

ECEE 5623, Real-Time Systems:
**Exercise #3 – Threading/Tasking and Real-Time
Synchronization**

Michelle Christian: https://github.com/Mich2899/RTES_Exercise3

Question 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.

In real-time OS optimization of memory is one of the major aspects. Often in such systems, two services find it useful to share a memory data segment or code segment. When such a mechanism is used, it is critical to take care that the shared memory is consistent so that one service in the middle of a write is not preempted by another, which could then read partially updated data. Synchronization primitives like semaphores, monitors, Ada rendezvous are used to provide synchronization of otherwise asynchronous tasks to coordinate resource usage such as shared memory. When we have one thread holding a resource and another thread of higher priority blocking on the same resource, the thread holding the resource inherits the blocked threads priority for the duration of critical section (CS). This phenomenon is known as *Priority Inheritance*.

The document in focus: "*Priority Inheritance Protocols: An Approach to Real-Time Synchronization*", provides an in-dept analysis of priority inheritance protocols used so as to minimize the possibilities of **uncontrolled priority inversion**, an instance where a higher priority job is blocked by lower priority jobs for an indeterminate period of time. Such priority inversion poses a serious problem in real-time embedded systems by adversely affecting both the schedulability and predictability of real-time systems. The protocols discussed are *basic priority inheritance protocol* and *priority ceiling protocol*. Both solve the problem of unbounded priority inversion and pride a priority management scheme for synchronization primitives. *Blocking* is a form of priority inversion where a higher priority job must wait for processing of a lower priority job. Prolonged duration of blocking may lead to missing deadlines even when there is negligible resource utilization.

1. The Basic Priority Inheritance Protocol

The basic priority inheritance protocol is that **when a job *J* blocks one or more higher priority jobs, it ignores its original priority assignment and executes its critical section at the highest priority level of all the jobs it blocks. After exiting its critical section, job *J* returns to its original priority level.** The blocking types discussed under this protocol are *direct blocking* and *push-through blocking*. Direct blocking is when a higher priority job attempts to lock a locked semaphore. It is necessary to ensure the consistency of shared data. While, Push-through blocking is when a medium priority job *J1* can be blocked by a low priority job *J2*, which inherits the priority of a higher priority job *J0*. Push-through blocking helps to avoid having a high-priority job *J0* being indirectly preempted by the execution of a medium priority job *J1*.

Definition:

- a) For instance, we assign the highest priority to the given job J , and it attempts to lock semaphore S . If S is already locked, job J is said to be blocked and must wait until S is released. If not, it acquires lock on S and executes its critical section.
- b) Job J uses its assigned priority unless it is in critical section and blocks higher priority jobs. If J blocks higher priority jobs, it inherits P_H , the highest priority of jobs blocked by J .
- c) The important point to be noted is that priority inheritance is transitive. For example, consider we have three jobs, J_1 , J_2 and J_3 with descending priority order. If J_3 blocks job J_2 and J_2 blocks job J_1 then J_3 would inherit the priority of J_1 via J_2 . At the end, the operations of priority inheritance and of the resumption of original priority must be indivisible.
- d) Any job J can preempt another job J_L if J is not blocked and its priority is higher than the priority, inherited or assigned, at which job J_L is executing.

Features of given protocol:

- a) Any higher priority job can be blocked by a lower priority job if it is within its critical section. (The critical section belongs to a set of critical section containing the longest duration of blocking)
- b) Under the basic priority inheritance protocol, a higher priority job can be blocked by a lower priority job for at most the duration of one critical section (of a set of critical section containing the longest duration of blocking), regardless of the number of semaphores given two jobs share.
- c) Given a job J_0 for which there are n lower priority jobs $\{J_1, J_2, \dots, J_n\}$, job J_0 can be blocked for at most duration of one critical section in each of set of critical section for each job respectively.
- d) A semaphore S can cause push-through blocking to job J , only if S is accessed both by a job which has priority lower than that of J and by a job which has or can inherit the priority equal to or higher than that of J .
- e) Under the basic priority inheritance protocol, a job J_i can encounter blocking by at most one critical section in a set of all longest critical sections corresponding to semaphore S_k which can block J_i for each semaphore S_k , $1 \leq k \leq m$, where m is the number of distinct semaphores.
- f) If there are m semaphores which can block job J , then J can be blocked by at most m times.

Problems with given protocol:

The protocol places a bound on the total blocking delay that a job can encounter. Hence, it becomes easier to determine the worst-case blocking duration for a job at compile time. However, the major issue with given protocol is that it does not solve the problem of **deadlock**. Another problem is that, blocking duration for a job, may be bounded but enough that it may miss the deadline because of the phenomenon called **chain blocking**. Deadlock problem may be solved by imposing a total ordering on the semaphore access but chain blocking is not addressed using this protocol.

2. The Priority Ceiling Protocol

Priority Ceiling protocol solves the issues faced with basic priority inheritance protocol, *deadlock* and *blocking chain*. The idea proposed here is to ensure that when a job J preempts the critical section of another job and executes its own critical section z, the priority at which this new critical section z will be executed must be higher than the inherited properties of all preempted critical sections. If not, job J is not allowed to enter critical section z and suspended, and the job that blocks J inherits J's priority.

To implement this protocol, each semaphore is assigned a ceiling priority which is equal to the highest priority task that may use this semaphore. **Job J is permitted to start a new critical section only if J's priority is higher than all priority ceilings of all the semaphores locked by jobs other than J.**

Definition:

- a) For instance, job J is assigned the highest priority among the jobs ready to run, is assigned the processor. S^* is the semaphore with the highest priority ceiling of all semaphores currently locked by jobs other than J. To enter its critical section, J needs to acquire a lock on S. If the priority of J is higher than the priority ceiling of semaphore S, it locks S and executes the critical section. Otherwise, job J is said to be blocked by job which holds lock on semaphore S^* . After J exits the critical section (once it acquires the lock and executes its CS), the related binary semaphore will be unlocked and highest priority job, if any, blocked by J will be resumed.
- b) A job J uses its assigned priority unless it is in its critical section and blocks higher priority jobs. If job J blocks higher priority jobs, J inherits P_H , the highest priority of the jobs blocked by J. When J exits a critical section, it resumes the priority it had at the point of entry into the critical section. Priority inheritance is transitive. Finally, the operations of priority inheritance and of the resumption of previous priority must be indivisible.
- c) Job J, when it does not attempt to enter a critical section, can preempt another job J_L if its priority is higher than the priority, inherited or assigned, at which job J_L is executing.

Additional type of blocking:

Given protocol introduces a third type of blocking in addition to direct blocking and push-through blocking caused by basic priority inheritance protocol. It is referred to as *ceiling blocking*. Using ceiling blocking, one can avoid *deadlock* and *chained blocking*.

Comparison between basic priority inheritance protocol and priority ceiling protocol:

Under the basic priority inheritance protocol, a job J can be blocked for at most the duration of $\min(n, m)$ critical sections, where n is the number of lower priority jobs that could block J and m is the number of semaphores that can be used to block J. On the contrary, under the priority ceiling protocol a job J can be blocked for at most the duration of one longest subcritical section.

Features of given protocol:

- a) Job J can be blocked by a lower priority job J_L only if the priority of job J is no higher than the highest priority ceiling of all the semaphores that are locked by all lower priority jobs when J is initiated.
- b) Suppose that the critical section $z_{j,n}$ of job J_j is preempted by job J_i which enters its critical section $z_{i,m}$. Under the priority ceiling protocol, job J_j cannot inherit a priority level which is higher than or equal to that of job J_i until job J_i completes.
- c) The priority ceiling protocol prevents transitive blocking.
- d) The priority ceiling protocol prevents deadlock.
- e) Let J_L be a job with a lower priority than that of job J_i . Job J_i can be blocked by job J_L for at most the duration of one critical section in the set containing longest critical sections of J_L which can block J_i .
- f) A job J can be blocked for at most the duration of one element of the set containing longest critical sections of lower priority jobs that may block J
- g) If a generalized job J suspends itself n times during its execution, it can be blocked by at most $n+1$ not necessarily distinct elements of the set containing longest critical sections of lower priority jobs that may block J.

3. Schedulability Analysis

The paper provides 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.

Assumptions:

- 1) All tasks are periodic
- 2) Each job in a periodic task has deterministic execution times for both its critical and non-critical sections and that it does not synchronize with external events, i.e., job will execute to its completion when it is the only job in the system
- 3) Periodic tasks are assigned priorities according to the rate-monotonic algorithm

When blocking is not taken into account (only preemption considered):

The document provides conditions for feasibility of RM policy using priority ceiling protocol.

Statement: "A set of n periodic tasks scheduled by RM algorithm can always meet their deadlines if

$$\frac{C_1}{T_1} + \dots + \frac{C_n}{T_n} \leq n(2^{\text{power}(\frac{1}{n})} - 1)$$

Here, C_i and T_i are the execution time and period of given task i respectively."

Statement: "A set of n periodic tasks scheduled by the RM algorithm will meet all their deadlines for all task phasings if and only if

for all i such that, $1 \leq i \leq n$,

$$\min_{(k,l) \in R_i} \sum_{j=1}^i C_j \frac{1}{lT_k} \left\lceil \frac{lT_k}{T_j} \right\rceil = \min_{(k,l) \in R_i} \sum_{j=1}^i U_j \frac{T_j}{lT_k} \left\lceil \frac{lT_k}{T_j} \right\rceil \leq 1$$

Here, C_j , T_j , and U_j are the execution time, period and utilization of task j , respectively and utilization of task j , respectively, and $R_i = \{(k,l) \mid 1 \leq k \leq i, l = 1, \dots, \lfloor T_i/T_k \rfloor\}$

After considering blocking:

Statement: "A set of n periodic tasks using the priority ceiling protocol can be scheduled by RM algorithm if the following conditions are satisfied:

For all i such that, $1 \leq i \leq n$

$$\frac{C_1}{T_1} + \dots + \frac{C_i}{T_i} + \frac{B_i}{T_i} \leq i(2^{\lceil \frac{1}{i} \rceil} - 1)$$

Statement: "A set of n periodic tasks using the priority ceiling protocol can be scheduled by RM algorithm if the following conditions are satisfied:

$$\frac{C_1}{T_1} + \dots + \frac{C_n}{T_n} + \max \left(\frac{B_1}{T_1}, \dots, \frac{B_{n-1}}{T_{n-1}} \right) \leq n(2^{\lceil \frac{1}{n} \rceil} - 1)$$

Statement: "A set of n periodic tasks using the priority ceiling protocol can be scheduled by RM algorithm for all task phasings if

For all i such that, $1 \leq i \leq n$

$$\min_{(k,l) \in R_i} \left[\sum_{j=1}^{i-1} U_j \frac{T_j}{lT_k} \left\lceil \frac{lT_k}{T_j} \right\rceil + \frac{C_i}{lT_k} + \frac{B_i}{lT_k} \right] \leq 1$$

Q1.b) Read the positions of Linux 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.

Excerpt from Linux Torvalds as described by Jonathan Corbet:

Corbet starts by explaining the problem of priority inversion. The document assumes a set of processes, one with a high priority and another one with lower priority that share resources which are protected by locks. When lower priority process acquires a lock on such resource and the higher priority tries to obtain lock on the same resource, it must wait for the lower priority process to end its execution. In such a condition, if a third process which has a priority between other two processes and utilizes a lot of processor time, it will push the low priority process out of the CPU and eventually the higher priority process as well. Such a phenomenon is called “priority inversion.”

The possible approaches to this problem are lockless designs, carefully thought-out locking scenarios, and *priority inheritance*. The concept suggests when a process holds a lock, it should run at (at least) the priority of the highest-priority process waiting got the lock. When a lock is taken by a low-priority process, the priority of that process might need to be boosts until lock is released. Different methods are used to implement priority inheritance. In effect, the kernel performs a very simple form of it by not allowing kernel code to be preempted while holding a spinlock. In some systems, each lock has a priority associated with it; whenever a process takes a lock, its priority is raised to the lock’s priority. In others, a high-priority process will have its priority inherited by another process which holds a needed lock. Most priority inheritance schemes have shown a tendency to complicate and slow down the locking code, and they can be used to paper over poor application designs. So, they are unpopular in many circles. The concept of priority inheritance has been criticized a lot due to complexity and latency.

Even after such opposition, Ingo molnar, member of a kernel development community proposed a priority inheriting futex implementation with a request that it be merged into mainline.

Excerpt from Ingo Molnar and Thomas Gleixner:

The paper provided by Ingo Molnar discusses the “lightweight user-space priority inheritance” (PI) support for futexes and why this approach is a lightweight one. He also explains why PI support is proposed for futexes. The major point Ingo discusses are the benefits of Priority Inheritance (majorly discussed in terms of futexes). The first method discussed is lockless algorithm. Sharing locks between multiple tasks is a common programming practice that are difficult to replace with lockless algorithm. Lockless algorithms, unlike the assumption are very rarely incorporated into the programming. Lockless algorithms are difficult as compared to the locky algorithms, and the complexity often endangers the robustness of code. Critical RT apps often choose lock structures to protect critical data structures, instead of lockless algorithms. In

cases like shared hardware and other resource limits, lockless access is mathematically not feasible. Example of such a system is Media player.

The technical counterarguments against doing priority inheritance mostly apply to kernel-space locks. User-space locks are different, the possibility of disabling interrupts or making the task non-preemptible in a critical section is negligible. Hence, the use of 'spinlocks' argument does not apply. Priority inheritance is pretty much the only technique that currently enables good determinism for user-space locks. For instance if we do not use PI, is a high-priority and a low priority task share a lock, which is common in non-trivial RT applications, even if we carefully code the critical sections to be deterministic, the kernel cannot guarantee any deterministic execution of the high-priority task: any medium-priority task could preempt the low-priority task while it holds the shared lock and executes the critical section, and could delay it indefinitely.

Conclusion:

Based on analysis of both the documents, it can be concluded that priority inversion is very much evident for systems that used shared resources. For optimization of the real-time application it is very advisable to use shared resources for different processes, but the problem of priority inversion comes along with it. Priority inheritance is an infamous but effective approach. The complexity, large overhead and latency do have an impact on the efficiency of the system but problems like deadlocks, livelocks and unbounded priority inversion have a worst effect. It may cause the system to miss deadline for some tasks. Priority inheritance protocols help avoid these situations. Protocols like Basic Priority Inheritance, Highest Locker or Priority Ceiling Emulation protocol and priority ceiling protocol are used for implementing PI. Even after using some of these protocols there are situations where system might run into a deadlock/livelock condition. To resolve these we can,

- I. Serialize multi-resource acquisitions to preclude circular wait.
- II. Use of monitors to detect failure in progress and providing random delay during acquisition
- III. Hardware watchdog, so as to detect that the system fails to provide service

Q1.c) Does the “FUTEX” as an alternative to MUTEX (Futex, Futexes are Tricky), 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?

What are futexes?

A futex (fast user-space mutex) is identified by a piece of memory which can be shared between processes or threads. In these different processes, the futex need not have identical addresses. In its bare form, a futex has semaphore semantics; it is a counter that can be incremented and decremented atomically; processes can wait for the value to become positive. It is a kernel system call that programmers can use to implement basic locking, or as a building block for higher-level locking abstractions such as semaphores and POSIX mutexes or condition variables. The futex() system call provides a method for a program to wait for a value at a given address to change, and a method to wake up anyone waiting on a particular address (while the addresses for the same memory in separate processes may not be equal, the kernel maps them internally so the same memory mapped in different locations will correspond for futex() calls).

This system call is typically used to implement the contended case of a lock in shared memory. When a futex operation did not finish uncontended in user space, a call needs to be made to the kernel to arbitrate. Arbitration can either mean putting the calling process to sleep or, conversely, waking a waiting process. Callers of this function are expected to adhere to the semantics as set out in futex. As these semantics involve writing nonportable assembly instructions, this in turn probably means that most users will in fact be library authors and not general application developers.

Futexes vs mutexes

Futexes operate faster than mutexes because they stay in user-space as much as possible, which means they require fewer system calls, which is inherently faster because the context switch between user and kernel mode is expensive. A kernel mutex doesn't stay in user-space when uncontested, it goes to kernel mode. Any thread operation in a kernel implementation of POSIX threads goes directly to kernel mode because there's no user-space portion of the implementation. Futex is an unsigned 32-bit integer on all platforms. It provides for very fast uncontended lock acquisition and release, as the futex state is store in a user space variable. Atomic operations are used in order to change the state of the futex in the uncontended case without the overhead of a syscall. In contended cases, the kernel is invoked to put tasks to sleep and to wake them up.

int futex(int uaddr, int op, int val, const struct timespec timeout, int uaddr2, int val3)

Using PI-Futex for locking the critical section removes the problem of priority inversion by inheriting the priority of the low priority process in the critical section when a high priority process waits on the futex. Along with the processes, futexes can be used for threads also. Futex is comparatively faster then semaphore, as the later involves kernel in both contended

and non-contended cases, whereas futex involves the kernel only in contended cases. Most of the popular systems have multi-core processors, hence forcing the processes to run on different processors, the priority inversion problem can be avoided.

Question 2) Review the terminology guide (glossary in the textbook).

a) Describe clearly what it means to write "thread safe" functions that are "re-entrant". 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.

What is a reentrant function?

Thread safe code is one that can be performed from multiple threads safely, even if the calls happen simultaneously on multiple threads, because all references to the shared data are serialized. Concurrency is an important factor in thread safe functions.

A function is said to be reentrant if there is a provision to interrupt the function during execution, service the interrupt service routine and then resume the earlier going on function, without hampering its earlier course of action. A reentrant function must be designed so that it can be called for multiple execution contexts (threads) and still provide correct results. A reentrant function must either allocate unique copies of global data for each execution context, provide mutex protection for global data, or avoid the use of global data completely as a pure function. So, reentrant code can be thread safe, but thread safe code is not necessarily reentrant.

The function must satisfy certain conditions to be called as reentrant:

1. It may not use global and static data. Though there are no restrictions, but it is generally not advised. because the interrupt may change certain global values and resuming the course of action of the reentrant function with the new data may give undesired results.
2. It should not modify its own code. This is important because the course of action of the function should remain the same throughout the code. But this may be allowed in case the interrupt routine uses a local copy of the reentrant function every time it uses different values or before and after the interrupt.
3. Should not call another non-reentrant function.

Example of a reentrant function

```
int x;

int my_function() {
    return x * 10;
}

int my_second_function() {
```

```
return my_function() * 20;  
}
```

Q2.b) Describe each method and how you would code it and how it would impact real-time threads/tasks.

What is a pure function?

A function coded so that it only uses stack (no global data at all), depends only in input parameters, and calls only other pure functions is a pure function and is also thread safe. Pure functions are completely independent of outside state and as such they are immune to entire class of bugs that have to do with shared mutable state. Pure functions are extremely independent and hence easy to move around, refactor and reorganize the code structure, making the flow of the code more flexible and adaptable to future changes.

The simplest way to ensure thread safety is to avoid the use of global data and to implement only pure functions that use only local data; however, this may be impractical as it may demand more than available resources. Insufficient stack space and declaration of large arrays as C locals can introduce bugs, but it is thread safe. For real-time applications with resource constraints, pure function implementation may not be feasible.

```
void swap(int *x, int *y) {
    int t = *x;
    *x = *y;

    // `my_func()` could be called here
    *y = t;
}

void my_func() {
    int x = 1, y = 2;
    swap(&x, &y);
}
```

Function which uses thread indexed global data:

It means to allocate unique copies of global data for each execution context. An example of such situations is where functions use a global variable to set an error condition (for example the global variable *errno* used by many functions of the C library). If *errno* were a global variable, a call of a system function on one thread may overwrite the value previously set by a call of a system function on a different thread, possibly before following code on that different thread could check for the error condition. The solution is to have *errno* be a variable that looks like it is global, but in fact exists once per thread—i.e., it lives in thread-local storage. A second use case would be multiple threads accumulating information into a global variable. To avoid a race condition, every access to this global variable would have to be protected by a mutex. Alternatively, each thread might accumulate into a thread-local variable (that, by definition, cannot be read from or written to from other threads, implying that there can be no race conditions). Threads then only must synchronize a final accumulation from their own thread-local variable into a single, truly global variable.

In the Pthreads API, memory local to a thread is designated with the term Thread-specific data. The functions *pthread_key_create* and *pthread_key_delete* are used respectively to create and delete a key for thread-specific data.

```
int t;

void swap(int *x, int *y)
{
    int s;
    s = t;
    t = *x;
    *x = *y;
    *y = t;
    t = s;
}
```


Functions which use shared memory global data, but synchronize access to it using a MUTEX semaphore critical section wrapper:

A Mutex is a specialized lock sometimes called a semaphore (compared to a binary semaphore) which is specifically used to protect critical sections of code for multi-thread safety- this semaphore is used to guarantee mutually exclusive access to a share resource such that only one thread may access a common resource at a time-with shared memory this prevents data corruption that could be caused by multiple readers/ writers. For instance, if a function that writes/ updates the data has partially updated a shared data structure, is preempted/ interrupted, and then a reader function accesses the partially updated data, the data may be completely inconsistent. Mutexes support ownership, recursive access, and task deletion safety.

```
#include <pthread.h>

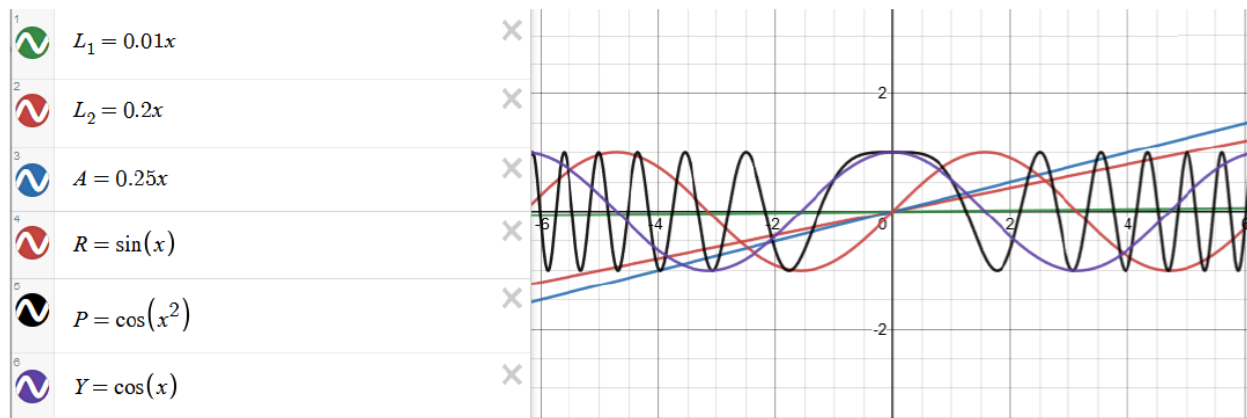
pthread_mutex_t count_mutex;
long long count;

void
increment_count()
{
    pthread_mutex_lock(&count_mutex);
    count = count + 1;
    pthread_mutex_unlock(&count_mutex);
}

long long
get_count()
{
    long long c;

    pthread_mutex_lock(&count_mutex);
    c = count;
    pthread_mutex_unlock(&count_mutex);
    return (c);
}
```

Q1.c) 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 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 using math library function generators (e.g., use simple periodic functions for Roll, Pitch, Yaw $\sin(x)$, $\cos(x^2)$, 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.



Code walkthrough: (uploaded as ex3_2c)

The `#defines` include macros for defining the clock to be used for the time based approach for writing and reading values. The `threadParams` structure contains thread index as an element. The code declares an array of threads so as to use 2 threads, one for read and another for write. The main application of the code starts by defining the structure that contains the timespec structure and the 6 double precision position and attitude state variables i.e. Latitude, Longitude, Altitude, Roll Pitch and Yaw. These variables are functions of time and they are updated in the `write_thread` function that is called by the `write_thread`. The object created using this structure is common to the reader and the writer threads. To protect the data from being corrupted, we use a semaphore that is locked and unlocked using `pthread_mutex_lock` and `pthread_mutex_unlock` library functions.

```

#define MY_CLOCK CLOCK_REALTIME //clock realtime
#define NUM_THREADS 2 //clock realtime
#define THREAD_1 1 //thread index for write thread
#define THREAD_2 2 //thread index for read thread
#define TOTAL_TIME 180 //total time for the application

//thread structure with index as the only element
typedef struct
{
    int thread_idx;
}threadParams_t;

//thread initialization for array of 2 threads
pthread_t threads[NUM_THREADS];
threadParams_t threadParams[NUM_THREADS];

//sched_param
struct sched_param rt_param;

//the below structure contains timespec structure and 6 variables that are updated in the write thread
//thus structure is common between two threads i.e. read and write
typedef struct
{
    struct timespec update; //timespec structure
    double Lat; //latitude
    double Long; //longitude
    double alt; //altitude
    double roll; //roll
    double pitch; //pitch
    double yaw; //yaw
}values;

```

pthread_mutex_t provides a instance to create shared_sem variable that is a mutex used to protect the shared value structure. PTHREAD_MUTEX_INITIALIZER initializes the mutex and the write_value_struct is an object for the values structure that contains all the values to be updated. The three integer variables t_count is used for counting total 180 seconds, t_wcount for displaying the number of times it performs write(every second so 10 times before each read, and t_rcount for counting the number of total reads performed(18).

```

pthread_mutex_t shared_sem=PTHREAD_MUTEX_INITIALIZER; //pthread_mutex_initialized
values write_value_struct; //value structure that is common between two threads

int t_count=1, t_wcount=1, t_rcount=1; //variables for time count

```

Write_thread() function:

```
void* write_thread()
{
    while(t_count<=180)                                //continue the process until 180 seconds
    {
        printf("Writing values in THREAD1 write_no: %d\n",t_wcount);
        pthread_mutex_lock(&shared_sem);                //provide shared semaphore to write thread
        double time=0.00;                                //time variable to store the current time value
        clock_gettime(MY_CLOCK, &write_value_struct.update); //clock_gettime provides the current time update
        time=write_value_struct.update.tv_sec;           //time variable stores the current time
        write_value_struct.Lat=0.01*time;                //update latitude
        write_value_struct.Long=0.2*time;                //update longitude
        write_value_struct.alt=0.25*time;                //update altitude
        write_value_struct.roll=sin(time);               //update roll
        write_value_struct.pitch=cos(time*time);         //update pitch
        write_value_struct.yaw=cos(time);                //update yaw
        printf("timestamp: %d sec %d nsec\n",write_value_struct.update.tv_sec, write_value_struct.update.tv_nsec);
        printf("Written values:\n Latitude: %0.2f Longitude: %0.2f Altitude: %0.2f\n Roll:%0.2f Pitch:%0.2f Yaw: %0.2f\n",
               write_value_struct.Lat, write_value_struct.Long, write_value_struct.alt,
               write_value_struct.roll, write_value_struct.pitch, write_value_struct.yaw);
        pthread_mutex_unlock(&shared_sem);              //unlock or give the shared semaphore
        sleep(1);                                        //wait for 1 second
        t_count++;                                       //increment the total count variable
        t_wcount++;                                     //increment the wcount variable so that once it reaches
                                                         //10 reset
    }
    if(t_count==180){
        exit(-1);
    }
}
```

The write_thread function uses **sleep()** and hence t_count variable counts total 180 times for the whole application. The update is performed until that count and hence the while loop is implemented accordingly. The user is updated every time a write/ update occurs using the t_wcount variable. The **pthread_mutex_lock** locks the semaphore and hence we can access the same structure without the fear of data corruption. The clock_gettime takes the value of time at that instant and hence we can update the state variables by applying the math functions to this time value. The 6 states are updated according to the given functions:

Lat= 0.01*time, Long=0.2*time, alt=0.25*time, roll=sin(time), pitch=cos(time*time) and yaw=cos(time);

As shown in the screenshot below, the write occurs every second and 10 times before every read. For every 10 writes there is one read and hence the output is printed accordingly. The write print count goes upto 180 while the read print count goes upto 18. The value of read is equal to the 10th print i.e. the previous write value.

Read_thread function:

```
void* read_thread()
{
    while(t_count<180)                //continue the process until 180 seconds
    {
        sleep(10);
        pthread_mutex_lock(&shared_sem);    //lock the shared semaphore
        printf("*****READ ALERT*****\n");
        printf("Reading values in THREAD2 read_number= %d\n", t_rcount);
        t_rcount++;
        printf("timestamp: %d sec %d nsec\n", write_value_struct.update.tv_sec, write_value_struct.update.tv_nsec);
        printf("Read values:\n Latitude: %0.2f Longitude: %0.2f Altitude: %0.2f\n Roll:%0.2f Pitch:%0.2f Yaw: %0.2f\n",
                write_value_struct.lat, write_value_struct.long, write_value_struct.alt,
                write_value_struct.roll, write_value_struct.pitch, write_value_struct.yaw);
        pthread_mutex_unlock(&shared_sem);    //unlock the shared semaphore
        printf("*****READ ALERT*****\n");
    }
}
```

The pthread_mutex_lock is used to lock the semaphore and hence not corrupt the common structure. Sleep function is called for 10 seconds, so as to perform a read every 10 seconds. The write_value_struct structure is accessed to print the values of the timestamp state.

OUTPUT:

READ1:

```
Writing values in THREAD1 write_no: 10
timestamp: 1625283134 sec 572079850 nsec
Written values:
Latitude: 16252831.34 Longitude: 325056626.80 Altitude: 406320783.50
Roll:-0.95 Pitch:0.96 Yaw: -0.31
*****READ ALERT*****
Reading values in THREAD2 read_number= 1
timestamp: 1625283134 sec 572079850 nsec
Read values:
Latitude: 16252831.34 Longitude: 325056626.80 Altitude: 406320783.50
Roll:-0.95 Pitch:0.96 Yaw: -0.31
*****READ ALERT*****
```

READ2:

```
Writing values in THREAD1 write_no: 20
timestamp: 1625283144 sec 575397317 nsec
Written values:
Latitude: 16252831.44 Longitude: 325056628.80 Altitude: 406320786.00
Roll:0.97 Pitch:-0.89 Yaw: -0.26
*****READ ALERT*****
Reading values in THREAD2 read_number= 2
timestamp: 1625283144 sec 575397317 nsec
Read values:
Latitude: 16252831.44 Longitude: 325056628.80 Altitude: 406320786.00
Roll:0.97 Pitch:-0.89 Yaw: -0.26
*****READ ALERT*****
```

READ3:

```
Writing values in THREAD1 write_no: 30
timestamp: 1625283154 sec 578627759 nsec
Written values:
Latitude: 16252831.54 Longitude: 325056630.80 Altitude: 406320788.50
Roll:-0.67 Pitch:-0.39 Yaw: 0.74
*****READ ALERT*****
Reading values in THREAD2 read_number= 3
timestamp: 1625283154 sec 578627759 nsec
Read values:
Latitude: 16252831.54 Longitude: 325056630.80 Altitude: 406320788.50
Roll:-0.67 Pitch:-0.39 Yaw: 0.74
*****READ ALERT*****
```

READ 4:

```
Writing values in THREAD1 write_no: 40
timestamp: 1625283164 sec 581876073 nsec
Written values:
Latitude: 16252831.64 Longitude: 325056632.80 Altitude: 406320791.00
Roll:0.16 Pitch:-0.21 Yaw: -0.99
*****READ ALERT*****
Reading values in THREAD2 read_number= 4
timestamp: 1625283164 sec 581876073 nsec
Read values:
Latitude: 16252831.64 Longitude: 325056632.80 Altitude: 406320791.00
Roll:0.16 Pitch:-0.21 Yaw: -0.99
*****READ ALERT*****
```

READ 5:

```
Writing values in THREAD1 write_no: 50
timestamp: 1625283174 sec 585282261 nsec
Written values:
Latitude: 16252831.74 Longitude: 325056634.80 Altitude: 406320793.50
Roll:0.40 Pitch:-0.78 Yaw: 0.91
*****READ ALERT*****
Reading values in THREAD2 read_number= 5
timestamp: 1625283174 sec 585282261 nsec
Read values:
Latitude: 16252831.74 Longitude: 325056634.80 Altitude: 406320793.50
Roll:0.40 Pitch:-0.78 Yaw: 0.91
*****READ ALERT*****
```

READ 6:

```
Writing values in THREAD1 write_no: 60
timestamp: 1625283184 sec 588522159 nsec
Written values:
Latitude: 16252831.84 Longitude: 325056636.80 Altitude: 406320796.00
Roll:-0.84 Pitch:1.00 Yaw: -0.55
*****READ ALERT*****
Reading values in THREAD2 read_number= 6
timestamp: 1625283184 sec 588522159 nsec
Read values:
Latitude: 16252831.84 Longitude: 325056636.80 Altitude: 406320796.00
Roll:-0.84 Pitch:1.00 Yaw: -0.55
*****READ ALERT*****
```

READ 18:

```
Writing values in THREAD1 write_no: 180
timestamp: 1625283304 sec 630403397 nsec
Written values:
Latitude: 16252833.04 Longitude: 325056660.80 Altitude: 406320826.00
Roll:-1.00 Pitch:0.10 Yaw: 0.04
*****READ ALERT*****
Reading values in THREAD2 read_number= 18
timestamp: 1625283304 sec 630403397 nsec
Read values:
Latitude: 16252833.04 Longitude: 325056660.80 Altitude: 406320826.00
Roll:-1.00 Pitch:0.10 Yaw: 0.04
*****READ ALERT*****
Write Thread 76cdc460 done
Read Thread 764db460 done
DONE
```

As observed in the screenshots above, the timestamp and the state values for both the writer thread and the reader thread are same. At the 10th second, the time value in seconds and nanoseconds is same. The 6 state variables values are also same as observed in the output.

Question 3) Download example-sync-updated-2/ and review, build and run it.

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

Deadlock: A multithread condition where 2 or more thread of execution are waiting on resources held by another and the graph of wait-for associations is circular. For instance, if A is waiting on resource R1 to produce resource R2; and B is waiting on resource R2 to produce resource R1 – this condition is a deadlock.

Code walkthtough:

Unlike normal applications, the given code tries to make a deadlock. To explain briefly, we have two resources A and B that are shared between two threads. Pthread mutex initializers initialize the mutex variables. We have a function called grab resources that provides resources to given threads based on their index. For Thread 1, the function provides resource A first and then tries for B while for Thread 2, the function provides resource B first and then tries for A. The flow of the program starts with creating two threads. The deadlock condition that is depicted in the given application can be observed not only in real-time but other applications as well. We use the default scheduling policy threads. There are three conditions to decide the release of resources and flow of the code. This is a user defined input which can be entered into the terminal. These conditions are “safe”, “race” and “unsafe”. Unsafe is the default condition while safe and race are entered by the user. When the code executed in unsafe condition we can observe deadlock. For real-time applications, this condition can cause the system to miss deadlines. When the code is executed in safe mode, it serializes the manner it gives out the resources and hence there is no deadlock. In this condition, we wait for the first thread to complete and then execute the second one. When it is in race condition, we try to join both at the same time and hence there may be situations where both the thread try to grab resources at the same time and end up in deadlock. We can solve such a condition using timeouts, sanity check monitors and hardware such as watchdog timers that restart the system saving from wasting CPU and other resources.

Declare 2 threads:

The code starts by defining the number of threads to be used for the current application. THREAD 1 and THREAD2 define the index of respective threads. We use two pthread mutexes for two threads to access them as per the sequence. There are declared as rsrcA and rsrcB. PTHREAD_MUTEX_INITIALIZER initializes these mutexes to their initial state. rsrcACount and rsrcBcount provide the number of times these resources were accessed. noWait variable is used when we operate the code in race condition. The code can be operated over three conditions:

1. Safe
2. Unsafe
3. Race

For the safe condition, there is no deadlock because there is a delay implemented in the code every time a thread tries to access a resource. When there is no argument provided to operate the code, it runs in unsafe condition. In this situation, the threads enter into a deadlock and the execution is stopped.


```
File Edit Tabs Help
#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 1
#define THREAD_2 2

typedef struct
{
    int threadIdx;
} threadParams_t;

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

struct sched_param nrt_param;

// 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;

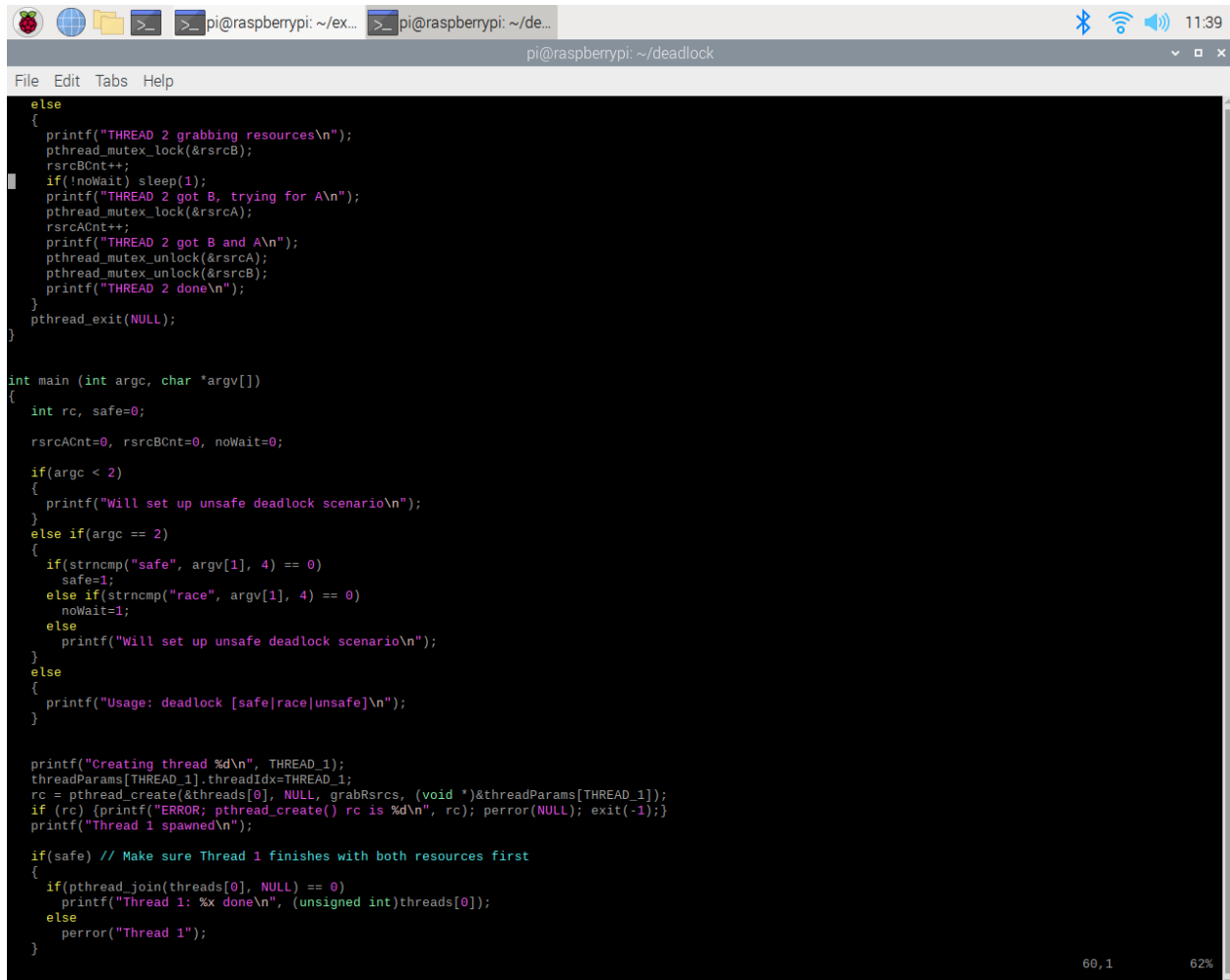
void *grabRsrcs(void *threadp)
{
    threadParams_t *threadParams = (threadParams_t *)threadp;
    int threadIdx = threadParams->threadIdx;

    if(threadIdx == THREAD_1)
    {
        printf("THREAD 1 grabbing resources\n");
        pthread_mutex_lock(&rsrcA);
        rsrcACnt++;
        if(!noWait) 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
    {

```

grabRsrcs function():

Given function takes pthread as an argument. A threadParams_t object stores this thread and its given index to operate on so as to provide resources accordingly. If the thread index is THREAD_1 it attempts to grab resource A first and the resource count for A is incremented. We check if the code is operated in race condition or the safe condition. If it is operated in race condition, noWait is 1 and hence the condition is not satisfied and hence there is no delay/ "no wait". If it is operated by giving a safe argument, it provides a delay of 1 second using sleep function. Once thread 1 grabs resource A, it attempts to grab B. Mind that, both the threads are operating simultaneously in race condition and by the time thread 1 grabbed A, thread 2 may have grabbed B and hence there may be a deadlock situation. If not thread 1 successfully grabs B. Later both are unlocked using pthread_mutex_unlock. "THREAD 1 done" message is printed at the end of thread1 execution. If the thread index passed is THREAD2 the else condition is satisfied.



```
else
{
    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;

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

    printf("Creating thread %d\n", THREAD_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");

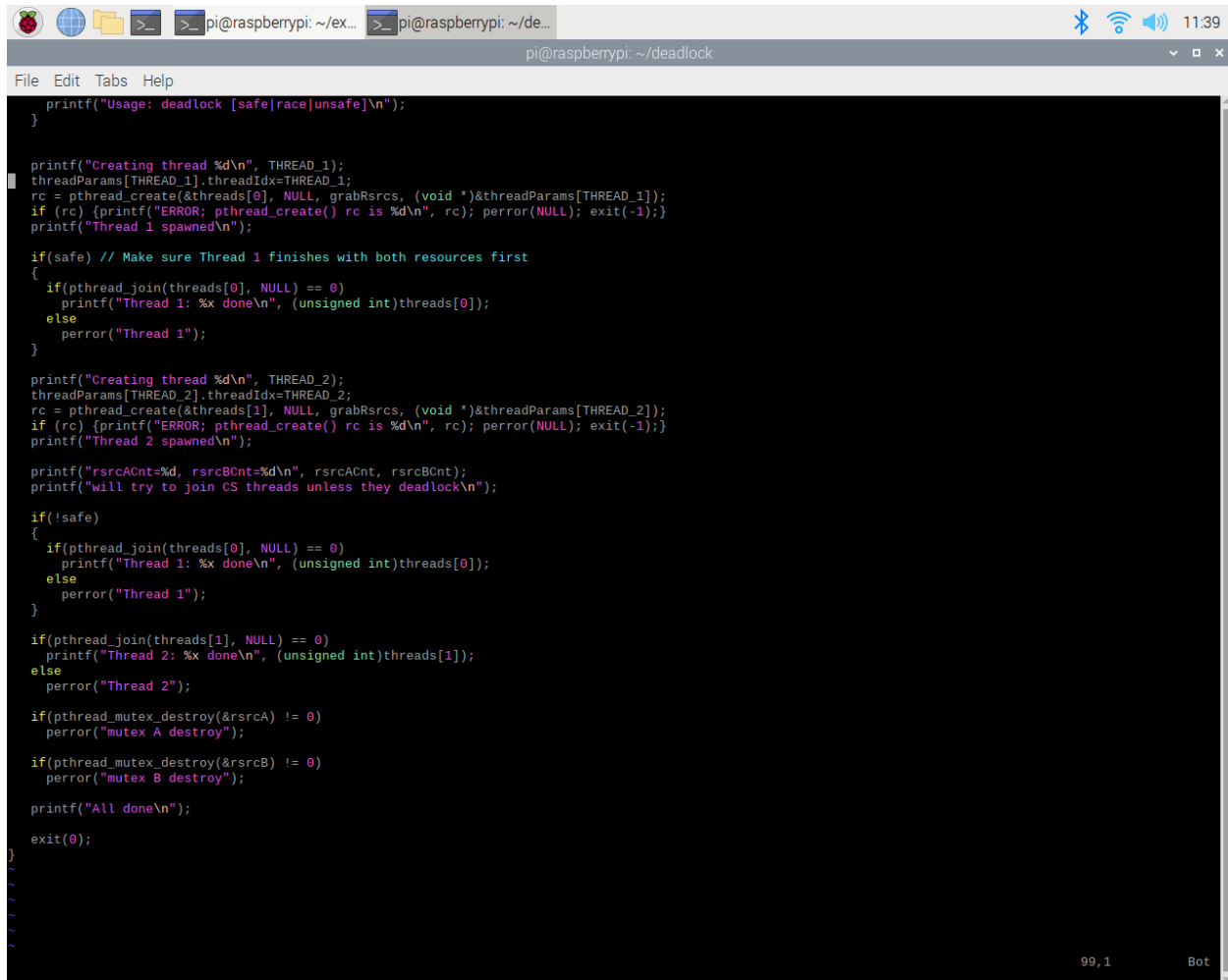
    if(safe) // Make sure Thread 1 finishes with both resources first
    {
        if(pthread_join(threads[0], NULL) == 0)
            printf("Thread 1: %x done\n", (unsigned int)threads[0]);
        else
            perror("Thread 1");
    }
}
```

The procedure for thread 2 is similar the only difference is that it attempts to grab resource B first and then try for resource A.

NOTE: Everytime a resource is grabbed, the respective resource count variable is incremented.

Main function:

The main function takes 2 arguments, **argc** and **argv**. The **argc** integer argument determines if there is any argument passed after the terminal command to run the program - **./filename**. So for the given example the filename is **deadlock**. Hence if the program is run using just **./deadlock**, the argument is just one and hence it will operate as an unsafe condition. A terminal message is also printed to inform the user regarding the same. If there is a string passed after the filename, it is stored in the **argv** variable. The **argv** variable is compared with "safe" and "race" strings to set **safe** and **noWait** variables accordingly. If none of these conditions is satisfied, it goes into the else condition and prints "USAGE: deadlock [safe|race|unsafe]". Threads are created after setting the thread index. Thread 1 is created with index 1 and the **grabRsrcs** function as the routine to be executed. The return value is checked to display if error is detected in creating the thread. We check the **safe** variable so that we can join it first for it to complete its execution before we create and join another thread. If **safe** condition is satisfied, the thread is joined and done message is printed.



```
File Edit Tabs Help
printf("Usage: deadlock [safe|race|unsafe]\n");
}

printf("Creating thread %d\n", THREAD_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);
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");

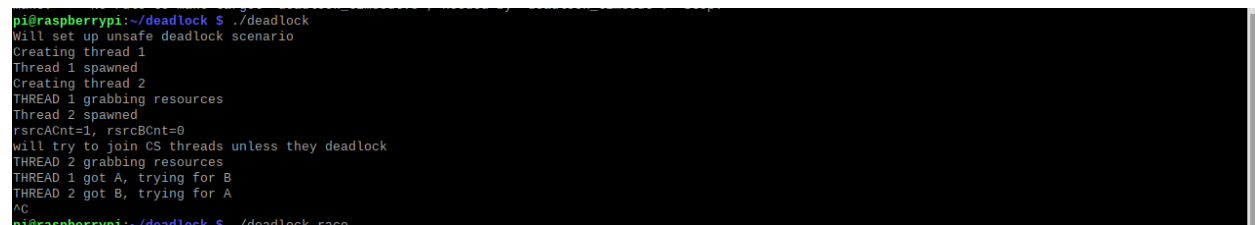
printf("All done\n");

exit(0);
}
~
~
~
~
~
99,1 Bot
```

If safe condition is not satisfied, second thread is created and joined. The resource access counts are printed at the current state and print statement is provided to state the attempt of joining the threads. If the safe condition is not satisfied, thread 1 is joined later. If the execution is successful, the mutexes are destroyed and all done message is printed.

OUTPUT:

UNSAFE condition (Enters deadlock)



```
pi@raspberrypi:~/deadlock $ ./deadlock
Will set up unsafe deadlock scenario
Creating thread 1
Thread 1 spawned
Creating thread 2
Thread 2 spawned
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
^C
pi@raspberrypi:~/deadlock $ ./deadlock race
```

When we operate the code using just “./deadlock”, the threads enter into a deadlock(exit the code using CTRL+C). Both the threads are created, the code attempts to join the threads, Both the threads attempt

to grab resources. Thread 1 ends up grabbing A and tries for B while Thread 2 ends up grabbing B and tries for A. None of them back off and hence the code hangs.

RACE condition:

The “race” argument is passed this time to operate the code. Race condition means both the threads race to grab the resources. This code may operate successfully sometimes and may enter into a deadlock. For this case, both the threads execute successfully.

```
pi@raspberrypi:~/deadlock $ ./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 1 done
THREAD 2 grabbing resources
THREAD 2 got B, trying for A
Thread 2 spawned
rsrcACnt=2, rsrcBCnt=2
will try to join CS threads unless they deadlock
THREAD 2 got B and A
THREAD 2 done
Thread 1: 76d54460 done
Thread 2: 76553460 done
All done
pi@raspberrypi:~/deadlock $ ./deadlock safe
```

SAFE condition:

When we pass the “safe” argument, the code waits for thread 1 to grab one resource, the another one, complete the execution and then starts the execution of thread2. Both the threads get resources and hence the execution is successful.

```
pi@raspberrypi:~/deadlock $ ./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: 76da9460 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: 76da9460 done
All done
pi@raspberrypi:~/deadlock $
```

Pthread3.c:

Code Walkthrough:

The given project depicts a condition of unbounded priority inversion. Unbounded priority inversion causes the system to miss deadlines. It is completely unacceptable for hard real time applications where missing deadlines may result into severe consequences. The code contains services with high(H), low(L) and medium priorities(M). The high priority service and the low priority service share a mutex according to the requirement conditions for unbounded priority inversion. M services that interfere during execution of L in critical section are printed in the code to prove M is interfering and causing the inversion. H enters the critical section only once L exits. The number of M services depends on the parameter passes while executing the program. The more the number of M services the longer it will interfere and H will get blocked for a longer period of time. As M does not share any resources/ semaphore, it will never be in the critical section. The main program starts all these services. The simpleTask function is what the M service executes. It is mainly included in the code to waste time and CPU cycles. Fibonacci sequence is used to burn the CPU time. H and L also perform the same computations but it is based on the length of critical section. *pthread_mutex_lock* and *pthread_mutex_unlock* protect the critical section. H tries to lock the semaphore while L is in critical section attemptinf to get the work done. M tries to execute simple task and interfere with anything lower priority as it computes the sequence over and over.

#defines and global declarations

There are 4 total threads used in the code, one of each of the priority services(H, M and L) and one for the start service. We define the high(H), medium(M) and low(L) priority services and the length of the critical section. Start service is the thread that is created and joined in main. This service starts the other prioritized threads according to the define macro indexes shown below.

```
#define NUM_THREADS      4
#define START_SERVICE    0
#define HIGH_PRIO_SERVICE 1
#define MID_PRIO_SERVICE 2
#define LOW_PRIO_SERVICE 3
#define CS_LENGTH        10
```

Main function:

The main function starts by defining the basic scheduler parameters and the CPU variables. The `threadcpu` and the `core id` variables and the functionalities related to them are present in the code to make sure all the services are running on the same core for the priority inversion to happen. We use `SCHED_FIFO` scheduling policy and hence set the maximum and minimum priority next. The `argc` counts if the user has passed any arguments to be used. If not the code exits. If not, the argument value is saved from `argv` to the interference variable. Initially, the Fibonacci Cycle Burner code is run for testing. Once the core is set we initialize all the `pthread` attribute parameters. If the scheduling parameters are not set, the code exits. After setting the scope, we initialize the shared semaphore. The priority of the start service is set lower than that of H and M, but just a little bit higher than L. Using `pthreadCreate` we create the `startService` thread and wait for it to complete(`join`). Once the execution of all the services is complete we destroy the mutex and print completion message.

startService function:

Start service takes the start service thread as a parameter. The interference time is stored in the `runInterference` variable for the M services to run for that period of time. We use `timestamp` function to determine the start time of the test. We create L with the minimum priority and send `CriticalSectionTask` function as a routine to be executed using this thread. Hence L reaches critical section before H starts. Another `timestamp` depicts the current time at which L is spawned. A while loop is executed that waits until L enters critical section. Thread to execute the high priority service is created, and the same routine, `CriticalSectionTask` is passed for execution. As L is executing in the critical section and has a lock on the semaphore, H gets blocked. `Timestamp` also prints the time at which H is spawned. We create M thread that has a priority lower than that of H but greater than that of L. Hence this thread executes and interferes with L which cannot complete critical section until M is done with any amount of computation(unbounded). We pass `simpleTask` as the routine to be executed from M thread. `Timestamp` prints the time at which M gets spawned. We first `join` H and then if the value of interference time is greater than 0, M is `joined`. We now `join` thread that executes low priority service. Here `join` means wait until the execution of the service the thread is running is completed.

```

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

    printf("Fibonacci Cycle Burner test ...\n");
    fibCycleBurner(47, 2, 1);
    printf("\ndone\n");

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

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

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

```

```

CPU_ZERO(&threadcpu);
coreid=0;
printf("Setting thread %d to core %d\n", i, coreid);
CPU_SET(coreid, &threadcpu);
for(i=0; i<numberOfProcessors; i++)
    if(CPU_ISSET(i, &threadcpu)) printf(" CPU-%d ", i);
printf("\nLaunching thread %d\n", i);

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

pthread_attr_init(&nrt_sched_attr);
pthread_attr_setinheritsched(&rt_sched_attr, PTHREAD_EXPLICIT_SCHED);
pthread_attr_setschedpolicy(&rt_sched_attr, SCHED_RR);
//pthread_attr_setschedpolicy(&rt_sched_attr, SCHED_OTHER);

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

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

```

```

if(scope == PTHREAD_SCOPE_SYSTEM)
    printf("PTHREAD SCOPE SYSTEM\n");
else if (scope == PTHREAD_SCOPE_PROCESS)
    printf("PTHREAD SCOPE PROCESS\n");
else
    printf("PTHREAD SCOPE UNKNOWN\n");

pthread_mutex_init(&sharedMemSem, NULL);

// Set priority lower than H and M, but just a bit higher than L
rt_param.sched_priority = rt_min_prio+1;
pthread_attr_setschedparam(&rt_sched_attr, &rt_param);

printf("\nCreating RT thread %d\n", START_SERVICE);
threadParams[START_SERVICE].threadIdx=START_SERVICE;
rc = pthread_create(&threads[START_SERVICE], &rt_sched_attr, startService, (void *)&threadParams[START_SERVICE]);

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

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

if(pthread_join(threads[START_SERVICE], NULL) == 0)
    printf("START SERVICE joined\n");

```

```

else
    perror("START SERVICE");

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

printf("All threads done\n");

exit(0);
}

```


dTime finds the difference between two timestamps.

```
double dTime(struct timespec now, struct timespec start)
{
    double nowReal=0.0, startReal=0.0;

    nowReal = (double)now.tv_sec + ((double)now.tv_nsec / 1000000000.0);
    startReal = (double)start.tv_sec + ((double)start.tv_nsec / 1000000000.0);

    return (nowReal-startReal);
}
```

fibCycleBurner function:

This function computes Fibonacci sequence based on the seqCnt and the iterCnt parameters passed in the function. traceOn variable is just to display the Fibonacci sequence.

```
void fibCycleBurner(unsigned seqCnt, unsigned iterCnt, int traceOn)
{
    volatile unsigned int fib = 0, fib0 = 0, fib1 = 1;
    int idx, jdx=1;

    if(traceOn) printf("%u %u ", fib0, fib1);

    for(idx=0; idx < iterCnt; idx++)
    {
        fib = fib0 + fib1;
        if(traceOn) printf("%u ", fib);

        while(jdx < seqCnt)
        {
            fib0 = fib1;
            fib1 = fib;
            fib = fib0 + fib1;
            if(traceOn) printf("%u ", fib);
            jdx++;
        }
        jdx=1;
        fib = 0, fib0 = 0, fib1 = 1;
        if(traceOn && (idx < iterCnt-1)) printf("\n\n%u %u ", fib0, fib1);
    }
}
```

simpleTask function:

```
void *simpleTask(void *threadp)
{
    struct timespec timeNow;
    pthread_t thread;
    cpu_set_t cpuset;
    threadParams_t *threadParams = (threadParams_t *)threadp;
    int idleIdx = threadParams->threadIdx, cpucore;

    thread=pthread_self();
    cpucore=sched_getcpu();

    CPU_ZERO(&cpuset);
    pthread_getaffinity_np(thread, sizeof(cpu_set_t), &cpuset);

    do
    {
        fibCycleBurner(fibLength, fibComputeSequences, 0);
        idleCount[idleIdx]++;
        if(idleIdx == LOW_Prio_SERVICE) printf("L%u ", idleCount[idleIdx]);
        else if(idleIdx == MID_Prio_SERVICE) printf("M%u ", idleCount[idleIdx]);
        else if(idleIdx == HIGH_Prio_SERVICE) printf("H%u ", idleCount[idleIdx]);
    } while(idleCount[idleIdx] < runInterference);

    clock_gettime(CLOCK_REALTIME, &timeNow);

    if(idleIdx == LOW_Prio_SERVICE)
        printf("\n**** LOW Prio %d on core %d INTERFERE NO SEM COMPLETED at %lf sec\n", idleIdx, cpucore, dTime(timeNow, timeStartTest));
    else if(idleIdx == MID_Prio_SERVICE)
        printf("\n**** MID Prio %d on core %d INTERFERE NO SEM COMPLETED at %lf sec\n", idleIdx, cpucore, dTime(timeNow, timeStartTest));
    else if(idleIdx == HIGH_Prio_SERVICE)
        printf("\n**** HIGH Prio %d on core %d INTERFERE NO SEM COMPLETED at %lf sec\n", idleIdx, cpucore, dTime(timeNow, timeStartTest));

    pthread_exit(NULL);
}
```

The fibCycleBurner is run for fibLength=47 and fibComputeSequences= 100000 iterations. This loop is executed until the IdleCount with the respective index of the called thread reaches interference time count. Depending on the index of the service passed, message is printed with the time difference between the start of the thread (time it was spawned) and the time at which execution completed. This function is called from the middle priority thread and hence it is just used to waste time/ burn CPU cycles for interference.

criticalSectionTask:

As the name suggests, this function is the critical section that H and L attempt to execute. It takes thread as an argument and executes the thread with respect to the thread index. After acquiring the index of the thread and setting the cpucore, we print the service which requests to execute the function. We lock the mutex and increment the critical section count. Another printf is set to display which priority thread has entered the critical section. The IdleCount of the respective thread is reset. The fibCycleBurner is executed in a loop that operates for the length of critical section (defined as a macro value=10). A print in this section depicts that the service is in critical section. Once this loop is exited, which means service leaves the critical section and hence the mutex is unlocked. A timestamp determines when the service completed the execution of the critical section and a print message displays that the respective service has completed its work in critical section.

```

void *criticalSectionTask(void *threadp)
{
    struct timespec timeNow;
    pthread_t thread;
    threadParams_t *threadParams = (threadParams_t *)threadp;
    int idleIdx = threadParams->threadIdx, cpucore;

    thread=pthread_self();
    cpucore=sched_getcpu();

    if(idleIdx == LOW_PRIO_SERVICE) printf("\nCS-L REQUEST\n");
    else if(idleIdx == MID_PRIO_SERVICE) printf("\nCS-M REQUEST\n");
    else if(idleIdx == HIGH_PRIO_SERVICE) printf("\nCS-H REQUEST\n");

    pthread_mutex_lock(&sharedMemSem);
    CScnt++;

    if(idleIdx == LOW_PRIO_SERVICE) printf("\nCS-L ENTRY %u\n", CScnt);
    else if(idleIdx == MID_PRIO_SERVICE) printf("\nCS-M ENTRY %u\n", CScnt);
    else if(idleIdx == HIGH_PRIO_SERVICE) printf("\nCS-H ENTRY %u\n", CScnt);

    idleCount[idleIdx]=0;

    do
    {
        fibCycleBurner(fibLength, fibComputeSequences, 0);
        idleCount[idleIdx]++;
        if(idleIdx == LOW_PRIO_SERVICE) printf("CS-L%u ", idleCount[idleIdx]);
        else if(idleIdx == MID_PRIO_SERVICE) printf("CS-M%u ", idleCount[idleIdx]);
        else if(idleIdx == HIGH_PRIO_SERVICE) printf("CS-H%u ", idleCount[idleIdx]);
    } while(idleCount[idleIdx] < CS_LENGTH);

```

```

    if(idleIdx == LOW_PRIO_SERVICE) printf("\nCS-L LEAVING\n");
    else if(idleIdx == MID_PRIO_SERVICE) printf("\nCS-M LEAVING\n");
    else if(idleIdx == HIGH_PRIO_SERVICE) printf("\nCS-H LEAVING\n");

    pthread_mutex_unlock(&sharedMemSem);

    if(idleIdx == LOW_PRIO_SERVICE) printf("\nCS-L EXIT\n");
    else if(idleIdx == MID_PRIO_SERVICE) printf("\nCS-M EXIT\n");
    else if(idleIdx == HIGH_PRIO_SERVICE) printf("\nCS-H EXIT\n");

    clock_gettime(CLOCK_REALTIME, &timeNow);

    if(idleIdx == LOW_PRIO_SERVICE)
        printf("\n**** LOW PRIO %d on core %d CRIT SECTION WORK COMPLETED at %lf sec\n", idleIdx, cpucore, dTime(timeNow, timeStartTest));
    else if(idleIdx == MID_PRIO_SERVICE)
        printf("\n**** MID PRIO %d on core %d CRIT SECTION WORK COMPLETED at %lf sec\n", idleIdx, cpucore, dTime(timeNow, timeStartTest));
    else if(idleIdx == HIGH_PRIO_SERVICE)
        printf("\n**** HIGH PRIO %d on core %d CRIT SECTION WORK COMPLETED at %lf sec\n", idleIdx, cpucore, dTime(timeNow, timeStartTest));

    pthread_exit(NULL);
}

```

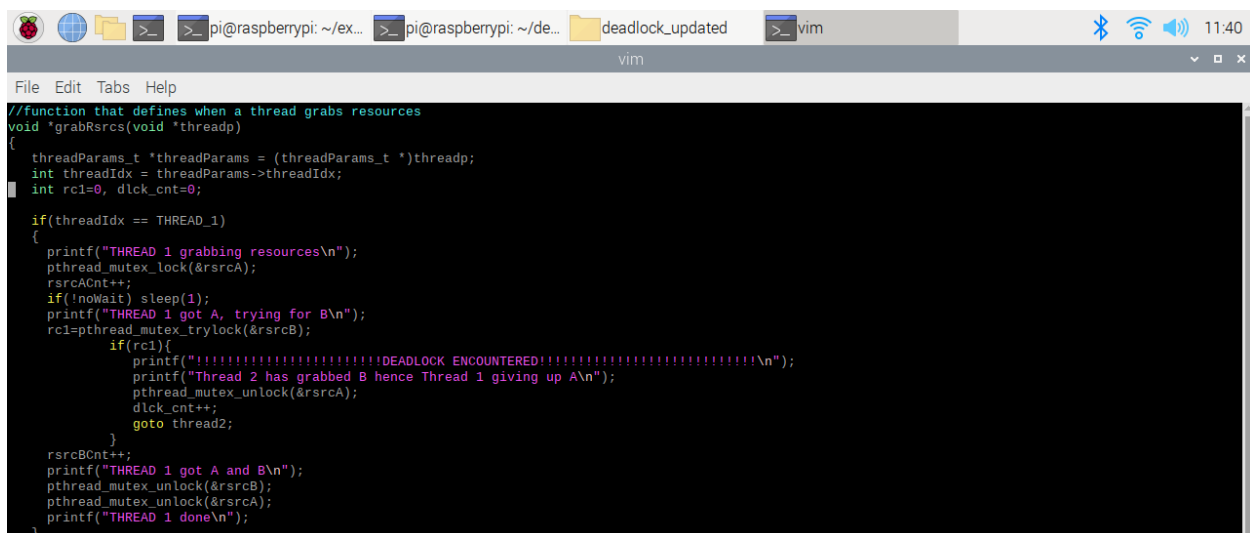
Q3.b) Fix the deadlock so that it does not occur by using a random back-off scheme to resolve. For the unbounded inversion, is there a real fix in Linux – if not, why not?

OTHER RANDOM BACK-OFF Schemes:

If a collision takes place between 2 threads in grabbing resources, they may restart attempts to grab the resources as soon as they can. This will always lead to another deadlock and form an infinite loop leading to livelock. To avoid deadlock and hence livelock we can use random back-off schemes. If we remove necessary conditions, we can have timeouts, sanity monitors and hardware like watchdog timers that restart the entire system, saving the system from wasting resources. There are multiple ways to avoid a deadlock, I have tried to attempt trylock method below. Random backoff is when there is a deadlock condition raised and one of the processes/ threads gives up resource grabbed by them. Hence, the other thread will grab the resource and complete its execution. The previous thread continues its execution by both the resources later and hence deadlock is avoided.

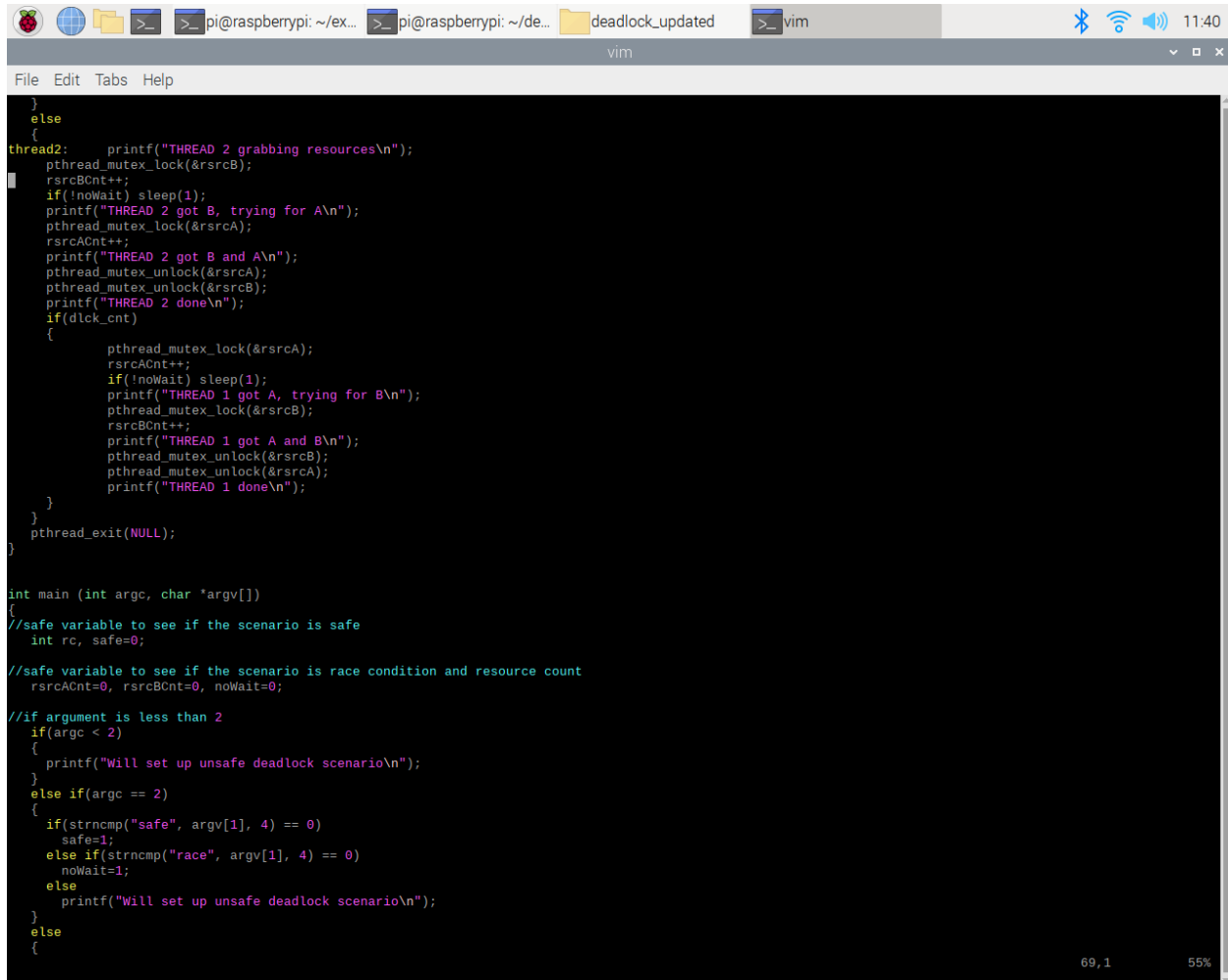
DEADLOCK SOLUTION USING RANDOM BACKOFF:

To avoid the deadlock condition in the unsafe and race conditions, I have used *pthread_mutex_trylock* library function. This function is similar to *pthread_mutex_lock* except that if the mutex object referenced by mutex is currently locked (by any thread, including the current thread), the call shall return immediately. It returns zero if a lock on the mutex object referenced by mutex is acquired. Otherwise, an error number is returned to indicate the error.



```
File Edit Tabs Help
//function that defines when a thread grabs resources
void *grabRsrcs(void *threadp)
{
    pthreadParams_t *threadParams = (pthreadParams_t *)threadp;
    int threadIdx = threadParams->threadIdx;
    int rc1=0, dlock_cnt=0;

    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");
        rc1=pthread_mutex_trylock(&rsrcB);
        if(rc1){
            printf("!!!!!!!!!!!!!!DEADLOCK ENCOUNTERED!!!!!!!!!!!!!!\n");
            printf("Thread 2 has grabbed B hence Thread 1 giving up A\n");
            pthread_mutex_unlock(&rsrcA);
            dlock_cnt++;
            goto thread2;
        }
        rsrcBCnt++;
        printf("THREAD 1 got A and B\n");
        pthread_mutex_unlock(&rsrcB);
        pthread_mutex_unlock(&rsrcA);
        printf("THREAD 1 done\n");
    }
}
```



```

}
else
{
thread2:    printf("THREAD 2 grabbing resources\n");
    pthread_mutex_lock(&srcB);
    rsrcBCnt++;
    if(!noWait) sleep(1);
    printf("THREAD 2 got B, trying for A\n");
    pthread_mutex_lock(&srcA);
    rsrcACnt++;
    printf("THREAD 2 got B and A\n");
    pthread_mutex_unlock(&srcA);
    pthread_mutex_unlock(&srcB);
    printf("THREAD 2 done\n");
    if(dlock_cnt)
    {
        pthread_mutex_lock(&srcA);
        rsrcACnt++;
        if(!noWait) sleep(1);
        printf("THREAD 1 got A, trying for B\n");
        pthread_mutex_lock(&srcB);
        rsrcBCnt++;
        printf("THREAD 1 got A and B\n");
        pthread_mutex_unlock(&srcB);
        pthread_mutex_unlock(&srcA);
        printf("THREAD 1 done\n");
    }
    pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
//safe variable to see if the scenario is safe
    int rc, safe=0;

//safe variable to see if the scenario is race condition and resource count
    rsrcACnt=0, rsrcBCnt=0, noWait=0;

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

```

To implement this function changes are made in the grabRsrcs function. Thread 1 tries to grab resource A, and once successfully that happens, it tries to grab B. For the same, I have used *pthread_mutex_trylock* instead of *pthread_mutex_lock*. The rc1 variable stores the return value of trylock function. If it is successful rc1 will be 0 and if condition will not be satisfied and the execution will be the same. If not, deadlock condition is encountered as thread 2 has grabbed B. Hence, thread 1 gives up resource A and the deadlock flag is set. The code executes for thread 2 and hence the execution for thread 2 is completed. In execution of thread 2 it checks if the deadlock flag is set, we execute the code for thread 1 and hence both the threads are executed successfully.

OUTPUT:

The given output depicts the execution in unsafe mode(deadlock usually occurs in this state).

```
pi@raspberrypi: ~/de... deadlock_updated
pi@raspberrypi:~/ex3_2c $ cd
pi@raspberrypi:~$ cd deadlock_updated/
pi@raspberrypi:~/deadlock_updated$ make
gcc -MD -O3 -g -c deadlock_updated.c
gcc -O3 -g -o deadlock_updated deadlock_updated.o -lpthread -lrt
pi@raspberrypi:~/deadlock_updated$ ./deadlock_updated
Will set up unsafe deadlock scenario
Creating thread 1
Thread 1 spawned
Creating thread 2
Thread 2 spawned
THREAD 1 grabbing resources
rsrccnt=1, rsrcBcnt=0
will try to join CS threads unless they deadlock
THREAD 2 grabbing resources
THREAD 1 got A, trying for B
!!!!!!!!!!!!!!!!!!!!!!!!DEADLOCK ENCOUNTERED!!!!!!!!!!!!!!!!!!!!!!!!!!!!
Thread 2 has grabbed B hence Thread 1 giving up A
THREAD 2 grabbing resources
THREAD 2 got B, trying for A
THREAD 2 got B and A
THREAD 2 done
THREAD 2 got B, trying for A
THREAD 2 got B and A
THREAD 2 done
THREAD 1 got A, trying for B
THREAD 1 got A and B
THREAD 1 done
Thread 1: 76dcb460 done
Thread 2: 765ca460 done
All done
pi@raspberrypi:~/deadlock_updated$
```

SOLUTION FOR UNBOUNDED PRIORITY INVERSION:

The best solution for unbounded priority inversion is priority ceiling protocol. The basic priority inheritance protocol proposes that when a job J blocks one or more higher priority jobs, it ignores its original priority assignment and executes its critical section at the highest priority level of all the jobs it blocks. But this protocol does not solve the deadlock and chained blocking issue. The idea behind priority ceiling protocol is to ensure that when a job J preempts the critical section of another job and executes its own critical section, the priority at which this new critical section will execute is guaranteed to be higher than the inherited priorities of all the preempted critical sections. If this condition cannot be satisfied, job J is denied entry into the critical section and is suspended while the job that blocks J inherits J's priority. The idea is realized by first assigning a priority ceiling to each semaphore, which is equal to the highest priority task that may use this semaphore. We then allow a job J to start a new critical section only if J's priority is higher than all priority ceilings of all the semaphores locked by jobs other than J. The given protocol does solve the issue but the practical implementation of it is not available for Linux. There have been instances where this protocol was implemented for Ada.

Q3.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, but would this really help?

According to Linux kernel developer at Red Hat and maintainer of the stable version of the real-time Linux kernel patch *Steven Rostedt*, the PREEMPT_RT patch (-rt patch or RT patch) makes Linux into a real-time system. However, if the embedded application must meet deadlines, PREEMPT_RT is the way to go. Using PREEMPT_RT, normal kernel provides faster response time and removes unbounded latencies. An unbounded latency is where the amount of delay that can occur is dependent on the situation. Unbounded priority inversion is an example of unbounded latency. The goal of Real-Time OS is to create a predictable and deterministic environment.

PREEMPT_RT is a “hardening” Linux. It is difficult to solve an unbounded latency problem in PREEMPT_RT. Linux kernel is filled with unbounded latencies and hence it is difficult to implement hard-real time applications using Linux. Unbounded latencies bring along non-deterministic behavior. PREEMPT_RT kernel removes jitter and makes the application much smoother. The benefit of PREEMPT_RT over other real-time Linux implementations is that PREEMPT_RT makes Linux itself real-time whereas the others usually create a small micro kernel that runs like a hypervisor and Linux runs as a task. Realtime is much more sensitive to bugs than the mainline kernel and PREEMPT_RT patch helps reliably trigger bugs in mainline that may take months to trigger normally. RT patch implements the concept of priority inheritance. PI algorithm used by RT patch although complex, but it is vigorously tested and used in production environments.

The main aim of the PREEMPT_RT patch is to minimize the amount of kernel code that is non-preemptible. There are different mechanisms implemented that are partially available in mainline linux. With the help of methods like High resolution timers, Threaded interrupt handler, rt_mutex, RCU mechanisms and sleeping spinlocks we can minimize the non-preemptible section in the code. The high resolution timers allow precise thread timed scheduling and removes the dependency of timers on the periodic scheduler tick.

All mutexes in the mainline Linux kernel are replaced by rt_mutexes. A rt_mutex implements priority inheritance to avoid priority inversion.

Q3.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?

The primary approach taken by PREEMPT_RT is to convert the spinlocks into semaphores that allow the blocked process to sleep so that any kernel process that is holding a spinlock can sleep and allow the scheduler to run. This also requires that interrupts be run in high-priority threads instead of directly in interrupt contexts. With the interrupts in a process context, they can sleep as well while still preserving the protection of data between interrupts and kernel threads. The only remaining sections that cannot be preempted are the areas that deal directly with scheduler resources. These sections get a new designation function call, and the number of these new non-preemptible sections is kept to a minimum. The more flexible approach reduces the complexity since there are fewer areas that require great care to not sleep the thread. This does come with an impact from increased overhead. If a section holding a spin lock is brief, the window of time that the thread is sleeping is small. Especially when the overhead cost of saving and restoring contexts is large, this can slow down the response in many cases.

An RTOS is a software that is deterministic behavior. Events are handled within a time constraint. Embedded linux is not an RTOS as events/interrupts are typically handled by deferred procedures or 'bottom half' code which cannot guarantee latency. An RTOS provides scheduling guarantees to ensure deterministic behavior and timely response events and interrupts. In most cases this is through a priority based preemptive scheduling algorithm. Linux has a few scheduling options, including a real-time scheduler, but this is at best soft real time. Linux, prior to the 2.5 kernel, was a non-preemptive kernel. That means, whenever a thread was running in kernel context (a user application making a system call) that thread would not be scheduled out unless it volunteered to schedule (calls the scheduler function). The 2.5 kernel has an option to enable kernel preemption. Kernel preemption has improved reaction time and lowered latencies. Although, kernel preemption has brought Linux one step close to an RTOS.

Hence, using one PREEMPT_RT one can convert the Linux functionalities to real-time. There are still some factors in linux that may not allow us to use it for Hard real time applications. As for the soft real time applications, PREEMPT_RT provides a feasible approach and hence it would be possible to implement SRT using linux.

Question 4) Review heap_mq.c and posix_mq.c and build and run them.

a) First, re-write the VxWorks code so that it uses RT-Linux Pthreads (FIFO) instead of VxWorks tasks, and then write a brief paragraph describing how these two messages queue applications are similar and how they are different. You may find the following Linux POSIX demo code useful as a starting point, but code and test your own solutions.

(Code included in posix_mq and heap_mq folders)

Both posix and heap implement message queues to send and receive data. We use two threads sender and receiver for both the programs. As the name suggests, heap_mq uses heap space for storing the data and hence dynamic memory allocation is used.

Sender function:

Posix_mq.c

The sender_thread calls the sender routine that sends the test message. A message queue instance is created at the start of the function. We use *mq_open* to open the message queue. *mq_open()* creates a new POSIX message queue or opens an existing queue. The queue is identified by name. We have defined the message queue as SNDRCV_MW “/send_receive_mq”. This is the first parameter passed in the open function. The next are the flags that control the call:

O_CREAT: Create the message queue if it does not exist. The owner (user ID) of the message queue is set to the effective user ID of the calling process. The group ownership (group ID) is set to the effective group ID of the calling process.

O_RDWR: Open the queue to both send and receive messages.

The next parameter defines the mode in which the message queue operates and the last defines the attributes of the message queue.

If the message queue is not created it returns error. We send the message using *mq_send* function passing the message queue instance, the message to be sent, the size of the message and the priority at which it is supposed to execute. Messages are placed on the queue in decreasing order of priority, with newer messages of the same priority being placed after older messages with the same priority. If the message is sent successfully we print the success message and move to receiver or else error message is printed.

```

void* sender(void)
{
    mqd_t mymq;
    int prio;
    int nbytes;

    mymq = mq_open(SNDRCV_MQ, O_CREAT|O_RDWR, S_IRWXU, &mq_attr);

    if(mymq < 0)
    {
        perror("sender mq_open");
        exit(-1);
    }
    else
    {
        printf("sender opened mq\n");
    }

    /* send message with priority=30 */
    if((nbytes = mq_send(mymq, canned_msg, sizeof(canned_msg), 30)) == ERROR)
    {
        perror("mq_send");
    }
    else
    {
        printf("send: message successfully sent\n");
    }
    mq_close(mymq);
    if(mymq==(mqd_t)ERROR)
        perror("mqclose");
}

```

Heap_mq.c

The sender function in heap_mq application has some aspects different as compared to posix_mq. Here we use a character array called buffer with size equals to size of a void* + size of an integer variable. The buffptr pointer stores the address at which the message is stored. The id variable makes sure that the message that sender sends is going to the correct receiver. This is the major difference between both the applications: we malloc the message size data in the buffptr. Once the space is allocated we copy the data into the buffptr and print it on the terminal. We use memcpy to store the message in the character array called buffer and we copy the id at the address of the character buffer. We open the message queue in the same manner as posix_mq and send the data with priority 30. If the message is sent successfully, the pointer address is printed and message queue is closed.

NOTE: The message to be communicated is initially stored in imagebuff character array that has a size of 4096.

```

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

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

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

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

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

    mymq = mq_open(Q_NAME, O_CREAT|O_RDWR, S_IRWXU, &mq_attr);

    if(mymq == (mqd_t)ERROR)
        perror("mq_open");

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

    mq_close(mymq);
    if(mymq == (mqd_t)ERROR)
        perror("mq_close");

    sleep(3);
}

```

The mq_send function uses the message queue instance, the character array buffer and the amount of data to be sent is similar to that assigned to the character array buffer. The priority at which the data is sent is 30.

Receiver function:

Posix_mq.c

```
void* receiver(void)
{
    mqd_t mymq;
    char buffer[MAX_MSG_SIZE];
    int prio;
    int nbytes;

    mymq = mq_open(SNDRCV_MQ, O_CREAT|O_RDWR, S_IRWXU, &mq_attr);

    if(mymq == (mqd_t)ERROR)
    {
        perror("receiver mq_open");
        exit(-1);
    }

    /* read oldest, highest priority msg from the message queue */
    if((nbytes = mq_receive(mymq, buffer, MAX_MSG_SIZE, &prio)) == ERROR)
    {
        perror("mq_receive");
    }
    else
    {
        buffer[nbytes] = '\0';
        printf("receive: msg %s received with priority = %d, length = %d\n",
              buffer, prio, nbytes);
    }

    mq_close(mymq);
    if(mymq==(mqd_t)ERROR)
        perror("mqclose");
}
```

The receiver flow is similar to that of sender. Message queue instance created, opened the new message queue using *mq_open* function. For receiving the data we call the *mq_receive* function. A character array called buffer is created with size equal to MAX_MSG_SIZE = 128 as defined in the macro. The buffer value, priority and number of bytes are printed at the receive if no error is encountered and data is received successfully.

Heap_mq.c

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

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

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

    mymq = mq_open(Q_NAME, O_CREAT|O_RDWR, S_IRWXU, &mq_attr);

    if(mymq == (mqd_t)ERROR)
        perror("mq_open");

    if((nbytes = mq_receive(mymq, buffer, (sizeof(void*)+sizeof(int)), &prio)) == ERROR)
/*
*/
    if((nbytes = mq_receive(mymq, (void *)&buffptr, (size_t)sizeof(void *), &prio)) == ERROR)
    {
        perror("mq_receive");
    }
    else
    {
        memcpy(&buffptr, buffer, sizeof(void *));
        memcpy((void *)&id, &(buffer[sizeof(void *)]), sizeof(int));
        printf("receive: ptr msg 0x%X received with priority = %d, length = %d, id = %d\n", buffptr, prio, nbytes, id);

        printf("contents of ptr = \n%s\n", (char *)buffptr);

        free(buffptr);

        printf("heap space memory freed\n");
    }
    mq_close(mymq);
    if(mymq == (mqd_t)ERROR)
        perror("mq_close");
}
```

Similar variables are used for the receiver function as for the sender function. The mq_recieve function is called with buffer as the space to store the received data. If no error is encountered, we copy the received buffer into the address pointed by buffer pointer. We copy the buffer address into id variable to confirm synchronization between sender and receiver.

DIFFERENCES:

The major difference between posix and heap applications is the usage of heap in the heap program. We dynamically allocate memory to store the message in this application which is not one in posix program.

SIMILARITIES:

The implementation and the flow of the program is exactly similar, sender and receiver threads which execute the sender and receiver functions. The mqopen, mqclose, mqsend and mqrecieve functions are used to implement the send_recieve functionality.

Posix_mq.c

Heap_mq.c

[illegible]

Q4.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?

Message queues are used to synchronize tasks and to pass data between them. The enqueue and dequeue operations are thread-safe (a partial-write or read is not possible). Both the types of message queues can be read and written in blocking or non-blocking modes. For blocking modes, when a task does a read on an empty message queue, it blocks until a message is enqueued by another task, allowing it to read and continue. Likewise, when a task tries to write a full message queue in blocking mode, it is blocked until the queue has room for the new message. It is wise to set a timeout upper bound rather than using WAIT_FOREVER so that indefinite blocking will not occur and errors in synchronization and resource management will be detected through timeouts rather than failure to make the progress. For non-blocking message queues, a task does a read on an empty queue, it will be returned the error code EAGAIN, indicating that no messages were available to read and that the task should try again later. Likewise, for a non-blocking message queue, a task would get the same EAGAIN error code if it tried to write to a full queue. This indicates that the writer should try again later, perhaps allowing a read of that queue to create space. The posix message queues include a feature to enqueue messages with a priority level and to read the priority when messages are dequeued. always dequeued first. This essentially allows a sender to put an important message onto the head of the queue.

Message queues implement the enqueue and dequeue function at an atomic level. While for the semaphores/ mutexes, there are possibilities of deadlock, livelock and priority inversion. Message queues do not encounter these problems. Message queues are like named pipes, but without the complexity associated with opening and closing the pipe.

Question 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 kernel can reset the system if serious problems are detected. This can be implemented via special watchdog hardware, or via a slightly less reliable software-only watchdog inside the kernel. Either way, there needs to be a daemon that tells the kernel the system is working fine. If the daemon stops doing that, the system is reset. Watchdog is such a daemon. It opens `/dev/watchdog` and keeps writing to it often enough to keep the kernel from resetting, at least once per minute. Each write delays the reboot time another minute. After a minute of inactivity, the watchdog hardware will cause the reset. In the case of the software watchdog the ability to reboot will depend on the state of the machines and interrupts.

The watchdog daemon can be stopped without causing a reboot if the device `/dev/watchdog` is closed correctly, unless your kernel is compiled with the `CONFIG_WATCHDOG_NOWAYOUT` option enabled.

Watchdog performs certain tests to keep check on sanity of the software. These are:

- Is the process table full?
- Is there enough free memory?
- Are some files accessible?
- Have some files changed within a given interval?
- Is the average workload too high?
- Has a file table overflow occurred?
- Is a process still running? The process is specified by a pid file.
- Do some IP addresses answer to ping?
- Do network interfaces receive traffic?
- Is the temperature too high? (Temperature data not always available.)
- Execute a user defined command to do arbitrary tests.
- Execute one or more test/repair commands found in `/etc/watchdog.d`. These commands are called with the argument `test` or `repair`.

If any of these checks fail watchdog will cause a shutdown. Should any of these tests except the user defined binary last longer than one minute the machine will be rebooted, too.

Q5.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 <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.

Code changes from 2c:

A timespec struct is added to the reader function to provide realtime value. To this we add 10 seconds to the tv_sec element of the timespec variable. We provide this variable to the pthread_mutex_timedlock in place of pthread_mutex_lock. We check the return value of this function so as to determine if the reader thread has successfully acquired the mutex. If not it waits for 10 seconds and prints the message “No new data available at the given <timestamp>”.

```
struct timespec timed;
clock_gettime(MY_CLOCK,&timed);
timed.tv_sec+=10;
timed.tv_nsec+=0;
```

```
rc=pthread_mutex_timedlock(&shared_sem,&timed);           //lock the shares semaphore
if(rc)
{
    printf("No new data available at time %lf sec %lf nsec\n", write_value_struct.update.tv_sec, write_value_struct.update.tv_nsec);
}
else{
    printf("*****READ ALERT*****\n");
    printf("Reading values in THREAD2 read_number= %d\n", t_rcount);
    t_rcount++;
    printf("timestamp: %d sec %d nsec\n", write_value_struct.update.tv_sec, write_value_struct.update.tv_nsec);
    printf("Read values:\n Latitude: %0.2f Longitude: %0.2f Altitude: %0.2f\n Roll:%0.2f Pitch:%0.2f Yaw: %0.2f\n",
           write_value_struct.lat, write_value_struct.long, write_value_struct.alt,
           write_value_struct.roll, write_value_struct.pitch, write_value_struct.yaw);
    pthread_mutex_unlock(&shared_sem);                     //unlock the shared semaphore
    printf("*****READ ALERT*****\n");
}
}
```

The reader thread always receives lock on the mute and hence there will never be an instant where we observe the no new data message.

References:

1. **"REAL-TIME EMBEDDED COMPONENTS AND SYSTEMS with LINUX and RTOS"** by *Sam Siewert and John Pratt*
2. **"Priority Inheritance Protocols: An approach to Real-Time Synchronization"** by *Lui Sha, Raghunathan Rajkumar and John P. Lehoczky*
3. **"SoC drawer: Shared resource management"** by *Dr. Sam Seiwert*.
4. **"Priority inheritance in the kernel" (Linux Torvalds)** as described by *Jonathan Corbet*
5. **"PI-Futex: V1"** by *Ingo Molnar and Thomas Gleixner*.
6. **"Futexes are Tricky"** by *Ulrich Drepper*