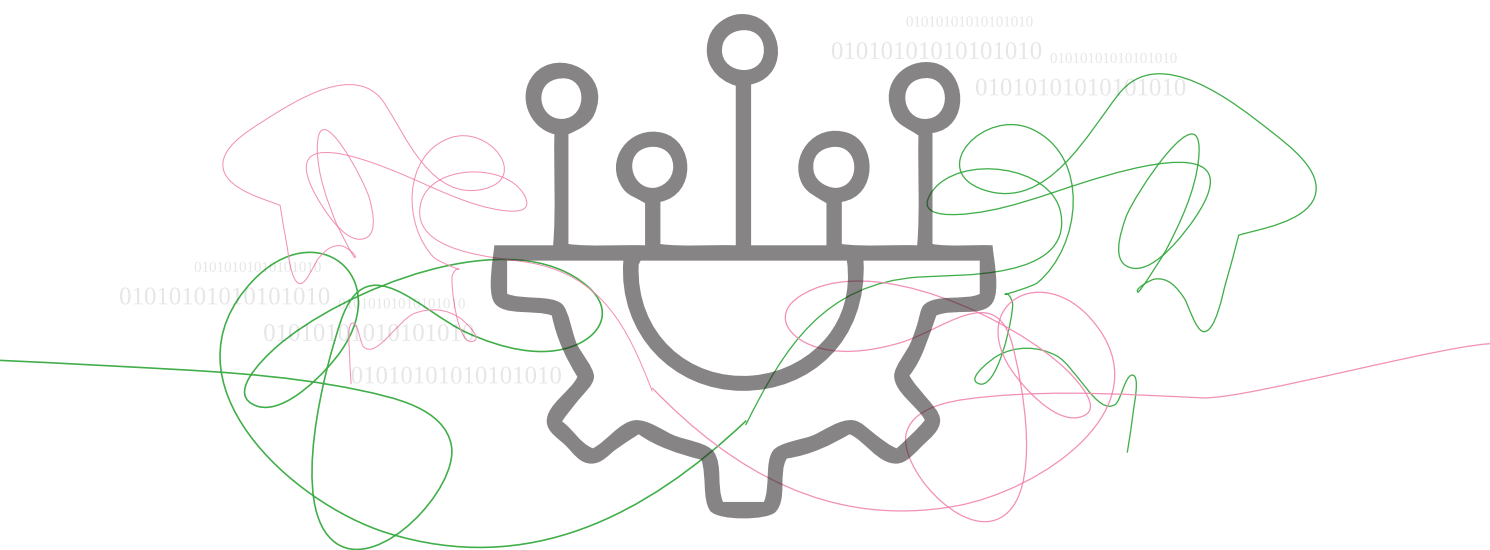


Solutions to Priority Inversion in a Multi-threaded Programming Environment

Priority inversion occurs in priority-based scheduling. This results in slower performance and unexpected results in general-purpose operating systems. This article demonstrates the use of Pthread locking mechanisms to simulate the priority inversion problem and how mutex attributes, called protocols, can be used to avoid priority inversion problems.



Multi-threading has become a widely used programming technique in the design of modern software. This results in the increased responsiveness of the application and the effective use of multi-processor systems. Threads share several resources such as address space, code section and data section, which enables them to achieve faster context switches and easier communication. However, this sharing of resources leads to synchronisation and scheduling problems. One such problem is priority inversion, which occurs in priority-based scheduling. In general-purpose OSs, this results in slower performance and unexpected results.

Many real-time operating systems use pre-emption to ensure rapid response times. This allows a higher priority task to pre-empt a lower priority task that is running. Once the higher priority task is complete, the lower priority task can resume its execution from the point where it was pre-empted.

This pre-emption guarantees worst case performance in safety critical situations. The need to share resources between tasks running in a pre-emptive multi-tasking environment can lead to problems such as deadlock and priority inversion.

In 1997, one such problem arose in the Mars Pathfinder. The rover was collecting meteorological data on Mars. Because of priority inversion, the spacecraft started resetting itself, which resulted in data loss and delays in data collection. The problem was fixed in a few days by patching the onboard software.

Mutex locks provide synchronisation mechanisms between threads. They prevent data inconsistencies that arise due to race conditions. It is a lock that is set before using a shared resource and then released once the resource is used. Whenever the lock is set, no other thread can access the locked region of the code. This ensures synchronised access of shared resources. One such mutex is Pthread mutex, and this locking mechanism supports the priority inheritance protocol.

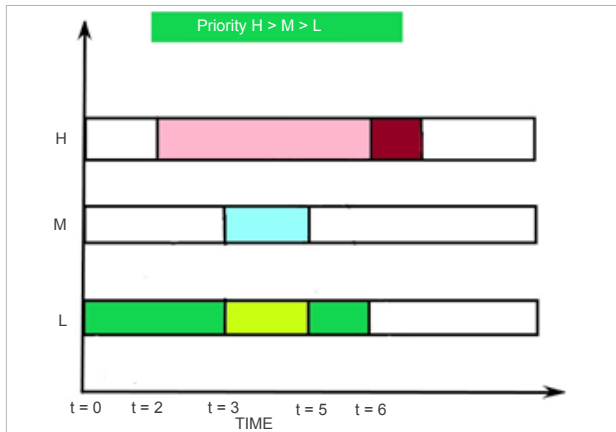


Figure 1: Priority inversion

Figure 1 shows a typical priority inversion scenario. The sequence of events is as follows. The figure depicts three threads: L, M and H. Out of these, L and M share the critical section. L starts executing at $t = 0$ and at $t = 2$, H comes in. Since L holds the lock, H can't pre-empt L and H sleeps. Now at $t = 3$, M comes and pre-empts L since it doesn't share the critical section with L. M finishes at $t = 5$ and L completes its remaining work. Finally, at $t = 6$, H gets to run.

An experimental setup

This entire experiment was carried out on an Ubuntu 17.10 machine with kernel version 4.14. The system specifications are quad-core Intel (R) Core(TM) i7-4710HQ CPU clocked at 2.50GHz with 8GB of RAM.

The entire source code is available at <https://github.com/sourabhkumar0308/PE> and what follows is a walk-through of the code. There are three files in this source code.

The name of the first file in this source code is *pri-inh-singlecore.c*. In this source code, lines 1 to 7 are the header files. The macros on lines 9 to 11 are the duration of time for which L, M and H will execute respectively.

From lines 13 to 16, policy and mutex variables are declared. `pthread_mutex_t` and `pthread_mutexattr_t` are datatypes used to declare a mutex variable and mutex attribute variable, respectively. `SCHED_FIFO` is a real-time scheduling policy with a priority value in the range 1 (low) to 99 (high). This gives the real-time threads higher priority than the normal threads. With this policy, a thread will continue to

run until it voluntarily yields the CPU or gets pre-empted by a higher priority thread.

The function `stick_thread_to_core` (lines 19 to 30) is used to set the CPU affinity. The function `sysconf` is declared in `unistd.h` and returns the number of cores available, which prevents the user from setting invalid cores. `cpu_set_t` is a data structure that represents a set of CPUs. `CPU_ZERO` clears the set and `CPU_SET` adds the CPU to the set. `pthread_setaffinity_np` sets the CPU affinity of a thread and it takes calling the thread ID as its first argument, the length of the CPU buffer pointed by `CPU_SET` in bytes as the second argument, and address of the data structure as its third argument.

The function *doSomething* (lines 38 to 47) simulates a thread that is busy doing some work. It increases a counter for 'n' seconds. *gettime* is a user defined function that returns the current time in seconds. It prints the sequence in which threads arrive and execute.

Lines 49 to 73 make up a low priority thread. Medium (lines 96 to 112) and high (lines 74 to 95) priority threads also have a similar structure except for a few changes in the case of the former. Medium priority threads don't share a critical section with low and high priority threads. The thing to note here is that all of them are made to run on the same core (0).

sched_param is a structure defined in *sched.h*, and is used to set and get the priority value. Here, two variables are declared first. One is *the_priority*, which is used to set the priority value and the other is *get_prio*, which is used to get the priority value that is set.

to get the priority value that is set.
`the_priority.sched_priority = sched_`
`get_priority_min(SCHED_FIFO)` sets
the priority.

`sched_get_priority_min` returns the minimum priority value for the `SCHED_FIFO` policy.

pthread_setschedparam applies the priority to the thread and it takes the ID of the calling thread, policy variable and the address of the priority structure as its arguments.

pthread_getschedparam returns the priority of the thread to the *get_prio* structure and it takes the ID of the calling thread, the address of the policy variable and the address of the priority structure as its arguments.

`pthread_mutex_lock` and `pthread_mutex_unlock` are used to lock and unlock the critical section and they take the `pthread_mutex_t` type variables as their arguments.

These functions are defined in *pthread.h*. They take the address of the *pthread_mutex_t* type variable as their argument.

[illegible]

Figure 2: L is running

```

✖️🐞 sourabh@sourabh-Lenovo-Y50-70: ~
M is running with priority 89 on cpu: 2
M is running with priority 89 on cpu: 2
M is running with priority 89 on cpu: 2
M is running with priority 89 on cpu: 2
M is running with priority 89 on cpu: 2
M is running with priority 89 on cpu: 2
M is running with priority 89 on cpu: 2
M is running with priority 89 on cpu: 2
M is running with priority 89 on cpu: 2
M is running with priority 89 on cpu: 2
L is running with priority 1 on cpu: 0
L is running with priority 1 on cpu: 0
L is running with priority 1 on cpu: 0
M is running with priority 89 on cpu: 2
M is running with priority 89 on cpu: 2
M is running with priority 89 on cpu: 2
M is running with priority 89 on cpu: 2
L is running with priority 1 on cpu: 0
L is running with priority 1 on cpu: 0
L is running with priority 1 on cpu: 0
L is running with priority 1 on cpu: 0

```

Figure 3: L and M running simultaneously

The functions in lines 119 to 122 and 129 to 132 just initialise the mutex attribute and then apply those attributes to the shared mutex variable. The attributes can be protocol (how a thread behaves in terms of priority when a higher-priority thread wants the mutex) and robustness (what happens when you acquire a mutex and the original owner died while possessing it). `pthread_mutexattr_init` takes the address of `pthread_mutexattr_t` type variable as its argument. `pthread_mutex_init` takes the address of `pthread_mutex_t` type variable and `pthread_mutexattr_t` type variable as its argument.

The code on lines 123 to 126 is used to set the inheritance protocol for the mutex attribute variable. First the mutex attribute is initialised and then the protocol is set. The protocol value `PTHREAD_PRIO_INHERIT` makes a thread inherit the priority of a thread it is blocking.

The code in lines 134 to 140 is for thread creation. `pthread_create` is used to create threads in the calling process. `pthread_t` is the data type used to create thread variables. It takes four parameters as its argument. The first argument is the thread address, while the second argument is used to set the thread attributes such as stack size, scheduling policy and priority. If `NULL` is specified, then the default attributes are applied. The third argument is the function that the thread will execute and, finally, the fourth argument is the argument for the thread function. A single argument can be directly passed here, but if there is more than one argument, then we must declare a structure for arguments and pass the address of the structure.

`pthread_join` binds the threads to the main process so that it waits for the threads to finish execution. It is used to synchronise thread activities that work like a wait system call in process.

The functions `pthread_mutexattr_destroy` and `pthread_mutex_destroy` in lines 137 to 142 destroy a mutex and mutex attribute object; the object becomes uninitialised.

The name of the second file in this source code is `pri-inv-singlecore.c`. The

```
sourabh@sourabh-Lenovo-Y50-70: ~
H is running with priority 99 on cpu: 1
H is running with priority 99 on cpu: 1
H is running with priority 99 on cpu: 1
H is running with priority 99 on cpu: 1
H is running with priority 99 on cpu: 1
H is running with priority 99 on cpu: 1
H is running with priority 99 on cpu: 1
H is running with priority 99 on cpu: 1
H is running with priority 99 on cpu: 1
H is running with priority 99 on cpu: 1
H is running with priority 99 on cpu: 1
H is running with priority 99 on cpu: 1
H is running with priority 99 on cpu: 1
H is running with priority 99 on cpu: 1
H is running with priority 99 on cpu: 1
H is running with priority 99 on cpu: 1
H is running with priority 99 on cpu: 1
H is running with priority 99 on cpu: 1
```

Figure 4: H is running

```
sourabh@sourabh-Lenovo-Y50-70: ~/PE/PE
sourabh-Lenovo-Y50-70# ./a.out

L is running with priority 1

M is running with priority 89
```

Figure 5: L is pre-empted by M

```
sourabh@sourabh-Lenovo-Y50-70: ~
L is running with priority 1

L is running with priority 1

L is running with priority 1

L is running with priority 1

L is running with priority 1

L is running with priority 1

L is running with priority 1

L is running with priority 1

L is running with priority 1

L is running with priority 1

L is running with priority 1

H is running with priority 99
```

Figure 6: L executes again and then H runs

```
sourabh@sourabh-Lenovo-Y50-70: ~
L is running with priority 1

L is running with priority 1

L is running with priority 1

L is running with priority 1

L is running with priority 1

L is running with priority 1

L is running with priority 1

L is running with priority 1

L is running with priority 1

L is running with priority 1

L is running with priority 1

H is running with priority 99

M is running with priority 89
sourabh-Lenovo-Y50-70# █
```

Figure 7: Priority inheritance

code is the same as that for `pri-inv-singlecore.c` except that it doesn't have the priority inheritance protocol applied to `pthread_mutex`.

The third file in this source code is named `no-inv-multicore.c`. The code is the same as that for `pri-inv-singlecore.c` except for the changes mentioned above (`pri-inv-singlecore.c`), and also all the threads are made to run on different cores (1, 2 and 3).

Results

Setup: The experiments were conducted for single-core and multi-core systems. The program consisted of three threads with different priorities. Let us denote them as L, M and H where L, M and H are lower, medium and higher priority threads respectively.

The real-time scheduling policy `SCHED_FIFO`, which is a first in, first out scheduling algorithm without time slices, is used. L and H share a critical section while M doesn't share anything with either of the threads.

Execution on multi-core systems: The execution of the program in a multi-core environment showed normal behaviour. Since more than one core is available for threads to run, M doesn't pre-empt L; instead, there is a simultaneous execution of L and M. When L finishes execution, it releases the mutex lock and, finally, H gets to execute.


Execution in a single-core environment:

For single-core execution, the CPU affinity is set. CPU affinity binds a thread to a core. This forces all the threads to run on a single core.

The behaviour of the program is as follows. L starts execution; meanwhile, M and H arrive. H waits for the mutex lock and M starts executing by pre-empting L. When M finishes off, L continues and runs to completion. When the lock is released, H gets to execute and runs to completion. This happens because they are all running on a single core. M has a higher priority than L; so it can pre-empt L without waiting for any resources. H must wait because it shares the critical section with L, and can run only when L releases the mutex lock. This is what we call priority inversion. When H was supposed to run, in its turn, M gets to run.

Using priority inheritance: To overcome the above problem, the priority inheritance protocol

is used. For setting this protocol, the `pthread_mutexattr_setprotocol` function is used and the value of the protocol is `PTHREAD_PRIO_INHERIT`. When a thread blocks a higher priority thread owing to a mutex, with `PTHREAD_PRIO_INHERIT` it inherits the priority of the higher priority thread being blocked and executes with that priority. Once the execution is over, it goes back to normal priority. In such a case, M cannot pre-empt L; hence, L completes its execution with boosted priority. Once it releases the mutex lock, H starts executing and, finally, when H finishes off, M starts executing.

Priority inversion is a problematic scenario that is more common in single-core processor systems. In the case of multi-core processor systems, this problem is avoided because of the execution of threads on different cores. Multiple solutions exist for this problem such as priority ceiling, random boosting, etc. We have used the priority inheritance protocol to avoid priority inversion. Pthread mutex locks not only provide a synchronisation mechanism, but also provide support for priority inheritance. **END** 

References

- [1] Tarek Helmy and Syed S. Jafri. 'Avoidance of Priority Inversion in Real-Time Systems Based on Resource Restoration', International Journal of Computer Science & Applications © 2006 Technomathematics Research Foundation Vol. III, No. I, pp. 40 – 50
- [2] Michael Barr (2002). 'Introduction to Priority Inversion', available at <https://www.embedded.com/electronics-blogs/beginner-s-corner/4023947/Introduction-to-Priority-Inversion>
- [3] Riset Mahmud Pathan. 'Report for the Seminar Series on Software Failures, Mars Pathfinder: Priority Inversion Problems', available at http://www.cse.chalmers.se/edu/year/2015/course/EDA222/Documents/Misc/Report_MarsPathFinder.pdf

By: Sourabh Kumar and Prof. B. Thangaraju

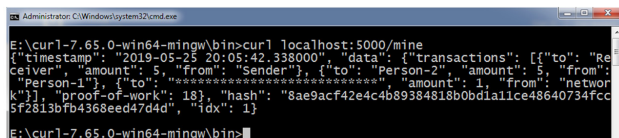
The authors are associated with the Open Source Technology Lab in the International Institute of Information Technology, Bengaluru. Sourabh Kumar can be reached at sourabh.kumar@iiitb.org.

Continued from Page 67....



```
E:\>cd Python27
E:\Python27>cd blockchain
E:\Python27\blockchain>python blockchainserver.py
* Serving Flask app "blockchainserver" (lazy loading)
* Environment: production
  WARNING: This is a development server. In a production environment,
  use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Figure 4: Performing the transaction using cURL



```
E:\curl-7.65.0-win64-mingw\bin>curl localhost:5000/mine
{"timestamp": "2019-05-25 20:05:42.338000", "data": {"transactions": [{"to": "Receiver", "amount": 5, "from": "Sender"}, {"to": "Person-2", "amount": 5, "from": "Person-1"}, {"to": "Person-1", "amount": 1, "from": "network"}], "proof-of-work": 18, "hash": "8ae9ac42e4c4b89384818b0bd1a1ce48640734fcc5f2813bf4368ec47d4d", "idx": 1}}
E:\curl-7.65.0-win64-mingw\bin>
```

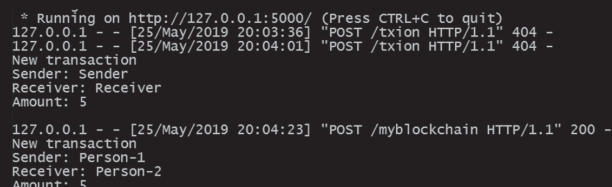
Figure 5: Mining the records and transactions

blockchain using a Web server so that there is distributed deployment.

```
$ curl "http://localhost:5000/myblockchain" -d '{"from": "Sender", "to": "Receiver", "amount": 5}' -H "Content-Type: application/json"
```

Using cURL, the transaction can be implemented and its impact on the blockchain can be visualised. The cURL library for the Windows OS can be installed from <https://curl.haxx.se/windows/>.

As depicted in Figure 6, the execution of the code and the overall implementation with all the records and transactions can be analysed so that there is transparency in the operations, preventing any hacking attempts. Using Proof of Work



```
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [25/May/2019 20:03:36] "POST /txion HTTP/1.1" 404 -
127.0.0.1 - - [25/May/2019 20:04:01] "POST /txion HTTP/1.1" 404 -
New transaction
Sender: Sender
Receiver: Receiver
Amount: 5

127.0.0.1 - - [25/May/2019 20:04:23] "POST /myblockchain HTTP/1.1" 200 -
New transaction
Sender: Person-1
Receiver: Person-2
Amount: 5
```

Figure 6: Recording all transactions on the server

(PoW), the integrity of transactions is logged and committed.

Scope for research and development

In the current scenario, governments as well as corporate organisations are striving towards implementing blockchain technology for secured applications. For such integrations, there is a need to associate the secured algorithms of Proof of Work (PoW) to ensure the privacy and integrity in the implementations. Research scholars and forensic scientists can make use of blockchain technologies to make accurate predictions about specific identities, for forensic as well as law enforcement scenarios. **END** 

By: Dr Gaurav Kumar

The author is the MD of Magma Research and Consultancy Services, Ambala. He is associated with various universities, institutes and autonomous organisations, where he delivers lectures, conducts technical workshops and consults on the latest technologies and tools. He can be contacted at kumargaurav.in@gmail.com and www.gauravkumarindia.com.