RTES Assignment 3

Gerritt Luoma

Exercise 1

> Read Sha, Rajkumar, et al paper, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization".

> A: Summarize 3 main key points the paper makes. Read my summary paper on the topic as well, which might be easier to understand.

The paper "Priority Inheritance Protocols: An Approach to Real-Time Synchronization" by Sha, Rajkumar, et al is explaining the issue in realtime systems known as Priority Inversion and approaches for how to solve the problem. The first key point in the paper is the definition of the Priority Inversion problem. The section explains how the problem arises when a lower priority task is able to revent a higher priority task from running, typically through taking a shared resource and being unable to release the resource due to a higher priority task. If this inversion is unbounded, it can lead to missing deadlines causing catastrophic consequences in critical systems. The second main point of the paper is the introduction of Priority Inheritance Protocols which are solutions to the Priority Inversion problem. The base Priority Inheritance Protocol proposes having lower priority tasks that are blocking a higher priority task inherit the priority of the higher priority task for the duration of the task block so that priority inversion is unable to occur. By doing this, the lower priority task can never be preempted by middle priority tasks causing a potential unbounded inversion. Section 4 of the paper also introduces the concept of the Priority Ceiling Protocol which builds upon the Priority Inheritance Protocol to also prevent potential issues like deadlocks. The third and final key point made in the paper is the introduction of Schedulability Analysis which sets a "set of sufficient conditions under which a set of periodic tasks using the priority ceiling protocol can be scheduled by the rate monotonic algorithm.

> B: Read the historical positions of Linux Torvalds as described by Jonathan Corbet and Ingo Molnar and Thomas Gleixner on this topic as well. 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 know before this).

The Linux communicty resisted addition of priority ceiling/priority inheritance because the Linux Kernel maintainers prioritize simplicity and general purpose performance. Many of the priority inversion solutions added significant performance overhead to the kernal reducing performance. Even Linus was very against priority inheritance going as far as saying "If you really need [priority inheritance], your system is broken anyway.". It wasn't until Ingo introduced his solution for his futex that the first form of realtime priority inheritance was added into the kernel. Within the comments people are mentioning that Ingo was able to do this despite heavy resistance is because he "writes sane patches and doesn't irritate people needlessly" which is key for working with others and finding ways to get what they want.

> C: 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?

Based off of my reading and personal experience, I believe that priority inheritance is the way to go for the majority of scenarios. Based off the simplicity and minimal performance overhead it seems like the obvious choice.

Exercise 2

> Review the terminology guide (glossary in the textbook).

> A: Describe clearly what it means to write "thread safe" functions that are "re-entrant".

Functions that are "re-entrant" so that they are "thread safe" are written in such a way that concurrent calls to the function will not result in the corruption of data. To put it simply, the function must be able to be called, be paused at any point in its execution, a new instance of the function runs, and then it can resume with no changed data or behavior. The most threadsafe/reentrant functions are functions that use exclusively the stack and local variables. If global variables must be used, synchronization primitives must be used to prevent data races or the corruption of data.

> B: There are generally three main ways to do this: 1) pure functions that use only stack and have no global memory, 2) functions which use thread indexed global data, and 3) functions which use shared memory global data, but synchronize access to it using a MUTEX semaphore critical section wrapper. Describe each method and how you would code it and how it would impact real-time threads/tasks. (E.g., if you were to create a simple function to average 2 numbers and return the average, how would you make sure this function is thread safe using each of the three methods).

The easiest and most consistent of the three ways of making a funciton re-entrant is by keeping the function pure and only use the stack. For the case of averaging two numbers, you would take the two numbers as inputs and return the result as such:

```
double pure_solution(double a, double b)
{
    return (a + b) / 2.0;
}
```

This function is pure because it does not reference any global variables and only uses the stack. This function could be called by an infinite number of threads all at the same time and there would be no blocking or performance overhead introduced by interthread synchronization.

The second option is through the use of thread indexed global data. This can be achieved through the use of a global key and the `pthread_getspecific()`/`pthread_setspecific()` functions. When using these functions in tandem with a key, threads are able to store and retrieve data that is reserved for use by the calling thread. This example can be seen below:

```
pthread_key_t key; // Assume key has been initialized in the main or
calling thread
```

```c
void* average_thread_func(void* arg)
{
    double* data = malloc(sizeof(double));
    *data = 0;
    pthread_setspecific(key, data); // Set the data in the key store

    thread_arg_t* thread_args = (thread_arg_t*)arg; // Assume thread_arg_t
is defined
    *data = (thread_args->a + thread_args->b) / 2.0;

    // ...

    double* retrieve_data = pthread_getspecific(key); // Grab the stored
pointer later in the thread

    return NULL;
}
```

This method seems to require a lot of setup and will require extra memory allocation since each thread can allocate its own data to be stored.

The final method is through the use of synchronization primitives like a mutex which allow exclusive access to a shared resource. If a mutex has already been locked, then any other threads that want access to the shared resource will block on the acquisition of the mutex. This method can be seen below:

```c
double global_data = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void* average_thread_func(void* arg)
{
    // Assume that thread_arg_t is defined
    thread_arg_t* thread_args = (thread_arg_t*)args;
    double avg = (thread_args->a + thread_args->b) / 2.0;

    // Can block if another thread has already locked the mutex
    pthread_mutex_lock(&lock);
    global_data = avg;
    pthread_mutex_unlock(&lock);

    return NULL;
}
```

> C: Now, using a MUTEX, provide an example using RT-Linux Pthreads that does a threadsafe 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 position and 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 usin gmath library function generators (e.g., use simple periodic functions for Roll, Pitch, Yaws in(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 add 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.

I have added my program fulfilling the requirements of this problem along with its output to the directory `problem-2/` of this repository. I used Sam Siewert's provided `seqgen3.c` program as the base which can be found in the RTES-ECEE-5623 repository. I have modified the program to only have the two required services, the writer and the reader, which are running at 1Hz and 0.1Hz respectively being released by a timer sequencer operating at 2Hz. Instrucitons to run the program can be found below:

```
// First terminal in the problem-2 directory - builds and runs the program
$ make clean
$ make
$ sudo ./seqgen3

// Second terminal in any directory - outputs the syslogs of the program
$ journalctl -f
```

I am using practically randomized data in the writer thread, Service_1, to "simulate" the Attitude and SCLK of a satellite. If any satellite has the true ATD of this satellite, you will want to say goodbye to it because it is spinning like a top and dropping fast. The logic of the writer thread running at 1Hz is found below:

```
void *Service_1(void *threadp)
{
    struct timespec current_time_val;
    double current_realtime;
    unsigned long long S1Cnt=0;
    threadParams_t *threadParams = (threadParams_t *)threadp;

    // Start up processing and resource initialization
    clock_gettime(MY_CLOCK_TYPE, &current_time_val);
current_realtime=realtime(&current_time_val);
    syslog(LOG_CRIT, "S1 thread @ sec=%6.9lf\n", current_realtime-
start_realtime);
    printf("S1 thread @ sec=%6.9lf\n", current_realtime-start_realtime);

    while(!abortS1) // check for synchronous abort request
    {
        // wait for service request from the sequencer, a signal handler
or ISR in kernel
        sem_wait(&semS1);

        S1Cnt++;

        nav_state_t new_state;
```

```
        new_state.lat = 50.0 + (1.0 * S1Cnt);
        new_state.lon = -40.0 - (2.0 * S1Cnt);
        new_state.alt = 150.0 - (0.05 * S1Cnt);

        new_state.roll = sin((double)S1Cnt);
        new_state.pitch = cos((double)S1Cnt * (double)S1Cnt);
        new_state.yaw = cos((double)S1Cnt);

        clock_gettime(MY_CLOCK_TYPE, &current_time_val);
current_realtime=realtime(&current_time_val);
        new_state.sample_time = current_realtime;

        pthread_mutex_lock(&shared_state_mutex);
        shared_state = new_state;
        // Shouldn't do additional proc like logging in mutex lock but
keep locked to print current vals
        syslog(LOG_CRIT,
               "S1 1 Hz on core %d for release  %3llu: lat=%6.5lf,
lon=%6.5lf, alt=%6.5lf, roll=%6.5lf, pitch=%6.5lf, yaw=%6.5lf @
sec=%6.9lf\n",
               sched_getcpu(),
               S1Cnt,
               shared_state.lat,
               shared_state.lon,
               shared_state.alt,
               shared_state.roll,
               shared_state.pitch,
               shared_state.yaw,
               shared_state.sample_time
        );
        pthread_mutex_unlock(&shared_state_mutex);

    }

    // Resource shutdown here
    //
    pthread_exit((void *)0);
}
```

The second thread, Service_2, runs at 0.1Hz simply reading the shared state and printing it to the syslogs. Its code is found below:

```
void *Service_2(void *threadp)
{
    struct timespec current_time_val;
    double current_realtime;
    unsigned long long S2Cnt=0;
    threadParams_t *threadParams = (threadParams_t *)threadp;

    clock_gettime(MY_CLOCK_TYPE, &current_time_val);
current_realtime=realtime(&current_time_val);
```

```
    syslog(LOG_CRIT, "S2 thread @ sec=%6.9lf\n", current_realtime-
start_realtime);
    printf("S2 thread @ sec=%6.9lf\n", current_realtime-start_realtime);

    while(!abortS2)
    {
        sem_wait(&semS2);
        S2Cnt++;

        pthread_mutex_lock(&shared_state_mutex);
        // Shouldn't do additional proc like logging in mutex lock but
keep locked to print current vals
        syslog(LOG_CRIT,
               "S2 0.1 Hz on core %d for release %2llu: lat=%6.5lf,
lon=%6.5lf, alt=%6.5lf, roll=%6.5lf, pitch=%6.5lf, yaw=%6.5lf @
sec=%6.9lf\n",
               sched_getcpu(),
               S2Cnt,
               shared_state.lat,
               shared_state.lon,
               shared_state.alt,
               shared_state.roll,
               shared_state.pitch,
               shared_state.yaw,
               shared_state.sample_time
        );
        pthread_mutex_unlock(&shared_state_mutex);
    }

    pthread_exit((void *)0);
}
```

As mentioned before, the full output of the program can be found in the `problem-2` directory but the main task of this problem was to ensure the mutex synchronization was preventing any data corruption for all 18 times both of the threads access the shared state. After analyzing the logs, I can confirm that all 18 occurrences result in no data corruption. The writer thread, having the higher priority, first updates the state and then the reader thread outputs it. The output of all 18 of these occurrences can be found below:

```
Jun 28 09:24:20 arthur seqgen3[31547]: S1 1 Hz on core 2 for release   10:
lat=60.00000, lon=-60.00000, alt=149.50000, roll=-0.54402, pitch=0.86232,
yaw=-0.83907 @ sec=1728732.540957039
Jun 28 09:24:20 arthur seqgen3[31547]: S2 0.1 Hz on core 2 for release  1:
lat=60.00000, lon=-60.00000, alt=149.50000, roll=-0.54402, pitch=0.86232,
yaw=-0.83907 @ sec=1728732.540957039

Jun 28 09:24:30 arthur seqgen3[31547]: S1 1 Hz on core 2 for release   20:
lat=70.00000, lon=-80.00000, alt=149.00000, roll=0.91295, pitch=-0.52530,
yaw=0.40808 @ sec=1728742.541062533
Jun 28 09:24:30 arthur seqgen3[31547]: S2 0.1 Hz on core 2 for release  2:
lat=70.00000, lon=-80.00000, alt=149.00000, roll=0.91295, pitch=-0.52530,
yaw=0.40808 @ sec=1728742.541062533
```

```
Jun 28 09:24:40 arthur seqgen3[31547]: S1 1 Hz on core 2 for release   30:
lat=80.00000, lon=-100.00000, alt=148.50000, roll=-0.98803, pitch=0.06625,
yaw=0.15425 @ sec=1728752.541168657
Jun 28 09:24:40 arthur seqgen3[31547]: S2 0.1 Hz on core 2 for release  3:
lat=80.00000, lon=-100.00000, alt=148.50000, roll=-0.98803, pitch=0.06625,
yaw=0.15425 @ sec=1728752.541168657

Jun 28 09:24:50 arthur seqgen3[31547]: S1 1 Hz on core 2 for release   40:
lat=90.00000, lon=-120.00000, alt=148.00000, roll=0.74511, pitch=-0.59836,
yaw=-0.66694 @ sec=1728762.541265651
Jun 28 09:24:50 arthur seqgen3[31547]: S2 0.1 Hz on core 2 for release  4:
lat=90.00000, lon=-120.00000, alt=148.00000, roll=0.74511, pitch=-0.59836,
yaw=-0.66694 @ sec=1728762.541265651

Jun 28 09:25:00 arthur seqgen3[31547]: S1 1 Hz on core 2 for release   50:
lat=100.00000, lon=-140.00000, alt=147.50000, roll=-0.26237,
pitch=0.75983, yaw=0.96497 @ sec=1728772.541368367
Jun 28 09:25:00 arthur seqgen3[31547]: S2 0.1 Hz on core 2 for release  5:
lat=100.00000, lon=-140.00000, alt=147.50000, roll=-0.26237,
pitch=0.75983, yaw=0.96497 @ sec=1728772.541368367

Jun 28 09:25:10 arthur seqgen3[31547]: S1 1 Hz on core 2 for release   60:
lat=110.00000, lon=-160.00000, alt=147.00000, roll=-0.30481,
pitch=0.96505, yaw=-0.95241 @ sec=1728782.541465065
Jun 28 09:25:10 arthur seqgen3[31547]: S2 0.1 Hz on core 2 for release  6:
lat=110.00000, lon=-160.00000, alt=147.00000, roll=-0.30481,
pitch=0.96505, yaw=-0.95241 @ sec=1728782.541465065

Jun 28 09:25:20 arthur seqgen3[31547]: S1 1 Hz on core 2 for release   70:
lat=120.00000, lon=-180.00000, alt=146.50000, roll=0.77389, pitch=0.63365,
yaw=0.63332 @ sec=1728792.541573429
Jun 28 09:25:20 arthur seqgen3[31547]: S2 0.1 Hz on core 2 for release  7:
lat=120.00000, lon=-180.00000, alt=146.50000, roll=0.77389, pitch=0.63365,
yaw=0.63332 @ sec=1728792.541573429

Jun 28 09:25:30 arthur seqgen3[31547]: S1 1 Hz on core 2 for release   80:
lat=130.00000, lon=-200.00000, alt=146.00000, roll=-0.99389,
pitch=-0.83878, yaw=-0.11039 @ sec=1728802.541682201
Jun 28 09:25:30 arthur seqgen3[31547]: S2 0.1 Hz on core 2 for release  8:
lat=130.00000, lon=-200.00000, alt=146.00000, roll=-0.99389,
pitch=-0.83878, yaw=-0.11039 @ sec=1728802.541682201

Jun 28 09:25:40 arthur seqgen3[31547]: S1 1 Hz on core 2 for release   90:
lat=140.00000, lon=-220.00000, alt=145.50000, roll=0.89400, pitch=0.56188,
yaw=-0.44807 @ sec=1728812.541776806
Jun 28 09:25:40 arthur seqgen3[31547]: S2 0.1 Hz on core 2 for release  9:
lat=140.00000, lon=-220.00000, alt=145.50000, roll=0.89400, pitch=0.56188,
yaw=-0.44807 @ sec=1728812.541776806

Jun 28 09:25:50 arthur seqgen3[31547]: S1 1 Hz on core 2 for release  100:
lat=150.00000, lon=-240.00000, alt=145.00000, roll=-0.50637,
pitch=-0.95216, yaw=0.86232 @ sec=1728822.541879652
Jun 28 09:25:50 arthur seqgen3[31547]: S2 0.1 Hz on core 2 for release 10:
```

```
lat=150.00000, lon=-240.00000, alt=145.00000, roll=-0.50637,
pitch=-0.95216, yaw=0.86232 @ sec=1728822.541879652

Jun 28 09:26:00 arthur seqgen3[31547]: S1 1 Hz on core 2 for release  110:
lat=160.00000, lon=-260.00000, alt=144.50000, roll=-0.04424,
pitch=0.15526, yaw=-0.99902 @ sec=1728832.541985998
Jun 28 09:26:00 arthur seqgen3[31547]: S2 0.1 Hz on core 2 for release 11:
lat=160.00000, lon=-260.00000, alt=144.50000, roll=-0.04424,
pitch=0.15526, yaw=-0.99902 @ sec=1728832.541985998

Jun 28 09:26:10 arthur seqgen3[31547]: S1 1 Hz on core 2 for release  120:
lat=170.00000, lon=-280.00000, alt=144.00000, roll=0.58061, pitch=0.48824,
yaw=0.81418 @ sec=1728842.542088511
Jun 28 09:26:10 arthur seqgen3[31547]: S2 0.1 Hz on core 2 for release 12:
lat=170.00000, lon=-280.00000, alt=144.00000, roll=0.58061, pitch=0.48824,
yaw=0.81418 @ sec=1728842.542088511

Jun 28 09:26:20 arthur seqgen3[31547]: S1 1 Hz on core 2 for release  130:
lat=180.00000, lon=-300.00000, alt=143.50000, roll=-0.93011,
pitch=-0.19640, yaw=-0.36729 @ sec=1728852.542189560
Jun 28 09:26:20 arthur seqgen3[31547]: S2 0.1 Hz on core 2 for release 13:
lat=180.00000, lon=-300.00000, alt=143.50000, roll=-0.93011,
pitch=-0.19640, yaw=-0.36729 @ sec=1728852.542189560

Jun 28 09:26:30 arthur seqgen3[31547]: S1 1 Hz on core 2 for release  140:
lat=190.00000, lon=-320.00000, alt=143.00000, roll=0.98024,
pitch=-0.92239, yaw=-0.19781 @ sec=1728862.542293554
Jun 28 09:26:30 arthur seqgen3[31547]: S2 0.1 Hz on core 2 for release 14:
lat=190.00000, lon=-320.00000, alt=143.00000, roll=0.98024,
pitch=-0.92239, yaw=-0.19781 @ sec=1728862.542293554

Jun 28 09:26:40 arthur seqgen3[31547]: S1 1 Hz on core 2 for release  150:
lat=200.00000, lon=-340.00000, alt=142.50000, roll=-0.71488,
pitch=0.99625, yaw=0.69925 @ sec=1728872.542394289
Jun 28 09:26:40 arthur seqgen3[31547]: S2 0.1 Hz on core 2 for release 15:
lat=200.00000, lon=-340.00000, alt=142.50000, roll=-0.71488,
pitch=0.99625, yaw=0.69925 @ sec=1728872.542394289

Jun 28 09:26:50 arthur seqgen3[31547]: S1 1 Hz on core 2 for release  160:
lat=210.00000, lon=-360.00000, alt=142.00000, roll=0.21943,
pitch=-0.66855, yaw=-0.97563 @ sec=1728882.542497764
Jun 28 09:26:50 arthur seqgen3[31547]: S2 0.1 Hz on core 2 for release 16:
lat=210.00000, lon=-360.00000, alt=142.00000, roll=0.21943,
pitch=-0.66855, yaw=-0.97563 @ sec=1728882.542497764

Jun 28 09:27:00 arthur seqgen3[31547]: S1 1 Hz on core 2 for release  170:
lat=220.00000, lon=-380.00000, alt=141.50000, roll=0.34665,
pitch=-0.88272, yaw=0.93799 @ sec=1728892.542616666
Jun 28 09:27:00 arthur seqgen3[31547]: S2 0.1 Hz on core 2 for release 17:
lat=220.00000, lon=-380.00000, alt=141.50000, roll=0.34665,
pitch=-0.88272, yaw=0.93799 @ sec=1728892.542616666

Jun 28 09:27:10 arthur seqgen3[31547]: S1 1 Hz on core 2 for release  180:
lat=230.00000, lon=-400.00000, alt=141.00000, roll=-0.80115,
```

```
pitch=-0.72830, yaw=-0.59846 @ sec=1728902.542711160
Jun 28 09:27:10 arthur seqgen3[31547]: S2 0.1 Hz on core 2 for release 18:
lat=230.00000, lon=-400.00000, alt=141.00000, roll=-0.80115,
pitch=-0.72830, yaw=-0.59846 @ sec=1728902.542711160
```

Exercise 3

> Download example-sync-updated-2/ and review, build and run it.

`deadlock.c` is a program that will intentionally create a deadlock. It does so by creating two threads, thread 1 and thread 2, and two mutexes, A and B. Thread 1 starts and grabs Mutex A before sleeping for 1 second. At the same time, thread 2 grabs mutex B and sleeps for 1 second. After their sleeps each thread attempts to grab the mutex held by the other thread before releasing the mutex of their own. This causes a deadlock scenario where the program is unable to progress. The output of `deadlock.c` is seen below:

```
$ sudo ./deadlock
Will set up unsafe deadlock scenario
Creating thread 0
Thread 1 spawned
Creating thread 1
THREAD 1 grabbing resources
Thread 2 spawned
rsrcACnt=1, rsrcBCnt=0
will try to join CS threads unless they deadlock
THREAD 2 grabbing resources
THREAD 1 got A, trying for B
THREAD 2 got B, trying for A
// Never proceeds past here
```

`deadlock_timeout.c` is a program that performs a very similar operation to `deadlock.c` in that it intentionally creates a deadlock scenario. The difference, however, is that in `deadlock_timeout.c` the second mutext is acquired using `pthread_mutex_timedlock` with a timeout of 2 seconds. When the deadlock occurs, one of the threads eventually errors out on the mutext acquisition and gives up its own resource. This allows the other thread to acquire its second resource declaring success. This can be seen below:

```
$ sudo ./deadlock_timeout
Will set up unsafe deadlock scenario
Creating thread 0
Creating thread 1
Thread 1 started
THREAD 1 grabbing resource A @ 1751143126 sec and 238666354 nsec
Thread 1 GOT A
rsrcACnt=1, rsrcBCnt=0
will sleep
will try to join both CS threads unless they deadlock
Thread 2 started
THREAD 2 grabbing resource B @ 1751143126 sec and 238892666 nsec
```

```
Thread 2 GOT B
rsrcACnt=1, rsrcBCnt=1
will sleep
THREAD 1 got A, trying for B @ 1751143127 sec and 238830363 nsec
THREAD 2 got B, trying for A @ 1751143127 sec and 239147675 nsec
Thread 2 TIMEOUT ERROR
Thread 1 GOT B @ 1751143129 sec and 240233700 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
```

`pthread3.c` is a program that intentionally creates an unbounded priority inversion situation. It does so by creating 3 threads, one high priority, one low priority, and one medium priority. The low priority thread is started first which acquires a shared memory mutex before starting a long running computation. The high rate thread is then started which preempts the low priority task until it attempts to acquire the mutex which it cannot do since the low priority task already has it locked. A medium priority task is then spawned which once again preempts the low priority task. This medium priority task also performs a long running computation which prevents the low priority task from releasing the shared resource. Once it completes the low priority task can complete unlocking the mutex which allows the high priority task to finish which then allows the program as a whole to complete. This can be seen below:

```
$ sudo ./pthread3
Fibonacci Cycle Burner test ...
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946
17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309
3524578 5702887 9227465 14930352 24157817 39088169 63245986 102334155
165580141 267914296 433494437 701408733 1134903170 1836311903 2971215073
512559680

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946
17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309
3524578 5702887 9227465 14930352 24157817 39088169 63245986 102334155
165580141 267914296 433494437 701408733 1134903170 1836311903 2971215073
512559680
done
interference time = 10 secs
unsafe mutex will be created
Pthread Policy is SCHED_OTHER
Setting thread 0 to core 0

Launching thread 0
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM

Creating RT thread 0
Start services thread spawned
will join service threads
```

```
Creating Low Prio RT or BE thread 3
Low prio 3 thread SPAWNED at 0.000331 sec
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................
CS-L REQUEST

CS-L ENTRY 1
CS-L1 CS-L2 CScnt=1

Creating RT thread 1, CScnt=1

CS-H REQUEST
High prio 1 thread SPAWNED at 0.199399 sec

Creating RT thread 2
M1 M2 M3 M4 M5 M6 M7 M8 M9 M10
**** MID PRIO 2 on core 0 INTERFERE NO SEM COMPLETED at 0.539686 sec
Middle prio 2 thread SPAWNED at 0.540104 sec
CS-L3 CS-L4 CS-L5 CS-L6 CS-L7 CS-L8 CS-L9 CS-L10
CS-L LEAVING

CS-H ENTRY 2
CS-H1 CS-H2 CS-H3 CS-H4 CS-H5 CS-H6 CS-H7 CS-H8 CS-H9 CS-H10
CS-H LEAVING
```

```
CS-H EXIT

**** HIGH PRIO 1 on core 0 CRIT SECTION WORK COMPLETED at 1.124067 sec

CS-L EXIT

**** LOW PRIO 3 on core 0 CRIT SECTION WORK COMPLETED at 1.124238 sec
HIGH PRIO joined
MID PRIO joined
LOW PRIO joined
START SERVICE joined
All threads done
```

I am unable to run `pthread3amp.c` because it causes my SSH window to crash. The SA has also confirmed this issue with the code.

`pthread3ok.c` provides an example fix to the unbounded priority inversion found in `pthread3.c`. It does so by not using a mutex to synchromize the work between the thread. This is not a practical fix because it can lead to data races and undefined behavior. The output can be seen below:

```
$ sudo ./pthread3ok
Fibonacci Cycle Burner test ...
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946
17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309
3524578 5702887 9227465 14930352 24157817 39088169 63245986 102334155
165580141 267914296 433494437 701408733 1134903170 1836311903 2971215073
512559680

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946
17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309
3524578 5702887 9227465 14930352 24157817 39088169 63245986 102334155
165580141 267914296 433494437 701408733 1134903170 1836311903 2971215073
512559680
done
interference time = 10 secs
unsafe mutex will be created
Pthread Policy is SCHED_OTHER
Setting thread 0 to core 0

Launching thread 0
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM

Creating RT thread 0
Start services thread spawned
will join service threads

Creating BE thread 3
Low prio 3 thread SPAWNED at 0.000421 sec
.....................................................................
.....................................................................
```

```
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................
CS-L REQUEST

CS-L ENTRY 1
CS-L1 CS-L2
Creating RT thread 1, CScnt=1

CS-H REQUEST

CS-H ENTRY 2
CS-H1 CS-H2 CS-H3 CS-H4 CS-H5 CS-H6 CS-H7 CS-H8 CS-H9 CS-H10
CS-H LEAVING

CS-H EXIT

**** HIGH PRIO 1 on core 0 UNPROTECTED CRIT SECTION WORK COMPLETED at
0.536935 sec
High prio 1 thread SPAWNED at 0.537398 sec

Creating RT thread 2
M1 M2 M3 M4 M5 M6 M7 M8 M9 M10
**** MID PRIO 2 on core 0 INTERFERE NO SEM COMPLETED at 0.877677 sec
Middle prio 2 thread SPAWNED at 0.877747 sec
HIGH PRIO joined
MID PRIO joined
CS-L3 CS-L4 CS-L5 CS-L6 CS-L7 CS-L8 CS-L9 CS-L10
CS-L LEAVING
```

```
CS-L EXIT

**** LOW PRIO 3 on core 0 UNPROTECTED CRIT SECTION WORK COMPLETED at
1.136382 sec
LOW PRIO joined
START SERVICE joined
All threads done
```

> A: Describe both the issues of deadlock and unbounded priority inversion and the root cause for both in the example code.

`deadlock.c` demonstrates a deadlock scenario where two threads try to take shared reasources in a way that the threads can never progress. It does so by creating two threads, thread 1 and thread 2, and two mutexes, A and B. Thread 1 starts and grabs Mutex A before sleeping for 1 second. At the same time, thread 2 grabs mutex B and sleeps for 1 second. After their sleeps each thread attempts to grab the mutex held by the other thread before releasing the mutex of their own. This causes a deadlock scenario where the program is unable to progress.

`pthread3.c` demonstrates an unbounded priority inversion situation. It does so by creating 3 threads, one high priority, one low priority, and one medium priority. The low priority thread is started first which acquires a shared memory mutex before starting a long running computation. The high rate thread is then started which preempts the low priority task until it attempts to acquire the mutex which it cannot do since the low priority task already has it locked. A medium priority task is then spawned which once again preempts the low priority task. This medium priority task also performs a long running computation which prevents the low priority task from releasing the shared resource. Once it completes the low priority task can complete unlocking the mutex which allows the high priority task to finish which then allows the program as a whole to complete.

> B: 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?

I have added a new C source file at `problem-3/example-sync-updated-2/deadlock-backoff.c` which uses a random backoff technique to address any possible form of deadlock. On each thread it will acquire its original mutex, for thread 1 it is A, and then it will attempt to acquire its second mutex using `pthread_mutex_trylock()`. If it fails, it will log an error, release its original mutex, and sleep for a random amount of time between 1 and 10ms. This code example consistently sees thread 1 backing off allowing thread 2 to finish before it loops back and finishes up itself. Its output is seen below here:

```
$ sudo ./deadlock-backoff
Will set up unsafe deadlock scenario
Creating thread 0
Thread 1 spawned
Creating thread 1
THREAD 1 grabbing resources
Thread 2 spawned
rsrcACnt=1, rsrcBCnt=0
will try to join CS threads unless they deadlock
THREAD 2 grabbing resources
```

```
THREAD 1 got A, trying for B
THREAD 1 failed to get B, backing off
THREAD 2 got B, trying for A
THREAD 2 got A and B
THREAD 2 done
THREAD 1 grabbing resources
THREAD 1 got A, trying for B
THREAD 1 got A and B
THREAD 1 done
Thread 1: 8ac5f180 done
Thread 2: 8a44f180 done
All done
```

Yes, Linux does have a fix to the priority inheritance problem. You can change attributes in the mutex to use priority inheritance to unblock the higher priority task and allow it to run. I have changed the initialization of the shared state mutex to:

```
pthread_mutexattr_t attr;
pthread_mutexattr_init(&attr);
pthread_mutexattr_setprotocol(&attr, PTHREAD_PRIO_INHERIT); // Explicitly
states that threads blocking on the mutex must use priority inheritance
pthread_mutex_init(&sharedMemSem, &attr);
```

When using this change, the output of the program has changed to show that the high and low tasks are finishing before the middle priority task is even starting. This shows that the priority inheritance is working to prevent an unbounded priority inversion. The output is seen below:

```
$ sudo ./pthread3 10
Fibonacci Cycle Burner test ...
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946
17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309
3524578 5702887 9227465 14930352 24157817 39088169 63245986 102334155
165580141 267914296 433494437 701408733 1134903170 1836311903 2971215073
512559680

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946
17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309
3524578 5702887 9227465 14930352 24157817 39088169 63245986 102334155
165580141 267914296 433494437 701408733 1134903170 1836311903 2971215073
512559680
done
interference time = 10 secs
unsafe mutex will be created
Pthread Policy is SCHED_OTHER
Setting thread 0 to core 0

Launching thread 0
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM
```

```
Creating RT thread 0
Start services thread spawned
will join service threads

Creating Low Prio RT or BE thread 3
Low prio 3 thread SPAWNED at 0.000142 sec
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
```

```
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................
CS-L REQUEST

CS-L ENTRY 1
CS-L1 CS-L2 CScnt=1

Creating RT thread 1, CScnt=1

CS-H REQUEST
CS-L3 CS-L4 CS-L5 CS-L6 CS-L7 CS-L8 CS-L9 CS-L10
CS-L LEAVING

CS-H ENTRY 2
CS-H1 CS-H2 CS-H3 CS-H4 CS-H5 CS-H6 CS-H7 CS-H8 CS-H9 CS-H10
CS-H LEAVING

CS-H EXIT

**** HIGH PRIO 1 on core 0 CRIT SECTION WORK COMPLETED at 0.780355 sec

CS-L EXIT

**** LOW PRIO 3 on core 0 CRIT SECTION WORK COMPLETED at 0.780802 sec
High prio 1 thread SPAWNED at 0.780844 sec

Creating RT thread 2
M1 M2 M3 M4 M5 M6 M7 M8 M9 M10
**** MID PRIO 2 on core 0 INTERFERE NO SEM COMPLETED at 1.120031 sec
Middle prio 2 thread SPAWNED at 1.120168 sec
HIGH PRIO joined
MID PRIO joined
LOW PRIO joined
START SERVICE joined
All threads done
```

> C: 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?

I do not believe the `PREEMPT_RT` patch would be helpful in this scenario as this program is running entirely in userspace and the `PREEMPT_RT` patch just allows anything to be preempted anywhere else in the kernel/user space. This priority inversion problem seen in the `pthread3.c` program is running entirely in user space and isn't being preempted by the kernel to cause the problem. The priority inheritance on the mutex seems to do the trick.

> D: 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?

For Hard Real Time (HRT) services that, when in a fault, could cause loss of life, I believe a true RTOS could still be the correct decision. This is because they have been used in the field for a long time and are more proven while the `PREEMPT_RT` patch is still relatively new in the main linux kernel branch. For Soft Real Time (SRT) services, I believe Linux is a fine solution since they don't have hard requirements and the Linux Kernel seems to be able to easily support soft real time. Given a few more years proving the functionality of the `PREEMPT_RT` patch, I believe Linux could start being considered for hard real time services.

Exercise 4

> Review POSIX-Examples-Updated and POSIX_MQ_LOOP and build the code related to POSIX message queues and run them to learn about basic use.

POSIX-Examples-Updated `clock_nanosleep_test.c` takes in the input of the number of iterations and the number of milliseconds per iteration. It then loops through the number of specified iterations calling `nanosleep()` for the specified sleep duration and outputs the amount of time that the thread was asleep. It will also output the amount of error between the requested sleep time and the actual time with the error in microseconds. Below is the output of the program:

```
$ sudo ./clock_nanosleep_test 10 1000
Requested 10 iterations for 1000 msecs
Before adjustments to scheduling policy:
Pthread Policy is SCHED_OTHER
After adjustments to scheduling policy:
Pthread Policy is SCHED_FIFO


Thread sleep based delay test for 1 secs, 0 nsecs


Clock resolution:
    0 secs, 0 usecs, 1 nsecs OR 0.000000 secs


System ticks_per_sec=100

RT clock dt = 1.000056 secs for sleep of 1.000000 secs, err = 56.414000
usec, sleepcnt=1, ticks=100
RT clock dt = 1.000055 secs for sleep of 1.000000 secs, err = 55.228000
```

```
usec, sleepcnt=1, ticks=100
RT clock dt = 1.000042 secs for sleep of 1.000000 secs, err = 41.841000
usec, sleepcnt=1, ticks=100
RT clock dt = 1.000057 secs for sleep of 1.000000 secs, err = 56.766000
usec, sleepcnt=1, ticks=100
RT clock dt = 1.000050 secs for sleep of 1.000000 secs, err = 49.656000
usec, sleepcnt=1, ticks=100
RT clock dt = 1.000050 secs for sleep of 1.000000 secs, err = 50.360000
usec, sleepcnt=1, ticks=100
RT clock dt = 1.000039 secs for sleep of 1.000000 secs, err = 39.324000
usec, sleepcnt=1, ticks=100
RT clock dt = 1.000039 secs for sleep of 1.000000 secs, err = 39.046000
usec, sleepcnt=1, ticks=100
RT clock dt = 1.000042 secs for sleep of 1.000000 secs, err = 42.380000
usec, sleepcnt=1, ticks=100
RT clock dt = 1.000041 secs for sleep of 1.000000 secs, err = 41.270000
usec, sleepcnt=1, ticks=100
TEST COMPLETE
```

`clock_pitsig_test.c` is similar to `clock_nanosleep_test.c` in that it takes in arguments of the number of iterations and the number of milliseconds each iteration should take. The difference, however, is that instead of using `nanosleep()` to perform the periodic timing, the program uses a timer and a signal to call the periodic process. The process outputs a timestamp, the number of ticks it was asleep, and the delta from the expected amount of time in microseconds. This output can be seen below:

```
$ sudo ./clock_pitsig_test 10 1000
Requested 10 iterations for 1000 msecs (1 sec, 0 nsec) with
target=1.000000


PIT based delay test for 1 secs, 0 nsecs


Clock resolution:
    0 secs, 0 usecs, 1 nsecs OR 0.000000 secs


System ticks_per_sec=100

Before adjustments to scheduling policy:
Pthread Policy is SCHED_OTHER
After adjustments to scheduling policy:
Pthread Policy is SCHED_FIFO
PIT RT clock = 1751212519.265828 secs, ticks = 1899654233, dt = 1.000076,
err = -76.346000 usec
PIT RT clock = 1751212520.265821 secs, ticks = 100, dt = 0.999993, err =
7.413000 usec
PIT RT clock = 1751212521.265803 secs, ticks = 100, dt = 0.999983, err =
17.357000 usec
PIT RT clock = 1751212522.265810 secs, ticks = 100, dt = 1.000007, err =
-6.736000 usec
```

```
PIT RT clock = 1751212523.265805 secs, ticks = 100, dt = 0.999995, err =
4.985000 usec
PIT RT clock = 1751212524.265812 secs, ticks = 100, dt = 1.000007, err =
-6.754000 usec
PIT RT clock = 1751212525.265817 secs, ticks = 100, dt = 1.000006, err =
-5.848000 usec
PIT RT clock = 1751212526.265798 secs, ticks = 100, dt = 0.999981, err =
19.485000 usec
PIT RT clock = 1751212527.265818 secs, ticks = 100, dt = 1.000020, err =
-19.754000 usec
PIT RT clock = 1751212528.265810 secs, ticks = 100, dt = 0.999993, err =
7.168000 usec
Disabling interval timer with abort=1
Test aborted
TEST COMPLETE
```

`posix_clock.c` performs a single delay test of 3 seconds not taking in any user input. Based off a defined value the test will either be ran in a realtime context using `SCHED_FIFO` or will be performed within a non-realtime context with `SCHED_OTHER`. The defined value is commented out in the source code so it is not being ran in a realtime context. The test does one `nanosleep()` and compares the actual sleep time to the expected sleep time of 3 seconds. The output is seen below showing that the test had a considerable amount of error:

```
$ sudo ./posix_clock
Before adjustments to scheduling policy:
Pthread Policy is SCHED_OTHER


POSIX Clock demo using system RT clock with resolution:
    0 secs, 0 microsecs, 1 nanosecs

RT clock start seconds = 1751212711, nanoseconds = 141664812
RT clock stop seconds = 1751212714, nanoseconds = 141762205
RT clock DT seconds = 3, nanoseconds = 97393
Requested sleep seconds = 3, nanoseconds = 0

Sleep loop count = 1
RT clock delay error = 0, nanoseconds = 97393
TEST COMPLETE
```

`posix_linux_demo.c` creates one thread considered the idle thread that loops for the duration of the test sleeping 1 second at a time. The program also attaches a signal handler to output any received user signals while the main thread is emitting signals so that they will be received by the signal handler. The idle thread and the signal handler share a mutex for a shared output buffer which allows each of them to add to the output of the program. This can be seen below:

```
$ sudo ./posix_linux_demo
Signal generator process 33694: Idle process to signal 33695
```

```
THROW SIGNAL #### Signal 35 thrown with val=0
***** signal CS Shared buffer =
CATCH #### Caught user signal 35 1 times with val=0,
sival_ptr=0x5500000000
This is the idle process 33695
Signal generator exit_thread=0 (differnt copy than idle)
THROW SIGNAL #### Signal 35 thrown with val=1
***** signal CS Shared buffer = signal task was here!
CATCH #### Caught user signal 35 2 times with val=1,
sival_ptr=0x5500000001
*****idle thread in CS Shared buffer = signal task was here!
Signal generator exit_thread=0 (differnt copy than idle)
THROW SIGNAL #### Signal 35 thrown with val=2
***** signal CS Shared buffer = idle task was here!
CATCH #### Caught user signal 35 3 times with val=2,
sival_ptr=0x5500000002
*****idle thread in CS Shared buffer = signal task was here!
Signal generator exit_thread=0 (differnt copy than idle)
THROW SIGNAL #### Signal 35 thrown with val=3
CATCH #### Caught user signal 35 4 times with val=3,
sival_ptr=0x5500000003
***** signal CS Shared buffer = idle task was here!
*****idle thread in CS Shared buffer = signal task was here!
Signal generator exit_thread=0 (differnt copy than idle)
THROW SIGNAL #### Signal 35 thrown with val=4
***** signal CS Shared buffer = idle task was here!
*****idle thread in CS Shared buffer = idle task was here!
CATCH #### Caught user signal 35 5 times with val=4,
sival_ptr=0x5500000004
Signal generator exit_thread=0 (differnt copy than idle)
*****idle thread in CS Shared buffer = idle task was here!
CATCH #### Caught user signal 35 6 times with val=5,
sival_ptr=0x5500000005
THROW SIGNAL #### Signal 35 thrown with val=5
***** signal CS Shared buffer = idle task was here!
*****idle thread in CS Shared buffer = signal task was here!
Signal generator exit_thread=0 (differnt copy than idle)
THROW SIGNAL #### Signal 35 thrown with val=6
***** signal CS Shared buffer = idle task was here!
CATCH #### Caught user signal 35 7 times with val=6,
sival_ptr=0x5500000006
*****idle thread in CS Shared buffer = signal task was here!
Signal generator exit_thread=0 (differnt copy than idle)
THROW SIGNAL #### Signal 35 thrown with val=7
CATCH #### Caught user signal 35 8 times with val=7,
sival_ptr=0x5500000007
***** signal CS Shared buffer = idle task was here!
*****idle thread in CS Shared buffer = signal task was here!
Signal generator exit_thread=0 (differnt copy than idle)
THROW SIGNAL #### Signal 35 thrown with val=8
***** signal CS Shared buffer = idle task was here!
CATCH #### Caught user signal 35 9 times with val=8,
sival_ptr=0x5500000008
*****idle thread in CS Shared buffer = signal task was here!
```

```
Signal generator exit_thread=0 (differnt copy than idle)
THROW SIGNAL #### Signal 35 thrown with val=9
***** signal CS Shared buffer = idle task was here!
CATCH #### Caught user signal 35 10 times with val=9,
sival_ptr=0x5500000009
got expected signals, setting idle exit_thread=1
*****idle thread in CS Shared buffer = signal task was here!
Child idle thread doing a pthread_exit
Signal generator exit_thread=0 (differnt copy than idle)
Idle thread exited, process exiting
Multithreaded child just shutdown

All done
```

`posix_mq.c` is a very simple program that shows how to open a posix message queue, send a message on the queue, and receive it on the queue. This program is executed in a single thread and the output is seen below:

```
$ sudo ./posix_mq
sender opened mq
send: message successfully sent
receive: msg this is a test, and only a test, in the event of a real
emergency, you would be instructed ... received with priority = 30, length
= 95
```

`signal_demo.c` is a simple demo showing how to attach a realtime signal handler onto a process and how to send real time signals to the handler. The program attaches the handler and then sends 9 signals to the handler before sleeping for three seconds and exiting. The output is seen below:

```
$ sudo ./signal_demo
**** CATCH Rx signal: SIGRTMIN+1, value: 1
**** THROW signal: signo=35, value=1
**** CATCH Rx signal: SIGRTMIN+1, value: 2
**** THROW signal: signo=35, value=2
**** CATCH Rx signal: SIGRTMIN+1, value: 3
**** THROW signal: signo=35, value=3
**** CATCH Rx signal: SIGRTMIN+2, value: 1
**** THROW signal: signo=36, value=1
**** CATCH Rx signal: SIGRTMIN+2, value: 2
**** THROW signal: signo=36, value=2
**** CATCH Rx signal: SIGRTMIN+2, value: 3
**** THROW signal: signo=36, value=3
**** CATCH Rx signal: SIGRTMIN+3, value: 1
**** THROW signal: signo=37, value=1
**** CATCH Rx signal: SIGRTMIN+3, value: 2
**** THROW signal: signo=37, value=2
**** CATCH Rx signal: SIGRTMIN+3, value: 3
**** THROW signal: signo=37, value=3
```

POSIX_MQ_LOOP

`posix_mq.c` and `heap_mq.c` have the exact same output so I will cover them in the same section. Both programs open a message queue and start two threads, `sender()` and `receiver()`, that pass messages between each other indefinitely. Sender sends data with a priority of 30 while receiver reads the messages and outputs the message + priority of the message. This loops indefinitely so I will provide a small amount of the output below:

```
$ sudo ./posix_mq

Sender Thread Created with rc=0
sender - thread entry
sender - sending message of size=93
send: message successfully sent, rc=0
sender - sending message of size=93
send: message successfully sent, rc=0
sender - sending message of size=93
send: message successfully sent, rc=0
sender - sending message of size=93
send: message successfully sent, rc=0
sender - sending message of size=93
send: message successfully sent, rc=0
receiver - thread entry
sender - sending message of size=93
send: message successfully sent, rc=0
sender - sending message of size=93
send: message successfully sent, rc=0
sender - sending message of size=93
receiver - awaiting message

 Receiver Thread Created with rc=0
pthread join send
send: message successfully sent, rc=0
sender - sending message of size=93
receive: msg This is a test, and only a test, in the event of real
emergency, you would be instructed.... received with priority = 30, rc =
93
receiver - awaiting message
receive: msg This is a test, and only a test, in the event of real
emergency, you would be instructed.... received with priority = 30, rc =
93
send: message successfully sent, rc=0
sender - sending message of size=93
receiver - awaiting message
receive: msg This is a test, and only a test, in the event of real
emergency, you would be instructed.... received with priority = 30, rc =
93
receiver - awaiting message
send: message successfully sent, rc=0
sender - sending message of size=93
receive: msg This is a test, and only a test, in the event of real
```

```
emergency, you would be instructed.... received with priority = 30, rc =
93
receiver - awaiting message
send: message successfully sent, rc=0
sender - sending message of size=93
receive: msg This is a test, and only a test, in the event of real
emergency, you would be instructed.... received with priority = 30, rc =
93
receiver - awaiting message
send: message successfully sent, rc=0
sender - sending message of size=93
send: message successfully sent, rc=0
sender - sending message of size=93
receive: msg This is a test, and only a test, in the event of real
emergency, you would be instructed.... received with priority = 30, rc =
93
receiver - awaiting message
receive: msg This is a test, and only a test, in the event of real
emergency, you would be instructed.... received with priority = 30, rc =
93
send: message successfully sent, rc=0
sender - sending message of size=93
receiver - awaiting message
```

> A: First, re-write the simple message queue demonstration code so that it uses RT-Linux Pthreads (FIFO) instead of SCHED_OTHER, and then write a brief paragraph describing how the use of a heap message queue and a simple message queue for applications are similar and how they are different.

I have added a new file, `problem-4/POSIX-Examples-Updated/posix_mq_realtime.c`, which rewrites the basic message queue example to spawn the `sender()` and `receiver()` functions as their own posix threads. The output does change in this example compared to the original. This output can be seen below:

```
$ sudo ./posix_mq_realtime
sender opened mq
receive: msg "This is a test, and only a test, in the event of real
emergency, you would be instructed...." received with priority = 30,
length = 93
send: message successfully sent
```

Heap and simple message queues are similar in that they both provide a FIFO queue for passing messages between different parts of the code. This is useful in that it provides a method of passing data that doesn't require a Mutex to synchronize the communication. It also provides multiple messages to queue up so that the receiver can see how the messages/data change over time. The difference is that simple message queues are statically allocated and all messages take up the same amount of space. This improves speed and makes them more deterministic but it reduces the runtime flexibility of the queue. Heap queues dynamically allocate space on the heap for the messages at runtime which makes them much more flexibile

but this increases runtime memory consumption and makes them more deterministic since allocating space on the heap can be slow.

> B: Message queues are often suggested as a better 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?

I don't agree that it fully circumvents the issue of unbounded priority inversion. This is because high priority tasks can still be blocked while waiting on a message to arrive from a lower priority thread that is being indefinitely blocked by a medium priority thread. This is the same exact issue that mutexes have just with extra steps. The workaround for this is providing a timeout for receiving the message so that the higher priority task can still continue on without being fully blocked by the lower priority task.

Exercise 5

> For this problem, consider the Linux manual page on the watchdog daemon - https://linux.die.net/man/8/watchdog [you can read more about Linux watchdog timers, timeouts and timer services in this overview of the Linux Watchdog Daemon].

> A: Describe how it might be used if software caused an indefinite deadlock.

The Linux watchdog is a daemon that must be pet within a specified time window or else it will cause the kernel to restart the system. In the case of an unbounded priority inversion, you could have the lowest priority task in the program pet the watchdog occasionally. If the priority inversion ever occurs, the watchdog task wouldn't be able to run which would eventually cause the system to reset potentially fixing the inversion. This is similar to how I have done it at my previous jobs both on VxWorks and Linux.

> B: 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 ” 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.

I have added a new C file to `problem-5/seqgen3.c` in the repository that implements what is outlined in this problem as well as a capture of its output in the syslogs at `problem-5/problem_5_output.log`. I added in a new mutex named `service_3_mutex` which is locked by the main thread before spawning in the new thread, `Service_3`, which uses a `pthread_mutex_timedlock()` to try and acquire the mutex with a 10 second timeout. Since the mutex is locked by the main thread, it will always hit the 10 second timeout in which it will output to syslogs “No new data available at ”. Below is an example of when `Service_2` is being ran as well as `Service_3`, which is lower priority, is timing out on its mutex:

```
Jun 29 11:37:05 arthur seqgen3[34144]: S1 1 Hz on core 2 for release   20:
lat=70.00000, lon=-80.00000, alt=149.00000, roll=0.91295, pitch=-0.52530,
yaw=0.40808 @ sec=1823098.216962052
Jun 29 11:37:05 arthur seqgen3[34144]: S2 0.1 Hz on core 2 for release  2:
lat=70.00000, lon=-80.00000, alt=149.00000, roll=0.91295, pitch=-0.52530,
yaw=0.40808 @ sec=1823098.216962052
Jun 29 11:37:05 arthur seqgen3[34144]: No new data available at
1823098.217205571

Jun 29 11:37:15 arthur seqgen3[34144]: S1 1 Hz on core 2 for release   30:
lat=80.00000, lon=-100.00000, alt=148.50000, roll=-0.98803, pitch=0.06625,
```

```
yaw=0.15425 @ sec=1823108.217067120
Jun 29 11:37:15 arthur seqgen3[34144]: S2 0.1 Hz on core 2 for release  3:
lat=80.00000, lon=-100.00000, alt=148.50000, roll=-0.98803, pitch=0.06625,
yaw=0.15425 @ sec=1823108.217067120
Jun 29 11:37:15 arthur seqgen3[34144]: No new data available at
1823108.217384602

Jun 29 11:37:25 arthur seqgen3[34144]: S1 1 Hz on core 2 for release   40:
lat=90.00000, lon=-120.00000, alt=148.00000, roll=0.74511, pitch=-0.59836,
yaw=-0.66694 @ sec=1823118.217170225
Jun 29 11:37:25 arthur seqgen3[34144]: S2 0.1 Hz on core 2 for release  4:
lat=90.00000, lon=-120.00000, alt=148.00000, roll=0.74511, pitch=-0.59836,
yaw=-0.66694 @ sec=1823118.217170225
Jun 29 11:37:25 arthur seqgen3[34144]: No new data available at
1823118.217549392

Jun 29 11:37:35 arthur seqgen3[34144]: S1 1 Hz on core 2 for release   50:
lat=100.00000, lon=-140.00000, alt=147.50000, roll=-0.26237,
pitch=0.75983, yaw=0.96497 @ sec=1823128.217260219
Jun 29 11:37:35 arthur seqgen3[34144]: S2 0.1 Hz on core 2 for release  5:
lat=100.00000, lon=-140.00000, alt=147.50000, roll=-0.26237,
pitch=0.75983, yaw=0.96497 @ sec=1823128.217260219
Jun 29 11:37:35 arthur seqgen3[34144]: No new data available at
1823128.217715756
```

You can see that the mutex timeout is a little less accurate than the sequencer based off its timestamp. I believe this is due to using the `CLOCK_REALTIME` clock instead of the `CLOCK_MONOTONIC` clock which is a bit more accurate. Other than this, I believe all requirements for this problem have been satisfied.