Q-1

The three points are discussed as follows :-

In this paper in the ininitial phase the author has described about priority inheritance. It is caused by mainly 3 factors that are semaphores , monitors or the Ada Rendezvous.[1]

Point 1 :- Priority Inversion :- It is a method in which a higher priority job lets say J0 is blocked by a lower priority job J1 then such a situation is called priority inversion.[1]

Drawbacks of Priority Inversion :- When the problem of Priority Inversion arrives, then we are able to see the deadlines missing. Now if we miss the deadlines in the real time applications then we might also loose the system and it will be reported as a system failure since the objective is not met. Thus in order to avoid such situations, the authors came up with mainly 2 methods.

Two methods are :- 1) Priority Inheritance Protocol :-

Schedulibility of the system :- It is described as the level of resource utilisation which is attainable before a deadline is missed.[1] **Schedulibility and Blocking are inversely proportional to each other.**

Now, if the lower priority job blocks a higher priority job while it is executing its critical section , then the lower priority job will take place the priority of the job which is blocked by it and then it will continue to execute its own critical section with the higher priority task's priority.

After the completion of its critical section it will resume its own priority and at this time , the higher priority task can take its place.

1) **Direct Blocking Method :-** Lets assume a job J is assigned the processor. Before it enters the critical section , it should obtain a lock on semaphore which is a necessary requirement in order to enter the critical section to avoid data loss. This procedure is used to guard the critical section. There are also some instances when the job is already locked. In such cases the lock which it trying to get will be denied. This method of direct blocking is the method when such a condition of locking a locked semaphore happens. By using this method whenever we want to check the consistency we can do that.

2) **Push Through Blocking method :-** Lets assume there are three jobs J0,J1,j2 in the decreasing order of their priority. (J0>J1>J2)
   The job J1(MP) can be blocked job J2(LP) which gets inherited by J0(HP) this is called **push through blocking method.** It avoids using a J0(HP) being indirectly pre-empted by the execution of the job J1(MP).

   Sometimes there are certain condition where deadlock happens which these above methods can not prevent and also conditions of chain blocking.

Point 2 - Priority Ceiling Protocol

As discussed above about deadlocks and chain blocking, we need to solve those problems with some methods. One of the method which is described in this paper is the method of Priority ceiling protocol.

For example, when a job J gets a higher priority and then it preempts the priorities of another job and executes the critical section before that another job , then the priority at which the Job J will execute the critical will be the highest among all the jobs which it pre empted. So if Job J pre empted jobs J1, J2 then the priority of the job J will be higher then J1 & J2thus it will result into faster execution. [1]

Point 3 - Schedulibility analysis :-

In this section of the paper the author has mentioned about specific conditions under which a set of periodic tasks using the priority ceiling protocol can be scheduled by rate monotonic algorithm.

There are 3 conditions to assume in the beginning. First of all we need to consider that **all the tasks are periodic.** Second, we need to assume that **every job in those tasks has their determined execution times for critical and non critical sections**. In the end we assume that these tasks **which are periodic are assigned their respective priorities according to rate monotonic algorithm.[1]**

These are the 3 main points of paper.

According to Linus, we should not use Priority Inheritance and most priority inheritance schemes have shown a tendency to complicate and slow down the locking code.
In his paper, he has encouraged views on how lockless designs are useful and how they are implemented to demolish the priority inversion.

What is Priority Inheritance ?

Priority Inheritance :- In this method, what happens is if lets say process A is holding a lock at some instant of time. But at that time if higher priority also wants to join then the lower priority process needs to be boosted to equal the higher priority process in order to meet the needs untill the lock is released from the lower priority task

According to him, if we use PI in kernel then there will be a lot of overhead due to multiple system calls which will degrade the performance eventually.

I support Inglo’s view on this topic.

According to Inglo, it is almost **impossible to create a lockless solution** which according to Linus is one of the way of solving the problem of priority inversion. He explains more in detail with regard to this topic by telling that kernel is itself a very complex program and to find a lockless mechanism is an exception **when the ratio of lockless vs locky code is found in between 1:10 and 1:100.[2]**

Moreover, he describes that in few appliances, they do not prefer lockless algorithms. If we use lockless algorithms then we might find the that the code which was designed to be robust and easilty understandable and implemented wont be possible any further.There are many examples of multi-tasking(Music Jack), where a high priority audio playback thread is combined with medium priority construct audio data thread and low priority display colour stuff thread which share a short held locks.[2]

Finally, to conclude he suggests that if we do not use PI in kernel then while sharing a lock from J1(HP) and J0(LP) and even if we make sure that all the parts of the code are very systematically coded but still kernel wont be able to tell you the deterministic execution of HP[2].

Thus according to me, though the overhead is more but using locky algorithms is more suitable.

Now Lets discuss about PI Futex,

According to me , **PI Futex would fix the unbounded priority inversion issue but not completely.**

PI Futex is implemented in the user space. PI-futex can be taken without involving the kernel at all by the methods of locking and unlocking of PI.

A futex consists of a kernel space wait queue that is attached to an atomic integer in user space.[4] Mostly all the processes and threads , they operate on the integer entirely in user space and only on high priority system calls , they request operations on the wait queue.[4]

What would a normal Futex do?

A normal Futex would not execute anything untill and unless the lock is there in the system**[3]**

But in PI -Futex, the priorities are not the same as a normal futex. Here in this case,

In PI-Futex, if task J0 is blocking some task lets say J1 where according to priority (J1>J0) then J0 = J1 happens. Once J0 completes its task whose priority right now is J1, again the situation happens like (J1>J0)

These futexes have a very **less overhead** in the kernel space so even though it **does not solve the problem of deadlocks this method is very useful in solving the problem of unbounded priority inversion issue.**

Q-2

Re-entrant functions do not rely on global variables. There are few functions which need to store their ongoing progress of work. What will reentrant functions do? It will allow to specify this pointer but it wont get stored as a global. A thread can call a particular function at any time unless and untill if some other thread is not trying to get the same function at the same time.[5]

The class isn't thread-safe, because if multiple threads try to modify the data member `n`, the result is undefined.

This storage is exclusive to the calling function it can be re-entered. We can say that , re-entrant is often thread safe but thread safe does not always mean re-entrant. **For example, malloc(). If a thread safe function does not loose data or it does not gets unchanged then it can definitely be preempted by other call.**

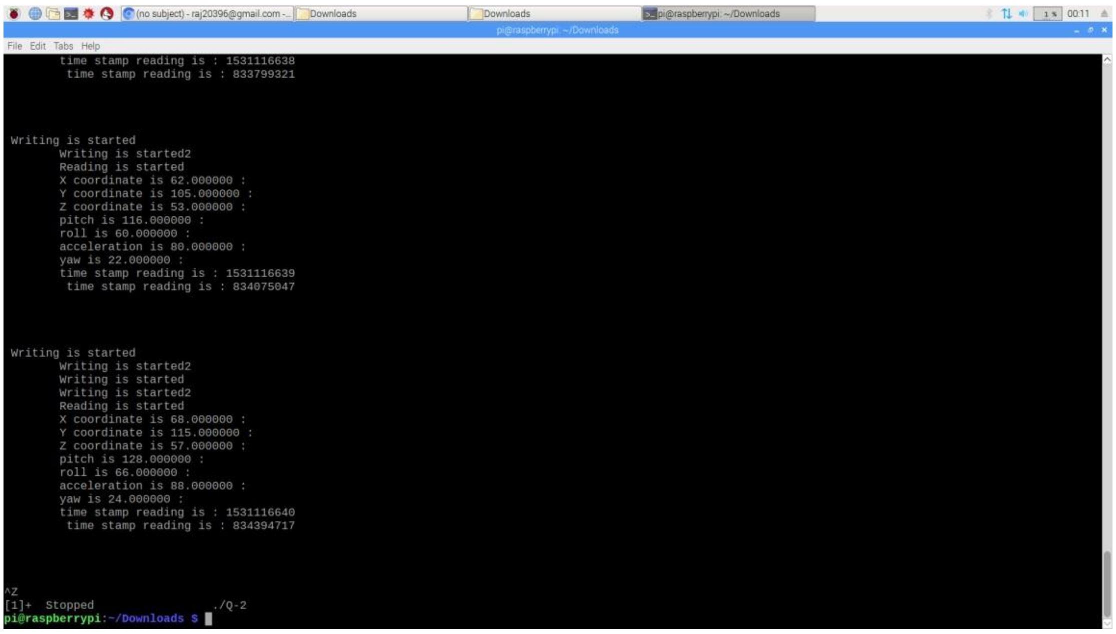
As described, there are 3 methods how we can handle a data in a thread safe way and their impact on real time services

1) **We should not use shared or global data in the function. We should only use the stack.**
   Implementing this method, we need to make sure that no static keyword is used in this, Just global shared data. When multiple threads access the same instance or static variable we need to place the path in such a way to that variable that inter connections do not happen. Here only Local arguments and values are used and thus making sure that .**This method uses no shared memory thus do not have much effect on real time systems. Data from other tasks would not matter in this type of method because here every task can work independently.**

2) **In second method, we need to make functions which use thread indexed global data.** For doing this , before writing the code we need to know how many threads we will need from the beginning only . Here in this case there is a possibility of data loss while in reentrant. We need to make sure that the data is not lost which is a very important aspect in real time systems because in such systems we want the data is protected. Moreover, we also want that after the completion of one task only the new task starts. If some tasks starts during the running of a another task then the possibility of data task increases a lot.

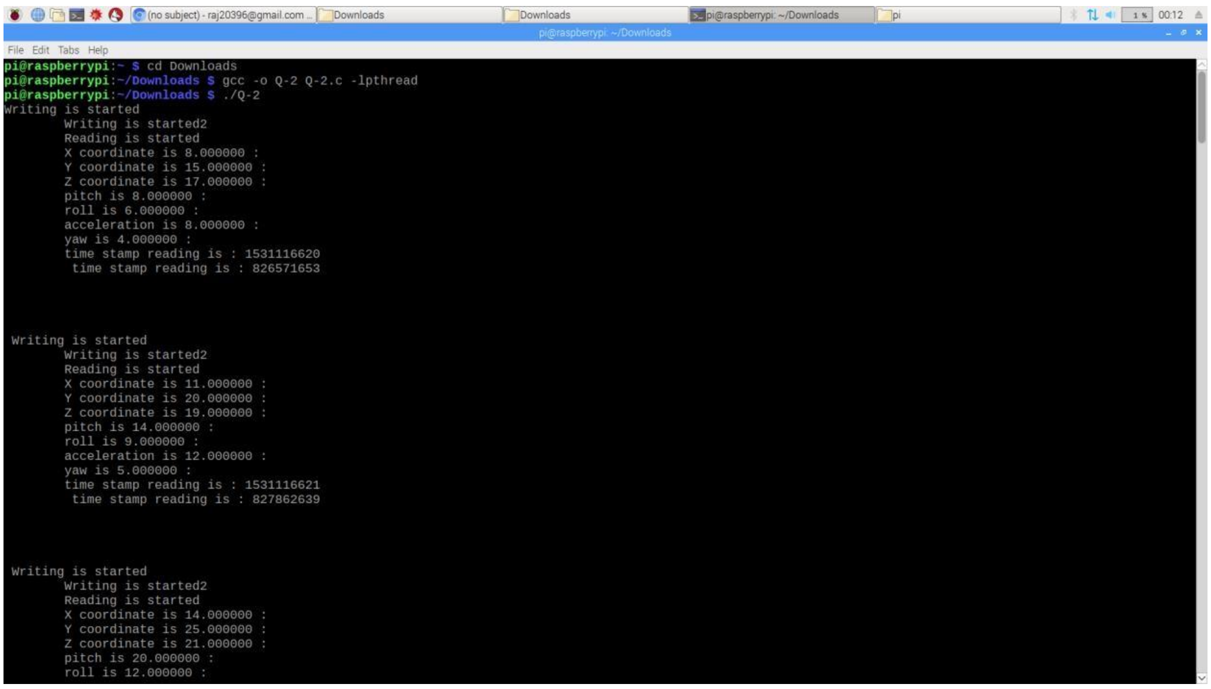
3) **The third method is of functions which use shared memory global data, but synchronizes access to it using a MUTEX semaphore critical section wrapper**. In this method, we need to protect critical sections of data while re-entrant function calls are executed. When using such a method we need to be very sure that data corruption does not exist which can be easily done if a low priority task is interrupted by a high priority task when both these tasks uses the same shared data. Mutex locking and unlocking mechanism is the best way to do that. If such schemes are implemented then we can protect the data which is an essential part while designing any real time embedded systems.

Using a MUTEX, provide an example using RT-Linux Pthreads that does a thread safe update of a complex state with a timestamp (pthread_mutex_lock).

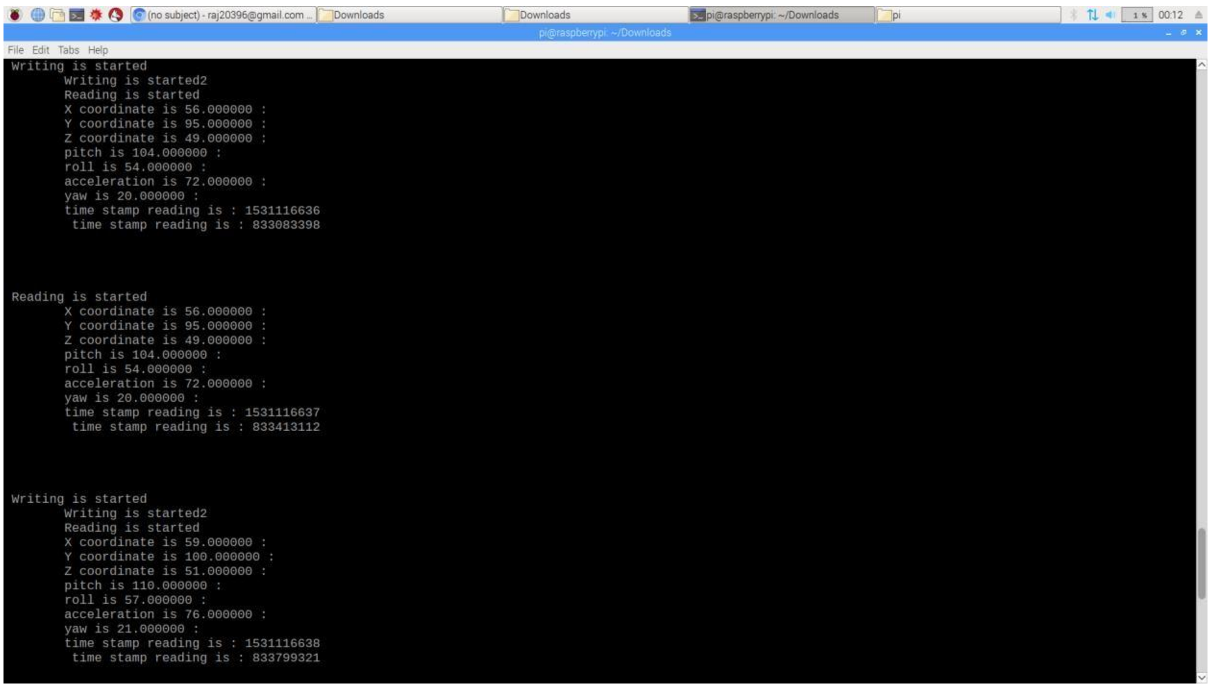

(no subject) - raj20396@gmail.com -...
Downloads
Downloads
pi@raspberrypi: ~/Downloads
00:11
pi@raspberrypi: ~/Downloads
File Edit Tabs Help

```
        time stamp reading is : 1531116638
         time stamp reading is : 833799321



 Writing is started
        Writing is started2
        Reading is started
        X coordinate is 62.000000 :
        Y coordinate is 105.000000 :
        Z coordinate is 53.000000 :
        pitch is 116.000000 :
        roll is 60.000000 :
        acceleration is 80.000000 :
        yaw is 22.000000 :
        time stamp reading is : 1531116639
         time stamp reading is : 834075047



 Writing is started
        Writing is started2
        Writing is started
        Writing is started2
        Reading is started
        X coordinate is 68.000000 :
        Y coordinate is 115.000000 :
        Z coordinate is 57.000000 :
        pitch is 128.000000 :
        roll is 66.000000 :
        acceleration is 88.000000 :
        yaw is 24.000000 :
        time stamp reading is : 1531116640
         time stamp reading is : 834394717



^Z
[1]+  Stopped                 ./Q-2
pi@raspberrypi:~/Downloads $ 
```

Fig ure 2 :- Co de out put (1)


(no subject) - raj20396@gmail.com ...
Downloads
Downloads
pi@raspberrypi: ~/Downloads
pi
00:12
pi@raspberrypi: ~/Downloads
File Edit Tabs Help

```
pi@raspberrypi:~ $ cd Downloads
pi@raspberrypi:~/Downloads $ gcc -o Q-2 Q-2.c -lpthread
pi@raspberrypi:~/Downloads $ ./Q-2
Writing is started
        Writing is started2
        Reading is started
        X coordinate is 8.000000 :
        Y coordinate is 15.000000 :
        Z coordinate is 17.000000 :
        pitch is 8.000000 :
        roll is 6.000000 :
        acceleration is 8.000000 :
        yaw is 4.000000 :
        time stamp reading is : 1531116620
         time stamp reading is : 826571653




 Writing is started
        Writing is started2
        Reading is started
        X coordinate is 11.000000 :
        Y coordinate is 20.000000 :
        Z coordinate is 19.000000 :
        pitch is 14.000000 :
        roll is 9.000000 :
        acceleration is 12.000000 :
        yaw is 5.000000 :
        time stamp reading is : 1531116621
         time stamp reading is : 827862639




 Writing is started
        Writing is started2
        Reading is started
        X coordinate is 14.000000 :
        Y coordinate is 25.000000 :
        Z coordinate is 21.000000 :
        pitch is 20.000000 :
        roll is 12.000000 :
```

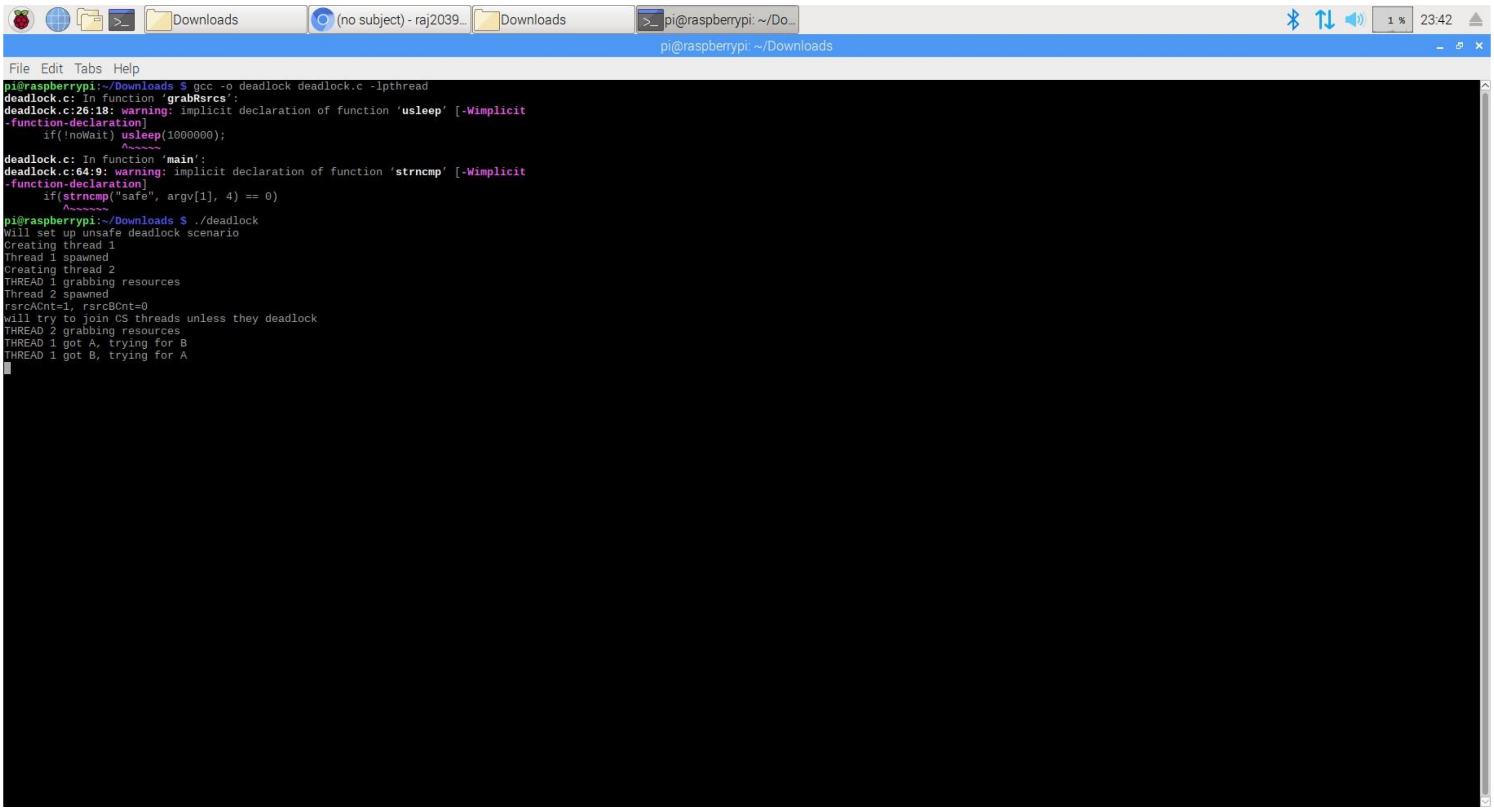

(no subject) - raj20396@gmail.com ...
Downloads
Downloads
pi@raspberrypi: ~/Downloads
pi
00:12
pi@raspberrypi: ~/Downloads
File
Edit
Tabs
Help

```
Writing is started
       Writing is started2
       Reading is started
       X coordinate is 56.000000 :
       Y coordinate is 95.000000 :
       Z coordinate is 49.000000 :
       pitch is 104.000000 :
       roll is 54.000000 :
       acceleration is 72.000000 :
       yaw is 20.000000 :
       time stamp reading is : 1531116636
        time stamp reading is : 833083398



Reading is started
       X coordinate is 56.000000 :
       Y coordinate is 95.000000 :
       Z coordinate is 49.000000 :
       pitch is 104.000000 :
       roll is 54.000000 :
       acceleration is 72.000000 :
       yaw is 20.000000 :
       time stamp reading is : 1531116637
        time stamp reading is : 833413112



Writing is started
       Writing is started2
       Reading is started
       X coordinate is 59.000000 :
       Y coordinate is 100.000000 :
       Z coordinate is 51.000000 :
       pitch is 110.000000 :
       roll is 57.000000 :
       acceleration is 76.000000 :
       yaw is 21.000000 :
       time stamp reading is : 1531116638
        time stamp reading is : 833799321
```

Figure 3 output part (2)

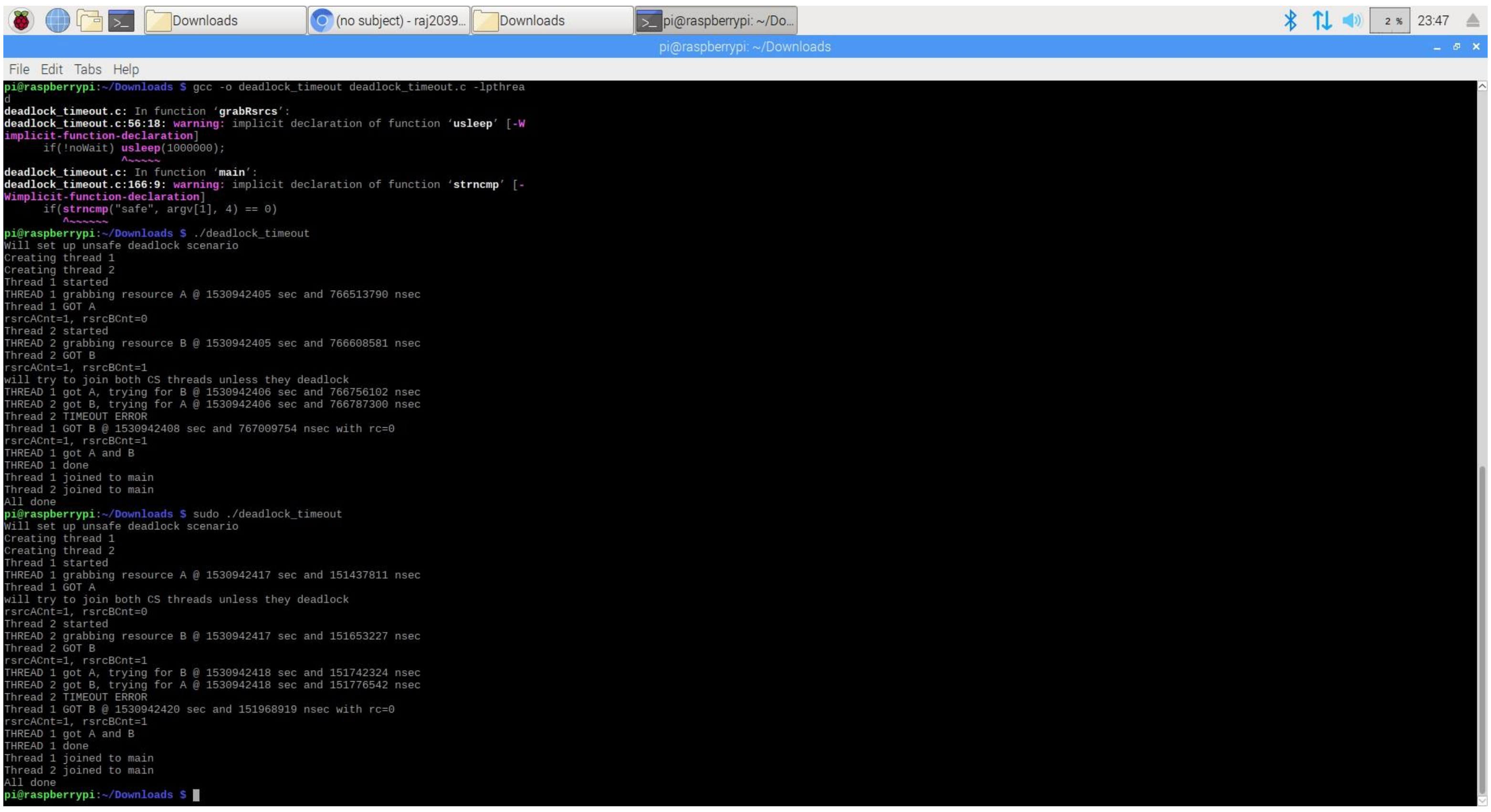
As we can see in the figure, along with x,y,z coordinate pitch , roll, acceleration, yaw are incresing by different values and everytime the change is made the timestamp is increased also. In this code I have used locking and unlocking mechanism which is used to remove the problem of possible deadlock in the system.

Q-3



```
pi@raspberrypi:~/Downloads $ gcc -o deadlock deadlock.c -lpthread
deadlock.c: In function ‘grabRsrcs’:
deadlock.c:26:18: warning: implicit declaration of function ‘usleep’ [-Wimplicit
-function-declaration]
      if(!noWait) usleep(1000000);
                  ^~~~~~
deadlock.c: In function ‘main’:
deadlock.c:64:9: warning: implicit declaration of function ‘strncmp’ [-Wimplicit
-function-declaration]
      if(strncmp("safe", argv[1], 4) == 0)
         ^~~~~~~
pi@raspberrypi:~/Downloads $ ./deadlock
Will set up unsafe deadlock scenario
Creating thread 1
Thread 1 spawned
Creating thread 2
THREAD 1 grabbing resources
Thread 2 spawned
rsrcACnt=1, rsrcBCnt=0
will try to join CS threads unless they deadlock
THREAD 2 grabbing resources
THREAD 1 got A, trying for B
THREAD 1 got B, trying for A
```

Figure 4 Deadlock Code Output(Code given by professor)



```
pi@raspberrypi:~/Downloads $ gcc -o deadlock_timeout deadlock_timeout.c -lpthrea
d
deadlock_timeout.c: In function ‘grabRsrcs’:
deadlock_timeout.c:56:18: warning: implicit declaration of function ‘usleep’ [-W
implicit-function-declaration]
      if(!noWait) usleep(1000000);
                  ^~~~~~
deadlock_timeout.c: In function ‘main’:
deadlock_timeout.c:166:9: warning: implicit declaration of function ‘strncmp’ [-
Wimplicit-function-declaration]
      if(strncmp("safe", argv[1], 4) == 0)
         ^~~~~~~
pi@raspberrypi:~/Downloads $ ./deadlock_timeout
Will set up unsafe deadlock scenario
Creating thread 1
Creating thread 2
Thread 1 started
THREAD 1 grabbing resource A @ 1530942405 sec and 766513790 nsec
Thread 1 GOT A
rsrcACnt=1, rsrcBCnt=0
Thread 2 started
THREAD 2 grabbing resource B @ 1530942405 sec and 766608581 nsec
Thread 2 GOT B
rsrcACnt=1, rsrcBCnt=1
will try to join both CS threads unless they deadlock
THREAD 1 got A, trying for B @ 1530942406 sec and 766756102 nsec
THREAD 2 got B, trying for A @ 1530942406 sec and 766787300 nsec
Thread 2 TIMEOUT ERROR
Thread 1 GOT B @ 1530942408 sec and 767009754 nsec with rc=0
rsrcACnt=1, rsrcBCnt=1
THREAD 1 got A and B
THREAD 1 done
Thread 1 joined to main
Thread 2 joined to main
All done
pi@raspberrypi:~/Downloads $ sudo ./deadlock_timeout
Will set up unsafe deadlock scenario
Creating thread 1
Creating thread 2
Thread 1 started
THREAD 1 grabbing resource A @ 1530942417 sec and 151437811 nsec
Thread 1 GOT A
will try to join both CS threads unless they deadlock
rsrcACnt=1, rsrcBCnt=0
Thread 2 started
THREAD 2 grabbing resource B @ 1530942417 sec and 151653227 nsec
Thread 2 GOT B
rsrcACnt=1, rsrcBCnt=1
THREAD 1 got A, trying for B @ 1530942418 sec and 151742324 nsec
THREAD 2 got B, trying for A @ 1530942418 sec and 151776542 nsec
Thread 2 TIMEOUT ERROR
Thread 1 GOT B @ 1530942420 sec and 151968919 nsec with rc=0
rsrcACnt=1, rsrcBCnt=1
THREAD 1 got A and B
THREAD 1 done
Thread 1 joined to main
Thread 2 joined to main
All done
pi@raspberrypi:~/Downloads $ 
```

Figure 5 Deadlock_timeout (Code given by professor)

Deadlock :- It is a situation when multiple threads are blocked because of resource requirements which can never be satisfied by either of tasks[10]

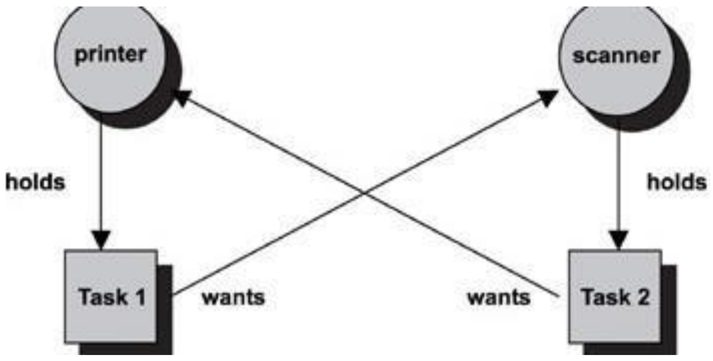


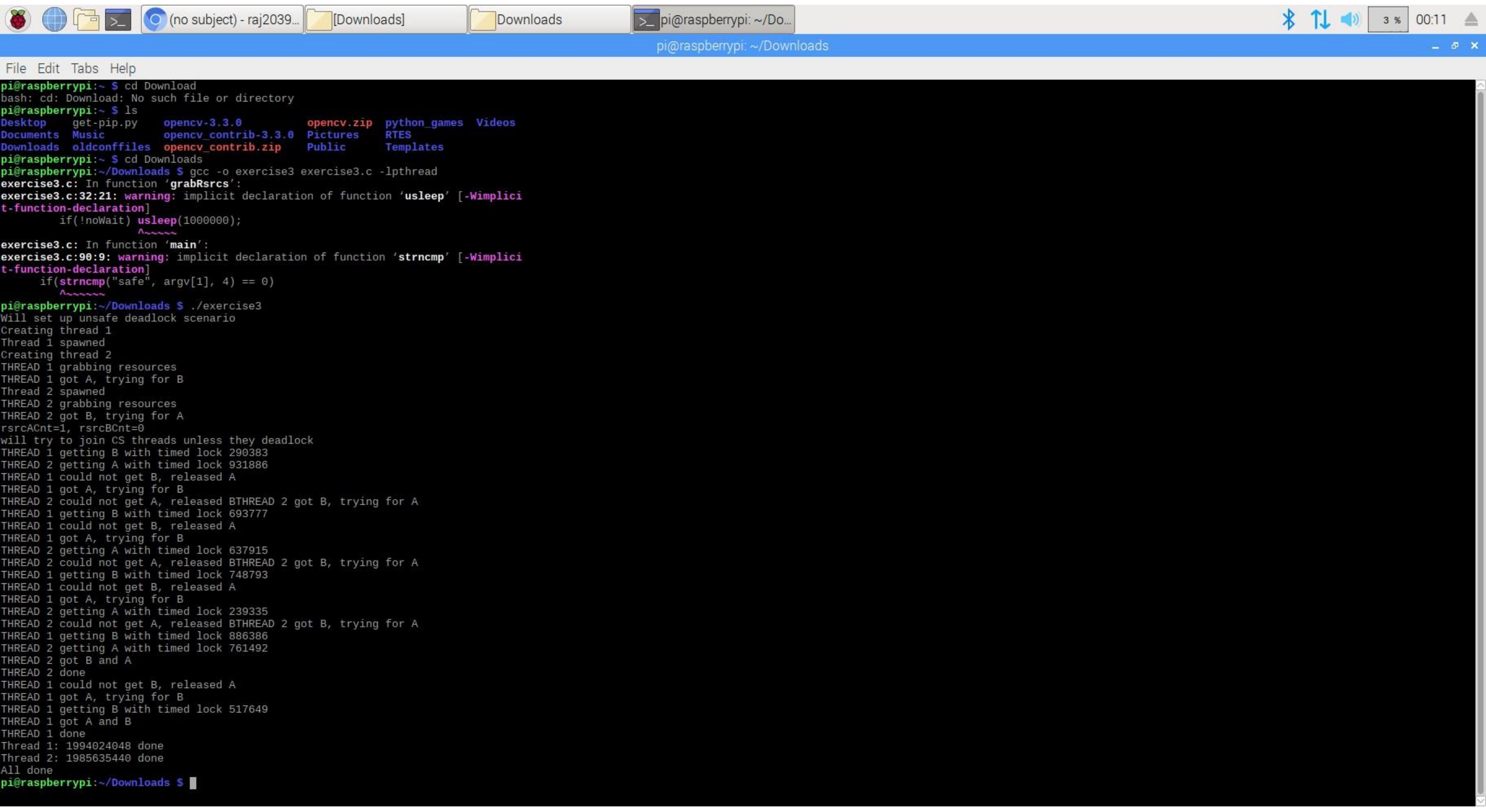
Figure 6 Deadlock Example [10]

As shown in the figure , task 1 wants scanner but it is holding the printer. Whereas task 2 wants printer but it is holding scanner. Now in this case task 1 cannot execute until it has scanner and task 2 cannot execute until it has printer. Thus creates a deadlock. To overcome this problem we need to set a SPECIFIC TIMEOUT. For example, after sometime if task 1 is not able to execute scanner then it will leave both printer and scanner, thereby making printer available for task 2 which initially wanted printer only to execute and as a result the deadlock can be removed. The same process for the task 2 leaving printer and scanner both , thus making the scanner available for task 1.This method can be implemented with the help of **pthread_mutex_timelock() function.**

Live lock :- The process in which the release happens at the same time. For example, Task 1 releases printer and scanner at the same time when the task 2 releases scanner printer. Thus in this case no task gets what they wanted for the execution.

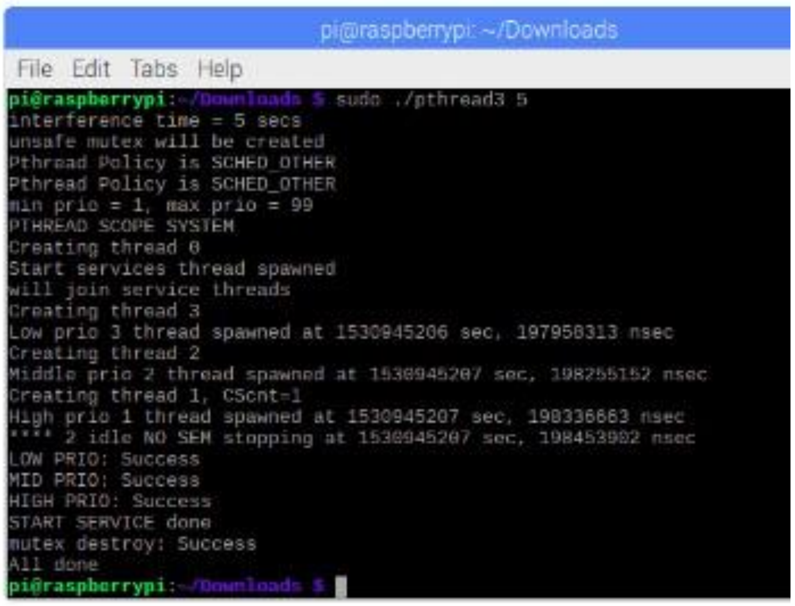
Thus there are 2 such problems which can be solved just by giving a little bit of timeout like **mutex timeout (it can be random)**. For example, if task 1 is given a timeout value then in such a way one task will still be blocking a thing while the other at that time can access another thing , thus solving the problem.

Figure 7 Solved Deadlock (Deadlock_Timeout demo code given by professor is edited to solve the deadloc)



```
pi@raspberrypi:~ $ cd Download
bash: cd: Download: No such file or directory
pi@raspberrypi:~ $ ls
Desktop    get-pip.py    opencv-3.3.0          opencv.zip  python_games  Videos
Documents  Music         opencv_contrib-3.3.0  Pictures    RTES
Downloads  oldconffiles  opencv_contrib.zip    Public      Templates
pi@raspberrypi:~ $ cd Downloads
pi@raspberrypi:~/Downloads $ gcc -o exercise3 exercise3.c -lpthread
exercise3.c: In function ‘grabRsrcs’:
exercise3.c:32:21: warning: implicit declaration of function ‘usleep’ [-Wimplici
t-function-declaration]
         if(!noWait) usleep(1000000);
                     ^~~~~~
exercise3.c: In function ‘main’:
exercise3.c:90:9: warning: implicit declaration of function ‘strncmp’ [-Wimplici
t-function-declaration]
      if(strncmp("safe", argv[1], 4) == 0)
         ^~~~~~~
pi@raspberrypi:~/Downloads $ ./exercise3
Will set up unsafe deadlock scenario
Creating thread 1
Thread 1 spawned
Creating thread 2
THREAD 1 grabbing resources
THREAD 1 got A, trying for B
Thread 2 spawned
THREAD 2 grabbing resources
THREAD 2 got B, trying for A
rsrcACnt=1, rsrcBCnt=0
will try to join CS threads unless they deadlock
THREAD 1 getting B with timed lock 290383
THREAD 2 getting A with timed lock 931886
THREAD 1 could not get B, released A
THREAD 1 got A, trying for B
THREAD 2 could not get A, released BTHREAD 2 got B, trying for A
THREAD 1 getting B with timed lock 693777
THREAD 1 could not get B, released A
THREAD 1 got A, trying for B
THREAD 2 getting A with timed lock 637915
THREAD 2 could not get A, released BTHREAD 2 got B, trying for A
THREAD 1 getting B with timed lock 748793
THREAD 1 could not get B, released A
THREAD 1 got A, trying for B
THREAD 2 getting A with timed lock 239335
THREAD 2 could not get A, released BTHREAD 2 got B, trying for A
THREAD 1 getting B with timed lock 886386
THREAD 2 getting A with timed lock 761492
THREAD 2 got B and A
THREAD 2 done
THREAD 1 could not get B, released A
THREAD 1 got A, trying for B
THREAD 1 getting B with timed lock 517649
THREAD 1 got A and B
THREAD 1 done
Thread 1: 1994024048 done
Thread 2: 1985635440 done
All done
pi@raspberrypi:~/Downloads $
```

For pthread3 and pthread3ok code,



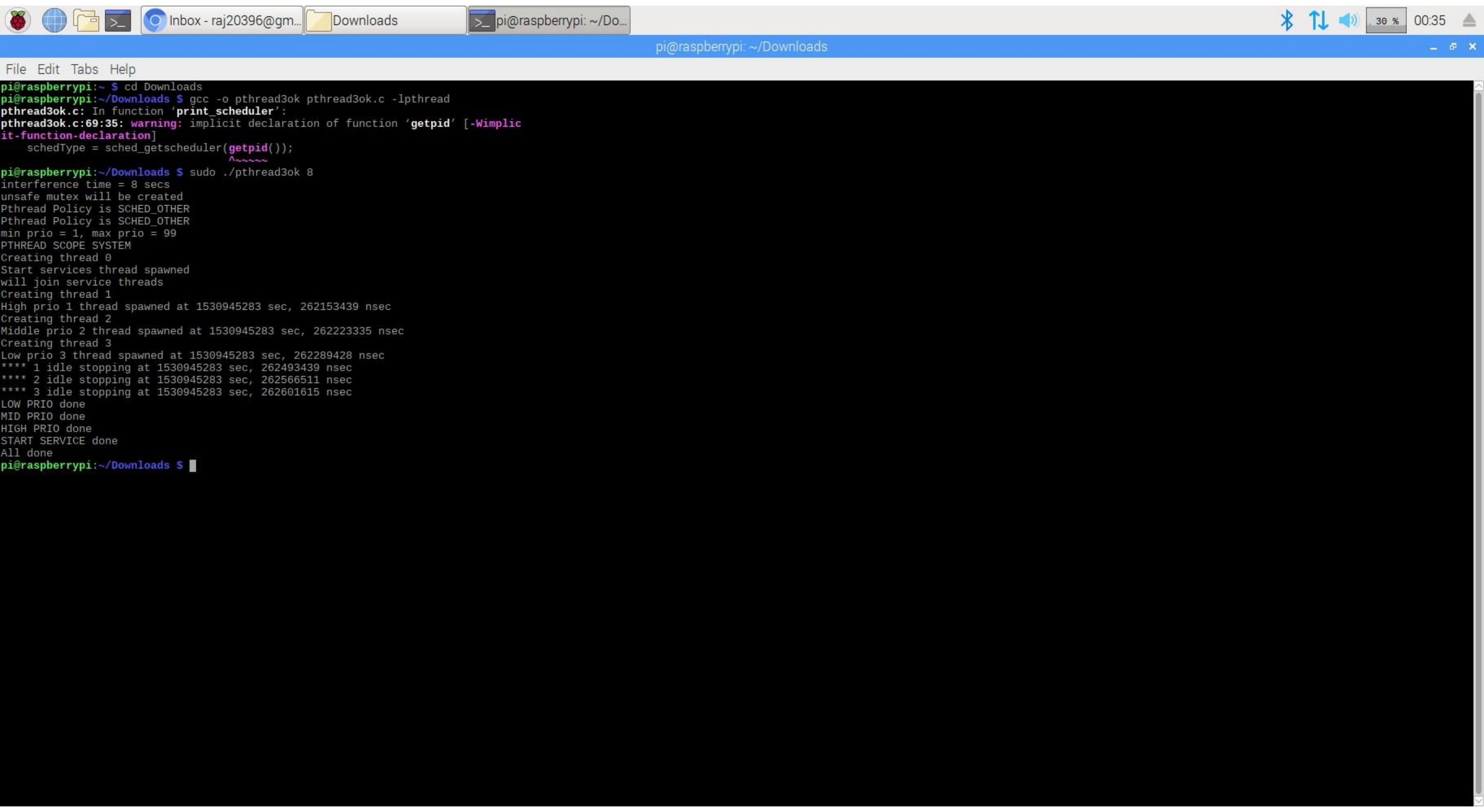
```
pi@raspberrypi:~/Downloads $ sudo ./pthread3 5
interference time = 5 secs
unsafe mutex will be created
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_OTHER
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM
Creating thread 0
Start services thread spawned
will join service threads
Creating thread 3
Low prio 3 thread spawned at 1530945206 sec, 197958313 nsec
Creating thread 2
Middle prio 2 thread spawned at 1530945207 sec, 198255152 nsec
Creating thread 1, CScnt=1
High prio 1 thread spawned at 1530945207 sec, 198336663 nsec
**** 2 idle NO SEM stopping at 1530945207 sec, 198453902 nsec
LOW PRIO: Success
MID PRIO: Success
HIGH PRIO: Success
START SERVICE done
mutex destroy: Success
All done
pi@raspberrypi:~/Downloads $
```

Figure 8 Pthread3 solved (Code edited from the demo code given by Professor)

In this code, I have changed the “gettimeofday” API to clock_gettime(CLOCK_REALTIME,&timeNow). **As far as priority inversion is concerned in this situation ,we have 3 priorities in this code. The code needs sudo privilages in order to run the sched fifo without that the code wont run.** Now lets assume that low priority has started doing its task and since it is sharing its resources with higher priority , higher priority can also come in the picture. Now while the higher priority is doing its task,there might be a possibility that medium priority task wants to interface with the highest priority task then it will be blocked by the higher priority task and cannot enter.

Figure 9 Pthread3ok solved (Code edited from the demo code given by Professor)

In this code, pthread3ok it increases the priority of low priority task so it can match with the high priority one which are already sharing their resources as described in the above code but here medium priority wont be able to interfere even so there is no chance of medium priority getting kicked off by the higher priority since it wont be able to enter. Thus we conclude that in this case only we can avoid priority inversion but since there are many other drawbacks we cannot completely rely on linux systems for the applications of real time systems. [11]



```
pi@raspberrypi:~ $ cd Downloads
pi@raspberrypi:~/Downloads $ gcc -o pthread3ok pthread3ok.c -lpthread
pthread3ok.c: In function ‘print_scheduler’:
pthread3ok.c:69:35: warning: implicit declaration of function ‘getpid’ [-Wimplic
it-function-declaration]
    schedType = sched_getscheduler(getpid());
                                   ^~~~~~
pi@raspberrypi:~/Downloads $ sudo ./pthread3ok 8
interference time = 8 secs
unsafe mutex will be created
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_OTHER
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM
Creating thread 0
Start services thread spawned
will join service threads
Creating thread 1
High prio 1 thread spawned at 1530945283 sec, 262153439 nsec
Creating thread 2
Middle prio 2 thread spawned at 1530945283 sec, 262223335 nsec
Creating thread 3
Low prio 3 thread spawned at 1530945283 sec, 262289428 nsec
**** 1 idle stopping at 1530945283 sec, 262493439 nsec
**** 2 idle stopping at 1530945283 sec, 262566511 nsec
**** 3 idle stopping at 1530945283 sec, 262601615 nsec
LOW PRIO done
MID PRIO done
HIGH PRIO done
START SERVICE done
All done
pi@raspberrypi:~/Downloads $ 
```

Description on RT_PREEMPT_PATCH According to me , in Linux Kernel, total removal of priority inversion is not possible. This method of RT_PREEMPT_PATCH is useful a lot but it can not completey remove the situation. I have discussed this in the below paragraph

Lets suppose there are 3 tasks.

Task 1 – Highest Priority

Task 2 – Middle Priority

Task 3 – Lower Priority

Now, in linux systems the problem of unbounded priority inversion happens. If task 3 is sharing its resources with the system call and then there is a system which has a priority between these

2 tasks. Now, the low priority is in its critical section and there is no possible way that it will allow the higher priority task to do its work and as a result when the middle priority task interferes into this situation the problem of unbounded priority inversion is created in this. Until and unless the work of Task 1 is not completed and it does not get out of the critical section other task cannot overcome this.

Now, lets discuss about the solution for this problm. One of the possible ways to solve this is the method of RT_PREEMPT _PATCH.

RT_PREEMPT_PATCH :- This patch is very useful in assigning the priorities of the tasks. Here you can change the priorities of any task you want. Since you are able to set the priorities then you can also solve the problem of unbounded priority inversion. Until the lower priority task is running we can definitely change the priority if the lower priority task to a higher value. But this change is only possible on a temperory basis. We can do it only till the lower priority task is running. Once the Lower priority task is out of the critical section then we can no longer implement that priority. Now in such cases even if medium priority task tried to jump in, still we can increase the lower priority task to a priority higher than the medium priority task and thus now the unbounded priority inversion problem will be converted into a fixed priority inversion thus we can overcome that problem.

Thus my point of using RTOS instead of Linux proves for the Hard real time and soft real time systems.

Q-4

As per the posix_mq.c code given by the professor, in POSIX message queue there are 2 threads. One is sender and another is receiver.

POSIX message queues allow processes to exchange data in the form of messages.

In such processes, the receive thread is always having the highest priority than the sender thread and thus we have written the receiving thread first which is followed by the sender thread.

There are many message queue attributes written in the code. For example, mq_maxmsg,mq_msgsize,mq_flags.As we move on, there is attribute called **taskSpawn(),** where it creates and activates a new task with a specified priority and options and returns a system assigned ID where according to this syntax the command is created.

Syntax of taskspawn int taskSpawn

```
   (
   char *  name,           /* name of new task (stored at pStackBase) */
   int     priority,        /* priority of new task */     int     options,
/* task option word */     int     stackSize,       /* size (bytes) of
stack needed plus name */     FUNCPTR entryPt,         /* entry
point of new task */     int     arg1,           /* 1st of 10 req'd task
args to pass to func */
   int     arg2,
int     arg3,     int
arg4,     int
arg5,     int
arg6,     int
arg7,     int
arg8,     int
arg9,     int
arg10
   )
```

Now , in the two functions made (receiver and sender), **mq_open** is written which means it will open a new message queue or it will create a new message queue. Once the message queue is opened it will start sharing the message queue for sender and the receiver. **mq_receive** is used to receive the message and **mq_send** is used to send the message.

Now lets assume a case if the queue is empty in the initial phase. In such scenerios the receiver wont be able to get anything. It has to wait untill the sender puts something in the queue. **In this code, priority is given as 30**,so when the message que is filled with some data then the mq_receive pre-empts the mq_sender and its priority increases.

This is how posix message queue is working in the given code.

Heap_mq:-

In the heap_mq.c code given by the professor, message queue implemented is a little different manner from the posix message queue.Here first of all the heap size is defined, which is **4096** in this code.Mostly all the API's are same as posix message queue. Here also, the priorities of receiver is higher than the sender.

In the heap implementation we use an additional API called the buffptr. Buffptr will point to the message that needs to be sent. Another API used here is the **memcpy** which is used to copy the address of the buffptr to the memory buffer. As soon as value reached the buffer than the receiver pre empts the sender .Thus mq_sender stops sending data any further and now mq_receive will start receiving the data.

Similarity between them :-

Both the message queue are almost the same structure wise. Both uses alsmost same API 's like mq_maxmsg, mq_msgsize. They both start sending messages and receive messages. Both are asynchronus.

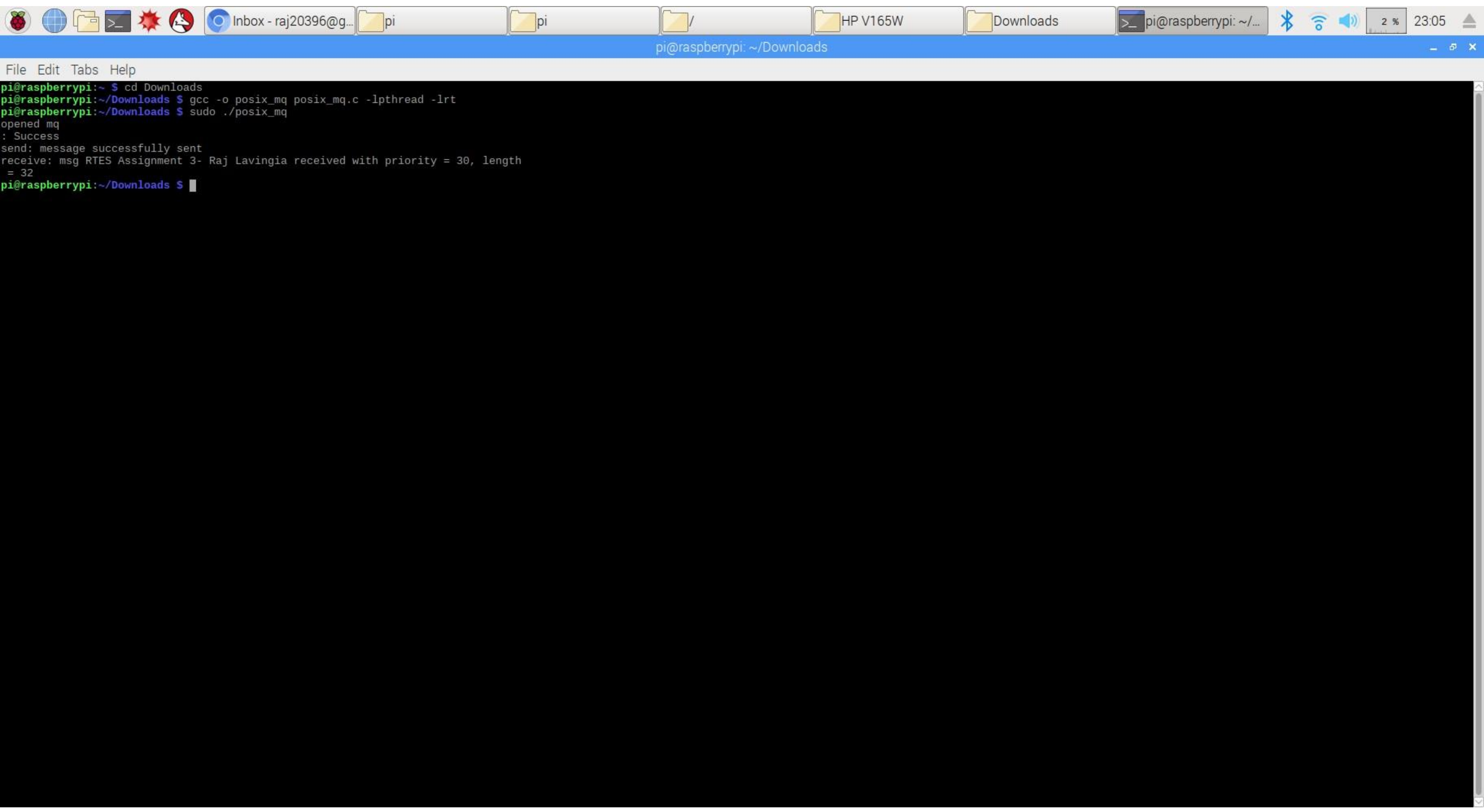
Difference between them :-

One of the difference between them is about the less overhead in the heap function due to large number of packets send at the same time instead of sending the whole message. Whereas POSIX_queue is used in POSIX mq_open function and thus creates a larger over head as compared to the heap.

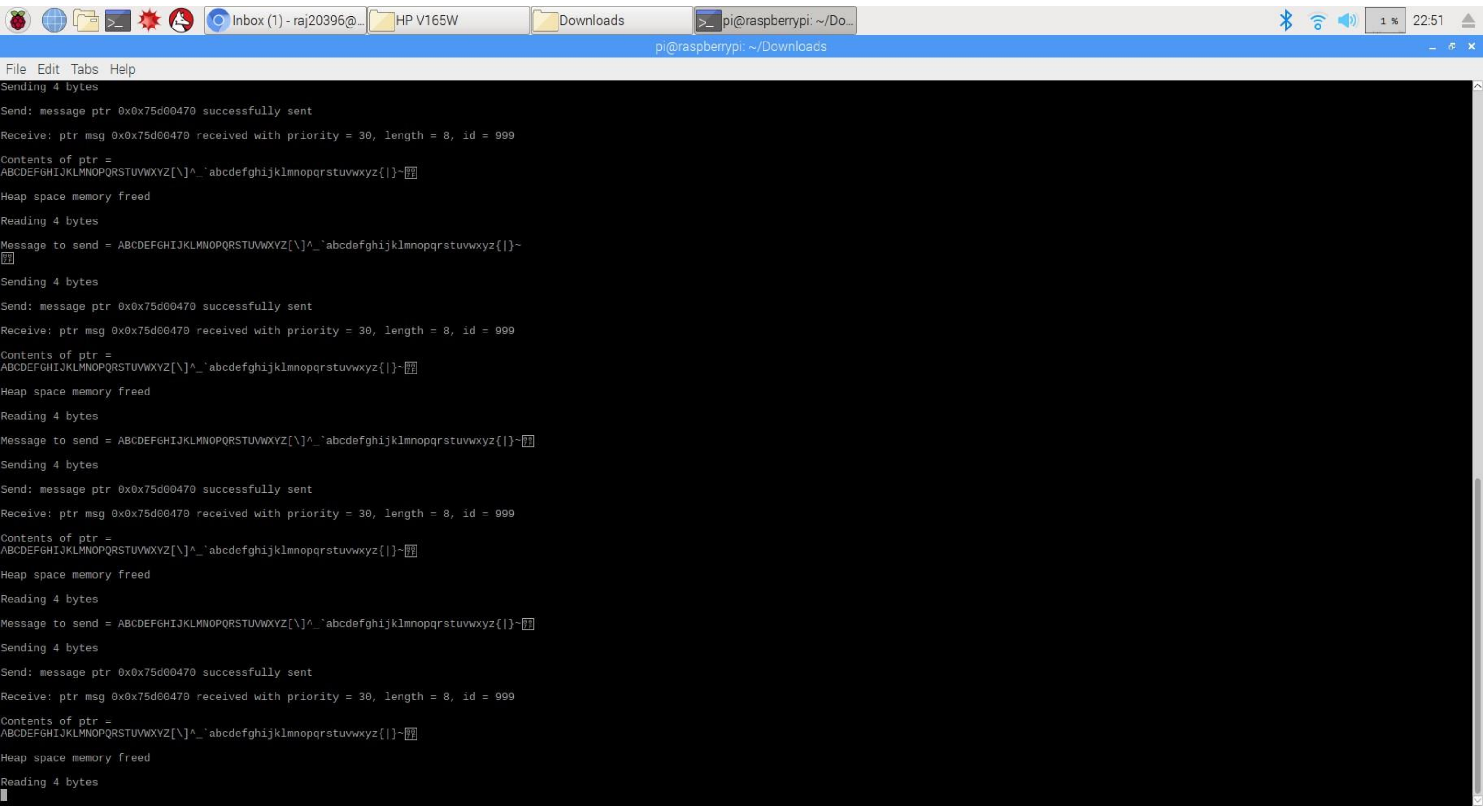
Message queues can avoid the problem of MUTEX priority inversion. Here, as we have seen from the codes above that when the receiver tries to receive the message even if the queue is emoty then it gets blocked and the priority is shifted to the sender which will come in the picture and it will eventually put something in the queue after which receiver can receive something

Similarly, if the message queue is filled but it tries to put some message in it then it is put into the queue and then when there is space in the message queue then only it allows the message to enter in it. As we have seen in this example, that when the data starts filling up then the priority shifts from sender to receiver. Thus the higher priority task executes first. Now whenever a lower priority task would try to enter , the higher priority task would block the lower priority task and thus the problem of priority inversion is sorted which can not be solved directly in linux since in linux, it does not have the property to avoid priority inversion.[11] **The best advantage of this is the higher priority task never gets replaced with a lower priority so the problem of priority inversion never happens.**

Posix_mq output picture

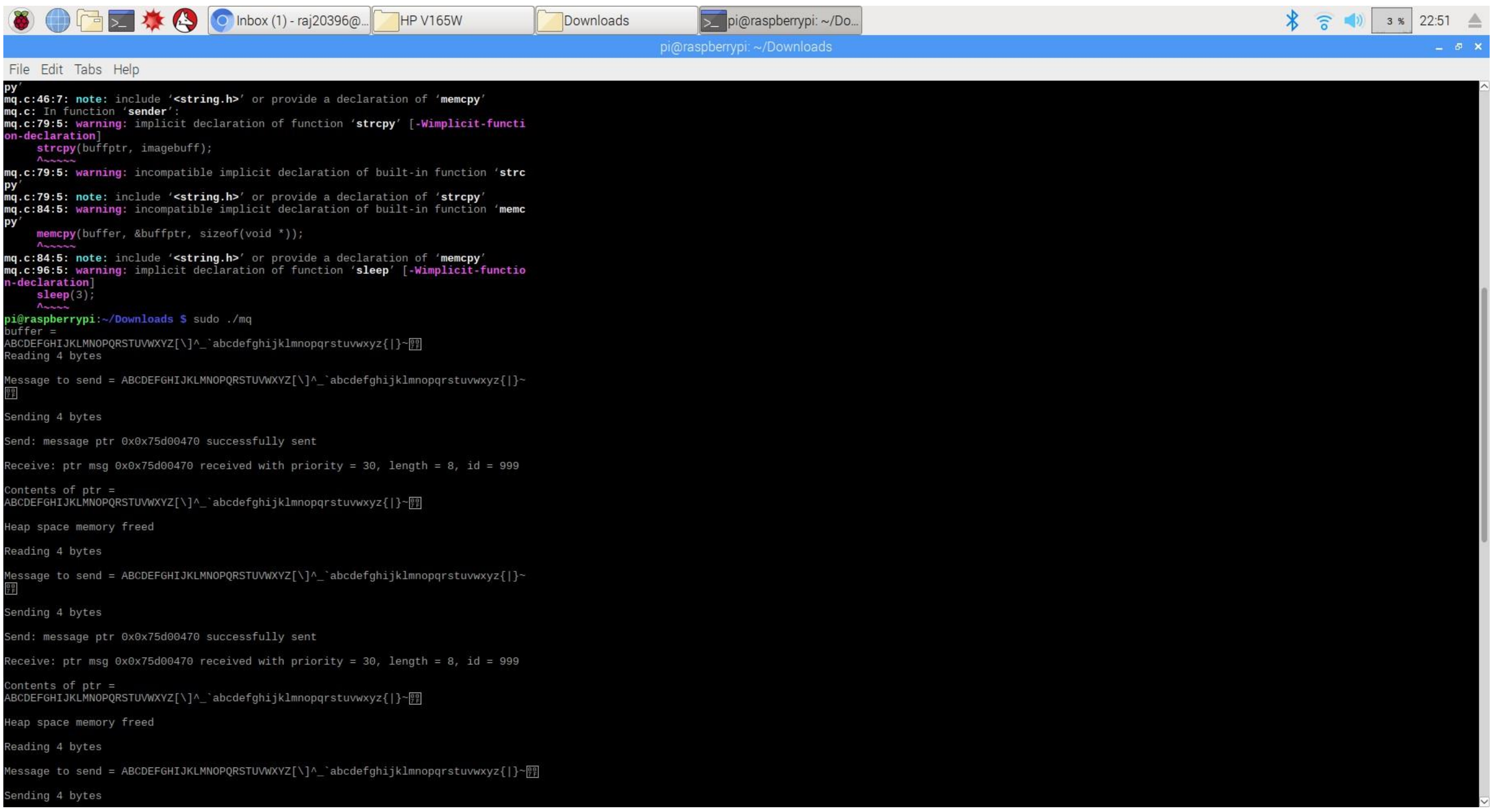


```
pi@raspberrypi:~ $ cd Downloads
pi@raspberrypi:~/Downloads $ gcc -o posix_mq posix_mq.c -lpthread -lrt
pi@raspberrypi:~/Downloads $ sudo ./posix_mq
opened mq
: Success
send: message successfully sent
receive: msg RTES Assignment 3- Raj Lavingia received with priority = 30, length
 = 32
pi@raspberrypi:~/Downloads $ 
```



```
Sending 4 bytes

Send: message ptr 0x0x75d00470 successfully sent

Receive: ptr msg 0x0x75d00470 received with priority = 30, length = 8, id = 999

Contents of ptr =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~▯

Heap space memory freed

Reading 4 bytes

Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
▯

Sending 4 bytes

Send: message ptr 0x0x75d00470 successfully sent

Receive: ptr msg 0x0x75d00470 received with priority = 30, length = 8, id = 999

Contents of ptr =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~▯

Heap space memory freed

Reading 4 bytes

Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~▯

Sending 4 bytes

Send: message ptr 0x0x75d00470 successfully sent

Receive: ptr msg 0x0x75d00470 received with priority = 30, length = 8, id = 999

Contents of ptr =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~▯

Heap space memory freed

Reading 4 bytes

Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~▯

Sending 4 bytes

Send: message ptr 0x0x75d00470 successfully sent

Receive: ptr msg 0x0x75d00470 received with priority = 30, length = 8, id = 999

Contents of ptr =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~▯

Heap space memory freed

Reading 4 bytes
```

Heap_mq output picture



```
py'
mq.c:46:7: note: include '<string.h>' or provide a declaration of 'memcpy'
mq.c: In function 'sender':
mq.c:79:5: warning: implicit declaration of function 'strcpy' [-Wimplicit-functi
on-declaration]
     strcpy(buffptr, imagebuff);
     ^~~~~~
mq.c:79:5: warning: incompatible implicit declaration of built-in function 'strc
py'
mq.c:79:5: note: include '<string.h>' or provide a declaration of 'strcpy'
mq.c:84:5: warning: incompatible implicit declaration of built-in function 'memc
py'
     memcpy(buffer, &buffptr, sizeof(void *));
     ^~~~~~
mq.c:84:5: note: include '<string.h>' or provide a declaration of 'memcpy'
mq.c:96:5: warning: implicit declaration of function 'sleep' [-Wimplicit-functio
n-declaration]
     sleep(3);
     ^~~~~
pi@raspberrypi:~/Downloads $ sudo ./mq
buffer =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~␡
Reading 4 bytes

Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
␡

Sending 4 bytes

Send: message ptr 0x0x75d00470 successfully sent

Receive: ptr msg 0x0x75d00470 received with priority = 30, length = 8, id = 999

Contents of ptr =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~␡

Heap space memory freed

Reading 4 bytes

Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
␡

Sending 4 bytes

Send: message ptr 0x0x75d00470 successfully sent

Receive: ptr msg 0x0x75d00470 received with priority = 30, length = 8, id = 999

Contents of ptr =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~␡

Heap space memory freed

Reading 4 bytes

Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~␡

Sending 4 bytes
```

Heap_mq output picture (2) Q-5

Watchdog Timer :- It is an electronic timer that is used to detect and recover from computer malfunctions.[12]. All the embedded systems products can not be continuously monitored by any human. It has to be an automated process. Here comes the importance the importance of a watchdog timer. It will sit in the place of human to keep a regular check on the system.

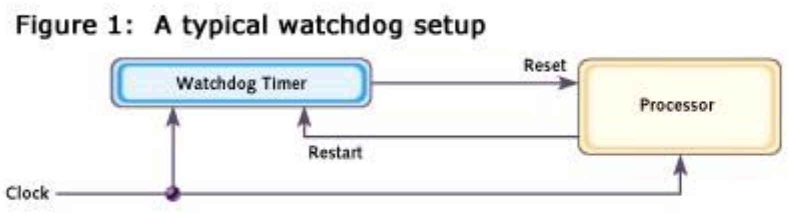


Figure 12 Watchdog Setup

[13]

Timeouts :- If one does not undertand the timing characteristics of the program then he might pick a timeout interval that is too short , this may lead to occasional resets of the system which may be difficult to diagnose.

Now lets again take the example provded in Q-3

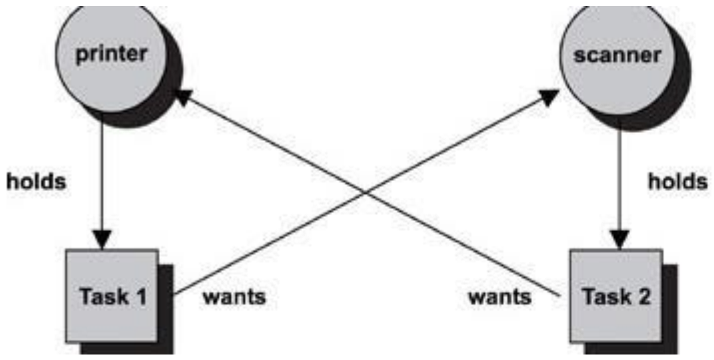


Figure 13 Deadlock Example [10]

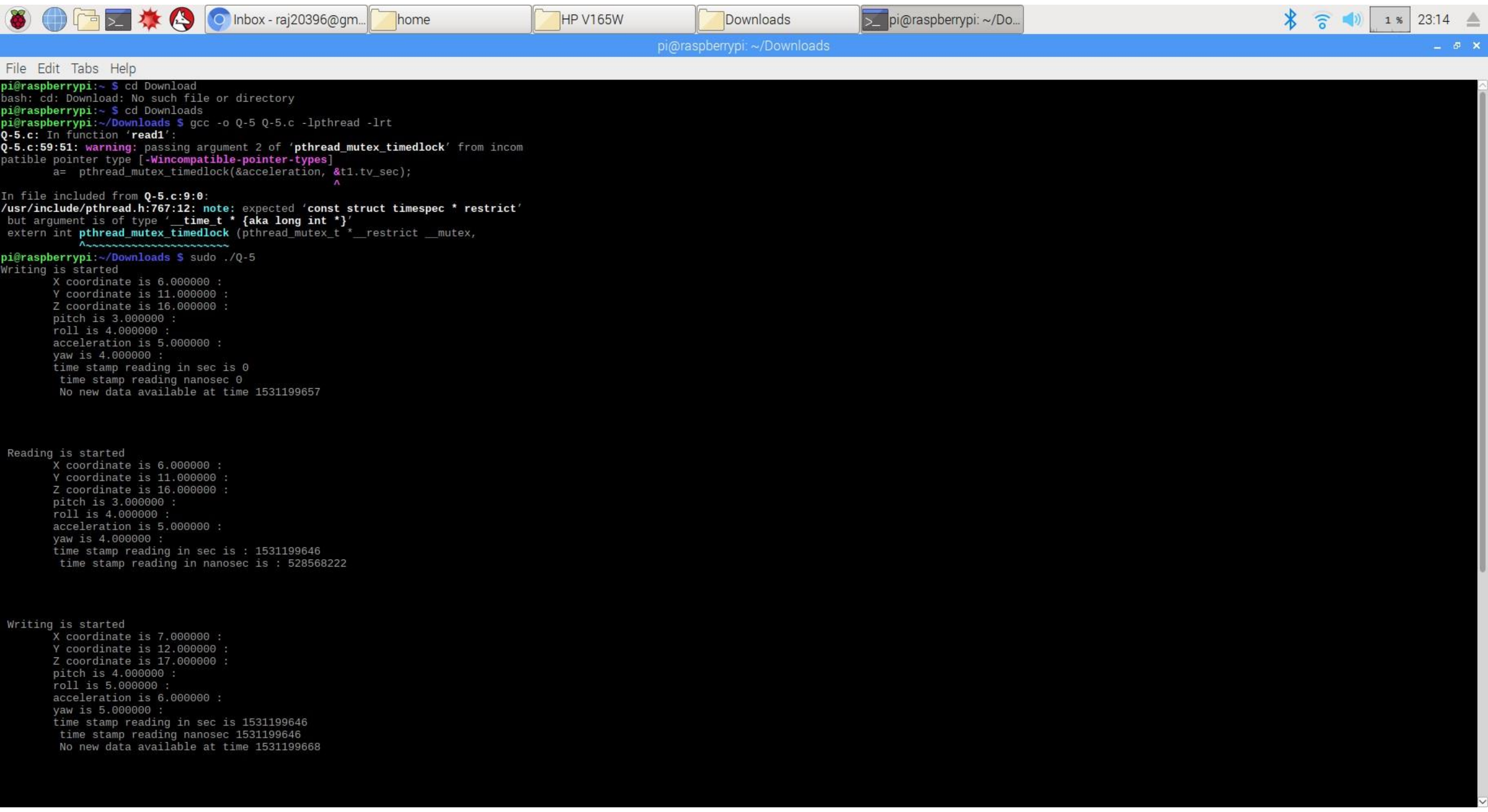
Here , if a situation of deadlock occurs then neither of the tasks will be able to do a timer reset. Thus the system is occupying a lot of resources because no periodic reset occurs. In such cases the system has to perform a hard reset. Because if the system does not do any hard reset that means the system is hanged eventually. While doing the hard reset it would find the problems in the system and then put those problems data into the log so anyone can check it the next time the system is loaded. Moreover, it will also demolish the deadlock occured at that time and also it will take care that the same deadlock does not happen again. If the same sitaution happens again and again then there is no point in it and finally the system can be called a damaged one.

To determine the proper watchdog timeout duration, the programmer must determine the amount of time it takes to execute the code, using worst case scenarios. It is pretty obvious that one cannot know the timing in which the program will execute but on a safer side can always set the value of timeout to be a little higher so that the problem of deadlock does not occur.

This is very useful in health monitoring systems, rockets launch, satellite configurations where

the real time data is very critical.

Using pthread_mutex_timedlock code output



```
pi@raspberrypi:~ $ cd Download
bash: cd: Download: No such file or directory
pi@raspberrypi:~ $ cd Downloads
pi@raspberrypi:~/Downloads $ gcc -o Q-5 Q-5.c -lpthread -lrt
Q-5.c: In function ‘read1’:
Q-5.c:59:51: warning: passing argument 2 of ‘pthread_mutex_timedlock’ from incom
patible pointer type [-Wincompatible-pointer-types]
        a=  pthread_mutex_timedlock(&acceleration, &t1.tv_sec);
                                                   ^
In file included from Q-5.c:9:0:
/usr/include/pthread.h:767:12: note: expected ‘const struct timespec * restrict’
 but argument is of type ‘__time_t * {aka long int *}’
 extern int pthread_mutex_timedlock (pthread_mutex_t *__restrict __mutex,
            ^~~~~~~~~~~~~~~~~~~~~~~
pi@raspberrypi:~/Downloads $ sudo ./Q-5
Writing is started
        X coordinate is 6.000000 :
        Y coordinate is 11.000000 :
        Z coordinate is 16.000000 :
        pitch is 3.000000 :
        roll is 4.000000 :
        acceleration is 5.000000 :
        yaw is 4.000000 :
        time stamp reading in sec is 0
         time stamp reading nanosec 0
         No new data available at time 1531199657




 Reading is started
        X coordinate is 6.000000 :
        Y coordinate is 11.000000 :
        Z coordinate is 16.000000 :
        pitch is 3.000000 :
        roll is 4.000000 :
        acceleration is 5.000000 :
        yaw is 4.000000 :
        time stamp reading in sec is : 1531199646
         time stamp reading in nanosec is : 528568222




 Writing is started
        X coordinate is 7.000000 :
        Y coordinate is 12.000000 :
        Z coordinate is 17.000000 :
        pitch is 4.000000 :
        roll is 5.000000 :
        acceleration is 6.000000 :
        yaw is 5.000000 :
        time stamp reading in sec is 1531199646
         time stamp reading nanosec 1531199646
         No new data available at time 1531199668
```

References :-

1) http://mercury.pr.erau.edu/%7Esiewerts/cec450/documents/Papers/prio_inheritance_protocols.pdf

Priority Inheritance Protocols :Approach to a Real Time Synchronisation paper by Lui Sha, Raghunathan Rajkumar, John P Lehoczky

2)http://lwn.net/Articles/177111/

Inglo’s View

3) http://lwn.net/Articles/178253/

Linus’ View

4) https://en.wikipedia.org/wiki/POSIX

Futex Wikipedia Defination

5)https://stackoverflow.com/questions/856823/threadsafe-vs-re-entrant

Thread safe and Re entrant Definations

6)**https://vmlens.com/articles/techniques_for_thread_safety/**

7)http://doc.qt.io/archives/qt-4.8/threads-reentrancy.html

8) http://linux.die.net/man/3/pthread_mutex_lock

9) http://linux.die.net/man/3/clock_gettime

10)https://www.e-reading.club/chapter.php/102147/284/Li%2C_Yao_-_RealTime_Concepts_for_Embedded_Systems.html

Deadlock Information

12)https://en.wikipedia.org/wiki/Watchdog_timer

Watchdog Timer

13)

https://www.embedded.com/electronics-blogs/beginner-s-corner/4023849/Introduction-toWatchdog-Timers

14)

http://www.sat.dundee.ac.uk/psc/watchdog/watchdog-background.html