Real-time task models in Linux

Assignment 1: Report

Submitted for the Subject:

Real Time Embedded Systems

CSE522

By:

Nagarjun chinnari (1213287788)

Nisarg Trivedi (1213314867)

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.

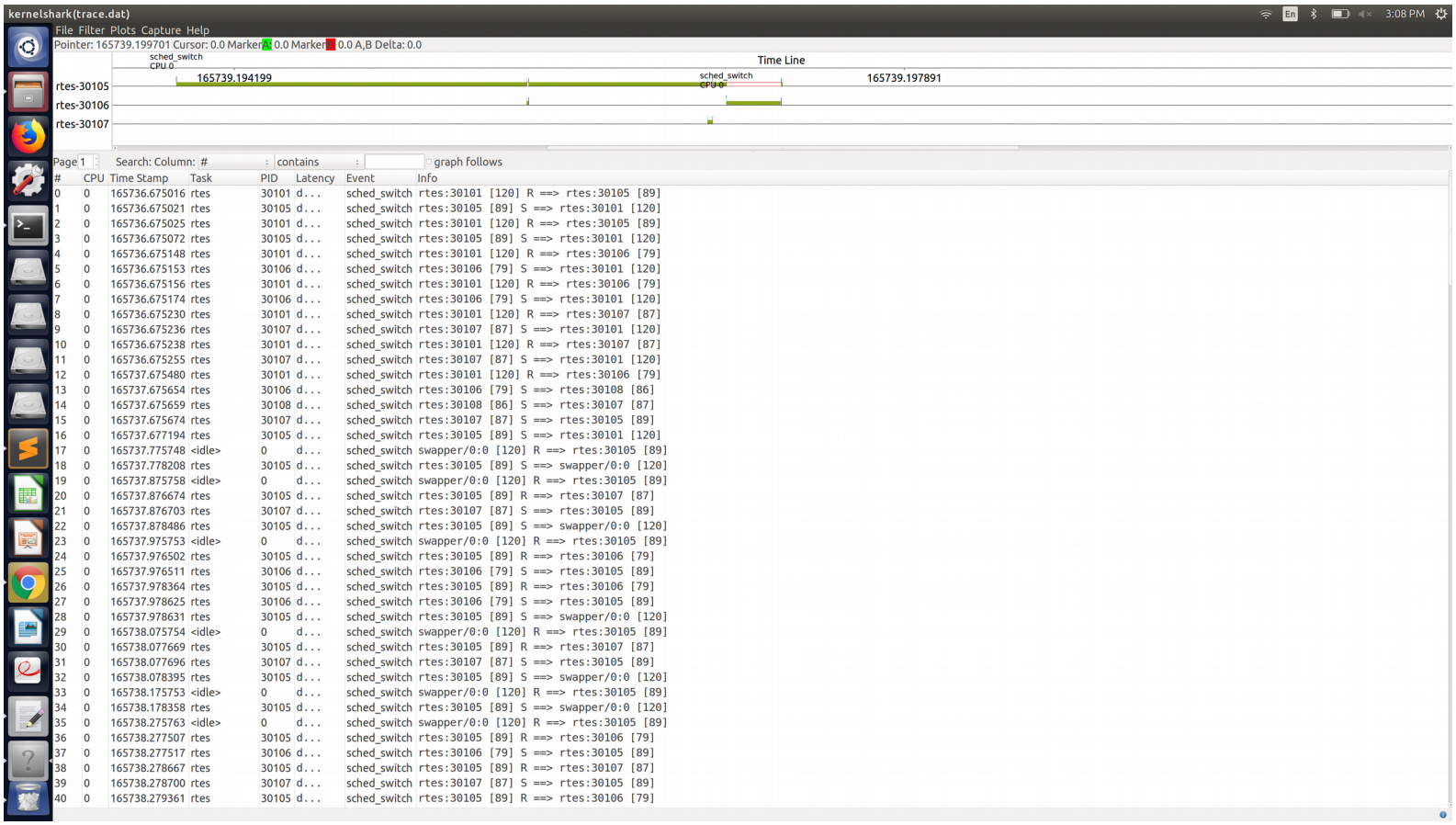
Thank You.

Priority inversion:

It 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.

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 prempting 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. The same can be seen below

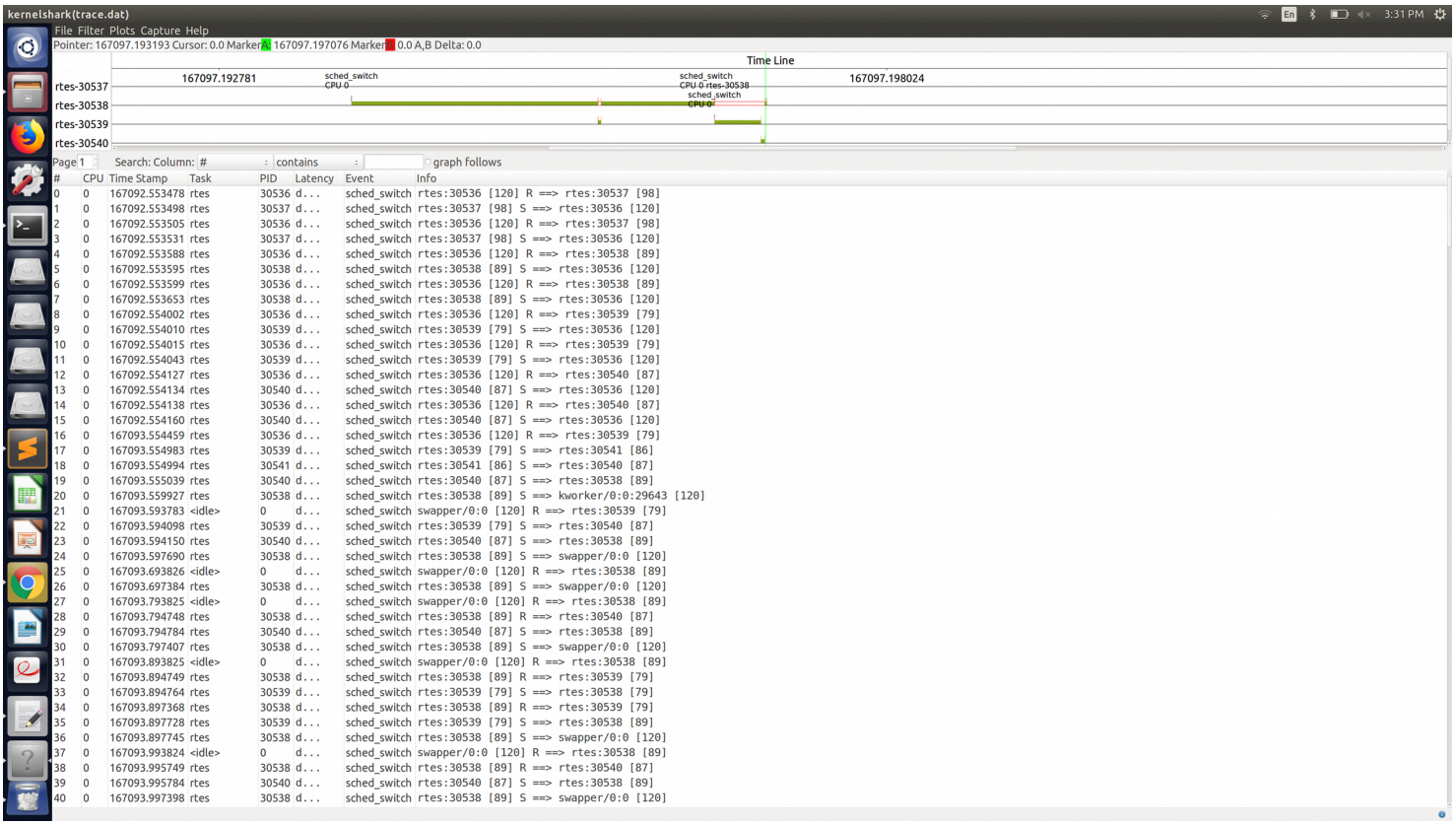


#	CPU	Time Stamp	Task	PID	Latency	Event	Info
0	0	165736.675016	rtes	30101	d...	sched_switch	rtes:30101 [120] R ==> rtes:30105 [89]
1	0	165736.675021	rtes	30105	d...	sched_switch	rtes:30105 [89] S ==> rtes:30101 [120]
2	0	165736.675025	rtes	30101	d...	sched_switch	rtes:30101 [120] R ==> rtes:30105 [89]
3	0	165736.675072	rtes	30105	d...	sched_switch	rtes:30105 [89] S ==> rtes:30101 [120]
4	0	165736.675148	rtes	30101	d...	sched_switch	rtes:30101 [120] R ==> rtes:30106 [79]
5	0	165736.675153	rtes	30106	d...	sched_switch	rtes:30106 [79] S ==> rtes:30101 [120]
6	0	165736.675156	rtes	30101	d...	sched_switch	rtes:30101 [120] R ==> rtes:30106 [79]
7	0	165736.675174	rtes	30106	d...	sched_switch	rtes:30106 [79] S ==> rtes:30101 [120]
8	0	165736.675230	rtes	30101	d...	sched_switch	rtes:30101 [120] R ==> rtes:30107 [87]
9	0	165736.675236	rtes	30107	d...	sched_switch	rtes:30107 [87] S ==> rtes:30101 [120]
10	0	165736.675238	rtes	30101	d...	sched_switch	rtes:30101 [120] R ==> rtes:30107 [87]
11	0	165736.675255	rtes	30107	d...	sched_switch	rtes:30107 [87] S ==> rtes:30101 [120]
12	0	165737.675480	rtes	30101	d...	sched_switch	rtes:30101 [120] R ==> rtes:30106 [79]
13	0	165737.675654	rtes	30106	d...	sched_switch	rtes:30106 [79] S ==> rtes:30108 [86]
14	0	165737.675659	rtes	30108	d...	sched_switch	rtes:30108 [86] S ==> rtes:30107 [87]
15	0	165737.675674	rtes	30107	d...	sched_switch	rtes:30107 [87] S ==> rtes:30105 [89]
16	0	165737.677194	rtes	30105	d...	sched_switch	rtes:30105 [89] S ==> rtes:30101 [120]
17	0	165737.775748	<idle>	0	d...	sched_switch	swapper/0:0 [120] R ==> rtes:30105 [89]
18	0	165737.778208	rtes	30105	d...	sched_switch	rtes:30105 [89] S ==> swapper/0:0 [120]
19	0	165737.875758	<idle>	0	d...	sched_switch	swapper/0:0 [120] R ==> rtes:30105 [89]
20	0	165737.876674	rtes	30105	d...	sched_switch	rtes:30105 [89] R ==> rtes:30107 [87]
21	0	165737.876703	rtes	30107	d...	sched_switch	rtes:30107 [87] S ==> rtes:30105 [89]
22	0	165737.878486	rtes	30105	d...	sched_switch	rtes:30105 [89] S ==> swapper/0:0 [120]
23	0	165737.975753	<idle>	0	d...	sched_switch	swapper/0:0 [120] R ==> rtes:30105 [89]
24	0	165737.976502	rtes	30105	d...	sched_switch	rtes:30105 [89] R ==> rtes:30106 [79]
25	0	165737.976511	rtes	30106	d...	sched_switch	rtes:30106 [79] S ==> rtes:30105 [89]
26	0	165737.978364	rtes	30105	d...	sched_switch	rtes:30105 [89] R ==> rtes:30106 [79]
27	0	165737.978625	rtes	30106	d...	sched_switch	rtes:30106 [79] S ==> rtes:30105 [89]
28	0	165737.978631	rtes	30105	d...	sched_switch	rtes:30105 [89] S ==> swapper/0:0 [120]
29	0	165738.075754	<idle>	0	d...	sched_switch	swapper/0:0 [120] R ==> rtes:30105 [89]
30	0	165738.077669	rtes	30105	d...	sched_switch	rtes:30105 [89] R ==> rtes:30107 [87]
31	0	165738.077696	rtes	30107	d...	sched_switch	rtes:30107 [87] S ==> rtes:30105 [89]
32	0	165738.078395	rtes	30105	d...	sched_switch	rtes:30105 [89] S ==> swapper/0:0 [120]
33	0	165738.175753	<idle>	0	d...	sched_switch	swapper/0:0 [120] R ==> rtes:30105 [89]
34	0	165738.178358	rtes	30105	d...	sched_switch	rtes:30105 [89] S ==> swapper/0:0 [120]
35	0	165738.275763	<idle>	0	d...	sched_switch	swapper/0:0 [120] R ==> rtes:30105 [89]
36	0	165738.277507	rtes	30105	d...	sched_switch	rtes:30105 [89] R ==> rtes:30106 [79]
37	0	165738.277517	rtes	30106	d...	sched_switch	rtes:30106 [79] S ==> rtes:30105 [89]
38	0	165738.278667	rtes	30105	d...	sched_switch	rtes:30105 [89] R ==> rtes:30107 [87]
39	0	165738.278700	rtes	30107	d...	sched_switch	rtes:30107 [87] S ==> rtes:30105 [89]
40	0	165738.279361	rtes	30105	d...	sched_switch	rtes:30105 [89] R ==> rtes:30106 [79]

ID 30105: Lowest priority thread
ID 30106: Highest priority thread
ID 30107: Medium priority thread

the above shown image is a kernelshark screen shot from host and shows the priority inversion where the high priority thread is locked out because low priority thread executes and then medium priority executes and then high priority executes.the same can also be inferred from pi_disabled.txt which was collected from galileo board using Ftracecmd.

Priorrity Inheritance:

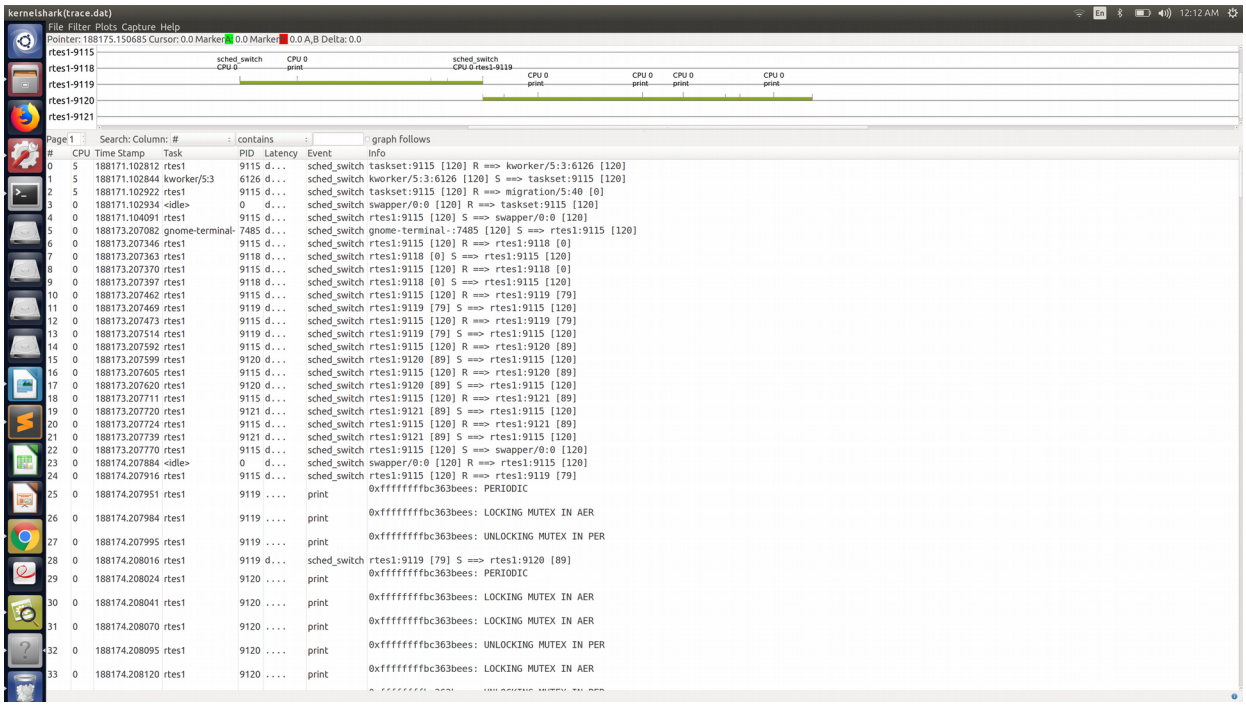


ID 30538 Low priority thread
ID 30539: High priority thread
ID 30540: Middle priority thread

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. The same can be inferred from text file pi_enabled.txt which was collected from galileo board using Ftracecmd.

FIFO SCHEDULING:



In the above image thread with thread id 9119 has a high priority and thread with thread id 9120 has low priority as they are scheduled as FIFO the higher priority thread is runned first and then lower priority runs.