Department of Electrical and Computer Engineering

University of Colorado at Boulder

ECEN5623 - Real Time Embedded Systems

Exercise 3

Submitted by

Shashank — Parth

Submitted on March 9, 2024

Contents

List of Figures

List of Tables

PDF is clickable

Objective

1. Understanding the concept of the Cyclic Executive in comparison to Linux POSIX RT threading and RTOS. Implementing and analyzing custom feasibility test code for different scheduling policies (RM, EDF, LLF) using Cheddar.

2. Moreover, understanding the constraints, assumptions, and derivation steps in Rate Monotonic (RM) Least Upper Bound (LUB) as outlined in Chapter 3 of the textbook.

1 Question 1

Q: [10 points] [All papers here also on Canvas] Read Sha, Rajkumar, et al paper, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization"

(a) **Q: Summarize 3 main key points the paper makes. Read Dr. Siewert's summary paper on the topic as well which might be easier to understand.**

Answer: This paper studies priority inheritance protocols, such as the basic and priority ceiling protocols, to address uncontrolled priority inversion. This occurs when a high-priority task is delayed by a low-priority task accessing a shared resource. The priority ceiling protocol limits the maximum blocking time of a high-priority task to the duration of the low-priority task, reducing the chance of deadlocks. Following are the 3 main key points the paper makes:

Priority Inheritance Protocols: Real-time systems face challenges in scheduling tasks effectively, especially when tasks need to share resources. Traditional synchronization methods, such as semaphores, can lead to priority inversion issues, where lower priority tasks hold up higher priority ones. Priority inheritance protocols offer a solution to this problem by dynamically adjusting the priorities of tasks to ensure that higher priority tasks are not unnecessarily delayed by lower priority ones.

Basic Priority Inheritance Protocol: The basic priority inheritance protocol temporarily boosts the priority of a lower priority task to match that of a higher priority one while it accesses a shared resource. This prevents priority inversion and helps ensure that tasks meet their deadlines. However, the basic protocol still has limitations, such as the potential for deadlocks and chains of blocking events, which can impact system reliability and predictability.

Priority Ceiling Protocol: To address the limitations of the basic protocol, the priority ceiling protocol introduces the concept of priority ceilings for shared resources. Each resource is assigned a maximum priority level, and when a task accesses the resource, its priority is temporarily raised to the ceiling level. This prevents lower priority tasks from blocking higher priority ones while still allowing for efficient resource sharing. A set of n periodic tasks can be scheduled using the priority ceiling protocol, if task priorities adhere to Rate Monotonic (RM) theory and specific conditions are met. Although the priority ceiling protocol introduces its own challenges, such as ceiling blocking, it offers a more robust solution compared to the basic protocol.

In summary, priority inheritance protocols play a crucial role in ensuring the effective scheduling of tasks in real-time systems. By addressing issues such as priority inversion, these protocols help improve system reliability and meet performance requirements. The priority ceiling protocol offers a promising solution by introducing priority ceilings for shared resources, although it requires careful consideration of its implications for system design and implementation.

(b) **Q: Read the historical positions of Linus Torvalds as described by Jonathan Corbet and Ingo Molnar and Thomas Gleixner on this topic as well. Take a position on this topic yourself and write at least one well supported paragraph or more to defend your position based on what we have learned in class. Does the PI-futex (Futex, Futexes are Tricky) that is described by Ingo Molnar provide safe and accurate protection from un-bounded priority inversion as described in the paper? If not, what is different about it?**

Answer: Linus Torvalds, the creator of Linux, firmly opposes using priority inheritance protocols to solve priority inversion. He suggests avoiding these protocols by designing software without locks or using carefully crafted locks to prevent priority inversion. However, this approach may not be practical for complex real-world applications where shared resources and task prioritization are essential.

In contrast, Ingo Molnar, a member of the kernel development community, believes that priority inheritance protocols are necessary and proposes using priority inheritance futexes (PI-Futex) to address priority inversion effectively. PI-Futexes work by temporarily boosting the priority of lower-priority tasks to prevent them from delaying higher-priority tasks indefinitely. This solution operates mainly in user space, providing efficient and precise protection against priority inversion.

Our stance aligns more closely with Ingo Molnar's perspective, advocating for the adoption of priority inheritance protocols in situations where priority inversion poses a significant risk. While Torvalds' argument against priority inheritance may apply in some cases, it overlooks the complexities of real-time systems and the need for efficient synchronization mechanisms.

PI-Futex The PI-Futex, created by Ingo Molnar, is an advanced tool for dealing with uncontrolled priority inversion in real-time systems. It mainly works in user space, handling locking tasks quickly and efficiently without relying too much on the kernel. If there's a conflict over a shared resource, the PI-Futex boosts the priority of lower-priority tasks temporarily so that higher-priority tasks can keep moving smoothly.

However, it's not a perfect solution for all cases of priority inversion, especially when the issue stems from deeper kernel-level locking mechanisms like mutexes or semaphores. In those situations, priority inversion can still happen even if we are using PI-Futexes.

Despite its flaws, the PI-Futex shows promise in preventing uncontrolled priority inversion, especially in systems where most locking happens in user space. However, how well it works depends on how it's implemented and the specific context of the application. Other factors, such as the nature of kernel-level locking mechanisms and the complexity of system interactions, can also affect how effective PI-Futexes are in tackling priority inversion.

(c) **Q: Note that more recently Red Hat has added support for priority ceiling emulation and priority inheritance and has a great overview on use of these POSIX real-time extension features here – general real-time overview. The key systems calls are pthread_mutexattr_setprotocol, pthread_mutexattr_getprotocol as well as pthread_mutex_setpriorityceiling and pthread_mutex_getpriorityceiling. Why did some of the Linux community resist addition of priority ceiling and/or priority inheritance until more recently? (Linux has been distributed since the early 1990's and the problem and solutions were known before this).**

Answer: The hesitation within the Linux community to adopt priority ceiling and priority inheritance features until more recently can be explained by a few reasons. Generally, in the world of Linux development, there's a reluctance to add new features because they might make things more complicated, less stable, or slower. Adding priority ceiling and inheritance to the kernel's scheduler and locking mechanisms could make the system more complex and might affect how well it runs and how easy it is to keep it running smoothly. There were also discussions about whether these features were really needed for most Linux uses, since Linux is used for many different things, not just real-time tasks.

Another reason for the hesitation is the belief that there are other ways to fix priority inversion issues without needing priority inheritance. Linus Torvalds, who created Linux, wasn't convinced that priority inheritance was necessary and thought there might be better solutions.

Additionally, there were concerns about how hard it would be to add these features to the Linux kernel and what problems it might cause. Making the kernel more complicated could lead to more bugs or security issues, which nobody wants. Plus, there were other things that people wanted to work on in the Linux community, so priority ceiling and inheritance might not have been seen as a top priority.

But as people started to realize how important real-time capabilities were becoming for Linux, especially in areas like embedded systems and industrial automation, there was more demand for these features. Red Hat, a big player in the Linux world, recently added support for priority ceiling and inheritance, showing that the community is starting to take these needs seriously. As Linux keeps evolving, adding these features shows that people are recognizing the importance of making Linux work well for real-time tasks.

(d) **Q: Given that there are two solutions to unbounded priority inversion, priority ceiling emulation and priority inheritance, based upon your reading on support for both by POSIX real-time extensions, which do you think is best to use and why?**
Answer: When deciding between priority ceiling emulation and priority inheritance to tackle unbounded priority inversion, it's essential to consider how each method works and their respective advantages and drawbacks. Priority inheritance temporarily boosts the priority of a task holding a resource, reducing the time higher-priority tasks are blocked. In contrast, priority ceiling emulation ensures that a task can only access a resource if its priority is equal to or higher than the highest-priority task that might use the resource, preventing priority inversion from happening.

The choice between the two methods depends on the specific needs and limitations of the system. Priority inheritance may seem simpler to implement and understand, but it could lead to unexpected scheduling outcomes if priorities are frequently elevated. Priority ceiling emulation, while more effective at preventing priority inversion, requires a deep understanding of the system's resource usage patterns to set appropriate ceiling priorities, which can be complex and prone to errors. In systems with well-defined and unchanging resource access patterns, priority ceiling emulation might be preferable because it proactively prevents priority inversion. However, in dynamic systems or environments where predicting task-resource interactions is challenging, priority inheritance could offer more flexibility. Ultimately, the best choice depends on factors like system performance, complexity, and ease of maintenance in specific application and environment.

2 Question 2

Q: [25 points] Review the terminology guide (glossary in the textbook)

(a) **Q: Describe clearly what it means to write "thread safe" functions that are "re-entrant". There are generally three main ways to do this:**

  i. **pure functions that use only stack and have no global memory,**
  ii. **functions which use thread indexed global data,**
  iii. **functions which use shared memory global data, but synchronize access to it using a MUTEX semaphore critical section wrapper**

Answer:
thread-safe functions that are re-entrant involves creating functions that can be safely called and executed by multiple threads simultaneously without causing data corruption or unpredictable behavior. A function is considered thread-safe if it can be safely invoked by multiple threads at the same time. A function is re-entrant if it can be interrupted in the middle of execution and safely called again **("re-entered")** before the previous executions are complete. For a function to be both thread-safe and re-entrant, we can use these three techniques

i. Pure Functions That Use Only Stack and Have No Global Memory Pure functions depend solely on their input arguments and do not use or modify any shared state, including global variables or static data. Each thread's invocation of a pure function is completely independent of another's, as all necessary data is passed in as parameters, and any state is kept on the thread's own stack. Since there is no shared state and no side effects, pure functions are inherently thread-safe and re-entrant.

ii. Functions Which Use Thread-Indexed Global Data These functions use global data that is indexed in such a way that each thread accesses a separate instance of the data. This can be achieved using thread-local storage (TLS), where each thread has its own instance of a variable. Although the data might be globally accessible within the process, each thread's view of the global data is unique and isolated, preventing data races and ensuring thread safety.

iii. Functions Which Use Shared Memory Global Data, But Synchronize Access to It Using a MUTEX Semaphore Critical Section Wrapper When functions need to access and modify shared global data, ensuring thread safety requires preventing multiple threads from modifying the data concurrently. This is typically achieved using mutual exclusion (mutex) locks or other synchronization primitives to create a critical section—a section of code that only one thread can execute at a time. Before a thread enters a critical section, it must acquire the mutex; when it's done, it releases the mutex. This ensures that only one thread at a time can access the shared data, preventing race conditions.

(b) **Q: Describe each method and how you would code it and how it would impact real-time threads/tasks**

Answer:

i. Pure Functions That Use Only Stack and Have No Global Memory Pure functions depend solely on their input arguments and do not use or modify any shared state, including global variables or static data. Each thread's invocation of a pure function is completely independent of another's, as all necessary data is passed in as parameters, and any state is kept on the thread's own stack. Since there is no shared state and no side effects, pure functions are inherently thread-safe and re-entrant.

```c
#include <stdio.h>
#include <pthread.h>

pthread_t threads[2];



void * task(void * arg){
  int counter = 0;
  for(int i=0; i<10000; i++){
    counter++;
    printf("Thread %ld, sharedCounter: %d\n", (long)arg, counter);
  }
  return NULL;

}



int main(){
  for (long i = 0; i < 2; i++) {
    pthread_create(&threads[i], NULL, task, (void*)i);
  }

  for (int i = 0; i < 2; i++) {
    pthread_join(threads[i], NULL);
  }
  return 0;
```

```
}
```

ii. Functions Which Use Thread-Indexed Global Data These functions use global data that is indexed in such a way that each thread accesses a separate instance of the data. This can be achieved using thread-local storage (TLS), where each thread has its own instance of a variable. Although the data might be globally accessible within the process, each thread's view of the global data is unique and isolated, preventing data races and ensuring thread safety.
Example:

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

pthread_t threads[2];
// Thread-local indexed variable to store counter
__thread int counter = 0;


void* task(void * arg){
  for(int i=0; i<10000; i++){
    counter++;
    printf("Thread %ld, sharedCounter: %d\n", (long)arg, counter);
  }

  return NULL;

}



int main(){
  for (long i = 0; i < 2; i++) {
    pthread_create(&threads[i], NULL, task, (void*)i);
  }

  for (int i = 0; i < 2; i++) {
    pthread_join(threads[i], NULL);
  }
  return 0;
}

```

iii. Functions Which Use Shared Memory Global Data, But Synchronize Access to It Using a MUTEX Semaphore Critical Section Wrapper When functions need to access and modify shared global data, ensuring thread safety requires preventing multiple threads from modifying the data concurrently. This is typically achieved using mutual exclusion (mutex) locks or other synchronization primitives to create a critical section—a section of code that only one thread can execute at a time. Before a thread enters a critical section, it must acquire the mutex; when it's done, it releases the mutex. This ensures that only one thread at a time can access the shared data, preventing race conditions.

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


  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

  pthread_t threads[2];

  int sharedCounter = 0;

  void* task(void * arg){
    for(int i=0; i<10000; i++){
      pthread_mutex_lock(&mutex);
```

```
        sharedCounter++;
        printf("Thread %ld, sharedCounter: %d\n", (long)arg, sharedCounter);
        // Unlock the mutex after updating
        pthread_mutex_unlock(&mutex);
      }

      return NULL;
    }


    int main(){
      pthread_mutex_init(&mutex, NULL);

      for (long i = 0; i < 2; i++) {
        pthread_create(&threads[i], NULL, task, (void*)i);
      }

      for (int i = 0; i < 2; i++) {
        pthread_join(threads[i], NULL);
      }
      pthread_mutex_destroy(&mutex);
      return 0;
    }

```

(c) **Q: Now, using a MUTEX, provide an example using RT-Linux Pthreads that does a thread safe update of a complex state (3 or more numbers – e.g., Latitude, Longitude and Altitude of a location) with a timestamp (pthread_mutex_lock). Your code should include two threads and one should update a timespec structure contained in a structure that includes a double precision attitude state of Lat, Long, Altitude and Roll, Pitch, Yaw at Sample_Time and the other should read it and never disagree on the values as function of time. You can just make up values for the navigational state using math library function generators (e.g., use simple periodic functions for Roll, Pitch, Yaw sin(x), cos(x2), and cos(x), where x=time and linear functions for Lat, Long, Alt) and see http://linux.die.net/man/3/clock_gettime for how to get a precision timestamp. The second thread should read the times-stamped state without the possibility of data corruption (partial update of one of the 6 floating point values). There should be no disagreement between the functions and the state reader for any point in time. Run this for 180 seconds with a 1 Hz update rate and a 0.1 Hz read rate. Make sure the 18 values read are correct.**

Answer:

In the provided code, a mutex (short for "mutual exclusion") is used to ensure that the nav_state structure is accessed in a thread-safe manner. This is crucial because the program has two separate threads that interact with this shared data structure: one thread updates it (update_thread), and the other reads from it (read_thread). Without proper synchronization, concurrent access by these threads could lead to race conditions, where the outcome depends on the non-deterministic scheduling of threads by the operating system. Race conditions can cause data corruption or lead to unpredictable program behavior.

How the Mutex is Used:

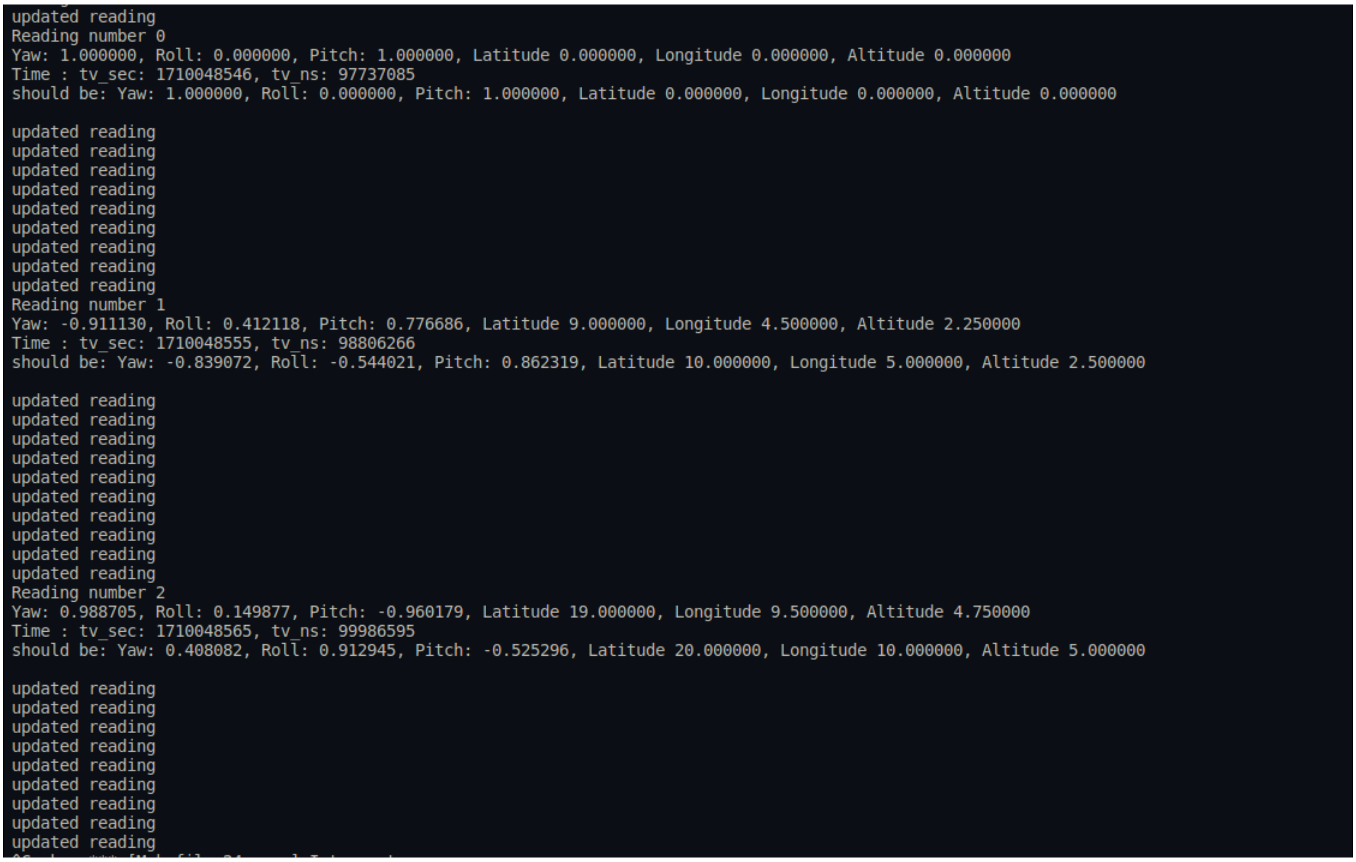
- **Initialization:** Before the threads are created, the mutex is initialized with pthread_mutex_init(&mutex, NULL);. This prepares the mutex for use.
- **Locking and Unlocking in update_thread:**
  - Before modifying nav_state, the update_thread acquires the mutex lock with pthread_mutex_lock(&mutex);. This ensures exclusive access to nav_state, preventing the read_thread from accessing it simultaneously.

  - After updating nav_state and printing the updated values, the update_thread releases the lock with pthread_mutex_unlock(&mutex);, allowing other threads to acquire the mutex and access nav_state.
- **Locking and Unlocking in read_thread:**
  - Similarly, the read_thread acquires the mutex lock before copying nav_state to a local variable temp_state for reading. This prevents it from reading nav_state while it might be concurrently modified by update_thread.
  - Once the data has been safely copied and the necessary information printed, the read_thread releases the mutex lock, allowing other threads (in this case, specifically the update_thread) to acquire the mutex.

Purpose of Mutex Use:

- **Ensure Data Consistency:** By enforcing mutual exclusion on access to nav_state, the mutex prevents scenarios where read_thread might see partially updated data or update_thread might overwrite changes while read_thread is in the middle of reading data.
- **Prevent Race Conditions:** The mutex ensures that only one thread can modify or read nav_state at a time.
- **Enable Safe Concurrency:** Although the mutex serializes access to nav_state, making the operations on it effectively atomic from each thread's perspective, it allows the rest of the program to execute concurrently.

Output:



```
updated reading
Reading number 0
Yaw: 1.000000, Roll: 0.000000, Pitch: 1.000000, Latitude 0.000000, Longitude 0.000000, Altitude 0.000000
Time : tv_sec: 1710048546, tv_ns: 97737085
should be: Yaw: 1.000000, Roll: 0.000000, Pitch: 1.000000, Latitude 0.000000, Longitude 0.000000, Altitude 0.000000

updated reading
updated reading
updated reading
updated reading
updated reading
updated reading
updated reading
updated reading
updated reading
Reading number 1
Yaw: -0.911130, Roll: 0.412118, Pitch: 0.776686, Latitude 9.000000, Longitude 4.500000, Altitude 2.250000
Time : tv_sec: 1710048555, tv_ns: 98806266
should be: Yaw: -0.839072, Roll: -0.544021, Pitch: 0.862319, Latitude 10.000000, Longitude 5.000000, Altitude 2.500000

updated reading
updated reading
updated reading
updated reading
updated reading
updated reading
updated reading
updated reading
updated reading
updated reading
Reading number 2
Yaw: 0.988705, Roll: 0.149877, Pitch: -0.960179, Latitude 19.000000, Longitude 9.500000, Altitude 4.750000
Time : tv_sec: 1710048565, tv_ns: 99986595
should be: Yaw: 0.408082, Roll: 0.912945, Pitch: -0.525296, Latitude 20.000000, Longitude 10.000000, Altitude 5.000000

updated reading
updated reading
updated reading
updated reading
updated reading
updated reading
updated reading
updated reading
updated reading
```

Figure 1: Output of simple mutex synchronization

Code is given in separated folder (Code_Q2_5/2c/simple.c)and in the appendices.

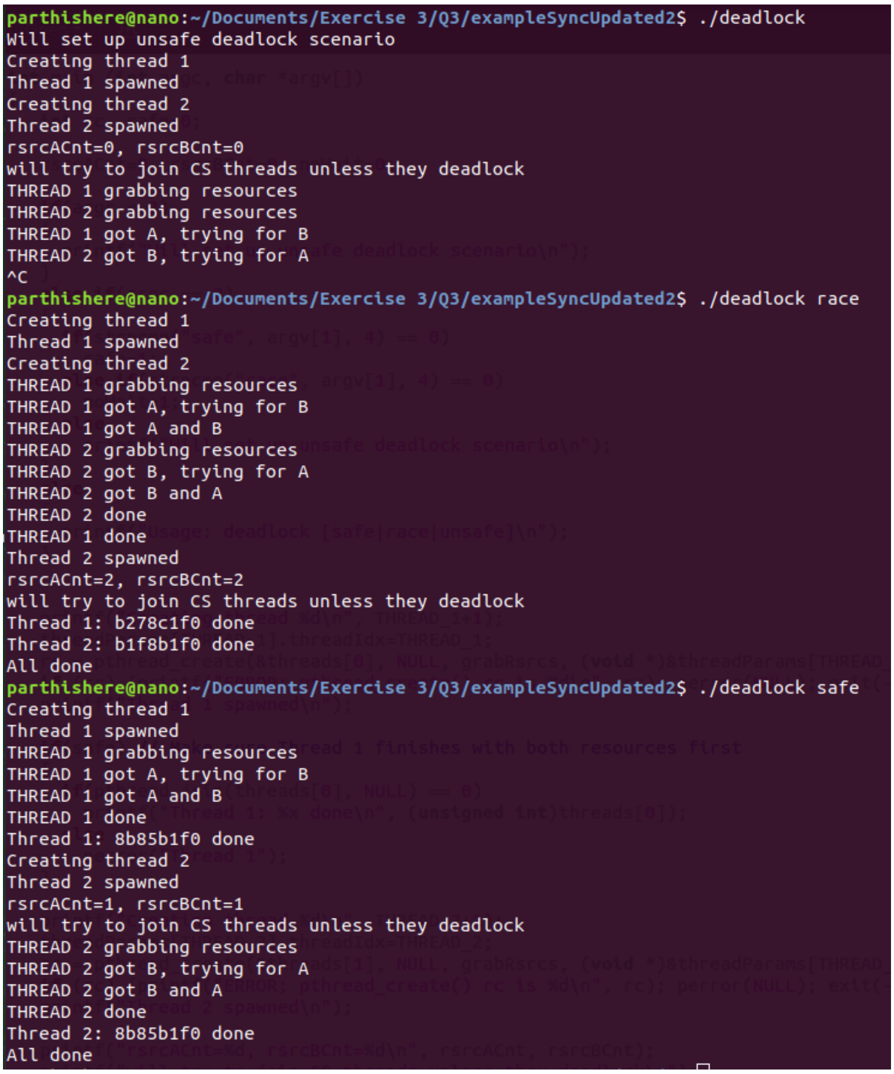
3 Question 3

Q: [20 points] Download example-sync-updated-2/ and review, build, and run it.

(a) **Q: Describe both the issues of deadlock and unbounded priority inversion and the root cause for both in the example code.**

Answer: Deadlock occurs when threads or processes are stuck waiting for each other to release resources they need. For instance, if thread 1 holds resource A and waits for B, while thread 2 holds B and waits for A, they're stuck.

Following are the outputs and root cause for the example codes:



```
parthishere@nano:~/Documents/Exercise 3/Q3/exampleSyncUpdated2$ ./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 1 grabbing resources
THREAD 2 grabbing resources
THREAD 1 got A, trying for B
THREAD 2 got B, trying for A
^C
parthishere@nano:~/Documents/Exercise 3/Q3/exampleSyncUpdated2$ ./deadlock race
Creating thread 1
Thread 1 spawned
Creating thread 2
THREAD 1 grabbing resources
THREAD 1 got A, trying for B
THREAD 1 got A and B
THREAD 2 grabbing resources
THREAD 2 got B, trying for A
THREAD 2 got B and A
THREAD 2 done
THREAD 1 done
Thread 2 spawned
rsrcACnt=2, rsrcBCnt=2
will try to join CS threads unless they deadlock
Thread 1: b278c1f0 done
Thread 2: b1f8b1f0 done
All done
parthishere@nano:~/Documents/Exercise 3/Q3/exampleSyncUpdated2$ ./deadlock safe
Creating thread 1
Thread 1 spawned
THREAD 1 grabbing resources
THREAD 1 got A, trying for B
THREAD 1 got A and B
THREAD 1 done
Thread 1: 8b85b1f0 done
Creating thread 2
Thread 2 spawned
rsrcACnt=1, rsrcBCnt=1
will try to join CS threads unless they deadlock
THREAD 2 grabbing resources
THREAD 2 got B, trying for A
THREAD 2 got B and A
THREAD 2 done
Thread 2: 8b85b1f0 done
All done
```

Figure 2: Deadlock.c

The deadlock.c code can be run in 3 different ways:

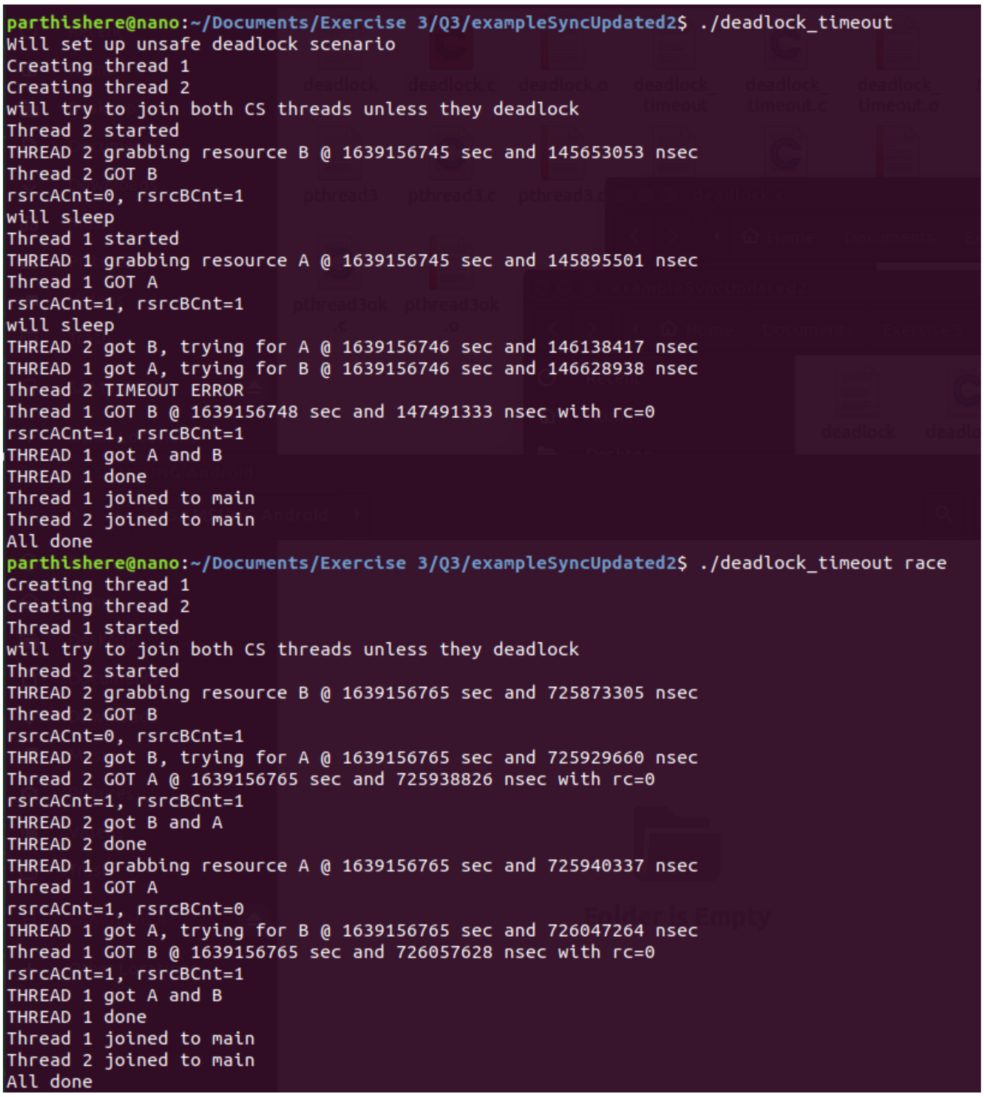
i. **Without any argument - ./deadlock**
This scenario depicts the actual deadlock where 2 threads are accessing the same function "grabRsrcs". Thread 1 acquires the lock "rsrcA" and sleeps for 1 second. Thread 2 acquires the lock "rsrcB" and sleeps for 1 second. Next, when thread 1 tries to acquire "rsrcB" and thread 2 tries to acquire "rsrcA", a deadlock situation is created as the lock required by one thread is already acquired by the other thread. Referring to the first portion of image, we can observe that since the deadlock has occurred, the code is stuck as both threads are waiting for the locks.

ii. **./deadlock race**
When race is used as an argument, a variable "noWait" is set to 1 which results in both threads not waiting for 1 second after acquiring their respective first lock. This results in

one thread completing and releasing both locks before the other thread requires the lock. The code doesn't create a deadlock but creates a race condition where any thread could execute any time and update the variables "rsrcACnt" and "rsrcBCnt" in a way which is not expected. The execution and values of variables are not fixed each time the code runs and hence the output is unpredictable. Since the threads are not blocking the resource, chances of a deadlock occurring are minimal but still can be possible.

iii. **./deadlock safe**

As observed in the last portion of the image, thread 1 executes first before thread 2 is created. In the code, when safe is passed as an argument, the "safe" variable is set to 1 which waits for the thread 1 to complete first before creating thread 2 and waiting for thread 2 to complete. In this way, the threads don't run in parallel, avoiding the deadlock scenario.



```
parthishere@nano:~/Documents/Exercise 3/Q3/exampleSyncUpdated2$ ./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 2 started
THREAD 2 grabbing resource B @ 1639156745 sec and 145653053 nsec
Thread 2 GOT B
rsrcACnt=0, rsrcBCnt=1
will sleep
Thread 1 started
THREAD 1 grabbing resource A @ 1639156745 sec and 145895501 nsec
Thread 1 GOT A
rsrcACnt=1, rsrcBCnt=1
will sleep
THREAD 2 got B, trying for A @ 1639156746 sec and 146138417 nsec
THREAD 1 got A, trying for B @ 1639156746 sec and 146628938 nsec
Thread 2 TIMEOUT ERROR
Thread 1 GOT B @ 1639156748 sec and 147491333 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
parthishere@nano:~/Documents/Exercise 3/Q3/exampleSyncUpdated2$ ./deadlock_timeout race
Creating thread 1
Creating thread 2
Thread 1 started
will try to join both CS threads unless they deadlock
Thread 2 started
THREAD 2 grabbing resource B @ 1639156765 sec and 725873305 nsec
Thread 2 GOT B
rsrcACnt=0, rsrcBCnt=1
THREAD 2 got B, trying for A @ 1639156765 sec and 725929660 nsec
Thread 2 GOT A @ 1639156765 sec and 725938826 nsec with rc=0
rsrcACnt=1, rsrcBCnt=1
THREAD 2 got B and A
THREAD 2 done
THREAD 1 grabbing resource A @ 1639156765 sec and 725940337 nsec
Thread 1 GOT A
rsrcACnt=1, rsrcBCnt=0
THREAD 1 got A, trying for B @ 1639156765 sec and 726047264 nsec
Thread 1 GOT B @ 1639156765 sec and 726057628 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
```

Figure 3: Deadlock_timeout.c

```
parthishere@nano:~/Documents/Exercise 3/Q3/exampleSyncUpdated2$ ./deadlock_timeout safe
Creating thread 1
Thread 1 started
THREAD 1 grabbing resource A @ 1639156770 sec and 388316845 nsec
Thread 1 GOT A
rsrcACnt=1, rsrcBCnt=0
will sleep
THREAD 1 got A, trying for B @ 1639156771 sec and 388847626 nsec
Thread 1 GOT B @ 1639156771 sec and 389054762 nsec with rc=0
rsrcACnt=1, rsrcBCnt=1
THREAD 1 got A and B
THREAD 1 done
Thread 1 joined to main
Creating thread 2
will try to join both CS threads unless they deadlock
Thread 2 started
THREAD 2 grabbing resource B @ 1639156771 sec and 394419658 nsec
Thread 2 GOT B
rsrcACnt=0, rsrcBCnt=1
will sleep
THREAD 2 got B, trying for A @ 1639156772 sec and 395626115 nsec
Thread 2 GOT A @ 1639156772 sec and 395899761 nsec with rc=0
rsrcACnt=1, rsrcBCnt=1
THREAD 2 got B and A
THREAD 2 done
Thread 2 joined to main
All done
```

Figure 4: Deadlock_timeout.c with different parameter

Like deadlock.c, the deadlock_timeout code can run in 3 different ways. The race and safe conditions are the same as deadlock.c and can be referred to in that section.
In deadlock_timeout.c, both the threads acquire their first locks – thread 1 acquires "rsrcA" and thread 2 acquires "rsrcB". However, both the threads have a timeout before acquiring the second lock (using "pthread_mutex_timedlock()"). This timeout is around 3 seconds from the current time obtained from clock_gettime with CLOCK_REALTIME. In the first portion of the image, we can observe that both the threads have acquired the first lock but are waiting for the second lock. However, thread 2 is not able to acquire the second lock before the timeout which results in it releasing its first lock and exiting which can be seen by "Thread 2 TIMEOUT ERROR" in the code output. Soon after this, thread 1 acquires the second lock and executes the remaining code. In this case, thread 1 is completely executing but thread 2 doesn't because of a timeout error in acquiring the second lock.

Priority inversion happens when a high-priority task must wait for a low-priority task to finish using a resource it needs. If the low-priority task gets interrupted by another high-priority task, the high-priority task must wait even longer until the low-priority task finishes. This delay is called bounded priority inversion, and we can plan for it by adding extra time to the schedule to make sure everything gets done on time. But if a medium-priority task comes along and interrupts the low-priority task, we can't predict how long it will take or how many times it will happen. This unpredictable delay is called unbounded priority inversion.

Following are the outputs and root cause for the example codes:
pthread3.c

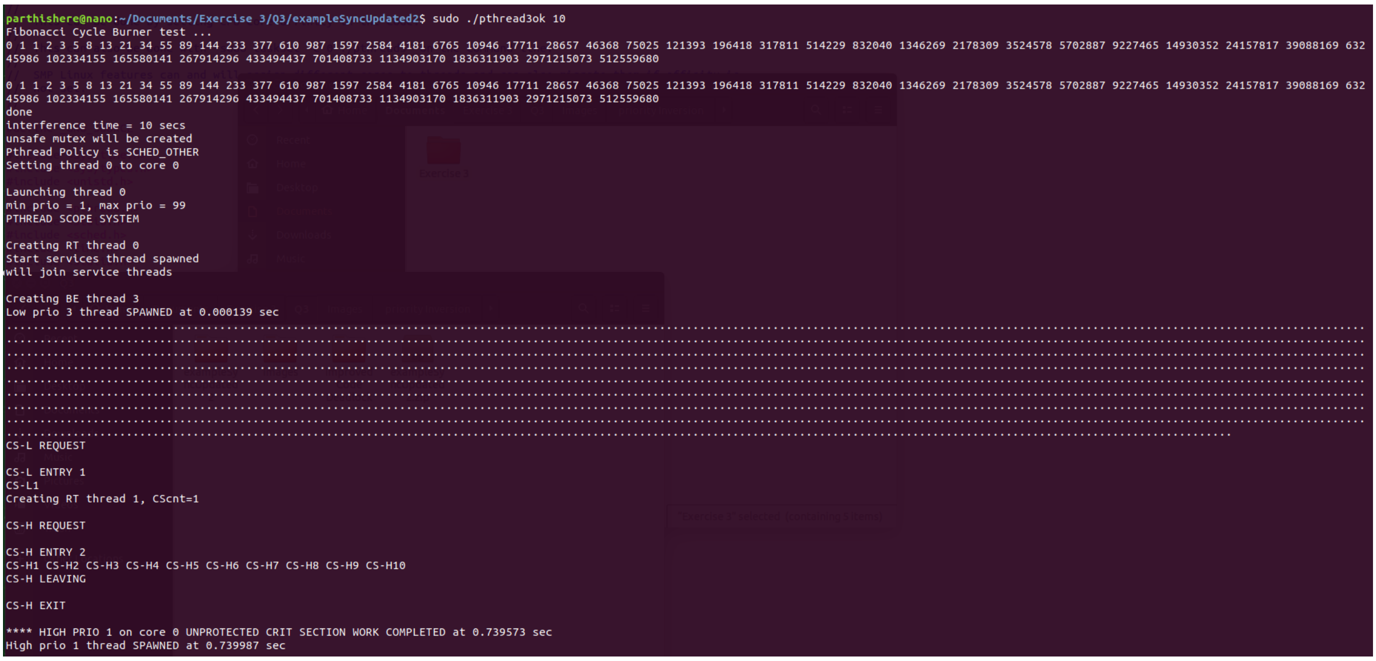


Figure 5: pthread3.c

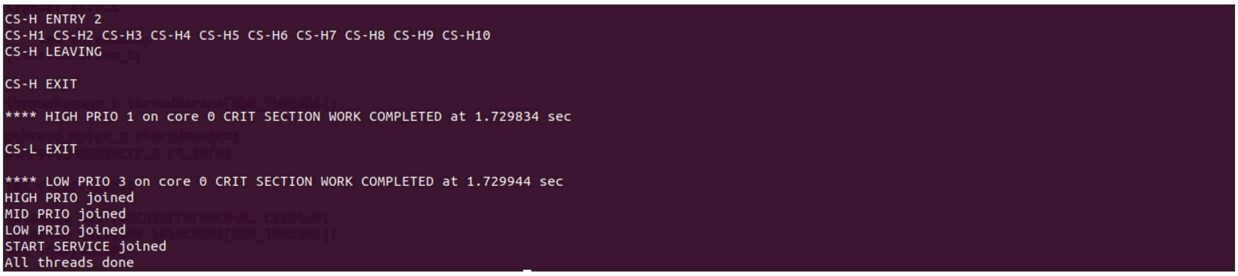


Figure 6: pthread3.c

This code is run with argument 10 which provides the input for interference time. The code has 4 threads – High, mid and low priority threads, and a service thread which schedules their execution. All the threads have their CPU affinity set to the same core. After the service thread, high priority as the name suggests is the highest priority thread followed by mid and low priority threads. Ideally, the highest priority should complete its execution first, followed by mid and then low. In the "startService" function executed by the service thread, the lowest priority thread is created first which enters the critical section in the "criticalSectionTask" function. To ensure that the lowest priority thread enters the critical section, there is a loop in "startService" function which delays the code till the lowest priority thread acquires the lock. The highest priority thread is created next which also executes the "criticalSectionTask" function but is unable to acquire the lock as the lowest priority thread has the lock and hasn't been able to release it. Next, the mid priority thread is created which executes the "simpleTask" function which doesn't have a critical section and executes the Fibonacci series based on the interference time entered as an argument. Since the highest priority is blocked due to unavailability of the resource, the mid priority runs the Fibonacci series 10 times (As seen by "M1 M2 .... M10" in the image) after which the lowest priority runs the Fibonacci series 10 times as "CS_LENGTH" is 10 (As seen by CS-L1 CS-L2 ....CS-L10). In the end, the highest priority thread acquires the lock and runs the Fibonacci series 10 times (As seen by CS-H1 CS-H2 ....CS_H10). This is the unbounded priority inversion where lowest priority thread acquires the lock when it is needed by the highest priority thread and mid priority thread completes its execution which causes further delay.

pthread3amp.c

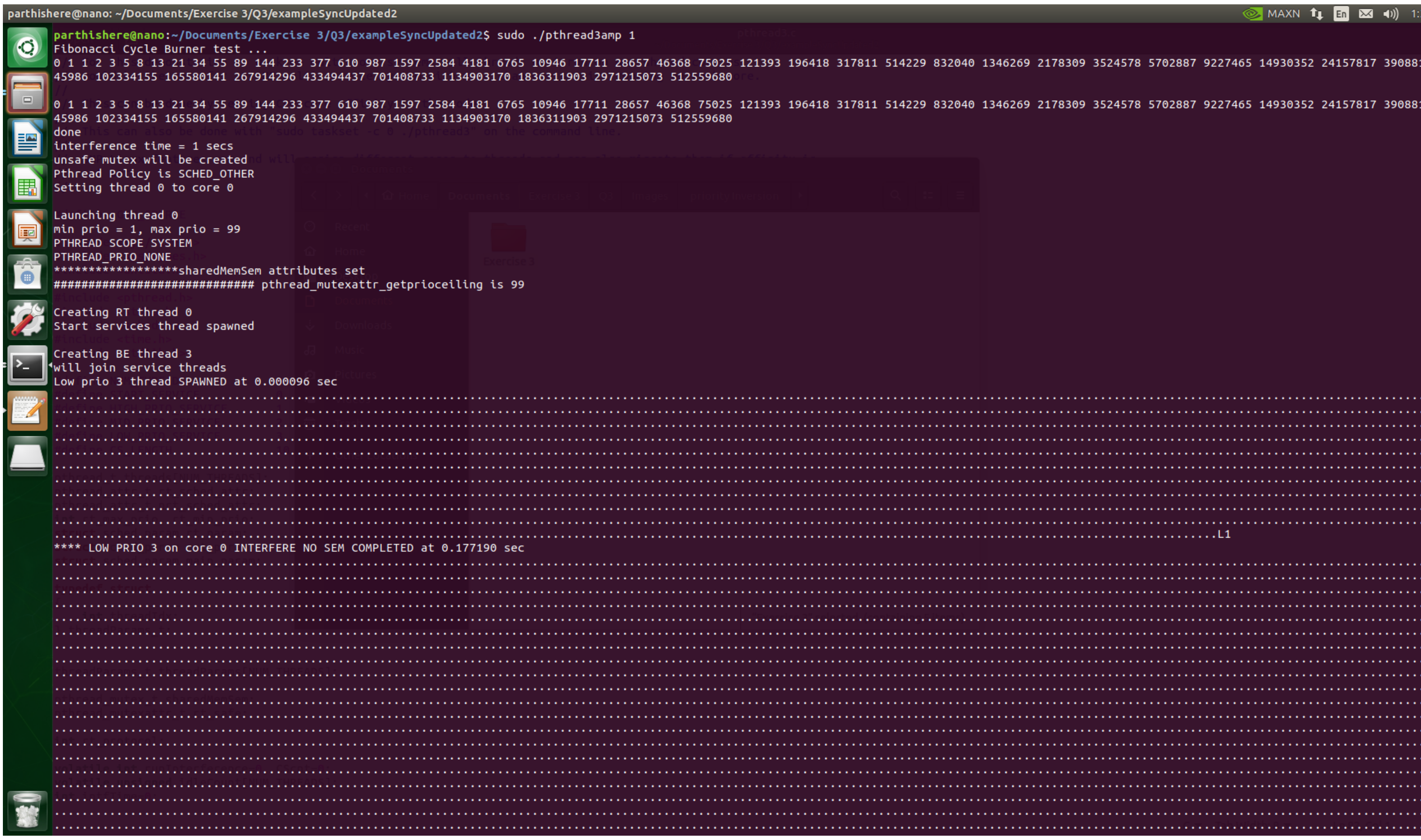


Figure 7: pthread3amp.c

This code is like pthread3.c except in this case, the lowest priority thread executes the "simpleTask" instead of the "criticalSectionTask" function. This would ideally avoid the condition for the unbounded priority inversion. But somehow, in the code, inside the "startService" function, the loop to ensure the lowest priority thread enters the critical section and acquires the lock in "criticalSectionTask" function still exists which causes an infinite spin lock. We can observe from the image that when interference time is set to 1, the lower priority thread executes the Fibonacci series once (L1), but the loop continues indefinitely as indicated by the ". . . . .". Hence, no other thread executes.

pthread3ok.c

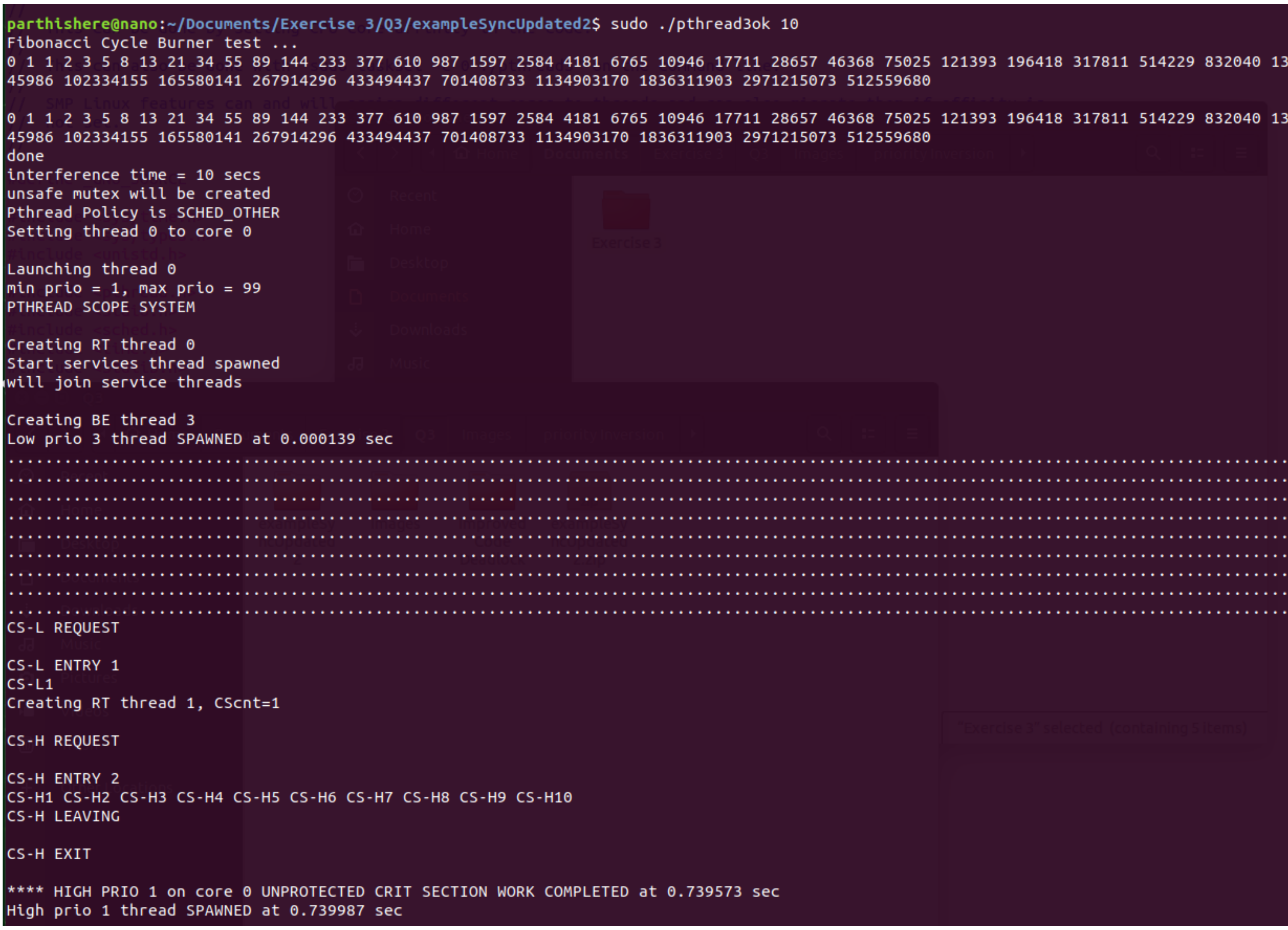


Figure 8: pthread3ok.c

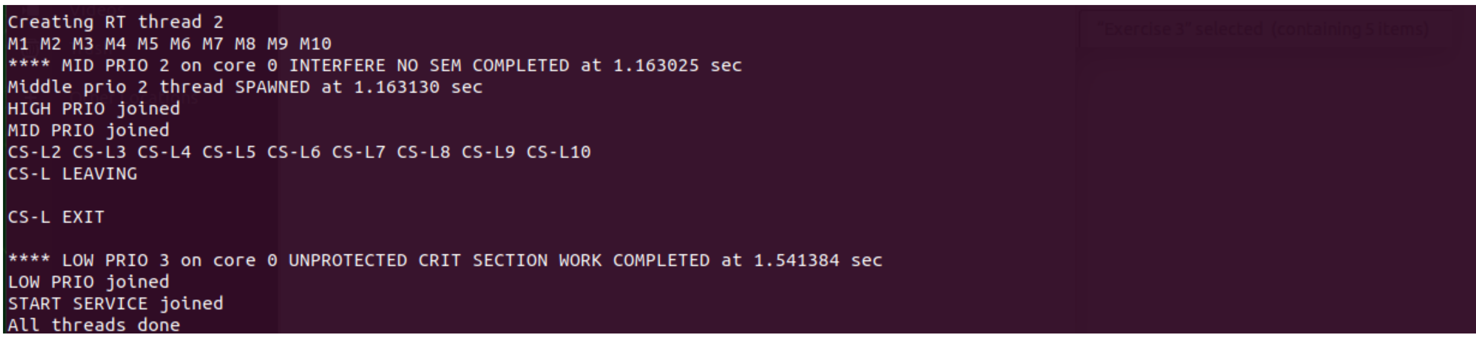
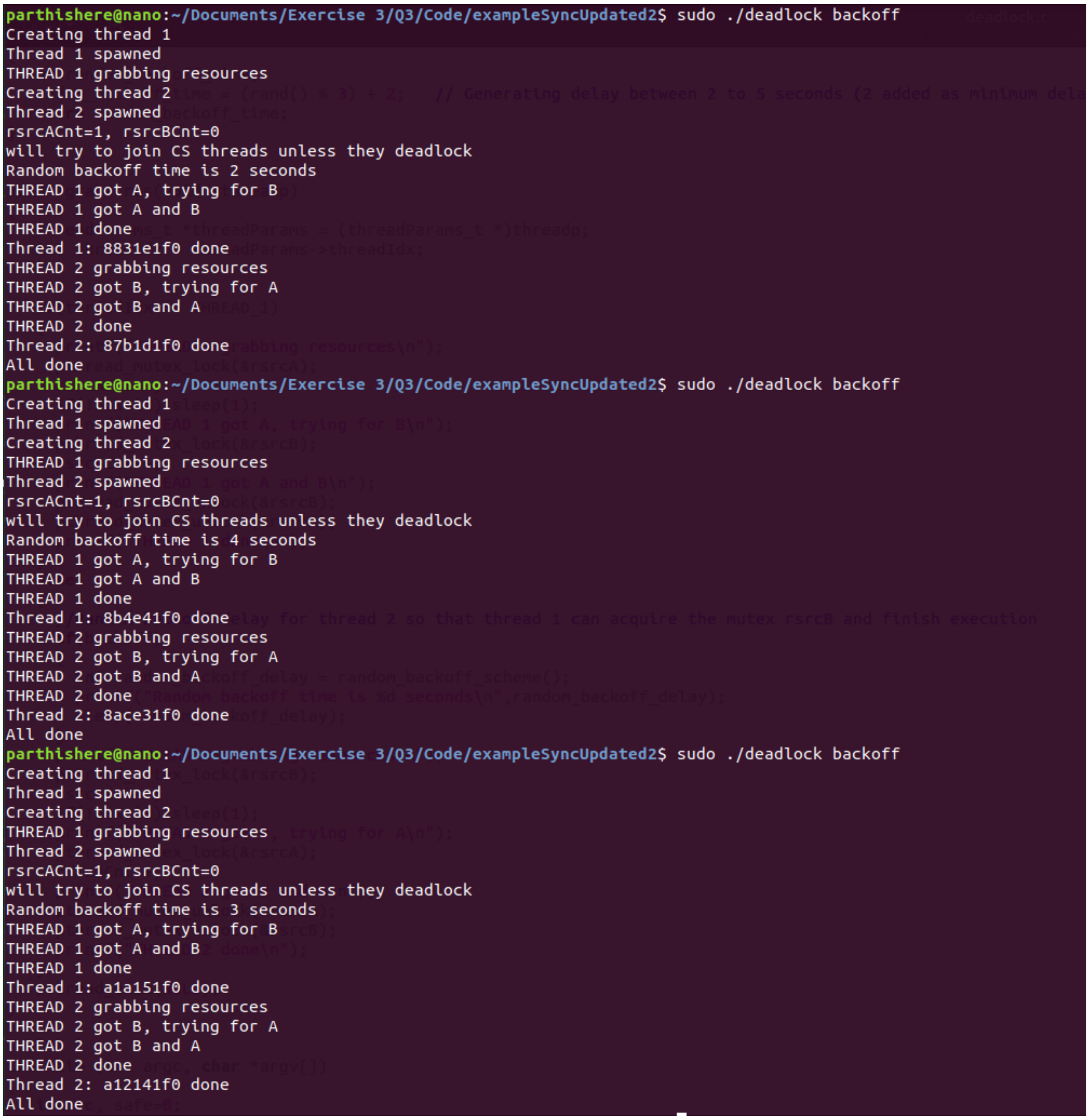


Figure 9: pthread3ok.c

This code is similar to pthread3.c but it solves the problem of unbounded priority inversion by a method which is not the best solution. As observed from the image, we can confirm that the highest priority thread executes first, followed by the mid priority and in the end lowest priority thread. The fix in the code was that in the "criticalSectionTask" function, the critical section was removed by uncommenting the mutex lock and unlock. In this way, the highest priority thread would preempt the lowest priority thread. This is not an advisable solution for unbounded priority inversion as the critical section needs to be protected and if not, it can lead to sequence and data corruption.

To prevent deadlock, we can set a time limit for holding resources. But if both threads release resources simultaneously, it could lead to another problem called live-lock. To avoid this, we use a random back-off scheme, causing one thread to release a resource while the other holds onto both, breaking the deadlock. The output of the updated deadlock.c can be found below:



```
parthishere@nano:~/Documents/Exercise 3/Q3/Code/exampleSyncUpdated2$ sudo ./deadlock backoff
Creating thread 1
Thread 1 spawned
THREAD 1 grabbing resources
Creating thread 2
Thread 2 spawned
rsrcACnt=1, rsrcBCnt=0
will try to join CS threads unless they deadlock
Random backoff time is 2 seconds
THREAD 1 got A, trying for B
THREAD 1 got A and B
THREAD 1 done
Thread 1: 8831e1f0 done
THREAD 2 grabbing resources
THREAD 2 got B, trying for A
THREAD 2 got B and A
THREAD 2 done
Thread 2: 87b1d1f0 done
All done
parthishere@nano:~/Documents/Exercise 3/Q3/Code/exampleSyncUpdated2$ sudo ./deadlock backoff
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
Random backoff time is 4 seconds
THREAD 1 got A, trying for B
THREAD 1 got A and B
THREAD 1 done
Thread 1: 8b4e41f0 done
THREAD 2 grabbing resources
THREAD 2 got B, trying for A
THREAD 2 got B and A
THREAD 2 done
Thread 2: 8ace31f0 done
All done
parthishere@nano:~/Documents/Exercise 3/Q3/Code/exampleSyncUpdated2$ sudo ./deadlock backoff
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
Random backoff time is 3 seconds
THREAD 1 got A, trying for B
THREAD 1 got A and B
THREAD 1 done
Thread 1: a1a151f0 done
THREAD 2 grabbing resources
THREAD 2 got B, trying for A
THREAD 2 got B and A
THREAD 2 done
Thread 2: a12141f0 done
All done
```

Figure 10: deadlock.c

The code was updated to add a random backoff scheme to avoid the deadlock scenario. In the code, a new argument "backoff" was added which would prevent the deadlock using random backoff scheme. A new function "random_backoff_scheme" was created which would return a random delay in seconds (between 2 to 5). A rand() function was used to generate a random number which was divided by 3 and the remainder was added to 2. These numbers, minimum range and the delay were calculated using various random combinations in which deadlock wouldn't occur. In the "grabRsrcs", before the thread 2 execution started, this delay was introduced so that thread 1 could execute completely acquiring both the locks and by the time thread 2 executes, both the locks would be available. As observed from the image, thread 1 executes first and then thread 2. The random backoff time for which thread 2 sleeps is also printed. The code was run multiple times and deadlock was not observed.

(b) **Q: Fix the deadlock so that it does not occur by using a random back-off scheme to resolve. For the unbounded inversion, is there a real fix in Linux – if not, why not?**
Answer: No, there isn't a fix for unbounded priority inversion in Linux without the RT_PREEMPT_PATCH. This patch, now part of the Linux kernel, supports priority inheritance. Without it, Linux lacks a mechanism to adjust the priority of system calls, leaving unbounded priority inversion unresolved.

(c) **Q: What about a patch for the Linux kernel? For example, Linux Kernel.org recommends the RT_PREEMPT Patch, also discussed by the Linux Foundation Realtime Start and this blog, but would this really help? Read about the patch and describe why think it would or would not help with unbounded priority inversion. Based on inversion, does it make sense to simply switch to an RTOS and not use Linux at all for both HRT and SRT services?**
Answer: The RT_PREEMPT_PATCH for the Linux kernel turns the kernel space into a preemptible environment like user space. This means that higher priority tasks can be executed immediately, with a few exceptions. It's especially useful for real-time applications. However, it's crucial to handle CPU variables carefully because the kernel isn't preemptible during certain operations like handling interrupts or holding locks.

This patch allows setting priorities for system calls and interrupts. It introduces the concept of priority inheritance to prevent unbounded priority inversion. When a low-priority task is preempted by a higher-priority task, its priority is temporarily raised to ensure the critical section is executed promptly. This strategy converts unbounded priority inversion to bounded, manageable inversion, improving task scheduling reliability. While the RT_PREEMPT_PATCH doesn't guarantee the complete elimination of priority inversion, it significantly reduces the risk, making sure unbounded priority inversion won't occur. It empowers users to adjust thread priorities, enhancing system responsiveness and predictability.

Moreover, this patch enables preempting kernel locks, critical sections, and interrupts, further mitigating priority inversion. By utilizing priority inheritance, it minimizes the blocking time of high-priority tasks, contributing to the development of robust real-time systems.

However, there are some limitations. For instance, if a low-priority thread inherits a high priority for an extended period, it may impact system performance. Despite these challenges, the RT_PREEMPT_PATCH effectively addresses and resolves the problem of unbounded priority inversion, improving the reliability of real-time systems built on the Linux kernel. Based on inversion, does it make sense to simply switch to an RTOS and not use Linux at all for both HRT and SRT services? Ans. Switching from Linux to an RTOS depends on the application's needs. Linux, with its extensive codebase and community support, isn't optimized for real-time tasks. Even with enhancements like the RT_PREEMPT_PATCH, Linux may struggle with priority inversion, where higher-priority tasks are delayed by lower-priority ones, affecting hard real-time applications. In contrast, RTOS is purpose-built for real-time tasks and typically handles priority inversion more effectively. This makes it suitable for projects with strict real-time requirements, especially those where developing code from scratch is feasible. RTOS provides the precise timing and reliability crucial for hard real-time systems.

However, Linux has its strengths, such as an excessive number of tools and existing code, making it popular for many applications. While Linux, with the RT_PREEMPT_PATCH, can handle soft real-time tasks well, relying solely on it for hard real-time services may pose challenges.

Additionally, the architecture and design of the Linux kernel impact its real-time capabilities. While patches like RT_PREEMPT_PATCH aim to enhance real-time performance, they may not fully resolve all issues, particularly in situations requiring strict real-time guarantees. Thus, for applications needing precise real-time performance, an RTOS may provide a more reliable solution.

The decision to switch to an RTOS depends on various factors, including the criticality of real-time requirements, available resources, and project specifics. While Linux, with appropriate patches, can address some real-time concerns, an RTOS remains the preferred choice for hard real-time services due to its specialized architecture and reliability.

4 Question 4

Q: [15 points] Review POSIX-examples and especially POSIX_MQ_LOOP and build the code related to POSIX message queues and run them to learn about basic use.

(a) **Q: First, re-write the simple message queue demonstration code in heap_mq.c and posix_mq.c so that it uses RT-Linux Pthreads (FIFO) instead of SCHEDULE_OTHER, and then write a brief paragraph describing how the two message queue applications are similar and how they are different. Prove that you got the POSIX message queue features working in Linux on your target board.**

Answer: The code in VxWorks provided by Sam Siewert was ported to RT-Linux Pthreads. For both heap_mq.c and posix_mq.c, 2 threads were created : sender and receiver where both were scheduled with SCHED_FIFO policy with the sender given higher priority than receiver. Both these threads were referencing a message queue "/send_receive_mq" where sender thread was enqueuing the messages in the message queue till maximum messages and then receiver thread was dequeuing one message from the message queue in an infinite loop. Both the threads were configured to use the same core.

heap_mq.c output

```
parthishere@nano:~/Documents/Exercise 3/Q4$ sudo ./heap_mq

Sender Thread Created with rc=0
sender - thread entry

 Receiver Thread Created with rc=0
receiver - thread entry
sender - Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
sender - Sending message of size=8
pthread join send
receiver - awaiting message
sender - message ptr 0x0x7f8c000b20 successfully sent
sender - Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
sender - Sending message of size=8
sender - message ptr 0x0x7f8c001b30 successfully sent
sender - Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
sender - Sending message of size=8
sender - message ptr 0x0x7f8c002b40 successfully sent
sender - Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
sender - Sending message of size=8
sender - message ptr 0x0x7f8c003b50 successfully sent
sender - Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
sender - Sending message of size=8
sender - message ptr 0x0x7f8c004b60 successfully sent
sender - Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
sender - Sending message of size=8
sender - message ptr 0x0x7f8c005b70 successfully sent
sender - Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
sender - Sending message of size=8
sender - message ptr 0x0x7f8c006b80 successfully sent
sender - Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
sender - Sending message of size=8
sender - message ptr 0x0x7f8c007b90 successfully sent
sender - Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
sender - Sending message of size=8
sender - message ptr 0x0x7f8c008ba0 successfully sent
sender - Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
sender - Sending message of size=8
sender - message ptr 0x0x7f8c009bb0 successfully sent
sender - Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
sender - Sending message of size=8
sender - message ptr 0x0x7f8c00abc0 successfully sent
sender - Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
sender - Sending message of size=8
receiver - ptr msg 0x0x7f8c000b20 received with priority = 30, length = 12, id = 999
receiver - Contents of ptr =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
receiver - heap space memory freed
receiver - awaiting message
sender - message ptr 0x0x7f8c00bbd0 successfully sent
sender - Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
sender - Sending message of size=8
receiver - ptr msg 0x0x7f8c001b30 received with priority = 30, length = 12, id = 999
receiver - Contents of ptr =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
receiver - heap space memory freed
```

Figure 11: heapmq.c

As observed from the output, the sender has enqueued 10 messages in the message queue (addresses of the heap memory where buffer data is copied) and the heap addresses and data contained in that memory is printed on terminal. The data contains ASCII characters and an id. The receiver thread dequeues the message queue (first-in message) to obtain the address. This address is copied to a local buffer which is then dereferenced to obtain the ASCII characters and id. From the prints, we can confirm the addresses enqueued by sender thread and dequeued by receiver thread are same and the data pointed by those addresses are same as well. After one message is dequeued, since the sender thread has a higher priority, it preempts the receiver thread and enqueues one more message before going into blocking state where receiver thread then dequeues the next in line message in the message queue. posix_mq.c output

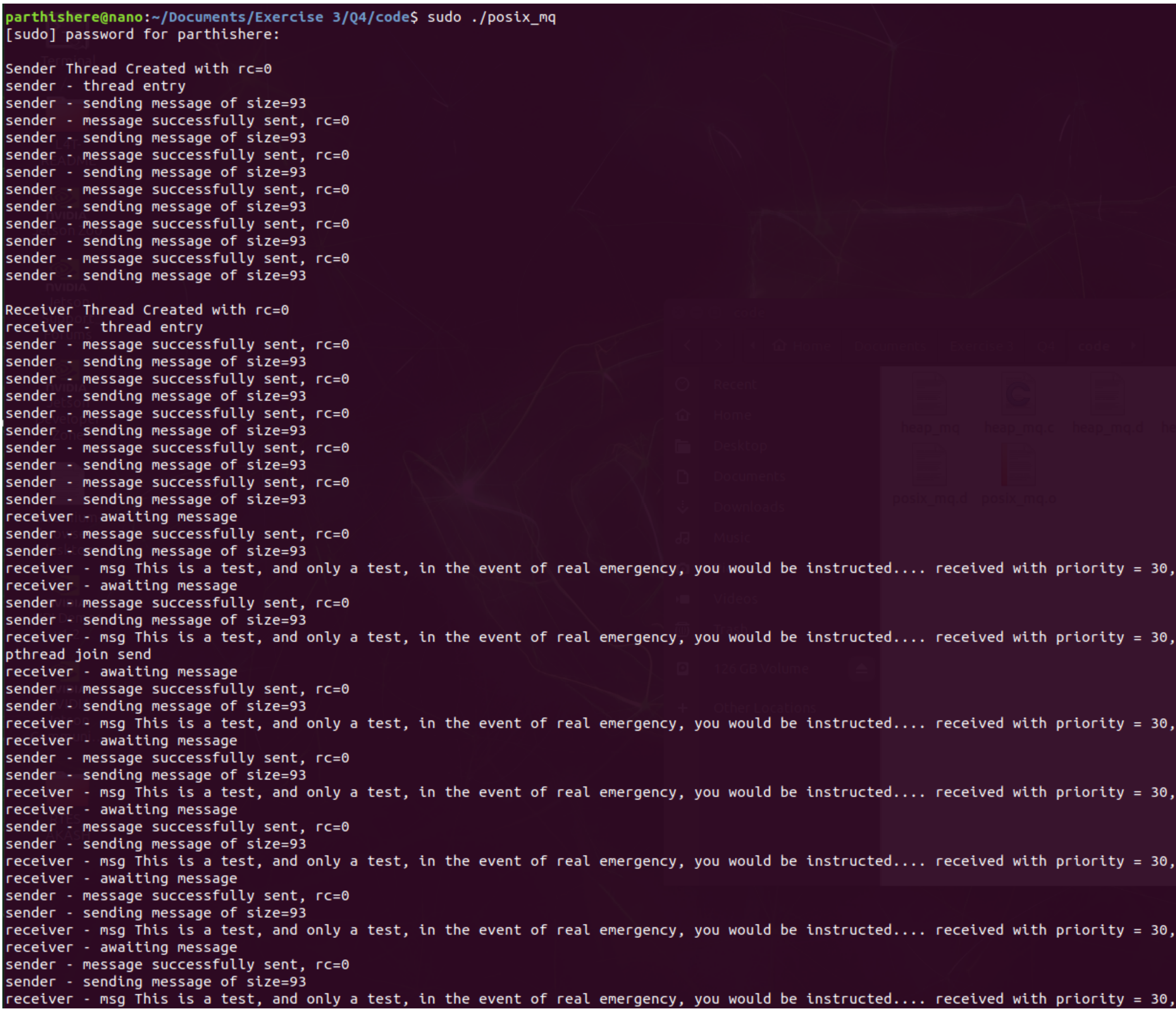


Figure 12: posix_mq.c

Observing the output, the sender has enqueued 10 messages in the message queue (string of fixed size) verified with a print of number of bytes transferred. The receiver thread dequeues the message queue (first-in message) to obtain the address. This address is copied to a local buffer which is then printed as a string. From the prints, we can confirm the string sent by the sender thread and received by the receiver thread are same. After one message is dequeued, since the sender thread has a higher priority, it preempts the receiver thread and enqueues one more message before going into blocking state where receiver thread then dequeues the next in line message in the message queue. Code for heap_mq.c can be found in Appendix 1 and code for posix_mq.c can be found in Appendix 2.

Similarities between both codes

i. Both the codes have 2 threads – sender and receiver which access the same message queue.
ii. Both codes work as expected – The sender enqueues messages in the queue whereas the receiver dequeues the messages. In both cases, the message sent by sender and received by receiver are the same.

iii. In both cases, the sender enqueues 10 messages (max number of messages that can be enqueued in message queue) before going into blocking state after which receiver thread executes and dequeues the first in message in the message queue after which the sender thread preempts the receiver thread. Differences between both codes

iv. In posix_mq.c, the actual data is transferred between the two threads through the message queue. In this case, the data has a restriction to be under the maximum size mentioned in the message queue attributes. In heap_mq.c, the address of the heap is passed instead of actual data. In this case, there is no restriction on the size of the data as the message queue will only contain the address. The receiver thread can dereference the heap address to obtain the data.

v. In posix_mq.c, the buffered message is stored as a static i.e. it is stored in the data segment of the memory whereas in heap_mq.c, the data is stored in heap. The heap is shared by both threads, the sender thread allocates the memory (by malloc) whereas the receiver thread frees the memory after reading the data in it.

(b) **Q: Message queues are often suggested as a way to avoid MUTEX priority inversion. Would you agree that this really circumvents the problem of unbounded inversion? If so why, if not, why not?**

Answer: . I would agree that message queues can effectively circumvent the issue of unbounded priority inversion because of the following reasons:

Priority Messaging: POSIX message queues have the capability to assign priorities to messages, ensuring that higher priority messages are dequeued before lower priority ones. This means that higher priority tasks don't need to wait indefinitely for lower priority tasks to release resources, thus avoiding unbounded priority inversion.

FIFO Logic: If two messages have the same priority, POSIX message queues follow a First-In-First-Out (FIFO) logic. This ensures fairness in message processing and helps prevent scenarios where lower priority tasks in definitely delay higher priority ones.

Thread-Safe Operations: Enqueue and dequeue operations on message queues are thread-safe. This means that multiple threads can safely access the queue without causing conflicts or race conditions.

Resource Management: Message queues provide a mechanism for threads to synchronize their execution by pausing until a message is available in the queue. This prevents threads from spinning needlessly and consuming CPU resources while waiting for shared resources.

In summary, message queues, particularly POSIX message queues, address the issues of unbounded priority inversion by providing priority-based message handling and ensuring thread-safe operations. They facilitate efficient communication between threads and help maintain system responsiveness even in scenarios involving shared resources and differing task priorities.

5 Question 5

Q: [20 points] Watchdog timers, timeouts and timer services – First, read this overview of the Linux Watchdog Daemon and the Linux manual page on the watchdog daemon - https://linux.die.net/man/8/watchdog . Also see the Watchdog Explained.

(a) **Q: Describe how it might be used if software caused an indefinite deadlock.**

Answer: The watchdog utility, as described in the Linux manual, is a daemon that monitors and/or restarts a system if it detects that the system has become unresponsive. It operates by continually kicking (updating) a watchdog timer; if the timer expires (i.e., it is not kicked within a certain period), the system is considered to have failed and is subsequently rebooted. This can be particularly useful in situations where software has caused the system to enter an indefinite deadlock, making it unresponsive.

Using watchdog in the Context of a Deadlock An indefinite deadlock occurs when two or more processes hold resources and each process is waiting for the other to release a resource, causing all processes to be stuck indefinitely. In critical systems, such deadlocks can lead to severe consequences, including loss of data, unavailability of services, and potential hardware damage due to overheating or overuse.

Here's how watchdog can be utilized to mitigate the impact of such a deadlock:

- Detection of Unresponsiveness: watchdog is configured to detect system unresponsiveness or failure to execute critical tasks within expected timeframes.
- Heartbeat Mechanism: For software that might cause a deadlock, implementing a heartbeat mechanism that periodically signals watchdog can be an effective strategy. The software must regularly kick the watchdog timer to prevent the system from being rebooted. If the software enters a deadlock and fails to kick the timer, watchdog will reboot the system once the timer expires.
- Recovery Actions: Upon detecting a failure (e.g., a timeout indicating a deadlock), watchdog can take predefined actions to recover the system. Typically, this involves rebooting the system to resolve the deadlock and restore service.

(b) **Q: Next, to explore timeouts, use your code from #2 and create a thread that waits on a MUTEX semaphore for up to 10 seconds and then un-blocks and prints out "No new data available at ¡time¿" and then loops back to wait for a data update again. Use a variant of the pthread_mutex_lock called pthread_mutex_timedlock to solve this programming problem.**

Answer:

Data Structures:

i. Data Structures:
   - NavigationState: Holds navigation-related data (latitude, longitude, altitude, roll, pitch, yaw) and a timestamp.
   - ThreadArgs_t: Designed to pass arguments to threads, though not extensively used in the provided snippet.
   - RmTask_t: Contains parameters for configuring threads, including scheduling properties, function pointers for thread routines, thread IDs, and CPU affinity settings.

ii. Global Variables:
   - mutex and state_mutex: Mutexes used for synchronization, although only state_mutex is utilized for protecting access to nav_state. nav_state and nav_state_shouldbe: Instances of NavigationState for holding current and expected navigation data. Thread Functions:
   - update_thread: Updates nav_state in a loop with simulated data and timestamps, protected by state_mutex to ensure exclusive access during updates. read_thread: Attempts to acquire state_mutex with a timeout using pthread_mutex_timedlock and reads nav_state if successful. It prints a message if it times out, indicating no new data was available for reading.

Use of Mutex for Synchronization

The mutex state_mutex is crucial for ensuring that updates and reads to the shared nav_state structure are performed atomically, avoiding potential data races. Here's how it's used:

- In update_thread: Before updating nav_state, the thread locks state_mutex. This ensures that read_thread cannot simultaneously access nav_state, potentially reading inconsistent data. After the update, it unlocks state_mutex, allowing other threads to access nav_state.

- In read_thread: It tries to lock state_mutex within a specific timeout period. If successful, it means update_thread is not currently updating nav_state, and it can safely read consistent data. If it times out, it indicates that update_thread has held the lock for too long, suggesting no new data is available for reading within the expected timeframe.

here is the output of the commenting out the 10ms delay so that both tasks are synchronized

```
Pthread Policy is SCHED_FIFO
Setting thread 0 to core 1
Setting thread 1 to core 1
Updated reading
Yaw: 1.000000, Roll: 0.000000, Pitch: 1.000000, Latitude 0.000000, Longitude 0.000000, Altitude 0.000000
Reading data:
Yaw: 1.000000, Roll: 0.000000, Pitch: 1.000000, Latitude 0.000000, Longitude 0.000000, Altitude 0.000000
Time: tv_sec: 1710050760, tv_nsec: 449989167
Updated reading
Yaw: 0.540302, Roll: 0.841471, Pitch: 0.540302, Latitude 1.000000, Longitude 0.500000, Altitude 0.250000
Updated reading
Yaw: -0.416147, Roll: 0.909297, Pitch: -0.653644, Latitude 2.000000, Longitude 1.000000, Altitude 0.500000
Updated reading
Yaw: -0.989992, Roll: 0.141120, Pitch: -0.911130, Latitude 3.000000, Longitude 1.500000, Altitude 0.750000
Updated reading
Yaw: -0.653644, Roll: -0.756802, Pitch: -0.957659, Latitude 4.000000, Longitude 2.000000, Altitude 1.000000
Updated reading
Yaw: 0.283662, Roll: -0.958924, Pitch: 0.991203, Latitude 5.000000, Longitude 2.500000, Altitude 1.250000
Updated reading
Yaw: 0.960170, Roll: -0.279415, Pitch: -0.127964, Latitude 6.000000, Longitude 3.000000, Altitude 1.500000
Updated reading
Yaw: 0.753902, Roll: 0.656987, Pitch: 0.300593, Latitude 7.000000, Longitude 3.500000, Altitude 1.750000
Updated reading
Yaw: -0.145500, Roll: 0.989358, Pitch: 0.391857, Latitude 8.000000, Longitude 4.000000, Altitude 2.000000
Updated reading
Yaw: -0.911130, Roll: 0.412118, Pitch: 0.776686, Latitude 9.000000, Longitude 4.500000, Altitude 2.250000
Reading data:
Yaw: -0.911130, Roll: 0.412118, Pitch: 0.776686, Latitude 9.000000, Longitude 4.500000, Altitude 2.250000
Time: tv_sec: 1710050769, tv_nsec: 450445427
Updated reading
Yaw: -0.839072, Roll: -0.544021, Pitch: 0.862319, Latitude 10.000000, Longitude 5.000000, Altitude 2.500000
Updated reading
Yaw: 0.004426, Roll: -0.999990, Pitch: -0.048664, Latitude 11.000000, Longitude 5.500000, Altitude 2.750000
Updated reading
Yaw: 0.843854, Roll: -0.536573, Pitch: 0.871147, Latitude 12.000000, Longitude 6.000000, Altitude 3.000000
Updated reading
Yaw: 0.907447, Roll: 0.420167, Pitch: 0.798496, Latitude 13.000000, Longitude 6.500000, Altitude 3.250000
Updated reading
Yaw: 0.136737, Roll: 0.990607, Pitch: 0.342466, Latitude 14.000000, Longitude 7.000000, Altitude 3.500000
Updated reading
Yaw: -0.759688, Roll: 0.650288, Pitch: 0.367319, Latitude 15.000000, Longitude 7.500000, Altitude 3.750000
Updated reading
Yaw: -0.957659, Roll: -0.287903, Pitch: -0.039791, Latitude 16.000000, Longitude 8.000000, Altitude 4.000000
Updated reading
Yaw: -0.275163, Roll: -0.961397, Pitch: 0.999648, Latitude 17.000000, Longitude 8.500000, Altitude 4.250000
Updated reading
Yaw: 0.660317, Roll: -0.750987, Pitch: -0.914730, Latitude 18.000000, Longitude 9.000000, Altitude 4.500000
Updated reading
Yaw: 0.988705, Roll: 0.149877, Pitch: -0.960179, Latitude 19.000000, Longitude 9.500000, Altitude 4.750000
Reading data:
Yaw: 0.988705, Roll: 0.149877, Pitch: -0.960179, Latitude 19.000000, Longitude 9.500000, Altitude 4.750000
Time: tv_sec: 1710050779, tv_nsec: 450942449
Updated reading
Yaw: 0.408082, Roll: 0.912945, Pitch: -0.525296, Latitude 20.000000, Longitude 10.000000, Altitude 5.000000
Updated reading
Yaw: -0.547729, Roll: 0.836656, Pitch: 0.383671, Latitude 21.000000, Longitude 10.500000, Altitude 5.250000
Updated reading
Yaw: -0.999961, Roll: -0.008851, Pitch: 0.981100, Latitude 22.000000, Longitude 11.000000, Altitude 5.500000
```

Figure 13: Tasks are in synchronization

Now enabling 10ms delay while updating tasks so that other task have to wait for the mutex to be unlocked by other task

```
gcc -g  -c pthread_mutex_timelock.c -o pthread_mutex_timelock.o -lm
gcc -g  -o program pthread_mutex_timelock.o -lpthread -lrt -lm
Runing Executable
This system has 12 processors configured and 12 processors available.
Before adjustments to scheduling policy:
Pthread Policy is SCHED_OTHER
After adjustments to scheduling policy:
Pthread Policy is SCHED_FIFO
Setting thread 0 to core 1
Setting thread 1 to core 1
Updated reading
Yaw: 1.000000, Roll: 0.000000, Pitch: 1.000000, Latitude 0.000000, Longitude 0.000000, Altitude 0.000000
No new data available at 1710050970 seconds
Updated reading
Yaw: 0.540302, Roll: 0.841471, Pitch: 0.540302, Latitude 1.000000, Longitude 0.500000, Altitude 0.250000
Reading data:
Yaw: 0.540302, Roll: 0.841471, Pitch: 0.540302, Latitude 1.000000, Longitude 0.500000, Altitude 0.250000
Time: tv_sec: 1710050972, tv_nsec: 736386317
Updated reading
Yaw: -0.416147, Roll: 0.909297, Pitch: -0.653644, Latitude 2.000000, Longitude 1.000000, Altitude 0.500000
^Cmake: *** [Makefile:24: run] Interrupt
```

Figure 14: Tasks are not in synchronization

6 Question 6

Q: [10 points] Demonstrate the results and answer questions from the TA regarding the code you developed in #2, #3, #4, and #5.

7 References

1. ECEN 5623 Lecture slides material and example codes.
2. REAL-TIME EMBEDDED COMPONENTS AND SYSTEMS with LINUX and RTOS, Sam Siewert John Pratt (Chapter 6, 7 & 8).
3. Exercise 3 requirements included links and documentation.

Appendices

A C Code for the Implementation

answers/Code_Q3_4/Q3/exampleSyncUpdated2/deadlock.c

```c
/*
 * Author: Sam Siewert
 * Modified by: Shashank and Parth
 * Description: Added random backoff scheme to avoid deadlock
 */

#include <pthread.h>
#include <stdio.h>
#include <sched.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define NUM_THREADS 2
#define THREAD_1 0
#define THREAD_2 1

typedef struct
{
    int threadIdx;
} threadParams_t;


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

struct sched_param nrt_param;

// On the Raspberry Pi, the MUTEX semaphores must be statically initialized
//
// This works on all Linux platforms, but dynamic initialization does not work
// on the R-Pi in particular as of June 2020.
//
pthread_mutex_t rsrcA = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t rsrcB = PTHREAD_MUTEX_INITIALIZER;

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

/*********************Random back off scheme to avoid deadlock********************/
int random_backoff_scheme(void)
{
  int random_backoff_time;
  random_backoff_time = (rand() % 3) + 2;    // Generating delay between 2 to 5 seconds (2 added as minimum delay needed is 2 to avoid deadlock)
  return random_backoff_time;
}


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

```c


   if(threadIdx == THREAD_1)
   {
     printf("THREAD 1 grabbing resources\n");
     pthread_mutex_lock(&rsrcA);
     rsrcACnt++;
     if(!noWait) sleep(1);
     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
   {
     //Random backoff delay for thread 2 so that thread 1 can acquire the mutex rsrcB and finish execution
     if(backoff)
     {
       int random_backoff_delay = random_backoff_scheme();
       printf("Random backoff time is %d seconds\n",random_backoff_delay);
       sleep(random_backoff_delay);
     }
     printf("THREAD 2 grabbing resources\n");
     pthread_mutex_lock(&rsrcB);
     rsrcBCnt++;
     if(!noWait) sleep(1);
     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, backoff=0;

   srand(time(NULL)); //Initialize random number generator

   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 if(strncmp("backoff", argv[1], 7) == 0)
        backoff=1;
      else
        printf("Will set up unsafe deadlock scenario\n");
    }
    else
    {
      printf("Usage: deadlock [safe|race|unsafe]\n");
    }


    printf("Creating thread %d\n", THREAD_1+1);
    threadParams[THREAD_1].threadIdx=THREAD_1;
    rc = pthread_create(&threads[0], NULL, grabRsrcs, (void *)&threadParams[THREAD_1]);
    if (rc) {printf("ERROR; pthread_create() rc is %d\n", rc); perror(NULL); exit(-1);}
    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+1);
    threadParams[THREAD_2].threadIdx=THREAD_2;
    rc = pthread_create(&threads[1], NULL, grabRsrcs, (void *)&threadParams[THREAD_2]);
    if (rc) {printf("ERROR; pthread_create() rc is %d\n", rc); perror(NULL); exit(-1);}
    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);
}

```

answers/Code_Q3_4/Q4/heap_mq.c

```c
 /*
  * Author: Sam Siewart for heap_mq.c code in Exercise3/Posix_MQ_loop
  * Modified by: Shashank and Parth
  * References:
  * 1. Sam Siewert - 10/14/97 heap_mq.c - vxWorks code
  * 2. heap_mq.c code in Exercise3/Posix_MQ_loop used as the base
  */

#define _GNU_SOURCE
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <pthread.h>
#include <mqueue.h>
#include <unistd.h>

// On Linux the file systems slash is needed
#define SNDRCV_MQ "/send_receive_mq"

#define ERROR (-1)

#define NUM_CPUS (1)

pthread_t th_receive, th_send; // create threads
pthread_attr_t attr_receive, attr_send;
struct sched_param param_receive, param_send;

static char imagebuff[4096];
struct mq_attr mq_attr;
mqd_t mymq;

/* receives pointer to heap, reads it, and deallocate heap memory */
void *receiver(void *arg)
{
  void *buffptr;
  char buffer[sizeof(void *)+sizeof(int)];
  int prio;
  int nbytes;
  int id;

  cpu_set_t cpuset;
  CPU_ZERO(&cpuset);

  printf("receiver - thread entry\n");

  /* read oldest, highest priority msg from the message queue until empty */
  while(1)
  {
    printf("receiver - awaiting message\n");

    if((nbytes = mq_receive(mymq, buffer, (size_t)(sizeof(void *)+sizeof(int)), &prio)) == ERROR)
    {
```

```c
      perror("mq_receive");
    }
    else
    {
      memcpy(&buffptr, &buffer, sizeof(void *));
      memcpy((void *)&id, &(buffer[sizeof(void *)]), sizeof(int));
      printf("receiver - ptr msg 0x%p received with priority = %d, length = %d, id = %d\n", buffptr, prio, nbytes, id);
      printf("receiver - Contents of ptr = \n%s\n", (char *)buffptr);

      free(buffptr);
      printf("receiver - heap space memory freed\n");
    }
  }
}

/*send pointer to heap which points to the data in imagebuff*/
void *sender(void *arg)
{
  char buffer[sizeof(void *)+sizeof(int)];
  void *buffptr;
  int prio;
  int nbytes;
  int id = 999;

  cpu_set_t cpuset;
  CPU_ZERO(&cpuset);

  printf("sender - thread entry\n");

  while(1)
  {
    buffptr = (void *)malloc(sizeof(imagebuff));
    strcpy(buffptr, imagebuff);
    printf("sender - Message to send = %s\n", (char *)buffptr);
    printf("sender - Sending message of size=%d\n", sizeof(buffptr));

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

      if((nbytes = mq_send(mymq, buffer, (size_t)(sizeof(void *)+sizeof(int)), 30)) == ERROR)
      {
        perror("mq_send");
      }
      else
      {
        printf("sender - message ptr 0x%p successfully sent\n", buffptr);
      }
  }

}

/*Fills imagebuff with ASCII data */
void fillbuffer(void)
{
  int i, j;
```

```c
  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';
}


void main(void)
{
  int i=0, rc=0;

  cpu_set_t cpuset;
  CPU_ZERO(&cpuset);
  for(i=0; i < NUM_CPUS; i++)
      CPU_SET(i, &cpuset);

  fillbuffer();

  /* setup common message q attributes */
  mq_attr.mq_maxmsg = 10;
  mq_attr.mq_msgsize = sizeof(void *)+sizeof(int);

  mq_attr.mq_flags = 0;

  mq_unlink(SNDRCV_MQ);  //Unlink if the previous message queue exists

  mymq = mq_open(SNDRCV_MQ, O_CREAT|O_RDWR, S_IRWXU, &mq_attr);
  if(mymq == (mqd_t)ERROR)
  {
    perror("mq_open");
  }

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

  //creating prioritized thread

  //initialize  with default atrribute
  rc = pthread_attr_init(&attr_receive);
  //specific scheduling for Receiving
  rc = pthread_attr_setinheritsched(&attr_receive, PTHREAD_EXPLICIT_SCHED);
  rc = pthread_attr_setschedpolicy(&attr_receive, SCHED_FIFO);
  rc=pthread_attr_setaffinity_np(&attr_receive, sizeof(cpu_set_t), &cpuset);
  param_receive.sched_priority = rt_min_prio;
  pthread_attr_setschedparam(&attr_receive, &param_receive);

```

```
  //initialize  with default atrribute
  rc = pthread_attr_init(&attr_send);
  //specific scheduling for Sending
  rc = pthread_attr_setinheritsched(&attr_send, PTHREAD_EXPLICIT_SCHED);
  rc = pthread_attr_setschedpolicy(&attr_send, SCHED_FIFO);
  rc=pthread_attr_setaffinity_np(&attr_send, sizeof(cpu_set_t), &cpuset);  //SC Added
  param_send.sched_priority = rt_max_prio;
  pthread_attr_setschedparam(&attr_send, &param_send);

  if((rc=pthread_create(&th_send, &attr_send, sender, NULL)) == 0)
  {
    printf("\n\rSender Thread Created with rc=%d\n\r", rc);
  }
  else
  {
    perror("\n\rFailed to Make Sender Thread\n\r");
    printf("rc=%d\n", rc);
  }

  if((rc=pthread_create(&th_receive, &attr_receive, receiver, NULL)) == 0)
  {
    printf("\n\r Receiver Thread Created with rc=%d\n\r", rc);
  }
  else
  {
    perror("\n\r Failed Making Reciever Thread\n\r");
    printf("rc=%d\n", rc);
  }

  printf("pthread join send\n");
  pthread_join(th_send, NULL);

  printf("pthread join receive\n");
  pthread_join(th_receive, NULL);
}
```

answers/Code_Q3_4/Q4/posix_mq.c

```c
 /*
  * Author: Sam Siewart for posix_mq.c code in Exercise3/Posix_MQ_loop
  * Modified by: Shashank and Parth
  * References:
  * 1. Sam Siewert - 10/14/97 posix_mq.c - vxWorks code
  * 2. posix_mq.c code in Exercise3/Posix_MQ_loop used as the base
  */

#define _GNU_SOURCE
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <pthread.h>
#include <mqueue.h>
#include <unistd.h>

// On Linux the file systems slash is needed
#define SNDRCV_MQ "/send_receive_mq"

#define MAX_MSG_SIZE 128
#define ERROR (-1)

#define NUM_CPUS (1)

pthread_t th_receive, th_send; // create threads
pthread_attr_t attr_receive, attr_send;
struct sched_param param_receive, param_send;

struct mq_attr mq_attr;
mqd_t mymq;

static char canned_msg[] = "This is a test, and only a test, in the event of real emergency, you would be instructed...."; // Message to be sent

/* receives pointer to heap, reads it, and deallocate heap memory */
void *receiver(void *arg)
{
  char buffer[MAX_MSG_SIZE];
  int prio;
  int nbytes;

  cpu_set_t cpuset;
  CPU_ZERO(&cpuset);

  printf("receiver - thread entry\n");

  while(1)
  {
    printf("receiver - awaiting message\n");

    if((nbytes = mq_receive(mymq, buffer, MAX_MSG_SIZE, &prio)) == ERROR)
    {
      perror("mq_receive");
```

```c
    }
    else
    {
      buffer[nbytes ] = '\0';
      printf("receiver - msg %s received with priority = %d, nbytes = %d\n", buffer, prio, nbytes);
    }
  }
}

/*send the data in the buffer*/
void *sender(void *arg)
{
  int prio;
  int rc;

  cpu_set_t cpuset;
  CPU_ZERO(&cpuset);

  printf("sender - thread entry\n");

  while(1)
  {
    printf("sender - sending message of size=%d\n", sizeof(canned_msg));

    if((rc = mq_send(mymq, canned_msg, sizeof(canned_msg), 30)) == ERROR)
    {
      perror("mq_send");
    }
    else
    {
      printf("sender - message successfully sent, rc=%d\n", rc);
    }
  }
}

void main(void)
{
  int i=0, rc=0;

  cpu_set_t cpuset;
  CPU_ZERO(&cpuset);
  for(i=0; i < NUM_CPUS; i++)
      CPU_SET(i, &cpuset);

  /* setup common message q attributes */
  mq_attr.mq_maxmsg = 10;
  mq_attr.mq_msgsize = MAX_MSG_SIZE;

  mq_attr.mq_flags = 0;

  mq_unlink(SNDRCV_MQ);  //Unlink if the previous message queue exists

  mymq = mq_open(SNDRCV_MQ, O_CREAT|O_RDWR, S_IRWXU, &mq_attr);
  if(mymq == (mqd_t)ERROR)
  {
```

```c
    perror("mq_open");
  }

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

  //initialize  with default atrribute
  rc = pthread_attr_init(&attr_receive);
  //specific scheduling for Receiving
  rc = pthread_attr_setinheritsched(&attr_receive, PTHREAD_EXPLICIT_SCHED);
  rc = pthread_attr_setschedpolicy(&attr_receive, SCHED_FIFO);
  rc=pthread_attr_setaffinity_np(&attr_receive, sizeof(cpu_set_t), &cpuset);
  param_receive.sched_priority = rt_min_prio;
  pthread_attr_setschedparam(&attr_receive, &param_receive);

  //initialize  with default atrribute
  rc = pthread_attr_init(&attr_send);
  //specific scheduling for Sending
  rc = pthread_attr_setinheritsched(&attr_send, PTHREAD_EXPLICIT_SCHED);
  rc = pthread_attr_setschedpolicy(&attr_send, SCHED_FIFO);
  rc=pthread_attr_setaffinity_np(&attr_send, sizeof(cpu_set_t), &cpuset);
  param_send.sched_priority = rt_max_prio;
  pthread_attr_setschedparam(&attr_send, &param_send);

  if((rc=pthread_create(&th_send, &attr_send, sender, NULL)) == 0)
  {
    printf("\n\rSender Thread Created with rc=%d\n\r", rc);
  }
  else
  {
    perror("\n\rFailed to Make Sender Thread\n\r");
    printf("rc=%d\n", rc);
  }

  if((rc=pthread_create(&th_receive, &attr_receive, receiver, NULL)) == 0)
  {
    printf("\n\rReceiver Thread Created with rc=%d\n\r", rc);
  }
  else
  {
    perror("\n\rFailed Making Reciever Thread\n\r");
    printf("rc=%d\n", rc);
  }

  printf("pthread join send\n");
  pthread_join(th_send, NULL);

  printf("pthread join receive\n");
  pthread_join(th_receive, NULL);

}

```

answers/Code_Q3_4/Q4/posix_mq.c

```c
 /*
  * Author: Sam Siewart for posix_mq.c code in Exercise3/Posix_MQ_loop
  * Modified by: Shashank and Parth
  * References:
  * 1. Sam Siewert - 10/14/97 posix_mq.c - vxWorks code
  * 2. posix_mq.c code in Exercise3/Posix_MQ_loop used as the base
  */

#define _GNU_SOURCE
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <pthread.h>
#include <mqueue.h>
#include <unistd.h>

// On Linux the file systems slash is needed
#define SNDRCV_MQ "/send_receive_mq"

#define MAX_MSG_SIZE 128
#define ERROR (-1)

#define NUM_CPUS (1)

pthread_t th_receive, th_send; // create threads
pthread_attr_t attr_receive, attr_send;
struct sched_param param_receive, param_send;

struct mq_attr mq_attr;
mqd_t mymq;

static char canned_msg[] = "This is a test, and only a test, in the event of real emergency, you would be instructed...."; // Message to be sent

/* receives pointer to heap, reads it, and deallocate heap memory */
void *receiver(void *arg)
{
  char buffer[MAX_MSG_SIZE];
  int prio;
  int nbytes;

  cpu_set_t cpuset;
  CPU_ZERO(&cpuset);

  printf("receiver - thread entry\n");

  while(1)
  {
    printf("receiver - awaiting message\n");

    if((nbytes = mq_receive(mymq, buffer, MAX_MSG_SIZE, &prio)) == ERROR)
    {
      perror("mq_receive");
```

```
    }
    else
    {
      buffer[nbytes ] = '\0';
      printf("receiver - msg %s received with priority = %d, nbytes = %d\n", buffer, prio, nbytes);
    }
  }
}

/*send the data in the buffer*/
void *sender(void *arg)
{
  int prio;
  int rc;

  cpu_set_t cpuset;
  CPU_ZERO(&cpuset);

  printf("sender - thread entry\n");

  while(1)
  {
    printf("sender - sending message of size=%d\n", sizeof(canned_msg));

    if((rc = mq_send(mymq, canned_msg, sizeof(canned_msg), 30)) == ERROR)
    {
      perror("mq_send");
    }
    else
    {
      printf("sender - message successfully sent, rc=%d\n", rc);
    }
  }
}

void main(void)
{
  int i=0, rc=0;

  cpu_set_t cpuset;
  CPU_ZERO(&cpuset);
  for(i=0; i < NUM_CPUS; i++)
      CPU_SET(i, &cpuset);

  /* setup common message q attributes */
  mq_attr.mq_maxmsg = 10;
  mq_attr.mq_msgsize = MAX_MSG_SIZE;

  mq_attr.mq_flags = 0;

  mq_unlink(SNDRCV_MQ);  //Unlink if the previous message queue exists

  mymq = mq_open(SNDRCV_MQ, O_CREAT|O_RDWR, S_IRWXU, &mq_attr);
  if(mymq == (mqd_t)ERROR)
  {
```

```c
    perror("mq_open");
  }

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

  //initialize  with default atrribute
  rc = pthread_attr_init(&attr_receive);
  //specific scheduling for Receiving
  rc = pthread_attr_setinheritsched(&attr_receive, PTHREAD_EXPLICIT_SCHED);
  rc = pthread_attr_setschedpolicy(&attr_receive, SCHED_FIFO);
  rc=pthread_attr_setaffinity_np(&attr_receive, sizeof(cpu_set_t), &cpuset);
  param_receive.sched_priority = rt_min_prio;
  pthread_attr_setschedparam(&attr_receive, &param_receive);

  //initialize  with default atrribute
  rc = pthread_attr_init(&attr_send);
  //specific scheduling for Sending
  rc = pthread_attr_setinheritsched(&attr_send, PTHREAD_EXPLICIT_SCHED);
  rc = pthread_attr_setschedpolicy(&attr_send, SCHED_FIFO);
  rc=pthread_attr_setaffinity_np(&attr_send, sizeof(cpu_set_t), &cpuset);
  param_send.sched_priority = rt_max_prio;
  pthread_attr_setschedparam(&attr_send, &param_send);

  if((rc=pthread_create(&th_send, &attr_send, sender, NULL)) == 0)
  {
    printf("\n\rSender Thread Created with rc=%d\n\r", rc);
  }
  else
  {
    perror("\n\rFailed to Make Sender Thread\n\r");
    printf("rc=%d\n", rc);
  }

  if((rc=pthread_create(&th_receive, &attr_receive, receiver, NULL)) == 0)
  {
    printf("\n\rReceiver Thread Created with rc=%d\n\r", rc);
  }
  else
  {
    perror("\n\rFailed Making Reciever Thread\n\r");
    printf("rc=%d\n", rc);
  }

  printf("pthread join send\n");
  pthread_join(th_send, NULL);

  printf("pthread join receive\n");
  pthread_join(th_receive, NULL);

}

```

answers/Code_Q2_6/2c/simple.c

```c
#include <pthread.h>
#include <stdio.h>
#include <time.h>
#include <aio.h>
#include <math.h>
#include <unistd.h>

pthread_mutex_t mutex;

typedef struct {
    double latitude;
    double longitude;
    double altitude;
    double roll;
    double pitch;
    double yaw;
    struct timespec sample_time;
} NavigationState;

NavigationState nav_state, nav_state_shouldbe;

pthread_mutex_t state_mutex;

void* update_thread(void * arg){
     for (int i = 0; i < 180; ++i) {
        pthread_mutex_lock(&mutex);
        nav_state.latitude = i;
        nav_state.longitude = 0.5 * i;
        nav_state.altitude = 0.25 * i;
        nav_state.roll = sin(i);
        nav_state.pitch = cos(i * i);
        nav_state.yaw = cos(i);
        clock_gettime(CLOCK_REALTIME, &nav_state.sample_time);
        pthread_mutex_unlock(&mutex);
        printf("updated reading\n");
        printf("Yaw: %f, Roll: %f, Pitch: %f, Latitude %f, Longitude %f, Altitude %f\n",nav_state.yaw, nav_state.roll, nav_state.pitch, nav_state.latitude, nav_state.longitude, nav_state.altitude);
        sleep(1);
    }
    return NULL;
}


void* read_thread(void* arg) {
    for (int i = 0; i < 18; ++i) {
        pthread_mutex_lock(&mutex);
        NavigationState temp_state = nav_state;
        pthread_mutex_unlock(&mutex);
        printf("Reading number %d\n", i);
        printf("Yaw: %f, Roll: %f, Pitch: %f, Latitude %f, Longitude %f, Altitude %f\n",temp_state.yaw, temp_state.roll, temp_state.pitch, temp_state.latitude, temp_state.longitude, temp_state.altitude);
```

```c
        printf("Time : tv_sec: %ld, tv_ns: %ld \n",temp_state.sample_time.tv_sec, temp_state.sample_time.tv_nsec);

        nav_state_shouldbe.latitude = i*10;
        nav_state_shouldbe.longitude = 0.5 * i * 10;
        nav_state_shouldbe.altitude = 0.25 * i * 10;
        nav_state_shouldbe.roll = sin(i*10);
        nav_state_shouldbe.pitch = cos(i * i * 100);
        nav_state_shouldbe.yaw = cos(i*10);

        printf("should be: Yaw: %f, Roll: %f, Pitch: %f, Latitude %f, Longitude %f, Altitude %f \n\n", nav_state_shouldbe.yaw, nav_state_shouldbe.roll, nav_state_shouldbe.pitch, nav_state_shouldbe.latitude, nav_state_shouldbe.longitude, nav_state_shouldbe.altitude);

        sleep(10);
    }
    return NULL;
}

int main(){
    pthread_t t1, t2;
    pthread_mutex_init(&mutex, NULL);
    pthread_create(&t1, NULL, update_thread, NULL);
    pthread_create(&t2, NULL, read_thread, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 0;
}
```

pthread_mutex_timedlock/pthread_mutex_timelock.c

```c
#define _GNU_SOURCE
#include <pthread.h>
#include <stdio.h>
#include <time.h>
#include <aio.h>
#include <math.h>
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>
#include <sched.h>
#include <stddef.h>
#include <sys/sysinfo.h>

pthread_mutex_t mutex;
#define NUMBER_OF_TASKS 2

typedef struct
{
    int threadId;
} ThreadArgs_t;

typedef struct
{
    int period;
    int burst_time;
    int count_for_period;
    struct sched_param priority_param;
    void *(*thread_handle)(void *);
    pthread_t thread;
    ThreadArgs_t thread_args;
    void *return_Value;
    pthread_attr_t attribute;
    int target_cpu;
} RmTask_t;


typedef struct {
    double latitude;
    double longitude;
    double altitude;
    double roll;
    double pitch;
    double yaw;
    struct timespec sample_time;
} NavigationState;

NavigationState nav_state, nav_state_shouldbe;

pthread_mutex_t state_mutex;

void* update_thread(void * arg){
    struct timespec update_interval = {1, 0};
    for (int i=0; i<180; i++) {
```

```c
        pthread_mutex_lock(&state_mutex);

        nav_state.latitude = i;
        nav_state.longitude = 0.5 * i;
        nav_state.altitude = 0.25 * i;
        nav_state.roll = sin(i);
        nav_state.pitch = cos(i * i);
        nav_state.yaw = cos(i);
        clock_gettime(CLOCK_REALTIME, &nav_state.sample_time);



        printf("Updated reading\n");
        printf("Yaw: %f, Roll: %f, Pitch: %f, Latitude %f, Longitude %f, Altitude %f\n",nav_state.yaw, nav_state.roll, nav_state.pitch, nav_state.latitude, nav_state.longitude, nav_state.altitude);

        // Uncomment below line to see reading thread waiting for mutex
        // sleep(10);
        pthread_mutex_unlock(&state_mutex);
        nanosleep(&update_interval, NULL);
    }
    return NULL;
}

void* read_thread(void* arg) {
    struct timespec ts;
    int s;
    struct timespec update_interval = {10, 0};
    for (int i=0; i<18; i++) { // Increased the iteration to match update_thread for continuous checking
        clock_gettime(CLOCK_REALTIME, &ts);
        ts.tv_sec += 10; // Try to acquire the lock within 10 seconds from now

        s = pthread_mutex_timedlock(&state_mutex, &ts);
        if (s == ETIMEDOUT) {
            printf("No new data available at %ld seconds\n", time(NULL));
            // No need to adjust ts because the loop will recalculate it
        } else if (s == 0) {
            // Mutex acquired, read data
            NavigationState temp_state = nav_state;
            pthread_mutex_unlock(&state_mutex);

            printf("Reading data:\n");
            printf("Yaw: %f, Roll: %f, Pitch: %f, Latitude %f, Longitude %f, Altitude %f\n",
                   temp_state.yaw, temp_state.roll, temp_state.pitch,
                   temp_state.latitude, temp_state.longitude, temp_state.altitude);
            printf("Time: tv_sec: %ld, tv_nsec: %ld\n",
                   temp_state.sample_time.tv_sec, temp_state.sample_time.tv_nsec);
        } else {
            // Handle other errors (e.g., EINVAL)
            break;
        }
        sleep(10);
    }
    return NULL;
```

```c
}

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() {

    pthread_t threads[NUMBER_OF_TASKS];
    int coreid = 1;
    cpu_set_t threadcpu;

    CPU_SET(coreid, &threadcpu);

    RmTask_t tasks[NUMBER_OF_TASKS] = {
        {.period = 20,      // ms
         .burst_time = 10, // ms
         .priority_param = {1},
         .thread = threads[0],
         .thread_handle = update_thread,
         .thread_args = {0},
         .return_Value = NULL,
         .attribute = {0, 0},
         .target_cpu = 2},

        {.period = 50,
         .burst_time = 20,
         .priority_param = {2},
         .thread = threads[1],
         .thread_handle = read_thread,
         .thread_args = {0},
         .attribute = {0, 0},
         .target_cpu = 0},

    };


    pthread_attr_t attribute_flags_for_main; // for schedular type, priority
    struct sched_param main_priority_param;
```

```c

    cpu_set_t cpuset;
    int target_cpu = 1; // core we want to run our process on

    printf("This system has %d processors configured and %d processors available.\n", get_nprocs_conf(), get_nprocs());

    printf("Before adjustments to scheduling policy:\n");
    print_scheduler();

    CPU_ZERO(&cpuset); // clear all the cpus in cpuset

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

    main_priority_param.sched_priority = rt_max_prio;
    for (int i = 0; i < NUMBER_OF_TASKS; i++)
    {
        tasks[i].priority_param.sched_priority = rt_max_prio - (2*i*i);

        // initialize attributes
        pthread_attr_init(&tasks[i].attribute);

        pthread_attr_setinheritsched(&tasks[i].attribute, PTHREAD_EXPLICIT_SCHED);
        pthread_attr_setschedpolicy(&tasks[i].attribute, SCHED_FIFO);
        pthread_attr_setschedparam(&tasks[i].attribute, &tasks[i].priority_param);
        pthread_attr_setaffinity_np(&tasks[i].attribute, sizeof(cpu_set_t), &threadcpu);
    }

    pthread_attr_init(&attribute_flags_for_main);

    pthread_attr_setinheritsched(&attribute_flags_for_main, PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setschedpolicy(&attribute_flags_for_main, SCHED_FIFO);
    pthread_attr_setaffinity_np(&attribute_flags_for_main, sizeof(cpu_set_t), &threadcpu);

    // Main thread is already created we have to modify the priority and scheduling scheme
    int status_setting_schedular = sched_setscheduler(getpid(), SCHED_FIFO, &main_priority_param);
    if (status_setting_schedular)
    {
        printf("ERROR; sched_setscheduler rc is %d\n", status_setting_schedular);
        perror(NULL);
        exit(-1);
    }

    printf("After adjustments to scheduling policy:\n");
    print_scheduler();


    for (int i = 0; i < NUMBER_OF_TASKS; i++)
    {
        // Create a thread
        // First paramter is thread which we want to create
        // Second parameter is the flags that we want to give it to
```

```
        // third parameter is the routine we want to give
        // Fourth parameter is the value
        printf("Setting thread %d to core %d\n", i, coreid);



        if (pthread_create(&tasks[i].thread, &tasks[i].attribute, tasks[i].thread_handle, &tasks[i]) != 0)
        {
            perror("Create_Fail");
        }


    }


    for (int i = 0; i < NUMBER_OF_TASKS; i++)
    {
        pthread_join(tasks[i].thread, (void *)&tasks[i].return_Value);
    }

    if (pthread_attr_destroy(&tasks[0].attribute) != 0)
        perror("attr destroy");
    if (pthread_attr_destroy(&tasks[1].attribute) != 0)
        perror("attr destroy");
    return 0;
}
```