RTES Exercise 3

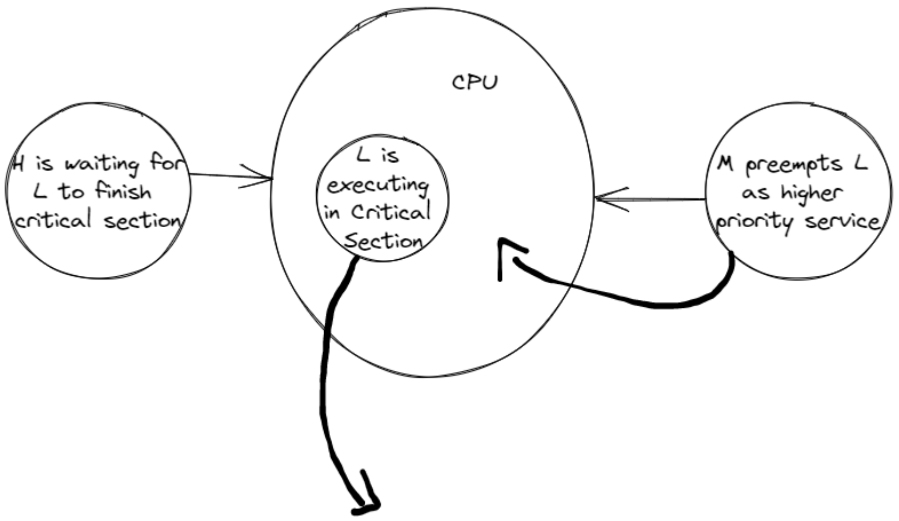
Dhiraj Bennadi
Jahnavi Pinnamaneni

March 2, 2022

Code Repository

Github Link

Question 1


CPU
H is waiting for L to finish critical section
L is executing in Critical Section
M preempts L as higher priority service

Priority Inversion Problem in case of 3 services of different priority H,M,L in descending order.

Summary of the paper titled "Priority Inheritance Protocols: An approach to Real-Time Synchronization

Basic Priority Inheritance Protocol

1. If a system has multiple services with priorities J in the order of $J_1 ... J_n$ and currently J_3 is executing in its critical section blocking a higher priority

service J_1.

2. The priority inheritance protocol requires J_3 to execute at a priority of J_1
3. This will result in J_2 unable to preempt J_3 and will itself be blocked
4. This effectively means that J_3 inherited the priority of J_1 solving the problem of indefinite wait by J_1 as a result of priority inversion
5. Once the J_3 completes its executions, its priority is restored

Blocking by a lower priority job can occur in 2 situations

1. Direct Blocking: Higher Priority try to lock a locked resource. This is needed for data consistency
2. Push-through Blocking: A high priority job can block a lower priority due to inheritance from highest priority

However 2 issues persist with the Priority Inheritance protocol:

1. Deadlock
2. Blocking Chain

Priority Ceiling Protocol

The goal of this protocol is to prevent the formation of deadlocks and of chained blocking. A job J is allowed only if the priority of J is higher than all priority ceilings of all the semaphores locked by jobs other than J

1. Job J will only be given to execute if it has the highest priority in the services and its priority is also higher than the priority ceiling of semaphore $S*$
2. Job J uses its assigned priority unless it is in critical section and blocks higher priority jobs. If job J blocks higher priority jobs, J inherits P_H. When J exits a critical section, it resumes the priority it had at the point of entry into the critical section
3. A job J when it does not attempt to enter a critical section, can preempt another job J_L if its priority is higher that the priority, inherited or assigned, at which job J_L is executing.

Highlights

1. Synchronization problem exits in the context of priority-driven preemptive scheduling
2. Priority Inheritance and Priority Ceiling Protocols solve the issue of priority inversion
3. Priority Ceiling Protocol prevents deadlocks and reduces the blocking to at most one critical section.

Position on the paper by Jonathan Corbet and Igno Molnar and Thomas Gleixner

I agree with Ingo Molnar and Thomas Gleixner, occurrence of priority inversion does not mean that the system is broken. The use of priority inheritance can help us write more deterministic codes for Realtime applications. As programmers, we would never want a situation like priority inheritance, but the requirements of the system can put us in a situation when the tasks are scheduled such that execution of higher priority task would in turn depend on the completion of lower priority task. This does not mean that we scrap the whole idea, we need to look for a work around to avoid non-deterministic behavior. Even in priority inversion, we don't mind bounded priority inversion since the results are deterministic. Hence, priority inheritance provides us the solution for achieving deterministic behavior and as long as the output of our real-time services are deterministic our system is not broken.

Position on the PI-futex

Futexes: The fundamental detail of the futex is the wait queue in the kernel which is associated with it. Waiters are enqueued, wakers dequeue threads. These operations are performed atomically. The operation to wait on a futex is composed of getting the lock for the futec, checking the current value, if necessary, adding the thread to the wait queue and releasing the lock. The guarantees that threads which go to sleep because the futex value is unchanged are going to be woken up if once the futex value is changed and threads are woken. Futexes like mutex allows at most one thread to own the mutex and if the mutex is free, either one or more threads try to lock it. One requirement of an inter-process synchronization primitive is that it is a) position independent and has no reference/pointers to any object in virtual address space. In futex this means that the wait queues need to be placed in the kernel. This code segment can be used by many processes on the same machine and they will use the same mutex, the kernel knows that all the virtual addresses are mapped to the same physical memory address and futexes are identified by their physical address. This offers a good method for synchronization.

Question 2

Re-entrant function:

A function is said to be re-entrant if a single copy of the routine can be called from many tasks simultaneously without conflict. Such conflict occurs due to the share of global or static variables because there is a single copy of the data and code. A re-entrant function can be interrupted in the middle of its execution and then safely be called again before its previous invocations' complete execution.

Thread-safe:

A thread safe code only manipulates shared data structures in a manner that ensures that all threads behave properly and fulfill their design specifications without unintended interaction. This property allows code to run in multi-threaded environments by re-establishing some of the correspondences between the actual flow of control and the text of the program, by means of synchronization.

Methods to achieve synchronization

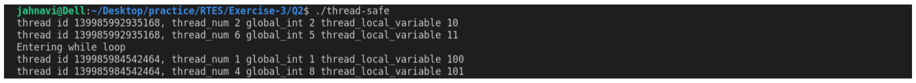
1. **Pure functions that use only stack and have no global memory:** This method is advantageous when number of variables is less. In real time applications, this method is very useful as it would avoid contention and data corruption. But this can be used only when the threads are independent of each other. But in most applications, threads often communicate with each other and share a common memory, in such cases only variables used for local computations can be stored on stack.

2. **Functions which use thread indexed global data:** This method is often useful when multithreading needs to be achieved. This method will allow each thread to have its own copy of the variable thus avoiding one thread from manipulating the data of another. Ex: errno

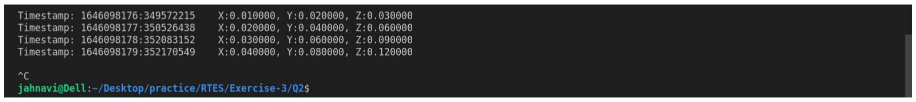
3. **Synchronization using MUTEX:** This method is useful when a shared memory needs to be accessed by multiple threads. But care needs to be taken when the number of mutexes used increases as it might lead to deadlocks.

The above three methods are illustrated in the code attached.
In the below screenshot we can see that the values of thread_num and global_int do not match despite both being global variables, the reason being that thread_num is thread_indexed.
The local variables update without any interference from the other threads.
Mutexes are demonstrated in the code while writing and reading from the temporary text file.

Output



```
jahnavi@Dell:~/Desktop/practice/RTES/Exercise-3/Q2$ ./thread-safe
thread id 139985992935168, thread_num 2 global_int 2 thread_local_variable 10
thread id 139985992935168, thread_num 6 global_int 5 thread_local_variable 11
Entering while loop
thread id 139985984542464, thread_num 1 global_int 1 thread_local_variable 100
thread id 139985984542464, thread_num 4 global_int 8 thread_local_variable 101
```



```
Timestamp: 1646098176:349572215    X:0.010000, Y:0.020000, Z:0.030000
Timestamp: 1646098177:350526438    X:0.020000, Y:0.040000, Z:0.060000
Timestamp: 1646098178:352083152    X:0.030000, Y:0.060000, Z:0.090000
Timestamp: 1646098179:352170549    X:0.040000, Y:0.080000, Z:0.120000

^C
jahnavi@Dell:~/Desktop/practice/RTES/Exercise-3/Q2$
```

Question 3

Deadlock

This is a condition that arises when two processes are waiting on the same shared resource to procedure with further execution but neither get access since each process is waiting for the other to release its hold on the resource.
In the code the root cause for deadlock is the mutex lock on the resources by the threads 1 and 2. The thread 1 tries to access resource A first but locking the mutex on it and then tries to get the resource B. Once it receives both, it releases the hold on resource B and then resource A. The thread 2 on the other hand tries to access resource B first and then tries to access resource A and releases them in the order A first then B.
When the code is executed, Thread 1 tries to access A and then holds it with mutex lock and then tries to access B, at this instant, Thread 2 would have already started execution and acquired B while it is waiting for A. Now thread 1 has hold on A and is waiting for B while thread 2 has hold on B and is waiting for A. Now both tasks cannot come out of this situation since both are waiting for a resource that is linked to the task itself releasing hold on the resource it has.
One way to avoid this is to access the resources in the same order from both the threads i.e., Thread 1 access A and then B. Thread 2 also tries to access A and then B. In this case, since both threads try to access the same resource, one of it will get the resource while the other is waiting. Once the thread finishes execution, the other can start and complete its execution.
Another way is to use random back off delays when a timeout occurs i.e., each tasks waits for a random amount of time before it tries to regain access over the resource, this will provide the other tasks to complete execution and release hold on the resource.

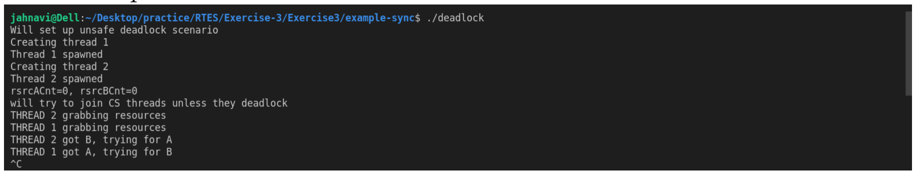
Unbounded Priority Inversion:

This is a situation where a high priority task cannot be executed due to a low priority task that is on hold. For example, let us consider 3 tasks A, B and C with decreasing priority i.e., A has highest priority and C has the lowest. Let us also consider that task A is dependent on Task C to release a mutex hold

that it would in return use.
Now, let's say task C gets preempted by task B that is of higher priority after task C locked a mutex, now the highest priority task A cannot execute unless task C releases that mutex and task C can execute only after task B completes execution. The time that task B would take is not deterministic hence, this is an unbounded condition, and the task A is waiting for task B to complete irrespective of their priority levels. This is the case of Unbounded priority inversion.
Yes, there is a fix in Linux for the condition of Unbounded inversion. Generally, the solution for Unbounded priority inversion is priority Inheritance. Priority Inheritance is where the priority of the task currently holding the mutex is boosted to the highest priority level of the tasks that use the mutex.
In the RT_PREEMPT_Patch, provides many API to schedule tasks for Real time applications. In the patch, even priority inheritance is implemented through futex. This implementation helps improving determinism for user-space applications. In the best-case, it can help achieve determinism and well-bound latencies. Even in the worst-case, PI will improve the statistical distribution of locking related application delays. This is help with Unbounded priority inversion.
Though Linux, provides a fix for priority inversion, if only inversion is considered it would make sense to do the project on RTOS instead of Linux since a RTOS is designed for this application specifically and can handle situations like priority inheritance in a more elegant fashion.

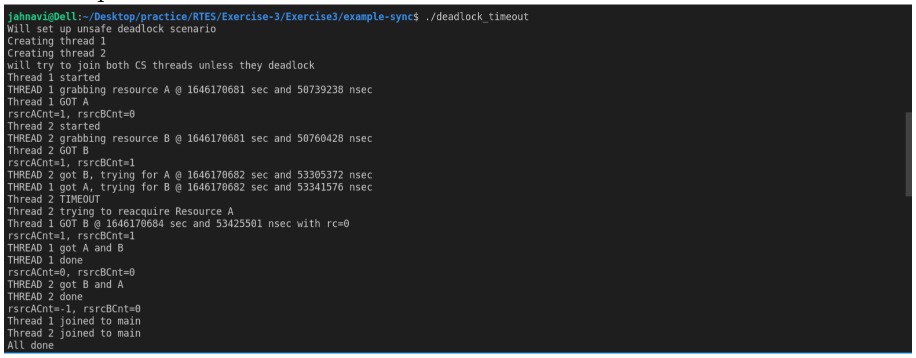
Output

This is output when deadlock.c is executed in unsafe mode.



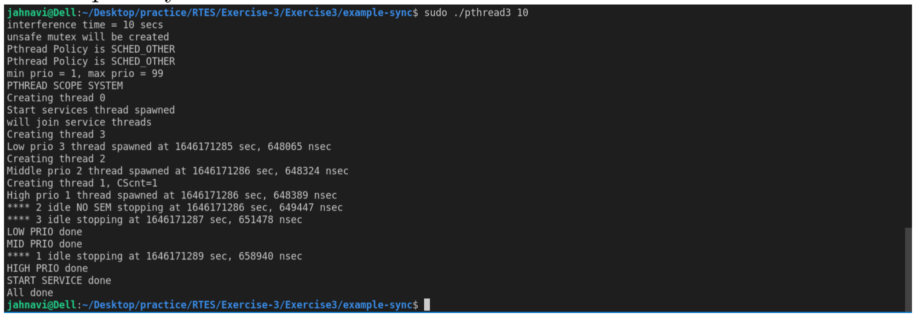
```
jahnavi@Dell:~/Desktop/practice/RTES/Exercise-3/Exercise3/example-sync$ ./deadlock
Will set up unsafe deadlock scenario
Creating thread 1
Thread 1 spawned
Creating thread 2
Thread 2 spawned
rsrcACnt=0, rsrcBCnt=0
will try to join CS threads unless they deadlock
THREAD 2 grabbing resources
THREAD 1 grabbing resources
THREAD 2 got B, trying for A
THREAD 1 got A, trying for B
^C
```

The output after random backoff scheme:



```
jahnavi@Dell:~/Desktop/practice/RTES/Exercise-3/Exercise3/example-sync$ ./deadlock_timeout
Will set up unsafe deadlock scenario
Creating thread 1
Creating thread 2
will try to join both CS threads unless they deadlock
Thread 1 started
THREAD 1 grabbing resource A @ 1646170681 sec and 50739238 nsec
Thread 1 GOT A
rsrcACnt=1, rsrcBCnt=0
Thread 2 started
THREAD 2 grabbing resource B @ 1646170681 sec and 50760428 nsec
Thread 2 GOT B
rsrcACnt=1, rsrcBCnt=1
THREAD 2 got B, trying for A @ 1646170682 sec and 53305372 nsec
THREAD 1 got A, trying for B @ 1646170682 sec and 53341576 nsec
Thread 2 TIMEOUT
Thread 2 trying to reacquire Resource A
Thread 1 GOT B @ 1646170684 sec and 53425501 nsec with rc=0
rsrcACnt=1, rsrcBCnt=1
THREAD 1 got A and B
THREAD 1 done
rsrcACnt=0, rsrcBCnt=0
THREAD 2 got B and A
THREAD 2 done
rsrcACnt=-1, rsrcBCnt=0
Thread 1 joined to main
Thread 2 joined to main
All done
```

The output for pthread3.c is as follows: Though thread with index 1 has the highest priority, it had to wait for the tasks with indices 2 and 3 to complete. This is priority inversion.



```
jahnavi@Dell:~/Desktop/practice/RTES/Exercise-3/Exercise3/example-sync$ sudo ./pthread3 10
interference time = 10 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 1646171285 sec, 648065 nsec
Creating thread 2
Middle prio 2 thread spawned at 1646171286 sec, 648324 nsec
Creating thread 1, CScnt=1
High prio 1 thread spawned at 1646171286 sec, 648389 nsec
**** 2 idle NO SEM stopping at 1646171286 sec, 649447 nsec
**** 3 idle stopping at 1646171287 sec, 651478 nsec
LOW PRIO done
MID PRIO done
**** 1 idle stopping at 1646171289 sec, 658940 nsec
HIGH PRIO done
START SERVICE done
All done
jahnavi@Dell:~/Desktop/practice/RTES/Exercise-3/Exercise3/example-sync$
```

Question 4

Message Queues

1. Message Queues are used to synchronize tasks and to pass data between them. The enqueue and dequeue operations are thread-safe (No data corruption)

2. The message queues can be read or written in blocking or non-blocking modes.

3. For blocking modes, when a task does a read on an empty message queue, it blocks the message until a message is enqueued by another task. When a task tries to write a full message queue in blocking mode, it is blocked until the queue has room for the new message.

4. For Non-Blocking message queues, when a task does a read on an empty queue, it will be returned an *ERROR_CODE* indicating no messages were available to read and the task should try again later. Similarly for writing to a full queue.

Priority Inversion

Priority Inversion is defined as the time that a high priority service has to wait while a lower priority service runs. For the duration which cannot be lumped into the response latency, it is known as unbounded priority inversion.

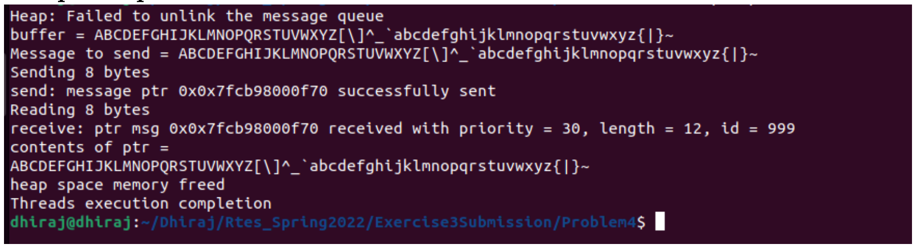
The issue of unbounded priority inversion can be overcome due to the following reasons:

1. Message queues are thread-safe which prevents preemptions in critical sections.

2. Message queues include a feature to enqueue messages with a higher priority level and to read the priority when the messages are dequeued. Higher -priority messages are always dequeued first. This essentially allows a sender to put an important message onto the head of queue.

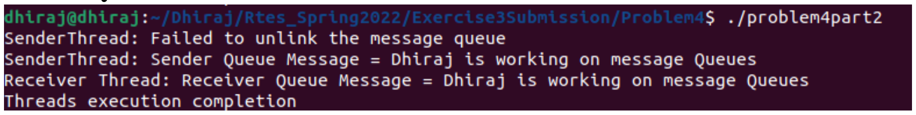
Output

Heap Output



```
Heap: Failed to unlink the message queue
buffer = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Sending 8 bytes
send: message ptr 0x0x7fcb98000f70 successfully sent
Reading 8 bytes
receive: ptr msg 0x0x7fcb98000f70 received with priority = 30, length = 12, id = 999
contents of ptr =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
heap space memory freed
Threads execution completion
dhiraj@dhiraj:~/Dhiraj/Rtes_Spring2022/Exercise3Submission/Problem4$
```

Posix Queue



```
dhiraj@dhiraj:~/Dhiraj/Rtes_Spring2022/Exercise3Submission/Problem4$ ./problem4part2
SenderThread: Failed to unlink the message queue
SenderThread: Sender Queue Message = Dhiraj is working on message Queues
Receiver Thread: Receiver Queue Message = Dhiraj is working on message Queues
Threads execution completion
```

Question 5

A watchdog in computer terms is something, usually hardware-based, that monitors a complex system for "normal" behaviour and if it fails, performs a system reset to hopefully recover normal operation.

In a typical software system, blocking can occur anytime a service can be dispatched by the CPU. If the duration of blocking is known, then its a bounded blocking else an unbounded blocking.
Three phenomena can cause unbounded blocking

1. Deadlock
2. Livelock
3. Unbounded Priority Inversion

The following situations might stop a watchdog being serviced:

1. Deadlock or livelock preventing the watchdog reset service from executing
2. Loss of software sanity due to programming errors, such as a bad pointer, an improperly handled processor exception, such as divide by zero or a bus error
3. Overload due to miscalculation of WCETs for the final approach service set
4. A hardware malfunction of the watchdog timer or associated circuitry
5. A multi-bit error in memory due to space radiation causing a bit upset

How a watchdog reset might recover from an indefinite deadlock

1. **Deadlock**

   (a) In the following figure, service S1 needs resources A and C, service S2 needs A and B and S3 needs B and C

   (b) If S1 acquires A, S2 acquires B, S3 acquires C a **Circular wait** situation arises causing infinite deadlock

   (c) Systems can be designed such that a keep-alive time can be posted for monitoring purpose and if the keep-alive is not posted due a a deadlock a **Supervisory service** can restart the deadlocked service.

2. **Livelock**

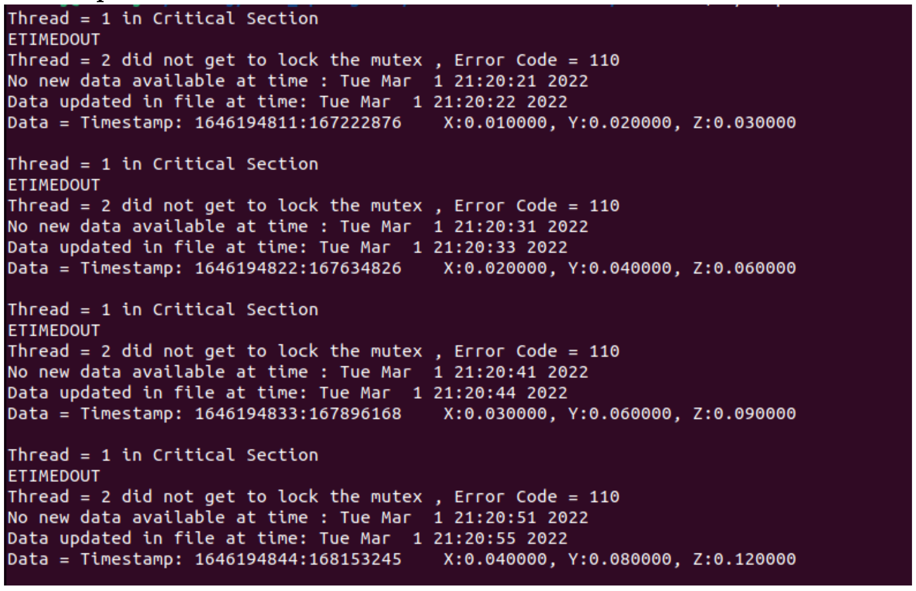
    (a) It is possible that when a deadlock is detected and the Supervisory service restarts the system, the services could again enter a deadlock over and over again. This is called a **Livelock**

    (b) One Solution is to include a random back-off time on restart.

    In situations where a deadlock and a livelock arise, the **Supervisory Service** usually recovers the system. However if a Supervisory Service is itself invovled in the deadlock then **Watchdog Timer** is needed to reset the entire system with a hardware reset

3. **Loss of software sanity**
    Accessing unrestricted memory, system exceptions, unhandled error cases in software might cause the system forcefully to enter into an unrecoverable state. The state might cause the system to wait for an indefinite time, failing to update the watchdog timer. In such situation, the watchdog resets the whole system.

Output



```
Thread = 1 in Critical Section
ETIMEDOUT
Thread = 2 did not get to lock the mutex , Error Code = 110
No new data available at time : Tue Mar  1 21:20:21 2022
Data updated in file at time: Tue Mar  1 21:20:22 2022
Data = Timestamp: 1646194811:167222876    X:0.010000, Y:0.020000, Z:0.030000

Thread = 1 in Critical Section
ETIMEDOUT
Thread = 2 did not get to lock the mutex , Error Code = 110
No new data available at time : Tue Mar  1 21:20:31 2022
Data updated in file at time: Tue Mar  1 21:20:33 2022
Data = Timestamp: 1646194822:167634826    X:0.020000, Y:0.040000, Z:0.060000

Thread = 1 in Critical Section
ETIMEDOUT
Thread = 2 did not get to lock the mutex , Error Code = 110
No new data available at time : Tue Mar  1 21:20:41 2022
Data updated in file at time: Tue Mar  1 21:20:44 2022
Data = Timestamp: 1646194833:167896168    X:0.030000, Y:0.060000, Z:0.090000

Thread = 1 in Critical Section
ETIMEDOUT
Thread = 2 did not get to lock the mutex , Error Code = 110
No new data available at time : Tue Mar  1 21:20:51 2022
Data updated in file at time: Tue Mar  1 21:20:55 2022
Data = Timestamp: 1646194844:168153245    X:0.040000, Y:0.080000, Z:0.120000
```

Source Code

Code for Problem 2

```
/*
@file: thread-safe.c
@desc: this code illustrates the different methods to achieve thread-safe code n
       1) use of local variables
       2) thread-indexed global variables
       3) mutexes
@author: Jahnavi Pinnamaneni; japi8358@colorado.edu
*/


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

#include <unistd.h>
#include <fcntl.h>
#include <errno.h>

#include <threads.h>
#include <pthread.h>

#include <time.h>

#define handle_err(err_no, arg) \
        do{errno = err_no; perror("arg"), exit(EXIT_FAILURE);}while(0);

//this is a thread_indexed global variable i.e., each thread would have a separa
static __thread int thread_num = 0;

int global_int = 0;

pthread_mutex_t mutex;

bool read_flag = false;

typedef struct thread_info{
    pthread_t thread_id;
}thread_info_t;


/*
```

```
The variable "thread_local_num" is initialised and updated to illustrate the usa
The variable "thread_num" is thread indexed, another global variable "global_int

This thread then writes the updates values of the timestamp and attitude status
data corruption.
*/
static void *thread_func_1(void * arg)
{
    thread_info_t *t_info = (thread_info_t *)arg;
    int thread_local_num = 10;
    thread_num = 2;
    global_int = 2;
    printf("thread␣id␣%ld,␣thread_num␣%d␣global_int␣%d␣thread_local_variable␣%d\

    thread_num += 4;
    global_int += 4;
    thread_local_num++;
    printf("thread␣id␣%ld,␣thread_num␣%d␣global_int␣%d␣thread_local_variable␣%d\

    typedef struct data{
        float X;
        float Y;
        float Z;
        struct timespec t_val;
    }data_t;

    printf("Entering␣while␣loop\n");
    data_t *att_data = calloc(1,sizeof(struct data));
    while(1)
    {
        att_data->X += 0.01;
        att_data->Y += 0.02;
        att_data->Z += 0.03;
        int clock_ret = clock_gettime(CLOCK_REALTIME,&att_data->t_val);

        if( att_data->Z>= 100.00)
        {
            att_data->X = 0.00;
            att_data->Y = 0.00;
            att_data->Z = 0.00;
        }

        char *str;
        asprintf(&str,"Timestamp:␣%d:%ld␣␣␣␣X:%f,␣Y:%f,␣Z:%f\n",att_data->t_val.
        pthread_mutex_lock(&mutex);
        int fd = open("/var/tmp/rtes",O_WRONLY|O_APPEND);
        if(fd < 0)
        handle_err(fd, "open");
        write(fd,str,strlen(str));
        close(fd);
        pthread_mutex_unlock(&mutex);
        read_flag = true;

        sleep(1);
    }

}

/*
This function reads the data from the file from the beginning and prints it out
*/
static void *thread_func_2(void * arg)
{
    thread_info_t *t_info = (thread_info_t *)arg;
    int thread_local_num = 100;
    thread_num = 1;
    global_int = 1;
    printf("thread␣id␣%ld,␣thread_num␣%d␣global_int␣%d␣thread_local_variable␣%d\

    thread_num += 3;
    global_int += 3;
    thread_local_num++;
    printf("thread␣id␣%ld,␣thread_num␣%d␣global_int␣%d␣thread_local_variable␣%d\

    while(1)
    {
        sleep(1);
        if(read_flag)
        {
            pthread_mutex_lock(&mutex);
            int fd = open("/var/tmp/rtes",O_RDONLY);
            if(fd < 0)
                handle_err(fd, "open");
            lseek(fd,0,SEEK_SET);
            int read_ret = 0;
            char s;
            do{
                read_ret = read(fd,&s,1);
                printf("%c",s);
            }while(read_ret);
            pthread_mutex_unlock(&mutex);
        }
        else
            continue;
    }

}

//Application entry point
/*
Two threads are created along with a mutex. These thread are then joined before
A temporary file is opened or a new file is created if it does not already exist
write operations.
*/
int main()
{
    int mut_ret = pthread_mutex_init(&mutex,NULL);
    if(mut_ret <0)
        handle_err(mut_ret,"mutex");

    int fd = open("/var/tmp/rtes",O_RDWR|O_CREAT|O_TRUNC, 0666);
    if(fd<0)
        handle_err(fd,"open");

    thread_info_t *t_info_1 = (thread_info_t *)calloc(1,sizeof(struct thread_inf
    int ret = pthread_create(&(t_info_1->thread_id),NULL, thread_func_1,t_info_1
    if(ret != 0)
        handle_err(ret, "pthread_create");

    thread_info_t *t_info_2 = (thread_info_t *)calloc(1,sizeof(struct thread_inf
    ret = pthread_create(&(t_info_2->thread_id),NULL, thread_func_2,t_info_2);
    if(ret != 0)
        handle_err(ret, "pthread_create");

    pthread_join(t_info_1->thread_id,NULL);
    free(t_info_1);

    pthread_join(t_info_2->thread_id,NULL);

    if(pthread_mutex_destroy(&mutex) != 0)
     perror("mutex destroy");

    free(t_info_2);
    close(fd);
    return 0;
}
```

Code for Problem 3 File: deadlock.c

```c
#include <pthread.h>
#include <stdio.h>
#include <sched.h>
#include <time.h>
#include <stdlib.h>

#define NUM_THREADS 2
#define THREAD_1 1
#define THREAD_2 2

typedef struct
{
    int threadIdx;
} threadParams_t;


pthread_t threads[NUM_THREADS];
threadParams_t threadParams[NUM_THREADS];

struct sched_param nrt_param;

pthread_mutex_t rsrcA, rsrcB;

volatile int rsrcACnt=0, rsrcBCnt=0, noWait=0;


void *grabRsrcs(void *threadp)
{
   threadParams_t *threadParams = (threadParams_t *)threadp;
   int threadIdx = threadParams->threadIdx;


   if(threadIdx == THREAD_1)
   {
     printf("THREAD 1 grabbing resources\n");
     pthread_mutex_lock(&rsrcA);
     rsrcACnt++;
     if(!noWait) usleep(1000000);
     printf("THREAD 1 got A, trying for B\n");
     pthread_mutex_lock(&rsrcB);
     rsrcBCnt++;
     printf("THREAD 1 got A and B\n");
     pthread_mutex_unlock(&rsrcB);
     pthread_mutex_unlock(&rsrcA);
     printf("THREAD␣1␣done\n");
   }
   else
   {
     printf("THREAD␣2␣grabbing␣resources\n");
     pthread_mutex_lock(&rsrcB);
     rsrcBCnt++;
     if(!noWait) usleep(1000000);
     printf("THREAD␣2␣got␣B,␣trying␣for␣A\n");
     pthread_mutex_lock(&rsrcA);
     rsrcACnt++;
     printf("THREAD␣2␣got␣B␣and␣A\n");
     pthread_mutex_unlock(&rsrcA);
     pthread_mutex_unlock(&rsrcB);
     printf("THREAD␣2␣done\n");
   }
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   int rc, safe=0;

   rsrcACnt=0, rsrcBCnt=0, noWait=0;

   if(argc < 2)
   {
     printf("Will␣set␣up␣unsafe␣deadlock␣scenario\n");
   }
   else if(argc == 2)
   {
     if(strncmp("safe", argv[1], 4) == 0)
       safe=1;
     else if(strncmp("race", argv[1], 4) == 0)
       noWait=1;
     else
       printf("Will␣set␣up␣unsafe␣deadlock␣scenario\n");
   }
   else
   {
     printf("Usage:␣deadlock␣[safe|race|unsafe]\n");
   }

   // Set default protocol for mutex
   pthread_mutex_init(&rsrcA, NULL);
   pthread_mutex_init(&rsrcB, NULL);
```

```
printf("Creating␣thread␣%d\n", THREAD_1);
threadParams[THREAD_1].threadIdx=THREAD_1;
rc = pthread_create(&threads[0], NULL, grabRsrcs, (void *)&threadParams[THREA
if (rc) {printf("ERROR;␣pthread_create()␣rc␣is␣%d\n", rc); perror(NULL); exit
printf("Thread␣1␣spawned\n");

if(safe) // Make sure Thread 1 finishes with both resources first
{
  if(pthread_join(threads[0], NULL) == 0)
    printf("Thread␣1:␣%x␣done\n", (unsigned int)threads[0]);
  else
    perror("Thread␣1");
}

printf("Creating␣thread␣%d\n", THREAD_2);
threadParams[THREAD_2].threadIdx=THREAD_2;
rc = pthread_create(&threads[1], NULL, grabRsrcs, (void *)&threadParams[THREA
if (rc) {printf("ERROR;␣pthread_create()␣rc␣is␣%d\n", rc); perror(NULL); exit
printf("Thread␣2␣spawned\n");

printf("rsrcACnt=%d,␣rsrcBCnt=%d\n", rsrcACnt, rsrcBCnt);
printf("will␣try␣to␣join␣CS␣threads␣unless␣they␣deadlock\n");

if(!safe)
{
  if(pthread_join(threads[0], NULL) == 0)
    printf("Thread␣1:␣%x␣done\n", (unsigned int)threads[0]);
  else
    perror("Thread␣1");
}

if(pthread_join(threads[1], NULL) == 0)
  printf("Thread␣2:␣%x␣done\n", (unsigned int)threads[1]);
else
  perror("Thread␣2");

if(pthread_mutex_destroy(&rsrcA) != 0)
  perror("mutex␣A␣destroy");

if(pthread_mutex_destroy(&rsrcB) != 0)
  perror("mutex␣B␣destroy");

printf("All␣done\n");

exit(0);
}
```

Code for Problem 3 File: deadlock$_{t}$*imeout.c*

```
#include <pthread.h>
#include <stdio.h>
#include <sched.h>
#include <time.h>
#include <stdlib.h>
#include <errno.h>

#define NUM_THREADS 2
#define THREAD_1 1
#define THREAD_2 2

typedef struct
{
    int threadIdx;
} threadParams_t;


threadParams_t threadParams[NUM_THREADS];
pthread_t threads[NUM_THREADS];
struct sched_param nrt_param;

pthread_mutex_t rsrcA, rsrcB;

volatile int rsrcACnt=0, rsrcBCnt=0, noWait=0;

int flag_A = 0;
int flag_B = 0;


void *grabRsrcs(void *threadp)
{
   struct timespec timeNow;
   struct timespec rsrcA_timeout;
   struct timespec rsrcB_timeout;
   int rc;
   threadParams_t *threadParams = (threadParams_t *)threadp;
   int threadIdx = threadParams->threadIdx;

   if(threadIdx == THREAD_1) printf("Thread␣1␣started\n");
   else if(threadIdx == THREAD_2) printf("Thread␣2␣started\n");
   else printf("Unknown␣thread␣started\n");

   clock_gettime(CLOCK_REALTIME, &timeNow);
```

```
rsrcA_timeout.tv_sec = timeNow.tv_sec + 2;
rsrcA_timeout.tv_nsec = timeNow.tv_nsec;
rsrcB_timeout.tv_sec = timeNow.tv_sec + 3;
rsrcB_timeout.tv_nsec = timeNow.tv_nsec;


if(threadIdx == THREAD_1)
{
  printf("THREAD␣1␣grabbing␣resource␣A␣@␣%d␣sec␣and␣%d␣nsec\n", (int)timeNow.
  //if((rc=pthread_mutex_timedlock(&rsrcA, &rsrcA_timeout)) != 0)
  if((rc=pthread_mutex_lock(&rsrcA)) != 0)
  {
      printf("Thread␣1␣ERROR\n");
      pthread_exit(NULL);
  }
  else
  {
      printf("Thread␣1␣GOT␣A\n");
      rsrcACnt++;
      printf("rsrcACnt=%d,␣rsrcBCnt=%d\n", rsrcACnt, rsrcBCnt);
  }

  // if unsafe test, immediately try to acquire rsrcB
  if(!noWait) usleep(1000000);

  clock_gettime(CLOCK_REALTIME, &timeNow);
  rsrcB_timeout.tv_sec = timeNow.tv_sec + 3;
  rsrcB_timeout.tv_nsec = timeNow.tv_nsec;

  printf("THREAD␣1␣got␣A,␣trying␣for␣B␣@␣%d␣sec␣and␣%d␣nsec\n", (int)timeNow.

  rc=pthread_mutex_timedlock(&rsrcB, &rsrcB_timeout);
  //rc=pthread_mutex_lock(&rsrcB);
  if(rc == 0)
  {
      clock_gettime(CLOCK_REALTIME, &timeNow);
      printf("Thread␣1␣GOT␣B␣@␣%d␣sec␣and␣%d␣nsec␣with␣rc=%d\n", (int)timeNow
      rsrcBCnt++;
      printf("rsrcACnt=%d,␣rsrcBCnt=%d\n", rsrcACnt, rsrcBCnt);
  }
  else if(rc == ETIMEDOUT)
  {
      printf("Thread␣1␣TIMEOUT\n");
      rsrcACnt--;
      pthread_mutex_unlock(&rsrcA);
      flag_A = 1;
```

```
        printf("Thread␣1␣trying␣to␣reacquire␣Resource␣B\n");
        clock_gettime(CLOCK_REALTIME, &timeNow);
         rsrcB_timeout.tv_sec = timeNow.tv_sec + (rand()%10);
         rsrcB_timeout.tv_nsec = timeNow.tv_nsec;
         int ret_B = pthread_mutex_timedlock(&rsrcB, &rsrcB_timeout);
        //pthread_exit(NULL);
    }
    else
    {
        printf("Thread␣1␣ERROR\n");
        rsrcACnt--;
        pthread_mutex_unlock(&rsrcA);
        pthread_exit(NULL);
    }

    printf("THREAD␣1␣got␣A␣and␣B\n");
    rsrcBCnt--;
    pthread_mutex_unlock(&rsrcB);
    if(flag_A == 0){
    rsrcACnt--;
    pthread_mutex_unlock(&rsrcA);}
    printf("THREAD␣1␣done\n");
    printf("rsrcACnt=%d,␣rsrcBCnt=%d\n", rsrcACnt, rsrcBCnt);
}

else
{
    printf("THREAD␣2␣grabbing␣resource␣B␣@␣%d␣sec␣and␣%d␣nsec\n", (int)timeNow.
    //if((rc=pthread_mutex_timedlock(&rsrcB, &rsrcB_timeout)) != 0)
    if((rc=pthread_mutex_lock(&rsrcB)) != 0)
    {
        printf("Thread␣2␣ERROR\n");
        pthread_exit(NULL);
    }
    else
    {
        printf("Thread␣2␣GOT␣B\n");
        rsrcBCnt++;
        printf("rsrcACnt=%d,␣rsrcBCnt=%d\n", rsrcACnt, rsrcBCnt);
    }

    // if unsafe test, immediately try to acquire rsrcB
    if(!noWait) usleep(1000000);

    clock_gettime(CLOCK_REALTIME, &timeNow);
    rsrcA_timeout.tv_sec = timeNow.tv_sec + 2;
```

```
        rsrcA_timeout.tv_nsec = timeNow.tv_nsec;

        printf("THREAD 2 got B, trying for A @ %d sec and %d nsec\n", (int)timeNow.
        rc=pthread_mutex_timedlock(&rsrcA, &rsrcA_timeout);
        //rc=pthread_mutex_lock(&rsrcA);
        if(rc == 0)
        {
            clock_gettime(CLOCK_REALTIME, &timeNow);
            printf("Thread 2 GOT A @ %d sec and %d nsec with rc=%d\n", (int)timeNow
            rsrcACnt++;
            printf("rsrcACnt=%d, rsrcBCnt=%d\n", rsrcACnt, rsrcBCnt);
        }
        else if(rc == ETIMEDOUT)
        {
            printf("Thread 2 TIMEOUT\n");
            rsrcBCnt--;
            pthread_mutex_unlock(&rsrcB);
            flag_B = 1;
            printf("Thread 2 trying to reacquire Resource A\n");
           clock_gettime(CLOCK_REALTIME, &timeNow);
           rsrcA_timeout.tv_sec = timeNow.tv_sec + (rand()%20);
           rsrcA_timeout.tv_nsec = timeNow.tv_nsec;
           int ret_A = pthread_mutex_timedlock(&rsrcA, &rsrcA_timeout);
            //pthread_exit(NULL);
        }
        else
        {
            printf("Thread 2 ERROR\n");
            rsrcBCnt--;
            pthread_mutex_unlock(&rsrcB);
            pthread_exit(NULL);
        }

        printf("THREAD 2 got B and A\n");
        rsrcACnt--;
        pthread_mutex_unlock(&rsrcA);
        if(flag_B == 0){
        rsrcBCnt--;
        pthread_mutex_unlock(&rsrcB);}
        printf("THREAD 2 done\n");
        printf("rsrcACnt=%d, rsrcBCnt=%d\n", rsrcACnt, rsrcBCnt);
    }
    pthread_exit(NULL);
}

int main (int argc, char *argv[])
```

```
{
   time_t  t;

   srand((unsigned)time(&t));
   int rc, safe=0;

   rsrcACnt=0, rsrcBCnt=0, noWait=0;

   if(argc < 2)
   {
     printf("Will set up unsafe deadlock scenario\n");
   }
   else if(argc == 2)
   {
     if(strncmp("safe", argv[1], 4) == 0)
       safe=1;
     else if(strncmp("race", argv[1], 4) == 0)
       noWait=1;
     else
       printf("Will set up unsafe deadlock scenario\n");
   }
   else
   {
     printf("Usage: deadlock [safe|race|unsafe]\n");
   }

   // Set default protocol for mutex which is unlocked to start
   pthread_mutex_init(&rsrcA, NULL);
   pthread_mutex_init(&rsrcB, NULL);

   printf("Creating thread %d\n", THREAD_1);
   threadParams[THREAD_1].threadIdx=THREAD_1;
   rc = pthread_create(&threads[0], NULL, grabRsrcs, (void *)&threadParams[THREA
   if (rc) {printf("ERROR; pthread_create() rc is %d\n", rc); perror(NULL); exit

   if(safe) // Make sure Thread 1 finishes with both resources first
   {
     if(pthread_join(threads[0], NULL) == 0)
       printf("Thread 1 joined to main\n");
     else
       perror("Thread 1");
   }

   printf("Creating thread %d\n", THREAD_2);
   threadParams[THREAD_2].threadIdx=THREAD_2;
   rc = pthread_create(&threads[1], NULL, grabRsrcs, (void *)&threadParams[THREA
    if (rc) {printf("ERROR;␣pthread_create()␣rc␣is␣%d\n", rc); perror(NULL); exit

    printf("will␣try␣to␣join␣both␣CS␣threads␣unless␣they␣deadlock\n");

    if(!safe)
    {
      if(pthread_join(threads[0], NULL) == 0)
        printf("Thread␣1␣joined␣to␣main\n");
      else
        perror("Thread␣1");
    }

    if(pthread_join(threads[1], NULL) == 0)
      printf("Thread␣2␣joined␣to␣main\n");
    else
      perror("Thread␣2");

    if(pthread_mutex_destroy(&rsrcA) != 0)
      perror("mutex␣A␣destroy");

    if(pthread_mutex_destroy(&rsrcB) != 0)
      perror("mutex␣B␣destroy");

    printf("All␣done\n");

    exit(0);
}
```

Code for Problem 3 File: pthread3.c

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

#define NUM_THREADS             4
#define START_SERVICE           0
#define HIGH_PRIO_SERVICE       1
#define MID_PRIO_SERVICE        2
#define LOW_PRIO_SERVICE        3
#define NUM_MSGS                3

pthread_t threads[NUM_THREADS];
pthread_attr_t rt_sched_attr[NUM_THREADS];
int rt_max_prio, rt_min_prio;
struct sched_param rt_param[NUM_THREADS];
struct sched_param nrt_param;

typedef struct
{
    int threadIdx;
} threadParams_t;


threadParams_t threadParams[NUM_THREADS];

pthread_mutex_t msgSem;
pthread_mutexattr_t rt_safe;

int rt_protocol;

volatile int runInterference=0, CScount=0;
volatile unsigned long long idleCount[NUM_THREADS];
int intfTime=0;


void *startService(void *threadid);

unsigned const int seqIterations = 47;
unsigned const int Iterations = 1000;

#define FIB_TEST(seqCnt, iterCnt)         \
   for(idx=0; idx < iterCnt; idx++)       \
  {                                    \
     fib = fib0 + fib1;                \
     while(jdx < seqCnt)               \
     {                                 \
        fib0 = fib1;                   \
        fib1 = fib;                    \
        fib = fib0 + fib1;             \
        jdx++;                         \
     }                                 \
     jdx=0;                            \
  }                                    \


void *idleNoSem(void *threadp)
{
  struct timespec timeNow;
  unsigned int idx = 0, jdx = 1;
  volatile unsigned int fib = 0, fib0 = 0, fib1 = 1;
  threadParams_t *threadParams = (threadParams_t *)threadp;
  int idleIdx = threadParams->threadIdx;

  do
  {
    FIB_TEST(seqIterations, Iterations);
    idleCount[idleIdx]++;
  } while(idleCount[idleIdx] < runInterference);

  gettimeofday(&timeNow, (void *)0);
  printf("****␣%d␣idle␣NO␣SEM␣stopping␣at␣%d␣sec,␣%d␣nsec\n", idleIdx, (int)timeN

  pthread_exit(NULL);
}


int CScnt=0;

void *idle(void *threadp)
{
  struct timespec timeNow;
  unsigned int idx = 0, jdx = 1;
  volatile unsigned int fib = 0, fib0 = 0, fib1 = 1;
  threadParams_t *threadParams = (threadParams_t *)threadp;
  int idleIdx = threadParams->threadIdx;

  pthread_mutex_lock(&msgSem);
  CScnt++;
  do
  {
    FIB_TEST(seqIterations, Iterations);
    idleCount[idleIdx]++;
  } while(idleCount[idleIdx] < runInterference);

  sleep(2);

  idleCount[idleIdx]=0;

  do
  {
    FIB_TEST(seqIterations, Iterations);
    idleCount[idleIdx]++;
  } while(idleCount[idleIdx] < runInterference);

  pthread_mutex_unlock(&msgSem);

  gettimeofday(&timeNow, (void *)0);
  printf("**** %d idle stopping at %d sec, %d nsec\n", idleIdx, (int)timeNow.tv_

  pthread_exit(NULL);
}


void print_scheduler(void)
{
   int schedType;

   schedType = sched_getscheduler(getpid());

   switch(schedType)
   {
     case SCHED_FIFO:
            printf("Pthread Policy is SCHED_FIFO\n");
            break;
     case SCHED_OTHER:
            printf("Pthread Policy is SCHED_OTHER\n");
        break;
     case SCHED_RR:
            printf("Pthread Policy is SCHED_OTHER\n");
            break;
     default:
        printf("Pthread Policy is UNKNOWN\n");
   }
```

```
}

int main (int argc, char *argv[])
{
   int rc, invSafe=0, i, scope;
   struct timespec sleepTime, dTime;

   CScount=0;

   if(argc < 2)
   {
     printf("Usage: pthread interfere-seconds\n");
     exit(-1);
   }
   else if(argc >= 2)
   {
     sscanf(argv[1], "%d", &intfTime);
     printf("interference time = %d secs\n", intfTime);
     printf("unsafe mutex will be created\n");
   }

   print_scheduler();

   pthread_attr_init(&rt_sched_attr[START_SERVICE]);
   pthread_attr_setinheritsched(&rt_sched_attr[START_SERVICE], PTHREAD_EXPLICIT_S
   pthread_attr_setschedpolicy(&rt_sched_attr[START_SERVICE], SCHED_FIFO);

   pthread_attr_init(&rt_sched_attr[HIGH_PRIO_SERVICE]);
   pthread_attr_setinheritsched(&rt_sched_attr[HIGH_PRIO_SERVICE], PTHREAD_EXPLI
   pthread_attr_setschedpolicy(&rt_sched_attr[HIGH_PRIO_SERVICE], SCHED_FIFO);

   pthread_attr_init(&rt_sched_attr[MID_PRIO_SERVICE]);
   pthread_attr_setinheritsched(&rt_sched_attr[MID_PRIO_SERVICE], PTHREAD_EXPLIC
   pthread_attr_setschedpolicy(&rt_sched_attr[MID_PRIO_SERVICE], SCHED_FIFO);

   pthread_attr_init(&rt_sched_attr[LOW_PRIO_SERVICE]);
   pthread_attr_setinheritsched(&rt_sched_attr[LOW_PRIO_SERVICE], PTHREAD_EXPLIC
   pthread_attr_setschedpolicy(&rt_sched_attr[LOW_PRIO_SERVICE], SCHED_FIFO);

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

   rc=sched_getparam(getpid(), &nrt_param);

   if (rc)
   {
       printf("ERROR;␣sched_setscheduler␣rc␣is␣%d\n", rc);
       perror(NULL);
       exit(-1);
   }

   print_scheduler();

   printf("min␣prio␣=␣%d,␣max␣prio␣=␣%d\n", rt_min_prio, rt_max_prio);
   pthread_attr_getscope(&rt_sched_attr[START_SERVICE], &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");

   pthread_mutex_init(&msgSem, NULL);

   rt_param[START_SERVICE].sched_priority = rt_max_prio;
   pthread_attr_setschedparam(&rt_sched_attr[START_SERVICE], &rt_param[START_SER

   printf("Creating␣thread␣%d\n", START_SERVICE);
   threadParams[START_SERVICE].threadIdx=START_SERVICE;
   rc = pthread_create(&threads[START_SERVICE], &rt_sched_attr[START_SERVICE], s

   if (rc)
   {
       printf("ERROR;␣pthread_create()␣rc␣is␣%d\n", rc);
       perror(NULL);
       exit(-1);
   }
   printf("Start␣services␣thread␣spawned\n");


   printf("will␣join␣service␣threads\n");

   if(pthread_join(threads[START_SERVICE], NULL) == 0)
     printf("START␣SERVICE␣done\n");
   else
     perror("START␣SERVICE");


   rc=sched_setscheduler(getpid(), SCHED_OTHER, &nrt_param);

   if(pthread_mutex_destroy(&msgSem) != 0)
```

```
        perror("mutex␣destroy");

    printf("All␣done\n");

    exit(0);
}


void *startService(void *threadid)
{
    struct timespec timeNow;
    int rc;

    runInterference=intfTime;

    rt_param[LOW_PRIO_SERVICE].sched_priority = rt_max_prio-20;
    pthread_attr_setschedparam(&rt_sched_attr[LOW_PRIO_SERVICE], &rt_param[LOW_PI

    printf("Creating␣thread␣%d\n", LOW_PRIO_SERVICE);
    threadParams[LOW_PRIO_SERVICE].threadIdx=LOW_PRIO_SERVICE;
    rc = pthread_create(&threads[LOW_PRIO_SERVICE], &rt_sched_attr[LOW_PRIO_SERVI

    if (rc)
    {
        printf("ERROR;␣pthread_create()␣rc␣is␣%d\n", rc);
        perror(NULL);
        exit(-1);
    }
    //pthread_detach(threads[LOW_PRIO_SERVICE]);
    gettimeofday(&timeNow, (void *)0);
    printf("Low␣prio␣%d␣thread␣spawned␣at␣%d␣sec,␣%d␣nsec\n", LOW_PRIO_SERVICE, (


    sleep(1);

    rt_param[MID_PRIO_SERVICE].sched_priority = rt_max_prio-10;
    pthread_attr_setschedparam(&rt_sched_attr[MID_PRIO_SERVICE], &rt_param[MID_PR

    printf("Creating␣thread␣%d\n", MID_PRIO_SERVICE);
    threadParams[MID_PRIO_SERVICE].threadIdx=MID_PRIO_SERVICE;
    rc = pthread_create(&threads[MID_PRIO_SERVICE], &rt_sched_attr[MID_PRIO_SERVI

    if (rc)
    {
        printf("ERROR;␣pthread_create()␣rc␣is␣%d\n", rc);
        perror(NULL);
        exit(-1);
    }
    //pthread_detach(threads[MID_PRIO_SERVICE]);
    gettimeofday(&timeNow, (void *)0);
    printf("Middle␣prio␣%d␣thread␣spawned␣at␣%d␣sec,␣%d␣nsec\n", MID_PRIO_SERVICE

    rt_param[HIGH_PRIO_SERVICE].sched_priority = rt_max_prio-1;
    pthread_attr_setschedparam(&rt_sched_attr[HIGH_PRIO_SERVICE], &rt_param[HIGH_


    printf("Creating␣thread␣%d,␣CScnt=%d\n", HIGH_PRIO_SERVICE, CScnt);
    threadParams[HIGH_PRIO_SERVICE].threadIdx=HIGH_PRIO_SERVICE;
    rc = pthread_create(&threads[HIGH_PRIO_SERVICE], &rt_sched_attr[HIGH_PRIO_SER

    if (rc)
    {
        printf("ERROR;␣pthread_create()␣rc␣is␣%d\n", rc);
        perror(NULL);
        exit(-1);
    }
    //pthread_detach(threads[HIGH_PRIO_SERVICE]);
    gettimeofday(&timeNow, (void *)0);
    printf("High␣prio␣%d␣thread␣spawned␣at␣%d␣sec,␣%d␣nsec\n", HIGH_PRIO_SERVICE,




    if(pthread_join(threads[LOW_PRIO_SERVICE], NULL) == 0)
      printf("LOW␣PRIO␣done\n");
    else
      perror("LOW␣PRIO");

    if(pthread_join(threads[MID_PRIO_SERVICE], NULL) == 0)
      printf("MID␣PRIO␣done\n");
    else
      perror("MID␣PRIO");

    if(pthread_join(threads[HIGH_PRIO_SERVICE], NULL) == 0)
      printf("HIGH␣PRIO␣done\n");
    else
      perror("HIGH␣PRIO");


    pthread_exit(NULL);
```

```
}
```

Code for Problem 3 File: pthread3amp.c

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

#define NUM_THREADS              4
#define START_SERVICE            0
#define HIGH_PRIO_SERVICE        1
#define MID_PRIO_SERVICE         2
#define LOW_PRIO_SERVICE         3
#define NUM_MSGS                 3

pthread_t threads[NUM_THREADS];
pthread_attr_t rt_sched_attr[NUM_THREADS];
int rt_max_prio, rt_min_prio;
struct sched_param rt_param[NUM_THREADS];
struct sched_param nrt_param;

typedef struct
{
    int threadIdx;
} threadParams_t;


threadParams_t threadParams[NUM_THREADS];

pthread_mutex_t msgSem;
pthread_mutexattr_t rt_safe;

int rt_protocol;

volatile int runInterference=0, CScount=0;
volatile unsigned long long idleCount[NUM_THREADS];
int intfTime=0;


void *startService(void *threadid);

unsigned const int seqIterations = 47;
unsigned const int Iterations = 1000;

#define FIB_TEST(seqCnt, iterCnt)       \
   for(idx=0; idx < iterCnt; idx++)     \
    {                                  \
      fib = fib0 + fib1;               \
      while(jdx < seqCnt)              \
      {                                \
        fib0 = fib1;                   \
        fib1 = fib;                    \
        fib = fib0 + fib1;             \
        jdx++;                         \
      }                                \
      jdx=0;                           \
    }                                  \


void *idleNoSem(void *threadp)
{
  struct timespec timeNow;
  unsigned int idx = 0, jdx = 1;
  volatile unsigned int fib = 0, fib0 = 0, fib1 = 1;
  threadParams_t *threadParams = (threadParams_t *)threadp;
  int idleIdx = threadParams->threadIdx;

  do
  {
    FIB_TEST(seqIterations, Iterations);
    idleCount[idleIdx]++;
  } while(idleCount[idleIdx] < runInterference);

  gettimeofday(&timeNow, (void *)0);
  printf("****␣%d␣idle␣NO␣SEM␣stopping␣at␣%d␣sec,␣%d␣nsec\n", idleIdx, (int)timeN

  pthread_exit(NULL);
}


int CScnt=0;

void *idle(void *threadp)
{
  struct timespec timeNow;
  unsigned int idx = 0, jdx = 1;
  volatile unsigned int fib = 0, fib0 = 0, fib1 = 1;
  threadParams_t *threadParams = (threadParams_t *)threadp;
  int idleIdx = threadParams->threadIdx;

  pthread_mutex_lock(&msgSem);
  CScnt++;
  do
  {
    FIB_TEST(seqIterations, Iterations);
    idleCount[idleIdx]++;
  } while(idleCount[idleIdx] < runInterference);

  sleep(2);

  idleCount[idleIdx]=0;

  do
  {
    FIB_TEST(seqIterations, Iterations);
    idleCount[idleIdx]++;
  } while(idleCount[idleIdx] < runInterference);

  pthread_mutex_unlock(&msgSem);

  gettimeofday(&timeNow, (void *)0);
  printf("****␣%d␣idle␣stopping␣at␣%d␣sec,␣%d␣nsec\n", idleIdx, (int)timeNow.tv_

  pthread_exit(NULL);
}


void print_scheduler(void)
{
   int schedType;

   schedType = sched_getscheduler(getpid());

   switch(schedType)
   {
     case SCHED_FIFO:
           printf("Pthread␣Policy␣is␣SCHED_FIFO\n");
           break;
     case SCHED_OTHER:
           printf("Pthread␣Policy␣is␣SCHED_OTHER\n");
       break;
     case SCHED_RR:
           printf("Pthread␣Policy␣is␣SCHED_OTHER\n");
           break;
     default:
       printf("Pthread␣Policy␣is␣UNKNOWN\n");
   }
```

```
}

int main (int argc, char *argv[])
{
   int rc, invSafe=0, i, scope;
   struct timespec sleepTime, dTime;

   CScount=0;

   if(argc < 2)
   {
     printf("Usage:␣pthread␣interfere-seconds\n");
     exit(-1);
   }
   else if(argc >= 2)
   {
     sscanf(argv[1], "%d", &intfTime);
     printf("interference␣time␣=␣%d␣secs\n", intfTime);
     printf("unsafe␣mutex␣will␣be␣created\n");
   }

   print_scheduler();

   pthread_attr_init(&rt_sched_attr[START_SERVICE]);
   pthread_attr_setinheritsched(&rt_sched_attr[START_SERVICE], PTHREAD_EXPLICIT_S
   pthread_attr_setschedpolicy(&rt_sched_attr[START_SERVICE], SCHED_FIFO);

   pthread_attr_init(&rt_sched_attr[HIGH_PRIO_SERVICE]);
   pthread_attr_setinheritsched(&rt_sched_attr[HIGH_PRIO_SERVICE], PTHREAD_EXPLI
   pthread_attr_setschedpolicy(&rt_sched_attr[HIGH_PRIO_SERVICE], SCHED_FIFO);

   pthread_attr_init(&rt_sched_attr[MID_PRIO_SERVICE]);
   pthread_attr_setinheritsched(&rt_sched_attr[MID_PRIO_SERVICE], PTHREAD_EXPLIC
   pthread_attr_setschedpolicy(&rt_sched_attr[MID_PRIO_SERVICE], SCHED_FIFO);

   pthread_attr_init(&rt_sched_attr[LOW_PRIO_SERVICE]);
   pthread_attr_setinheritsched(&rt_sched_attr[LOW_PRIO_SERVICE], PTHREAD_EXPLIC
   pthread_attr_setschedpolicy(&rt_sched_attr[LOW_PRIO_SERVICE], SCHED_FIFO);

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

   rc=sched_getparam(getpid(), &nrt_param);

   if (rc)
   {
       printf("ERROR;␣sched_setscheduler␣rc␣is␣%d\n", rc);
       perror(NULL);
       exit(-1);
   }

   print_scheduler();

   printf("min␣prio␣=␣%d,␣max␣prio␣=␣%d\n", rt_min_prio, rt_max_prio);
   pthread_attr_getscope(&rt_sched_attr[START_SERVICE], &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");

   pthread_mutex_init(&msgSem, NULL);

   rt_param[START_SERVICE].sched_priority = rt_max_prio;
   pthread_attr_setschedparam(&rt_sched_attr[START_SERVICE], &rt_param[START_SER

   printf("Creating␣thread␣%d\n", START_SERVICE);
   threadParams[START_SERVICE].threadIdx=START_SERVICE;
   rc = pthread_create(&threads[START_SERVICE], &rt_sched_attr[START_SERVICE], s

   if (rc)
   {
       printf("ERROR;␣pthread_create()␣rc␣is␣%d\n", rc);
       perror(NULL);
       exit(-1);
   }
   printf("Start␣services␣thread␣spawned\n");


   printf("will␣join␣service␣threads\n");

   if(pthread_join(threads[START_SERVICE], NULL) == 0)
     printf("START␣SERVICE␣done\n");
   else
     perror("START␣SERVICE");


   rc=sched_setscheduler(getpid(), SCHED_OTHER, &nrt_param);

   if(pthread_mutex_destroy(&msgSem) != 0)
```

```
        perror("mutex␣destroy");

    printf("All␣done\n");

    exit(0);
}


void *startService(void *threadid)
{
    struct timespec timeNow;
    int rc;

    runInterference=intfTime;

    rt_param[LOW_PRIO_SERVICE].sched_priority = rt_max_prio-20;
    pthread_attr_setschedparam(&rt_sched_attr[LOW_PRIO_SERVICE], &rt_param[LOW_PI

    printf("Creating␣thread␣%d\n", LOW_PRIO_SERVICE);
    threadParams[LOW_PRIO_SERVICE].threadIdx=LOW_PRIO_SERVICE;
    rc = pthread_create(&threads[LOW_PRIO_SERVICE], &rt_sched_attr[LOW_PRIO_SERVI

    if (rc)
    {
        printf("ERROR;␣pthread_create()␣rc␣is␣%d\n", rc);
        perror(NULL);
        exit(-1);
    }
    //pthread_detach(threads[LOW_PRIO_SERVICE]);
    gettimeofday(&timeNow, (void *)0);
    printf("Low␣prio␣%d␣thread␣spawned␣at␣%d␣sec,␣%d␣nsec\n", LOW_PRIO_SERVICE, (


    sleep(1);

    rt_param[MID_PRIO_SERVICE].sched_priority = rt_max_prio-10;
    pthread_attr_setschedparam(&rt_sched_attr[MID_PRIO_SERVICE], &rt_param[MID_PR

    printf("Creating␣thread␣%d\n", MID_PRIO_SERVICE);
    threadParams[MID_PRIO_SERVICE].threadIdx=MID_PRIO_SERVICE;
    rc = pthread_create(&threads[MID_PRIO_SERVICE], &rt_sched_attr[MID_PRIO_SERVI

    if (rc)
    {
        printf("ERROR;␣pthread_create()␣rc␣is␣%d\n", rc);
        perror(NULL);
        exit(-1);
    }
    //pthread_detach(threads[MID_PRIO_SERVICE]);
    gettimeofday(&timeNow, (void *)0);
    printf("Middle␣prio␣%d␣thread␣spawned␣at␣%d␣sec,␣%d␣nsec\n", MID_PRIO_SERVICE

    rt_param[HIGH_PRIO_SERVICE].sched_priority = rt_max_prio-1;
    pthread_attr_setschedparam(&rt_sched_attr[HIGH_PRIO_SERVICE], &rt_param[HIGH_


    printf("Creating␣thread␣%d,␣CScnt=%d\n", HIGH_PRIO_SERVICE, CScnt);
    threadParams[HIGH_PRIO_SERVICE].threadIdx=HIGH_PRIO_SERVICE;
    rc = pthread_create(&threads[HIGH_PRIO_SERVICE], &rt_sched_attr[HIGH_PRIO_SER

    if (rc)
    {
        printf("ERROR;␣pthread_create()␣rc␣is␣%d\n", rc);
        perror(NULL);
        exit(-1);
    }
    //pthread_detach(threads[HIGH_PRIO_SERVICE]);
    gettimeofday(&timeNow, (void *)0);
    printf("High␣prio␣%d␣thread␣spawned␣at␣%d␣sec,␣%d␣nsec\n", HIGH_PRIO_SERVICE,





    if(pthread_join(threads[LOW_PRIO_SERVICE], NULL) == 0)
      printf("LOW␣PRIO␣done\n");
    else
      perror("LOW␣PRIO");

    if(pthread_join(threads[MID_PRIO_SERVICE], NULL) == 0)
      printf("MID␣PRIO␣done\n");
    else
      perror("MID␣PRIO");

    if(pthread_join(threads[HIGH_PRIO_SERVICE], NULL) == 0)
      printf("HIGH␣PRIO␣done\n");
    else
      perror("HIGH␣PRIO");


    pthread_exit(NULL);
```

```
}
```

Code for Problem 3 File: pthread3ok.c

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

#define NUM_THREADS             4
#define START_SERVICE           0
#define HIGH_PRIO_SERVICE       1
#define MID_PRIO_SERVICE        2
#define LOW_PRIO_SERVICE        3
#define NUM_MSGS                3

pthread_t threads[NUM_THREADS];
pthread_attr_t rt_sched_attr[NUM_THREADS];
int rt_max_prio, rt_min_prio;
struct sched_param rt_param[NUM_THREADS];
struct sched_param nrt_param;

typedef struct
{
    int threadIdx;
} threadParams_t;


threadParams_t threadParams[NUM_THREADS];

int rt_protocol;

volatile int runInterference=0, CScount=0;
volatile unsigned long long idleCount[NUM_THREADS];
int intfTime=0;


void *startService(void *threadp);

unsigned const int seqIterations = 47;
unsigned const int Iterations = 1000;


#define FIB_TEST(seqCnt, iterCnt)       \
   for(idx=0; idx < iterCnt; idx++)     \
   {                                    \
      fib = fib0 + fib1;                \
        while(jdx < seqCnt)                 \
        {                                   \
           fib0 = fib1;                     \
           fib1 = fib;                      \
           fib = fib0 + fib1;               \
           jdx++;                           \
        }                                   \
     }                                      \


void *idle(void *threadp)
{
  struct timespec timeNow;
  unsigned int idx = 0, jdx = 1;
  volatile unsigned int fib = 0, fib0 = 0, fib1 = 1;
  threadParams_t *threadParams = (threadParams_t *)threadp;
  int idleIdx = threadParams->threadIdx;

  do
  {
    FIB_TEST(seqIterations, Iterations);
    idleCount[idleIdx]++;
  } while(idleCount[idleIdx] < runInterference);

  gettimeofday(&timeNow, (void *)0);
  printf("**** %d idle stopping at %d sec, %d nsec\n", idleIdx, (int)timeNow.tv_

  pthread_exit(NULL);
}


void print_scheduler(void)
{
   int schedType;

   schedType = sched_getscheduler(getpid());

   switch(schedType)
   {
     case SCHED_FIFO:
           printf("Pthread Policy is SCHED_FIFO\n");
           break;
     case SCHED_OTHER:
           printf("Pthread Policy is SCHED_OTHER\n");
       break;
     case SCHED_RR:
           printf("Pthread␣Policy␣is␣SCHED_OTHER\n");
           break;
     default:
       printf("Pthread␣Policy␣is␣UNKNOWN\n");
   }
}

int main (int argc, char *argv[])
{
   int rc, invSafe=0, i, scope;
   struct timespec sleepTime, dTime;

   CScount=0;

   if(argc < 2)
   {
     printf("Usage:␣pthread␣interfere-seconds\n");
     exit(-1);
   }
   else if(argc >= 2)
   {
     sscanf(argv[1], "%d", &intfTime);
     printf("interference␣time␣=␣%d␣secs\n", intfTime);
     printf("unsafe␣mutex␣will␣be␣created\n");
   }

   print_scheduler();

   pthread_attr_init(&rt_sched_attr[START_SERVICE]);
   pthread_attr_setinheritsched(&rt_sched_attr[START_SERVICE], PTHREAD_EXPLICIT_
   pthread_attr_setschedpolicy(&rt_sched_attr[START_SERVICE], SCHED_FIFO);

   pthread_attr_init(&rt_sched_attr[HIGH_PRIO_SERVICE]);
   pthread_attr_setinheritsched(&rt_sched_attr[HIGH_PRIO_SERVICE], PTHREAD_EXPLI
   pthread_attr_setschedpolicy(&rt_sched_attr[HIGH_PRIO_SERVICE], SCHED_FIFO);

   pthread_attr_init(&rt_sched_attr[MID_PRIO_SERVICE]);
   pthread_attr_setinheritsched(&rt_sched_attr[MID_PRIO_SERVICE], PTHREAD_EXPLIC
   pthread_attr_setschedpolicy(&rt_sched_attr[MID_PRIO_SERVICE], SCHED_FIFO);

   pthread_attr_init(&rt_sched_attr[LOW_PRIO_SERVICE]);
   pthread_attr_setinheritsched(&rt_sched_attr[LOW_PRIO_SERVICE], PTHREAD_EXPLIC
   pthread_attr_setschedpolicy(&rt_sched_attr[LOW_PRIO_SERVICE], SCHED_FIFO);

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

```
rc=sched_getparam(getpid(), &nrt_param);

if (rc)
{
    printf("ERROR; sched_setscheduler rc is %d\n", rc);
    perror(NULL);
    exit(-1);
}

print_scheduler();

printf("min prio = %d, max prio = %d\n", rt_min_prio, rt_max_prio);
pthread_attr_getscope(&rt_sched_attr[START_SERVICE], &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");


rt_param[START_SERVICE].sched_priority = rt_max_prio;
pthread_attr_setschedparam(&rt_sched_attr[START_SERVICE], &rt_param[START_SER

printf("Creating thread %d\n", START_SERVICE);
threadParams[START_SERVICE].threadIdx=START_SERVICE;
rc = pthread_create(&threads[START_SERVICE], &rt_sched_attr[START_SERVICE], s

if (rc)
{
    printf("ERROR; pthread_create() rc is %d\n", rc);
    perror(NULL);
    exit(-1);
}
printf("Start services thread spawned\n");


printf("will join service threads\n");

if(pthread_join(threads[START_SERVICE], NULL) == 0)
  printf("START SERVICE done\n");
else
  perror("START SERVICE");
```

```
    rc=sched_setscheduler(getpid(), SCHED_OTHER, &nrt_param);

    printf("All␣done\n");

    exit(0);
}


void *startService(void *threadp)
{
    struct timespec timeNow;
    int rc;

    runInterference=intfTime;

    rt_param[HIGH_PRIO_SERVICE].sched_priority = rt_max_prio-1;
    pthread_attr_setschedparam(&rt_sched_attr[HIGH_PRIO_SERVICE], &rt_param[HIGH_

    printf("Creating␣thread␣%d\n", HIGH_PRIO_SERVICE);
    threadParams[START_SERVICE].threadIdx=HIGH_PRIO_SERVICE;
    rc = pthread_create(&threads[HIGH_PRIO_SERVICE], &rt_sched_attr[HIGH_PRIO_SER

    if (rc)
    {
        printf("ERROR;␣pthread_create()␣rc␣is␣%d\n", rc);
        perror(NULL);
        exit(-1);
    }
    //pthread_detach(threads[HIGH_PRIO_SERVICE]);
    gettimeofday(&timeNow, (void *)0);
    printf("High␣prio␣%d␣thread␣spawned␣at␣%d␣sec,␣%d␣nsec\n", HIGH_PRIO_SERVICE,



    rt_param[MID_PRIO_SERVICE].sched_priority = rt_max_prio-10;
    pthread_attr_setschedparam(&rt_sched_attr[MID_PRIO_SERVICE], &rt_param[MID_PR

    printf("Creating␣thread␣%d\n", MID_PRIO_SERVICE);
    threadParams[START_SERVICE].threadIdx=MID_PRIO_SERVICE;
    rc = pthread_create(&threads[MID_PRIO_SERVICE], &rt_sched_attr[MID_PRIO_SERVI

    if (rc)
    {
        printf("ERROR;␣pthread_create()␣rc␣is␣%d\n", rc);
    perror(NULL);
    exit(-1);
}
//pthread_detach(threads[MID_PRIO_SERVICE]);
gettimeofday(&timeNow, (void *)0);
printf("Middle␣prio␣%d␣thread␣spawned␣at␣%d␣sec,␣%d␣nsec\n", MID_PRIO_SERVICE


rt_param[LOW_PRIO_SERVICE].sched_priority = rt_max_prio-20;
pthread_attr_setschedparam(&rt_sched_attr[LOW_PRIO_SERVICE], &rt_param[LOW_PI

printf("Creating␣thread␣%d\n", LOW_PRIO_SERVICE);
threadParams[START_SERVICE].threadIdx=LOW_PRIO_SERVICE;
rc = pthread_create(&threads[LOW_PRIO_SERVICE], &rt_sched_attr[LOW_PRIO_SERVI

if (rc)
{
    printf("ERROR;␣pthread_create()␣rc␣is␣%d\n", rc);
    perror(NULL);
    exit(-1);
}
//pthread_detach(threads[LOW_PRIO_SERVICE]);
gettimeofday(&timeNow, (void *)0);
printf("Low␣prio␣%d␣thread␣spawned␣at␣%d␣sec,␣%d␣nsec\n", LOW_PRIO_SERVICE, (




if(pthread_join(threads[LOW_PRIO_SERVICE], NULL) == 0)
  printf("LOW␣PRIO␣done\n");
else
  perror("LOW␣PRIO");

if(pthread_join(threads[MID_PRIO_SERVICE], NULL) == 0)
  printf("MID␣PRIO␣done\n");
else
  perror("MID␣PRIO");

if(pthread_join(threads[HIGH_PRIO_SERVICE], NULL) == 0)
  printf("HIGH␣PRIO␣done\n");
else
  perror("HIGH␣PRIO");


pthread_exit(NULL);
}
```

Code for Problem 4 File: heap_mq.c

```
#include <stdio.h>

#include <fcntl.h>
#include <mqueue.h>
#include <sys/stat.h>
#include <pthread.h>
#include <unistd.h>
#include <semaphore.h>
#include <string.h>
#include <stdlib.h>


#define MESSAGE_QUEUE_NAME "/sendRecvQueueNew"

#define MAX_SIZE 4096

static char imagebuff[MAX_SIZE];
struct mq_attr mq_attr;
static mqd_t mymq;

void* heap_mq(void* param)
{
  if(mq_unlink(MESSAGE_QUEUE_NAME) == -1)
  {
    printf("Heap:␣Failed␣to␣unlink␣the␣message␣queue\n");
  }

  int i, j;
  char pixel = 'A';

  for(i=0;i<4096;i+=64) {
    pixel = 'A';
    for(j=i;j<i+64;j++) {
      imagebuff[j] = (char)pixel++;
    }
    imagebuff[j-1] = '\n';
  }
  imagebuff[4095] = '\0';
  imagebuff[63] = '\0';

  printf("buffer␣=␣%s\n", imagebuff);

  /* setup common message q attributes */
  mq_attr.mq_maxmsg = 100;
  mq_attr.mq_msgsize = sizeof(void *)+sizeof(int);

  mq_attr.mq_flags = 0;

  mymq = mq_open(MESSAGE_QUEUE_NAME, O_CREAT|O_RDWR, S_IRUSR | S_IWUSR, &mq_attr

  if(mymq == -1)
    perror("mq_open");

}


void *newSenderThread(void *param)
{
  char buffer[sizeof(void *)+sizeof(int)];
  void *buffptr;
  int prio;
  int nbytes;
  int id = 999;

  if(mq_unlink(MESSAGE_QUEUE_NAME) == -1)
  {
    printf("SenderThread: Failed to unlink the message queue\n");
  }

  mqd_t queue = mq_open(MESSAGE_QUEUE_NAME , O_CREAT | O_RDWR, S_IRUSR | S_IWUSR
  if(queue == -1)
  {
      perror("Sender Thread: Failed to open queue");
  }



    /* send malloc'd message with priority=30 */

    buffptr = (void *)malloc( sizeof(imagebuff));
    strcpy(buffptr, imagebuff);
    printf("Message to send = %s\n", (char *)buffptr);

    printf("Sending %ld bytes\n", sizeof(buffptr));

    memcpy(buffer, &buffptr, sizeof(void *));
    memcpy(&(buffer[sizeof(void *)]), (void *)&id, sizeof(int));

    if(mq_send(queue, buffer, (size_t)(sizeof(void *)+sizeof(int)), 30) == -1)
    {
      perror("mq_send");
    }
    else
    {
      printf("send:␣message␣ptr␣0x%p␣successfully␣sent\n", buffptr);
    }

    //taskDelay(3000);

}


void *newreceiverThread(void *param)
{
  char buffer[sizeof(void *)+sizeof(int)];
  void *buffptr;
  int prio;
  int nbytes;
  int count = 0;
  int id;

  sleep(2);
  mqd_t queue = mq_open(MESSAGE_QUEUE_NAME , O_CREAT | O_RDONLY | O_NONBLOCK, S_
  if(queue == -1)
  {
    perror("Failed␣to␣open␣queue\n");
  }



    /* read oldest, highest priority msg from the message queue */

    printf("Reading␣%ld␣bytes\n", sizeof(void *));

    if((nbytes = mq_receive(queue, buffer, (size_t)(sizeof(void *)+sizeof(int)),
/*
    if((nbytes = mq_receive(mymq, (void *)&buffptr, (size_t)sizeof(void *), &pri
*/
    {
      perror("mq_receive");
    }
    else
    {
      memcpy(&buffptr, buffer, sizeof(void *));
      memcpy((void *)&id, &(buffer[sizeof(void *)]), sizeof(int));
      printf("receive:␣ptr␣msg␣0x%p␣received␣with␣priority␣=␣%d,␣length␣=␣%d,␣id
      printf("contents␣of␣ptr␣=␣\n%s\n", (char *)buffptr);

      free(buffptr);

      printf("heap␣space␣memory␣freed\n");

      if(mq_close(queue) == -1)
      {
        printf("Receiver␣Thread:␣Failed␣to␣close␣the␣queue\n");
      }

      if(mq_unlink(MESSAGE_QUEUE_NAME) == -1)
      {
        printf("Receiver:␣Failed␣to␣unlink␣the␣message␣queue\n");
      }

    }

}


#define THREAD_NUM 3


int main()
{
  pthread_t threads[THREAD_NUM];

  pthread_create(&threads[0] , NULL , heap_mq , NULL);

  pthread_join(threads[0] , NULL);

  pthread_create(&threads[1] , NULL , newSenderThread , NULL);

  pthread_join(threads[1] , NULL);

  pthread_create(&threads[2] , NULL , newreceiverThread , NULL);

  pthread_join(threads[2] , NULL);

  // for (size_t i = 0; i < THREAD_NUM; i++)
  // {
  //      pthread_join(threads[i] , NULL);
  // }
  printf("Threads␣execution␣completion\n");

  return 0;
}
```

Code for Problem 4 File: posix_mq.c

```c
#include <stdio.h>

#include <fcntl.h>
#include <mqueue.h>
#include <sys/stat.h>
#include <pthread.h>
#include <unistd.h>
#include <semaphore.h>
#include <string.h>
#include <stdlib.h>


#define MESSAGE_QUEUE_NAME "/sendRecvQueuePosix"

//#define MAX_SIZE 4096

//static char imagebuff[MAX_SIZE];
//struct mq_attr mq_attr;
//static mqd_t mymq;



typedef struct{
    char msgText[100];
} message;

void *senderThread(void* param)
{
    if(mq_unlink(MESSAGE_QUEUE_NAME) == -1)
    {
        printf("SenderThread: Failed to unlink the message queue\n");
    }
    struct mq_attr attributes = {
        .mq_flags = 0,
        .mq_maxmsg = 10,
        .mq_curmsgs = 0,
        .mq_msgsize = sizeof(message)
    };

    mqd_t queue = mq_open(MESSAGE_QUEUE_NAME , O_CREAT | O_WRONLY, S_IRUSR | S_I

    if(queue == -1)
    {
        perror("SenderThread: Failed to create queue");
    }


    message message_to_send;
    strcpy(message_to_send.msgText , "Dhiraj␣is␣working␣on␣message␣Queues");
    //memcpy(message_to_send.msgText , imagebuff , sizeof(message_to_send.msgTex

    printf("SenderThread:␣Sender␣Queue␣Message␣=␣%s\n", message_to_send.msgText)

    if(mq_send(queue , (char*) &message_to_send , sizeof(message_to_send) , 1) =
    {
        printf("SenderThread:␣Failed␣to␣send␣message\n");
    }


    //printf("Type any key to finish\n");
    //getchar();


    if(mq_close(queue) == -1)
    {
        printf("SenderThread:␣Failed␣to␣close␣the␣message␣queue\n");
    }
    // if(mq_unlink(MESSAGE_QUEUE_NAME) == -1)
    // {
    //     printf("SenderThread: Failed to unlink the message queue\n");
    // }
}

void *receiverThread(void* param)
{
    struct mq_attr attributes = {
        .mq_flags = 0,
        .mq_maxmsg = 10,
        .mq_curmsgs = 0,
        .mq_msgsize = sizeof(message)
    };


    sleep(2);
    mqd_t queue = mq_open(MESSAGE_QUEUE_NAME , O_CREAT | O_RDONLY /*| O_NONBLOCK

    if(queue == -1)
    {
        perror("Receiver␣Thread:␣Failed␣to␣create␣queue\n");
    }
    message message_to_receive;
    //strcpy(&message_to_receive.msgText[0] , "Dhiraj is working on message queu

    if(mq_receive(queue , (char*) &message_to_receive , sizeof(message_to_receiv
    {
        perror("Receiver␣Thread:␣Failed␣to␣receive␣message\n");
    }

    printf("Receiver␣Thread:␣Receiver␣Queue␣Message␣=␣%s\n", message_to_receive.

    //printf("Type any key to finish\n");
    //getchar();

    if(mq_close(queue) == -1)
    {
        printf("Receiver␣Thread:␣Failed␣to␣close␣the␣queue\n");
    }

    if(mq_unlink(MESSAGE_QUEUE_NAME) == -1)
    {
        printf("Receiver:␣Failed␣to␣unlink␣the␣message␣queue\n");
    }
}

#define THREAD_NUM 2


int main()
{
    pthread_t threads[THREAD_NUM];

    //pthread_create(&threads[0] , NULL , heap_mq , NULL);

    pthread_create(&threads[0] , NULL , senderThread , NULL);

    //pthread_join(threads[0] , NULL);

    // for(int i = 0; i < 10000; i++)
    // {
    //    ;
    // }


    pthread_create(&threads[1] , NULL , receiverThread , NULL);
    //pthread_join(threads[1] , NULL);
    for (size_t i = 0; i < THREAD_NUM; i++)
    {
        pthread_join(threads[i] , NULL);
    }


    // (void)senderThread(NULL);
    // (void)receiverThread(NULL);

    printf("Threads␣execution␣completion\n");

    return 0;
}
```

Code for Problem 5 File: timedMutex.c

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<time.h>
#include <stdio.h>

#include <fcntl.h>
#include <mqueue.h>
#include <sys/stat.h>
#include <pthread.h>
#include <unistd.h>
#include <semaphore.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <errno.h>

#define NSEC_PER_SEC (1000000000)
#define NSEC_PER_MSEC (1000000)
#define NSEC_PER_MICROSEC (1000)
#define DELAY_TICKS (1)
#define ERROR (-1)
#define OK (0)

pthread_mutex_t mutex1,mutex2;
#define perror(s) printf("\n" s "\n");
int criticalSectionVariable;


int delta_t(struct timespec *stop, struct timespec *start, struct timespec *delt


void * func1(void *param)
{
    struct timespec start_time = {0, 0};
    struct timespec finish_time = {0, 0};
    struct timespec thread_dt = {0, 0};
    //printf("Thread =%d\n", *(int*)param);
    time_t rawtime;
    struct tm * timeinfo;
        typedef struct data{
        float X;
        float Y;
        float Z;
        struct timespec t_val;
    }data_t;


    data_t *att_data = calloc(1,sizeof(struct data));



    while(1)
    {
        att_data->X += 0.01;
        att_data->Y += 0.02;
        att_data->Z += 0.03;
        int clock_ret = clock_gettime(CLOCK_REALTIME,&att_data->t_val);

        if( att_data->Z>= 100.00)
        {
            att_data->X = 0.00;
            att_data->Y = 0.00;
            att_data->Z = 0.00;
        }

        char *str;
        asprintf(&str,"Timestamp:␣%d:%ld␣␣␣␣X:%f,␣Y:%f,␣Z:%f\n",att_data->t_val.
        clock_gettime(CLOCK_MONOTONIC, &start_time);
        pthread_mutex_lock(&mutex1);
        printf("Thread␣=␣%d␣in␣Critical␣Section\n", *(int*)param);
        int fd = open("/var/tmp/rtes",O_WRONLY|O_APPEND);
        if(fd < 0)
        {
            perror("File␣Descriptor␣failed\n");
        }
        write(fd,str,strlen(str));
        close(fd);
        sleep(11);
        clock_gettime(CLOCK_MONOTONIC, &finish_time);
        delta_t(&finish_time, &start_time, &thread_dt);

        time ( &rawtime );
        timeinfo = localtime ( &rawtime );
        printf ( "Data␣updated␣in␣file␣at␣time:␣%s", asctime (timeinfo) );
        printf("Data␣=␣%s\n", str);
        pthread_mutex_unlock(&mutex1);
    }

}
```

```
void * func2(void *param)
{
    struct timespec start_time = {0, 0};
    struct timespec finish_time = {0, 0};
    struct timespec thread_dt = {0, 0};
    int retVal = 0;
    time_t rawtime;
    struct tm * timeinfo;
    sleep(0.5);

    while (1)
    {

        clock_gettime(CLOCK_MONOTONIC, &start_time);
        struct timespec *wait;
        struct timespec newWait;
        int ret = 0;
        wait =(struct timespec *)(malloc(sizeof(struct timespec)));

        clock_gettime(CLOCK_REALTIME, wait);
        newWait.tv_sec = wait->tv_sec + 10;
        newWait.tv_nsec = wait->tv_nsec + 0;

        ret = pthread_mutex_timedlock(&mutex1,&newWait);
        if(ret != 0)
        {
            switch(ret)
            {
                case EINVAL:
                    printf("EINVAL\n");
                break;
                case EAGAIN:
                    printf("EAGAIN\n");
                break;
                case ETIMEDOUT:
                    printf("ETIMEDOUT\n");
                break;
                case EDEADLK:
                    printf("EDEADLK\n");
                break;
                default:
                    printf("unknown\n");
                break;

            }
        }
```

```
            if(ret!=0)
            {
                printf("Thread = %d did not get to lock the mutex , Error Code = %d\

            }
            else
            {
                printf("Thread = %d in Critical Section\n", *(int*)param);
                retVal = pthread_mutex_unlock(&mutex1);
                if(retVal != 0)
                {
                    printf("Did not unlock the mutex\n");
                }
            }
            clock_gettime(CLOCK_MONOTONIC, &finish_time);

            time ( &rawtime );
            timeinfo = localtime ( &rawtime );
            printf ( "No new data available at time : %s", asctime (timeinfo) );

        }
    }




    int main() {

        int ret;
        int fd = open("/var/tmp/rtes",O_RDWR|O_CREAT|O_TRUNC, 0666);
        if(fd<0)
            perror("File Descriptor Issue\n");

        pthread_t thread1,thread2;
        int a = 1;
        int b = 2;

        ret=pthread_create(&thread1,NULL,&func1,&a);

        if(ret!=0)
        {
            perror("Unable to create thread");
        }
        ret=pthread_create(&thread2,NULL,&func2,&b);
        if(ret!=0)
        {
            perror("Unable␣to␣␣create␣thread");
        }

        pthread_join(thread1,NULL);
        pthread_join(thread2,NULL);

        return 0;
    }


    int delta_t(struct timespec *stop, struct timespec *start, struct timespec *delt
    {
      int dt_sec=stop->tv_sec - start->tv_sec;
      int dt_nsec=stop->tv_nsec - start->tv_nsec;

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

      // case 1 - less than a second of change
      if(dt_sec == 0)
      {
              //printf("dt less than 1 second\n");

              if(dt_nsec >= 0 && dt_nsec < NSEC_PER_SEC)
              {
                      //printf("nanosec greater at stop than start\n");
                      delta_t->tv_sec = 0;
                      delta_t->tv_nsec = dt_nsec;
              }

              else if(dt_nsec > NSEC_PER_SEC)
              {
                      //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");
                     return(ERROR);
              }
      }
```

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

            if(dt_nsec >= 0 && dt_nsec < NSEC_PER_SEC)
            {
                    //printf("nanosec greater at stop than start\n");
                    delta_t->tv_sec = dt_sec;
                    delta_t->tv_nsec = dt_nsec;
            }

            else if(dt_nsec > NSEC_PER_SEC)
            {
                    //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;
                    delta_t->tv_nsec = NSEC_PER_SEC + dt_nsec;
            }
    }

    return(OK);
}
```

References

[1] Priority Inheritance Protocols: An Approach to Real-Time synchronization
Lui Sha, Raghunathan Rajkumar, John P Lehoczky

[2] SoC drawer: Shared resource management
Dr Sam Siewert

[3] https://lwn.net/Articles/178253/

[4] https://lwn.net/Articles/177111/

[5] https://en.wikipedia.org/wiki/Futex

[6] Linux Watchdog Deamon

[7] https://linuxhint.com/linux-kernel-watchdog-explained/

[8] Code Source: http://mercury.pr.erau.edu/ siewerts/cec450/code/
Courtesy: Dr Sam Siewert
Professor Tim Scherr