東北大學秦皇島分校
Northeastern University at Qinhuangdao

Linux系统与内核分析

--内核中的同步

于七龙

目录

Part 1 临界区和竞争状态

临界区

- 内核中的各个任务并不是要严格按顺序依次执行，而是相互交错执行，称为并发
- 内核中的许多数据都是共享资源，对这些共享资源的访问必须遵循一定的访问规则，否则可能造成对共享资源的破坏
- 临界区是访问和操作共享数据的代码段

同步

- 为了避免对临界区并发访问，编程者必须保证临界区代码被原子的执行
  - 临界区代码在执行期间不可被打断，如同整个临界区是一个不可分割的指令
- 两个内核任务处于同一个临界区的错误现象称为竞争状态
- 内核通过同步机制防止多个内核任务访问或操作临界区

临界区示例

- ”i++“操作可以转化成三条机器指令序列
  - 得到当前变量i的值并拷贝到寄存器eax中
  - 将寄存器中的值加: addl $1，%eax
  - 把i的新值写回到内存中

✓

内核任务1	内核任务2
获得i(1)	---
增加 i(1->2)	---
写回 i(2)	---
	获得 i(2)
	增加 i(2->3)
	写回 i(3)

✗

内核任务1	内核任务2
获得 i(1)	---
---	获得 i(1)
增加 i(1->2)	---
---	增加 i(1->2)
写回 i(2)	---
---	写回 i(2)

临界区示例

- 对于简单的临界区访问，可通过将操作指令作为不可分割的整体执行即可
- 多数处理器都提供指令原子的读变量、增加变量再写回变量
  - 硬件相关

锁机制

- 锁机制可避免竞争状态
- 任何要访问共享资源的代码首先需占住相应的锁，这样该锁就能阻止来自其它内核任务的并发访问

任务 1

试图锁定队列

成功：获得锁

访问队列...

为队列解除锁

任务2

试图锁定队列

失败：等待...

等待...

等待...

成功：获得锁

访问队列...

为队列解除锁

锁机制

- Linux内核中实现了多种不同的锁机制
- 不同锁机制间的区别主要在于当锁被持有时的行为表现
  - 如一些锁被持有时会不断进行循环查询，等待锁重新可用
  - 一些锁会使当前任务睡眠，直到锁可用为止

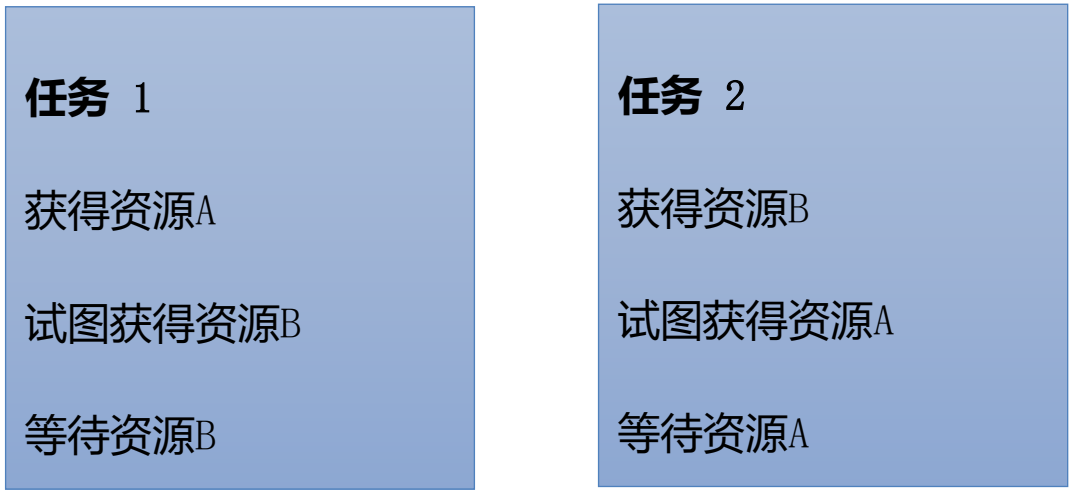
确定保护对象

- 找出哪些数据需要保护是关键所在
  - 内核任务的局部数据仅仅被它本身访问，不需要保护
  - 如果数据只会被特定的进程访问，不需加锁
- 大多数内核数据结构都需要加锁
  - 若有其它内核任务可以访问这些数据，那么就给这些数据加上某种形式的锁
  - 若任何其它任务能看到它，那么就要锁住它

死锁

死锁条件

- 一个或多个并发执行的内核任务和一个或多个资源，其中每个任务都在等待其中的一个资源，但所有的资源都已经被占用，以至所有任务都在相互等待，而所有任务都不主动释放已经占有的资源



死锁

死锁的避免

- 加锁的顺序是关键。使用嵌套的锁时必须保证以相同的顺序获取锁，这样可以阻止致命拥抱类型的死锁
- 防止发生饥饿，设置time_out时间
- 不重复请求同一个锁
- 越复杂的加锁方案越有可能造成死锁，因此设计应力求简单

并发

并发执行的原因

- 中断：中断几乎可以在任何时刻异步发生，也可能随时打断正在执行的代码
- 内核抢占：若内核具有抢占性，内核中的任务就可能会被另一任务抢占
- 睡眠：在内核执行的进程可能会睡眠，这将唤醒调度程序，导致调度一个新的用户进程执行
- 对称多处理：两个或多个处理器可以同时执行代码

并发

- 内核开发者必须理解并发执行的诱因
- 内核开发常见潜在错误
  - 一段内核代码访问某资源时系统产生一个中断，而该中断的处理程序也需要访问该资源
  - 一段内核代码在访问一个共享资源期间可以被抢占
  - 内核代码在临界区睡眠
  - 两个处理器同时访问同一共享数据

Part 2 内核同步措施

Linux同步机制发展

- Linux提供了一组同步方法提供对共享数据的保护，Linux的同步机制随着内核版本的发展而不断完善
- Linux基本同步措施
  - 原子变量
  - 自旋锁
  - 信号量
  - 读/写者锁
  - RCU

原子变量

- 原子变量用于实现对整数的互斥访问，通常用来实现计数器
- Linux内核提供专门的数据类型（原子访问计数器）

```
typedef struct {
        int counter;
}atomic_t;
```

本定义方法可让GCC在编译的时候加以更加严格的类型检查，防止原子变量被误操作

原子变量操作方法

- Linux内核提供了专门的函数和宏，用于原子类型数据操作

操作	效果
ATOMIC_INT(i)	声明一个atomic_t变量，并初始化为i
atomic_read(atomic_t *v)	读取原子变量的值
atomic_set(atomic_t *v, int i)	将v设置为i
…	…

```
atomic_t  u;                //定义u
atomic_t v = ATOMIC_INNT(0)     //定义v并初始化为0
atomic_set(&v, 4)           //v=4
atomic_add (2, &v)          //v = v + 2 = 6
atomic_inc(&v, 4)           //v = v + 1 = 7
```

自旋锁

- 自旋锁是专为防止多处理器并行而引入的一种锁，自旋锁在内核中大量应用于中断处理等部分，对于单处理器来说，可简单采用关闭中断的方式防止中断处理程序的并发执行

自旋锁的定义

```
Typedef  struct raw_spinlock { unsigned int slock;} raw_spinlock_t;
typedef struct {
        raw_spinlock_t  raw_lock;
              ...
} spinlock_t;
```

自旋锁的使用

```
spinlock_t  lock = SPIN_LOCK_UNLOCKED;
……
spin_lock(&lock);

      /*临界区*/

spin_unlock(&lock);
```

初始化自旋锁时，必须将锁置为未锁定状态

spin_lock为一个原子操作

对锁操作必须成对出现（锁与开锁）！

自旋锁特性

- 自旋锁在内核中主要用于防止多处理器并行访问临界区
- 一个被持有的自旋锁使得请求它的任务在等待重新可用期间进行自旋，自旋操作特别浪费处理器时间，所以**自旋锁不应该被长时间持有**，如果需长时间持有，则一般使用信号量
- 自旋锁只用于保护短的代码段
- 如果用户获得锁之后不释放，系统将变得不可用
- 由自旋锁保护的代码不能进入睡眠状态，否则引起自死锁
- 自旋锁保护代码调用的函数也不能进入睡眠状态
- 开关锁操作成对出现

信号量

- Linux中的信号量是一种睡眠锁。若有一个任务试图获得一个已被持有的信号量时，信号量会将其推入等待队列，然后让其睡眠

- 当持有信号量的进程将信号量释放后，在等待队列中的一个任务将被唤醒

信号量

- 信号量支持两个原子操作P()和V()，对应Linux中down()和up()操作
  - down()操作通过对信号量计数减1来请求获得一个信号量，如果信号量结果是0或大于0，信号量锁被获得，任务就可进入临界区；如果结果是负数，任务将别放入等待队列，处理器执行其他任务
  - up()操作通过对信号量计数加1来释放信号量

信号量

- 信号量的睡眠特性使其只适用于锁会被长时间持有的情况，因此只能在进程上下文中使用，而不能在中断上下文中使用，因为中断上下文不能调度

- 当任务持有信号量时，不可再持有自旋锁

信号量的定义

```
struct semaphore {

        spinlock_t      lock;

        unsigned int    count;

        struct list_head  wait_list;

}
```

lock:自旋锁，Linux内核高版本中加入，为防止多处理器并行造成错误

count：count>0，表示资源空闲；count=0,表示信号量忙，但没有进程等待资源； count<0，资源不可用，且已有进程在等待

wait_list：存放等待队列链表的地址，如果count>0，则链表为空

信号量的使用

- 信号量的声明与初始化

```
DECLARE_MUTEX(name);                    //定义互斥信号量

sema_init(struct semaphore *sem, int val);      //初始化信号量sem的使用者数量为val

init_MUTEX(sem);                //初始化信号量sem为未锁定的互斥信号量

init_MUTEX_LOCKED(sem);                 //初始化信号量sem为锁定的互斥信号量
```

信号量的使用

- 使用信号量的一般形式

```
static DECLARE_MUTEX(mr_sem);              //声明并初始化互斥信号量

if(!down_interruptible(&mr_sem))           //信号被接收，信号量还未获取

    /*临界区…*/

up(&mr_sem);
```

信号量的操作也需成对出现

信号量的操作函数

- 使用信号量常见操作函数

函数	描述
down(struct semaphore *);	获取信号量，如果不可获取，则进入不可中断睡眠状态（不建议使用）
down_interruptible(struct semaphore *);	获取信号量，如果不可获取，则进入可中断睡眠状态
down_killable(struct semaphore *);	获取信号量，如果不可获取，则进入可被致命信号中断的睡眠状态
down_trylock(struct semaphore *);	尝试获取信号量，如果不能获取，则立刻返回
down_timeout(struct semaphore *, long jiffies);	在给定时间（jiffies）内获取信号量，如果不能够获取，则返回
up(struct semaphore *);	释放信号量

信号量的操作函数

- 各种down()操作的比较

```
static noinline void __sched __down(struct semaphore *sem)
{
    __down_common(sem, TASK_UNINTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
}

static noinline int __sched __down_interruptible(struct semaphore *sem)
{
    return __down_common(sem, TASK_INTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
}

static noinline int __sched __down_killable(struct semaphore *sem)
{
    return __down_common(sem, TASK_KILLABLE, MAX_SCHEDULE_TIMEOUT);
}

static noinline int __sched __down_timeout(struct semaphore *sem, long timeout)
{
    return __down_common(sem, TASK_UNINTERRUPTIBLE, timeout);
}
```

- 不同down()操作对比

信号量的操作函数

- 获取信号量操作：以down_interruptible()为例（ kernel/locking/semaphore.c ）

```
int down_interruptible(struct semaphore *sem)
{
    unsigned long flags;
    int result = 0;

    raw_spin_lock_irqsave(&sem->lock, flags);
    if (likely(sem->count > 0))
        sem->count--;
    else
        result = __down_interruptible(sem);
    raw_spin_unlock_irqrestore(&sem->lock, flags);

    return result;
}
```

加锁，使信号量的操作在关中断状态下进行，防止多处理器并发操作造成错误

如果信号量可用，则计数器值减去1

如果信号量无可用，则进入睡眠状态

对信号量解锁

- down_interruptible()函数

信号量的操作函数

- __down_interruptible()函数

```
static noinline int __sched __down_interruptible(struct semaphore *sem)
{
    return __down_common(sem, TASK_INTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
}
```

- __down_interruptible()函数

信号量的操作函数

- __down_common()函数

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

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

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

将当前进程添加到信号量sem的等待队列的队尾

如果当前进程被信号唤醒，则返回

如果超时，则返回

设置进程状态

释放自旋锁

进入延时唤醒状态

当进程被唤醒时，如果再次获取信号量操作，则对其进行加锁

- __down_common ()函数

信号量的操作函数

- 释放信号量操作：up()函数

```
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);
}
```

对信号量操作进行加锁

判断该信号量的等待队列为空

唤醒该信号量的等待队列队头的进程

释放信号量

- up()函数

信号量与自旋锁对比

- 一般情况下，只用考虑是否在中断上下文中，以及任务持有锁是否需要睡眠

需求	建议
低开销加锁	优先使用自旋锁
短期锁定	优先使用自旋锁
长期加锁	优先使用信号量
中断上下文中加锁	只能使用自旋锁
持有锁时需要睡眠、调度	只能使用信号量

读写者锁

- 前述同步机制未区分数据结构的读写访问
- 读/写者锁定义为rwlock_t数据类型
- 进程对临界区进行读操作时，进入和离开需要分别执行read_lock和read_unlock
- 进程对临界区进行写操作时，进入和离开是需要分别执行write_lock和write_unlock

读写者锁

- 读写者锁之一是读写自旋锁

```
DEFINE_RWLOCK(my_lock);
read_lock(&my_lock);

    /*只读临界区*/

read_unlock(&my_lock);
```

```
write_lock(&my_lock);

    /*写临界区*/

write_unlock(&my_lock);
```

读写者锁

- 读写者锁用于对临界区的访问从逻辑上可以清晰的分为读和写两种模式的情况
- 任意数目的处理器都可以对数据结构进行**并发读访问**，但只有一个处理器能进行写访问，**在写访问的同时，读访问无法进行**

RCU

- RCU（Read-Copy Update，读-复制更新）是一种新型读写机制
  - 写者修改对象时，首先复制生成一个副本，然后更新此副本，最后使用新的对象替换旧的对象，在写者执行复制更新的时候读者可以读数据
  - 写者删除对象，必须等到所有访问被删除对象的读者访问结束，才能执行销毁操作
- 总之，RCU允许多个读者同时读，且同时允许一个写者

RCU

- RCU的优点是读者没有任何同步开销，不需要获取任何锁，不需要执行原子指令，不需要执行内存屏障。但写者的同步开销比较大，写者需要延迟对象的释放，复制被修改的对象，写者之间必须使用锁互斥

RCU

RCU主要API

- rcu_read_lock() / rcu_read_unlock()：组成RCU读临界
- rcu_dereference()：获取被RCU保护的指针，用于读者线程
- synchronize_rcu()：同步等待所有现存的读访问完成
- call_rcu()：注册一个回调函数，当所有现存读访问完成后，使用该回调函数销毁数据
- rcu_assign_pointer()：常用在写线程，用于发布更新后的数据，即在写者线程完成新数据修改后，调用该接口让RCU保护的 指针指向新建的数据

RCU

- RCU读者线程示例

```
#Document/RCU/whatisRCU.txt
int foo_get_a(void)    {
        int retval;
        rcu_read_lock();
        retval = rcu_dereference(gbl_foo)->a;  //获取被保护数据gbl_foo -> a 的指针副本
        rcu_read_unlock();
        return retval;
}
```

RCU

- RCU读者线程示例

```
void foo_update_a(int new_a) {    //写者线程目的是更新a值
        struct foo *new_fp;
        struct foo *old_fp;
        new_fp = kmalloc(sizeof(*new_fp), GFP_KERNEL);
        spin_lock(&foo_mutex);
        old_fp = gbl_foo;
        *new_fp = *old_fp;
        new_fp->a = new_a;
        rcu_assign_pointer(gbl_foo, new_fp);      //使gbl_foo指向新值
        spin_unlock(&foo_mutex);
        call_rcu(&old_fp->rcu, foo_reclaim);      //所有对旧数据的引用完成后，删除old_fp
}
```