Shreyan Prabhu Dhananjayan & Armando Pinales
02/19/2022
Exercise #3
ECEN 5623 - RTES

Question 1

Three key points from “Priority Inheritance Protocols: An Approach to Real-Time Synchronization”:

1. The basic idea of priority inheritance protocols 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 executing its critical section, job *J* returns to its original priority level.
2. The idea of priority ceiling protocol 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 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 *z* and suspended, and the job that blocks *J* inherits *J’s* priority.
3. In particular, the priority ceiling protocol prevents deadlocks and reduces the blocking to at most one critical section.

Three key points from “SoC drawer: Shared resource management (Dr. Siewert’s paper)”:

1. At a high level, an SoC can be characterized by the degree to which it includes each of the following features that emphasize a specific resource:
    a. Data movement
    b. Computation
    c. Data storage

    Typically, an SoC will include a combination of all three of these basic features, but oftentimes one of these features is more significant than the other two and can drive decisions about a system’s memory, I/O, and processing requirements significantly.

2. The necessary conditions for priority inversion are:
    a. Three or more services in the system sharing a CPU
    b. Two services with priority H (high-priority) and L (low-priority) accessing a shared resource.
    c. Prio(H) > prio(M1) > prio(M2) > … > prio(Mn) > prio(L)

    If the software service designer can avoid any of the above conditions, then an unbounded priority inversion will not arise.

3. Avoidance is the key to three protocols that prevent the third condition from persisting. There are:
    a. Priority Ceiling protocol: L is loaned H’s priority for the duration of the shared resource critical section.
    b. Highest Locker or Priority Ceiling Emulation Protocol: L’s priority is amplified to the maximum priority of all the threads involved in the critical section. This semaphore is known as the *highest locker* or *priority ceiling* for this semaphore.

c. Priority Ceiling protocol: A system priority ceiling based upon the maximum priority for ALL semaphore highest lockers is maintained by the OS. Any holder of the system ceiling or higher priority can access its critical section and any other protected resource. If a task is blocked, then the task in the critical section will inherit the blocked task's priority.

"Priority inheritance in the kernel"

The [Linux] kernel performs a very simple form of it [priority inheritance] by not allowing kernel code to be preempted while holding a spinlock. Linus doesn't believe in priority inheritance schemes and believes if you are using one, then your system is already broken. Ingo Molnar created a priority-inheriting futex almost in opposition to Linus' option on the subject.

The PI-futex patch adds a couple of new operations to the futex() system call: FUTEX_LOCK_PI and FUTEX_UNLOCK_PI. (PI = priority inheritance) (futex [fast userspace mutex] = a kernel system call that programmers can use to implement basic locking)

- Uncontended case: a PI-futex can be taken without involving the kernel at all, just like an ordinary futex.
- Contention case: the FUTEX_LOCK_PI operation is requested from the kernel. The requesting process is put into a special queue, and if necessary, that process lends its priority to the process holding the contended futex. The priority inheritance is chained, so that, if the holding process is blocked on a second futex, the boosted priority will propagate to the holder of that second futex. As soon as a futex is released, any associated priority boost is removed.

My stance:
I believe that the PI-Futex is a necessary patch to the kernel, especially if we consider deterministic application support a long-term goal for Linux. Molnar makes an excellent point when stating that PI is necessary more in user-space applications than it is in kernel-space. Molnar's PI-futex patch is incredibly useful for user-space applications that are attempting to run real-time (deterministic) applications. This is something he takes into consideration and why he finds this patch to be necessary. Linus's argument on the other hand is completely dismissive of the need for PI. However, his reasoning could be soley from the perspective of kernel-space applications, where such tools and conventions may not even be necessary. Molnar's distinction between user-space and kernel-space and their need for PI-futex makes his argument strong.

Does the PI-futex that is described by Ingo Molnar provide safe and accurate protection from un-bounded priority inversion as described in the paper? If not, what is different about it?

We believe that the PI-futex patch does indeed provide safe and accurate protection from unbounded priority inversion. The reason for that is that it prevents one of the three necessary conditions for priority inversion to occur (SoC drawer: Shared resource management). Due to the priority propagation which occurs upon processes being preempted in the PI-futex patch, we meet the conditions to avoid unbounded priority inversion.

Question 2

Review the terminology and describe clearly what it means to write "thread safe" functions that are "re-entrant".

Most of the real time services uses common functions so that code space can be reduced by a single function implementation. In the case of RTOS scheduling mechanism, multiple threads may call the same function at the same time. For example, assume a lower priority thread 1 calls a function get_position() which is prempted by higher priority thread 2 which calls the same function get_position() before thread 1 finishes its execution. If function get_position() only consists of local data present in stack, the concurrent call of threads is safe. If the function get_position() consists of global data, there is a chance of global data corruption leading to invalid results.

Sample Code:

```
typedef struct position {double x, y, z;} POSITION;
POSITION satellite_pos = {0.0, 0.0, 0.0};
POSITION get_position(void)
{
double alt, lat, long;
read_altitude(&alt);
read_latitude(&lat);
read_longitude(&long);

satellite_pos.x = update_x_position(alt, lat, long);
satellite_pos.y = update_y_position(alt, lat, long);
satellite_pos.z = update_z_position(alt, lat, long);
return satellite_pos;
}
```

Src: Extracted from Real Time Embedded Components and Systems

In the above example, if thread 1 completes executing the function update_x_position and it is pre-empted by thread 2, thread 2 will update the global variables x, y, z. Before thread 1 got preempted, the alt, lat and long values are stored in the stack. When the thread 1 starts its execution, it will start from the place where it has left off before preemption. So when thread 1 return the satellite_pos structure, the x will be recent value of thread 2 call whereas as the thread 1 will be using the alt, lat and long values stored in stack to compute y and z value. Thus thread 1 will return inconsistent results depending upon when it will be getting preempted by thread 2.

Threadsafe operation and Reentrant function

The above issue can be solved by using thread safe functions where even if a function is called concurrently called by more than 1 thread, the individual thread will be ensured safe execution leading to valid results and deterministic behaviour even when a global data is updated in the function call. A re-entrant function can ensure thread safety. A re-entrant function is one in which during execution, the function may be pre-empted/ interrupted due to an another context but can return to the function to complete its execution without hampering its earlier course of action. A thread safe function can be achieved by using a function that uses only global data or by using a function that uses a mutex semaphore to synchronize shared memory global data or a function which uses thread indexed global data.

Ways to achieve thread safe operation

Pure Function

A Pure function is one that has only stack and no global memory. Whenever a pure function is onvoked, it will be having it own dedicated stack space where the local variables and the function arguments will be stored. Whenever the function terminates, the space allocated will be removed from the memory as the scope of the variables are within the function and their values don't need to be stored throughout the program scope. In real-time threads/tasks, when a thread invokes a function, it will be allocated its own stack space where the variables will be stored. So even when another higher priority thread pre-empts/interrupts another thread with lower priority, it will be having its own stack frame. When a variable is updated or modified, it will be reflected only in its stack frame. The updated local variables can be returned to the calling function where further processing / decisions can be made. This ensures thread safe operation.

In the above example instead of initializing the satellite_pos globally, we can pass the structures as an argument by which the structure member element's memory will be allocated in the stack for each thread. Updation of member elements will take place in the their own stack frame and won't affect the member elements stored in other thread's stack frame. It is easier to implement but making a thread safe using pure functions is always not possible as there may be a requirement where we need to initialize the variable globally.

Sample code

```
typedef struct position {double x, y, z;} POSITION;
POSITION get_position(POSITION *satellite_pos)
{
double alt, lat, long;
read_altitude(&alt);
read_latitude(&lat);
read_longitude(&long);

satellite_pos.x = update_x_position(alt, lat, long);
satellite_pos.y = update_y_position(alt, lat, long);
satellite_pos.z = update_z_position(alt, lat, long);
return satellite_pos;
}
```

Functions that use thread-indexed global data

The local variables are unique to each thread which runs the function. But the global variable are shared by all the threads as we saw in the initial code snippet. Using thread indexed global data, each thread uses thread ID to access global data specific to it. Thus if two threads calls a same function, each thread can access/update their specific global data. The thread specific data is associated with the key. The key is global to all the threads in the process and the thead uses the key to access a pointer (void *) maintained per thread. Since each thread is having global data specific to it, even if a thread is prempted by a higher priority thread, changes in global data (thread-specific) won't affect the other's thread global value as they have spearate memory locations. In Linux user space, thread indexed global data is used for setting errno as multiples threads can encounter different eros which will be update errno variable of each thread. But a problem in thread-index global data, it may overhead (incase of many threads)in terms of storing it in memory as each thread need to store a copy of the variable

Sample code

```
#define NUMTHREADS 4
pthread_key_t variable_key;
int p =1;
void do_something()
{
 //get thread specific data
int* global_specific_variable = pthread_getspecific(variable_key);
printf("Thread %d before mod value is %d\n", (unsigned int) pthread_self(), *global_specific_variable );
*global_specific_variable r += 1;
printf("Thread %d after mod value is %d\n", (unsigned int) pthread_self(), *global_specific_variable );
}

void* get_postion(void *arg)
{
```

```
pthread_setspecific(variable_key, p);
do_something();
do_something();
pthread_setspecific(variable_key, NULL);
free(p);
pthread_exit(NULL);
}
int main(void)
{
pthread_t threads[NUMTHREADS];
int i;
pthread_key_create(&variable_key,NULL);
for (i=0; i < NUMTHREADS; i++)
pthread_create(threads+i,NULL,get_postion,NULL);
 for (i=0; i < NUMTHREADS; i++)
pthread_join(threads[i], NULL);
 return 0;
 }
```

Reference:
https://stackoverflow.com/questions/15100824/how-do-i-create-a-global-variable-that-is-thread-specific-in-c-using-posix-threa

In the above example, a key is created using the pthread_key_create() function. Using pthread_setspecific function, we need to pass the global variable for which we need to have separate copy for each thread along with the key. Using pthread_getspecific, we can get different values according to which thread is being executed at that time. Before exiting, we can use the pthread_setspecific to point the key value to NULL. In the above example, after getting specific thread's value of p, we can increment and check whether the value is increased by 1 for all the invoked threads. After all execution of all the threads, we will get an output of 3 for all the thread specific global variables p as this operation is thread safe where the one thread specific data won't affect the other. Also, we can use _thread keyword like __thread arguments = 3. This will create a different argument variable for each thread which will be intiialized to 3 whenever a new thread is invoked

Using a Mutex Semaphore critical section wrapper

Achieving thread safe operation using Mutex will come in handy incase if our real time services will require a global variable for its execution. The program block where the global data is updated is locked using a mutex and unlocked when the code block finished execution. The code block between the lock and unlock is called the critical section. Whenever a thread executes and obtains the mutex lock, the other thread will wait for the completion of critical section and the mutex unlock. Thus the lock and unlock mechanism prevents the inconsistent state for both the threads as it does not allow preemption to take place when the global data is updated. In real-time service/tasks, this mechanism will lead to thread safe operation but an higher priority tasks has to wait upon a lower priority task to complete its critical section. It leads to a priority inversion where an higher priority task leads to a lower priority task to release the mutex in order for it to run. So while scheduling and calculating the worst case execution time, we need to account for this extra time. Thus, it is better to have the critical section small so tat a thread executes it faster and a high priority task can pre-empt the low priority task.

Sample code

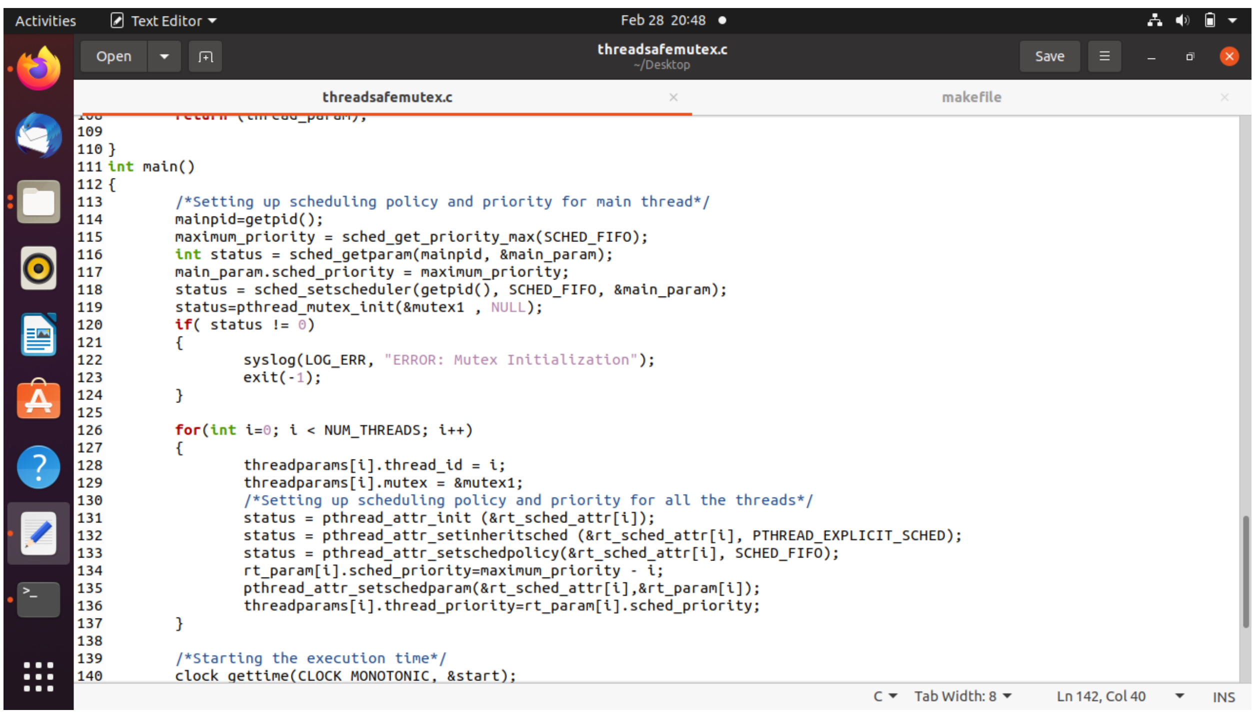
```
#define NUMTHREADS 4
typedef struct position {double x, y, z;} POSITION;
POSITION satellite_pos = {0.0, 0.0, 0.0};
pthread_mutex_t mutex1;
POSITION get_position(void)
{
double alt, lat, long;
read_altitude(&alt);
read_latitude(&lat);
read_longitude(&long);

pthread_mutex_lock(mutex1);

satellite_pos.x = update_x_position(alt, lat, long);
satellite_pos.y = update_y_position(alt, lat, long);
satellite_pos.z = update_z_position(alt, lat, long);

pthread_mutex_unlock(mutex1);
return satellite_pos;
}

int main()
{

POSITION position1;
status = pthread_mutex_init(&mutex1,NULL);
for (i=0; i < NUMTHREADS; i++)
pthread_create(threads+i,NULL,get_position,NULL);
pthread_mutex_destroy(&mutex1);
}
```

Using pthread_mutex_init function, a mutex is initialized. Whenever a thread is invoked and updates the global variable, pthrad_mutex_lock is used before the accessing global data to indicate the start of the critical section and pthread_mutex_unlock is used to indicate the critical section end. pthread-mutex_destroy destroys the mutex object referenced by mutex. This will lead to be thread safe operations as no pre-emmption will take unless the thread completes the execution of the critical section.

Code Explanation

Main function:

In main function, the priorities for main, update and get thread are assigned as 100, 99, 98, 97. This is done because the update thread should be executed so that read thread can fetch updated attitude structure. The SCHED_FIFO scheduling policy is used for scheduling and assigning priorities.

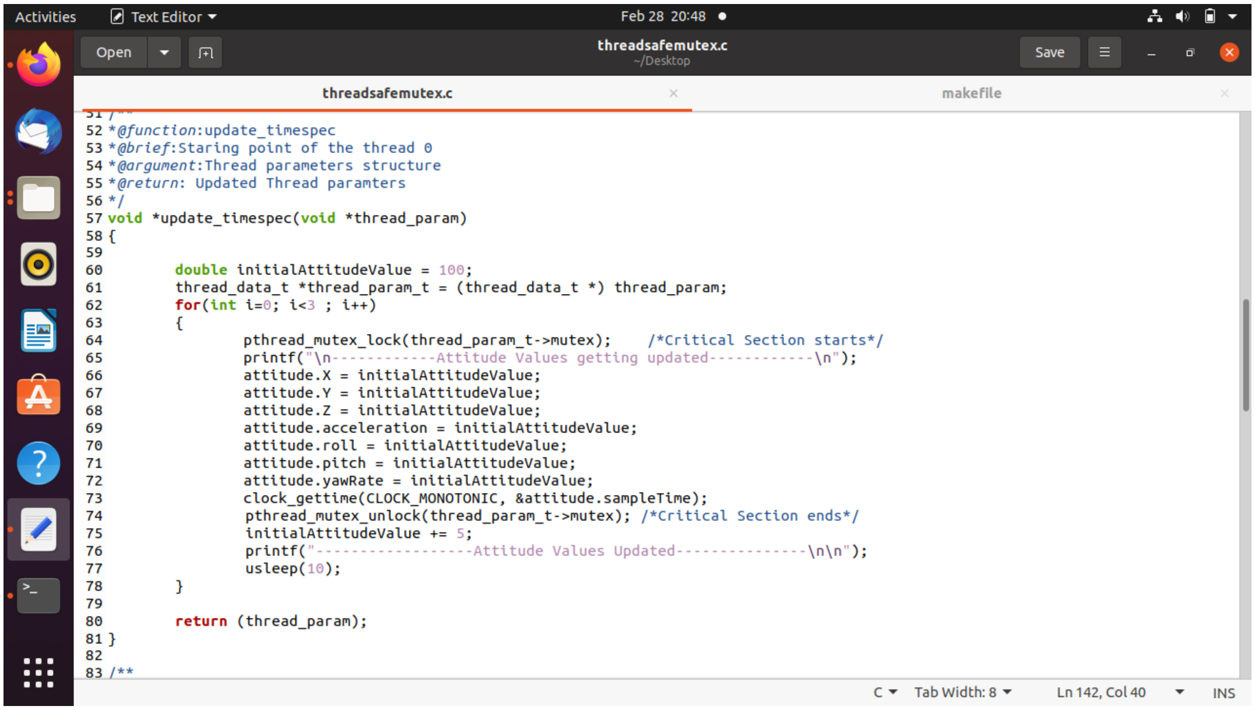


```
}
int main()
{
        /*Setting up scheduling policy and priority for main thread*/
        mainpid=getpid();
        maximum_priority = sched_get_priority_max(SCHED_FIFO);
        int status = sched_getparam(mainpid, &main_param);
        main_param.sched_priority = maximum_priority;
        status = sched_setscheduler(getpid(), SCHED_FIFO, &main_param);
        status=pthread_mutex_init(&mutex1 , NULL);
        if( status != 0)
        {
                syslog(LOG_ERR, "ERROR: Mutex Initialization");
                exit(-1);
        }

        for(int i=0; i < NUM_THREADS; i++)
        {
                threadparams[i].thread_id = i;
                threadparams[i].mutex = &mutex1;
                /*Setting up scheduling policy and priority for all the threads*/
                status = pthread_attr_init (&rt_sched_attr[i]);
                status = pthread_attr_setinheritsched (&rt_sched_attr[i], PTHREAD_EXPLICIT_SCHED);
                status = pthread_attr_setschedpolicy(&rt_sched_attr[i], SCHED_FIFO);
                rt_param[i].sched_priority=maximum_priority - i;
                pthread_attr_setschedparam(&rt_sched_attr[i],&rt_param[i]);
                threadparams[i].thread_priority=rt_param[i].sched_priority;
        }

        /*Starting the execution time*/
        clock_gettime(CLOCK_MONOTONIC, &start);
```

Update thread:

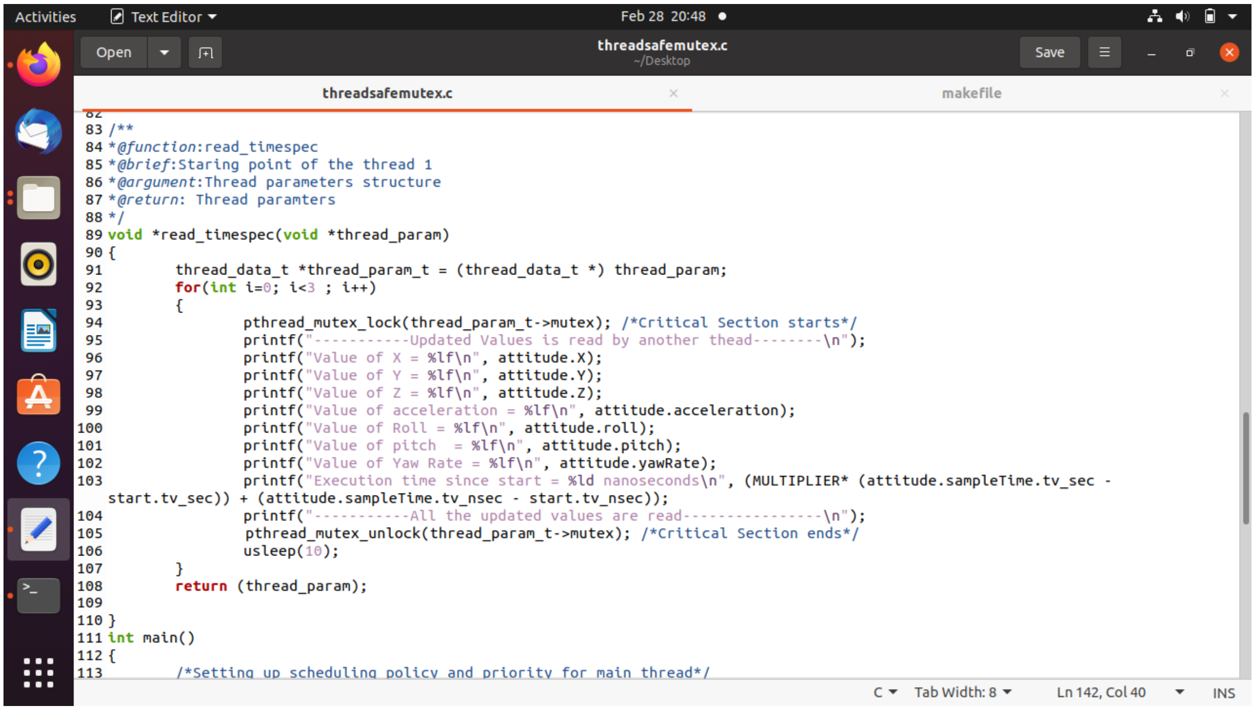
In update thread, the attitude structure value are updated to an arbitrary value. Since the attitude structure is a a global variable, pthread_mutex_lock and pthread_mutex_unlock is used to protect the critical section. This leads to thread safe operation as when read thread is trying to read the structure, it cannot access the data because the lock is acquired by update thread.

Shreyan Prabhu Dhananjayan & Armando Pinales
02/19/2022
Exercise #3
ECEN 5623 - RTES

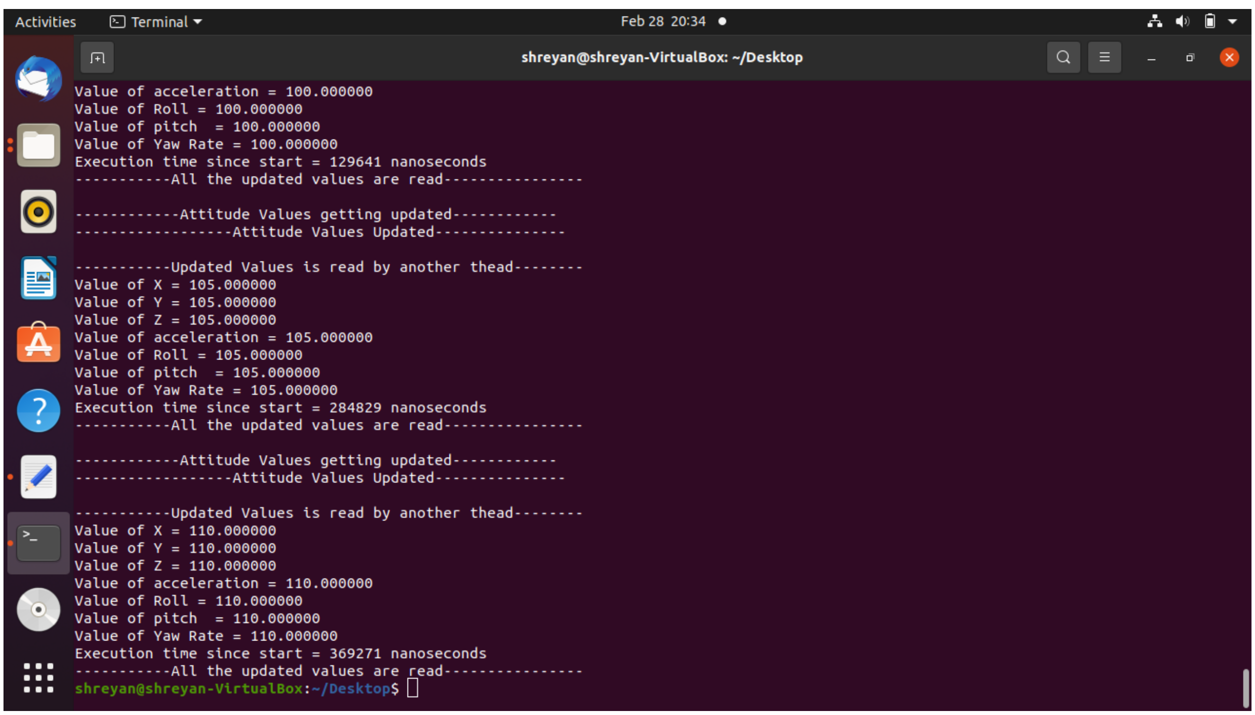


Read Thread:

In read thread, the updated attitude structure value are read. While reading the global m variables it is protected by mutex lock and mutex unlock which ensures thread safe operation.


Activities
Text Editor
Feb 28 20:48
Open
threadsafemutex.c
~/Desktop
Save
threadsafemutex.c
makefile
/**
*@function:read_timespec
*@brief:Staring point of the thread 1
*@argument:Thread parameters structure
*@return: Thread paramters
*/
void *read_timespec(void *thread_param)
{
thread_data_t *thread_param_t = (thread_data_t *) thread_param;
for(int i=0; i<3 ; i++)
{
pthread_mutex_lock(thread_param_t->mutex); /*Critical Section starts*/
printf("----------Updated Values is read by another thead--------\n");
printf("Value of X = %lf\n", attitude.X);
printf("Value of Y = %lf\n", attitude.Y);
printf("Value of Z = %lf\n", attitude.Z);
printf("Value of acceleration = %lf\n", attitude.acceleration);
printf("Value of Roll = %lf\n", attitude.roll);
printf("Value of pitch = %lf\n", attitude.pitch);
printf("Value of Yaw Rate = %lf\n", attitude.yawRate);
printf("Execution time since start = %ld nanoseconds\n", (MULTIPLIER* (attitude.sampleTime.tv_sec - start.tv_sec)) + (attitude.sampleTime.tv_nsec - start.tv_nsec));
printf("----------All the updated values are read---------------\n");
pthread_mutex_unlock(thread_param_t->mutex); /*Critical Section ends*/
usleep(10);
}
return (thread_param);
}
int main()
{
/*Setting up scheduling policy and priority for main thread*/
C
Tab Width: 8
Ln 142, Col 40
INS

Code Output


Activities
Terminal
Feb 28 20:34
shreyan@shreyan-VirtualBox: ~/Desktop
Value of acceleration = 100.000000
Value of Roll = 100.000000
Value of pitch = 100.000000
Value of Yaw Rate = 100.000000
Execution time since start = 129641 nanoseconds
----------All the updated values are read---------------
-----------Attitude Values getting updated-----------
-----------------Attitude Values Updated--------------
----------Updated Values is read by another thead--------
Value of X = 105.000000
Value of Y = 105.000000
Value of Z = 105.000000
Value of acceleration = 105.000000
Value of Roll = 105.000000
Value of pitch = 105.000000
Value of Yaw Rate = 105.000000
Execution time since start = 284829 nanoseconds
----------All the updated values are read---------------
-----------Attitude Values getting updated-----------
-----------------Attitude Values Updated--------------
----------Updated Values is read by another thead--------
Value of X = 110.000000
Value of Y = 110.000000
Value of Z = 110.000000
Value of acceleration = 110.000000
Value of Roll = 110.000000
Value of pitch = 110.000000
Value of Yaw Rate = 110.000000
Execution time since start = 369271 nanoseconds
----------All the updated values are read---------------
shreyan@shreyan-VirtualBox:~/Desktop$

Shreyan Prabhu Dhananjayan & Armando Pinales
02/19/2022
Exercise #3
ECEN 5623 - RTES

Question 3

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

In order to understand the issues of deadlock and unbounded priority, we must first understand the concept of *blocking.* Blocking occurs whenever a service can be dispatched by the CPU, but isn't because it is lacking some other resource. Sometimes, blocking has a known latency, and this can be accounted for in the expected response time of the service, which would as a result, not impact RM analysis. However, the real concern is *unbounded blocking* where the amount of time a service will be blocked is indefinite or completely unknown. There exists three phenomena related to resource sharing which can cause unbounded blocking:

1. Deadlock
2. Livelock
3. Unbounded priority inversion

The example code which causes a deadlock condition (deadlock.c) creates two threads, THREAD_1 and THREAD_2, in that order. THREAD_1 grabs the resource A, rcrcA, and then waits for a fixed period of time. At the same time, THREAD_2 grabs resource B, rcrcB, and then waits for a fixed period of time. After THREAD_A awakens, it attempts to grab rcrcB, however, it has already been obtained by THREAD_2. At some point, THREAD_2 awakens, and attempts to grab rcrcA, but we know that it has already been obtained by THREAD_1. At this point in the program, neither THREAD_1 or THREAD_2 can finish executing because they are both waiting on a resource which is held by the other. This is a deadlock in its purest form.

This deadlock condition can be fixed by using a random backoff scheme. The code for this solution is included in Appendix A, as well as in the included .zip file.

```
arman@raspberrypi:/home/pi/Desktop/ecen5623-rtes/exercise3/q3 $ ./deadlock_timeout
Creating thread 1
Creating thread 2
Thread 1 started
THREAD 1 grabbing resource A @ 1646090495 sec and 336960046 nsec
Thread 1 GOT A
count_rsrcA=1, count_rsrcB=0
THREAD1: 1994519616
THREAD2: 1986126912
Attempting to join both threads.
Thread 2 started
THREAD 2 grabbing resource B @ 1646090495 sec and 337145537 nsec
Thread 2 GOT B
count_rsrcA=1, count_rsrcB=1
THREAD 1 got A, trying for B @ 1646090496 sec and 337174366 nsec
THREAD 2 got B, trying for A @ 1646090496 sec and 337298409 nsec
Thread 2 TIMEOUT ERROR
Thread 1 GOT B @ 1646090498 sec and 337475115 nsec with rv=0
count_rsrcA=1, count_rsrcB=1
THREAD 1 got A and B
THREAD 1 done
Thread 1 joined to main
Thread 2 joined to main
TEST DONE
arman@raspberrypi:/home/pi/Desktop/ecen5623-rtes/exercise3/q3 $
```

Figure 1. Output from code deadlock_timeout.c running on Raspberry Pi. Successfully demonstrating the timeout implementation which allows for the deadlock program to resume computation.

As for the problem of unbounded priority inversion, the RT_PREEMPT patch for linux will fix this issue. However, the patch will not fix bounded priority inversion. (source). We believe it would be naive to not consider linux for real time applications, Patches such as RT_PREEMPT provide programmers with greater control over their applications from a real-time perspective. However, if the requirements of the system fall under the category of "hard real-time", then it would be wise of the designer to consider using a pure RTOS system in that specific case. Soft real-time applications can very easily run on Linux based systems given the software architect is knowledgeable in real-time application theory as well as patches such as RT_PREEMPT for Linux which minimize scheduling and context switching overhead.

Question 4

We will begin by discussing the operation of heap_mq.c and posix_mq.c. heap_mq.c

Shreyan Prabhu Dhananjayan & Armando Pinales
02/19/2022
Exercise #3
ECEN 5623 - RTES

---

heap_mq.c

heap_mp()

heap_mq.c contains four functions in total: **receiver()**, **sender()**, **heap_mq()**, and **shutdown()**. The first function we will discuss is heap_mq(). The heap_mq() function populates a global array named imagebuff[]. imagebuff[] is a char array of size 4096 bytes. The function fills one position in the array with the character 'A' (hex: 0x41), increments the character by 1 (results in 'B' or 0x42), and then writes the next position of the array. This procedure is repeated for a total of 64 times. After the 64th write to the array, the value being written to the array is set to 'A' again, and the procedure repeats for 64 more times. This results in 4096 'pixels' being written to the imagebuff[] array. Once the imagebuff[] array is populated, the application writes the imagebuf to the console via a printf() statement.

The next step of the heap_mq() function is to define message queue attributes. Three are specified: mq_maxmsg = 100, mq_msgsize = sizeof(void *)+sizeof(int), and mq_flags = 0. Once this step is completed, the heap_mq() function opens the message queue by calling mq_open(). The resulting message queue descriptor is stored in the variable mymq, and checked for an error condition. If no error is detected, then the heap_mq() function spawns the receiver task, followed by the sender task. The receiver priority is greater than the sender priority.

receiver()

The receiver() method starts by printing a statement to the console which informs the user how many bytes are about to be read from the message queue. The receiver() function then proceeds to read from the message queue by calling mq_receive. The result (if any bytes are read) is stored into the array named buffer[] of size: sizeof(void *)+sizeof(int) then the buffptr address is set to point to buffer[]. Finally, a console print is executed displaying the message contents, priority, length, and id. Buffptr is free(), and a final print is made notifying the user that the heap memory has been freed. The task then repeats infinitely.

sender()

The sender() method starts by malloc'ing memory the size of imagebuff[] array (4096 bytes). Then the sender() method performs a strcpy of the contents in imagebuff[] to the newly allocated memory. In essence, the copied imagebuff[] contents now live on the heap. The sender() method then proceeds to display some statistics about what is in the buffer and how many bytes are about to be sent. The dynamically allocated memory is then copied to the variable buffer[], which lives on the stack (buffer[] is declared in sender() method). The id is also memcpy'ed to the buffer location 'sizeof(void*)'. Finally, the sender() method attempts to write the buffer contents to the message queue with priority 30. If this is successful, the task then delays for 3 seconds and repeats infinitely.

shutdown()

The shutdown method closes the message queue and performs a taskDelete on the sender and receiver tasks.

posix_mq.c

posix_mq.c is very similar to heap_mq.c except for a few differences First we will discuss the similarities: both contain a sender() and receiver() method. Both contain a function which prepares and creates the sender() and receiver() tasks as well as sets up the message queue. The primary difference between the two is that heap_mq.c uses heap memory to pass via message queue while posix_mq.c passes a message which lives on the stack via the message queue. One other difference is that posix_mq.c exits after sending one message through the message queue while heap_mq.c will sit in an infinite loop sending data via the message queue.

Message queues are used to synchronize tasks and pass data between them. Message queues seem like a potential solution to avoiding mutex priority inversion. One approach that comes to mind is for a service to post a message to the queue notifying other services that it has obtained a certain resource and indicating which other resources it will need. Other services can then respond properly by not obtaining a listed resource and waiting for a new message in the queue to indicate that they are free to obtain the resource. This seems like a potential solution given that message queue operations are also atomic in nature. Message queues can allow various services to synchronize and avoid unintentionally creating deadlocks

```
armando@armando-VirtualBox:~/Desktop/ecen5623-rtes/exercise3/q4$ sudo ./posix_mq
[sudo] password for armando:
Pthread Policy is SCHED_OTHER
PTHREAD SCOPE SYSTEM
Pthread Policy is SCHED_FIFO
PTHREAD SCOPE SYSTEM
rt_max_prio=99
rt_min_prio=1
sender opened mq
send: message successfully sent
receive: msg this is a test, and only a test, in the event of a real emergency, you
 would be instructed ... received with priority = 30, length = 95

TEST COMPLETE
```

Figure 2. Output from posix_mq.c with pthread implementation. Output demonstrates two threads using message queues to communicate information.

Source code is included in Appendix B as well as in the included zip.

```
armando@armando-VirtualBox:~/Desktop/ecen5623-rtes/exercise3/q4$ sudo ./heap_mq
buffer =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~mymq: 3
Pthread Policy is SCHED_OTHER
PTHREAD SCOPE SYSTEM
Pthread Policy is SCHED_FIFO
PTHREAD SCOPE SYSTEM
rt_max_prio=99
rt_min_prio=1
CRETING THREADs
CRETING RECEIVER THREAD
CRETING THREADs
CRETING SENDER THREAD
Reading 8 bytes
Message to send = ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
Sending 8 bytes
receive: ptr msg 0xCC000B60 received with priority = 30, length = 12, id = 999
contents of ptr =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
heap space memory freed
Reading 8 bytes
send: message ptr 0xCC000B60 successfully sent
^X^C
armando@armando-VirtualBox:~/Desktop/ecen5623-rtes/exercise3/q4$
```

Figure 3. Output from pheap_mq.c with pthread implementation in Linux. Output demonstrates two threads using the heap and message queues to communicate information.

Source code is included in Appendix C as well as in the included zip.

Question 5:

A watch dog device consists of a hardware timer will reset the device upon timeout as an hardware reset switch. By default, the watch dog modules are not loaded because some of them start automatically because of which the machine would reboot spontaneously, if the watchdog daemon was not running/configured properly. There is an option of using softdog module to emulate some of the capabilities in software. But it may not help the system to reboot from problem kernel panic, a device driver that prevents a software reboot. The linux kernel watchdog is used to check if the system is running. The system keeps writing to /dev/watchdog which is called kicking or feeding the watchdog. If the system fails to kick or feed the watchdog, the system is hard reset.

Watchdog daemon

The watchdog daemon opens the device and provides the necessary refresh to keep the system from resetting. A watchdog daemon is not very effective without the actual watchdog device. A userspace daemon will inform the kernel watchdog driver via /dev/watchdog special device file that userspace is still alive at periodic intervals. When an update is received, the driver informs the hardware watchdog that everything is in order. If there is any failure like RAM error, kernel bug, the notification stops to occur, the hardware watchdog will reset the system causing a reboot after the timeout occurs. All the drivers support the basic mode of operation, where the watchdog activates as soon as /dev/watchdog is opened and if the watchdog is not pinged whiting a certain time called timeout of margin, the device will reboot.

Using watch dog daemon to prevent deadlock

Thread A, having a higher priority, is holding a resources X and would like resource Y. Thread B, having a lower priority is holding resource Y and would like resource X. Thread A and B can block indefinitely resulting in a deadlock. Schemes like Random Back-Off time, using time deadlock can be a possible solution to solve the deadlock. Watch dog timer can also be used solve deadlock issues. We can program the high priority thread that kicks the watchdog at a regular interval. If the thread is blocked due to a resource, the watchdog won't be kicked and the system will reboot. In this way, the watchdog thread can monitor any number of individual threads and if two of the threads resutled in a deadlock, the watch dog time will reboot. The watch dog will not kick a lower priority task. This is because if an higher priority task is executing, the system should not reboot because the lower priority task has timedout. This will cause the higher priority task to miss a deadline resulting in system failure. One of the easier ways to kick a watchdog it to create a thread. Thus instead of running the thread in foreground proces, it should starts a background process ie daemon

Configuring and start/stop/test watch dog daemon

A watchdog daemon has 4 settings related to the watch dog device, they are

watchdog-device = /dev/watchdog

watchdog-timeout = 60

interval = 1

sigterm-delay = 5

The initial parameters define the API point and timeout to be configured. The third parameter is the polling interval. Its default value is 1 second. Generally it must be atleast 2 seconds less than the timeout of the hardware because

- The timer hardware is not synchronized to the polling period
- The health checks could take a significant fraction to run and the interval value is the sleep time between loops.

Configuration of watchdog daemon is important because it can cause problems like

- Endless reboot loop

- File corruption due to hard reset
- Unpredictable random reboots

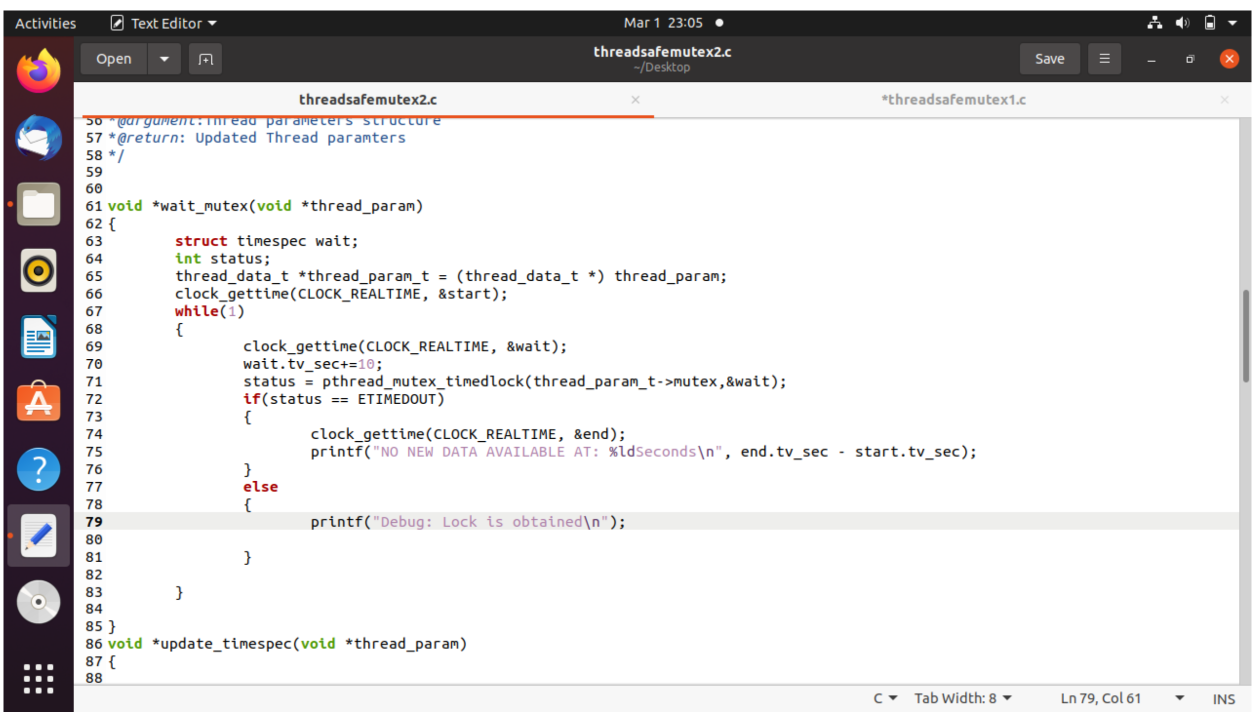
The watchdog daemon should start at boot time and runs in the background. ps-af | grep watch* is used to check if it is running. cat >> /dev/watchdog used to test the watchdog daemon and depending upon kernel setting, the system may perform hard reboot

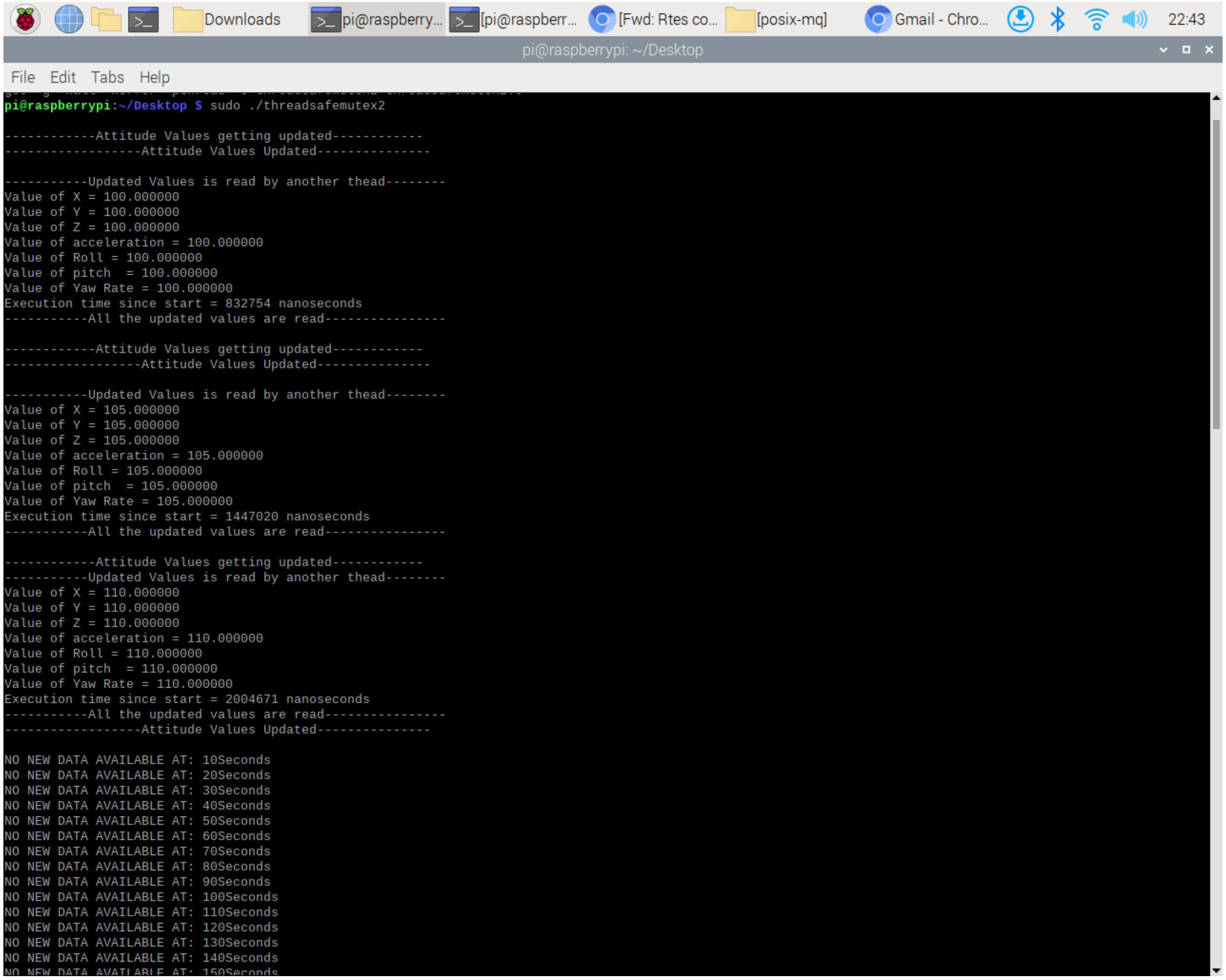
References:

https://www.quora.com/If-you-ever-implemented-a-watchdog-mechanism-under-an-RTOS-how-did-you-do-that-Did-your-mechanism-protect-against-a-deadlock

Exploring timeouts

The second code has been modified where a new thread wait_mutex is created using pthread_create. Before the second thread is created, the mutex is locked to implement the functionality pthread_mutex_timedlock(). For verifying time has elapsed 10 seconds, clock_gettime is used. Since the wait_mutex has to be executed in a indefinite number of times, it has been executed in a while(1). The wait sec timespec is incremented by 10. The pthread_mutex_timedlock is called and it waits till the wait timespec has been increased by 10 seconds. If the function return ETIMEDOUT, then the timeout is reached and loops back it in to the function again. As we can see in the code output the update, read thread executes and it loops back to wait thread code





Question 5 o/p

Code:

Appendix A

```
/**
 * @file deadlock_timeout.c
 * @author Armando Pinales
 * @author Shreyan Prabhu Dhananjayan
 * @brief This code implements the random back-off solution for blocking
 *        conditions. This code is built off of code written by Sam Siewert as
 *        well as heavily referencing Sam Siewert's solution titled
 *        "deadlock_timeout.c".
```

```
 * @version 0.1
 * @date 2022-02-24
 *
 * @copyright Copyright  Sam Siewert(c) 2022
 *
 */

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

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

// create a structure to pass args to thread
typedef struct
{
    int thread_id;
} threadParams_t;

// create an array of the arg structures and an array of type pthread to hold
// thread IDs.
threadParams_t thread_args[NUM_THREADS];
// pthread_t threads[NUM_THREADS];
pthread_t thread1, thread2;

// create the mutexes to be used by THREAD1 and THREAD2
pthread_mutex_t rsrcA;
pthread_mutex_t rsrcB;
```

```
volatile int count_rsrcA = 0;
volatile int count_rsrcB = 0;
volatile int no_wait   = 0;

// function decleration
void *get_resources(void *args);


int main (int argc, char *argv[])
{
  int rv;
  count_rsrcA = 0;
  count_rsrcB = 0;
  no_wait     = 0;

  // Set default protocol for mutex which is unlocked to start
  if(pthread_mutex_init(&rsrcA, NULL) != 0)
  {
    printf("Error initializing mutex rsrcA\r\n");
  }
  if(pthread_mutex_init(&rsrcB, NULL) != 0)
  {
    printf("Error initializing mutex rsrcB\r\n");
  }

  // CREATE THREAD 1
  printf("Creating thread %d\n", THREAD_1);
  thread_args[0].thread_id = THREAD_1;
  rv = pthread_create(
    &thread1,
    NULL,
    get_resources,
    (void *)&thread_args[0]
  );
  if(rv)
```

```
{
   printf("ERROR; pthread_create() rv is %d\n", rv);
   perror(NULL);
   exit(-1);
}



// CREATE THREAD 2
printf("Creating thread %d\n", THREAD_2);
thread_args[1].thread_id=THREAD_2;
rv = pthread_create(
  &thread2,
  NULL,
  get_resources,
  (void *)&thread_args[1]
);
if(rv)
{
  printf("ERROR; pthread_create() rv is %d\n", rv);
  perror(NULL);
  exit(-1);
}

printf("THREAD1: %lu\r\nTHREAD2: %lu\r\n", thread1, thread2);

printf("Attempting to join both threads.\n");
// ATTEMPT TO JOIN THREAD1
if(pthread_join(thread1, NULL) == 0)
{
  printf("Thread 1 joined to main\n");
}
else
{
  perror("Thread 1");
}
```

```
  // ATTEMPT TO JOIN THREAD2
  if(pthread_join(thread2, NULL) == 0)
  {
    printf("Thread 2 joined to main\n");
  }
  else
  {
    perror("Thread 2");
  }

  // Finally, destroy both mutexes
  if(pthread_mutex_destroy(&rsrcA) != 0)
  {
    perror("mutex A destroy");
  }
  if(pthread_mutex_destroy(&rsrcB) != 0)
  {
    perror("mutex B destroy");
  }



  printf("TEST DONE\n");

  return 0;
}



// Function to be run for threads,
void *get_resources(void *args)
{
  // declare necessary variables
  int rv;
  struct timespec current_time;
  struct timespec timeout_rsrcA;
  struct timespec timeout_rsrcB;
```

```
// cast the thread arguments and save the thread_id
threadParams_t *thread_args = (threadParams_t *)args;
int thread_id = thread_args->thread_id;

// Debug prints
if(thread_id == THREAD_1)
{
  printf("Thread 1 started\n");
}
else if(thread_id == THREAD_2)
{
  printf("Thread 2 started\n");
}
else
{
  // big error if we end up here
  printf("Unknown thread started\n");
}

// get the current clock time
clock_gettime(CLOCK_REALTIME, &current_time);

// Create timeouts for rsrcA and rsrcB
timeout_rsrcA.tv_sec  = current_time.tv_sec + 2;
timeout_rsrcA.tv_nsec = current_time.tv_nsec;

timeout_rsrcB.tv_sec  = current_time.tv_sec + 3;
timeout_rsrcB.tv_nsec = current_time.tv_nsec;



if(thread_id == THREAD_1)
{
  // Attempt to grab rsrcA
  printf("THREAD 1 grabbing resource A @ %d sec and %d nsec\n",
```

```
  (int)current_time.tv_sec,
  (int)current_time.tv_nsec
);
if((rv=pthread_mutex_lock(&rsrcA)) != 0)
{
    printf("Thread 1 ERROR\n");
    pthread_exit(NULL);
}
else
{
    printf("Thread 1 GOT A\n");
    count_rsrcA++;
    printf("count_rsrcA=%d, count_rsrcB=%d\n", count_rsrcA, count_rsrcB);
}

// intentionally sleep in order for thread2 to grab rsrcB, to intentionally
// cause a deadlock condition.
if(!no_wait) usleep(1000000);

// get the current time
clock_gettime(CLOCK_REALTIME, &current_time);
timeout_rsrcB.tv_sec  = current_time.tv_sec + 3;
timeout_rsrcB.tv_nsec = current_time.tv_nsec;

printf("THREAD 1 got A, trying for B @ %d sec and %d nsec\n",
  (int)current_time.tv_sec,
  (int)current_time.tv_nsec
);

//Attempt to grab rsrcB
rv = pthread_mutex_timedlock(&rsrcB, &timeout_rsrcB);
if(rv == 0)
{
    clock_gettime(CLOCK_REALTIME, &current_time);
    printf("Thread 1 GOT B @ %d sec and %d nsec with rv=%d\n",
```

```
        (int)current_time.tv_sec,
        (int)current_time.tv_nsec, rv
      );
      count_rsrcB++;
      printf("count_rsrcA=%d, count_rsrcB=%d\n", count_rsrcA, count_rsrcB);
  }
  else if(rv == ETIMEDOUT)
  {
      printf("Thread 1 TIMEOUT ERROR\n");
      count_rsrcA--;
      pthread_mutex_unlock(&rsrcA);
      pthread_exit(NULL);
  }
  else
  {
      printf("Thread 1 ERROR\n");
      count_rsrcA--;
      pthread_mutex_unlock(&rsrcA);
      pthread_exit(NULL);
  }

  printf("THREAD 1 got A and B\n");
  count_rsrcB--;
  pthread_mutex_unlock(&rsrcB);
  count_rsrcA--;
  pthread_mutex_unlock(&rsrcA);
  printf("THREAD 1 done\n");
}

else // WE ARE THREAD 2
{
  printf("THREAD 2 grabbing resource B @ %d sec and %d nsec\n",
  (int)current_time.tv_sec,
  (int)current_time.tv_nsec
  );
```

```
// Attempt to get rsrcB
if((rv=pthread_mutex_lock(&rsrcB)) != 0)
{
    printf("Thread 2 ERROR\n");
    pthread_exit(NULL);
}
else
{
    printf("Thread 2 GOT B\n");
    count_rsrcB++;
    printf("count_rsrcA=%d, count_rsrcB=%d\n", count_rsrcA, count_rsrcB);
}

// intentionally sleep in order for thread1 to grab rsrcA, to intentionally
// cause a deadlock condition.
if(!no_wait) usleep(1000000);

// get the current clock time
clock_gettime(CLOCK_REALTIME, &current_time);
timeout_rsrcA.tv_sec = current_time.tv_sec + 2;
timeout_rsrcA.tv_nsec = current_time.tv_nsec;

printf("THREAD 2 got B, trying for A @ %d sec and %d nsec\n",
  (int)current_time.tv_sec,
  (int)current_time.tv_nsec
);

// attempt to grab rsrcA using a TIMED LOCK
rv=pthread_mutex_timedlock(&rsrcA, &timeout_rsrcA);
if(rv == 0)
{
    clock_gettime(CLOCK_REALTIME, &current_time);
    printf("Thread 2 GOT A @ %d sec and %d nsec with rv=%d\n",
      (int)current_time.tv_sec,
```

```
          (int)current_time.tv_nsec,
          rv
        );
        count_rsrcA++;
        printf("count_rsrcA=%d, count_rsrcB=%d\n", count_rsrcA, count_rsrcB);
    }
    else if(rv == ETIMEDOUT)
    {
        printf("Thread 2 TIMEOUT ERROR\n");
        count_rsrcB--;
        pthread_mutex_unlock(&rsrcB);
        pthread_exit(NULL);
    }
    else
    {
        printf("Thread 2 ERROR\n");
        count_rsrcB--;
        pthread_mutex_unlock(&rsrcB);
        pthread_exit(NULL);
    }

    printf("THREAD 2 got B and A\n");
    count_rsrcA--;
    pthread_mutex_unlock(&rsrcA);
    count_rsrcB--;
    pthread_mutex_unlock(&rsrcB);
    printf("THREAD 2 done\n");
  }

  // terminate thread(self)
  pthread_exit(NULL);
}
```

Appendix B

```
/**
 * @file posix_mq.c
 * @author Armando Pinales
 * @author Shreyan Prabhu Dhananjayan
 * @brief This code implements the message queue feature in Linux while also
 * using pthreads. This code builds off of Sam Siewert's 'posix_mq.c' which is a
 * ported version of the VxWorks code.
 * @version 0.1
 * @date 2022-02-24
 *
 * @copyright Copyright  Sam Siewert(c) 2022
 *
 */

/****************************************************************************/
/* Function: Basic POSIX message queue demo                                 */
/*                                                                          */
/* Sam Siewert, 02/05/2011                                                  */
/****************************************************************************/

//   Mounting the message queue file system
//       On  Linux,  message queues are created in a virtual file system.
//       (Other implementations may also provide such a feature, but  the
//       details  are likely to differ.)  This file system can be mounted
//       (by the superuser) using the following commands:

//           # mkdir /dev/mqueue
//           # mount -t mqueue none /dev/mqueue


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

```
#include <errno.h>
#include <pthread.h>
#include <string.h>
#include <mqueue.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <pthread.h>
#include <unistd.h>

#define SNDRCV_MQ "/send_receive_mq"
#define MAX_MSG_SIZE 128
#define ERROR (-1)

#define NUM_THREADS (2)
#define MY_SCHEDULER SCHED_FIFO
// #define MY_SCHEDULER SCHED_RR
// #define MY_SCHEDULER SCHED_OTHER

struct mq_attr mq_attr;

// POSIX thread declarations and scheduling attributes
typedef struct
{
    int threadIdx;
} threadParams_t;

pthread_t threads[NUM_THREADS];
threadParams_t threadParams[NUM_THREADS];
pthread_attr_t rt_sched_attr[NUM_THREADS];
int rt_max_prio, rt_min_prio;
struct sched_param rt_param[NUM_THREADS];
struct sched_param main_param;
pthread_attr_t main_attr;
```

```
pid_t mainpid;

void *receiver(void *threadp)
{
  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);
  }

}

static char canned_msg[] = "this is a test, and only a test, in the event of a real emergency,
you would be instructed ...";
```

```
void *sender(void *threadp)

{

  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");

  }


}


void print_scheduler(void)

{

   int schedType, scope;
```

```
   schedType = sched_getscheduler(getpid());

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

   pthread_attr_getscope(&main_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");

}

void main(void)
{
  int rc;
  int i;

  /* setup common message q attributes */
```

```
mq_attr.mq_maxmsg = 10;
mq_attr.mq_msgsize = MAX_MSG_SIZE;
mq_attr.mq_flags = 0;


// Create two communicating processes right here
rt_max_prio = sched_get_priority_max(SCHED_FIFO);
rt_min_prio = sched_get_priority_min(SCHED_FIFO);

print_scheduler();
rc = sched_getparam(mainpid, &main_param);
main_param.sched_priority=rt_max_prio;

if(MY_SCHEDULER != SCHED_OTHER)
{
  if(rc=sched_setscheduler(getpid(), MY_SCHEDULER, &main_param) < 0)
    perror("******** WARNING: sched_setscheduler");
}
print_scheduler();
printf("rt_max_prio=%d\n", rt_max_prio);
printf("rt_min_prio=%d\n", rt_min_prio);

 for(i=0; i < NUM_THREADS; i++)
 {
    //  CPU_ZERO(&threadcpu);

    //  coreid=i%numberOfProcessors;
    //  printf("Setting thread %d to core %d\n", i, coreid);

    //  // assign thread to specific CPU
    //  CPU_SET(coreid, &threadcpu);

     // set new thread attributes
     rc=pthread_attr_init(&rt_sched_attr[i]);
```

```
     rc=pthread_attr_setinheritsched(&rt_sched_attr[i], PTHREAD_EXPLICIT_SCHED);

     rc=pthread_attr_setschedpolicy(&rt_sched_attr[i], MY_SCHEDULER);

    //  rc=pthread_attr_setaffinity_np(&rt_sched_attr[i], sizeof(cpu_set_t), &threadcpu);


     // set schedule priority

     rt_param[i].sched_priority=rt_max_prio-i-1;

     // set parameters

     pthread_attr_setschedparam(&rt_sched_attr[i], &rt_param[i]);


     threadParams[i].threadIdx=i;


    if(i == 0)

    {

      // create threads

      pthread_create(&threads[i],               // pointer to thread descriptor

                     &rt_sched_attr[i],         // use AFFINITY AND SCHEDULER attributes

                     sender,              // thread function entry point

                     (void *)&(threadParams[i]) // parameters to pass in

                     );

    }

    else

    {

      // create threads

      pthread_create(&threads[i],               // pointer to thread descriptor

                     &rt_sched_attr[i],         // use AFFINITY AND SCHEDULER attributes

                     receiver,              // thread function entry point

                     (void *)&(threadParams[i]) // parameters to pass in

                     );

    }



 }
```

```
    for(i=0;i<NUM_THREADS;i++)
        pthread_join(threads[i], NULL);

    printf("\nTEST COMPLETE\n");

}
```

Shreyan Prabhu Dhananjayan & Armando Pinales
02/19/2022
Exercise #3
ECEN 5623 - RTES

Appendix C

```
/**
 * @file posix_mq.c
 * @author Armando Pinales
 * @author Shreyan Prabhu Dhananjayan
 * @brief This code implements the heap message queue feature in Linux while also
 * using pthreads. This code builds off of Sam Siewert's 'heap_mq.c' which is
 * originally VxWorks code.
 * @version 0.1
 * @date 2022-03-01
 *
 * @copyright Copyright  Sam Siewert(c) 2022
 *
 */

/****************************************************************************/
/*                                                                          */
/* Sam Siewert - 10/14/97                                                   */
/*                                                                          */
/*                                                                          */
/****************************************************************************/

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

#define SNDRCV_MQ "/send_receive_mq"
#define NUM_THREADS (2)
#define MY_SCHEDULER SCHED_FIFO

struct mq_attr mq_attr;
```

```
static mqd_t mymq;

// POSIX thread declarations and scheduling attributes
typedef struct
{
    int threadIdx;
} threadParams_t;

pthread_t threads[NUM_THREADS];
threadParams_t threadParams[NUM_THREADS];
pthread_attr_t rt_sched_attr[NUM_THREADS];
int rt_max_prio, rt_min_prio;
struct sched_param rt_param[NUM_THREADS];
struct sched_param main_param;
pthread_attr_t main_attr;
pid_t mainpid;


/* receives pointer to heap, reads it, and deallocate heap memory */

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

  while(1) {

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

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

```
    if((nbytes = mq_receive(mymq, buffer, (size_t)(sizeof(void *)+sizeof(int)), &prio)) == -1)
    {
      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",
(unsigned int)buffptr, prio, nbytes, id);

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

      free(buffptr);

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

    }

  }

}



static char imagebuff[4096];

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


  while(1) {
```

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

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

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

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

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

    sleep(3);

  }

}



static int sid, rid;

void print_scheduler(void)
{
   int schedType, scope;

   schedType = sched_getscheduler(getpid());
```

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

   pthread_attr_getscope(&main_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");


}

void heap_mq(void)
{
  int i, j;
  int rc;
  char pixel = 'A';

  for(i=0;i<4096;i+=64) {
    pixel = 'A';
```

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

printf("buffer =\n%s", imagebuff);

/* setup common message q attributes */
mq_attr.mq_maxmsg = 100;
mq_attr.mq_msgsize = sizeof(void *)+sizeof(int);
mq_attr.mq_flags = 0;

/* note that VxWorks does not deal with permissions? */
mymq = mq_open(SNDRCV_MQ, O_CREAT|O_RDWR, 0, &mq_attr);

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

printf("mymq: %d\r\n", mymq);

// Create two communicating processes right here
rt_max_prio = sched_get_priority_max(SCHED_FIFO);
rt_min_prio = sched_get_priority_min(SCHED_FIFO);

print_scheduler();
rc = sched_getparam(mainpid, &main_param);
main_param.sched_priority=rt_max_prio;

if(MY_SCHEDULER != SCHED_OTHER)
{
  if(rc=sched_setscheduler(getpid(), MY_SCHEDULER, &main_param) < 0)
    perror("******** WARNING: sched_setscheduler");
```

```
}
print_scheduler();
printf("rt_max_prio=%d\n", rt_max_prio);
printf("rt_min_prio=%d\n", rt_min_prio);

 for(i=0; i < NUM_THREADS; i++)
 {
    //  CPU_ZERO(&threadcpu);

    //  coreid=i%numberOfProcessors;
    //  printf("Setting thread %d to core %d\n", i, coreid);

    //  // assign thread to specific CPU
    //  CPU_SET(coreid, &threadcpu);

     // set new thread attributes
     rc=pthread_attr_init(&rt_sched_attr[i]);
     rc=pthread_attr_setinheritsched(&rt_sched_attr[i], PTHREAD_EXPLICIT_SCHED);
     rc=pthread_attr_setschedpolicy(&rt_sched_attr[i], MY_SCHEDULER);
    //  rc=pthread_attr_setaffinity_np(&rt_sched_attr[i], sizeof(cpu_set_t), &threadcpu);

    // set schedule priority
    rt_param[i].sched_priority=rt_max_prio-i-1;
    // set parameters
    pthread_attr_setschedparam(&rt_sched_attr[i], &rt_param[i]);

    threadParams[i].threadIdx=i;

    printf("CRETING THREADs\r\n");
    if(i == 0)
    {
      printf("CRETING RECEIVER THREAD\r\n");
      // create threads
      pthread_create(&threads[i],               // pointer to thread descriptor
                     &rt_sched_attr[i],         // use AFFINITY AND SCHEDULER attributes
```

```
                      receiver,              // thread function entry point
                      NULL // parameters to pass in
                      );
      }
      else
      {
        printf("CRETING SENDER THREAD\r\n");
        // create threads
        pthread_create(&threads[i],               // pointer to thread descriptor
                      &rt_sched_attr[i],          // use AFFINITY AND SCHEDULER attributes
                      sender,              // thread function entry point
                      NULL // parameters to pass in
                      );
      }
   }
   for(i=0;i<NUM_THREADS;i++)
       pthread_join(threads[i], NULL);

  mq_close(mymq);
}


void main()
{
  heap_mq();
}
```

Code for 2nd Question

/*

*@File:threadsafemutex.c

*@author:Shreyan Prabhu Dhananjayan and Armando Pinales

*@brief: To ensure thread safe operation for 2 threads using Mutex

*@date:28th February, 2021

```
Shreyan Prabhu Dhananjayan & Armando Pinales
02/19/2022
Exercise #3
ECEN 5623 - RTES
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <sys/time.h>
#include <syslog.h>
#include <pthread.h>
#include <unistd.h>

#define NUM_THREADS 2
#define MULTIPLIER 1000000000L          /*To convert in to nanseconds*/
struct thread_data
{
    pthread_mutex_t *mutex;
    pthread_t thread_id;
    int thread_priority;
};
typedef struct thread_data thread_data_t;

struct attitude_data                    /*Altitude data*/
{
    double X;
    double Y;
    double Z;
    double acceleration;
```

```
        double roll;
        double pitch;
        double yawRate;
    struct timespec sampleTime;
};

pthread_attr_t rt_sched_attr[NUM_THREADS];
pthread_attr_t main_attr;
pid_t mainpid;
struct sched_param rt_param[NUM_THREADS];
struct sched_param main_param;
int maximum_priority;
thread_data_t threadparams[NUM_THREADS];
typedef struct attitude_data attitude_t;
attitude_t attitude;
pthread_mutex_t mutex1;
struct timespec start;

/**
*@function:update_timespec
*@brief:Staring point of the thread 0
*@argument:Thread parameters structure
*@return: Updated Thread paramters
*/
void *update_timespec(void *thread_param)
{
```

```
        double initialAttitudeValue = 100;

        thread_data_t *thread_param_t = (thread_data_t *) thread_param;

        for(int i=0; i<3 ; i++)

        {

                pthread_mutex_lock(thread_param_t->mutex);    /*Critical Section starts*/

                printf("\n------------Attitude Values getting updated------------\n");

                attitude.X = initialAttitudeValue;

                attitude.Y = initialAttitudeValue;

                attitude.Z = initialAttitudeValue;

                attitude.acceleration = initialAttitudeValue;

                attitude.roll = initialAttitudeValue;

                attitude.pitch = initialAttitudeValue;

                attitude.yawRate = initialAttitudeValue;

                clock_gettime(CLOCK_MONOTONIC, &attitude.sampleTime);

            pthread_mutex_unlock(thread_param_t->mutex); /*Critical Section ends*/

            initialAttitudeValue += 5;

            printf("------------------Attitude Values Updated---------------\n\n");

            usleep(10);

        }


        return (thread_param);

}


/**

*@function:read_timespec

*@brief:Staring point of the thread 1

*@argument:Thread parameters structure
```

```
*@return: Thread paramters
*/
void *read_timespec(void *thread_param)
{
	thread_data_t *thread_param_t = (thread_data_t *) thread_param;
	for(int i=0; i<3 ; i++)
	{
		pthread_mutex_lock(thread_param_t->mutex); /*Critical Section starts*/
		printf("-----------Updated Values is read by another thead--------\n");
		printf("Value of X = %lf\n", attitude.X);
		printf("Value of Y = %lf\n", attitude.Y);
		printf("Value of Z = %lf\n", attitude.Z);
		printf("Value of acceleration = %lf\n", attitude.acceleration);
		printf("Value of Roll = %lf\n", attitude.roll);
		printf("Value of pitch  = %lf\n", attitude.pitch);
		printf("Value of Yaw Rate = %lf\n", attitude.yawRate);
		printf("Execution time since start = %ld nanoseconds\n", (MULTIPLIER*
(attitude.sampleTime.tv_sec - start.tv_sec)) + (attitude.sampleTime.tv_nsec - start.tv_nsec));
		printf("-----------All the updated values are read----------------\n");
	    pthread_mutex_unlock(thread_param_t->mutex); /*Critical Section ends*/
		usleep(10);
	}
	return (thread_param);


}
int main()
{
```

```
/*Setting up scheduling policy and priority for main thread*/
mainpid=getpid();
maximum_priority = sched_get_priority_max(SCHED_FIFO);
int status = sched_getparam(mainpid, &main_param);
main_param.sched_priority = maximum_priority;
status = sched_setscheduler(getpid(), SCHED_FIFO, &main_param);
status=pthread_mutex_init(&mutex1 , NULL);
if( status != 0)
{
        syslog(LOG_ERR, "ERROR: Mutex Initialization");
        exit(-1);
}

for(int i=0; i < NUM_THREADS; i++)
{
        threadparams[i].thread_id = i;
        threadparams[i].mutex = &mutex1;
        /*Setting up scheduling policy and priority for all the threads*/
        status = pthread_attr_init (&rt_sched_attr[i]);
        status = pthread_attr_setinheritsched (&rt_sched_attr[i], PTHREAD_EXPLICIT_SCHED);
        status = pthread_attr_setschedpolicy(&rt_sched_attr[i], SCHED_FIFO);
        rt_param[i].sched_priority=maximum_priority - i;
        pthread_attr_setschedparam(&rt_sched_attr[i],&rt_param[i]);
        threadparams[i].thread_priority=rt_param[i].sched_priority;
}

/*Starting the execution time*/
```

```
	clock_gettime(CLOCK_MONOTONIC, &start);


	/*Creating and invoking threads*/

	pthread_create(&(threadparams[0].thread_id) ,&rt_sched_attr[0], update_timespec, (void *)&threadparams[0]);

	pthread_create(&(threadparams[1].thread_id) ,&rt_sched_attr[1], read_timespec, (void *)&threadparams[1]);


	/*Thread and Mutex cleanup*/

	for(int j=0; j < NUM_THREADS; j++)

	{

		pthread_join(threadparams[j].thread_id, NULL);

	}

	pthread_mutex_destroy(&mutex1);

	return 0;


}
```

Makefile for 2 question

```
CC= gcc

CFLAGS=-g -Wall -Werror -pthread

all:

	$(CC) $(CFLAGS) -o threadsafemutex threadsafemutex.c


clean:

	rm -f *.o
```

Shreyan Prabhu Dhananjayan & Armando Pinales
02/19/2022
Exercise #3
ECEN 5623 - RTES

Code for 5th question

```
/*
*@File:threadsafemutex.c
*@author:Shreyan Prabhu Dhananjayan and Armando Pinales
*@brief:
*@date:28th February, 2021
*@Reference:https://stackoverflow.com/questions/46365448/pthread-mutex-timedlock-exiting-prematurely-without-waiting-for-timeout
*/


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include <syslog.h>
#include <pthread.h>
#include <time.h>
#include <unistd.h>
#include <errno.h>


#define NUM_THREADS 3
#define MULTIPLIER 1000000000L          /*To convert in to nanseconds*/
struct thread_data
{
        pthread_mutex_t *mutex;
        pthread_t thread_id;
```

```
    int thread_priority;
};
typedef struct thread_data thread_data_t;

struct attitude_data                    /*Altitude data*/
{
    double X;
    double Y;
    double Z;
    double acceleration;
    double roll;
    double pitch;
    double yawRate;
  struct timespec sampleTime;
};

pthread_attr_t rt_sched_attr[NUM_THREADS];
pthread_attr_t main_attr;
pid_t mainpid;
struct sched_param rt_param[NUM_THREADS];
struct sched_param main_param;
int maximum_priority;
thread_data_t threadparams[NUM_THREADS];
typedef struct attitude_data attitude_t;
attitude_t attitude;
pthread_mutex_t mutex1;
struct timespec start,end;
```

```
/**
*@function:update_timespec
*@brief:Staring point of the thread 0
*@argument:Thread parameters structure
*@return: Updated Thread paramters
*/


void *wait_mutex(void *thread_param)
{
	struct timespec wait;
	int status;
	thread_data_t *thread_param_t = (thread_data_t *) thread_param;
	clock_gettime(CLOCK_REALTIME, &start);
	while(1)
	{
		clock_gettime(CLOCK_REALTIME, &wait);
		wait.tv_sec+=10;
		status = pthread_mutex_timedlock(thread_param_t->mutex,&wait);
		if(status == ETIMEDOUT)
		{
			clock_gettime(CLOCK_REALTIME, &end);
			printf("NO NEW DATA AVAILABLE AT: %ldSeconds\n", end.tv_sec -
start.tv_sec);
		}
		else
```

```
        {
            printf("Lock is obtained\n");
        }

    }

}
void *update_timespec(void *thread_param)
{

    double initialAttitudeValue = 100;
    thread_data_t *thread_param_t = (thread_data_t *) thread_param;
    for(int i=0; i<3 ; i++)
    {
        pthread_mutex_lock(thread_param_t->mutex);    /*Critical Section starts*/
        printf("\n------------Attitude Values getting updated------------\n");
        attitude.X = initialAttitudeValue;
        attitude.Y = initialAttitudeValue;
        attitude.Z = initialAttitudeValue;
        attitude.acceleration = initialAttitudeValue;
        attitude.roll = initialAttitudeValue;
        attitude.pitch = initialAttitudeValue;
        attitude.yawRate = initialAttitudeValue;
        clock_gettime(CLOCK_MONOTONIC, &attitude.sampleTime);
       pthread_mutex_unlock(thread_param_t->mutex); /*Critical Section ends*/
       initialAttitudeValue += 5;
       printf("------------------Attitude Values Updated---------------\n\n");
```

```
        usleep(10);
    }

    return (thread_param);
}

/**
*@function:read_timespec
*@brief:Staring point of the thread 1
*@argument:Thread parameters structure
*@return: Thread paramters
*/
void *read_timespec(void *thread_param)
{
    thread_data_t *thread_param_t = (thread_data_t *) thread_param;
    for(int i=0; i<3 ; i++)
    {
        pthread_mutex_lock(thread_param_t->mutex); /*Critical Section starts*/
        printf("-----------Updated Values is read by another thead--------\n");
        printf("Value of X = %lf\n", attitude.X);
        printf("Value of Y = %lf\n", attitude.Y);
        printf("Value of Z = %lf\n", attitude.Z);
        printf("Value of acceleration = %lf\n", attitude.acceleration);
        printf("Value of Roll = %lf\n", attitude.roll);
        printf("Value of pitch  = %lf\n", attitude.pitch);
        printf("Value of Yaw Rate = %lf\n", attitude.yawRate);
```

```
            printf("Execution time since start = %ld nanoseconds\n", (MULTIPLIER*
(attitude.sampleTime.tv_sec - start.tv_sec)) + (attitude.sampleTime.tv_nsec - start.tv_nsec));

            printf("-----------All the updated values are read----------------\n");

          pthread_mutex_unlock(thread_param_t->mutex); /*Critical Section ends*/

            usleep(10);

      }

      return (thread_param);


}

int main()

{

      /*Setting up scheduling policy and priority for main thread*/

      mainpid=getpid();

      maximum_priority = sched_get_priority_max(SCHED_FIFO);

      int status = sched_getparam(mainpid, &main_param);

      main_param.sched_priority = maximum_priority;

      status = sched_setscheduler(getpid(), SCHED_FIFO, &main_param);

      status=pthread_mutex_init(&mutex1 , NULL);

      if( status != 0)

      {

            syslog(LOG_ERR, "ERROR: Mutex Initialization");

            exit(-1);

      }


      for(int i=0; i < NUM_THREADS; i++)

      {

            threadparams[i].thread_id = i;
```

```
            threadparams[i].mutex = &mutex1;

            /*Setting up scheduling policy and priority for all the threads*/

            status = pthread_attr_init (&rt_sched_attr[i]);

            status = pthread_attr_setinheritsched (&rt_sched_attr[i],
PTHREAD_EXPLICIT_SCHED);

            status = pthread_attr_setschedpolicy(&rt_sched_attr[i], SCHED_FIFO);

            rt_param[i].sched_priority=maximum_priority - i;

            pthread_attr_setschedparam(&rt_sched_attr[i],&rt_param[i]);

            threadparams[i].thread_priority=rt_param[i].sched_priority;

      }


      /*Starting the execution time*/

      clock_gettime(CLOCK_MONOTONIC, &start);


      /*Creating and invoking threads*/

      pthread_create(&(threadparams[0].thread_id) ,&rt_sched_attr[0], update_timespec, (void
*)&threadparams[0]);

      pthread_create(&(threadparams[1].thread_id) ,&rt_sched_attr[1], read_timespec, (void
*)&threadparams[1]);

      /*Thread and Mutex cleanup*/

      for(int j=0; j < 2; j++)

      {

            pthread_join(threadparams[j].thread_id, NULL);

      }

      pthread_mutex_lock(&mutex1);

      pthread_create(&(threadparams[2].thread_id), &rt_sched_attr[2], wait_mutex, (void
*)&threadparams[2]);

      pthread_join(threadparams[2].thread_id, NULL);
```

```
    pthread_mutex_destroy(&mutex1);
    return 0;

}
```

Makefile

```
CC= gcc
CFLAGS=-g -Wall -Werror -pthread
all:
	$(CC) $(CFLAGS) -o threadsafemutex2 threadsafemutex2.c

clean:
	rm -f *.o
```