

How to Avoid Priority Inversion and Enable Priority Inheritance in Linux Kernel Programming

This article focuses on avoiding priority inversion with the help of priority inheritance in real-time process execution, which accesses the kernel code. Kernel synchronisation mechanisms such as semaphore and rt-mutex are also covered. Priority inheritance in rt-mutex eliminates the priority inversion problem and hence improves real-time performance in the Linux environment.



Application programs access hardware through the kernel code. Improper synchronisation or locking at the kernel can result in random crashes and unexpected results. To resolve this issue, multiple kernel level synchronisation and locking mechanisms have been developed. This article mainly focuses on semaphore and rt-mutex.

Semaphore: This is a synchronisation mechanism that implements mutual exclusion with a variable and associated library functions. This variable is used to solve critical section problems and to achieve process synchronisation in a multi-processing environment.

rt-mutex: This extends the semantics of simple mutexes by the priority inheritance protocol. A low priority owner of

rt-mutex inherits the priority of a higher priority waiter until rt-mutex is released. If the temporarily boosted owner blocks rt-mutex, it propagates the priority boosting to the owner of the other rt-mutex on which it is blocked. Priority boosting is immediately removed once rt-mutex has been unlocked.

The priority inversion problem

Let's consider three processes with high, middle and low priorities.

Tasks

High priority (H)

Medium priority (M)

Low priority (L)

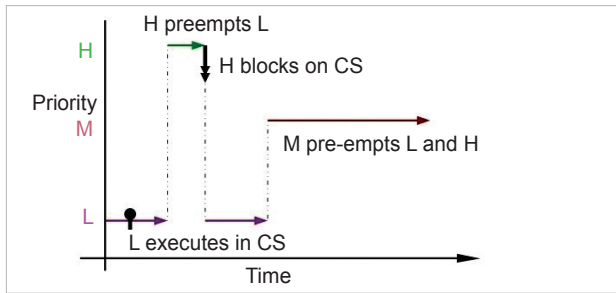


Figure 1: Priority inversion

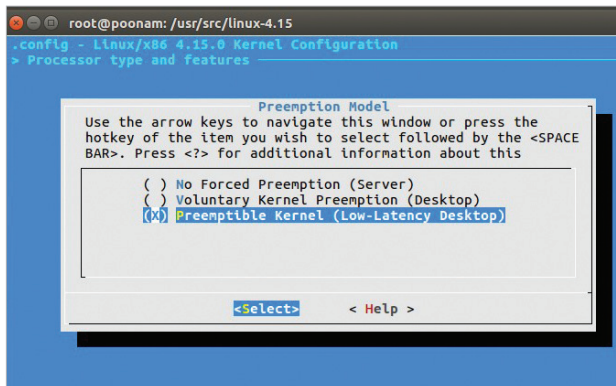


Figure 2: Kernel pre-emption model

Shared resource: Critical section

The shared buffer memory between H and L is protected by a lock (say semaphore). Let's consider the following different scenarios:

- L is running but not in the critical section (CS) and H needs to run. So, in priority based pre-emptive scheduling, H pre-empts L. L then runs once H completes its execution.
- L is running in CS, H needs to run but not necessarily in CS. Even then, H pre-empts L, and then executes. L resumes execution once H finishes its execution.
- L is running in CS. H also needs to run in CS. H waits for L to come out of CS and then executes.

The above three scenarios are normal and don't lead to any problems. The real problem arises when another task of middle priority needs to be executed.

Let us consider a scenario again:

1. L runs, and H tries to access CS while L is executing it.
2. As CS is not available, H sleeps.
3. M arrives. As M has higher priority than L and it doesn't need to be executed in CS, L gets pre-empted by M, which then runs.
4. M finishes, then L executes for the remaining portion and releases CS; only then does H get resources.
5. H executes.

What if M is a CPU bound process which needs a lot of CPU time? In this case, M keeps on executing and apparently, L is waiting for M to get executed, and indirectly H too is waiting on M.

This is known as priority inversion.

An experimental setup

This experimental work is done on Ubuntu 16.04 with kernel version 4.15, and a core processor with 8GB RAM. The vanilla kernel offers the following three options for the pre-emptible kernel. We have used a pre-emptible (low-latency desktop) model as shown in Figure 2.

The source code is available at <https://github.com/Poonam0602/Linux-System-programming/tree/master/Linux%20Kernel%20Locking%20Mechanisms>.

withoutLock.c: In the source code, Lines 1 to 9 represent header files. The `my_read` function is used to retrieve data from the device. A non-negative return value represents the number of bytes successfully read (the return value is a 'signed size' type, usually the native integer type for the target platform). Four arguments should be passed in the read function—the first and second are pointers to the file structure and user buffer, respectively. The third is the size of the data to be read, while the fourth is the file pointer position. When an application program issues the read system call, this function is invoked, and the data is to be copied from the kernel space to the user space using the `copy_to_user` function. `copy_to_user` accepts three arguments—a pointer to the user buffer, a pointer to the kernel buffer and the size of the data to be transferred.

The function `my_write` is similar to `read`. Here, data is transferred from the user space to the kernel space. `Copy_from_user` is used for this data transfer. The delay mentioned in the above function at Line 25 is used to demonstrate how the code works, with screenshots. This delay provides enough time to capture the behaviour on the screen.

Opening the device is the first operation performed on the device file. Two arguments are passed in the `my_open` function. The first argument is the inode structure, which is used to retrieve information such as the user ID, group ID, access time, size, the major and minor number of a device, etc.

The `file_operations` structure (Lines 44 to 49) is an array of function pointers, which is used by the kernel to access the driver's functions. All the device drivers record the `read`, `write`, `open` and `release` functions in the `file_operations` structures. The functions `my_read`, `my_write`, `my_open` and `my_close` are assigned to the corresponding device entry points.

`My_init()` function at Lines 53 to 60 is used to register a character device called `mydevice` using `register_chrdev()`, which returns the major number. The zero passed as the parameter enables the dynamic allocation of a major number to the device.

`init_module()` replaces one of the kernel functions with the `my_init` function. The `exit_module()` function is supposed to undo whatever `init_module()` did, so the module can be unloaded safely.

semaphore.c: This complete code is the same as that of `withoutLock.c` except for some changes mentioned below.

Line 11 represents `DEFINE_Semaphore`, which defines and initialises a global semaphore named `test_Semaphore`.

The `down()` in `my_write()` function acquires the semaphore `test_Semaphore`. If no more tasks can acquire the semaphore, calling this function will put the task to sleep until the semaphore is released. `up()` is used to release the semaphore. Unlike mutexes, `up()` may be called from any context and even by tasks which have never called `down()`.

In Line 62, `sem_init()` initialises `test_semaphore` dynamically at run time.

rt-mutex.c: This complete code is the same as that of `withoutLock.c` except for some changes mentioned below.

`DEFINE_RT_MUTEX` defines and initialises `rt-mutex`. It is useful to define global `rt-mutexes`.

`rt_mutex_lock` (line 24): Locks `rt-mutex`.

`rt_mutex_unlock` (line 28): Unlocks `rt_mutex`

`rt_mutex_init` (line 62): Initialises the `rt` lock to the unlocked state. Initialising of a locked `rt` lock is not allowed.

`rt_mutex_destroy` (line 69): This function marks the mutex as uninitialised, and any subsequent use of the mutex is forbidden. The mutex must not be locked when this function is called.

Compiling the kernel module

To compile the module, refer to the `makefile` available at the GitHub link. After compiling and loading this module, the output of the `printk` statement in the `init_module` is displayed. The major number allocated for `mydevice` can be seen in the output; if not, check the `/proc/devices` file, where it can be seen in the character device list. In my system, a major number 241 was allocated to `mydevice`. This module only registers the device. You also need to create a device file for accessing the device. You can create the device file in the present working directory using the following command:

```
mknod mydevice c 241 0
```

Application programs: `low.c`, `middle.c` and `high.c`

Lines 2 to 13 represent header files. A call to `open()` at Line 22 creates a new open file description, an entry in the system-wide table of open files. This entry records the file offset. A file descriptor is a reference to one of these entries.

The standard Linux kernel provides two real-time scheduling policies, `SCHED_FIFO` and `SCHED_RR`. The main real-time policy is `SCHED_FIFO`. It implements a first-in, first-out scheduling algorithm. When a `SCHED_FIFO` task starts running, it continues to run until it voluntarily yields the processor, blocks it or is pre-empted by a higher-priority real-time task. It has no time-slices. All other tasks of lower priority will not be scheduled until it relinquishes the CPU. Two equal priority `SCHED_FIFO` tasks do not pre-empt each other. `SCHED_RR` is like `SCHED_FIFO`, except that such tasks are allotted time-slices based on their priority and run until they exhaust their time-slice. Non-real-time tasks use the `SCHED_NORMAL` scheduling policy (older kernels had a policy named `SCHED_OTHER`).

For this experiment, we have considered the FIFO

scheduling policy (Lines 25 - 37).

In the standard Linux kernel, real-time priorities range from zero to `MAX_RT_PRIO-1`, inclusive. By default, `MAX_RT_PRIO` is 100. Non-real-time tasks have priorities in the range of `MAX_RT_PRIO` to `(MAX_RT_PRIO + 40)`. These constitute the nice values of `SCHED_NORMAL` tasks. By default, the -20 to 19 nice range maps directly onto the priority range of 100 to 139.



Note: Root privileges are needed to choose schedulers. So, before proceeding to the execution, make sure that you are executing the code with the root login.

`Write()` at Line 42 writes up to `sizeof(bufw)` bytes from the buffer, starting at `bufw` to the file referred to by the file descriptor `fd`.

.c, m.c and h.c: All the code descriptions are the same as mentioned above, except for CPU affinity. These application programs are bound to get executed on a single CPU with the help of CPU affinity.

`void CPU_ZERO (cpu_set_t *set):` This macro initialises the CPU set to be the empty set.

`void CPU_SET (int cpu, cpu_set_t *set):` This macro sets the CPU affinity to `cpu`. The `cpu` parameter must not have side effects since it is evaluated more than once.

`int sched_setaffinity (pid_t pid, size_t cpusetsize, const cpu_set_t *cpuset):` `sched_setaffinity ()` system call is used to pin the given process to the specified processor(s). You can get more information from `man 2 sched_setaffinity`. If

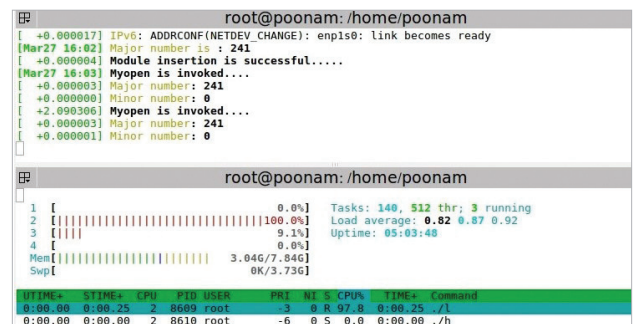


Figure 3: A low priority process is running

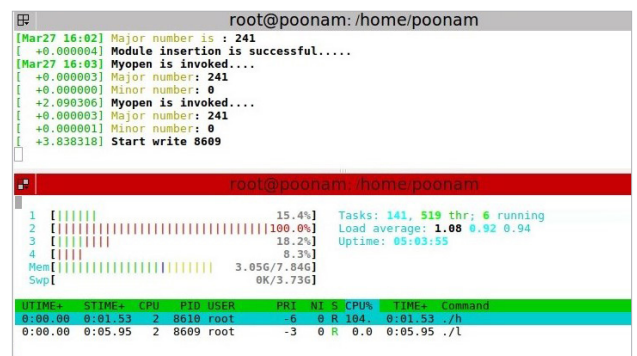


Figure 4: The high priority process pre-empts the low priority process

successful, the function returns zero and the scheduler will in the future take the affinity information into account.

Analysis setup

htop is used to analyse the process execution flow. The parameters, like the CPU number, Utime (the user CPU time, which is the amount of time the process has taken to execute on the CPU in user mode), Stime (the system CPU time, which is the amount of time the kernel has spent executing system calls on behalf of the process, measured in clock ticks) and the time for each application process, are observed.

Please note that the *htop* gets process information from a file called *stat* under each process' *procfs* entry. For processes running a real-time scheduling policy (see *sched_setscheduler(2)*), this is the negated scheduling priority, minus one; that is, a number in the range of -2 to -100, corresponding to real-time priorities 1 to 99.

Hence, the priority of 5 is displayed as -6; that of 3 as -4 and 2 as -3 in *htop* results. Therefore, all figures have process priorities as -6, -4 and -3.

Along with this, *dmesg* is used to check the kernel's response to application processes.

Results and observations

Let us perform a small experiment on a character device driver to check semaphore and rt-mutex lock behaviour at the kernel level. The basic idea is that a device can be accessed by multiple application programs, where two or more devices need to update a shared memory or buffer. In this case, how a pre-emptible kernel behaves is what we learned during this experiment.

The experiment is performed on two platforms – on a single core processor and multi-core processor system.

Three simple application programs are written to check the kernel lock behaviour. Two of the applications try to access the device driver code (which is our critical section) and the third process does not access the device driver.

These three applications are written with different static priorities assigned to them. Let's name these as low (priority 2), middle (priority 3) and high (priority 5).

Uniprocessor systems

Without lock setup: A low priority process is executed first. Then the high priority process comes in, which also wants to write in the same buffer.

Observation: The high priority process pre-empts the low priority process. So, when two processes are executed, the buffer is overwritten by the higher priority process. Thus, when a lower priority process tries to read the buffer content, it gets wrong data.

Semaphore

The setup: A low priority process is executed first. Then a high priority process comes in which also wants to write to the same buffer. A middle priority process comes in which does not want to access the device driver. Middle is a simple application program.

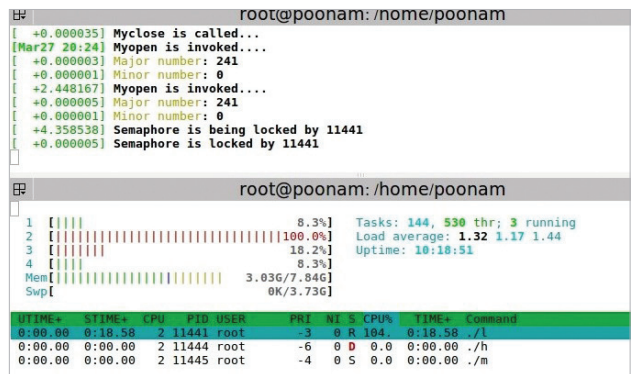


Figure 5: Low executes, high waits for low to release CS

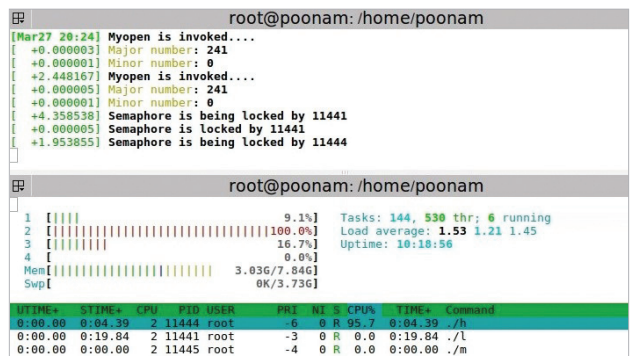


Figure 6: High executes after low completes execution in CS

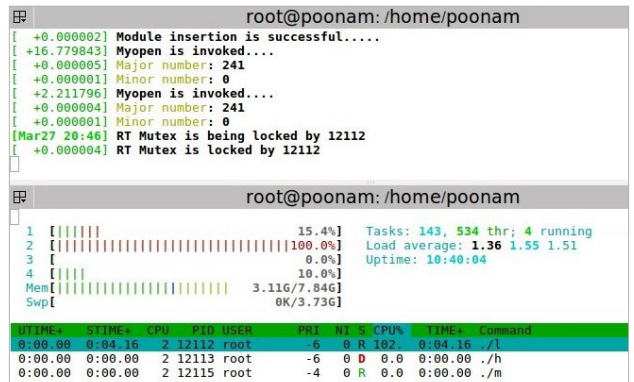


Figure 7: Low process priority gets inherited

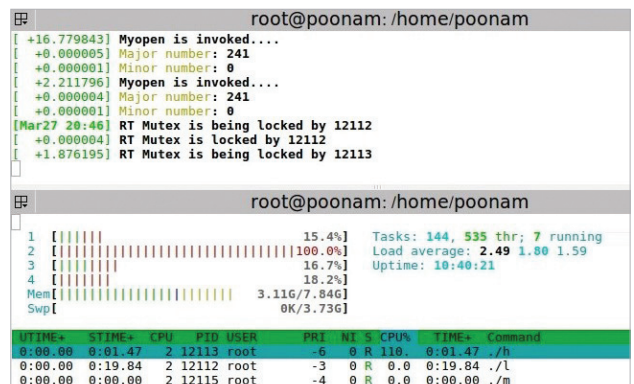


Figure 8: Inherited priority gets reset after execution

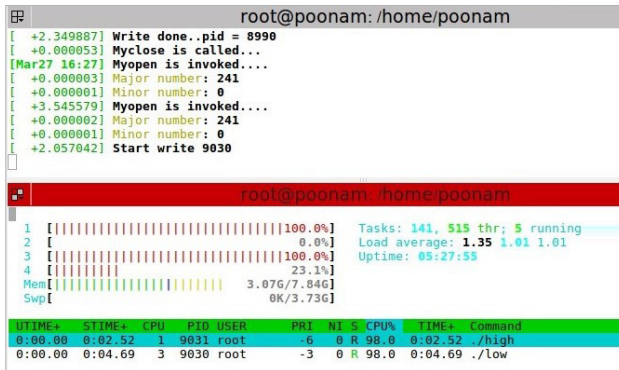


Figure 9: Both processes execute simultaneously

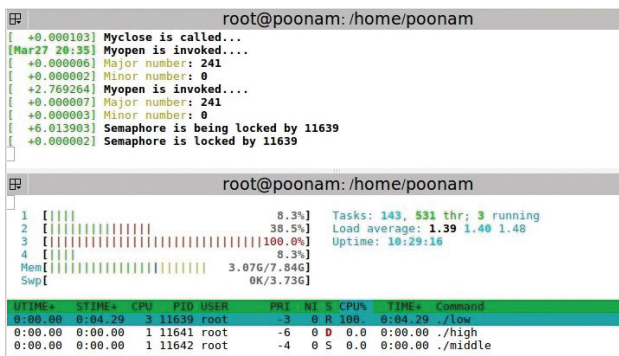


Figure 10: High waits for low to complete the execution of CS

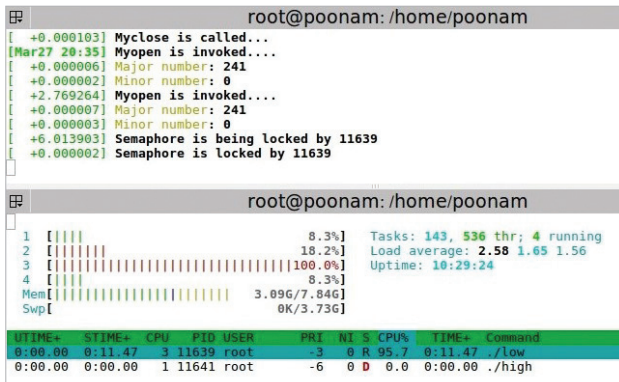


Figure 11: Low resumes the execution in CS

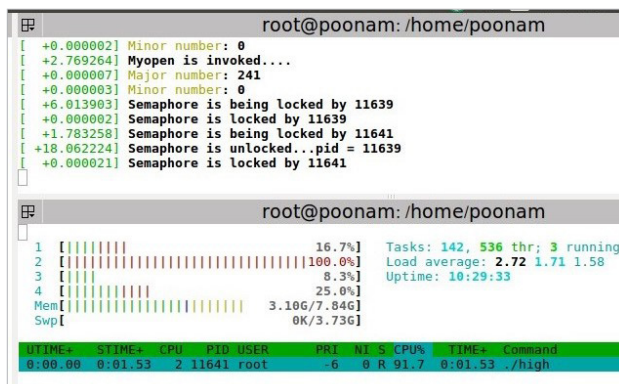


Figure 12: High executes in CS

Observation: When the kernel is PRE-EMPTIVE, that means, at the kernel level, pre-emptions are allowed except for critical sections. Here, the low priority process is running in the critical section, and the high priority and middle priority processes come in. High doesn't pre-empt the low priority process but rather, it waits for the lower priority process to complete executing the critical section. The middle priority process runs in between.

Note that the priorities of processes remain unchanged.

RTMutex

The setup: A low priority process is executed first. Then a high priority process comes in which also wants to write to the same buffer. A middle priority process comes in which does not want to access the device driver. Middle is a simple application program.

Observation: The low priority process is running in critical section, and the high and middle priority processes come in. As soon as the high priority process comes in, the priority of the low priority process gets increased to a high priority value and remains the same till the process completes executing the critical section. So, high doesn't pre-empt the low priority process. After completing the low priority process, like in the case of semaphores, the middle process can't execute as a high priority process is available.

Observe that the priority of low priority process updates to 5 when in the critical section, and returns to 2 after execution.

Multiprocessor systems

Without lock setup: A low priority process is executed first. Then a high priority process comes in, which also wants to write to the same buffer.

Observation: When both processes are running on different processors, both execute in parallel. But the buffer update problem remains the same. At this point, we can't make out which process has updated the buffer.

Semaphore

The setup: A low priority process is executed first. Then a high priority process comes in, which also wants to write to the same buffer. A middle priority process comes in, which does not want to access the device driver. Middle is a simple application program.

Observations: Here, all processes are running on different processors. So, the middle priority process has no dependency on the other two processes. For the other two processes, one waits for the other to complete executing the critical section.

rt-mutex

The setup: The low priority process is executed first. Then the high priority process comes in, which also wants to write in the same buffer. Then the middle priority process comes in and it does not want to access the device driver. Middle is a simple application program.

Continued On Page...92

```

root@poonam: /home/poonam
[ +0.000110] Myclose is called...
[ +5.757576] Myopen is invoked....
[ +0.000004] Major number: 241
[ +0.000001] Minor number: 0
[ +3.126418] Myopen is invoked....
[ +0.000003] Major number: 241
[ +0.000001] Minor number: 0
[Mar27 21:16] RT Mutex is being locked by 12498
[ +0.000002] RT Mutex is locked by 12498

root@poonam: /home/poonam
1  [|||||] 15.4% Tasks: 143, 530 thr; 3 running
2  [|||||] 16.7% Load average: 1.69 1.20 1.12
3  [|||||] 100.0% Uptime: 11:09:53
4  [|||||] 0.0%
Mem[|||||] 3.21G/7.84G
Swp[|||||] 0K/3.73G


UTIME+ STIME+ CPU PID USER PRI NI S CPU% TIME+ Command
0:00.00 0:04.40 3 12498 root -6 0 R 104. 0:04.40 ./low
0:00.00 0:00.00 1 12499 root -6 0 D 0.0 0:00.00 ./high
0:00.00 0:00.00 4 12500 root -4 0 S 0.0 0:00.00 ./middle

```

Figure 13: Priority gets inherited for the low priority process

Observation: Here, all processes are running on different processors. So, the middle priority process has no dependency on the other two processes (just like semaphore). For the other two processes, one waits for the other to complete executing the critical section. Also, priority inheritance is observed.

After the experimentation we observed that rt-mutex with priority inheritance can be used to avoid the priority inversion problem at the device driver level. If rt-mutex is used at the device driver level, user programs can be simplified a lot, and the locking and signalling mechanisms can be saved for critical tasks at the user level.

Also, we found that *PRE-EMPTIBLE Kernel* is not a fully pre-emptible kernel. It still doesn't pre-empt a process executing in the critical section. To get a fully pre-emptible kernel, we need to use an RT pre-emptible patch. **END** 

```

root@poonam: /home/poonam
[ +0.000110] Myclose is called...
[ +5.757576] Myopen is invoked....
[ +0.000004] Major number: 241
[ +0.000001] Minor number: 0
[ +3.126418] Myopen is invoked....
[ +0.000003] Major number: 241
[ +0.000001] Minor number: 0
[Mar27 21:16] RT Mutex is being locked by 12498
[ +0.000002] RT Mutex is locked by 12498

root@poonam: /home/poonam
1  [|||||] 27.3% Tasks: 143, 533 thr; 7 running
2  [|||||] 16.7% Load average: 2.13 1.31 1.16
3  [|||||] 100.0% Uptime: 11:09:59
4  [|||||] 16.7%
Mem[|||||] 3.22G/7.84G
Swp[|||||] 0K/3.73G

UTIME+ STIME+ CPU PID USER PRI NI S CPU% TIME+ Command
0:00.00 0:10.93 3 12498 root -6 0 R 102. 0:10.93 ./low
0:00.00 0:00.00 1 12499 root -6 0 D 0.0 0:00.00 ./high

```

Figure 14: Inherited priority restores after CS

References

- [1] 'Study of Priority Inversion in Embedded Linux', by Chun-tao Man and Peng Li (IEEE 2006) <https://ieeexplore.ieee.org/document/1692154>
- [2] 'Kernel Corner: A Simple Read and Write Pseudo Character Device Driver' by Dr. B. Thangaraju, *Linux For You*, June 2003, pg.71-73.
- [3] 'Linux Device Drivers' by Jonathan Corbet, pg. 42-70. <https://bootlin.com/doc/books/ldd3.pdf>

 By: Poonam S. Warade and Prof. B. Thangaraju

The authors are associated with the Open Source Technology Lab in the International Institute of Information Technology, Bengaluru. Poonam Warade can be reached at waradepoonam@gmail.com.