**Real-time task models in Linux**

**Assignment 1: Report**

*Submitted for the Subject:*

**Real Time Embedded Systems CSE522**

*By:*

**Sarvesh Patil (1213353386) Vishwakumar Doshi (1213322381)**

*Under the guidance of:* **Prof. Yann-Hang Lee**



**Arizona State University Tempe Campus, AZ**

**February, 2018**

**ACKNOWLEDGEMENT**

We owe a great thanks to many people who helped and supported us during the completion of this project.

Our deepest thanks go to **Prof. Yann-Hang Lee**, Professor for the subject Real Time Embedded System who acted as our mentor and guided us throughout this project. We thank him for giving ideas and briefs for the work that had to be done.

A special thanks to **Shiksha Patel**, Teaching Assistant, Embedded System Programming, for constant help and guidance in the labs as well as discussion group.

We also thank all our friends for helping us out during the downs. We respect everyone for giving their time when we were in need.

Thank You.

Priority inversion is a peculiar scheduling scenario where a high priority tasks is denied execution opportunity because of synchronization lock acquisition by a thread running at a low priority.

*τ1(H)*

***Blocked***

*τ2(M)*

*τ3(L)*

*Time*

*τ1 :{…P(S1)…V(S1)…}*

*τ3 :{…P(S1)…V(S1)…}*

***S1 unlocked***

***Attempt to lock S1 (blocked)***

***S1 locked***

***S1 unlocked***

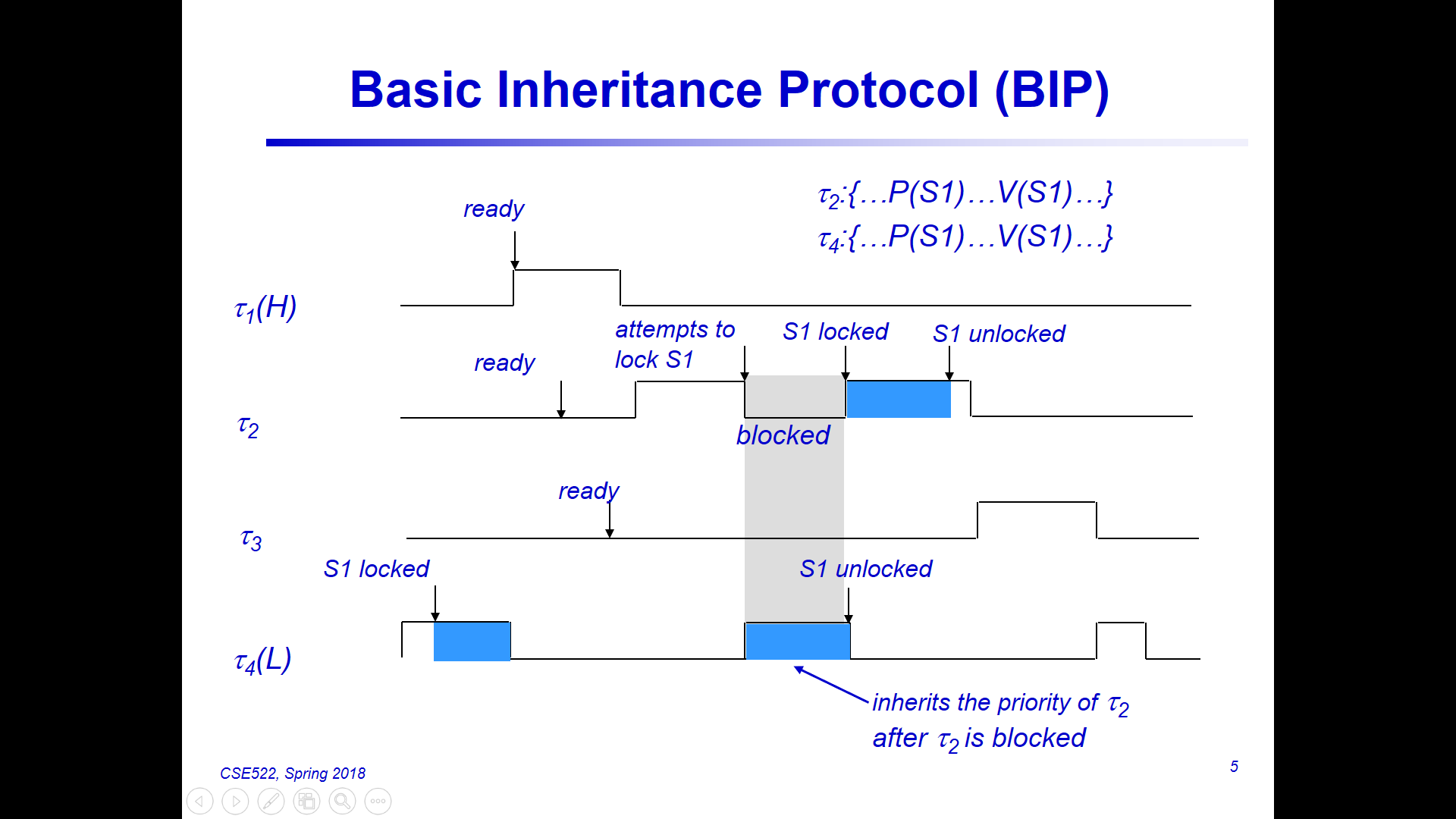
***S1 locked***

A low priority task when acquires a mutex or a similar synchronization lock to carry out execution in its critical section, it is capable of blocking even a high priority task waiting to execute its critical section by acquiring the same mutex.

Imagine a case when a thread with priority higher than the low priority thread but lower than the high priority thread becomes ready for execution. Here, if this middle thread is not trying to acquire the same mutex, it can easily pre empting the low priority thread and further delaying the execution of high priority thread by execution time of the middle thread.

This situation is termed as ‘Priority Inversion’ and it can delay execution of a high priority thread indefinitely.

**Priority Inheritance:**



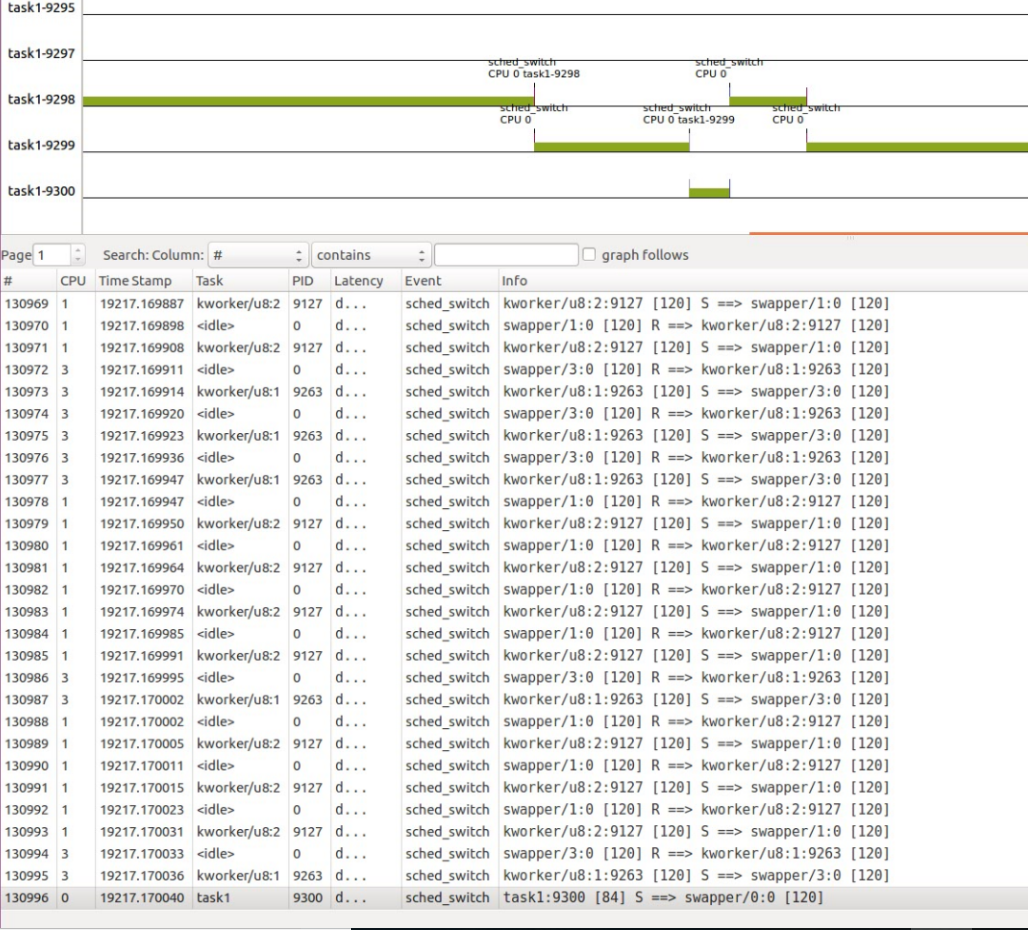
One of the methods to overcome the problem of priority inversion is priority inheritance. In this method, the running low priority thread, during execution of its critical section, inherits the highest of the priorities of threads waiting for that particular mutex.

Although this doesn’t solve the first problem of high priority thread being blocked by low priority thread, it does prevent any other thread to preempt the low priority thread during execution of its critical section and avoids further delaying of execution of waiting high priority thread.

Assignment:

Here, we read an input text file and create periodic and aperiodic threads with computations as specified in the file. To show the difference in operation with priority inversion and priority inheritance, we run the program in 2 modes in both, the host Ubuntu Linux and Galileo Gen 2 board. In one mode, normal mutex functionality is used. In another mode, priority inheritance capable mutex is used.

Case 1.1 : Normal mutex on Ubuntu



ID 9298: Lowest priority thread

ID 9299: Highest priority thread

ID 9300: Medium priority thread

As we can see from kernelshark tool snippet, initially, the low priority thread is executing in its critical section by acquiring a “Normal” mutex.   
  
  
High priority arrives trying to acquire the same mutex to access resource being used by Low priority task. Although it is denied of service & gets blocked as low priority task is holding the subsequent resource with mutex lock, other threads can function normally.

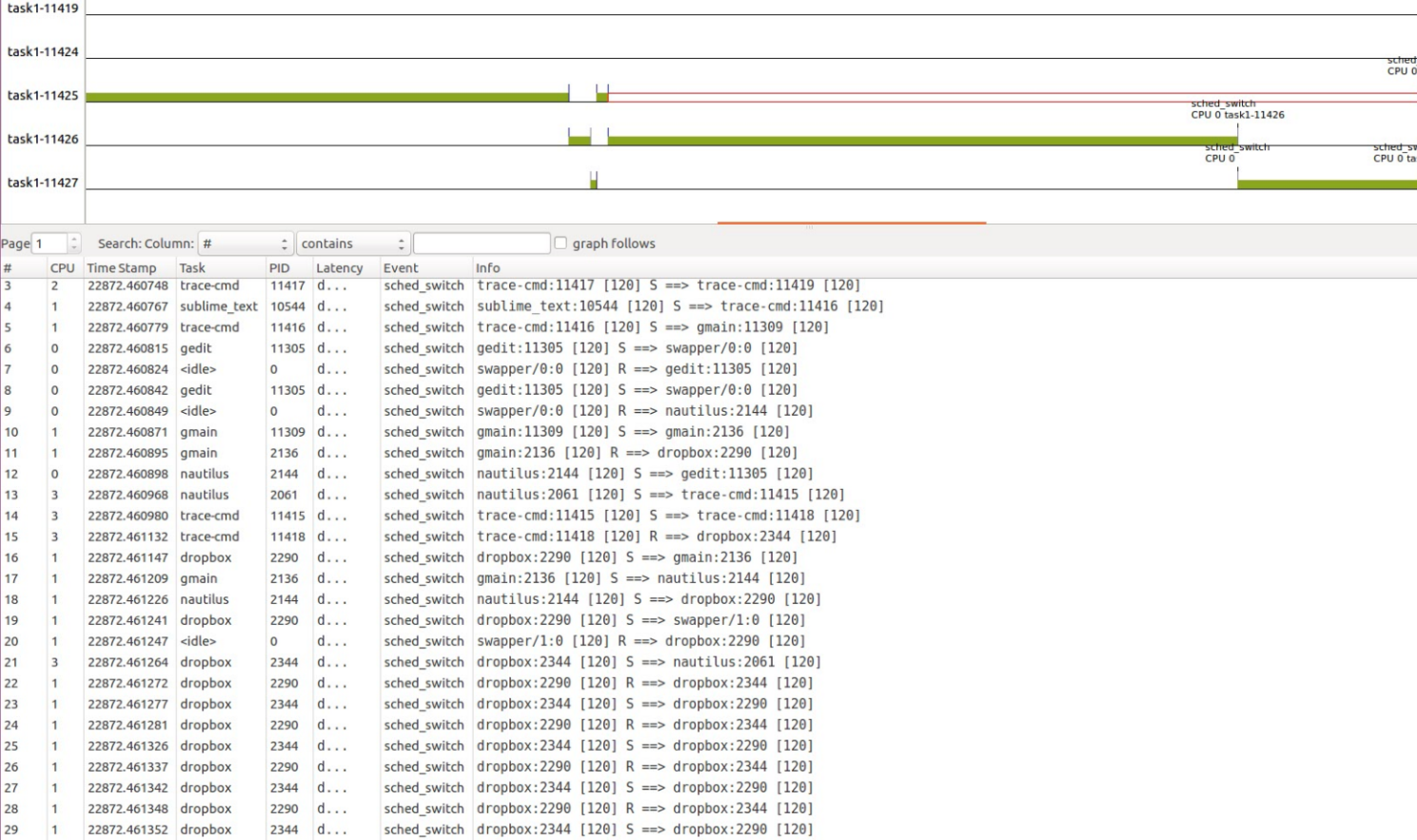
Therefore, we can see that as the middle priority thread arrives, it can definitely preempt the low priority thread as it is not attempting to acquire any mutex which low priority thread might be holding. The low priority thread now waits for middle thread to finish execution before resuming and it worsens the situation for high priority thread as it now has to wait for both low and middle priority thread to finish execution even though it has the highest priority among them.

In the figure shown, the low priority thread is running in critical section. Then as high priority thread arrives it preempts the low priority thread and runs till it tries to acquire mutex and gets blocked. At that time middle priority thread which was waiting, gets chance to execute fully. It then allows low priority to finish executing and at last in the end high priority gets execution time.

Case 1.2: Normal mutex in Galileo Gen 2:

Similar results were found on Galileo environment. For detailed ftrace results, please refer to pi\_disabled.txt file in zip file. Each column represents different thread.

Case 2.1: Priority-Inheritance enabled mutex on Ubuntu:



ID 11425: Low priority thread

ID 11426: High priority thread

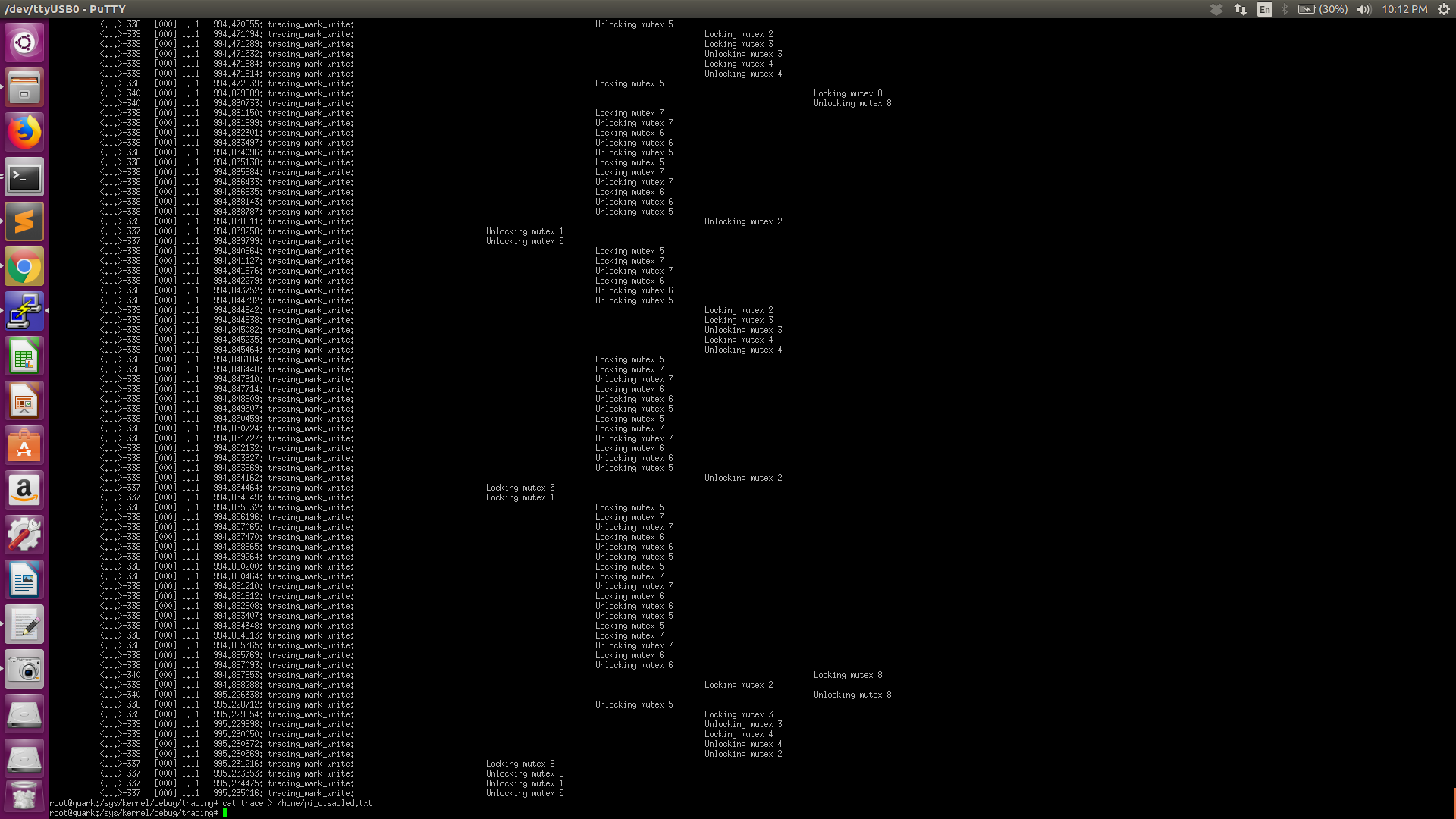
ID 11427: Middle priority thread

As we can see from the figure, low priority thread is running in critical section by acquiring a PI enabled mutex. High priority thread arrives by preempting low priority thread. As soon as it tries to acquire the mutex, it gets blocked and control is momentarily given to middle thread which is preempted by the then low priority thread because it now has the priority of the high priority thread.

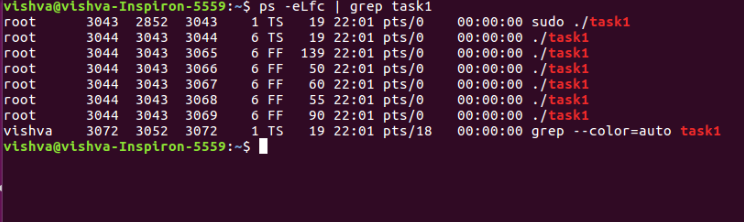
Now, low priority thread finishes execution first, then transfers control to high priority thread when it releases the mutex; and only when high priority thread finishes execution, only then the middle and other threads will get a chance to execute.

Case 2.2: Priority Inheritance enabled mutex on Galileo 2:

Please refer to pi\_enabled.txt file in zip file to get detailed results. A snapshot of ftrace generated log file is shown below. Here also, each column represents different thread.



Process status for a specific input file:



Here we can see threads have FIFO scheduling policy and priorities for them are shown. Main thread doesn’t have FIFO policy. Highest priority is given to mouse click reading thread.