Linux 内核实战课（同步篇）：信号量 、互斥锁 已付费

原创 内核实战 人人极客社区 2023-07-05 08:03 发表于江苏

收录于合集
#Linux内核实战课 27个 ›

前言

最全最详细的Linux内核实战教程，把碎片化的内核知识系统地串联起来；邀请了华为、OPPO和字节跳动的大佬，和大佬们一起讨论实战；可免费进知识星球。

人人极客社区
工程师们自己的Linux底层技术社区，分享体系架构、内核、网络、安全和驱动。
289篇原创内容
公众号

信号量 Semaphore

信号量是这样一种同步机制：信号量在创建时设置一个初始值count，用于表示当前可用的资源数。一个任务要想访问共享资源，首先必须得到信号量，获取信号量的操作为count-1，若当前count为负数，表明无法获得信号量，该任务必须挂起在该信号量的等待队列等待；若当前count为非负数，表示可获得信号量，因而可立刻访问被该信号量保护的共享资源。当任务访问完被信号量保护的共享资源后，必须释放信号量，释放信号量通过把count+1实现，如果count为非正数，表明有任务等待，它也唤醒所有等待该信号量的任务。

信号量是在多线程环境下使用的一种措施，它负责协调各个进程，以保证他们能够正确、合理的使用公共资源。它和 spin_lock 最大的不同之处就是：无法获取信号量的进程可以睡眠，因此会导致系统调度。

信号量的定义如下：

```
struct semaphore {
 raw_spinlock_t  lock;       //利用自旋锁同步
 unsigned int  count;       //用于资源计数
 struct list_head wait_list; //等待队列
};
```

信号量在创建时设置一个初始值 count，用于表示当前可用的资源数。一个任务要想访问共享资源，首先必须得到信号量，获取信号量的操作为 count - 1。若当前 count 为负数，表明无法获得信号量，该任务必须挂起在该信号量的等待队列等待；若当前 count 为非负数，表示可获得信号量，因而可立刻访问被该信号量保护的共享资源。

当任务访问完被信号量保护的共享资源后，必须释放信号量，释放信号量是操作 count + 1，如果加一后的 count 为非正数，表明有任务等待，则唤醒所有等待该信号量的任务。

了解了信号量的结构与定义，接下来我们看下常用的信号量接口：

API	说明
DEFINE_SEMAPHORE(name)	声明信号量并初始化为 1
void sema_init(struct semaphore *sem, int val)	声明信号量并初始化为 val
down	获得信号量，task 不可被中断，除非是致命信号
down_interruptible	获得信号量，task 可被中断
down_trylock	能够获得信号量时，count --，否则立刻返回，不加入 waitlist
down_killable	获得信号量，task 可被 kill
up	释放信号量

这里我们看下最核心的两个实现 **down** 和 **up**。

- down

down 用于调用者获得信号量，若 count 大于0，说明资源可用，将其减一即可。

```
void down(struct semaphore *sem)
{
 unsigned long flags;

 raw_spin_lock_irqsave(&sem->lock, flags);
 if (likely(sem->count > 0))
  sem->count--;
 else
  __down(sem);
 raw_spin_unlock_irqrestore(&sem->lock, flags);
}
EXPORT_SYMBOL(down);
```

若 count < 0，调用函数 __down()，将 task 加入等待队列，并进入等待队列，并进入调度循环等待，直至其被 __up 唤醒，或者因超时以被移除等待队列。

```
static inline int __sched __down_common(struct semaphore *sem, long state,
        long timeout)
{
 struct semaphore_waiter waiter;

 list_add_tail(&waiter.list, &sem->wait_list);
 waiter.task = current;
 waiter.up = false;

 for (;;) {
  if (signal_pending_state(state, current))
   goto interrupted;
  if (unlikely(timeout <= 0))
   goto timed_out;
  __set_current_state(state);
  raw_spin_unlock_irq(&sem->lock);
  timeout = schedule_timeout(timeout);
  raw_spin_lock_irq(&sem->lock);
  if (waiter.up)
   return 0;
 }

 timed_out:
 list_del(&waiter.list);
 return -ETIME;

 interrupted:
 list_del(&waiter.list);
 return -EINTR;
}
```

- up

up 用于调用者释放信号量，若 waitlist 为空，说明无等待任务，count + 1，该信号量可用。

```
void up(struct semaphore *sem)
{
 unsigned long flags;

 raw_spin_lock_irqsave(&sem->lock, flags);
 if (likely(list_empty(&sem->wait_list)))
  sem->count++;
 else
  __up(sem);
 raw_spin_unlock_irqrestore(&sem->lock, flags);
}
EXPORT_SYMBOL(up);
```

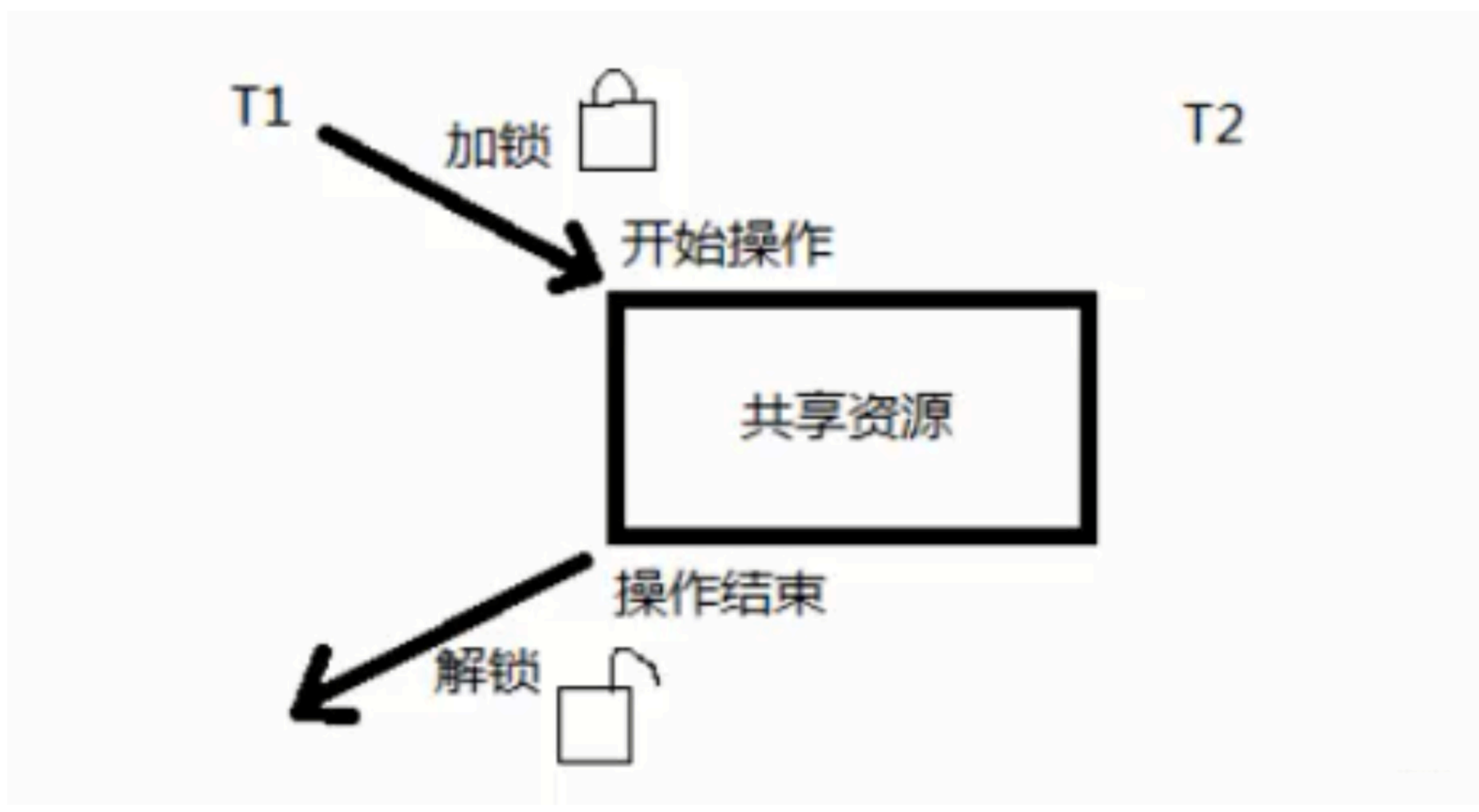
若 waitlist 非空，将 task 从等待队列移除，并唤醒该 task，对应 __down 条件。

```
static noinline void __sched __up(struct semaphore *sem)
{
 struct semaphore_waiter *waiter = list_first_entry(&sem->wait_list,
      struct semaphore_waiter, list);
 list_del(&waiter->list);
 waiter->up = true;
 wake_up_process(waiter->task);
}
```

互斥锁 mutex

Linux 内核中，还有一种类似信号量的同步机制叫做互斥锁。互斥锁类似于 count 等于 1 的信号量。所以说信号量是在多个进程/线程访问某个公共资源的时候，进行保护的一种机制。而互斥锁是单个进程/线程访问某个公共资源的一种保护，于互斥操作。

互斥锁有一个特殊的地方：只有持锁者才能解锁。如下图所示：



用一句话来讲信号量和互斥锁的区别，就是**信号量用于线程的同步，互斥锁用于线程的互斥。**

互斥锁的结构体定义：

```
struct mutex {
 atomic_long_t  owner; //互斥锁的持有者
 spinlock_t  wait_lock; //利用自旋锁同步
#ifdef CONFIG_MUTEX_SPIN_ON_OWNER
 struct optimistic_spin_queue osq; /* Spinner MCS lock */
#endif
 struct list_head wait_list; //等待队列
......
};
```

其常用的接口如下所示：

API	说明
DEFINE_MUTEX(name)	静态声明互斥锁并初始化解锁状态
mutex_init(mutex)	动态声明互斥锁并初始化解锁状态
void mutex_destroy(struct mutex *lock)	销毁该互斥锁
bool mutex_is_locked(struct mutex *lock)	判断互斥锁是否被锁住
mutex_lock	获得锁，task 不可被中断
mutex_unlock	解锁
mutex_trylock	尝试获得锁，不能加锁则立刻返回
mutex_lock_interruptible	获得锁，task 可以被中断
mutex_lock_killable	获得锁，task 可以被中断
mutex_lock_io	获得锁，在该 task 等待琐时，它会被调度器标记为 io 等待状态

上面讲的自旋锁，信号量和互斥锁的实现，都是使用了原子操作指令。由于原子操作会 lock，当线程在多个 CPU 上争抢进入临界区的时候，都会操作那个在多个 CPU 之间共享的数据 lock。CPU 0 操作了 lock，为了数据的一致性，CPU 0 的操作会导致其他 CPU 的 L1 中的 lock 变成 invalid，在随后的来自其他 CPU 对 lock 的访问会导致 L1 cache miss（更准确的说是communication cache miss），必须从下一个 level 的 cache 中获取。

这就会使缓存一致性变得很糟，导致性能下降。所以内核提供一种新的同步方式：RCU（读-复制-更新）。