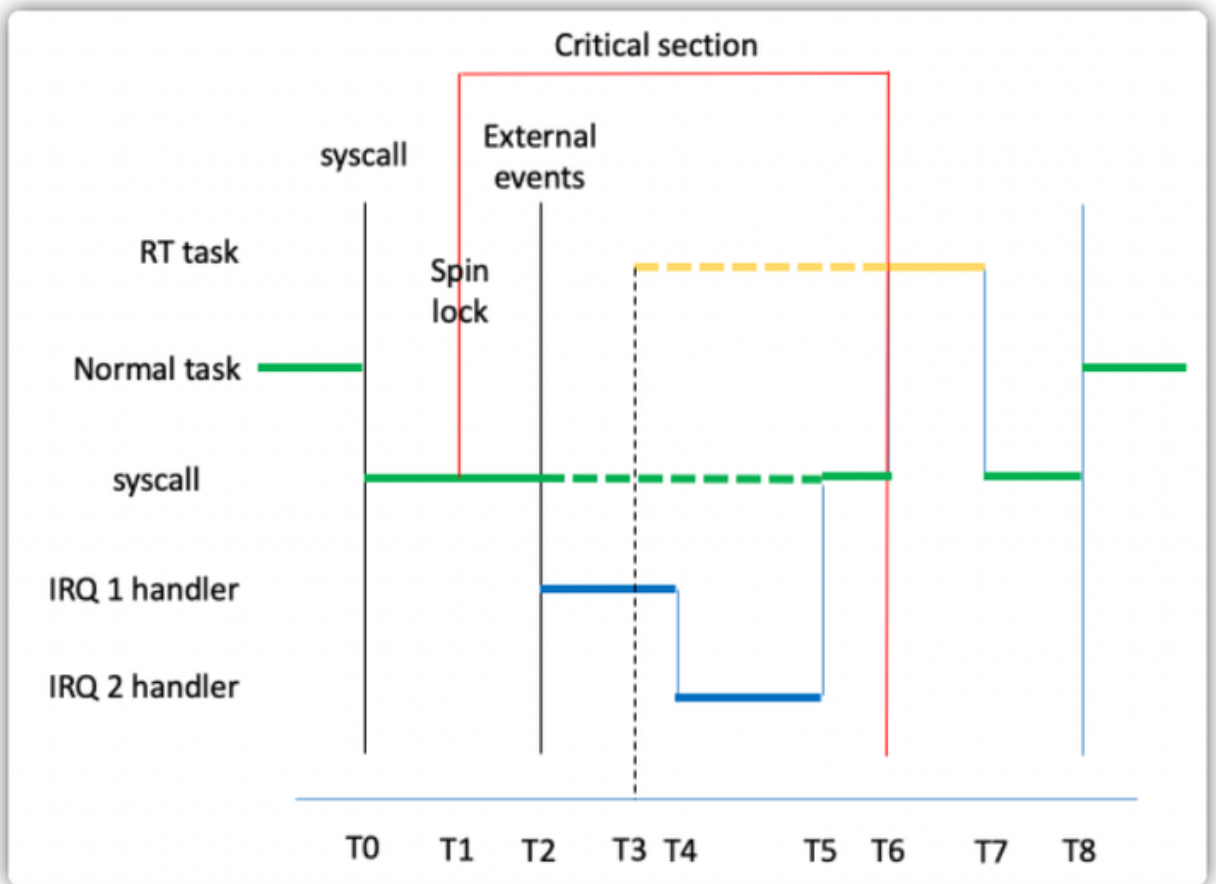


# RT-Patch理论学习

## 1. 传统Linux

标准的Linux并不是硬时操作系统，因为以下四类区间中只有一类支持抢占调度

- 中断 x
- 软中断 x
- 进程上下文的spinlock x
- 进程上下文的其他区域，只有这里支持抢占调度！ ✓



- T3时刻系统唤醒了高优先级的RT task，但由于此时系统处于不可调度区域，所以RT task无法立即运行
  - T5时刻，中断都结束，但spin lock仍然没有释放，系统仍然处于不可调度区域
  - T6时刻，spin lock释放，高优先级的RT task立马得到调度
- 从T1到T6，这个区间的时间是不可预测的，所以通用的Linux系统无法达到硬实时的标准。

## 2. RT改进

两种思路：

1. **双内核法**：通过在Linux内核与硬件中断之间增加一个可抢先的**实时内核**，把标准的Linux内核作为该实时内核的一个优先级。它可以被实时进程抢断，正常的Linux进程仍可以在Linux内核上运行，这样既可以使用标准分时操作系统即Linux的各种服务，又能提供低延时的实时环境。**RT-Linux**是采用双内核法改造Linux实时性的典型代表。
2. **修改内核代码**：通过打补丁的方式，对内核的进程调度、中断服务程序等代码进行修改与优化，提高系统的实时性能，最出名的就是Preempt-RT补丁

### Abstract

原理：

最小化内核中不可抢占部分的代码，同时将为支持抢占性必须要修改的代码量最小化。对**临界区**、**中断处理函数**等代码进行抢占改进

## 2.1临界区抢占

重新实现rt\_mutex使内核里的spinlock可被抢占。以前被spinlock\_t和rwlock\_t保护的临界区现在变得可以被抢占：

Spinlock是Linux内核锁机制的一种，被大量用于访问临界区。当一个进程持有自旋锁，另外一个进程想获得这个自旋锁时，CPU就在不停打转盲等状态。但传统的spinlock是禁止抢占的，因此会影响系统的实时性。

RT-patch Linux 用支持PI的rt\_mutex代替传统的禁用抢占的spin\_lock。

```
/**
 * rt_mutex_lock - lock a rt_mutex
 *
 * @lock: the rt_mutex to be locked
 */
void __sched rt_mutex_lock(struct rt_mutex *lock)
{
    __rt_mutex_lock(lock, 0);
}
```

**Warning**

注意要结合[优先级继承](#)使用

## 2.2 中断线程化：

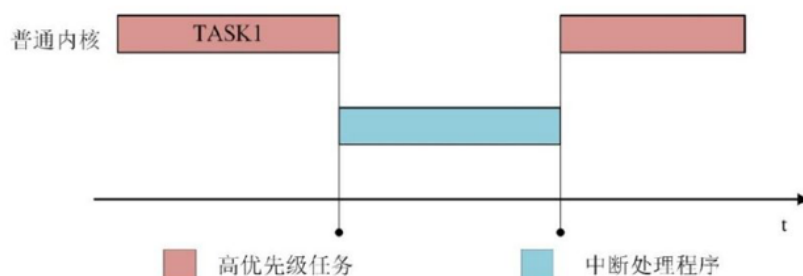
在内核线程上下文中处理中断：

线程化中断也是RT-patch Linux 使Linux实时化的主要工作。

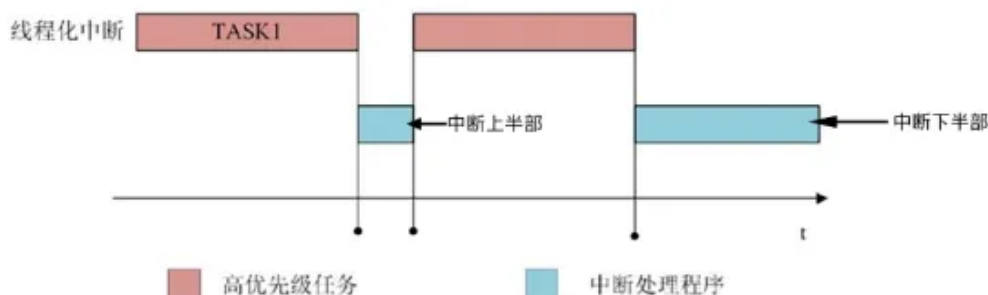
使用中断处理线程来处理中断，使得中断处理程序可以在普通内核线程中运行，因为可以为其配合优先级。

中断处理函数可分为上半部和下半部。

其上半部依然做最小的工作，就是处理紧急的事情比如对硬件、寄存器的修改，但仍然是禁止抢占并且禁止中断的，因为**RAM**处理器会在中断处理期间默认关闭**CPU**中断响应来避免中断欠通。下半部则转化为内核线程用来处理中断。



TASK1是一个高优先级的任务。在普通内核中，高优先级任务TASK1就会因为被中断而被挂起，直到中断处理程序结束，TASK1才能得以继续运行。



中断线程化后的内核中，发生中断后中断上半部依然正常执行，但由于其所做工作十分少，所以并不占用太多时间，中断下半部的工作由内核线程来执行，由于其优先级比TASK1的优先级低，故中断上半部结束后调度器选择TASK1继续执行，TASK1结束之后，中断处理程序才得以执行。

## Summary

通过这种方法，高优先级任务受中断延迟的影响便会减少。在实现中断线程化之后，几乎所有内核空间变为可抢占。其中，中断可以发生在任何时刻；被中断唤醒的进程也会得到立刻执行。

## 2.3. 优先级继承

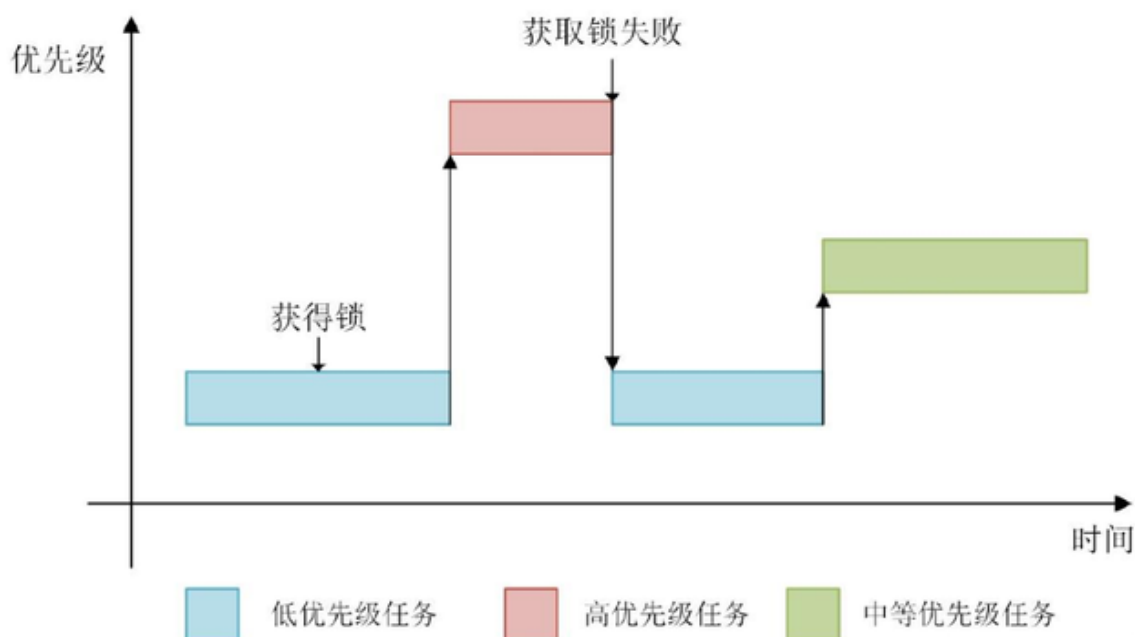
优先级翻转是实时系统中必须解决的问题，它可能导致一个高优先级任务被无限期推迟执行。

优先级翻转的情况：

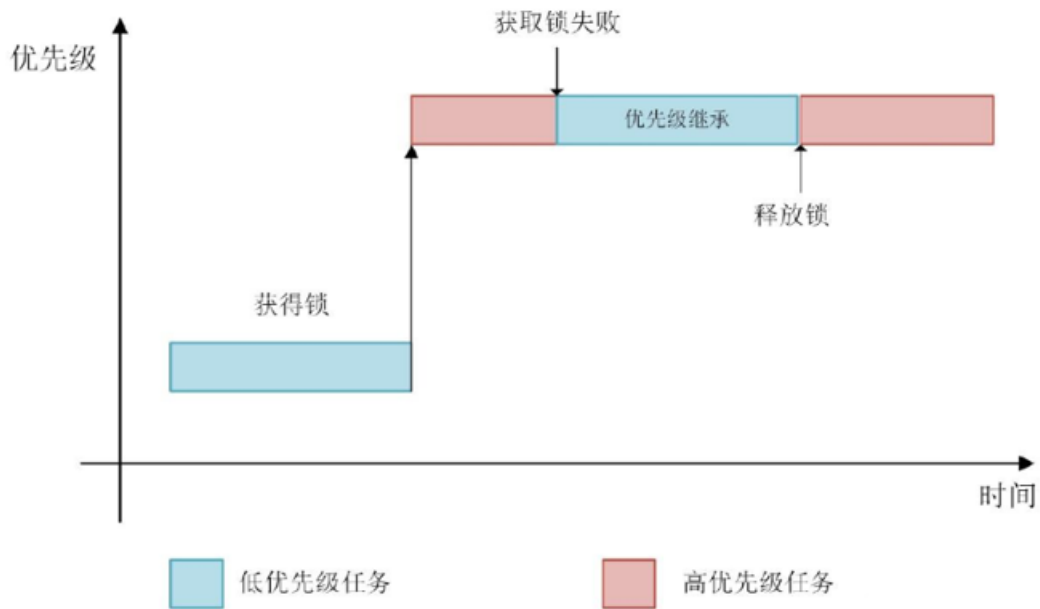
- 低优先级的任务A获取一个资源，比如一个锁（L）
- 中优先级任务B开始执行，抢占了任务A
- 高优先级任务C尝试获取资源L，因为中优先级的任务B抢占了任务A，（任务A无法释放锁L）那么高优先级的任务会阻塞

没有优先级继承时：

优先级继承



有优先级继承：



高优先级抢占低优先级任务之后，由于无法获得互斥资源便进入等待状态。原本的低优先级进程继承了此时高优先级进程的优先级后继续执行，当该进程释放了锁之后，就被高优先级进程抢占。

### 3. 打入RT-patch方法

SHELL

```
gnuzip patch-xxx-rt-xx
patch -p1 < patch-xxx-rt-xx
```

编译内核的时候在menuconfig中的Preemption Model选择Fully Preemptible Kernel

### 4. 测试方法

可以使用cyclicttest工具进行测试，如果要上压力就用stress工具

### 5. 总结

- 抢占调度区域：

1. 中断
2. 软中断
3. 进程上下文的spinlock
4. 进程上下文的其他区域

一般server仅在 **4. 进程上下文的其他区域** 支持抢占

- 优化方向

1. 中断线程化（内核线程）
  2. spinlock临界区抢占（拥锁进程优先级继承）
  3. 硬件（如Cortex R系列）
-