**Real Time Embedded System**

**ECEN 5623**

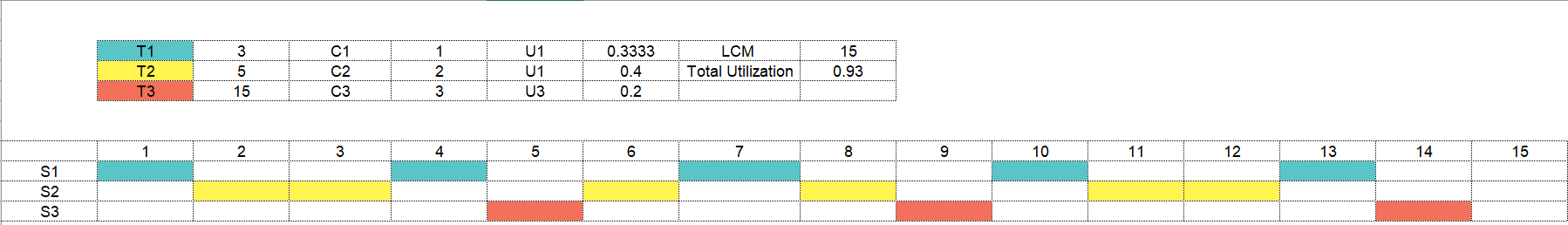
**Exercise 1**

**Ayush Dhoot & Hardik Senjaliya**

**----------------------------------------------------------------------------------------------------**

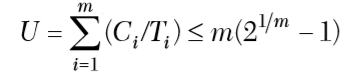
**Question 1)**

The timing diagram for the scheduling of the three services is shown below:

****

The total CPU Utilization for the three services can be calculated using the LCM invariant method U=∑Ci/Ti= ⅓+⅖+3/15= 14/15 = 0.93.

According to Rate Monotonic theory, this scheduling is failing the Least Upper Bound definition.



**U=0.93 > 0.78**

But, as we can see that the schedule is well completed before the deadline (LCM of the periods), it can be concluded that the system is feasible.

We know that Rate Monotonic Least Upper Bound is a ‘sufficient’ test and it always fail a service which is not safe. Hence, we can say that even as the system is feasible but it is not safe.

**Question 2)**

**Summary:**

The journal describes the two program alarms that occurred during the landing of the Apollo 11 lunar module. The author tells us about the difficulty to program back in the day. He mentions that the memory had to be shared between 7 things and they had made sure that the same memory location cannot be used in two different processes. There were erasable and fixed memory available. The code usually used to be in fixed memory but in case of patches they were executed out of the erasable memory. The author mentions that they had developed a real time operating system which were interrupt driven and priority-based tasks. The erasable memory was used in order to store variables and do immediate calculations during the execution of the tasks. They had made a set of 12 memory locations which were called as ‘Core Sets’. If the task would need more memory, there were VACs (Vector Accumulator) allocated to it. In total, there were 7 core sets and 5 VACs available to the module.

Now, if there is scheduling of a job which requires additional memory, the system would check all the 5 VACs to find the one which is ready and allocate it to the task at hand. If there will be no available VACs, the system would show ‘NOVAC’ and go into reset/abort mode causing Alarm 1201 to set. In normal operation, after the VAC has been allocated, the system would then find a core set to allocate it to the task. Same as above, if there is no available core set, the system would go into reset/abort mode and cause Alarm 1202 to be set.

In the Apollo 11, there was a similar case happening because of the jobs scheduled to process radar data which had occurred due to wrong configuration of radar switches. The scheduled jobs took all the core sets and caused Alarm 1202 and because of it, even the VAC area got filled and caused Alarm 1201.

The author tells us that they at MIT Labs had tested each aspect of that alarms. They had developed a system which will restart important tasks like steering but not restart the flawed tasks like obtaining the radar data on the setting of the two Alarms. This was an important aspect as when the Apollo 11 faced this problem, there was no commotion and the mission went forward to completion.

**Root Cause ? Violation of RMA?**

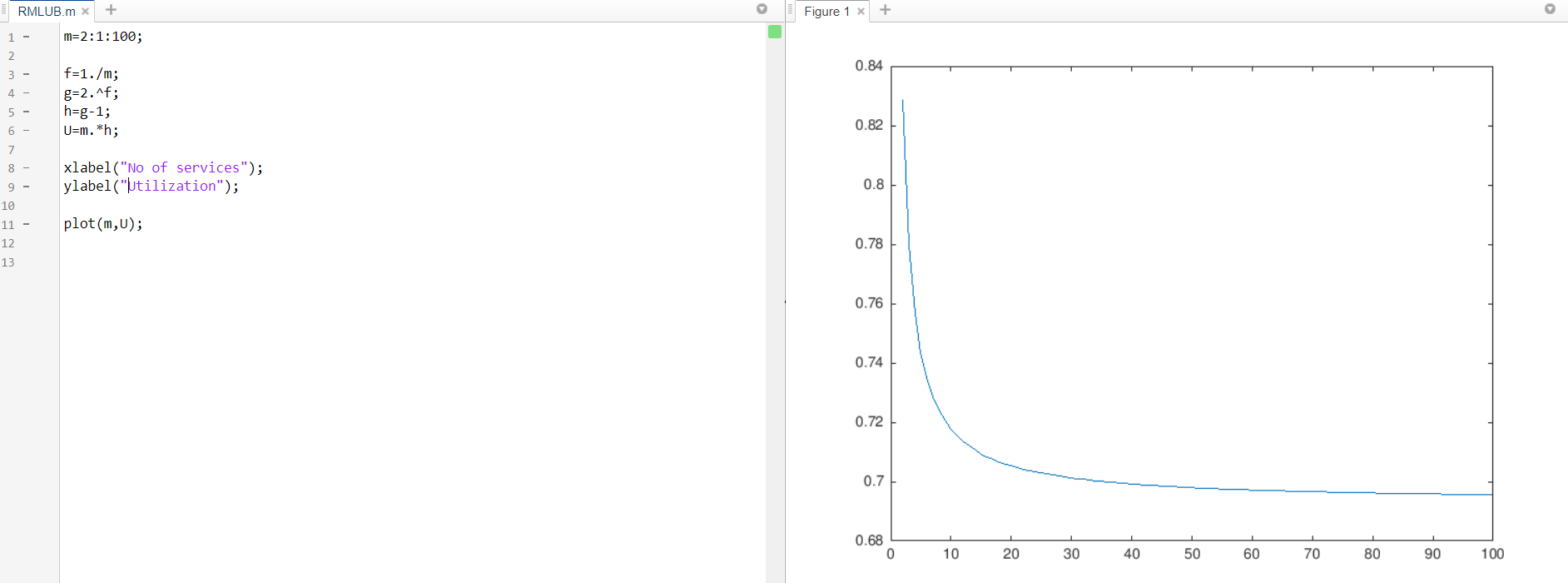
The root cause of the Alarms being set was the scheduling of jobs caused due to the misalignment of radar switches. There was violation of Rate Monotonic as there was no preemption of tasks. Also, RM states that the task with higher frequency should get higher priority but, in this case, after the reboot the radar data task did not restart. This can also be called as a violation of the RM theory.

**Could RM analysis prevent Apollo 11 Alarms?**

In our opinion, RM Analysis would not have prevented the Apollo 11 Alarms. In this case due to the faulty switches, the radar data task was being scheduled again and again even before it was being executed. Thus, if we consider RM theory, the one with the most frequency should be given the highest priority and, in this case, it should have been the radar data task. Hence, only the radar data should have been running while the rest of the tasks like steering the module would be overlooked.

If there could be some scheduling and priority defined for the more important tasks, we can try to use Rate Monotonic analysis.

**RM LUB Plot:**

****

As we can see that on increasing the number of services, we get the CPU utilization approximately as 0.7. This proves the Least Upper Bound theory which states that 30% of the CPU should be kept as margin if we keep on increasing the number of services on a single processor.

There are few assumptions that the Liu and Layland paper makes are that i) all services are serviced on a periodic request whose period remains the same throughout ii) the service requests are independent of each other iii) the completion period is always less than the total time period.

The aspects of the derivation which we have not understood are i) how the least upper bound to the processor utilization factor is same for the tasks whose ratio of any two request period is less than two? II) why it is necessary to keep the restriction of above mentioned ratio to two because later they proved that the restriction can be removed. iii) In the paper the definition of ‘Overflow’ is given but couldn’t understand what exactly it is and how it can play an important factor in fixed priority scheduling algorithm.

**Question 3)**

**Our understanding of RT\_Clock:**

This function runs a thread for two different scheduling policies: 1) SCHED\_OTHER 2)SCHED\_FIFO

The test is set to run for 3 sec.

For SCHED\_OTHER thread scheduling policy it just run delay\_test function. To implement the requested sleep time of 3 secs.

This function runs a do while loop to check the condition for max sleep count and remaining sleep time. It also uses CLOCK\_REALTIME and clock\_gettime() function to find the difference between requested sleep time and actual sleep time by calling the delta\_() function.

For SCHED\_FIFO thread scheduling policy the program creates a thread with attributes (priority = maximum priority of 99 and thread scheduling policy = SCHED\_FIFO).

To use used defined thread scheduling policy the program sets thread attributes to PTHREAD\_EXPLICIT\_SCHED. The thread runs the function delay\_test() and performs the same task as mentioned above for the SCHED\_OTHER scheduling policy.

The clock\_gettime() function is used to get the current instance of the time from a specific instance of the time. This function saves the current time in terms of seconds and nanoseconds in the passed structure as second parameter.

The delta\_t() functions calculates total time taken by the thread to run. It also calculates the difference between actual time taken by the thread to run and requested sleep time to find the error.

When we run the clock with policy SCHED\_OTHER this is result we have obtained:

Requested sleep seconds = 3, nanoseconds = 0

RT clock DT seconds = 3, nanoseconds = 106232

RT clock delay error = 0, nanoseconds = 106232

From the result we can infer that the clock is giving an error of 106232 nanoseconds

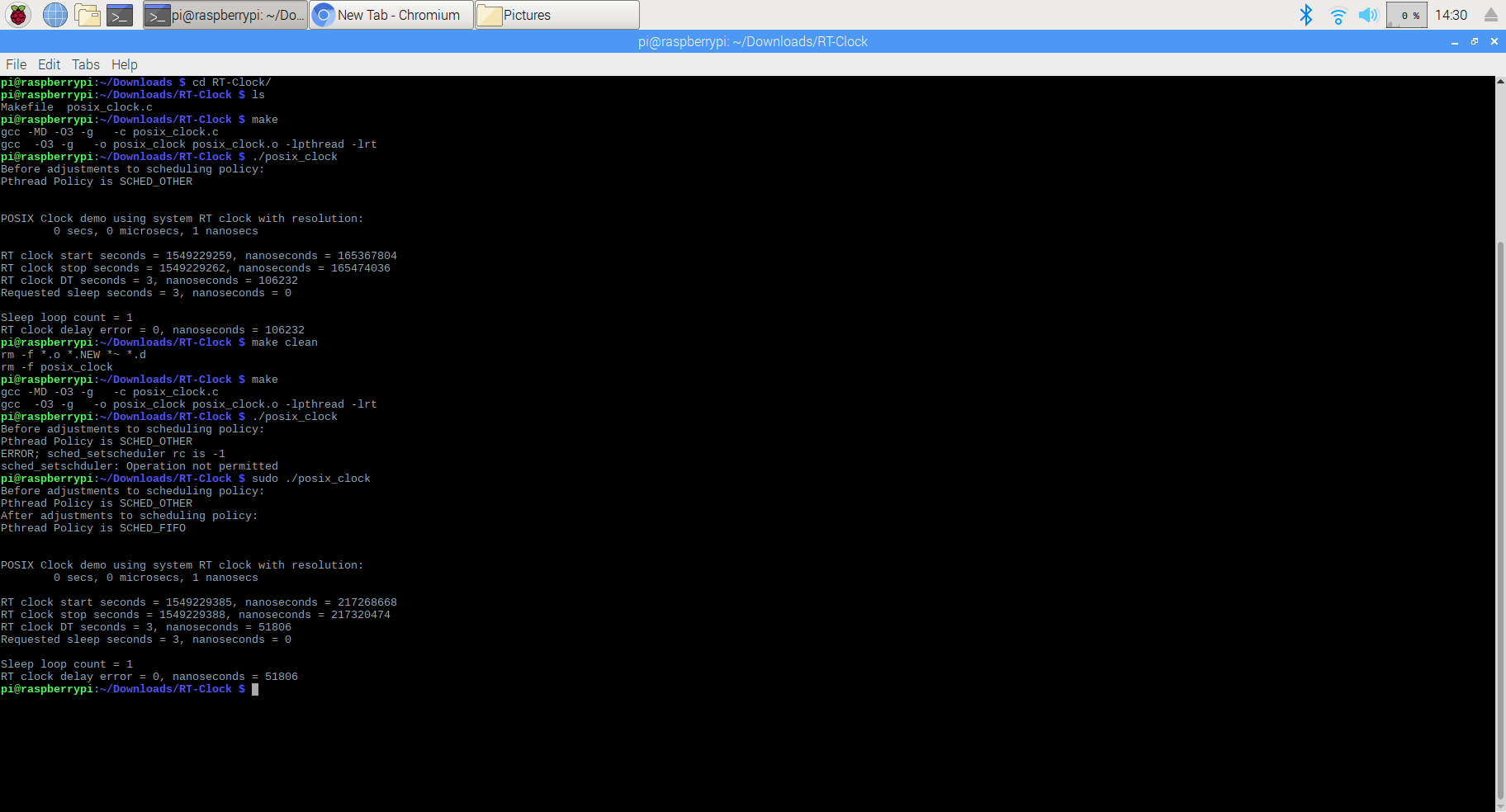
When we run the clock with policy SCHED\_FIFO this is result we have obtained:

Requested sleep seconds = 3, nanoseconds = 0

RT clock DT seconds = 3, nanoseconds = 51806

RT clock delay error = 0, nanoseconds = 51806

From the result we can infer that the clock is giving an error of 51806 nanoseconds



In conclusion we found that we get less delay error when we used SCHED\_FIFO.

**RTOS concepts:**

**Low Interrupt Handler Latency:**

While entering the ISR, the microcontroller or microprocessor usually disables global interrupt. The global interrupt will be enabled again while leaving the ISR. Hence, if the ISR is too long then the global interrupt will be disabled for a long period of time and during this time if any high priority interrupt will not be served as the global interrupt is disabled.

Thus, ISR should have low latency which will increase the responsiveness of the system.

**Low Context Switch time:**

Context switch is the process where the CPU changes it's execution from one process to another process. During context switch, CPU needs to save the current state of the currently running process so that when it comes back from the new process, it can resumes the execution. However, during this process of context switching the CPU is doing no useful work. Hence, low context switch time is desired to increase the utilization of the CPU.

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

Each of these are important to achieve the real time performance at highest possible level. The lowest the value of each of these parameter, highest is the real time performance we can achieve.

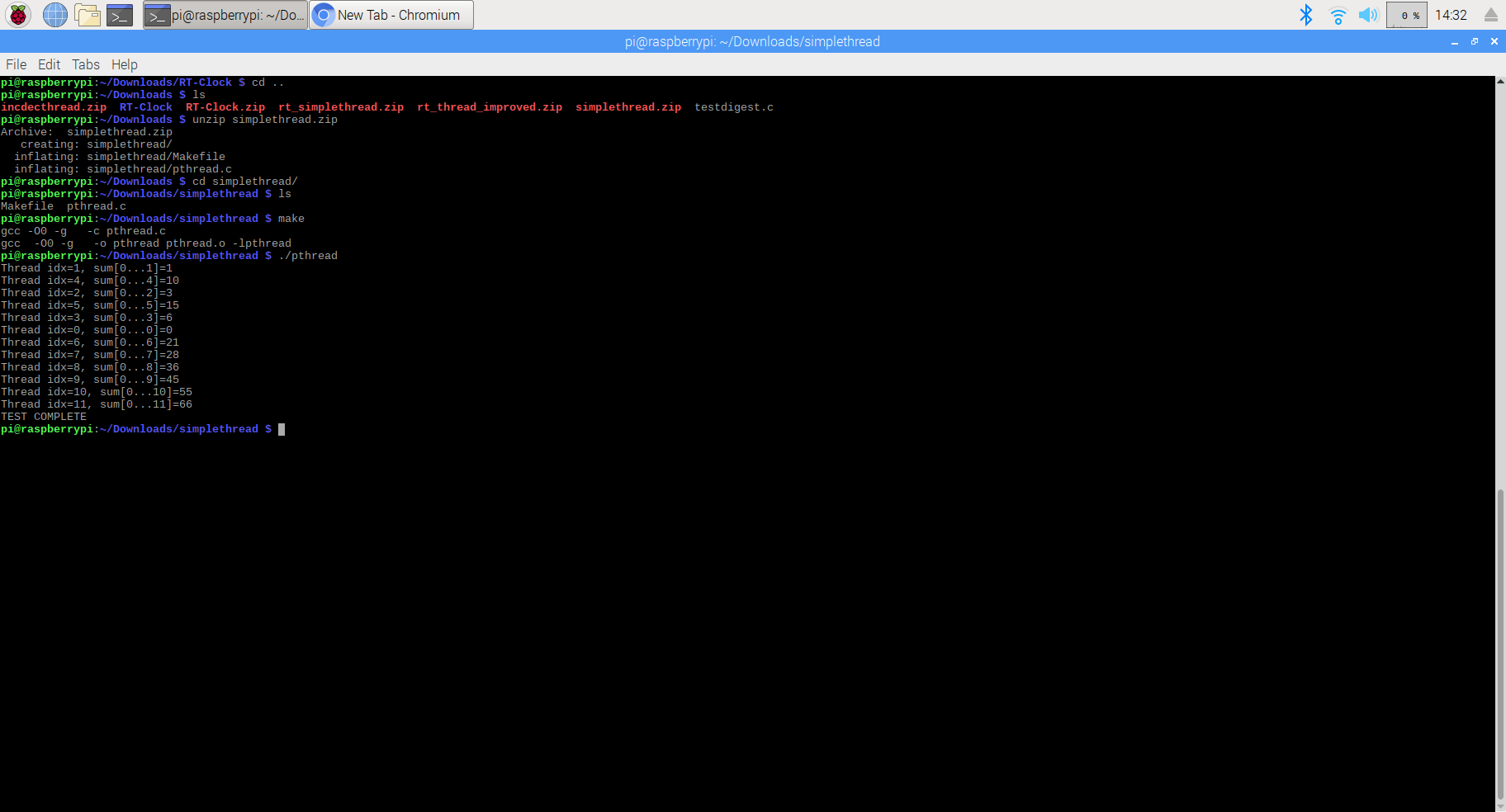
**Accuracy of the RT\_Clock:**

From the results we have achieved after running RT\_Clock program, we came to the conclusion that the RT\_clock is not 100% accurate. The expected time is 3 ms but because of the context switching overhead we are getting an error of +106232 ns for SCHED\_OTHER and +51806 ns for SCHED\_FIFO thread scheduling policy and hence it can be used in SOFT REAL time systems where occasionally missed deadline is acceptable. However, RT\_clock should not be used in HARD REAL time system as the error we are getting can be the cause of missed deadlines which can result in system failure.

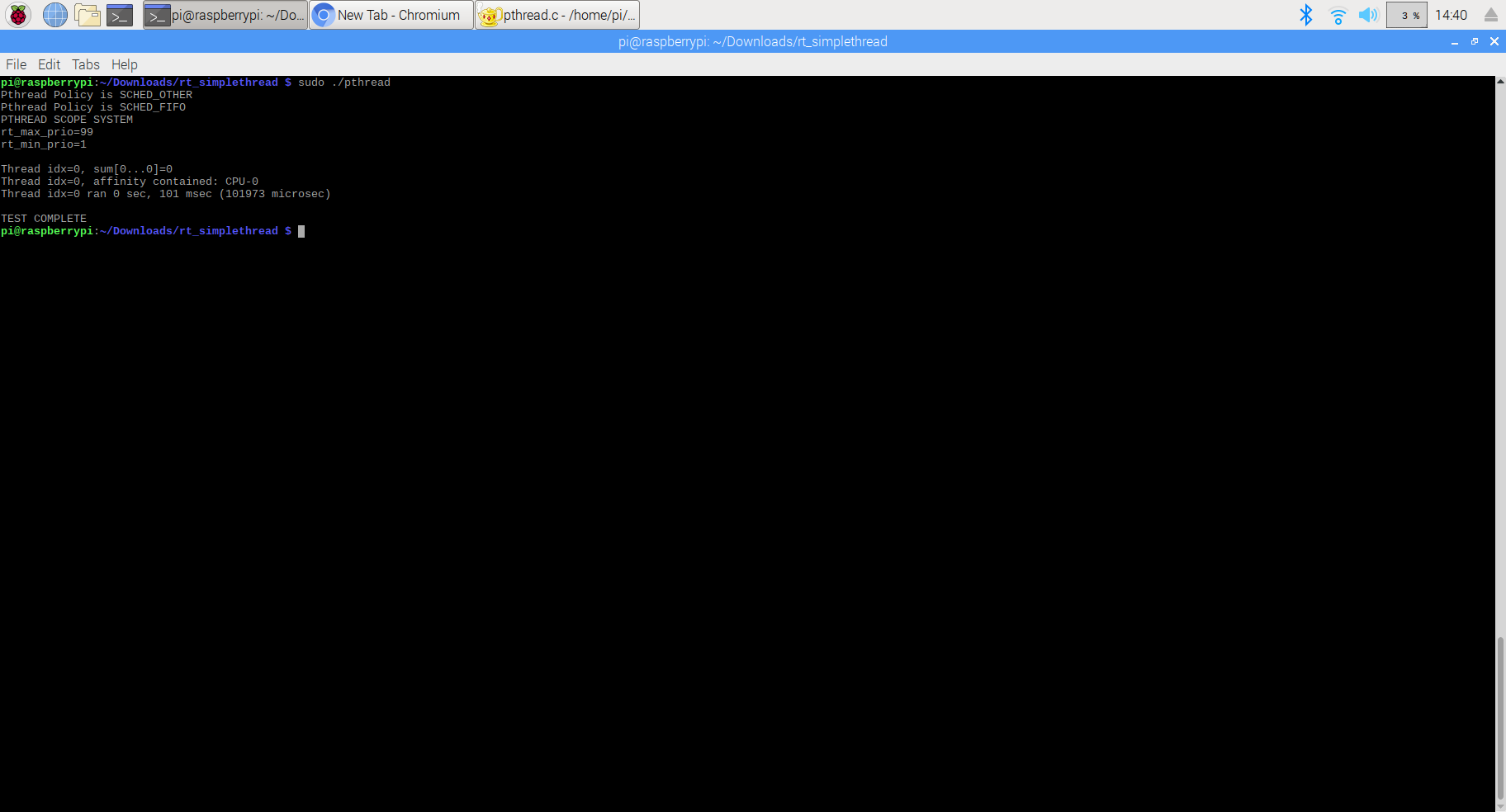
**Question 4)**

**Description and Output of Codes:**

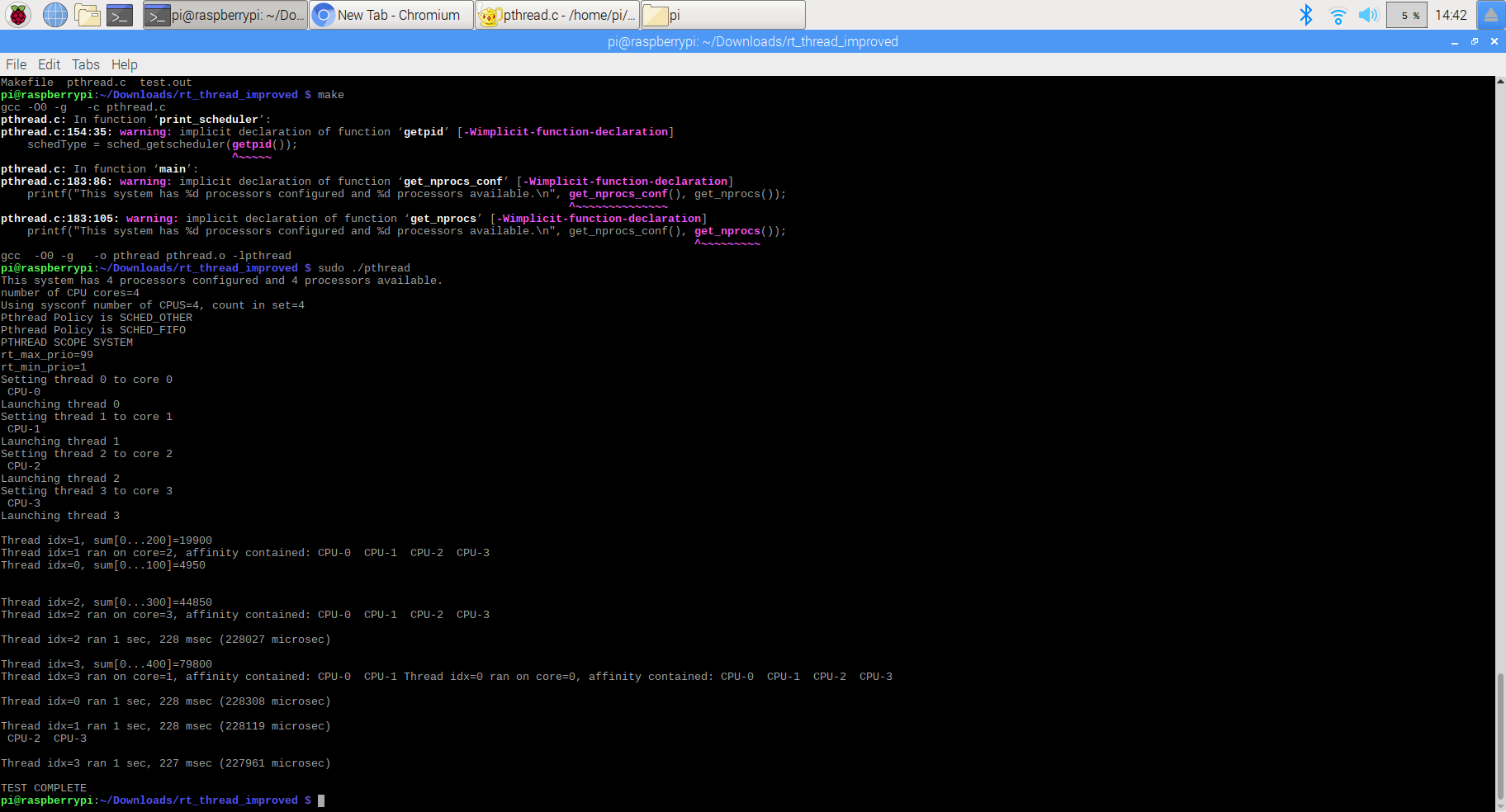
**I) simplethread:** this program creates 12 threads and each thread execution the function named 'counterThread'. This function adds numbers starting from 1 to (id of the thread). The execution of the threads is depends on the OS scheduler and is in random order.

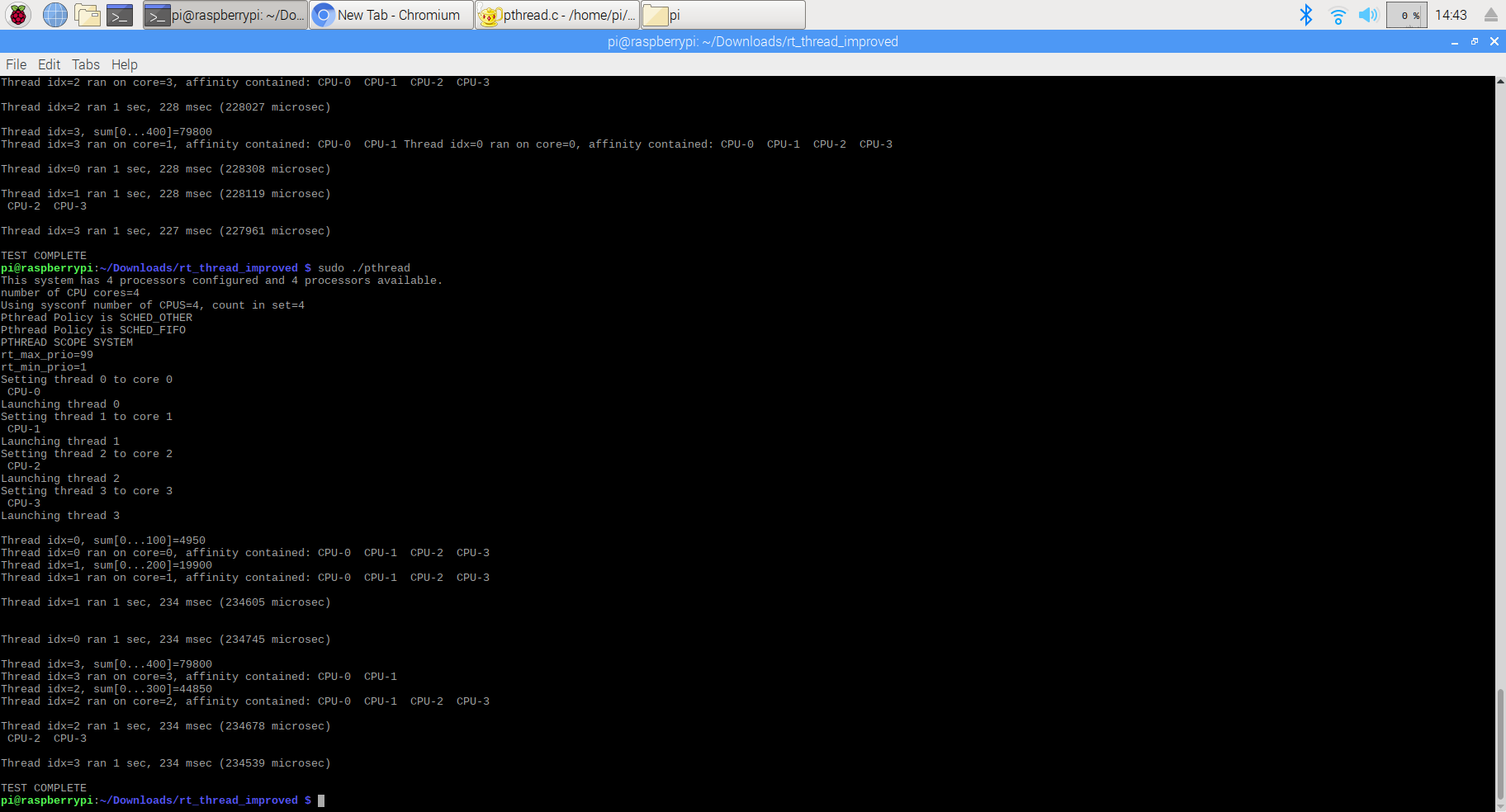


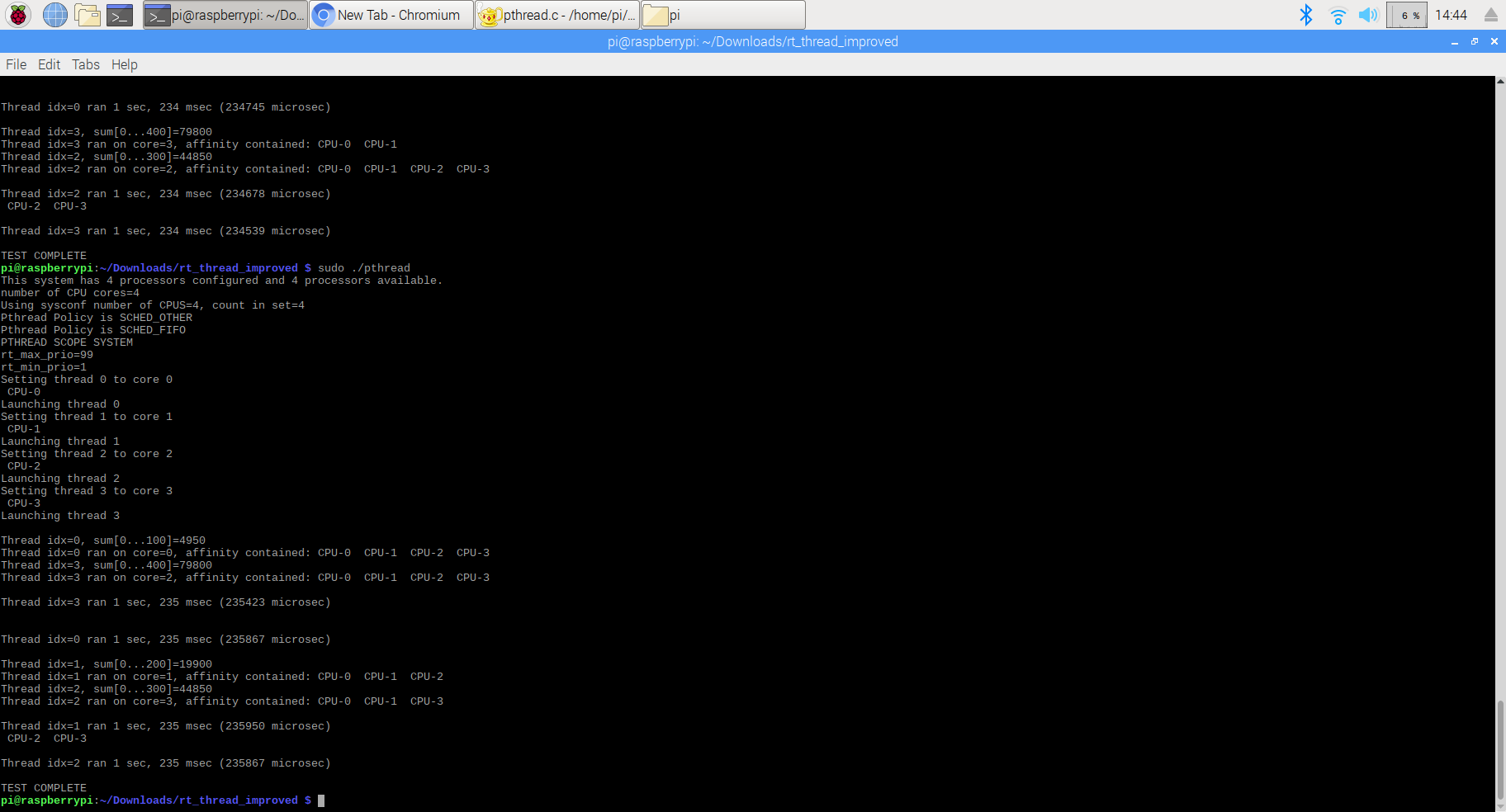
**II)rt\_simplethread:** this program runs 1 thread on a user specified CPU core(0) using policy SCHED\_FIFO. The thread function (counterThread) calculates sum from 1 to (thread\_id) and also calculates fibonacci series upto 47 terms but the loop runs for 140000 times. The threads runs for about 3 ms. The delta\_t() function calculates time taken by the thread to run.



**III)rt\_thread\_imroved:** this program creates 4 threads and runs each threads on a specified CPU core using SCHED\_FIFO policy. The program uses the 'Configured CPU cores' out of available CPU cores. The program runs each thread on a user specified CPU core by setting the CPU affinity. The program also sets the priority for each thread, setting thread 0 to highest priority and thread 3 having the lowest priority. As shown in the pictures below, the threads with the lower priorities are preempted by the higher priority threads.The delta\_t() function calculates time taken by the thread to run.







**Tasking vs Threading:**

A thread is defined as a program segment which executes within a process. It can be seen as a small code snippet which has a specific function which serves the overall purpose. A thread can have its own stack, kernel resources and memory is shared within the process. It is more costly and the overheads associated is also more than that of a task. A thread is used for longer running processes as it has more control over the function.

A task can be defined as a portion of a program which carries out a very specific purpose. A task is very similar to a thread but it does not have its own OS thread. This can be advantageous because it then has much lower overheads and costs. A thread is therefore much more simpler to use and more efficient than a thread.

**Semaphores - Wait and Sync:**

Semaphores is used for managing the resources and properly sharing it. A task needs to first procure a semaphore in order to gain control over a shared resource. It should drop the semaphore once it is done with its task.

sem\_wait() is used for decrementing the value of the semaphore pointed to by sem. Its function is to lock the thread from executing.

sem\_post() is used to increment the value of the semaphore and by doing so, unblock the thread.

These two functions are important as they are used to control the scheduling of the threads as per the requirements.

**Synthetic Workload generation and its Analysis and Adjustment on Test system:**

Synthetic workload is a program or a set of programs which acts as the test task assigned to the system. We can change the parameters of the function to generate different resources as per the requirements.

In our code, Synthetic workload is generated by generating the Fibonacci series for a 47 terms but the outer loop in the macro to generate the workload is adjusted by doing analysis/trial and error to generate the load for a specified/required period of time.

**Implementation of LCM invariant schedule in LINUX:**

We could implement LCM invariant scheduler of VxWorks on LINUX system in a way mentioned below.

* Create two threads and a main threads and assign a priority of 99 to main thread which will act as sequencer. Other two threads, let’s say, fib10 and fib20, will have the priority of 98 and 97 respectively.
* To generate a load of 10 ms for the thread fib10 and a load of 20ms for thread fib20, we can do any simple mathematical operation in the thread function. For instance, as implemented in the VxWorks system by generating Fibonacci series, we can do the same by adjusting number of terms to generate for keeping the thread busy for 10ms and 20 ms.

Or we can just run a loop to generate a load of 10 and 20 ms. We have to trial and error to adjust the Fibonacci terms to generate or a loop to run.

* If scheduling the thread Fib10 and Fib20 is to be done at every 20ms and 50ms respectively then we could achieve this by using the sleep() function.

VxWorks to Linux implementation functions mapping:

1. **Functions related to thread/task**

* **thread creation:**

**VxWorks:** *taskSpawn(*) function is used for creating a new child thread from the main thread. Different parameters passed to this function are, name of the thread, priority of the thread, function to be executed by the thread, stack size to be used by the thread, and 10 arguments to the function. This function returns task id for successful operation or error if memory is not sufficient to create a new task.

**Linux**: *pthread\_create()* function is used for creating a new child thread from the main thread. Different parameters passed to this function are, thread id, function to be executed by the thread, attributes of the thread and arguments to the thread function.

* **Task delaying**:

**VxWorks:** *taskDelay()* function is used for stopping the task from executing for certain period of time.

**Linux:** *usleep()* function is used for stopping the task from executing for a certain period of time.

1. **Functions related to semaphore**

* **Semaphore creation:**

**VxWorks:** *semBCreate()* is used for creating a binary semaphore. Parameters passed to this function are, initial state(1 or 0) and options(priority or FIFO).

**Linux**: *sem\_init()* is used for creating a unnamed semaphore. Parameters passed to this function are, address at the semaphore to be created, whether to be shared between processes or between threads of a process and initial value.

* **Semaphore release:**

**VxWorks:** *semGive()* function is used for releasing the semaphore and semaphore id is passed to the function.

**Linux:** *sem\_post()* function is used for releasing the semaphore and the address of the semaphore is passed to the function.

* **Semaphore holding:**

**VxWorks:** *semTake()* function is used for holding the semaphore. Parameters passed to this function are, id of the semaphore and wait time(wait indefinitely, do not wait or wait for a specified period of time).

**Linux:** *sem\_wait()* function is used for holding the semaphore and address of the semaphore is passed to this function.

1. **Other functions used by Linux**

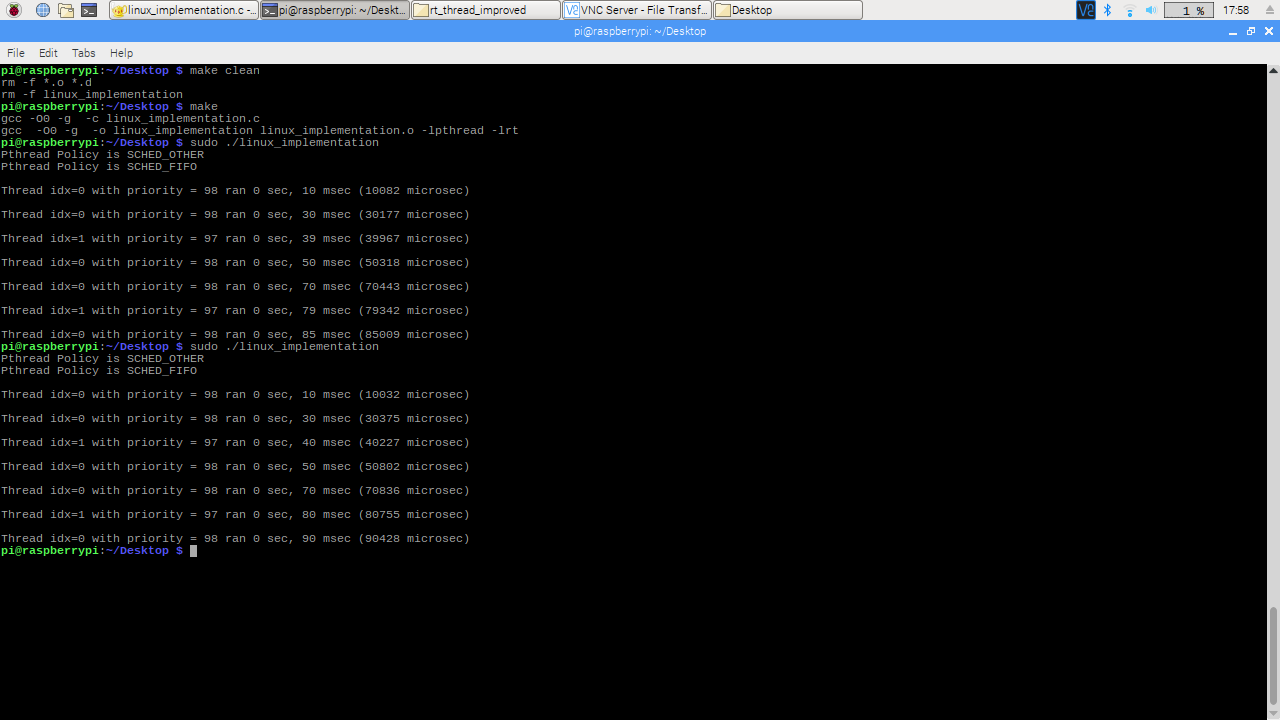
Linux uses pthread\_exit() (to terminate the thread) and sem\_destroy() (to destroy the semaphore)

**Were we able to achieve predictable reliable results or not?**

**Describe whether your able to achieve predictable reliable results in terms of the C (CPU time) values alone and how you would sequence execution.**

Yes, we were able to achieve the predictable and reliable results as shown in the picture below.

We have used usleep() function to implement the periodicity of both threads and binary semaphores are used to sequence the execution. We have used Fibonacci series generator to generate the synthetic load.



*Note: to implement this linux version of the code we have taken reference from the rt\_thread\_improved program.*