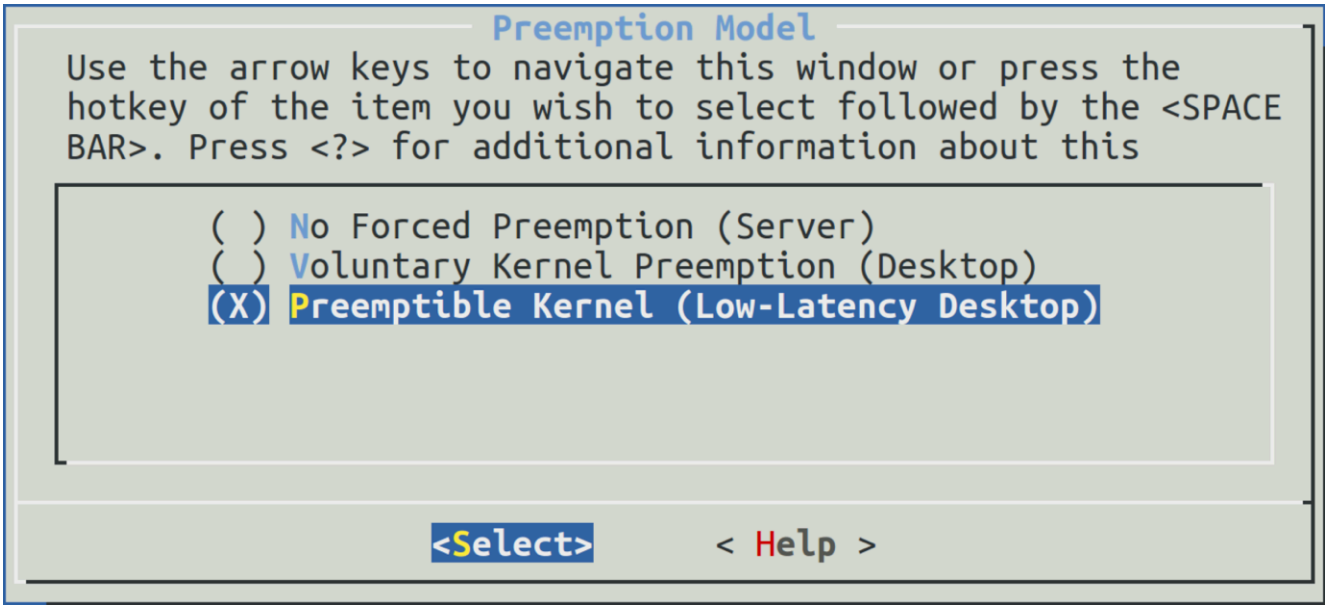
Understanding Linux Kernel Preemption

While configuring a Linux kernel, we can set some parameters that effect the system behavior. You can work with different priorities, scheduling classes and preemption models. It is very important to understand and choose the right parameters.

In this post I will cover the different preemption models and how does each one effect the user and kernel behaviour

If you configure the kernel (using make menuconfig) , you can find, usually in kernel features sub menu the option – Preemption model:



To understand each option lets take an example:

- We have 2 threads – one with high real time

priority(50) and the other with low RT priority(30)
- The high priority thread went to sleep for 3 seconds
- The low priority threads uses the CPU for user space calculations
- After 3 seconds the high priority thread will wake up

This case is easy and reasonable but what happens if the low priority thread call a kernel code while the high priority is sleeping? It is depends on the above configuration

No Forced Preemption

The context switch is done only when we return from the kernel. Lets take an example :

- We have 2 threads – one with high real time priority(50) and the other with low RT priority(30)
- The high priority thread went to sleep for 3 seconds
- The low priority threads calls a kernel code that last for 5 seconds
- After 5 seconds the low priority thread returns from the kernel
- The high priority thread will wake up (2 seconds late)

Lets see the code:

Kernel code – simple character device driver:

```
1
2
```

```
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
```

```
#include <asm/uaccess.h>
#include <linux/fs.h>
#include <linux/gfp.h>
#include <linux/cdev.h>
#include <linux/sched.h>
#include <linux/kdev_t.h>
#include <linux/delay.h>
#include <linux/ioctl.h>
#include <linux/slab.h>
#include <linux/mempool.h>
#include <linux/mm.h>
#include <asm/io.h>
static dev_t my_dev;
static struct cdev *my_cdev;
```

```
25 // callback for read system call on the device
26 static ssize_t my_read(struct file *file, char __user
   *buf,size_t count,loff_t *ppos)
27 {
28   int len=5;
29   if(*ppos > 0)
30   {
31 return 0;
32   }
33   mdelay(5000); // busy-wait for 5 seconds
34   if (copy_to_user(buf , "hello" , len)) {
35     return -EFAULT;
36   } else {
37      *ppos +=len;
38      return len;
39   }
40 }
41 static struct file_operations my_fops =
42 {
43 .owner = THIS_MODULE,
44 .read = my_read,
45 };
46
```

```
47  static int hello_init (void)
48  {
49  my_dev = MKDEV(400,0);
50  register_chrdev_region(my_dev,1,"demo");
51  my_cdev=cdev_alloc();
52  if(!my_cdev)
53  {
54  printk (KERN_INFO "cdev alloc error.\n");
55  return -1;
56  }
57  my_cdev->ops = &my_fops;
58  my_cdev->owner = THIS_MODULE;
59  if(cdev_add(my_cdev,my_dev,1))
60  {
61  printk (KERN_INFO "cdev add error.\n");
62  return -1;
63  }
64    return 0;
65  }
66  static void
67  hello_cleanup (void)
68  {
```

```
69 cdev_del(my_cdev);
70 unregister_chrdev_region(my_dev, 1);
71 }
72 module_init (hello_init);
73 module_exit (hello_cleanup);
74 MODULE_LICENSE("GPL");
75
76
77
78
79
80
81
82
83
84
```

The read is delaying for 5 seconds (delay is a busy wait loop) and returns some data

The User space code:

```
1
2
```

```
3
4
5
6
7  #include<stdio.h>
8  #include<unistd.h>
9  #include<pthread.h>
10 #include <sys/types.h>
11 #include <sys/stat.h>
12 #include <fcntl.h>
13 void *hi_prio(void *p)
14 {
15 printf("thread1 start time=%ld\n",time(NULL));
16 sleep(3);
17 printf("thread1 stop time=%ld\n",time(NULL));
18 return NULL;
19 }
20 void *low_prio(void *p)
21 {
22 char buf[20];
23 sleep(1);
24 int fd=open("/dev/demo",O_RDWR);  // #mknod
```

```
25  /dev/demo c 400 0
26  puts("thread2 start");
27  read(fd,buf,20);
28  puts("thread2 stop");
29  return NULL;
30  }
31  int main()
32  {
33  pthread_t t1,t2,t3;
34  pthread_attr_t attr;
35  struct sched_param param;
36
37  pthread_attr_init(&attr);
38  pthread_attr_setschedpolicy(&attr, SCHED_RR);
39  param.sched_priority = 50;
40  pthread_attr_setschedparam(&attr, &param);
41  pthread_create(&t1,&attr,hi_prio,NULL);
42  param.sched_priority = 30;
43  pthread_attr_setschedparam(&attr, &param);
44  pthread_create(&t2,&attr,low_prio,NULL);
45  sleep(10);
46  puts("end test");
```

```
47  return 0;
48  }
49
50
51
52
53
```

- The high priority goes to sleep for 3 seconds.
- The low priority thread is sleeping for one second and then calls the kernel
- The high priority is wake after 6 seconds:

```
1  # insmod demo.ko
2  # ./app
3  thread1 start time=182
4  thread2 start
5  thread1 stop time=188
6  thread2 stop
7  end test
```

Preemptible Kernel

In this configuration the context switch is done on time

also in the kernel, means if we run the above test we will see the high priority thread waking up after 3 seconds:

It means that in this options the system will perform more context switches per second but it is more “real time”. On embedded systems with soft real time requirements it is a best practice to use this option but in a server system that we are usually work asynchronously the first option is better – less context switches – more cpu time

The output:

```
1  # insmod ./demo.ko
2  #./app
3  thread1 start time=234
4  thread2 start
5  thread1 stop time=237
6  thread2 stop
7  end test
```

Voluntary Kernel Preemption

In this configuration the system is working like “no forced preemption” but if the kernel developer is writing a complex code it is responsible to check from time to time if a re scheduling is needed. He can do that with might_resched() function

So in this example, if we want to add this “check point” we

will change the code:

```
1  // callback for read system call on the device
2  static ssize_t my_read(struct file *file, char __user
   *buf,size_t count,loff_t *ppos)
3  {
4    int len=5;
5    if(*ppos > 0)
6    {
7  return 0;
8    }
9    mdelay(4000); // busy-wait for 4 seconds
10   might_resched();
11   delay(3000);  // busy wait for 3 seconds
12   if (copy_to_user(buf , "hello" , len)) {
13      return -EFAULT;
14   } else {
15       *ppos +=len;
16       return len;
17   }
18 }
```

If we comment out the line might_resched() it will be delayed for 7 seconds total, adding a cond_resched call

will check and perform the context switch if other hi priority threads is awake. It will be called after 5 seconds (1 second before the call and 4 seconds in the kernel)

Output:

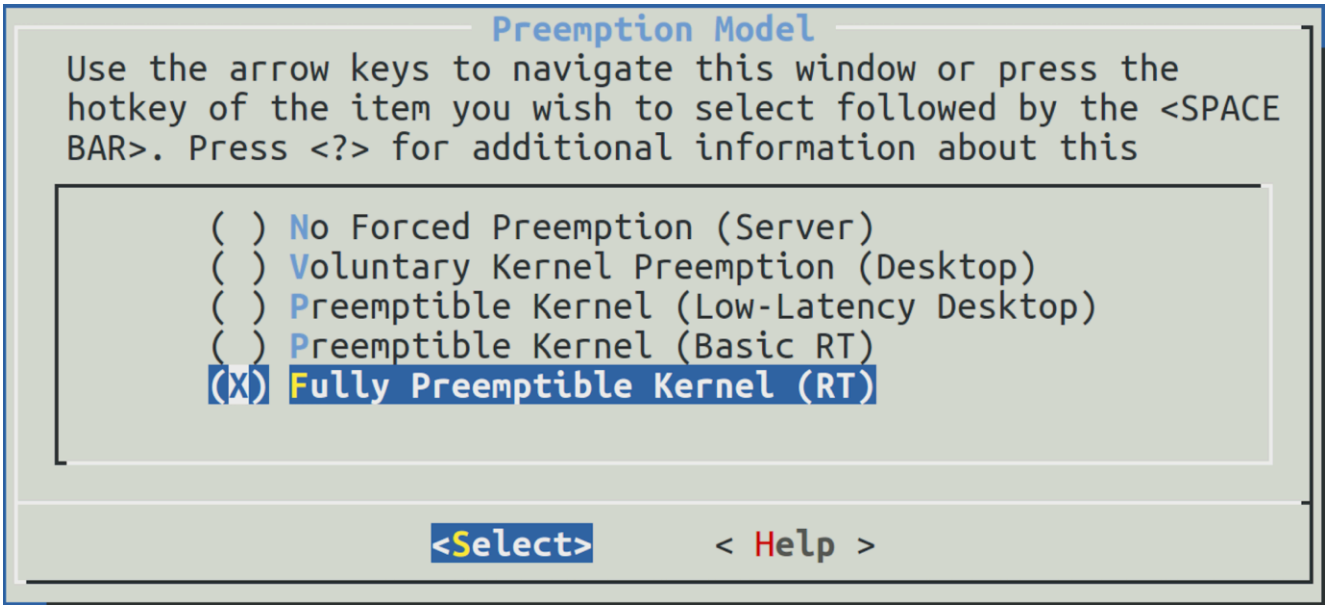
```
1  # insmod ./demo.ko
2  #./app
3  thread1 start time=320
4  thread2 start
5  thread1 stop time=325
6  thread2 stop
7  end test
```

Full Real Time Preemption

If you apply the RT patch , You get an Hard realtime kernel. This means any code can block other , if you run an interrupt service routine code and something more urgent need to be handled it will block the ISR code. The patch changes the following:

- Converting hardware Interrupts to threads with RT priority 50
- Converting SoftIRQs to threads with RT 49 priority
- Converting all spinlocks to mutexes
- Configuring and using Hi resolution timers
- some more minor features

After applying the patch you can see 2 more options in the menu:



The option “Preemptible Kernel (Basic RT)” is for debugging (see documentation)

To make all the above changes you need to select the last option – Fully Preemptible Kernel.

Now if you create a thread with RT priority bigger than 50 – it will block interrupts

Note that in this configuration the system has more tasks and performs more context switches per second. i.e. the CPU is spending more time switching tasks but we can hit any deadline required (1ms or more)