

Exercise #3 – Threading/Tasking and Real-Time Synchronization

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

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

The paper, priority inheritance protocols: an approach to real-time synchronization by Lui Sha, Ragunathan Rajkumar and John Lehoczky describes two priority inheritance protocols- basic priority inheritance protocol and priority ceiling protocol. The usage of synchronization primitives like semaphores and mutexes leads to uncontrolled priority inversion (a higher priority job is blocked by lower priority jobs for an indefinite period of time). The paper proves that both resolve the issue but in particular, priority ceiling protocol reduces the worst-case task blocking time and prevents deadlocks.

The three key points made in the paper are as follows:

1. Priority inversion can occur in two cases:
 - When a resource is shared by both a higher priority job and a lower priority task: if the lower priority job gains the shared resource first (eg. Semaphore) and then the higher priority task requests access to the shared resource, the higher priority job is blocked until the former completes its work in the critical section.
 - In the case of 3 different jobs (j_1, j_2, j_3) where J_1 and J_3 have a shared resource and suppose the lowest priority job, J_3 starts executing first. When in its critical section, J_1 is initiated and should ideally pre-empt the execution of J_3 . However, it is blocked on the binary semaphore held by J_3 and thus when at a later time J_2 is initiated, J_3 is pre-empted by J_2 . Thus, the blocking of J_3 and hence that of J_1 will continue until J_2 and other pending intermediate jobs are completed.

Blocking leads to missing of deadlines even at low level of resource utilisation and effects the schedulability of the system.

2. **Basic priority protocol:** When a job J blocks one or more higher priority jobs, it ignores its original priority and executes its critical section only at the highest priority level of all the jobs it blocks. After the critical section, job J returns to its original priority. The priority inheritance is transitive: if job J_3 blocks J_2 , and job J_2 blocks job J_1 , J_3 would inherit the priority of J_1 via J_2 .
The service sets would face two types of blocking here: direct blocking- a higher priority job attempts to lock a locked semaphore, and push-through blocking- a medium priority job J_1 can be blocked by a lower priority job J_2 , which inherits the priority of a high priority job J_0 .
However, this protocol still faces some issues. Mainly, it does not prevent deadlocks and even though the duration of blocking a job is bounded, it can be substantial enough because of a chain of blocking.
3. **Priority ceiling protocol:** When a job J pre-empts 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 pre-empted critical sections. If this condition cannot be satisfied, job J is denied entry into the critical section and suspended, and the job that blocks J inherits J 's priority.
This is realised by assigning a priority ceiling to each semaphore, which is equal to the highest priority task that may use this semaphore. A job can be blocked for at most the duration of one longest subcritical section.
4. A set of n periodic tasks using the priority ceiling protocol can be scheduled by the RM algorithm if

$$\forall i, 1 \leq i \leq n, \quad \frac{C_1}{T_1} + \frac{C_2}{T_2} + \dots + \frac{C_i}{T_i} + \frac{B_i}{T_i} \leq i(2^{1/i} - 1).$$

where B_i is the worst case blocking time duration.

b) Read the historical positions of Linux Torvalds as described by Jonathan Corbet and Ingo Molnar and Thomas Gleixner on this topic as well. Note that more recently Red Hat has added support for priority ceiling emulation and priority inheritance and has a great overview on use of these POSIX real-time extension features here – general real-time overview. The key systems calls are `pthread_mutexattr_setprotocol`, `pthread_mutexattr_getprotocol` as well as `pthread_mutex_setpriorityceiling` and `pthread_mutex_getpriorityceiling`. Why did some of the Linux community resist addition of priority ceiling and/or priority inheritance until more recently (Linux has been distributed since the early 1990's and the problem and solutions were known before this).

The Linux kernel implements a simple form of priority inheritance by not allowing kernel code to be pre-empted while holding a spinlock. In some other systems, a process will inherit priority of another process in case it needs to hold a needed lock. However, these schemes tend to complicate and slow down the code and are implemented in poor application designs. This is mainly why Linus was resistant of priority inheritance protocols. Imagining backlash from Linus and other community members, developers would also be hesitant to implement their code to resolve priority inheritance. The overhead of scheduling and synchronisation mechanisms would have also limited the overall system performance, especially when in the past there were limited computing resources. Further, it could have the general consensus that Linux was designed to a general-purpose OS and thus real-time features would not have been a priority. Linux also uses a CFS (completely fair scheduler) before the application of NPTL and POSIX SCHED_FIFO. It can be observed that priority inversion is not an issue with this scheduling policy as it uses time slices to ensure that all services make continual progress through periodic timer-based pre-emption.

However, Ingo Molnar designed a priority inheriting futex implementation which provided a useful functionality to the user space and that would incur less overhead on the kernel side. It aims to achieve determinism and well-bound latencies for user-space applications.

I believe that maybe historically this decision made sense when there was less demand for real-time applications. However, with an increase in RT applications, synchronisation mechanics are more common and there is a need for deterministic locking implementation. As Ingo mentions, most of the hesitancy in applying priority inheritance comes for kernel-space locks. In user-space, priority inheritance is required.

c) Given that there are two solutions to unbounded priority inversion, priority ceiling emulation and priority inheritance, based upon your reading on support for both by POSIX real-time extensions, which do you think is best to use and why?

Currently the Linux kernel does not natively support priority ceiling protocol (PCP), which is essentially for ensuring deadlock free execution and preventing chain of blocking. However, the issue of unbounded priority can still be resolved with priority inheritance protocol. This is shown in Lemma 4 in the paper 'Priority inheritance protocols': 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 priority equal to or higher than that of J.

Using function `int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int protocol)`, the protocol of the mutex attribute can be set to `PTHREAD_PRIO_INHERIT`. When a thread is blocking a higher priority thread because of owning mutexes with the `PTHREAD_PRIO_INHERIT` protocol attribute, it executes at the higher of its priority or priority of the higher priority thread waiting on any of the mutexes owned by this thread and initialised with this protocol.

However, a common issue faced with this solution is chain blocking: When a high priority task blocks, it is possible that shortly after it loans its priority to a lower priority task, another higher priority service ($H+n$) will block on the same semaphore, therefore requiring a new inheritance of ($H+n$) by the lower priority task so that it is not blocked by services of priority ($H+1$) to ($H+n-1$) interfering with the current service, which has the inherited priority H . The chaining is complex, so it would be easier to simply give the lower priority a priority that is so high that chaining is not necessary.

This problem gave rise to priority ceiling emulation protocol. Since PCP cannot actually be implemented in an OS, the priority ceiling emulation protocol is used to set the priority of the task that is currently executing to the highest priority of all those services that can potentially request access to the critical section based on programmer understanding. When inversion occurs, the task is temporarily loaned the pre-configured ceiling priority, ensuring that

it completes the critical section with one priority amplification. However, over-amplification caused by this solution could lead to significant interference to high-frequency, high-priority services.

Thus, priority ceiling emulation is the better protocol. But it is very crucial to maintain the priorities of the users of the critical section and make sure that there isn't any significant interference being caused to other high priority services.

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.

Thread safety: Thread safety refers to the property of a function or code segment that can be safely executed by multiple threads concurrently without causing unexpected behaviour or data corruption.

Re-entrant Functions: A re-entrant function is designed to be called simultaneously by multiple threads without interfering with each other's execution. In other words, re-entrant functions are thread-safe functions that can handle concurrent invocations from different threads. If there is more than one thread executing or awaiting execution in the ready state in the dispatch queue, the threads are considered to be concurrently active.

Characteristics of re-entrant functions include:

- No shared data: Re-entrant functions avoid using shared data or, if they do, they ensure that access to shared data is properly synchronized.
- Local storage: Re-entrant functions rely heavily on local variables and stack allocation for temporary storage. Each invocation of the function gets its own copy of local variables, ensuring that one thread's execution does not interfere with another's.
- No dependency on global or static data: Re-entrant functions do not rely on global or static variables that may be modified by other threads.

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

- pure functions that use only stack and have no global memory: local variables are dynamically allocated on the stack and thus, any function that does not use static data or other shared resources is thread safe. However, the stack memory is limited in size and it must be managed for each thread to prevent stack overflow or memory exhaustion. In real-time threads/tasks, one drawback is that it is not possible to share data with other threads as well. With the use of local memory, there is reduced contention and improved determinism in real-time systems. It can be implemented as shown in the figure below.

```
1  /* threadsafe function */
2  int diff(int x, int y)
3  {
4      int delta;
5
6      delta = y - x;
7      if (delta < 0)
8          delta = -delta;
9
10     return delta;
11 }
```

- functions which use thread indexed global data: Functions that use thread-indexed global data refer to functions that utilize global variables but ensure thread safety by associating different instances of the global data with each thread. Thread-local storage (TLS) is a memory management method that uses static or global memory local to a thread. This approach allows each thread to have its own separate copy of the global data, avoiding conflicts and ensuring thread safety.

Many systems impose restrictions on the size of the thread-local memory block. On the other hand, if a system can provide a pointer, then this allows the use of arbitrarily sized memory blocks in a thread-local manner, by allocating such a memory block dynamically and storing the memory address of that block in the thread-local variable.

In real-time threads/tasks, by allotting each thread its own copy of global data can increase memory consumption and this places a constraint on system memory. If the threads need to coordinate or share data across the threads, synchronization mechanisms are more difficult to design. Further, cleanup can be challenging to avoid memory leaks or data inconsistencies. It can be implemented as shown in the figure below.

```
pthread_key_t threadKey;

void* thread_function(void *arg)
{
    int *p = malloc(sizeof(int));
    *p = 1;
    pthread_setspecific(threadKey, p);
    int* thread_value = pthread_getspecific(threadKey);
    printf("thread id: %d, global value before: %d\n", (unsigned int) pthread_self(), *thread_value);
    *thread_value += 1;
    printf("thread id: %d, global value before: %d\n", (unsigned int) pthread_self(), *thread_value);
    free(p);
    return 0;
}
```

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

Functions that use static data or any other shared resource must serialize the access to these resources with locks in order to be threadsafe. The synchronisation mechanism varies according to application but the most common methods are mutexes, semaphores and locks. The lock is placed around the critical section which uses the shared resource.

With these mechanism in-place, there is chance of unbounded priority inversion or deadlocks. With the occurrence of these issues, deadlines could be missed or there would be lower schedulability for a real-time system. Thus, it is important to use priority inheritance protocols or implement proper locking strategy. It can be implemented as shown in the figure below.

```
int gsum=0;

void *incThread(void *threadp)
{
    sem_wait(threadParams->semaphore);
    gsum++;
    printf("gsum=%d\n", gsum);
    sem_post(threadParams->semaphore);
}
```

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.

The program threads1.c is designed to run an update thread at 1Hz and a read thread at 0.1Hz rate. The update thread updates a structure that includes a double precision position, attitude and a timespec structure while the read thread prints out the values. Using mutexes, the structure is protected from data corruption. The program sets the affinity of the threads so that AMP environment can be emulated while the scheduler function and semaphores make sure of the frequency of the threads. The output of the program is shown below, run for 180 seconds. It can be seen that the 18 values read match the updated values.

Output of the program execution:

```
sudo ./threads1
```

```
*****Update thread*****Time now: 388272.716915
Latitude: 1911759956.000000, Longitude: 749241873.000000, Altitude: 137806862.000000
Roll: 42999170.000000, Pitch: 982906996.000000, Yaw: 135497281.000000
*****Read thread*****Time now: 388272.825776
Latitude: 1911759956.000000, Longitude: 749241873.000000, Altitude: 137806862.000000
Roll: 42999170.000000, Pitch: 982906996.000000, Yaw: 135497281.000000
*****Update thread*****Time now: 398554.187723
Latitude: 402724286.000000, Longitude: 593209441.000000, Altitude: 1194953865.000000
Roll: 894429689.000000, Pitch: 364228444.000000, Yaw: 1947346619.000000
*****Read thread*****Time now: 398554.291042
Latitude: 402724286.000000, Longitude: 593209441.000000, Altitude: 1194953865.000000
Roll: 894429689.000000, Pitch: 364228444.000000, Yaw: 1947346619.000000
*****Update thread*****Time now: 408550.696381
Latitude: 1275373743.000000, Longitude: 387346491.000000, Altitude: 350322227.000000
Roll: 841148365.000000, Pitch: 1960709859.000000, Yaw: 1760281936.000000
*****Read thread*****Time now: 408550.774566
Latitude: 1275373743.000000, Longitude: 387346491.000000, Altitude: 350322227.000000
Roll: 841148365.000000, Pitch: 1960709859.000000, Yaw: 1760281936.000000
*****Update thread*****Time now: 418650.749179
Latitude: 933110197.000000, Longitude: 6939507.000000, Altitude: 740759355.000000
Roll: 1285228804.000000, Pitch: 1789376348.000000, Yaw: 502278611.000000
*****Read thread*****Time now: 418650.834707
Latitude: 933110197.000000, Longitude: 6939507.000000, Altitude: 740759355.000000
Roll: 1285228804.000000, Pitch: 1789376348.000000, Yaw: 502278611.000000
*****Update thread*****Time now: 428656.389050
Latitude: 2025187190.000000, Longitude: 1967681095.000000, Altitude: 1850952926.000000
Roll: 437116466.000000, Pitch: 1704365084.000000, Yaw: 1176911340.000000
*****Read thread*****Time now: 428656.468998
Latitude: 2025187190.000000, Longitude: 1967681095.000000, Altitude: 1850952926.000000
Roll: 437116466.000000, Pitch: 1704365084.000000, Yaw: 1176911340.000000
*****Update thread*****Time now: 438667.954565
Latitude: 1046741222.000000, Longitude: 337739299.000000, Altitude: 1896306640.000000
Roll: 1343606042.000000, Pitch: 1111783898.000000, Yaw: 446340713.000000
*****Read thread*****Time now: 438668.032473
Latitude: 1046741222.000000, Longitude: 337739299.000000, Altitude: 1896306640.000000
Roll: 1343606042.000000, Pitch: 1111783898.000000, Yaw: 446340713.000000
*****Update thread*****Time now: 448752.795405
Latitude: 501772890.000000, Longitude: 1781999754.000000, Altitude: 150517567.000000
Roll: 212251746.000000, Pitch: 1983690368.000000, Yaw: 364319529.000000
*****Read thread*****Time now: 448752.855745
Latitude: 501772890.000000, Longitude: 1781999754.000000, Altitude: 150517567.000000
Roll: 212251746.000000, Pitch: 1983690368.000000, Yaw: 364319529.000000
*****Update thread*****Time now: 458766.342685
Latitude: 1722060049.000000, Longitude: 1455590964.000000, Altitude: 328298285.000000
Roll: 70636429.000000, Pitch: 136495343.000000, Yaw: 1472576335.000000
*****Read thread*****Time now: 458766.428187
Latitude: 1722060049.000000, Longitude: 1455590964.000000, Altitude: 328298285.000000
Roll: 70636429.000000, Pitch: 136495343.000000, Yaw: 1472576335.000000
*****Update thread*****Time now: 468890.130985
Latitude: 1708302647.000000, Longitude: 1857756970.000000, Altitude: 1874799051.000000
Roll: 1426819080.000000, Pitch: 885799631.000000, Yaw: 1314218593.000000
```

```

*****Read thread*****Time now: 468890.214602
Latitude: 1708302647.000000, Longitude: 1857756970.000000, Altitude: 1874799051.000000
Roll: 1426819080.000000, Pitch: 885799631.000000, Yaw: 1314218593.000000
*****Update thread*****Time now: 478953.368329
Latitude: 394633074.000000, Longitude: 983631233.000000, Altitude: 1675518157.000000
Roll: 1645933681.000000, Pitch: 1943003493.000000, Yaw: 553160358.000000
*****Read thread*****Time now: 478953.416191
Latitude: 394633074.000000, Longitude: 983631233.000000, Altitude: 1675518157.000000
Roll: 1645933681.000000, Pitch: 1943003493.000000, Yaw: 553160358.000000
*****Update thread*****Time now: 488955.645327
Latitude: 395279207.000000, Longitude: 606199759.000000, Altitude: 358984857.000000
Roll: 435889744.000000, Pitch: 1344593499.000000, Yaw: 378469911.000000
*****Read thread*****Time now: 488955.733116
Latitude: 395279207.000000, Longitude: 606199759.000000, Altitude: 358984857.000000
Roll: 435889744.000000, Pitch: 1344593499.000000, Yaw: 378469911.000000
*****Update thread*****Time now: 499053.152852
Latitude: 1910300925.000000, Longitude: 2030449291.000000, Altitude: 120306710.000000
Roll: 1986894018.000000, Pitch: 1007277217.000000, Yaw: 551836836.000000
*****Read thread*****Time now: 499053.240919
Latitude: 1910300925.000000, Longitude: 2030449291.000000, Altitude: 120306710.000000
Roll: 1986894018.000000, Pitch: 1007277217.000000, Yaw: 551836836.000000
*****Update thread*****Time now: 509062.458310
Latitude: 628966950.000000, Longitude: 1520982030.000000, Altitude: 1761250573.000000
Roll: 1089653714.000000, Pitch: 1003886059.000000, Yaw: 168057522.000000
*****Read thread*****Time now: 509062.600372
Latitude: 628966950.000000, Longitude: 1520982030.000000, Altitude: 1761250573.000000
Roll: 1089653714.000000, Pitch: 1003886059.000000, Yaw: 168057522.000000
*****Update thread*****Time now: 519061.656975
Latitude: 670752506.000000, Longitude: 2025554010.000000, Altitude: 1649709016.000000
Roll: 262178224.000000, Pitch: 82173109.000000, Yaw: 1105816539.000000
*****Read thread*****Time now: 519061.737819
Latitude: 670752506.000000, Longitude: 2025554010.000000, Altitude: 1649709016.000000
Roll: 262178224.000000, Pitch: 82173109.000000, Yaw: 1105816539.000000
*****Update thread*****Time now: 529052.714647
Latitude: 420687483.000000, Longitude: 1669475776.000000, Altitude: 1813080154.000000
Roll: 1579068977.000000, Pitch: 395191309.000000, Yaw: 1431742587.000000
*****Read thread*****Time now: 529052.808080
Latitude: 420687483.000000, Longitude: 1669475776.000000, Altitude: 1813080154.000000
Roll: 1579068977.000000, Pitch: 395191309.000000, Yaw: 1431742587.000000
*****Update thread*****Time now: 539054.675189
Latitude: 1892430639.000000, Longitude: 1449228398.000000, Altitude: 989241888.000000
Roll: 1012836610.000000, Pitch: 627992393.000000, Yaw: 481928577.000000
*****Read thread*****Time now: 539054.756358
Latitude: 1892430639.000000, Longitude: 1449228398.000000, Altitude: 989241888.000000
Roll: 1012836610.000000, Pitch: 627992393.000000, Yaw: 481928577.000000
*****Update thread*****Time now: 549052.822583
Latitude: 1104561852.000000, Longitude: 915711850.000000, Altitude: 737703662.000000
Roll: 108375482.000000, Pitch: 202550399.000000, Yaw: 1738076217.000000
*****Read thread*****Time now: 549052.899559
Latitude: 1104561852.000000, Longitude: 915711850.000000, Altitude: 737703662.000000
Roll: 108375482.000000, Pitch: 202550399.000000, Yaw: 1738076217.000000

```

TEST COMPLETE

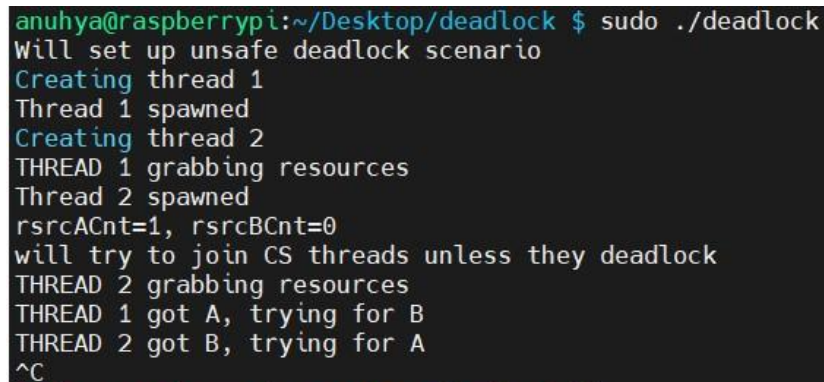
3) [20 points] Download example-sync-updated-2/ and review, build and run it.

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

Deadlocks occur in a multi-thread condition where 2 or more threads of execution are waiting on resources held by another and there is a circular wait- e.g., if A is waiting on resource R1 to produce resource R2; and B is waiting on resource R2 to produce resource R1 – this is a deadlock. Basically, the threads are stuck and unable to make any progress and are waiting indefinitely for resources that will never become available.

In the program ‘deadlock’, two threads are started simultaneously. Thread 1 holds the resource A while thread 2 acquires resource B. However, to complete their respective work in their critical sections, thread 2 needs to acquire resource B while thread 2 needs to acquire resource A. Thus, both the threads are waiting on the resources the other one holds currently. The program encounters a deadlock. The program output is shown in Figure 1.

Deadlocks can be prevented by proper resource allocation and thread serialisation.



```
anuhya@raspberrypi:~/Desktop/deadlock $ sudo ./deadlock
Will set up unsafe deadlock scenario
Creating thread 1
Thread 1 spawned
Creating thread 2
THREAD 1 grabbing resources
Thread 2 spawned
rsrcACnt=1, rsrcBCnt=0
will try to join CS threads unless they deadlock
THREAD 2 grabbing resources
THREAD 1 got A, trying for B
THREAD 2 got B, trying for A
^C
```

Figure 1

Unbounded priority inversion: In a simple priority inversion case, the high priority task is blocked by a lower priority task as it waits for the shared resource that the latter thread is using in its critical section. Here, the former experiences priority inversion for the duration of the critical section of the lower priority task. However, if the low priority thread suffers interference from a medium priority thread, the high priority thread could be blocked for an indeterminate amount time. This is called unbounded priority inversion.

In the program ‘pthread3’, the lower priority task is created first and it executes a critical section where it uses a shared resource that a higher priority task also requires when it commences. So, it waits for the lowest priority thread to release the shared resource. When the interference argument is passed, a medium priority task is created which does not have any critical section or shared resource. Here, it pre-empt the lowest priority task currently executing and now the highest priority task has to wait for an unbounded period of time, till the intermediate task completes its execution. The program output shows that when the intermediate thread completes its task, the lowest priority task which still holds the shared resource completes first and finally, the highest priority task is able to execute its critical section. The program output is shown in Figure 2.

In the program ‘pthread3ok’, there is no lock on the shared resource so the unbounded priority inversion is remediated but the problem of sequence corruption persists. The program output is shown in Figure 3.


```
anuhya@raspberrypi:~/Desktop/example-sync-updated-2 $ sudo ./pthread3_00
Fibonacci Cycle Burner test ...
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578
5702887 9227465 14930352 24157817 39088169 63245986 102334155 165580141 267914296 433494437 701408733 1134903170 1836311903 2971215073 512559680

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578
5702887 9227465 14930352 24157817 39088169 63245986 102334155 165580141 267914296 433494437 701408733 1134903170 1836311903 2971215073 512559680
done
interference time = 300 secs
unsafe mutex will be created
Pthread Policy is SCHED_OTHER
Setting thread 0 to core 0

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

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

Creating Low Prio RT or BE thread 3
Low prio 3 thread SPANNED at 0.001358 sec
.....
.....
.....
.....
.....
.....
.....
CS-L REQUEST

CS-L ENTRY 1
CS-L1 CScnt=1

Creating RT thread 1, CScnt=1

CS-H REQUEST
High prio 1 thread SPANNED at 0.199980 sec

Creating RT thread 2
M1 M2 M3 M4 M5 M6 M7 M8 M9 M10 M11 M12 M13 M14 M15 M16 M17 M18 M19 M20 M21 M22 M23 M24 M25 M26 M27 M28 M29 M30 M31 M32 M33 M34 M35 M36 M37 M38 M39 M40 M41 M42
M43 M44 M45 M46 M47 M48 M49 M50 M51 M52 M53 M54 M55 M56 M57 M58 M59 M60 M61 M62 M63 M64 M65 M66 M67 M68 M69 M70 M71 M72 M73 M74 M75 M76 M77 M78 M79 M80 M81 M
82 M83 M84 M85 M86 M87 M88 M89 M90 M91 M92 M93 M94 M95 M96 M97 M98 M99 M100 M101 M102 M103 M104 M105 M106 M107 M108 M109 M110 M111 M112 M113 M114 M115 M116 M1
17 M118 M119 M120 M121 M122 M123 M124 M125 M126 M127 M128 M129 M130 M131 M132 M133 M134 M135 M136 M137 M138 M139 M140 M141 M142 M143 M144 M145 M146 M147 M148
M149 M150 M151 M152 M153 M154 M155 M156 M157 M158 M159 M160 M161 M162 M163 M164 M165 M166 M167 M168 M169 M170 M171 M172 M173 M174 M175 M176 M177 M178 M179 M18
0 M181 M182 M183 M184 M185 M186 M187 M188 M189 M190 M191 M192 M193 M194 M195 M196 M197 M198 M199 M200 M201 M202 M203 M204 M205 M206 M207 M208 M209 M210 M211 M
212 M213 M214 M215 M216 M217 M218 M219 M220 M221 M222 M223 M224 M225 M226 M227 M228 M229 M230 M231 M232 M233 M234 M235 M236 M237 M238 M239 M240 M241 M242 M243
M244 M245 M246 M247 M248 M249 M250 M251 M252 M253 M254 M255 M256 M257 M258 M259 M260 M261 M262 M263 M264 M265 M266 M267 M268 M269 M270 M271 M272 M273 M274 M2
75 M276 M277 M278 M279 M280 M281 M282 M283 M284 M285 M286 M287 M288 M289 M290 M291 M292 M293 M294 M295 M296 M297 M298 M299 M300
**** MID PRIO 2 on core 0 INTERFERE NO SEM COMPLETED at 10.707083 sec
Middle prio 2 thread SPANNED at 10.707769 sec
CS-L2 CS-L3 CS-L4 CS-L5 CS-L6 CS-L7 CS-L8 CS-L9 CS-L10
CS-L LEAVING

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

CS-H EXIT

**** HIGH PRIO 1 on core 0 CRIT SECTION WORK COMPLETED at 11.320240 sec

CS-L EXIT

**** LOW PRIO 3 on core 0 CRIT SECTION WORK COMPLETED at 11.320368 sec
HIGH PRIO joined
MID PRIO joined
LOW PRIO joined
START SERVICE joined
All threads done
```

Figure 2: Output of 'pthread3' run for 300 times of the middle priority task.


```
anuhya@raspberrypi:~/Desktop/example-sync-updated-2 $ sudo ./pthread3ok 300
Fibonacci Cycle Burner test ...
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578
5702887 9227465 14930352 24157817 39088169 63245986 102334155 165580141 267914296 433494437 701408733 1134903170 1836311903 2971215073 512559680

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578
5702887 9227465 14930352 24157817 39088169 63245986 102334155 165580141 267914296 433494437 701408733 1134903170 1836311903 2971215073 512559680
done
interference time = 300 secs
unsafe mutex will be created
Pthread Policy is SCHED_OTHER
Setting thread 0 to core 0

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

Creating RT thread 0
Start services thread spawned

Creating BE thread 3
will join service threads
Low prio 3 thread SPAWNED at 0.000615 sec
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
CS-L REQUEST

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

CS-H REQUEST

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

CS-H EXIT

**** HIGH PRIO 1 on core 0 UNPROTECTED CRIT SECTION WORK COMPLETED at 0.537538 sec
High prio 1 thread SPAWNED at 0.538113 sec

Creating RT thread 2
M1 M2 M3 M4 M5 M6 M7 M8 M9 M10 M11 M12 M13 M14 M15 M16 M17 M18 M19 M20 M21 M22 M23 M24 M25 M26 M27 M28 M29 M30 M31 M32 M33 M34 M35 M36 M37 M38 M39 M40 M41 M42
M43 M44 M45 M46 M47 M48 M49 M50 M51 M52 M53 M54 M55 M56 M57 M58 M59 M60 M61 M62 M63 M64 M65 M66 M67 M68 M69 M70 M71 M72 M73 M74 M75 M76 M77 M78 M79 M80 M81 M
82 M83 M84 M85 M86 M87 M88 M89 M90 M91 M92 M93 M94 M95 M96 M97 M98 M99 M100 M101 M102 M103 M104 M105 M106 M107 M108 M109 M110 M111 M112 M113 M114 M115 M116 M1
17 M118 M119 M120 M121 M122 M123 M124 M125 M126 M127 M128 M129 M130 M131 M132 M133 M134 M135 M136 M137 M138 M139 M140 M141 M142 M143 M144 M145 M146 M147 M148
M149 M150 M151 M152 M153 M154 M155 M156 M157 M158 M159 M160 M161 M162 M163 M164 M165 M166 M167 M168 M169 M170 M171 M172 M173 M174 M175 M176 M177 M178 M179 M18
0 M181 M182 M183 M184 M185 M186 M187 M188 M189 M190 M191 M192 M193 M194 M195 M196 M197 M198 M199 M200 M201 M202 M203 M204 M205 M206 M207 M208 M209 M210 M211 M
212 M213 M214 M215 M216 M217 M218 M219 M220 M221 M222 M223 M224 M225 M226 M227 M228 M229 M230 M231 M232 M233 M234 M235 M236 M237 M238 M239 M240 M241 M242 M243
M244 M245 M246 M247 M248 M249 M250 M251 M252 M253 M254 M255 M256 M257 M258 M259 M260 M261 M262 M263 M264 M265 M266 M267 M268 M269 M270 M271 M272 M273 M274 M2
75 M276 M277 M278 M279 M280 M281 M282 M283 M284 M285 M286 M287 M288 M289 M290 M291 M292 M293 M294 M295 M296 M297 M298 M299 M300
**** MID PRIO 2 on core 0 INTERFERE NO SEM COMPLETED at 11.299227 sec
Middle prio 2 thread SPAWNED at 11.299395 sec
HIGH PRIO joined
MID PRIO joined
CS-L3 CS-L4 CS-L5 CS-L6 CS-L7 CS-L8 CS-L9 CS-L10
CS-L LEAVING

CS-L EXIT

**** LOW PRIO 3 on core 0 UNPROTECTED CRIT SECTION WORK COMPLETED at 11.540353 sec
LOW PRIO joined
START SERVICE joined
All threads done
```

Figure 3: Output of 'pthread3ok' code

b) Fix the deadlock so that it does not occur by using a random back-off scheme to resolve.

The deadlock is fixed such that the thread execution is serialised. Thread 2 is only created after the termination of Thread 1; this makes sure that both the threads would not be acquiring the resources at the same time. This is shown in the figure below.

```
anuhya@raspberrypi:~/Desktop/example-sync-updated-2 $ ./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: f76a2440 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: f76a2440 done
All done
```

c) For the unbounded inversion, is there a real fix in Linux – if not, why not? What about a patch for the Linux kernel? For example, Linux Kernel.org recommends the RT_PREEMPT Patch, also discussed by the Linux Foundation Realtime Start and this blog, but would this really help? 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?

Currently, for unbounded priority inversion there is a real fix for only user space mutexes- futexes. For programs that implement kernel modules and drivers which use syscalls to the kernel, there is no real fix. Linux kernel was originally built to be a general-purpose multiuser operating system. So, it was tuned to maximise average throughput even at the expense of latency, while real-time operating systems attempt to minimize latency with reduced throughput. Linux can be considered to be unfit for real-time use for the following reasons:

1. Kernel system calls are not preemptible. Once a process enters the kernel, it cannot be preempted until it is ready to exit the kernel.
2. “Fairness” in Scheduling: the conventional Linux scheduler (CFS) does its best to be fair to all processes. Thus, the scheduler may give the processor to a low priority process that has been waiting a long time, even though a higher priority process is ready to run.

The second reason can be remediated by using an alternate scheduling process ie. SCHED_FIFO. One of the approaches being considered to give Linux a more deterministic behaviour is modifying the kernel to make it more responsive. This involves introducing additional pre-emption points in the kernel to reduce process latency. This is what the RT_PREEMPT patch aims to do: it minimises the amount of kernel code that is non-preemptible, while also minimizing the amount of code that must be changed in order to provide this added pre-emptibility. It does this with the following mechanics in place: high resolution timers, threaded interrupt handler, rt_mutex and sleeping spinlocks. All the spinlocks are converted to mutexes and there is priority inheritance inside the kernel to all the sleeping locks. Thus, with the patch, the kernel is able to provide real-time capabilities. The issue of unbounded priority inversion caused by the kernel during user-space tasks can now be prevented as the latter can now prevent them. However, the issue is not fully resolved as the kernel is not yet 100% pre-emptible: the scheduler, timing code.

The blog mentions that PREEMPT_RT patch makes the Linux into a real-time system by removing all unbounded latencies. Since the problem of unbounded priority inversion is not fully resolved even with the RT_PREEMPT patch, the decision to switch from a Linux based system to RTOS is utility and application based. If the system is a soft real time required for applications like multi-media, or audio and other real-time service where an occasional missed deadline will not cause loss of life or property, Linux can be used to provide predictable response services. These systems can also make use of the extensions in RT-LINUX to provide priority inversion protection semaphores (futex). However, hard real-time services remain best implemented by an RTOS because applications can run with deterministic minimal latency from an interrupt driven service request. With issues like context switch latency, kernel pre-emptability or interrupt handling latency faced in Linux, RTOS remains the best option for hard-real time services.

4) Review POSIX-examples and POSIX_MQ_LOOP and build the code related to POSIX message queues and run them to learn about basic use.

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

The simple message queue code is demonstrated in the code ‘posix_mq’, shown in Figure 4. The heap message queue is demonstrated in the code ‘heap_mq’, shown in Figure 5.

```

anuhya@raspberrypi:~/Desktop/code/question_4 $ sudo ./posix_mq

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

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

```

Figure 4: Output of simple message queue.

```

anuhya@raspberrypi:~/Desktop/code/question_4 $ sudo ./heap_mq

Sender Thread Created with rc=0
sender - thread entry

Receiver Thread Created with rc=0
pthread join send
sender - sending message of size=93
message sent: This is a test, and only a test, in the event of real emergency, you would be instructed....
Receiver - thread entry
send: message successfully sent, 0x0xf65005f8
receive: msg This is a test, and only a test, in the event of real emergency, you would be instructed.... received with priority = 30, rc = 4
sender - sending message of size=93
message sent: This is a test, and only a test, in the event of real emergency, you would be instructed....
send: message successfully sent, 0x0xf6500660
receive: msg This is a test, and only a test, in the event of real emergency, you would be instructed.... received with priority = 30, rc = 4
sender - sending message of size=93
message sent: This is a test, and only a test, in the event of real emergency, you would be instructed....
send: message successfully sent, 0x0xf65006c8
receive: msg This is a test, and only a test, in the event of real emergency, you would be instructed.... received with priority = 30, rc = 4
^C

```

Figure 5: Output of heap message queue.

A heap message queue improves the efficiency of a message queue. Pointers are sent as messages rather than data. The pointers are set to point a buffer allocated by the sender thread, and the pointer received is used to access and process the buffer. The receiver thread also frees the buffer to avoid exhaustion of the associated buffer heap. The heap message queue avoids the copy otherwise required, which takes considerable CPU time and wastes memory due to double-buffering of the same data.

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 can be implemented to deal with unbounded priority inversion as the enqueue and dequeue operations are atomic (based upon enable/disable of scheduling or interrupts). The implementation of the message queue should ideally not disable scheduling and should prevent unbounded priority inversion. It aims to minimise the effects of priority inversion. However, the priorities of message queues must be considered according to the task's priorities. However, this still depends on the programmer and not an inherit feature of the RT Linux system.

5) [20 points] 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.

A watchdog daemon is a hardware-based time interval timer that counts down and when it reaches zero, it generates an H-reset signal causing the system to reboot. Critical software services post keep-alives to a system sanity monitor that normally resets the watchdog time before it expires. If the software loses sanity ie. it fails to reset the watchdog timer, then the system can recover by rebooting.

An indefinite deadlock occurs there is an attempt to break the deadlock by having the services drop their requests and re-request them at a random time later. However, if the requests are well synchronised, the system would keep cycling between a deadlock and dropping requests repeatedly.

A watchdog timer can be used to avoid an indefinite deadlock. The watchdog daemon continuously monitors the system by periodically feeding the watchdog timer. It does this by writing a specific sequence of values to the watchdog device file, effectively resetting the timer. If the countdown watchdog timer is not reset periodically by the tasks holding the shared resources, the system can be designed to reset and reboot to recover from software failures. Further, if the system reboots itself more than three times, the system can safe itself-expecting external intervention to fix the issue.

b) Next, to explore timeouts, use your code from #2 and create a thread that 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.

In the code, a third thread is created which uses the pthread_mutex_timedlock() function. If the mutex is already locked, the calling thread-watchdog blocks until the mutex becomes available for the duration of the time passed in the abs_timeout parameter. The update thread runs at 1Hz frequency and the values are printed out every 10 seconds so the values read by the read thread can be verified. Since all the threads have the same priority and are run on the same CPU, the pthread_mutex_timedlock function’s wait terminates when the specified timeout expires (10 seconds). This is because the update thread acquires it every one second. The program output is shown in the figure below.

```
anuhya@raspberrypi:~/Desktop/question_5 $ sudo ./thread
*****Update thread*****Time now: 1689043926582.550537
Latitude: 1911759956.000000, Longitude: 749241873.000000, Altitude: 137806862.000000
Roll: 42999170.000000, Pitch: 982906996.000000, Yaw: 135497281.000000
*****Read thread*****Time now: 1689043926582.734375
Latitude: 1911759956.000000, Longitude: 749241873.000000, Altitude: 137806862.000000
Roll: 42999170.000000, Pitch: 982906996.000000, Yaw: 135497281.000000
*****Update thread*****Time now: 1689043936582.271729
Latitude: 402724286.000000, Longitude: 593209441.000000, Altitude: 1194953865.000000
Roll: 894429689.000000, Pitch: 364228444.000000, Yaw: 1947346619.000000
*****Read thread*****Time now: 1689043936582.499756
Latitude: 402724286.000000, Longitude: 593209441.000000, Altitude: 1194953865.000000
Roll: 894429689.000000, Pitch: 364228444.000000, Yaw: 1947346619.000000
*****Update thread*****Time now: 1689043946582.254150
Latitude: 1275373743.000000, Longitude: 387346491.000000, Altitude: 350322227.000000
Roll: 841148365.000000, Pitch: 1960709859.000000, Yaw: 1760281936.000000
*****Read thread*****Time now: 1689043946582.435059
Latitude: 1275373743.000000, Longitude: 387346491.000000, Altitude: 350322227.000000
Roll: 841148365.000000, Pitch: 1960709859.000000, Yaw: 1760281936.000000
^C
```

However, if the update thread is made to sleep and yield the CPU for 15 seconds every 10 seconds, the ‘watchdog’ function waits on the lock and when no data has been updated, it prints out “No new data available at <time>”. This program output shown in the figure below.

```
anuhya@raspberrypi:~/Desktop/question_5 $ sudo ./thread
no new data available at 1689044016092.879639
*****Update thread*****Time now: 1689044021092.233398
Latitude: 1911759956.000000, Longitude: 749241873.000000, Altitude: 137806862.000000
Roll: 42999170.000000, Pitch: 982906996.000000, Yaw: 135497281.000000
no new data available at 1689044026094.412598
*****Update thread*****Time now: 1689044036092.509277
Latitude: 402724286.000000, Longitude: 593209441.000000, Altitude: 1194953865.000000
Roll: 894429689.000000, Pitch: 364228444.000000, Yaw: 1947346619.000000
no new data available at 1689044036095.650879
no new data available at 1689044046096.854980
*****Update thread*****Time now: 1689044051092.779785
Latitude: 1275373743.000000, Longitude: 387346491.000000, Altitude: 350322227.000000
Roll: 841148365.000000, Pitch: 1960709859.000000, Yaw: 1760281936.000000
no new data available at 1689044056098.097412
^C
```