

EXERCISE 3

BY

Amreeta Sengupta and Vatsal Sheth

03/08/2019

Question 2, Question 3, Question 4 and Question 5 performed on Raspberry Pi.

QUESTION 1

Priority inheritance key points made in paper “Priority Inheritance Protocols: An approach to Real-time Synchronization”

Paper investigates synchronization problems and solutions in context of priority driven preemptive scheduling used in many real-time applications. Protocols discussed here are for uniprocessor systems in terms of binary semaphores. Paper investigates following points:

- Problems in existing synchronization primitives

Using binary semaphores to manage sharing of global data between two threads creates a problem where high priority task must wait for low priority task to finish first as it has locked the resource using semaphore. This causes priority inversion as high priority task has to wait, also it may result in task not meeting its deadline making the real-time system unpredictable. Also, medium priority task may cause interference even if it doesn't depend on these shared resources making this priority inversion unbounded. Theorem 3 make an important result stating that if a task T has N lower priority tasks then, it can be blocked at most by one WCET of critical section of each task with priorities lower than that. We can also understand from paper that each task at max can be blocked by one semaphore once only. This is stated by Theorem 6, which says that if there are M semaphores that can block a task T , then task T can be blocked by at most M times.

- Priority inheritance protocols and its properties

Priority inheritance is a concept where a low priority task (priority L), temporarily during the duration of its critical section, inherits a priority that is one level above that of the high priority task i.e. $H+1$ when it is preempted by a high priority task (priority H). This solves the current problem but may cause a chaining issue if during this period, a task with priority $H+n$ (where n is any number greater than zero) preempts causing the priority of low task to be raised from $H+1$ to $H+n+1$. To overcome this chaining issue concept, Priority Ceiling protocol is used. In this protocol, on preemption, priority of lower task is increased to a ceiling and not $H+1$. This solves the chaining issue, but it causes another problem that it may block a genuine high priority task causing inversion. If priority of task L is raised to $H+n$ on preemption by task of priority H , then if a task with priority between H and $H+n$ comes, then it will be blocked too, which results in inversion. Hence, this protocol demands that the programmer sets the ceiling accordingly. Priority ceiling protocol prevents transitive blocking [Lemma 9]. Transitive blocking is a condition where one task is blocked by some task, which in turn is blocked by another task. As this protocol raises priority of lower task to a ceiling, which automatically avoids this condition of transitive blocking.

- Analyze impact of this protocol on schedulability using RM policy

Priority inheritance using Priority ceiling protocol converts unbounded inversion to bounded inversions which can be then taken care of while designing schedule using RM policy. Maximum block that a higher priority task will get is the sum of WCET of critical section of all tasks with priority less than that. This can be added in the WCET of the task itself while designing schedule using RM policy.

The Schedulability analysis is based on the following assumptions:

- All tasks are periodic.
- Each task has deterministic execution time for both its critical and non-critical section.
- Tasks are not synchronized with any external events i.e. if there is only one task in a system, then it will execute to completion.
- Priorities to the tasks are assigned using RM policy.

Theorem 18 states that a set of n periodic tasks using priority ceiling protocol can be scheduled by RM policy if:

$$\forall i, 1 \leq i \leq n,$$

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

where C_i = Computation time for the i^{th} task

T_i = Request period for the i^{th} task

U_i = Utilization for the i^{th} task

B_i = Worst case blocking time for the i^{th} task

Why the Linus position makes sense or not for priority inheritance:

Linus Torvalds is of the view that priority inheritance is not at all required. Instead he suggests that the issue of priority inversion can be resolved using lockless design or carefully thought out locks. As we know, priority inversion occurs due to locking of shared resources by low priority task which leads to blocking of high priority task. If a task of medium priority preempts in this situation, then it causes further interference and may lead to unbounded inversion. Thus, lockless design means designing the code that doesn't use mutex or semaphore. An alternate solution which he also suggests is to consider all the locking scenarios beforehand so that no priority inversion occurs.

Contrary to this, Inglo argues that locking is inevitable and cannot be ignored. This was observed in the case of Mars Path Finder as well. Designing lockless code on the other hand increases its complexity.

In the context of priority inheritance, we support the view of Inglo. Operating systems in today's world use a fair scheduler and hence it will not result in priority inversion in most cases. This is because each task runs for a fair amount of time and thus, the other tasks won't preempt current task. Despite this, in

cases of SCHED_FIFO policy or in case of RTOS, preemption is bound to happen leading to priority inversion. With increasing number of tasks, this complexity may increase. Lockless designs will require each task to have its own copy of memory rather than shared memory. This is not an efficient approach as it requires large quantity of memory. This further strengthens the view of using priority inheritance as suggested by Inglo. If we consider hard real-time applications then, priority inheritance is advantageous despite its drawback listed above. Also, Linux was never specifically designed for hard real time applications.

Reasoning on whether FUTEX fixes unbounded priority inversion

We think that PI-FUTEX can resolve the problem of priority inversion. Mutex is used to protect the shared memory in user space but the actual toggling will happen in kernel space by requesting a system call. This makes the mutex implementation complex and resource hungry as code needs to jump between user and kernel space. This is implemented by moving the thread to wait queue in case it is blocked for acquiring lock. When the lock is released, OS moves this thread back to ready queue. This scheduling is implemented by requesting a system call to switch the queues.

To overcome the limitations of mutex, Futex was introduced which provides access to wait queues in user space. This is implemented by providing a link between wait queue in kernel space and a futex-word in user space. This gives an advantage as system call is only required to acquire the lock if resource is locked previously. If a thread tries to access the locked resource, then the WAIT call is used to put this thread to sleep and then push it to the wait queue. However, in newer Linux kernel, pthread mutex is implemented using the logic of Futex. Futex itself stands for fast user space mutex. Futex is a system call that waits for some condition to be true. This blocking approach is used for shared memory synchronization.

PI-FUTEX developed by Inglo provides support for priority inheritance in futex. This implementation is similar to the Basic Priority Inheritance Protocol which we used for priority inheritance in task. This means that if the lower priority task has locked the resource using PI-FUTEX, then a preemption by high priority task wanting this resource, will increase the priority of the PI-FUTEX to the level of the high priority task. This results in a transitive priority inheritance and hence the problem of chaining in priority inheritance remains. This chaining may increase worst case block time further.

Despite the limitation of PI-FUTEX in dealing with chaining issues of priority inheritance, it provides better performance and much less overhead compared to other synchronization techniques that use kernel space which makes them viable to resolve unbounded priority inheritance.

QUESTION 2

Thread safety with global variable means that the operations by any thread on the global data should be atomic. If the operations are not atomic and two threads share a global variable, then it can corrupt the data during an ongoing operation by a thread, if a context switch occurs in between. This can be resolved by locking and unlocking operations on shared global data using mutex or semaphore. This makes these operations virtually atomic.

A function is reentrant if it is interruptible without hindering the earlier course of action. Use of global variable in a non-reentrant function may change the value of this global variable during ongoing operations, if preempted by an interrupt which changes this global value. Even protection by mutex or semaphore won't help as an interrupt can still change the flow of execution while waiting for mutex/semaphore lock. Also, a reentrant function should not call non-reentrant functions or use static data or return static data pointers. This ensures that there is no relation between successive calls of same function even if the function calls become recursive as each call will be allocated a new space on stack.

Description of methods and Impact of each to real-time services

- Pure functions that use only stack and have no global memory: In C language, local variables are allocated dynamically on stack during a function call. Thus, by not using global variables, static local variables and any other form of shared resources, the function by itself is thread safe and reentrant. Eliminating the use of global variables and static data ensures that there is no relation between successive calls of same function even if the function calls become recursive as each call will be allocated a new space on stack. It can be implemented by avoiding the use of global variables and static keywords. This approach does not have any significant effect on real time services because threads do not share memory and each thread is implemented to work independently.
- Functions which use thread indexed global data: Thread indexed global data is a concept where each thread can get thread specific data stored in global scope with the use of shared key which is known to all the threads of that process. For each data that needs to be shared between different threads, a specific pthread key needs to be created. This data can be stored and received in different threads using a dynamically allocated variable of the required type and their respective data specific key with the help of pthread_getspecific and pthread_setspecific functions. This implementation gives the advantage of shared global memory without its associated disadvantages. This implementation may have impact on real time services due to memory latency. Maintaining of Thread Local Storage (TLS) can have a fast implementation if OS supports that using Memory Management Unit where it stores this data in a special data segment, which it updates on each thread context switch. In case OS doesn't support this, then taking care of this entire process will require user intervention using mutex or semaphore locks for synchronization, while manually managing this data in global memory.
- Functions which use shared memory global data but synchronize access to it using a MUTEX semaphore critical section wrapper: For making functions that use global variables thread-safe, we can wrap the critical sections using mutex or semaphore. This ensures only one thread has access to this critical section even in multi-core CPUs, making the shared global data protected from updating while it is being used by other threads. This is implemented by using pthread mutex by locking and unlocking the critical section. Only one thread can lock onto a mutex, blocking locks of all other threads until it is unlocked. Similarly, with semaphores, sem_wait is used to block the critical section till other threads use sem_post, making the resource available. By default, semaphores are of counting type which means multiple sem_post can result in semaphore value



greater than one, differentiating it from mutex that is in a way similar to binary semaphore. Semaphores can be named so that they are available between processes and this liberty is not available to mutex. Drawback of this implementation for real time services is that it is a blocking task. This results in wastage of time waiting for locks which may cause unaccounted increase in execution time which affects scheduling deadlines for real time systems.

Screenshot of code output

```
pi@raspberrypi:~/Desktop/Question 2 $ ./ques2
[1549188386 : 317 : 317568 : 317568230] Updated: X=64.672710, Y=30.045032, Z=42.005408, Roll=53.261758, Pitch=33.711820, Yaw=62.612466
[1549188386 : 317 : 317568 : 317568230] Received: X=64.672710, Y=30.045032, Z=42.005408, Roll=53.261758, Pitch=33.711820, Yaw=62.612466
[1549188387 : 317 : 317974 : 317974115] Updated: X=19.910415, Y=27.099201, Z=34.094662, Roll=13.448219, Pitch=56.138684, Yaw=21.178079
[1549188387 : 317 : 317974 : 317974115] Received: X=19.910415, Y=27.099201, Z=34.094662, Roll=13.448219, Pitch=56.138684, Yaw=21.178079
[1549188388 : 318 : 318108 : 318108438] Updated: X=59.834025, Y=1.070560, Z=50.339062, Roll=24.129050, Pitch=23.203092, Yaw=66.033757
[1549188388 : 318 : 318108 : 318108438] Received: X=59.834025, Y=1.070560, Z=50.339062, Roll=24.129050, Pitch=23.203092, Yaw=66.033757
[1549188389 : 318 : 318237 : 318237865] Updated: X=48.104915, Y=47.120793, Z=64.278988, Roll=39.673062, Pitch=64.650981, Yaw=35.810620
[1549188389 : 318 : 318237 : 318237865] Received: X=48.104915, Y=47.120793, Z=64.278988, Roll=39.673062, Pitch=64.650981, Yaw=35.810620
[1549188390 : 318 : 318370 : 318370208] Updated: X=15.062975, Y=53.363465, Z=56.682971, Roll=6.066220, Pitch=7.953184, Yaw=35.148698
[1549188390 : 318 : 318370 : 318370208] Received: X=15.062975, Y=53.363465, Z=56.682971, Roll=6.066220, Pitch=7.953184, Yaw=35.148698
[1549188391 : 318 : 318500 : 318500155] Updated: X=8.539182, Y=1.625894, Z=65.193730, Roll=50.544590, Pitch=54.887651, Yaw=27.905590
[1549188391 : 318 : 318500 : 318500155] Received: X=8.539182, Y=1.625894, Z=65.193730, Roll=50.544590, Pitch=54.887651, Yaw=27.905590
[1549188392 : 318 : 318623 : 318623801] Updated: X=42.157056, Y=3.798066, Z=54.944750, Roll=5.251718, Pitch=17.246285, Yaw=40.083434
[1549188392 : 318 : 318623 : 318623801] Received: X=42.157056, Y=3.798066, Z=54.944750, Roll=5.251718, Pitch=17.246285, Yaw=40.083434
[1549188393 : 318 : 318755 : 318755363] Updated: X=26.429797, Y=6.080310, Z=41.153994, Roll=5.768859, Pitch=30.209360, Yaw=64.357086
[1549188393 : 318 : 318755 : 318755363] Received: X=26.429797, Y=6.080310, Z=41.153994, Roll=5.768859, Pitch=30.209360, Yaw=64.357086
[1549188394 : 318 : 318890 : 318890415] Updated: X=0.802616, Y=7.314275, Z=40.477879, Roll=65.081604, Pitch=46.987336, Yaw=34.128860
[1549188394 : 318 : 318890 : 318890415] Received: X=0.802616, Y=7.314275, Z=40.477879, Roll=65.081604, Pitch=46.987336, Yaw=34.128860
[1549188395 : 319 : 319027 : 319027289] Updated: X=29.892223, Y=62.850311, Z=16.492324, Roll=15.575195, Pitch=68.916531, Yaw=24.445508
[1549188395 : 319 : 319027 : 319027289] Received: X=29.892223, Y=62.850311, Z=16.492324, Roll=15.575195, Pitch=68.916531, Yaw=24.445508
[1549188396 : 319 : 319170 : 319170466] Updated: X=50.723893, Y=6.455713, Z=26.071402, Roll=44.917622, Pitch=57.000303, Yaw=9.959053
[1549188396 : 319 : 319170 : 319170466] Received: X=50.723893, Y=6.455713, Z=26.071402, Roll=44.917622, Pitch=57.000303, Yaw=9.959053
[1549188397 : 319 : 319319 : 319319320] Updated: X=1.823172, Y=28.157359, Z=13.757119, Roll=56.767922, Pitch=33.409077, Yaw=31.003404
[1549188397 : 319 : 319319 : 319319320] Received: X=1.823172, Y=28.157359, Z=13.757119, Roll=56.767922, Pitch=33.409077, Yaw=31.003404
[1549188398 : 319 : 319469 : 319469007] Updated: X=25.851356, Y=59.838873, Z=37.083713, Roll=67.005350, Pitch=65.607733, Yaw=67.293073
[1549188398 : 319 : 319469 : 319469007] Received: X=25.851356, Y=59.838873, Z=37.083713, Roll=67.005350, Pitch=65.607733, Yaw=67.293073
[1549188399 : 319 : 319617 : 319617340] Updated: X=60.362437, Y=66.410349, Z=3.607348, Roll=29.840315, Pitch=60.491953, Yaw=50.594685
[1549188399 : 319 : 319617 : 319617340] Received: X=60.362437, Y=66.410349, Z=3.607348, Roll=29.840315, Pitch=60.491953, Yaw=50.594685
[1549188400 : 319 : 319764 : 319764423] Updated: X=63.969176, Y=19.384176, Z=42.444896, Roll=9.461499, Pitch=34.959371, Yaw=40.361527
[1549188400 : 319 : 319764 : 319764423] Received: X=63.969176, Y=19.384176, Z=42.444896, Roll=9.461499, Pitch=34.959371, Yaw=40.361527
[1549188401 : 319 : 319917 : 319917287] Updated: X=33.907007, Y=14.683263, Z=46.817240, Roll=59.978409, Pitch=59.600885, Yaw=32.817543
[1549188401 : 319 : 319917 : 319917287] Received: X=33.907007, Y=14.683263, Z=46.817240, Roll=59.978409, Pitch=59.600885, Yaw=32.817543
[1549188402 : 320 : 320070 : 320070151] Updated: X=69.937462, Y=61.424057, Z=60.974902, Roll=12.694582, Pitch=47.191979, Yaw=23.383978
[1549188402 : 320 : 320070 : 320070151] Received: X=69.937462, Y=61.424057, Z=60.974902, Roll=12.694582, Pitch=47.191979, Yaw=23.383978
[1549188403 : 320 : 320238 : 320238536] Updated: X=43.697985, Y=2.043336, Z=12.222852, Roll=9.781699, Pitch=69.046886, Yaw=6.830585
[1549188403 : 320 : 320238 : 320238536] Received: X=43.697985, Y=2.043336, Z=12.222852, Roll=9.781699, Pitch=69.046886, Yaw=6.830585
pi
```

This code does thread safe update of structure acc with random values of double type for x, y, z with current timestamp in one thread. In the other thread, it reads this structure and prints it. This thread implementation is done using mutex.

QUESTION 3

Description of deadlock with threads using deadlock.c

Deadlocks occur in a situation where all the processes are waiting for some or the other resource and it gets blocked there indefinitely. For example, consider a situation where process 1 is using resource 1 and waiting for resource 2 to be available. Similarly, process 2 is using resource 2 and is waiting for resource 1 which gives rise to a deadlock situation.

Screenshot of deadlock.c

```
pi@raspberrypi:~/Desktop/example-sync $ ./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 1 got B, trying for A
AC
pi@raspberrypi:~/Desktop/example-sync $ ./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: 1994433648 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 1 got B, trying for A
Thread 2 got B and A
Thread 2 done
Thread 2: 1994433648 done
All done
pi@raspberrypi:~/Desktop/example-sync $ ./deadlock race
Creating thread 1
Thread 1 spawned
Creating thread 2
Thread 1 grabbing resources
Thread 1 got A, trying for B
Thread 1 got A and B
Thread 1 done
Thread 2 spawned
rsrcACnt=1, rsrcBCnt=1
will try to join CS threads unless they deadlock
Thread 2 grabbing resources
Thread 1 got B, trying for A
Thread 2 got B and A
Thread 2 done
Thread 1: 1994527856 done
Thread 2: 198619240 done
All done
pi@raspberrypi:~/Desktop/example-sync $
```

In the code `deadlock.c`, two threads are created. Thread 1 locks resource A first, increases its count and sleeps for 1 second. Then, it locks resource B, increases its count and finally unlocks both the resources. Thread 2 locks resource B first, increases its count and sleeps for 1 second. Then, it locks resource A, increases its count and finally unlocks both the resources. Sleep between locking two resources and unlocking both the resources while exiting in each thread causes a situation where thread 1 is waiting for thread 2 to unlock resource B and thread 2 is waiting for thread 1 to unlock resource A. This creates a situation where deadlock occurs. Passing runtime option “safe” delays creation of thread 2 till thread 1 is completed, making execution of two threads sequential. This ensures there is no deadlock. Similarly, passing the runtime option “race” causes each thread to bypass one second sleep between locking two resources. This ensures that thread 1 gets locked to both the resources before thread 2 tries to lock its first resource. This results in thread 1 getting both the resources and then releasing it, subsequently thread 2 gets the resources. This helps us avoid any deadlock situation.

Screenshot of deadlock_timeout.c

```

pi@raspberrypi:~/Desktop/example-sync $ ./deadlock_timeout
Will set up unsafe deadlock scenario
Creating thread 1
Creating thread 2
Thread 1 started
THREAD 1 grabbing resource A @ 1549186291 sec and 805456998 nsec
will try to join both CS threads unless they deadlock
Thread 2 started
THREAD 2 grabbing resource B @ 1549186291 sec and 805531113 nsec
Thread 2 got B
rsrcACnt=0, rsrcBCnt=1
Thread 1 got A
rsrcACnt=1, rsrcBCnt=1
THREAD 2 got A, trying for A @ 1549186292 sec and 805765904 nsec
THREAD 1 got A, trying for B @ 1549186292 sec and 806094966 nsec
Thread 2 TIMEOUT ERROR
Thread 1 got B @ 1549186294 sec and 806953872 nsec with rc=0
rsrcACnt=1, rsrcBCnt=1
Thread 1 got A and B
THREAD 1 done
Thread 1 joined to main
Thread 2 joined to main
All done
pi@raspberrypi:~/Desktop/example-sync $ ./deadlock_timeout safe
Creating thread 1
Thread 1 started
THREAD 1 grabbing resource A @ 1549186306 sec and 973408503 nsec
Thread 2 got A
rsrcACnt=1, rsrcBCnt=0
THREAD 1 got A, trying for B @ 1549186307 sec and 973647721 nsec
Thread 2 got B @ 1549186307 sec and 973765638 nsec with rc=0
rsrcACnt=1, rsrcBCnt=1
Thread 1 got A and B
THREAD 1 done
Thread 1 joined to main
Creating thread 2
will try to join both CS threads unless they deadlock
Thread 2 started
THREAD 2 grabbing resource B @ 1549186307 sec and 975130221 nsec
Thread 2 got B
rsrcACnt=0, rsrcBCnt=1
THREAD 2 got B, trying for A @ 1549186308 sec and 975297200 nsec
Thread 2 got A @ 1549186308 sec and 975382825 nsec with rc=0
rsrcACnt=1, rsrcBCnt=1
Thread 2 got B and A
THREAD 2 done
Thread 2 joined to main
All done
pi@raspberrypi:~/Desktop/example-sync $ ./deadlock_timeout race
Creating thread 1
Creating thread 2
Thread 1 started
THREAD 1 grabbing resource A @ 1549186320 sec and 797847872 nsec
Thread 2 started
THREAD 2 grabbing resource B @ 1549186320 sec and 797955164 nsec
Thread 2 got B
rsrcACnt=0, rsrcBCnt=1
THREAD 2 got B, trying for A @ 1549186320 sec and 798144956 nsec
Thread 1 got A
rsrcACnt=1, rsrcBCnt=1
THREAD 1 got A, trying for B @ 1549186320 sec and 798283237 nsec
will try to join both CS threads unless they deadlock
Thread 2 TIMEOUT ERROR
Thread 1 got B @ 1549186322 sec and 798413184 nsec with rc=0
rsrcACnt=1, rsrcBCnt=1
Thread 1 got A and B
THREAD 1 done
Thread 1 joined to main
Thread 2 joined to main
All done
pi@raspberrypi:~/Desktop/example-sync $

```

The only difference in this code compared to the one described above is that `mutex_timedlock` API is used for locking second resource in case of both the threads. This ensures that in case of deadlock, after specified amount of timeout, mutex is released automatically returning error code `ETIMEDOUT`. Assignment of different timeout values for mutex in both threads, will result in automatic release of mutex, thus avoiding a deadlock.

Screenshot of deadlock_improved.c

```

pi@raspberrypi:~/Desktop/example-sync $ ./deadlock_improved
Will set up unsafe deadlock scenario
Creating thread 1
Thread 1 spawned
Creating thread 2
Thread 2 spawned
THREAD 1 grabbing resources
Thread 2 spawned
rsrcACnt=1, rsrcBCnt=0
will try to join CS threads unless they deadlock
THREAD 1 got A, trying for B
THREAD 1 got A and B
THREAD 1 done
Thread 1: 1994318960 done
Thread 2 random backed off by 1273224 usecs
THREAD 2 grabbing resources
THREAD 1 got B, trying for A
THREAD 2 got B and A
THREAD 2 done
Thread 2: 1985930352 done
All done
pi@raspberrypi:~/Desktop/example-sync $

```

The issue of deadlock in `deadlock.c` is resolved here by random back-off method. This means that execution of thread 2 needs to be delayed at least by amount of time for which thread 1 sleeps in between locking of resources. We have kept this delay to randomly sleep between 1 to 2 seconds. This ensures that thread 1 is able to lock to both the resources which it releases subsequently and then thread 2 can lock onto these resources.

Description of priority inversion with threads using pthread3.c

If a situation arises where the higher priority task waits while a lower priority task is serviced which can occur in any blocking scenario, then it is called priority inversion. In a case where the low and high priority task share a resource, if high priority task preempts the low priority task that has locked onto the resource, then the high priority task has to wait till the low priority task releases it. Maximum time for which the high priority task is blocked is equal to WCET of the critical section of low priority task. This value can be added to the WCET of high priority task while scheduling using RM policy making sure that no task misses its deadline. This is called bounded priority inversion. If in this scenario, a third medium priority task is added that is not dependent on the shared resource, it can result in the medium priority task preempting the low priority task which is actually working to release resources for the high priority task. Number of times and the execution time of the medium priority task which is causing interference cannot be determined beforehand, making it unpredictable. This is called unbounded priority inversion.

Screenshot of pthread3.c

```
pi@raspberrypi:~/Desktop/example-sync $ sudo ./pthread3 5
interference time = 5 secs
unsafe mutex will be created
Pthread Policy is SCHED_OTHER
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM
Creating thread 0
Start services thread spawned
will join service threads
Creating thread 3
Low prio 3 thread spawned at 1549186702 sec, 231252310 nsec
Creating thread 2
Middle prio 2 thread spawned at 1549186703 sec, 231602518 nsec
Creating thread 1, CSct=1
High prio 1 thread spawned at 1549186703 sec, 231789445 nsec
**** 2 idle NO SEM stopping at 1549186703 sec, 231913924 nsec
**** 3 idle stopping at 1549186704 sec, 231508195 nsec
LOW PRIO done
MID PRIO done
**** 1 idle stopping at 1549186706 sec, 231585173 nsec
HIGH PRIO done
START SERVICE done
All done
```

In this code, runtime argument is provided to specify interference time. It creates four threads, all of which have their CPU affinity set to core 0 to ensure single core behavior and all the threads are scheduled using SCHED_FIFO policy. Thread 0 is created with maximum priority and it is responsible for creating and joining three new threads which have priorities successively less than thread 0. Out of these three threads, thread 1 has the highest priority followed by thread 2 and thread 3. Thread 1 and 3 share common resources which is implemented by wrapping synthetic load (Fibonacci series) using mutex. These two threads iterate Fibonacci series twice by user specified interference time and two seconds sleep in between. Thread 2 has no dependency on shared resources and it just iterates the Fibonacci series once by user specified interference time. Thread 0 creates three threads in order of thread 3, thread 2, thread 1 which have priorities low, medium, high respectively. This will create the scenario of unbounded priority inversion as thread 1 will be blocked because of mutex lock by thread 3 and thread 2 will work as a medium priority interference. Same thing can be verified by the various timestamps at specific locations.

Screenshot of pthread3ok.c

```

All done
pi@raspberrypi:~/Desktop/example-sync $ sudo ./pthread3ok 5
interference time = 5 secs
unsafe mutex will be created
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_OTHER
min prio = 1, max prio = 99
PTHREAD SCOPE SYSTEM
Creating thread 0
Start services thread spawned
Will join service threads
Creating thread 1
High prio 1 thread spawned at 1549186720 sec, 795839022 nsec
Creating thread 2
**** 1 idle stopping at 1549186720 sec, 795897772 nsec
Middle prio 2 thread spawned at 1549186720 sec, 795986939 nsec
Creating thread 3
**** 2 idle stopping at 1549186720 sec, 796031105 nsec
Low prio 3 thread spawned at 1549186720 sec, 796130045 nsec
**** 3 idle stopping at 1549186720 sec, 796172407 nsec
LOW Prio done
MID Prio done
HIGH Prio done
START SERVICE done
All done
pi@raspberrypi:~/Desktop/example-sync $

```

The only difference in this code compared to the one described above is that thread 0 creates three threads in order of thread 1, thread 2, thread 3 which have priorities high, medium, low respectively. This avoids unbounded priority inversion as thread with higher priority is created first and the subsequent thread won't be able to preempt it.

For the unbounded inversion, is there a real fix in Linux – if not, why not?

No, there is no fix for unbounded inversions in Linux without the RT_PREEMPT_PATCH. Priority inheritance is supported through this patch, and nowadays it comes as a part of Linux kernel.

Description of RT_PREEMPT_PATCH and assessment of whether Linux can be made real-time safe

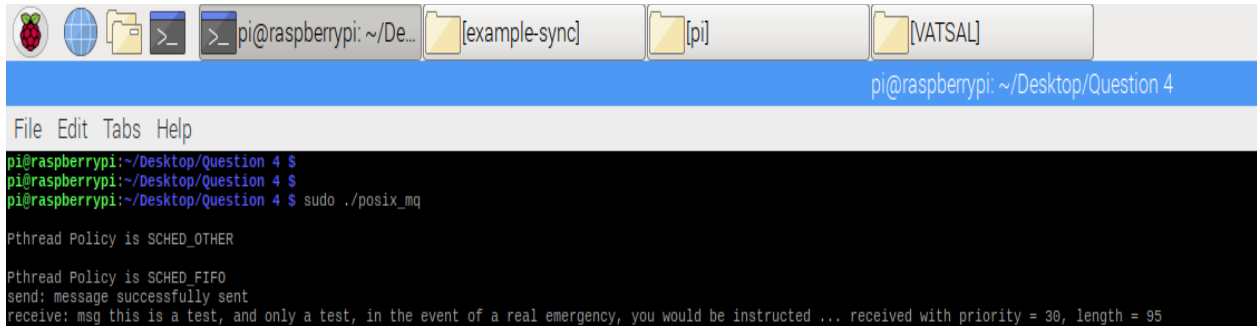
RT_PREEMPT_PATCH makes kernel space preemptible like user space. This ensures that higher priority tasks when ready are executed immediately with some exceptions. This is useful for real-time applications. As this patch makes the kernel preemptible, special care needs to be taken while handling CPU variables. Kernel is not preemptible while handling interrupts, doing bottom half processing, holding a spin lock, read lock, write lock and executing scheduler. This patch gives provision to set priorities for system calls, interrupts etc. With this, we can implement the concept of priority inheritance to avoid unbounded inversions. Whenever, a low priority task is preempted by a high priority task, then the priority is raised to a level higher than that of the high priority task. This ensures medium priority task cannot interfere and the critical section of the low priority task is executed immediately to make the shared resource available for the high priority task. Once this critical section of the low priority task is over, its priority is reverted to its original value. This is called priority inversion and it is useful to avoid unbounded priority inversion and convert it to bounded inversion which can be taken care while designing the task schedule.

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?

It makes sense to use RTOS for both HRT and SRT services because even though the RT_PREEMPT_PATCH takes care of priority inversion in Linux but given the complexity of Linux there is always a possibility of priority inversion. Linux is not an OS for hard real-time system and thus it is difficult to emulate hard real time behavior using this patch. Thus, RTOS seems to be the favorable choice.

QUESTION 4

Screenshot of posix.c



```

pi@raspberrypi: ~/Desktop/Question 4
pi@raspberrypi:~/Desktop/Question 4 $ sudo ./posix_mq
Pthread Policy is SCHED_OTHER
Pthread Policy is SCHED_FIFO
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

```

Objective of this code is to send a string of data from one thread to another. This code creates two threads, one of which runs the sender function while the other one runs the receiver function. Both the threads open their local queue ID using queue name which remains universal. Receive thread opens queue with create flag as this thread is created first and the sender thread opens the queue with just read/write flag. mq_receive and mq_send are blocking functions when the queue is empty and full respectively. We create receive thread first so that the thread will block till the sender sends the string of data. The sender sends the data with message priority 30 and then exits. On receiving, the receiver thread prints the data and exits.

Screenshot of heap.c

```

pi@raspberrypi:~/Desktop/Question 4 $ sudo ./heap_mq
buffer =
ABCEFGHIJKLMNOPQRSTUVWXYZ[\^_`abcdefghijklmnopqrstuvwxyz{|}~\000]
Pthread Policy is SCHED_OTHER

Pthread Policy is SCHED_FIFO
Reading 8 bytes
Message to send = ABCEFGHIJKLMNOPQRSTUVWXYZ[\^_`abcdefghijklmnopqrstuvwxyz{|}~\000]
Sending 8 bytes
send: message ptr 0x0x75c00470 successfully sent
receive: ptr msg 0x0x75c00470 received with priority = 30, length = 8, id = 999
contents of ptr =
ABCEFGHIJKLMNOPQRSTUVWXYZ[\^_`abcdefghijklmnopqrstuvwxyz{|}~\000]
heap space memory freed
Reading 8 bytes
Message to send = ABCEFGHIJKLMNOPQRSTUVWXYZ[\^_`abcdefghijklmnopqrstuvwxyz{|}~\000]
Sending 8 bytes
send: message ptr 0x0x75c00470 successfully sent
receive: ptr msg 0x0x75c00470 received with priority = 30, length = 8, id = 999
contents of ptr =
ABCEFGHIJKLMNOPQRSTUVWXYZ[\^_`abcdefghijklmnopqrstuvwxyz{|}~\000]
heap space memory freed
Reading 8 bytes
Message to send = ABCEFGHIJKLMNOPQRSTUVWXYZ[\^_`abcdefghijklmnopqrstuvwxyz{|}~\000]
Sending 8 bytes
send: message ptr 0x0x75c00470 successfully sent
receive: ptr msg 0x0x75c00470 received with priority = 30, length = 8, id = 999
contents of ptr =
ABCEFGHIJKLMNOPQRSTUVWXYZ[\^_`abcdefghijklmnopqrstuvwxyz{|}~\000]
heap space memory freed
Reading 8 bytes
Message to send = ABCEFGHIJKLMNOPQRSTUVWXYZ[\^_`abcdefghijklmnopqrstuvwxyz{|}~\000]
Sending 8 bytes
send: message ptr 0x0x75c00470 successfully sent
receive: ptr msg 0x0x75c00470 received with priority = 30, length = 8, id = 999
contents of ptr =
ABCEFGHIJKLMNOPQRSTUVWXYZ[\^_`abcdefghijklmnopqrstuvwxyz{|}~\000]
heap space memory freed
Reading 8 bytes
Message to send = ABCEFGHIJKLMNOPQRSTUVWXYZ[\^_`abcdefghijklmnopqrstuvwxyz{|}~\000]
Sending 8 bytes
send: message ptr 0x0x75c00470 successfully sent
receive: ptr msg 0x0x75c00470 received with priority = 30, length = 8, id = 999
contents of ptr =
ABCEFGHIJKLMNOPQRSTUVWXYZ[\^_`abcdefghijklmnopqrstuvwxyz{|}~\000]
heap space memory freed
Reading 8 bytes
Message to send = ABCEFGHIJKLMNOPQRSTUVWXYZ[\^_`abcdefghijklmnopqrstuvwxyz{|}~\000]
Sending 8 bytes

```

Objective of this code is to transfer large quantity of data one way continuously. Unlike the above code, we open queue once in main thread and two child threads will inherit this queue ID. Static image buffer data is first generated in main of 4096 bytes which needs to be exchanged between two threads. Similar to the above code, here also receiver thread is created first so that it blocks and waits till sender sends the data and this cycle can then continue infinitely. Sender thread first mallocs a pointer with size equal to image buffer data. This is then filled with data and id and address to this pointer is passed through queue which basically is passing of double pointer. Both the sender and receiver thread are of same the process, so they can share common memory address. Thus, memory malloced in sender can be accessed by receiver, and we just need to pass address through queue, it can then be freed after receiver thread reads it. Message is passed with priority 30. This data sending cycle goes on.

Similarities between two codes:

- Both codes give higher priority to receiver thread compared to sender thread.
- Both codes ensure error checks in thread creation, message queue opening and message send and receive APIs.
- Both the code uses POSIX message queue as inter process communication to exchange data between two threads.

Differences between two codes:

- In `posix_mq.c` queue is created and opened in receiver function and sender also opens this queue with read/write access, and both threads work with the local copy of queue id. In case of `heap_mq.c`, queue is created and opened in the main thread and both the child threads have access to this queue id.
- In `posix_mq.c` actual data is passed in the queue, that is 95 bytes of character data. Whereas in `heap_mq.c`, as data to be exchanged is large (4096 bytes of image buffer + 4 bytes of integer ID), we pass the address of a pointer to the memory malloced and as both sender and receiver thread are created from same process, they have access to this common memory. Thus, 8bytes of void pointer plus 4 bytes of integer ID i.e. 12 bytes of data is sent through queue.
- In `posix_mq.c`, static initialized global data that will be stored in data segment, is passed. Whereas in `heap_mq.c`, each time memory of image buffer is malloced, data is copied there, and this pointer is passed, and on reception, receiver frees this memory.
- In `posix_mq.c`, data is exchanged only once whereas in `heap_mq.c` data is exchanged continuously.

Description of how message queues would or would not solve issues associated with global memory sharing

Use of Mutex / Semaphore, in shared global memory, may result in priority inversion. It is difficult to account for it, especially if this inversion is unbounded, while making schedule for real time applications. POSIX message queues have the feature of priority messaging. This means that message with higher priority will be placed ahead of lower priority message in queue. If two messages are with same priority then, they are placed according to FIFO logic. This helps in avoiding priority inversion because in the situation if higher priority task preempts lower priority task, then its message is placed in queue ahead of the lower priority message. Thus, higher priority task doesn't need to wait for lower priority task to release the resources, as was the case with shared global data using mutex / semaphore. Thus, bounded

inversions are avoided. Even if medium priority task causes interference, it will be placed at appropriate position in queue as per its priority and that will avoid situation of unbounded inversions, as was the case with mutex / semaphore.

Thus, we can say that POSIX message queue will circumvent the issue of unbounded priority inversion as was the case with mutex / semaphore. Point to note here is not any message queue is capable of solving this inversion issue, only POSIX message queue can do that because it has the feature to assign priority to each message and it places the message in queue according to this priority.

QUESTION 5

Watchdog timers (for the system) and timeouts (for API calls)

Deadlocks occur in a situation where all the processes are waiting for some or the other resource and it gets blocked there indefinitely. For example, consider a situation where process 1 is using resource 1 and waiting for resource 2 to be available. Similarly, process 2 is using resource 2 and is waiting for resource 1 which gives rise to a deadlock situation.

Watchdog timers are hardware timers whose count is periodically updated in order to prevent it from timing out. If this timer times out, then it resets the entire system which is similar to a hardware reset. In Linux, standard interface for watchdog hardware is provided as `/dev/watchdog`. In case the device driver is not available by default, it needs to be configured manually. This can be done by loading the kernel module.

API Timeout in a way is similar to watchdog timer. Timeouts specified either explicitly or by default ensures that the API does not block more than this timeout. This checks that API is not stuck indefinitely. In case of timeouts, it takes required action. For example, in `pthread_mutex_timedlock()`, it unlocks the mutex on timeout.

Description of how the Linux WD timer can help with recovery from a total loss of software sanity

`Watchdog()` is a daemon that periodically updates the watchdog timer to avoid timeout. Its configurations can be updated through a config file. By default, it keeps on opening `/dev/watchdog` and writing to it periodically at least once every minute. Each write further delays the timeout by a minute. In case of a timeout, it resets the system and maintains the log.

In case of loss of software sanity like deadlocks, watchdog daemon won't be able to service the watchdog timer resulting in reset. On reset, all processes are terminated and resources are freed. This ensures a likely scenario where deadlock does not occur again. This is how Linux watchdog timer can help recover from a total loss of software sanity.

Screenshot of code output

```

pi@raspberrypi:~/Desktop/Question 5 $ ./ques5
[1549188437 : 645 : 645595 : 645595815] Updated: X=48.883185, Y=34.725530, Z=1.851496, Roll=5.395881, Pitch=43.256967, Yaw=45.660520
[1549188447 : 646 : 646034 : 646034092] No Data Received
[1549188449 : 646 : 646088 : 646088414] Updated: X=13.694547, Y=65.743409, Z=39.719155, Roll=36.086033, Pitch=19.343912, Yaw=3.846778
[1549188459 : 646 : 646283 : 646283150] No Data Received
[1549188461 : 646 : 646242 : 646242316] Updated: X=22.038057, Y=10.983838, Z=19.876145, Roll=20.061432, Pitch=6.288832, Yaw=28.875522
[1549188471 : 646 : 646448 : 646448771] No Data Received
[1549188473 : 646 : 646395 : 646395280] Updated: X=2.519465, Y=65.208905, Z=42.217128, Roll=41.222203, Pitch=18.202742, Yaw=66.035842
[1549188483 : 646 : 646573 : 646573766] No Data Received
[1549188485 : 646 : 646539 : 646539547] Updated: X=31.306337, Y=14.198791, Z=70.613396, Roll=40.164344, Pitch=23.868697, Yaw=70.884747
[1549188495 : 646 : 646704 : 646704282] No Data Received
[1549188497 : 646 : 646699 : 646699698] Updated: X=3.425691, Y=1.751882, Z=34.610277, Roll=5.277187, Pitch=7.147763, Yaw=6.867244
[1549188507 : 646 : 646857 : 646857351] No Data Received
[1549188509 : 646 : 646850 : 646850579] Updated: X=50.937707, Y=20.842310, Z=1.610653, Roll=19.656861, Pitch=56.928343, Yaw=20.954565
[1549188519 : 647 : 647011 : 647011565] No Data Received
[1549188521 : 647 : 647003 : 647003439] Updated: X=23.503640, Y=7.966400, Z=31.938403, Roll=43.379784, Pitch=28.027832, Yaw=38.227235
[1549188531 : 647 : 647200 : 647200623] No Data Received
[1549188533 : 647 : 647160 : 647160466] Updated: X=1.255307, Y=30.547298, Z=32.436140, Roll=43.472434, Pitch=0.769501, Yaw=50.638882
[1549188543 : 647 : 647324 : 647324785] No Data Received
[1549188545 : 647 : 647315 : 647315982] Updated: X=38.508276, Y=32.075837, Z=64.837673, Roll=38.121673, Pitch=1.240181, Yaw=17.706370
^C
pi@raspberrypi:~/Desktop/Question 5 $

```

In thread 1, the values of the structure (x, y, z, Roll, Pitch, Yaw) are updated. In thread 2, values of the structure are read, printed and mutex is unlocked if pthread_mutex_timedlock returns with no error. In case of timeout (10 seconds) in getting mutex lock, it prints "No data available." Thread 1's critical section is protected using mutex lock and unlock. Before unlocking, this thread is made to sleep for 12 seconds. This sleep ensures that thread 1 does not unlock the mutex before mutex timeout in thread 2. Thus, the flow goes on, where thread 1 updates the structure every 12 seconds and thread 2 always results in timeout, printing no data available.

REFERENCES

- “Real Time Embedded Components and Systems with Linux and RTOS” - Sam Siewert, John Pratt.
- “Priority Inheritance Protocols: An approach to Real-Time Synchronization” - Lui Sha, Ragunathan Rajkumar, John P. Lehoczky.
- Priority Inheritance in Kernel
- “Futexes are Tricky” – Ingo Molnar
- “Overview of Real Time Linux” – Ed Hursey
- Linux man Pages.