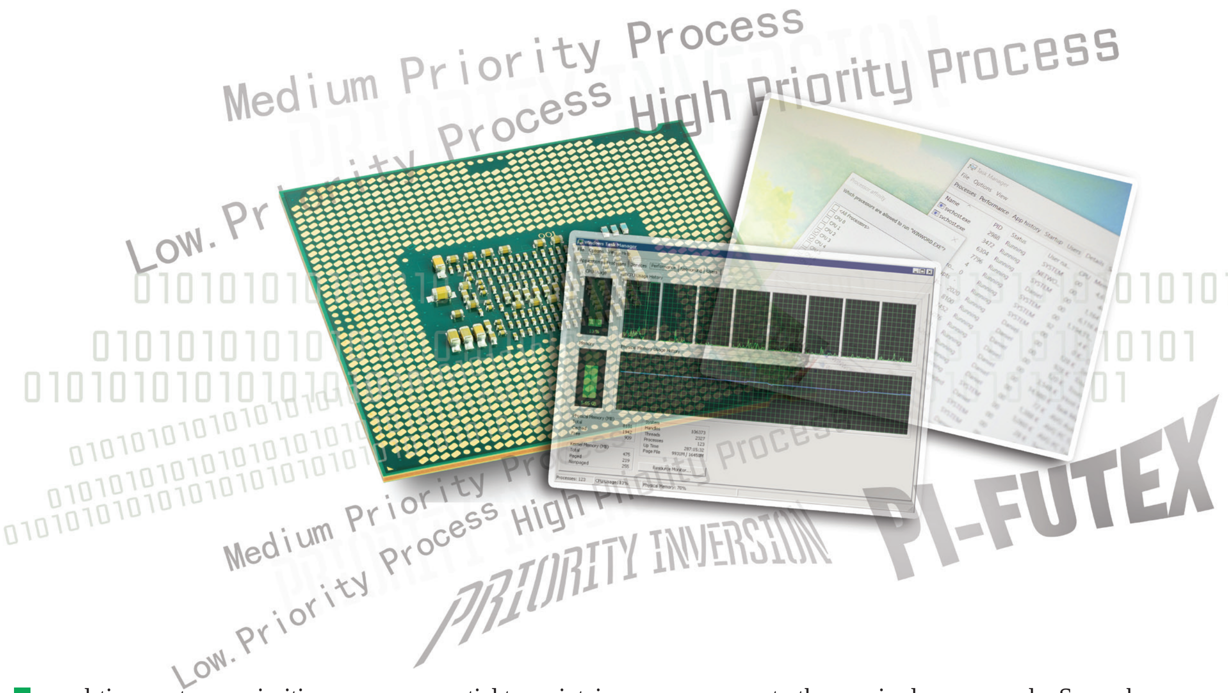
Get Rid of Priority Inversion with PI-Futex

This article helps the reader understand the priority inversion problem of process scheduling at the user space level and provides solutions for it. Also discussed are priority inversion with a semaphore, avoiding priority inversion by using different CPU affinities for processes, and priority inheritance with PI-Futex.



In real-time systems, priorities are very essential to maintain predictability. The scheduler of these systems ensures that the higher priority task should be executed before a low priority task. For this purpose, high priority tasks may pre-empt low priority tasks. But as tasks share resources, those independent of a particular resource but with a lower priority can prevent the highest priority ready task from running when it should. This is 'priority inversion'. Due to this, a critical deadline could be missed and systems might fail.

This problem was experienced by the Mars Pathfinder spacecraft. It landed on Mars and began to transmit data back to the earth. Days later, the flow of information and images was interrupted by a series of total systems resets. The source of the problem was priority inversion, which subsequently caused a critical task missing a deadline. This was identified by a watchdog timer and finally, the only solution was to reset the spacecraft — a short C program was uploaded to the spacecraft, which when interpreted, changed the values of the mutex flag for priority inheritance from false to true. No more systems resets occurred!

In order to achieve locking, a semaphore can be used — it provides synchronisation by restricting access of shared resources to the required process only. Semaphores are non-negative integer values that support two atomic operations, *semaphore-P()* and *semaphore-V()*. P allows waiting for a semaphore to be positive and then decrements it by one, while V allows incrementing a semaphore by one, which implies it wakes up a waiting P. But a semaphore causes the priority inversion problem.

To resolve the priority inversion problem Futex is used. It provides priority inheritance, which is explained later.

An experimental setup

We used a 64-bit, four-core Ubuntu 16.04 LTS system with i686 architecture to conduct the experimental work.

All the code for this setup is available at *https://github.com/DhanashreeMohite/ProjectElectiveWork.*

Priority inversion with a semaphore

sem.c creates a binary semaphore. Here, the *ftok* function returns the key depending on the current path files and ID 'a'. The *semget* creates a semaphore identifier associated with a key. The *semctl* function sets the value of *semval* to *arg.val* for the 0th semaphore of the set.

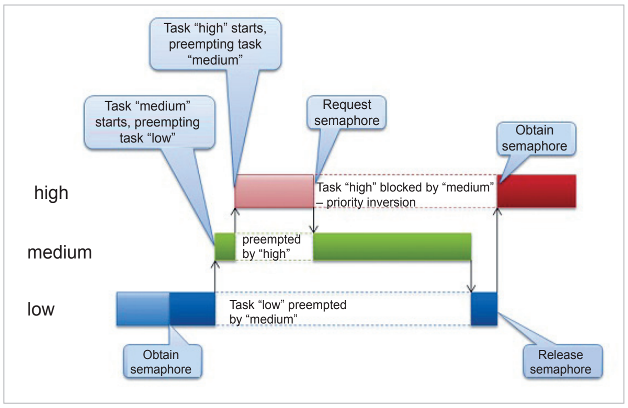

Task “high” starts, preempting task “medium”
Task “medium” starts, preempting task “low”
Request semaphore
Obtain semaphore
high
medium
low
Task “high” blocked by “medium” – priority inversion
preempted by “high”
Task “low” preempted by “medium”
Obtain semaphore
Release semaphore

Figure 1: Priority inversion

common.h includes all the header files and functions that are common to *PriInvLow.c*, *PriInvMed.c* and *PriInvHigh.c*.

The *gettime* function returns the current time. It is used in the function *doSomething*, which provides a delay of exactly ‘n’ seconds. This is considered as a delay for a process to complete its critical section operation. So, in order to understand when a process is going to start and end the critical section, we keep track of the start time and the end time for this function. Here, the *sched getparam* returns the scheduling parameters for the calling process. Using this we can access the priority of the current running process.

PriInvLow.c, PriInvHigh.c and *PriInvMed.c*

Lines 7 to 15 in programs *PriInvLow.c*, *PriInvHigh.c* and *PriInvMed.c* are used to set parameters for the respective processes and, hence, kept common in all three programs with just a change in priority value. Function *sched_setaffinity* makes the process run on a dedicated CPU only, which is mentioned through the CPU affinity mask. In this case, we set this value to 1 for all three processes to make sure that all of them share a single processor. Line No.13 in the priority is assigned to that process. In the case of *PriInvLow.c*, the priority is 20; for *PriInvHigh.c*, the priority is 38; and for *PriInvMed.c*, it is 28.

In the case of *PriInvLow.c* and *PriInvHigh.c*, a key is again generated with the same parameters as that of *sem.c* in order to get the lock of the created semaphore.

The parameters of *struct sembuf* are as shown below:

```
struct sembuf {
ushort sem num; /* semaphore index in array */
short sem op; /* semaphore operation */
short sem flg; /* operation flags */
};
```

Here, *sem num* indicates the number of the semaphore in the set on which the operation has to be performed. *sem op* can be positive, negative or zero. If *sem op* is negative, the value will be subtracted from the semaphore and the resource is locked. Other calling processes for those resources will go to sleep. If *sem op* is positive, then the value will be added to the semaphore and the resource is unlocked. If *sem op* is zero, the calling process will sleep until the value is zero.

The *sem flg* indicates the operation flag. It takes the value *IPC NOWAIT* and *SEM UNDO*. If the semaphore is locked, *sem flg* is *IPC NOWAIT*. If the process wants to acquire a semaphore, instead of actually doing so, it will immediately return *-1* and *errno == EAGAIN*. If *sem flg* is *SEM UNDO*, then the semaphore is undone on exiting the process.

In *PriInvLow.c* and *PriInvHigh.c*, the values are set for the *sembuf* buffer. *semget* will return the ID related to the key. This ID and *sembuf* will be used by the *semop* function to lock and unlock the semaphore. This locking and unlocking operation is performed in *PriInvLow.c* and *PriInvHigh.c* before and after calling the *doSomething* function, respectively.

However, in *PriInvMed.c*, no such locking and unlocking is performed either before or after calling the *doSomething* function.

Different CPU affinities for processes

The sets of codes used for these sections are exactly the same as that of the code mentioned for the ‘Priority inversion with a semaphore’ section, except for a different CPU affinity for each process.

In all three programs – *High.c*, *Low.c* and *Medium.c* – Line No. 9 represents the affinity set for that process. The affinity for the program *High.c* is set to 2, *Low.c* is set to 0 and *Medium.c* is set to 1.

Priority inheritance with PI-futex

Futex is short for ‘fast user space Mutex’. It is used for efficient user space locking. The futex mechanism is a fast, lightweight kernel-assisted locking primitive for user space applications. Futex is an unsigned 32-bit integer on all platforms. It provides for very fast uncontended lock acquisition and release, as the futex state is stored in a user space variable. Atomic operations are used in order to change the state of the futex in the uncontended case without the overhead of a syscall. In contended cases, the kernel is invoked to put tasks to sleep and to wake them up.

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

The ‘uaddr’ represents the address of futex, and which action is to be performed depends on the ‘op’ argument. Other parameters can be used depending on the required operation. There is no *glibc* wrapper for this system call; hence the *syscall* function is used.

The two most commonly used operations are *FUTEX WAIT* and *FUTEX WAKE*.

FUTEX WAIT: The kernel checks the value at *uaddr* with *val*; then if both are the same, it blocks the calling thread/ process. The last two parameters are ignored for this operation and the timeout parameter is not relevant, at least for now.

FUTEX WAKE: The kernel can wake up a maximum *val* number of processes waiting on this futex. The last three parameters are ignored for this operation. In our experiment, we have used PI-FUTEX, which provides priority inheritance. Functions FUTEX LOCK PI and FUTEX UNLOCK PI are used for locking and unlocking, respectively. The futex value stored at *uaddr* is either zero for unlocked or PID of the owner process. The unlocking of futex allows only a high priority process to wake up, thus avoiding priority inversion.

fu.h: The *fu.h* includes all the header files and functions which are common to *PriInhLow.c*, *PriInhMed.c* and *PriInhHigh.c*. It also includes the *gettime* and *dosomething* functions, which do exactly the same work as that mentioned in the previous section. The *fu.h* includes *struct shm* (shared memory) containing the *futex_add* variable, which is about to be shared between *PriInhLow.c* and *PriInhHigh.c* for locking purposes. The *FutexLock* and *FutexUnlock* function calls *FUTEX_LOCK_PI* and *FUTEX_UNLOCK_PI* for *Futex_add*, respectively. When futex is locked, then the value of *Futex_add* is equal to the PID of the process that has the lock; else, it will be zero.

fu.c: The *fu.c* creates the shared memory for *Futex_add*. Here, the *ftok* function returns the key depending on the current path files and ID 'b'. The *shmget* creates an identifier associated with the key. The *shmat* attaches the shared memory segment to the address space of the calling process.

PriInhLow.c*, *PriInhHigh.c* and *PriInhMed.c: As *PriInhLow.c* and *PriInhHigh.c* share a futex, before calling *doSomething*, we call the *FutexLock* function and after *doSomething*, the *FutexUnlock* function is called. Whereas in *PriInhMed.c*, there is no need to call the *FutexLock* and *FutexUnlock* functions to call the *doSomething* function.

Results and discussions

Single-core operation: We started our discussion with a single-core processor. Three processes with different priorities (say, *Hi*, *Med* and *Lo* processes) are allowed to run on a single-core processor by setting the same CPU affinity. The priorities and policies (*sched FIFO*) of the processes are set before entering into the critical section (CS).

In our experimental setup, we considered the time to execute the CS, which records the start and end time of executing the CS for each process. To execute this CS, *Hi* and *Lo* are sharing resources, whereas *Med* executes a different CS. Since *Lo* and *Hi* share resources, they also share the same lock to access this CS. And since *Med* doesn't share any resources with them, it doesn't need the lock.

When processes *Lo* and *Hi* use a binary semaphore: All the code for *sem.c*, *PriInvLow.c*, *PriInvHigh.c* and *PriInvMed.c* is compiled and their executables are stored as *a.out*, *low*, *high* and *med* respectively. *a.out* is executed first, and then all the remaining run, almost together.

First *Lo* acquires the semaphore and enters into the CS. During this period, *Hi* and *Med* arrive. As *Hi* needs the semaphore to enter into the CS, it can't start its execution. But *Med* doesn't need the semaphore and also, its priority is greater than *Lo*; hence, it starts its execution pre-empting *Lo*. Thus, *Lo* requires extra time to complete its task, resulting in extra delay in providing resources to the high priority process (*Hi*).

```
root@dhanashree-Inspiron-5559:/media/dhanashree/New Volume/
Study/sem2/ProjectElective/Final Test# ./low
Low process arrived at : 1522160415
Press any button to enter into critical section

start Time: 1522160428
Low Priority Process started working at : 1522160428
My Priority is : 20

Low.Priority Process

root@dhanashree-Inspiron-5559:/media/dhanashree/New Volume/
Study/sem2/ProjectElective/Final Test# ./high
High process arrived at : 1522160421
Press any button to enter into critical section

High process waiting for lock

High Priority Process

root@dhanashree-Inspiron-5559:/media/dhanashree/New Volume/
Study/sem2/ProjectElective/Final Test# ./med
Med Priority Process arrived at: 1522160425
Press any button to start work...

start Time: 1522160431
Med Priority Process started working at : 1522160431
My Priority is : 28

Medium Priority Process
```

From the above code, we can see that *Med* pre-empted and started its work when *Lo* was actually into its critical section.

```
root@dhanashree-Inspiron-5559:/media/dhanashree/New Volume/
Study/sem2/ProjectElective/Final Test# ./low
Low process arrived at : 1522160415
Press any button to enter into critical section

start Time: 1522160428
Low Priority Process started working at : 1522160428
My Priority is : 20
My Priority is : 20
endTime: 1522160451
Process completed
root@dhanashree-Inspiron-5559:/media/dhanashree/New Volume/
Study/sem2/ProjectElective/Final Test#
```

```
Low Priority Process

root@dhanashree-Inspiron-5559:/media/dhanashree/New Volume/
Study/sem2/ProjectElective/Final Test# ./high
High process arrived at : 1522160421
Press any button to enter into critical section

High process waiting for lock start Time: 1522160451
High Priority Process started working at : 1522160451
My Priority is : 38
My Priority is : 38
endTime: 1522160456
Process completed
root@dhanashree-Inspiron-5559:/media/dhanashree/New Volume/
Study/sem2/ProjectElective/Final Test#

High Priority Process

root@dhanashree-Inspiron-5559:/media/dhanashree/New Volume/
Study/sem2/ProjectElective/Final Test# ./med
Med Priority Process arrived at: 1522160425
Press any button to start work...
start Time: 1522160431
Med Priority Process started working at : 1522160431
My Priority is : 28
My Priority is : 28
end Time: 1522160451
Process completed
root@dhanashree-Inspiron-5559:/media/dhanashree/New Volume/
Study/sem2/ProjectElective/Final Test#

Medium Priority Process
```

From the above code, we can see that *Lo* took extra time than the delay mentioned to come out of the critical section, i.e., almost along with the *Med* process. And then *Hi* started its work. This is due to all the processes running on the same processor, and because semaphore doesn't allow priority inheritance.

When processes *Lo* and *Hi* use PI-Futex: All the code for *fu.c*, *PriInhLow.c*, *PriInhHigh.c* and *PriInhMed.c* is compiled and their executables are stored as *fu*, *low*, *high* and *med* respectively. First *fu* is executed and then all the remaining run almost together. In this case, *Lo* and *Hi* use futex for the locking mechanism.

```
root@dhanashree-Inspiron-5559:/media/dhanashree/New Volume/
Study/sem2/ProjectElective/Final Test# ./low
My pid: 677
Press any button to enter into critical section

Aquired lock...
futex_add : 677
Low Priority Process started working at : 1522220815
Priority : 20
```

```
Low Priority Process

root@dhanashree-Inspiron-5559:/media/dhanashree/New Volume/
Study/sem2/ProjectElective/Final Test# ./high
High Priority Process arrived at 1522220809 and waiting for
lock
My pid: 679
Press any button to enter into critical section

futex_add : 677
Waiting for lock...

High Priority Process

root@dhanashree-Inspiron-5559:/media/dhanashree/New Volume/
Study/sem2/ProjectElective/Final Test# ./med
Medium Priority Process arrived at : 1522220813
Press any button to enter into critical section

Medium Priority Process
```

So, here, when Lo has futex and is performing the CS task, *Hi* is waiting for futex lock. And when *Med* arrives, it can't pre-empt *Lo*.

```
root@dhanashree-Inspiron-5559:/media/dhanashree/New Volume/
Study/sem2/ProjectElective/Final Test# ./low
My pid: 677
Press any button to enter into critical section

Aquired lock...
futex_add : 677
Low Priority Process started working at : 1522220815
Priority : 20
Priority : 20
Work completed at : 1522220830

Low Priority Process

root@dhanashree-Inspiron-5559:/media/dhanashree/New Volume/
Study/sem2/P ojectElective/Final Test# ./high
High Priority Process arrived at 1522220809 and waiting for
lock
My pid: 679
Press any button to enter into critical section

futex_add : 677
Waiting for lock...
High Priority Process started working at : 1522220830
Priority : 38

High Priority Process

root@dhanashree-Inspiron-5559:/media/dhanashree/New Volume/
```

```
Study/sem2/ProjectElective/Final Test# ./med
Medium Priority Process arrived at : 1522220813
Press any button to enter into critical section

Medium Priority Process
```

Lo continues its task and completes it in the desired time. After this, *Hi* acquires the lock and performs its task.

```
root@dhanashree-Inspiron-5559:/media/dhanashree/New Volume/
Study/sem2/ProjectElective/Final Test# ./low
My pid: 677
Press any button to enter into critical section

Aquired lock...
futex_add : 677
Low Priority Process started working at : 1522220815
Priority : 20
Priority : 20
Work completed at : 1522220830
Low Priority Process releasing lock...futex_add : 0
root@dhanashree-Inspiron-5559:/media/dhanashree/New Volume/
Study/sem2/ProjectElective/Final Test#

Low Priority Process

root@dhanashree-Inspiron-5559:/media/dhanashree/New Volume/
Study/sem2/ProjectElective/Final Test# ./high
High Priority Process arrived at 1522220809 and waiting for
lock
My pid: 679
Press any button to enter into critical section

futex_add : 677
Waiting for lock...
High Priority Process started working at : 1522220830
Priority : 38
Priority : 38
Work completed at : 1522220835
High Priority Process releasing lock...
futex_add : 0
root@dhanashree-Inspiron-5559:/media/dhanashree/New Volume /
Study/sem2/P ojectElective/Final Test#

High Priority Process

root@dhanashree-Inspiron-5559:/media/dhanashree/New Volume/
Study/sem2/ProjectElective/Final Test# ./med
Medium Priority Process arrived at : 1522220813
Press any button to enter into critical section

Medium Priority Process started working at : 1522220835
Priority : 28
Priority : 28
Work completed at : 1522220840
root@dhanashree-Inspiron-5559:/media/dhanashree/New Volume/
Study/sem2/ProjectElective/Final Test#

Medium Priority Process
```

After the completion of *Hi*, *Med* is allowed to start its work. Here, even though all processes are executing on a single-core processor, there is no extra time delay in order to provide resources to the high priority process *(Hi)*.

Multi-core operation: All the code for *sem.c*, *Low.c*, *High.c* and *Medium.c* is compiled and the executables are stored as *a.out*, *low*, *high* and *med*, respectively. *a.out* is executed first, and then all the remaining run almost together.

Now, instead of running all processes on a single core, if we allow processes to execute their tasks on different cores, then we can execute processes in parallel. Hence, the priority inversion problem will not occur. Here, different CPU cores are allocated to *Hi*, *Lo* and *Med* by setting different CPU affinities. We have observed that when *Lo* has a semaphore and performs the CS task, *Hi* is waiting for semaphore and *Med* arrives, which then starts performing its own task on another core in a parallel manner. Hence, no extra delay is observed in the execution time of *Lo*, and *Hi* gets resources as soon as *Lo* completes execution and starts its work, even though *Med* is performing its task on another core.

```
root@dhanashree-Inspiron-5559:/media/dhanashree/New Volume/
Study/sem2/ProjectElective/Final Test# ./low
Low process arrived at : 1522160686
Press any button to enter into critical section

start Time: 1522160699
Low Priority Process started working at : 1522160699
My Priority is : 20

Low Priority Process

root@dhanashree-Inspiron-5559:/media/dhanashree/New Volume/
Study/sem2/ProjectElective/Final Test# ./high
High process arrived at : 1522160693
Press any button to enter into critical section

High process waiting for lock

High Priority Process

root@dhanashree-Inspiron-5559:/media/dhanashree/New Volume/
Study/sem2/ProjectElective/Final Test# ./med
Med Priority Process arrived at: 1522160696
Press any button to start work...

start Time: 1522160702
```

```
Med Priority Process started working at : 1522160702
My Priority is : 28

Medium Priority Process

root@dhanashree-Inspiron-5559:/media/dhanashree/New Volume/
Study/sem2/ProjectElective/Final Test# ./low
Low process arrived at : 1522160686
Press any button to enter into critical section

start Time: 1522160699
Low Priority Process started working at : 1522160699
My Priority is : 20
My Priority is : 20
endTime: 1522160719
Process completed
root@dhanashree-Inspiron-5559:/media/dhanashree/New Volume/
Study/sem2/ProjectElective/Final Test#

Low Priority Process

-------------------------------

root@dhanashree-Inspiron-5559:/media/dhanashree/New Volume/
Study/sem2/ProjectElective/Final Test# ./high
High process arrived at : 1522160693
Press any button to enter into critical section

High process waiting for lock start Time: 1522160719
High Priority Process started working at : 1522160719

My Priority is : 38
My Priority is : 38
endTime: 1522160724
Process completed
root@dhanashree-Inspiron-5559:/media/dhanashree/New Volume/
Study/sem2/ProjectElective/Final Test#

High Priority Process

root@dhanashree-Inspiron-5559:/media/dhanashree/New Volume/
Study/sem2/ProjectElective/Final Test# ./med
Med Priority Process arrived at: 1522160696
Press any button to start work...

start Time: 1522160702

Med Priority Process started working at : 1522160702
My Priority is : 28
My Priority is : 28
endTime: 1522160722
Process completed

root@dhanashree-Inspiron-5559:/media/dhanashree/New Volume/
Study/sem2/ProjectElective/Final Test#

Medium Priority Process
```

Using PI-Futex for locking the critical section removes the problem of priority inversion—by inheriting the priority of the low priority process in the critical section when a high priority process waits on futex. Along with processes, futexes can be used for threads also. Futex is comparatively faster than semaphore, as the latter involves the kernel in both contended and non-contended cases, whereas futex involves the kernel only in contended cases. Nowadays, almost all systems have multi-core processors; so by forcing processes to run on different processors, the priority inversion problem can be avoided. END

References

[1] *https://www.embedded.com/electronics-blogs/beginner-s-corner/4023947/Introduction-to-Priority-Inversion*
[2] *https://locklessinc.com/articles/futex_cheat_sheet/*
[3] *https://lwn.net/Articles/360699/*
[4] *https://www.akkadia.org/drepper/futex.pdf*
[5] *http://man7.org/linux/man-pages/man2/futex.2.html*
[6] *https://opensourceforu.com/2013/12/things-know-futexes/*

By: Dhanashree Mohite and Prof. B. Thangaraju

The authors are associated with the Open Source Technology Lab at the International Institute of Information Technology, Bengaluru.