nanosec is unchanged
        }

        else if(dt_nsec > NSEC_PER_SEC) //if dt_nsec is greater than 1e9, increment time diff in sec
and update time diff in nanosec
        {
            //printf("nanosec overflow\n");
            delta_t->tv_sec = delta_t->tv_sec + 1;
            delta_t->tv_nsec = dt_nsec-NSEC_PER_SEC;
        }

        else // dt_nsec < 0 means roll over
        {
            //printf("nanosec roll over\n");
            delta_t->tv_sec = dt_sec-1; //decrement time diff in nanosec
            delta_t->tv_nsec = NSEC_PER_SEC + dt_nsec; //handle nanosecond rollover
        }
    }

    return(OK);
}

```

- **structures and #define declarations:**

The following code snippet shows the declaration of start time and stop time for the real time clock. `rtclk_start_time` and `rtclk_stop_time` is used to calculate the time difference. The structures `my_rtclk_start_time` and `my_rtclk_stop_time` are implemented by me to demonstrate my understanding of `clock_gettime` function and to calculate time taken by the code to run all test iterations.

The next part of the code defines clock types supported in Linux. These clock types are explained in detail in next part of the answer.

```
//define real time start time, stop time, time difference and timing error for realtime clock
static struct timespec rtclk_dt = {0, 0};
static struct timespec rtclk_start_time = {0, 0};
static struct timespec rtclk_stop_time = {0, 0};
static struct timespec delay_error = {0,0};
static struct timespec my_rtclk_diff = {0, 0};
static struct timespec my_rtclk_start_time = {0, 0};
static struct timespec my_rtclk_stop_time = {0, 0};

//define clock type
//#define MY_CLOCK CLOCK_REALTIME
//#define MY_CLOCK CLOCK_MONOTONIC
#define MY_CLOCK CLOCK_MONOTONIC_RAW
//#define MY_CLOCK CLOCK_REALTIME_COARSE
//#define MY_CLOCK CLOCK_MONOTONIC_COARSE

//run the realtime clock for 100 tests
#define TEST_ITERATIONS (100)
```

- **void \*delay\_test(void \*threadID):**

This function is used to calculate the time difference based on the clock start time and stop time. For this delay calculation, the function first sets up clock resolution. Linux supports inbuilt function `clock_getres` which calculates the resolution of selected clock type and stores it in `rtclk_resolution` which is user defined. Clock resolution is calculated in seconds and nanoseconds.

```
//finds resolution of MY_CLOCK and store it in struct timespec rtclk_resolution and checks for an error
if(clock_getres(MY_CLOCK, &rtclk_resolution) == ERROR)
{
    perror("clock_getres");
    exit(-1);
}
else
{
    //prints the resolution of MY_CLOCK
    printf("\n\nPOSIX Clock demo using system RT clock with resolution:\n\t%ld secs, %ld microsecs, %ld nanosecs\n", rtclk_resolution.tv_sec, (rtclk_resolution.tv_nsec/1000), rtclk_resolution.tv_nsec);
}
```

Then the code initializes variables to specify required system sleep time. Before running the time test, code takes the iteration start time using `clock_gettime`. `clock_gettime` function retrieves timestamp of specified clock

type (here MY\_CLOCK) at that instant and stores the time in `rtclk_start_time`. After storing start time, the system stops running the thread. `nanosleep(&sleep_time, &remaining_time)` function suspends execution of the task or thread until `sleep_time` elapses. If `nanosleep` is interrupted by other task or service, then it stores the `remaining_time`. According to the below code snippet, if `nanosleep` is interrupted, then the `sleep_time` is updated and `nanosleep` will execute for remaining time period.

```

/* run test for defined seconds */
sleep_time.tv_sec=TEST_SECONDS;           //TEST_SECONDS=0
sleep_time.tv_nsec=TEST_NANOSECONDS;      //TEST_NANOSECONDS=10ms
sleep_requested.tv_sec=sleep_time.tv_sec;  //0
sleep_requested.tv_nsec=sleep_time.tv_nsec; //10ms

/* start time stamp */
clock_gettime(MY_CLOCK, &rtclk_start_time); //clock_gettime retrieves time of the specified clock

/* request sleep time and repeat if time remains */
do
{
    if(rc=nanosleep(&sleep_time, &remaining_time) == 0) break; //suspends execution of thread until
    sleep_time has elapsed //here sleep_time = 10ms. if nanosleep is interrupted before sleep time passes, the remaining time will be updated
    sleep_time.tv_sec = remaining_time.tv_sec;
    sleep_time.tv_nsec = remaining_time.tv_nsec;
    sleep_count++; //initially sleep count is 0
    printf("nanosleep is getting interrupted, sleep count is incremented\n");
}
while (((remaining_time.tv_sec > 0) || (remaining_time.tv_nsec > 0)) //executes until remaining time is greater than 0 or
    && (sleep_count < max_sleep_calls)); //sleep_count<3
//printf("Sleep count=%d\n",sleep_count);
//printf("remaining time=%ld\n",remaining_time.tv_nsec);

```

At the end of every sleep cycle, `clock_gettime` stores the stop time. `Delta_t` function then calculates the time difference. This time difference is then compared with the requested sleep time to calculate real time clock error.

```

//gets current stop time after completing each sleep time
clock_gettime(MY_CLOCK, &rtclk_stop_time);

//calculates time diff between start and stop time and stores it in rtclk_dt
delta_t(&rtclk_stop_time, &rtclk_start_time, &rtclk_dt);

//calculates diff between requested sleep time(10ms) and the real time diff and considers it as delay
delta_t(&rtclk_dt, &sleep_requested, &delay_error);

```



- **void main(void):**

The main function creates the task thread and assigns scheduler attributes and parameters.

`pthread_attr_init` initializes thread attributes.

`pthread_attr_setinheritsched` specifies the source of attributes to be inherited by the scheduler.

`pthread_attr_setschedpolicy` sets the scheduling policy of the thread.

`sched_get_priority_max` and `sched_get_priority_min` returns the maximum and minimum priority that can be set for the scheduling policy.

`sched_setscheduler` sets the scheduling policy and its parameters for specified process or thread.

`pthread_attr_setschedparam` sets the attributes of the thread to the parameters specified.

`pthread_create` creates a thread with specified name and parameters and invokes the function passed in the parameters after the thread is created.

`pthread_join` executes the thread until it is terminated or pre-empted.

`pthread_attr_destroy` destroys the parameters set for the thread in order to facilitate its reinitialization.

Aside from the specified functionality, I have updated the code to note the start time and stop time of the code and calculated the time taken to execute the complete code. Start and stop time stamps are stored using `clock_gettime` function.

```

#ifdef RUN_RT_THREAD
    pthread_attr_init(&main_sched_attr);           //initialize thread attributes object of main_sched_attr
    pthread_attr_setinheritsched(&main_sched_attr, PTHREAD_EXPLICIT_SCHED); //specifies from where main_
    sched_attr will inherit its attrib
        utes
    pthread_attr_setschedpolicy(&main_sched_attr, SCHED_FIFO); //set scheduling policy of main_thread_attr
    to first in first out

    rt_max_prio = sched_get_priority_max(SCHED_FIFO); //returns maximum allowable priority to FIFO
    rt_min_prio = sched_get_priority_min(SCHED_FIFO); //returns minimum allowable priority to FIFO

    main_param.sched_priority = rt_max_prio; //stores maximum allowable priority
    rc=sched_setscheduler(getpid(), SCHED_FIFO, &main_param); //sets SCHED_FIFO and main_param both for th
    read specified by getpid()

    if (rc) //if rc is nonzero show error message
    {
        printf("ERROR; sched_setscheduler rc is %d\n", rc);
        perror("sched_setscheduler"); exit(-1); //produce error message
    }

    printf("After adjustments to scheduling policy:\n");
    print_scheduler(); //print updated scheduling policy

    main_param.sched_priority = rt_max_prio;
    pthread_attr_setschedparam(&main_sched_attr, &main_param); //sets main_sched_attr attributes to paramet
    ers specified by main_param

    rc = pthread_create(&main_thread, &main_sched_attr, delay_test, (void *)0); //creates a thread main_thre
    ad with attributes main_sched_attr. In
        vokes delay_test with threadid=0

    if (rc) //shows error message if there is an error creating the thread
    {
        printf("ERROR; pthread_create() rc is %d\n", rc);
        perror("pthread_create");
        exit(-1);
    }

    pthread_join(main_thread, NULL); //executes main_thread until it is terminated

    if(pthread_attr_destroy(&main_sched_attr) != 0) //destroys main_sched_attr attributes so that it can
    be reinitialized
        perror("attr destroy");
    #else
        delay_test((void *)0);
        printf("it is in else delay_test\n");
    #endif

    printf("TEST COMPLETE\n\n");

    //get code end time
    clock_gettime(MY_CLOCK, &my_rtclock_stop_time);
    delta_t(&my_rtclock_stop_time, &my_rtclock_start_time, &my_rtclock_diff); //calculate difference between code
    start and end time
    printf("Code start time - %ld sec, %ld nsec\nCode stop time - %ld sec, %ld nsec\n",my_rtclock_start_time.tv
    _sec,my_rtclock_start_time.tv_nsec, my_rtclock_stop_time.tv_sec, my_rtclock_stop_time.tv_nsec);
    printf("Time taken to execute the code is %ld sec, %ld msec, %ld usec, %ld nsec\n",
        my_rtclock_diff.tv_sec, my_rtclock_diff.tv_nsec/1000000, my_rtclock_diff.tv_nsec/1000, my_rtclock_diff.tv_n
    sec);
}

```

- b. Which clock is best to use? `CLOCK_REALTIME`, `CLOCK_MONOTONIC` or `CLOCK_MONOTONIC_RAW`? Please choose one and update code and improve the commenting.

Linux manual pages explains the different clock types as follows:

- **`CLOCK_REALTIME`**

This is a settable system clock which measures the real-time like a watch or wall-clock. This is sort of system's best effort to sync with real time. Although, this clock is affected by changes in NTP or manual changes by user. NTP (Network Time Protocol) is an application which facilitates time synchronization of systems across a network. Realtime clock is not affected by system suspension.

- **`CLOCK_MONOTONIC`**

This is a non-settable system clock which measures time from a specific point in the past. In Linux, it is the time since the system was booted. This clock is not affected by any manual changes by user, but it is affected by changes in NTP. Monotonic clock does not count the time system was suspended.

- **`CLOCK_MONOTONIC_RAW`**

This clock is like monotonic clock. But it provides raw hardware timing which is not subject to change with changes in NTP. This clock also does not measure time when the system was suspended.

In my opinion, **`CLOCK_MONOTONIC_RAW`** is best to use in this application. For this real-time clock, instead of getting real-time we are testing the accuracy of the clock. Also, we do not want any arbitrary changes in time difference which could be made by the user. Here, any changes in the clock time will inhibit us from making an informed decision regarding the performance of the designed real time clock. These changes are also possible with `CLOCK_MONOTONIC`. Hence, `CLOCK_MONOTONIC_RAW` is the best clock to use for tasks requiring time synchronization.

- c. 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?

- **Low Interrupt Handler Latency**

Interrupt latency is the time required by system from when the interrupt was requested to when the interrupt routine source was serviced. This interrupt latency depends on processor design and utilization, interrupt overhead and time taken for system to execute a command. Normally, while designing any process, overhead and latency period are considered. The commands to be executed in the interrupt handler should also be minimized. Moreover, these interrupts can be disabled while handling critical sections of the code. All interrupt handlers as well as disabling interrupts and enabling interrupts in critical sections in RTOS have a prologue and epilogue. Hence, RTOS developers have highly optimized their interrupt handlers. Most real time systems which have hard deadlines must complete a task in a particular time frame. For this it is vital to guarantee that the interrupt latency will not exceed from certain time. These hard requirements in handling interrupts makes designing real-time systems very complicated. Even a small variation or extension in interrupt latency can cause significant life or assets damage. Also, most real time systems require instant response after an interrupt request. Hence, low interrupt latency is very important in any real time system.

- **Low Context Switch time**

The process of saving the state of currently running task and switching to another task is called context switching. A context switch is also required after completing the pre-empted task and to run the task whose state was previously saved. The number of machine cycles required to perform context switching is called context switch overhead. Context switch has one of the largest overheads in any RTOS interrupt processing. The context switching time depends on this overhead and the type of thread being executed. Context switching steps involve saving the state of previous task and then executing another task. Switching between tasks requires some CPU cycles and consumes time. During

this time, the CPU does nothing productive. Hence, context switching is purely overhead. In RTOS, each context switch can take 50-500 cycles. Time taken by context switching can impact real-time responsiveness significantly. Hence, code optimization to reduce context switches is crucial for real time systems.

- **Stable Timer Services**

Timers are used in real-time systems to run a task for certain amount of time or queue a task for near future. RTOSs have timer services to execute these functionalities. Most important aspect of any real time system is accurate timing to run a task, schedule a task, meet deadlines, or respond to a request. Even a few milliseconds of a delay in response can be fatal for the system. To avoid any such undesirable outcomes, a stable timer with low jitter and drift should be designed. Of course, it is impossible to design a perfect timer with 100% accuracy but a good enough timer which would not cause the real time system to fail is sufficient.

- d. Do you believe the accuracy provided by the example RT-Clock code? Why or why not?

Below I have attached the screenshots of the output of RTclock.

```

saloni@saloni-VBUbuntu:~/exercise1/posix$ 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

test 0
MY_CLOCK start seconds = 5762, nanoseconds = 496574275
MY_CLOCK clock stop seconds = 5762, nanoseconds = 506756198
MY_CLOCK clock DT seconds = 0, msec=10, usec=10181, nsec=10181923, sec=0.010181923
Requested sleep seconds = 0, nanoseconds = 10000000
Sleep loop count = 0
MY_CLOCK delay error = 0, nanoseconds = 181923

test 1
MY_CLOCK start seconds = 5762, nanoseconds = 506790558
MY_CLOCK clock stop seconds = 5762, nanoseconds = 516832411
MY_CLOCK clock DT seconds = 0, msec=10, usec=10041, nsec=10041853, sec=0.010041853
Requested sleep seconds = 0, nanoseconds = 10000000
Sleep loop count = 0
MY_CLOCK delay error = 0, nanoseconds = 41853

test 2
MY_CLOCK start seconds = 5762, nanoseconds = 516856098
MY_CLOCK clock stop seconds = 5762, nanoseconds = 527423712
MY_CLOCK clock DT seconds = 0, msec=10, usec=10567, nsec=10567614, sec=0.010567614
Requested sleep seconds = 0, nanoseconds = 10000000
Sleep loop count = 0
MY_CLOCK delay error = 0, nanoseconds = 567614

test 97
MY_CLOCK start seconds = 5763, nanoseconds = 489371626
MY_CLOCK clock stop seconds = 5763, nanoseconds = 499412980
MY_CLOCK clock DT seconds = 0, msec=10, usec=10041, nsec=10041354, sec=0.010041354
Requested sleep seconds = 0, nanoseconds = 10000000
Sleep loop count = 0
MY_CLOCK delay error = 0, nanoseconds = 41354

test 98
MY_CLOCK start seconds = 5763, nanoseconds = 499440751
MY_CLOCK clock stop seconds = 5763, nanoseconds = 509469193
MY_CLOCK clock DT seconds = 0, msec=10, usec=10028, nsec=10028442, sec=0.010028442
Requested sleep seconds = 0, nanoseconds = 10000000
Sleep loop count = 0
MY_CLOCK delay error = 0, nanoseconds = 28442

test 99
MY_CLOCK start seconds = 5763, nanoseconds = 509491331
MY_CLOCK clock stop seconds = 5763, nanoseconds = 519700406
MY_CLOCK clock DT seconds = 0, msec=10, usec=10209, nsec=10209075, sec=0.010209075
Requested sleep seconds = 0, nanoseconds = 10000000
Sleep loop count = 0
MY_CLOCK delay error = 0, nanoseconds = 209075

TEST COMPLETE

Code start time - 5762 sec, 496366421 nsec
Code stop time - 5763 sec, 519762628 nsec
Time taken to execute the code is 1 sec, 23 msec, 23396 usec, 23396207 nsec

```

From the above output we can conclude that, the time difference is not always the specified sleep\_time i.e., 10ms. The time difference is always a bit higher than required. We can observe that there is delay error of a fraction of milliseconds in every test case. Moreover, this delay error is not constant across

all test cases. The study of time difference for 10 test cases to 200 test cases showed similar output.

```
test 198
MY_CLOCK start seconds = 5269, nanoseconds = 534853959
MY_CLOCK clock stop seconds = 5269, nanoseconds = 544915104
MY_CLOCK clock DT seconds = 0, msec=10, usec=10061, nsec=10061145, sec=0.010061145
Requested sleep seconds = 0, nanoseconds = 10000000
Sleep loop count = 0
MY_CLOCK delay error = 0, nanoseconds = 61145

test 199
MY_CLOCK start seconds = 5269, nanoseconds = 544949749
MY_CLOCK clock stop seconds = 5269, nanoseconds = 555395834
MY_CLOCK clock DT seconds = 0, msec=10, usec=10446, nsec=10446085, sec=0.010446085
Requested sleep seconds = 0, nanoseconds = 10000000
Sleep loop count = 0
MY_CLOCK delay error = 0, nanoseconds = 446085

TEST COMPLETE

Code start time - 5267 sec, 508755527 nsec
Code stop time - 5269 sec, 555537424 nsec
Time taken to execute the code is 2 sec, 46 msec, 46781 usec, 46781897 nsec
```

### OUTPUT FOR 200 TEST CASES

Also, use of a different clock time did not improve the timing response.

```
test 98
MY_CLOCK start seconds = 1623471134, nanoseconds = 146351720
MY_CLOCK clock stop seconds = 1623471134, nanoseconds = 156416789
MY_CLOCK clock DT seconds = 0, msec=10, usec=10065, nsec=10065069, sec=0.010065079
Requested sleep seconds = 0, nanoseconds = 10000000
Sleep loop count = 0
MY_CLOCK delay error = 0, nanoseconds = 65069

test 99
MY_CLOCK start seconds = 1623471134, nanoseconds = 156458808
MY_CLOCK clock stop seconds = 1623471134, nanoseconds = 166524857
MY_CLOCK clock DT seconds = 0, msec=10, usec=10066, nsec=10066049, sec=0.010066032
Requested sleep seconds = 0, nanoseconds = 10000000
Sleep loop count = 0
MY_CLOCK delay error = 0, nanoseconds = 66049

TEST COMPLETE

Code start time - 1623471133 sec, 146566834 nsec
Code stop time - 1623471134 sec, 166638229 nsec
Time taken to execute the code is 1 sec, 20 msec, 20071 usec, 20071395 nsec
```

### OUTPUT OF REALTIME CLOCK

```

test 98
stop is earlier than start
MY_CLOCK start seconds = 1623471251, nanoseconds = 701428326
MY_CLOCK clock stop seconds = 1623471251, nanoseconds = 709428264
MY_CLOCK clock DT seconds = 0, msec=7, usec=7999, nsec=7999938, sec=0.007999897
Requested sleep seconds = 0, nanoseconds = 10000000
Sleep loop count = 0
MY_CLOCK delay error = 0, nanoseconds = 1999905

test 99
MY_CLOCK start seconds = 1623471251, nanoseconds = 709428264
MY_CLOCK clock stop seconds = 1623471251, nanoseconds = 721428169
MY_CLOCK clock DT seconds = 0, msec=11, usec=11999, nsec=11999905, sec=0.011999846
Requested sleep seconds = 0, nanoseconds = 10000000
Sleep loop count = 0
MY_CLOCK delay error = 0, nanoseconds = 1999905

TEST COMPLETE

Code start time - 1623471250 sec, 701436217 nsec
Code stop time - 1623471251 sec, 721428169 nsec
Time taken to execute the code is 1 sec, 19 msec, 19991 usec, 19991952 nsec

```

### OUTPUT OF REALTIME\_COARSE CLOCK

```

test 98
MY_CLOCK start seconds = 5416, nanoseconds = 342573174
MY_CLOCK clock stop seconds = 5416, nanoseconds = 352678520
MY_CLOCK clock DT seconds = 0, msec=10, usec=10105, nsec=10105346, sec=0.010105346
Requested sleep seconds = 0, nanoseconds = 10000000
Sleep loop count = 0
MY_CLOCK delay error = 0, nanoseconds = 105346

test 99
MY_CLOCK start seconds = 5416, nanoseconds = 352739736
MY_CLOCK clock stop seconds = 5416, nanoseconds = 362953829
MY_CLOCK clock DT seconds = 0, msec=10, usec=10214, nsec=10214093, sec=0.010214093
Requested sleep seconds = 0, nanoseconds = 10000000
Sleep loop count = 0
MY_CLOCK delay error = 0, nanoseconds = 214093

TEST COMPLETE

Code start time - 5415 sec, 339219727 nsec
Code stop time - 5416 sec, 363047356 nsec
Time taken to execute the code is 1 sec, 23 msec, 23827 usec, 23827629 nsec

```

### OUTPUT OF MONOTONIC CLOCK



```

test 98
stop is earlier than start
MY_CLOCK start seconds = 5521, nanoseconds = 352994518
MY_CLOCK clock stop seconds = 5521, nanoseconds = 360994460
MY_CLOCK clock DT seconds = 0, msec=7, usec=7999, nsec=7999942, sec=0.007999942
Requested sleep seconds = 0, nanoseconds = 10000000
Sleep loop count = 0
MY_CLOCK delay error = 0, nanoseconds = 1999915

test 99
MY_CLOCK start seconds = 5521, nanoseconds = 360994460
MY_CLOCK clock stop seconds = 5521, nanoseconds = 372994375
MY_CLOCK clock DT seconds = 0, msec=11, usec=11999, nsec=11999915, sec=0.011999915
Requested sleep seconds = 0, nanoseconds = 10000000
Sleep loop count = 0
MY_CLOCK delay error = 0, nanoseconds = 1999915

TEST COMPLETE

Code start time - 5520 sec, 353001673 nsec
Code stop time - 5521 sec, 372994375 nsec
Time taken to execute the code is 1 sec, 19 msec, 19992 usec, 19992702 nsec

```

### OUTPUT OF MONOTONIC\_COARSE CLOCK

This inaccuracy is because of the use of nanosleep. Implementation of system sleep for certain time to control real time systems is not a recommended design practice. Hence, it can be concluded that the given example of real time clock is not highly accurate.

4. This is a challenging problem that requires you to learn quite a bit about Pthreads in Linux and to implement a schedule that is predictable.
  - a. Download, build and run code in <http://ecee.colorado.edu/~ecen5623/ecen/ex/Linux/simplethread/> and describe how it works and what it does.

**Code Gist:** This code is to demonstrate creating POSIX threads and executing these threads simultaneously like parallel programming. All the threads created are executed with the task to calculate summation of numbers.

```

/*****
/* Code to demonstrate working of pthreads and
/* perform summation of numbers as task for each thread */
/*****/

#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <sched.h>

#define NUM_THREADS 64 //number of threads to be executed

typedef struct //required thread parameters
{
    int threadIdx; //threadidx denotes thread number
} threadParams_t;

// POSIX thread declarations and scheduling attributes
pthread_t threads[NUM_THREADS];
threadParams_t threadParams[NUM_THREADS];

```

In the above code snippet, a structure is defined for the thread attributes wherein the structures have threadIdx (thread number) as its parameters. Further, arrays are defined for thread and thread attributes.

```

/*****/
/* Function to perform summation as the task assigned to each thread */
/* Parameters: *threadp - taken from argument passed in pthread_create */
/* (here thread number) */
/* Return type:void */
/*****/
void *counterThread(void *threadp)
{
    int sum=0, i;
    threadParams_t *threadParams = (threadParams_t *)threadp; //set the parameters of the thread as parameters passed in function argument

    // printf("Executing thread number %d\n",threadParams->threadIdx);

    for(i=1; i < (threadParams->threadIdx)+1; i++) //compute the summation of all numbers from 1 to threadIdx(currently running thread)
        sum=sum+i;

    printf("Thread idx=%d, sum[1..%d]=%d\n", //print the output of the summation
        threadParams->threadIdx,
        threadParams->threadIdx, sum);

    // printf("Terminated thread number %d\n",threadParams->threadIdx);
}

```

The function defined in above snippet void \*counterthread(void \*threadp) is invoked after a thread is created. This function contains the task to be executed by the thread. First, thread attributes threadParams is set based on threadp argument passed while thread creation. This thread attribute is the thread number. After assigning attributes, summation of numbers from 1 to i is computed. Here i is the currently running thread number i.e., threadIdx. Then the computation is printed on the terminal.

```

int main (int argc, char *argv[])
{
    int rc;
    int i;

    for(i=1; i <= NUM_THREADS; i++)    //create NUM_THREADS number of threads to be executed
    {
        threadParams[i].threadIdx=i;    //set thread parameter threadIdx as the thread number

        pthread_create(&threads[i],    // pointer to thread descriptor
                      (void *)0,      // use default attributes
                      counterThread,  // thread function entry point(function to be invoked after the thread
is created)
                      (void *)&(threadParams[i]) // parameters to pass in
                      );
        // printf("thread number %d created\n",i);
        // pthread_join(threads[i], NULL);
        // printf("thread number %d executed\n",i);
    }

    for(i=1;i<=NUM_THREADS;i++) {      //join all threads
        pthread_join(threads[i], NULL);
        // printf("thread number %d executed\n",i);
    }

    printf("TEST COMPLETE\n");
}

```

In above snippet, main code thread is shown. In the first part of the function, NUM\_THREADS number of threads are created. Attributes of each thread is set, and the thread is created using pthread\_create function. counterthread function is invoked after a thread is created with the threadIdx as argument. In the second part of the code, pthread\_join is called to synchronize termination of all threads.

## OUTPUT OF PTHREAD.C

```

saloni@saloni-VBUbuntu:~/exercise1/pthread$ make
gcc -O0 -c pthread.c
gcc -O0 -o pthread pthread.o -lpthread
saloni@saloni-VBUbuntu:~/exercise1/pthread$ sudo ./pthread
Thread idx=7, sum[1...7]=28
Thread idx=8, sum[1...8]=36
Thread idx=9, sum[1...9]=45
Thread idx=10, sum[1...10]=55
Thread idx=11, sum[1...11]=66
Thread idx=12, sum[1...12]=78
Thread idx=6, sum[1...6]=21
Thread idx=13, sum[1...13]=91
Thread idx=14, sum[1...14]=105
Thread idx=15, sum[1...15]=120
Thread idx=16, sum[1...16]=136
Thread idx=17, sum[1...17]=153
Thread idx=18, sum[1...18]=171
Thread idx=19, sum[1...19]=190
Thread idx=20, sum[1...20]=210
Thread idx=5, sum[1...5]=15
Thread idx=21, sum[1...21]=231
Thread idx=22, sum[1...22]=253
Thread idx=23, sum[1...23]=276
Thread idx=24, sum[1...24]=300
Thread idx=25, sum[1...25]=325
Thread idx=26, sum[1...26]=351
Thread idx=27, sum[1...27]=378
Thread idx=28, sum[1...28]=406
Thread idx=30, sum[1...30]=465
Thread idx=29, sum[1...29]=435
Thread idx=4, sum[1...4]=10
Thread idx=31, sum[1...31]=496
Thread idx=32, sum[1...32]=528
t VBox_GAs_6.1.22 sum[1...33]=561
Thread idx=34, sum[1...34]=595
Thread idx=35, sum[1...35]=630
Thread idx=36, sum[1...36]=666
Thread idx=37, sum[1...37]=703
Thread idx=38, sum[1...38]=741
Thread idx=39, sum[1...39]=780
Thread idx=40, sum[1...40]=820
Thread idx=41, sum[1...41]=861
Thread idx=42, sum[1...42]=903
Thread idx=43, sum[1...43]=946
Thread idx=3, sum[1...3]=6
Thread idx=44, sum[1...44]=990
Thread idx=45, sum[1...45]=1035
Thread idx=46, sum[1...46]=1081
Thread idx=63, sum[1...63]=2016
Thread idx=58, sum[1...58]=1711
Thread idx=64, sum[1...64]=2080
Thread idx=52, sum[1...52]=1378
Thread idx=51, sum[1...51]=1326
Thread idx=57, sum[1...57]=1653
Thread idx=47, sum[1...47]=1128
Thread idx=50, sum[1...50]=1275
Thread idx=53, sum[1...53]=1431
Thread idx=55, sum[1...55]=1540
Thread idx=56, sum[1...56]=1596
Thread idx=62, sum[1...62]=1953
Thread idx=48, sum[1...48]=1176
Thread idx=49, sum[1...49]=1225
Thread idx=54, sum[1...54]=1485
Thread idx=59, sum[1...59]=1770
Thread idx=60, sum[1...60]=1830
Thread idx=61, sum[1...61]=1891
Thread idx=2, sum[1...2]=3
Thread idx=32553, sum[1...32553]=529865181
TEST COMPLETE

```

The above screenshot shows the computation result of 64 threads. We can observe from the output that the output of the threads is not in sequence. This is because all threads are executed parallelly. The output of the thread which completes the computation first is printed first. For every execution of this code, this sequence varies. Thread pre-emption and interrupt are a major factor in this.

- b. Download and run the examples for creation of 2 threads provided by `incdecthread/pthread.c`, as well as `testdigest.c` which make use of `SCHED_FIFO` and `sem_post` and `sem_wait`. Describe POSIX API functions used by reading of POSIX manual pages as needed.

## OUTPUT OF INCDECTHREAD/PTHREAD.C

```
saloni@saloni-VBUbuntu:~/exercise1/incdecthread$ sudo ./pthread
Decrement thread idx=1, gsum=0
Decrement thread idx=1, gsum=-1
Decrement thread idx=1, gsum=-3
Decrement thread idx=1, gsum=-6
Decrement thread idx=1, gsum=-10
Decrement thread idx=1, gsum=-15
Decrement thread idx=1, gsum=-21
Decrement thread idx=1, gsum=-28
Decrement thread idx=1, gsum=-36
Decrement thread idx=1, gsum=-45
Decrement thread idx=1, gsum=-55
Decrement thread idx=1, gsum=-66
Decrement thread idx=1, gsum=-78
Decrement thread idx=1, gsum=-91
Decrement thread idx=1, gsum=-105
Decrement thread idx=1, gsum=-120
Decrement thread idx=1, gsum=-136
Decrement thread idx=1, gsum=-153
Decrement thread idx=1, gsum=-171
Decrement thread idx=1, gsum=-190
Decrement thread idx=1, gsum=-210
Decrement thread idx=1, gsum=-231
Decrement thread idx=1, gsum=-253
Decrement thread idx=1, gsum=-276
Decrement thread idx=1, gsum=-300
Decrement thread idx=1, gsum=-325
Decrement thread idx=1, gsum=-351
Decrement thread idx=1, gsum=-378
Decrement thread idx=1, gsum=-406
Decrement thread idx=1, gsum=-435
Decrement thread idx=1, gsum=-465
Decrement thread idx=1, gsum=-496
Decrement thread idx=1, gsum=-528
Decrement thread idx=1, gsum=-561
Decrement thread idx=1, gsum=-595
Decrement thread idx=1, gsum=-630
Decrement thread idx=1, gsum=-666
Decrement thread idx=1, gsum=-703
Decrement thread idx=1, gsum=-741
Decrement thread idx=1, gsum=-780
Decrement thread idx=1, gsum=-820
```

```
Increment thread idx=0, gsum=-37259
Increment thread idx=0, gsum=-36297
Increment thread idx=0, gsum=-35334
Increment thread idx=0, gsum=-34370
Increment thread idx=0, gsum=-33405
Increment thread idx=0, gsum=-32439
Increment thread idx=0, gsum=-31472
Increment thread idx=0, gsum=-30504
Increment thread idx=0, gsum=-29535
Increment thread idx=0, gsum=-28565
Increment thread idx=0, gsum=-27594
Increment thread idx=0, gsum=-26622
Increment thread idx=0, gsum=-25649
Increment thread idx=0, gsum=-24675
Increment thread idx=0, gsum=-23700
Increment thread idx=0, gsum=-22724
Increment thread idx=0, gsum=-21747
Increment thread idx=0, gsum=-20769
Increment thread idx=0, gsum=-19790
Increment thread idx=0, gsum=-18810
Increment thread idx=0, gsum=-17829
Increment thread idx=0, gsum=-16847
Increment thread idx=0, gsum=-15864
Increment thread idx=0, gsum=-14880
Increment thread idx=0, gsum=-13895
Increment thread idx=0, gsum=-12909
Increment thread idx=0, gsum=-11922
Increment thread idx=0, gsum=-10934
Increment thread idx=0, gsum=-9945
Increment thread idx=0, gsum=-8955
Increment thread idx=0, gsum=-7964
Increment thread idx=0, gsum=-6972
Increment thread idx=0, gsum=-5979
Increment thread idx=0, gsum=-4985
Increment thread idx=0, gsum=-3990
Increment thread idx=0, gsum=-2994
Increment thread idx=0, gsum=-1997
Increment thread idx=0, gsum=-999
Increment thread idx=0, gsum=0
TEST COMPLETE
```

To run this code, I made relevant makefile and a file for the code. The code ran without any errors or warnings and produced above output. Initially confused by the output sequence, I also modified the code by changing the `pthread_join` function to get sequenced output. After making those changes, I got more insight on working of the original code.

## OUTPUT OF TESTDIGEST.C

```
saloni@saloni-VBUBuntu:~/exercise1/testdigest$ make
cc -O0 -g -c testdigest.c
cc -O0 -g -o testdigest testdigest.o md5.o sha1.o crc.o sha2.o -lpthread
saloni@saloni-VBUBuntu:~/exercise1/testdigest$ sudo ./testdigest
Will default to 4 synthetic IO workers

***** MULTI THREAD TESTS
Pthread Policy is SCHED_OTHER
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM

***** TOTAL PERFORMANCE SUMMARY
For 4 threads, Total rate=2262204.573797
saloni@saloni-VBUBuntu:~/exercise1/testdigest$ vim Makefile
saloni@saloni-VBUBuntu:~/exercise1/testdigest$ make
make: Nothing to be done for 'all'.
saloni@saloni-VBUBuntu:~/exercise1/testdigest$ sudo ./testdigest
Will default to 4 synthetic IO workers

***** MULTI THREAD TESTS
Pthread Policy is SCHED_OTHER
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM

***** TOTAL PERFORMANCE SUMMARY
For 4 threads, Total rate=2308151.731999
```

To execute this code, I imported all relevant header files and source files. Initial compilation of code showed a few warnings. To eliminate these warnings, I changed the code at some points. The above outputs were produced after running the code.

### POSIX API functions used in testdigest.c

- sched\_getscheduler (pid\_t pid) – returns the current scheduling policy of the thread identified by pid. Different real time and non-real time scheduling policies have been explained above.
- getpid() – returns the process ID of the calling function
- pthread\_attr\_init (pthread\_attr\_t \*attr) - initializes thread attributes objects pointed to by attr to default values
- pthread\_attr\_setinheritsched (pthread\_attr\_t \*attr, int inheritsched) - sets the inherit-scheduler attributes of the thread attributes objects referred to by attr to the value specified in inheritsched.



The values of `inheritsched` may be `PTHREAD_INHERIT_SCHED` or `PTHREAD_EXPLICIT_SCHED`. For `PTHREAD_INHERIT_SCHED`, threads inherit their attributes from the creating thread and `attr` is ignored. For `PTHREAD_EXPLICIT_SCHED`, threads take their scheduling attributes from values specified by attributes objects.

- `pthread_attr_setschedpolicy (pthread_attr_t *attr, int policy)` - sets the scheduling policy of the thread referred to by `attr` to value specified in `policy`. Valid entries for `policy` are `SCHED_FIFO`, `SCHED_RR` or `SCHED_OTHER`.
- `sched_get_priority_max (int policy)` - returns the maximum priority that can be set for the scheduling algorithm identified by `policy`
- `sched_get_priority_min (int policy)` - returns the minimum priority that can be set for the scheduling algorithm identified by `policy`
- `sched_getparam (pid_t pid, struct sched_param *param)` – retrieves scheduling parameters for thread identified by `pid`
- `sched_setscheduler (pid_t pid, int policy, struct sched_param *param)` - sets scheduling policy and parameters for the threads whose id is specified by `pid`
- `pthread_attr_getscope (const pthread_attr_t *restrict attr, int *restrict contentscope)` – get the `contentscope` attribute in the `attr` object
- `pthread_attr_setschedparam (pthread_attr_t *attr, const struct sched_param *param)` - sets the scheduling parameters of the thread referred to by `attr` to values specified in `param`
- `sem_wait (sem_t *sem)` – decrements (locks) the semaphore pointed to by `sem`. If the value of semaphore is less than 0 then it gets locked until its value rises above 0
- `gettimeofday (struct timeval *restrict tv, struct timezone *restrict tz)` – gets time as well as timezone. Structure `timeval` has parameters for time value in seconds and microseconds. Structure `timezone` has parameters minutes west of

Greenwich and type of DST correction. The time value returned by this function is affected by the discontinuous jumps in system time.

- perror (const char \*s) – produces a message on standard error
- pthread\_create (pthread\_t \*restrict thread, const pthread\_attr\_t \*restrict attr, void \*(\*start\_routine) (void \*), void \*restrict arg) - creates a thread with specified name and parameters and invokes the function start\_routine with parameters arg after the thread is created
- sem\_post (sem\_t \*sem) – increments (unlocks) semaphore pointed to by sem
- pthread\_detach (pthread\_t thread) – marks the thread identified by thread detached. After execution of the thread is terminated, the resources are returned to the system
- sem\_destroy (sem\_t \*sem) - destroys semaphore pointed to by address in sem

### **RTOS to Linux Porting Requirements**

After widespread use of Linux in all systems, there have been attempts to port RTOS driver, applications, functions, models, and attributes to Linux.

Main aspects to be considered while porting is:

- Memory Management for tasks and threads
- Real time operations – response time of Linux should be good enough
- Emulating RTOS functions – some RTOS functions have equivalent Linux functions, but some functions need recoding
- Map RTOS tasks to Linux Threads
- Most RTOS applications like semaphores, mutex, timers, delays have equivalent Linux functionality. But some RTOS specific functions like watchdog and task registers need to be recoded.

### **THREADS in Linux vs TASKS in RTOS**

A task in RTOS has its own stack like a thread in Linux. In RTOS, a task could be created statically while system boot but in Linux all thread is created dynamically. A task in RTOS is very lightweight as it does not require a memory space, but it shares the memory with other tasks. This also makes context

switching less time consuming which is a requirement in real time system. However, in Linux, all threads require separate virtual memory spaces. Both have completely different API functions for execution. Both threads and tasks have similar setting for scheduling policies. Tasks in RTOS have very strict priority assignment where a lower priority task can be pre-empted by a higher priority task only when it is ready. Whereas for a thread in Linux, a running thread can be interrupted by other threads for various reasons. Generally, threads in Linux and tasks in RTOS mean the same, just different nomenclature for different operating systems. Apart from some basic priority assignments and memory management, these two words are interchangeable.

### **Semaphores in Linux**

According to Merriam-Webster dictionary, a semaphore is a signaling mechanism. In programming, a semaphore is a non-negative integer shared between threads which provides low-level task synchronization and are used in critical sections of the code. Semaphores are atomic functions and are used to avoid race conditions. There are two types of semaphores: Counting semaphores (used for arbitrary resource count) and Binary semaphore (used to manage locking and unlocking). Semaphores are quite simple and can be used for multiple threads. The two main operations on semaphores is wait() and signal().

Wait() decrements the variable value by 1. If the value of variable becomes negative, then the processing is blocked.

Signal() increments the variable value by 1.

Although semaphores are simple and convenient, it is required to keep track of all calls to wait and signal. Also, wait() and signal() should be used in correct sequence to avoid deadlocks. Moreover, the biggest disadvantage of semaphores is priority inversion where it can give access to the critical section of the code to low priority tasks.

### **Semaphores in RTOS**

In RTOS, semaphores are used for task synchronization as well as accessing shared resources. Whenever a task needs to access a resource shared with other tasks, it must acquire a semaphore and then after using the resources it must release the semaphore. Here semaphore works like a key to the common room

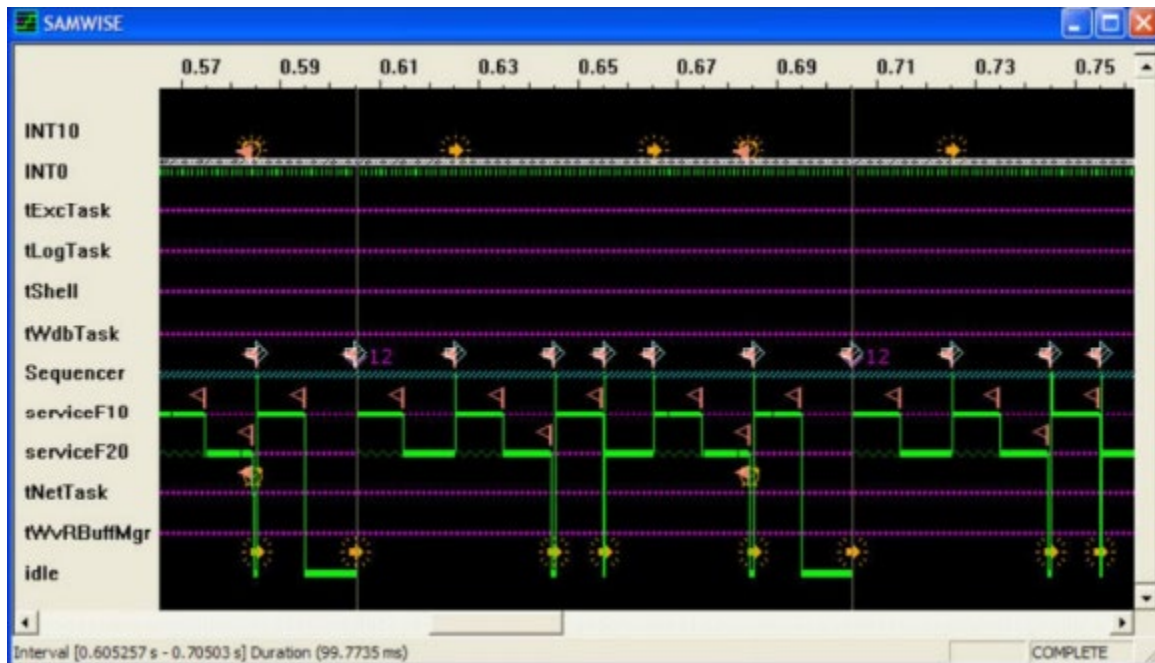
and only one person can access the room at a time. Moreover, even when a higher priority task needs access to the resources, it must wait until the lower priority task releases the semaphore. This can perhaps cause the task to miss deadlines, which is why resource sharing should be optimized to avoid missing deadlines.

RTOS semaphores have 3 operations: create, acquire, and release. RTOS also has different types of semaphores: binary resource semaphore (to manage access of a single resource) and multiple resource semaphore (to regulate access to multiple resources). RTOS and Linux have different API functions for semaphores.

### **Synthetic Workload Generation**

All real time systems have strict performance requirements, so rigorous testing and evaluation of the system is mandatory. In most real time systems simulation of real workload is not possible. In such cases, a synthetic workload is generated. A software system generates synthetic workloads to replicate the real workload. This synthetic workload evaluates the performance and accuracy of the designed system. The compiler is generally called the Synthetic Workload Generator (SWG) which produces executable synthetic workloads. In synthetic workloads, the user can generate different types of loads involving read, write functions. User can control many aspects of the workload like probability and intensity.

- c. Describe how you would attempt to implement Linux code to replicate the LCM invariant schedule implemented in the VxWorks RTOS as shown below:



The observed timing above fits our theory for RM policy on a priority preemptive scheduling system as shown by the timing diagram below:

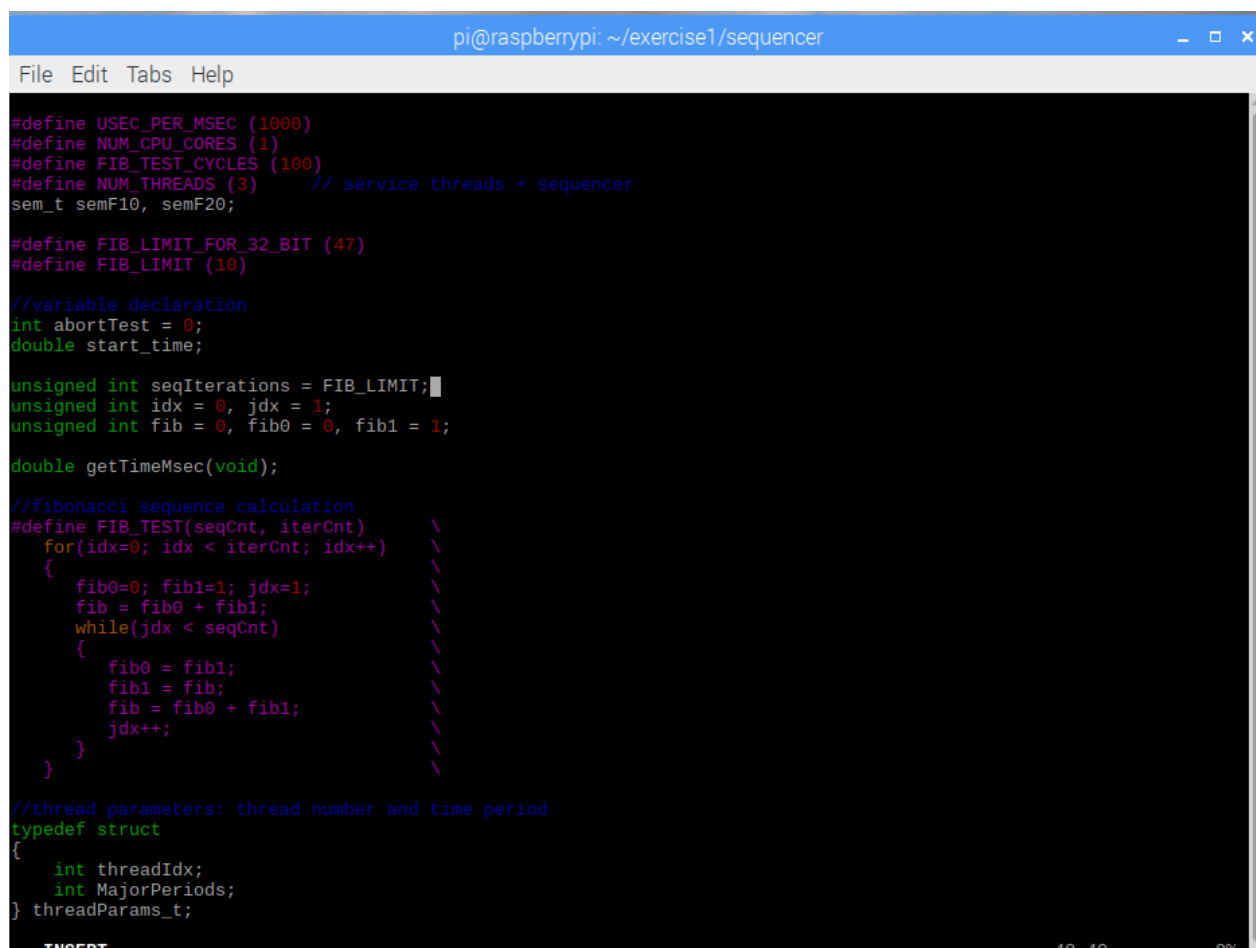
Example 5	T1	2	C1	1	U1	0.5	LCM =	10		
	T2	5	C2	2	U2	0.4				
	T3	10	C3	1	U3	0.1	U <sub>tot</sub> =	1		
RM Schedule	1	2	3	4	5	6	7	8	9	10
S1										
S2										
S3										

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 produce reliable 10 millisecond and 20 millisecond loads on your Raspberry Pi. Describe whether your able to achieve predictable reliable results in terms of the C (CPU time) values alone and how you would sequence execution.

Unable to implement the Linux code for the scheduler, I ran the solution code written by Dr. Siewert.

**Code Gist:** This code demonstrates the design of a sequencer for a scheduling policy for two service tasks. These threads perform the summation of Fibonacci sequence for 10ms and 20ms. This code is a hard-coded scheduler. Based on the terminal output, we can check the accuracy of the scheduler.

In the below code snippet, firstly variables and #defines are defined for the sequencer. After that a #define is used to calculate the Fibonacci sequence for specified number of cycles. A structure is defined for the thread attributes. The parameters of this structure are threadIdx – thread number and MajorPeriods – the time for which the thread is to be executed.



```

pi@raspberrypi: ~/exercise1/sequencer
File Edit Tabs Help

#define USEC_PER_MSEC (1000)
#define NUM_CPU_CORES (1)
#define FIB_TEST_CYCLES (100)
#define NUM_THREADS (3) // service threads + sequencer
sem_t semF10, semF20;

#define FIB_LIMIT_FOR_32_BIT (47)
#define FIB_LIMIT (10)

//variable declaration
int abortTest = 0;
double start_time;

unsigned int seqIterations = FIB_LIMIT;
unsigned int idx = 0, jdx = 1;
unsigned int fib = 0, fib0 = 0, fib1 = 1;

double getTimeMsec(void);

//fibonacci sequence calculation
#define FIB_TEST(seqCnt, iterCnt) \
{ \
    fib0=0; fib1=1; jdx=1; \
    fib = fib0 + fib1; \
    while(jdx < seqCnt) \
    { \
        fib0 = fib1; \
        fib1 = fib; \
        fib = fib0 + fib1; \
        jdx++; \
    } \
}

//thread parameters: thread number and time period
typedef struct
{
    int threadIdx;
    int MajorPeriods;
} threadParams_t;

```

In the below code snippet, a function is defined which is to be executed by the thread. In this function, the thread must compute the Fibonacci sequence for 10ms. Before starting the Fibonacci sequence computation, the function calibrates the number of cycles required to run the sequence for 10ms. To calibrate the cycle number, the Fibonacci sequence is run for 100 cycles and time taken for it is calculated. Based on this period, number of cycle requirements to run the sequence for 10ms is calculated. After that, the start time and stop time of the sequence computation of the thread is printed on the terminal. Similar function is written to compute Fibonacci sequence for 20ms.

```

pi@raspberrypi: ~/exercise1/sequencer
File Edit Tabs Help
void *fib10(void *threadp)
{
    double event_time, run_time=0.0;
    int limit=0, release=0, cpucore, i;
    threadParams_t *threadParams = (threadParams_t *)threadp;
    unsigned int required_test_cycles;

    // Assume FIB_TEST short enough that preemption risk is minimal
    //calibrate number of cycles that fibonacci sequence can run in 10ms
    FIB_TEST(seqIterations, FIB_TEST_CYCLES); //warm cache
    event_time=getTimeMsec();
    FIB_TEST(seqIterations, FIB_TEST_CYCLES);
    run_time=getTimeMsec() - event_time;

    required_test_cycles = (int)(10.0/run_time);
    printf("F10 runtime calibration %lf msec per %d test cycles, so %u required\n", run_time, FIB_TEST_CYCLES, requi
red_test_cycles);

    //terminate the thread by locking the semaphore
    while(!abortTest)
    {
        sem_wait(&semF10); //decrements semaphore pointed by semF10 by 1

        if(abortTest)
            break;
        else
            release++;

        cpucore=sched_getcpu(); //returns number of CPU on which the calling thread is currently executing
        printf("F10 start %d @ %lf on core %d\n", release, (event_time=getTimeMsec() - start_time), cpucore);
        //calculate start time of the thread

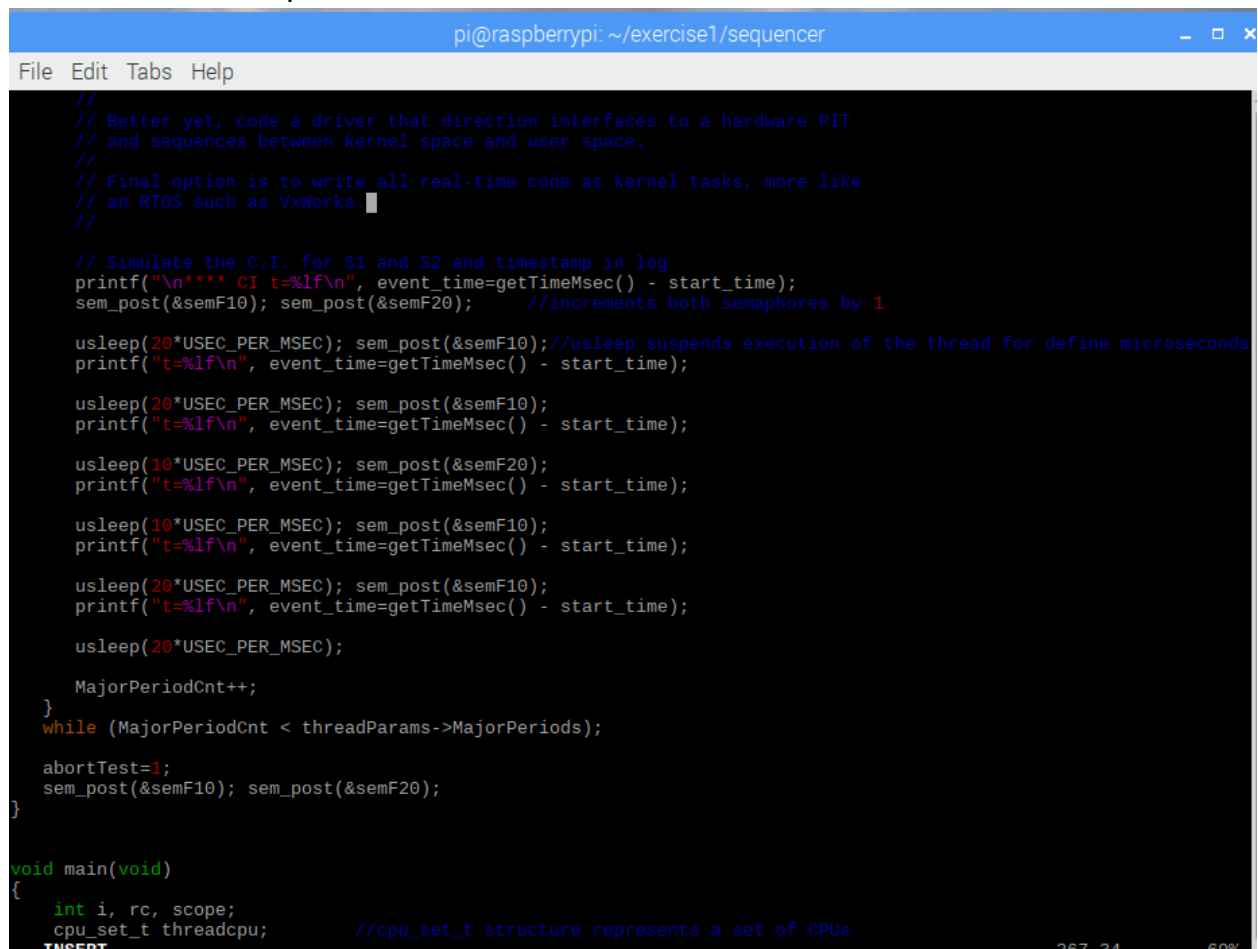
        do
        {
            FIB_TEST(seqIterations, FIB_TEST_CYCLES);
            limit++;
        }
        while(limit < required_test_cycles);

        printf("F10 complete %d @ %lf, %d loops\n", release, (event_time=getTimeMsec() - start_time), limit);
        //calculate completion time of the thread
        limit=0;

-- INSERT --
99,45 23%

```

In the below snippet, working of sequencer function is defined which is the highest priority thread to be executed. In this function, the desired scheduling flow is hard-coded and the threads fib10 and fib20 are executed using sem\_post. Sem\_post function unlocks the semaphore specified in the argument. For this sequence, fib10 is executed at 0ms, 20ms, 40ms, 60ms and 80ms. The thread fib20 is executed at 0ms but since fib10 has higher priority it is pre-empted. Second time fib20 is executed at 50ms. This sequence is hard coded to get above mentioned scheduling policy. Here, usleep is used to suspend the thread execution for certain microseconds. The use of sleep functions is not a recommended design practice for real-time systems since it creates many anomalies and reduces accurate timing which we will later observe in the output.



```

pi@raspberrypi: ~/exercise1/sequencer
File Edit Tabs Help

//
// Better yet, code a driver that direction interfaces to a hardware PIT
// and sequences between kernel space and user space.
//
// Final option is to write all real-time code as kernel tasks, more like
// an RTOS such as VxWorks.
//

// Simulate the C.I. for S1 and S2 and timestamp in log
printf("\n**** CI t=%lf\n", event_time=getTimeMsec() - start_time);
sem_post(&semF10); sem_post(&semF20); //increments both semaphores by 1

usleep(20*USEC_PER_MSEC); sem_post(&semF10); //usleep suspends execution of the thread for define microseconds
printf("t=%lf\n", event_time=getTimeMsec() - start_time);

usleep(20*USEC_PER_MSEC); sem_post(&semF10);
printf("t=%lf\n", event_time=getTimeMsec() - start_time);

usleep(10*USEC_PER_MSEC); sem_post(&semF20);
printf("t=%lf\n", event_time=getTimeMsec() - start_time);

usleep(10*USEC_PER_MSEC); sem_post(&semF10);
printf("t=%lf\n", event_time=getTimeMsec() - start_time);

usleep(20*USEC_PER_MSEC); sem_post(&semF10);
printf("t=%lf\n", event_time=getTimeMsec() - start_time);

usleep(20*USEC_PER_MSEC);

    MajorPeriodCnt++;
}
while (MajorPeriodCnt < threadParams->MajorPeriods);

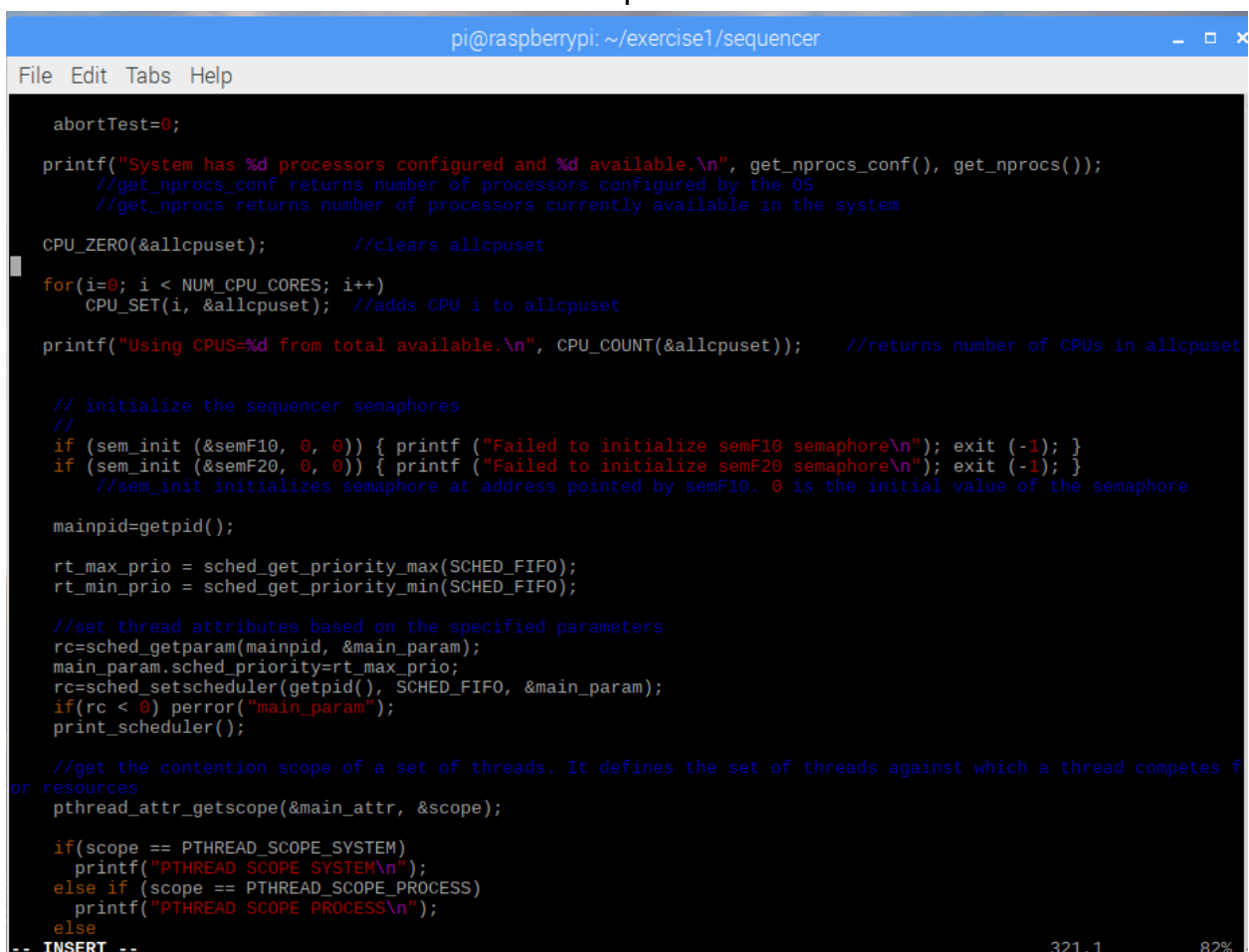
abortTest=1;
sem_post(&semF10); sem_post(&semF20);
}

void main(void)
{
    int i, rc, scope;
    cpu_set_t threadcpu; //cpu_set_t structure represents a set of CPUs
    INSERT
}

```



In the below snippet, basic thread and semaphore initialization is performed to execute the complete task. In the first part, the available CPU code is set. Threads obtain resources for execution from this CPU core. After setting the CPU, two semaphores are initialized for the two threads. Thereafter, scheduling attributes and parameters are set for the threads. The scheduling policy is set to SCHED\_FIFO for real time operation. Then the contention scope is checked for the thread. PTHREAD\_SCOPE\_SYSTEM means the thread competes with all other threads in all the processes which will be executed by the system for resources. PTHREAD\_SCOPE\_PROCESS means the thread will compete for resources with threads within that same process.



```

pi@raspberrypi: ~/exercise1/sequencer
File Edit Tabs Help

abortTest=0;

printf("System has %d processors configured and %d available.\n", get_nprocs_conf(), get_nprocs());
//get_nprocs_conf returns number of processors configured by the OS
//get_nprocs returns number of processors currently available in the system

CPU_ZERO(&allcpuset);          //clears allcpuset
for(i=0; i < NUM_CPU_CORES; i++)
    CPU_SET(i, &allcpuset); //adds CPU i to allcpuset

printf("Using CPUS=%d from total available.\n", CPU_COUNT(&allcpuset)); //returns number of CPUs in allcpuset

// initialize the sequencer semaphores
//
if (sem_init (&semF10, 0, 0)) { printf ("Failed to initialize semF10 semaphore\n"); exit (-1); }
if (sem_init (&semF20, 0, 0)) { printf ("Failed to initialize semF20 semaphore\n"); exit (-1); }
//sem_init initializes semaphore at address pointed by semF10. 0 is the initial value of the semaphore

mainpid=getpid();

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

//set thread attributes based on the specified parameters
rc=sched_getparam(mainpid, &main_param);
main_param.sched_priority=rt_max_prio;
rc=sched_setscheduler(getpid(), SCHED_FIFO, &main_param);
if(rc < 0) perror("main_param");
print_scheduler();

//get the contention scope of a set of threads. It defines the set of threads against which a thread competes f
or resources
pthread_attr_getscope(&main_attr, &scope);

if(scope == PTHREAD_SCOPE_SYSTEM)
    printf("PTHREAD SCOPE SYSTEM\n");
else if (scope == PTHREAD_SCOPE_PROCESS)
    printf("PTHREAD SCOPE PROCESS\n");
else
    printf("PTHREAD SCOPE UNKNOWN\n");

-- INSERT --
321.1 82%

```

In the below snippet, CPU core is set in which the thread is to be executed. Then after, attributes thread parameters are defined for all three threads: sequence, fib10 and fib20. The parameterization involves attribute initialization, setting inheritance of scheduling policy, setting the scheduling policy, setting up the thread for that cpu, assigning priorities and setting scheduling parameters. After all the initialization, the threads are created, and respective functions are invoked.

```

pi@raspberrypi: ~/exercise1/sequencer
File Edit Tabs Help

//set CPU processors for the threads and define thread attributes
for(i=0; i < NUM_THREADS; i++)
{
    CPU_ZERO(&threadcpu);
    CPU_SET(3, &threadcpu);

    rc=pthread_attr_init(&rt_sched_attr[i]);
    rc=pthread_attr_setinheritsched(&rt_sched_attr[i], PTHREAD_EXPLICIT_SCHED);
    rc=pthread_attr_setschedpolicy(&rt_sched_attr[i], SCHED_FIFO);
    rc=pthread_attr_setaffinity_np(&rt_sched_attr[i], sizeof(cpu_set_t), &threadcpu); //sets cpu affinity mask o
f the thread

    rt_param[i].sched_priority=rt_max_prio-i;
    pthread_attr_setschedparam(&rt_sched_attr[i], &rt_param[i]);

    threadParams[i].threadIdx=i;
}

printf("Service threads will run on %d CPU cores\n", CPU_COUNT(&threadcpu));

// Create Service threads which will block awaiting release for:
//
// serviceF10
rc=pthread_create(&threads[1],           // pointer to thread descriptor
                 &rt_sched_attr[1],     // use specific attributes
                 //(void *)0,             // default attributes
                 fib10,                  // thread function entry point
                 (void *)&(threadParams[1]) // parameters to pass in
                );
// serviceF20
rc=pthread_create(&threads[2],           // pointer to thread descriptor
                 &rt_sched_attr[2],     // use specific attributes
                 //(void *)0,             // default attributes
                 fib20,                  // thread function entry point
                 (void *)&(threadParams[2]) // parameters to pass in
                );

// Wait for service threads to calibrate and await release by sequencer
usleep(300000);
-- INSERT --
363,28 94%

```

Below screenshot shows the output of the designed scheduler. The additional printf statements give an insight on the sequence of locking and unlocking of semaphores. The notable output to observe in this is the timing of the thread execution. Although the runtime calibration and cycles calculated based on this time are as accurate as they could be, the threads are executed for significantly less time than desired. In a hard real-time system if this policy were implemented, this difference of a few milliseconds could have been fatal. As seen before, this inaccuracy in timing is because of two main factors: use of usleep in the function and printf statements which have huge overheads. These extra utilization of processor resources, affected the timing analysis. Although we can eliminate the printf statements, but the use of sleep function in this real time system will still drive down the system accuracy. This is exemplified in the next part.

```

pi@raspberrypi: ~/exercise1/sequencer
File Edit Tabs Help
rt_min_prio=1
Service threads will run on 1 CPU cores
F10 runtime calibration 0.109531 msec per 100 test cycles, so 91 required
F20 runtime calibration 0.082760 msec per 100 test cycles, so 241 required
Start sequencer
Starting Sequencer: [S1, T1=20, C1=10], [S2, T2=50, C2=20], U=0.9, LCM=100

**** CI t=0.001615
-----fib10 incremented
-----fib20 incremented
-----fib10 decremented
F10 start 1 @ 0.214895 on core 3
F10 complete 1 @ 7.781792, 91 loops
-----fib20 decremented
F20 start 1 @ 7.865541 on core 3
t=20.204397
-----fib10 incremented
-----fib10 decremented
F10 start 2 @ 20.360386 on core 3
F10 complete 2 @ 27.504316, 91 loops
F20 complete 1 @ 33.234761, 241 loops
-----fib10 incremented
t=40.500149
-----fib10 decremented
F10 start 3 @ 40.648273 on core 3
F10 complete 3 @ 46.366375, 91 loops
-----fib20 incremented
t=50.716201
-----fib20 decremented
F20 start 2 @ 50.836617 on core 3
-----fib10 incremented
t=60.831733
-----fib10 decremented
F10 start 4 @ 60.898243 on core 3
F10 complete 4 @ 66.542908, 91 loops
F20 complete 2 @ 71.501117, 241 loops
-----fib10 incremented
t=81.077745
-----fib10 decremented
F10 start 5 @ 81.233838 on core 3
F10 complete 5 @ 86.972044, 91 loops
**** CI t=101.220226

```

Here I have eliminated all the additional printf statements. We can observe that initially the thread execution timing is roughly the same as we expected. The first test is the best accuracy we can get from a system clock using delay functions. But for the scheduling algorithm to be feasible, it is required for it to behave the same way in long run. Below are the outputs of the thread execution farther ahead in time.

```

pi@raspberrypi: ~/exercise1/sequencer
File Edit Tabs Help
pi@raspberrypi:~ $ cd exercise1
pi@raspberrypi:~/exercise1 $ cd sequencer
pi@raspberrypi:~/exercise1/sequencer $ ls
lab1.c Makefile
pi@raspberrypi:~/exercise1/sequencer $ make
gcc -O0 -g -c lab1.c
gcc -O0 -g -o lab1 lab1.o -lpthread
pi@raspberrypi:~/exercise1/sequencer $ sudo ./lab1
System has 4 processors configured and 4 available.
Using CPUS=1 from total available.
Pthread Policy is SCHED_FIFO
PTHREAD SCOPE SYSTEM
rt_max_prio=99
rt_min_prio=1
Service threads will run on 1 CPU cores
F10 runtime calibration 0.072084 msec per 100 test cycles, so 138 required
F20 runtime calibration 0.055157 msec per 100 test cycles, so 362 required
Start sequencer
Starting Sequencer: [S1, T1=20, C1=10], [S2, T2=50, C2=20], U=0.9, LCM=100

**** CI t=0.000989
F10 start 1 @ 0.139374 on core 3
F10 complete 1 @ 11.531184, 138 loops
F20 start 1 @ 11.597903 on core 3
t=20.158362
F10 start 2 @ 20.267945 on core 3
F10 complete 2 @ 31.633557, 138 loops
t=40.317401
F10 start 3 @ 40.468963 on core 3
t=50.478431
F10 complete 3 @ 52.077492, 138 loops
t=60.599930
F10 start 4 @ 60.658419 on core 3
F10 complete 4 @ 72.004084, 138 loops
F20 complete 1 @ 76.372620, 362 loops
F20 start 2 @ 76.524026 on core 3
t=80.715896
F10 start 5 @ 80.830532 on core 3
F10 complete 5 @ 92.222029, 138 loops

**** CI t=100.835457
F10 start 6 @ 100.979154 on core 3
F10 complete 6 @ 107.063314, 138 loops

```

In this screenshot we can observe that the timing becomes highly inaccurate as we go ahead in time. Although the number of cycles to be performed is still the same, time analyzed by the clock is not what we expected.

```

pi@raspberrypi: ~/exercise1/sequencer
File Edit Tabs Help
F10 complete 6 @ 107.063314, 138 loops
F20 complete 2 @ 114.490493, 362 loops
F20 start 3 @ 114.550285 on core 3
t=120.983506
F10 start 7 @ 121.013246 on core 3
F10 complete 7 @ 125.892667, 138 loops
F20 complete 3 @ 132.193858, 362 loops
t=141.044838
F10 start 8 @ 141.120098 on core 3
F10 complete 8 @ 146.014624, 138 loops
t=151.124149
F20 start 4 @ 151.148889 on core 3
t=161.243044
F10 start 9 @ 161.470804 on core 3
F10 complete 9 @ 166.406059, 138 loops
F20 complete 4 @ 169.181889, 362 loops
t=181.430312
F10 start 10 @ 181.459635 on core 3
F10 complete 10 @ 186.339942, 138 loops

**** CI t=201.495029
F10 start 11 @ 201.615810 on core 3
F10 complete 11 @ 206.512783, 138 loops
F20 start 5 @ 206.542783 on core 3
F20 complete 5 @ 219.258706, 362 loops
t=221.750370
F10 start 12 @ 221.982766 on core 3
F10 complete 12 @ 226.932968, 138 loops
t=241.949566
F10 start 13 @ 241.984982 on core 3
F10 complete 13 @ 246.877477, 138 loops
t=252.110231
F20 start 6 @ 252.320856 on core 3
t=262.294282
F10 start 14 @ 262.366365 on core 3
F10 complete 14 @ 267.254172, 138 loops
F20 complete 6 @ 270.119950, 362 loops
t=282.510561
F10 start 15 @ 282.726862 on core 3
F10 complete 15 @ 287.655867, 138 loops

TEST COMPLETE
pi@raspberrypi:~/exercise1/sequencer $

```

In this example, to test the accuracy and feasibility of the designed sequencer, a Fibonacci sequence was executed as a synthetic workload. The threads were asked to execute the sequence to get an idea of the scheduling algorithm implemented in the code. From the above outputs it is clear that the algorithm is not highly accurate and also the synthetic workload failed to analyze the system in greater detail. Instead of a hard coded scheduler, a generic sequencer would have given more precise output.

## References:

- Rate Monotonic Policy – Real-Time Embedded Components and System in Linux and RTOS by Dr. Sam Siewert and John Pratt
- Liu and Layland's Paper - [ecee.colorado.edu/~ecen5623/ecen/rtpapers/archive/PAPERS\\_READ\\_IN\\_CLASS/Rate-Monotonic-Theory-Liu-and-Layland.pdf](http://ecee.colorado.edu/~ecen5623/ecen/rtpapers/archive/PAPERS_READ_IN_CLASS/Rate-Monotonic-Theory-Liu-and-Layland.pdf)
- Apollo 11 story - [Apollo 11 Lunar Surface Journal: Program Alarms \(nasa.gov\)](http://www.nasa.gov/pdf/1969_01-27/main_image.html), [Margaret Hamilton \(software engineer\) - Wikipedia](https://en.wikipedia.org/wiki/Margaret_Hamilton), [Margaret Hamilton: the Apollo software engineer who saved the moon landing - Vox](https://www.vox.com/2019/7/1/18688882/margaret-hamilton-apollo-11-software-engineer)
- Code reference - [Index of /~ecen5623/ecen/ex/Linux \(colorado.edu\)](http://www.colorado.edu/~ecen5623/ecen/ex/Linux)
- Synthetic Workload Generation – Generating Synthetic Workloads for Real-time Systems by D.L.Kiskis and K.G.Shin, Elsevier, IFAC Proceedings Volumes, Volume 24, Issue 2, May 1991
- Linux Man Pages