



实例分析——第二部分

——第三组：张晨浩 胡力杭 热依汉古丽▪依明 曹贺一
——讲述人：曹贺一



问题1: 请查阅资料, 给出pthread提供的条件变量操作, 并与xv6提供的sleep & wakeup操作进行比较



pthread提供的条件变量操作:

operation	function
block on a condition variable	pthread_cond_wait()
unlock a specific thread	pthread_cond_signal()
unlock all threads	pthread_cond_broadcast()
initialize a condition variable	pthread_cond_init()
destroy condition variable state	pthread_cond_destroy()
block until a specified event	pthread_cond_timedwait()



问题1：请查阅资料，给出pthread提供的条件变量操作，并与xv6提供的sleep & wakeup操作进行比较



xv6提供的sleep & wakeup操作：

```
sleep(void *chan, struct spinlock *lk){...}  
wakeup(void *chan){...}
```

① xv6的锁，用的时候都是acquire和release包起来，保证互斥。中间可以sleep。睡眠过程中，放弃锁lk，别人可以进入临界区。醒来又能获得锁。综上，实际上sleep-lock在这里，相当于实现了一个Brinch Hansen(1975)的monitor。

② XV6中的sleep函数，功能与pthread_cond_wait相似，都使得调用线程阻塞在对应的条件变量上。但是，XV6中的wakeup函数的语义，并不对应pthread_cond_signal函数，而是与pthread_cond_broadcast函数一样会唤醒阻塞在条件变量上的所有线程。



问题2:为什么sleep需要一个lk作为参数? (参考xv6配套讲义)

中的sleep and wakeup一节中的代码改进

错误实现1:

```
void*
send(struct q *q, void *p)
{
    while(q->ptr != 0)
        ;
    q->ptr = p;
    wakeup(q); /*wake recv*/
}

void*
recv(struct q *q)
{
    void *p;
    while((p = q->ptr) == 0)
        sleep(q);
    q->ptr = 0;
    return p;
}
```

考虑一个生产者-消费者问题, 如果用左边代码实现的话, 当send发现q->ptr为0的时候, 给q->ptr赋值, 然后将recv唤醒; recv唤醒后, 进行各种操作, 当q->ptr为0时, 又睡眠。

但是在这之中, 当recv进行到“蓝色线条”也就是while循环判断和执行sleep操作之间时, 被send切换走的话q->ptr就会被赋值, 然后q就会被wakeup, 但是之后recv中q又会被sleep, 导致后来的send无法执行, 使得程序无效。

下面考虑使用锁

问题2:为什么sleep需要一个lk作为参数? (参考xv6配套讲义)
中的sleep and wakeup一节中的代码改进

• 错误实现 2

```
struct q {
    struct spinlock lock;
    void *ptr;
};

void *
send(struct q *q, void *p)
{
    acquire(&q->lock);
    while(q->ptr != 0)
        ;
    q->ptr = p;
    wakeup(q);
    release(&q->lock);
}
```

```
void*
recv(struct q *q)
{
    void *p;
    acquire(&q->lock);
    while((p = q->ptr) == 0)
        sleep(q);
    q->ptr = 0;
    release(&q->lock);
    return p;
}
```

- 错误实现2:
 - receiver在睡眠时持有锁, 导致sender无法获取到锁, 更不能执行wakeup将receiver唤醒
 - 造成死锁
- 需要将锁的信息传给sleep, 在进入睡眠之前释放锁, 被唤醒之后重新获得锁

问题2:为什么sleep需要一个lk作为参数? (参考xv6配套讲义)
中的sleep and wakeup一节中的代码改进



xv6的正确实现:

```
struct q {
    struct spinlock lock;
    void *ptr;
};

void *
send(struct q *q, void *p)
{
    acquire(&q->lock);
    while(q->ptr != 0)
        ;
    q->ptr = p;
    wakeup(q);
    release(&q->lock);
}
```

```
void*
recv(struct q *q)
{
    void *p;
    acquire(&q->lock);
    while((p = q->ptr) == 0)
        sleep(q, &q->lock);
    q->ptr = 0;
    release(&q->lock);
    return p;
}
```

这就是lk参数

问题3:为什么sleep（以及wakeup）要使用ptable.lock?xv6如何通过锁的使用来解决lost wakeup问题?



sleep和wakeup在执行过程中也需要维持一个稳定的状态，该状态通过ptable.lock来维护，保证了其他进程无法再对其进行sleep和wakeup操作。

通过lk和ptable.lock这两把锁的使用，保证了wakeup必须等待sleep让进程睡眠后才能执行，从而解决lost wakeup问题。



问题4. sleep函数中，2890行if语句的作用是什么？

```
if(lk != &ptable.lock){ //DOC: sleeplock0  
    acquire(&ptable.lock); //DOC: sleeplock1  
    release(lk);  
}
```

如果sleep函数传递进来的锁lk与ptable.lock相同，在执行请求ptable.lock（其实就是lk）、释放lk时会产生死锁。

所以如果lk与ptable.lock相同，可以认为acquire和release作用相互抵消，就跳过。



问题4. sleep函数中，2890行if语句的作用是什么？



举个lk与ptable.lock相同的例子：wait函数2964行

```
// Wait for children to exit. (See wakeup1 call in proc_exit.)  
sleep(curproc, &ptable.lock); //DOC: wait-sleep
```



问题5. sleep 函数中，2891行和2892行能不能交换顺序？为什么？

```
if(lk != &ptable.lock){ //DOC: sleeplock0
    acquire(&ptable.lock); //DOC: sleeplock1
    release(lk);
}
```

不能交换顺序。

acquire ptable.lock可以保护临界区资源p->state，release lk是为了防止receiver在睡眠时持有锁，从而造成死锁。拿到ptable.lock锁之后，即使有另外的进程调用wakeup函数，wakeup函数也会等待直到它拿到了ptable.lock，wakeup函数一定是在sleep执行完毕之后。所以acquire ptable.lock之后release lk是安全的。

如果交换顺序，在两个指令之间存在未被锁保护的区域。如果在此处有新的进程进行wakeup函数，将会漏掉正在进行sleep操作的进程，产生错误。

问题6. 阐述sleep函数执行时，进程是如何转入睡眠态，又转入就绪态和运行态，并继续执行sleep的。



①在sleep函数中转入睡眠态：

```
// Go to sleep.  
p->chan = chan;  
p->state = SLEEPING;
```

释放lk之后，将chan记录到pcb的chan域，将当前pcb的state域置为SLEEPING。

②执行调度：

```
sched();
```

问题6. 阐述sleep函数执行时，进程是如何转入睡眠态，又转入就绪态和运行态，并继续执行sleep的。



③当被其他进程进入wakeup函数，进入就绪态

```
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)  
    if(p->state == SLEEPING && p->chan == chan)  
        p->state = RUNNABLE;
```

遍历process table，将所有处于SLEEPING状态且chan域为chan的进程改为RUNNABLE状态。



问题6. 阐述sleep函数执行时，进程是如何转入睡眠态，又转入就绪态和运行态，并继续执行sleep的。



④当进行了下一次调度的时候（进入scheduler函数）

```
if(p->state != RUNNABLE)
    continue;
c->proc = p;
switchvm(p);
p->state = RUNNING;
```

将进程状态从RUNNABLE改为RUNNING并切换进程



问题6. 阐述sleep函数执行时，进程是如何转入睡眠态，又转入就绪态和运行态，并继续执行sleep的。



⑤重新进入sleep函数，完成剩下的内容

```
// Tidy up.  
p->chan = 0;  
  
// Reacquire original lock.  
if(lk != &ptable.lock){ //DOC: sleeplock2  
    release(&ptable.lock);  
    acquire(lk);  
}
```

将chan域清零，表示该pcb并未睡眠（不能够被wakeup）
释放ptable.lock，重新获得lk



问题7: xv6的 wakeup 操作,为什么要拆分成 wakeup 和 wakeup1 两个函数,请举例说明。

```
2953 wakeup1(void *chan)
2954 {
2955     struct proc *p;
2956
2957     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2958         if(p->state == SLEEPING && p->chan == chan)
2959             p->state = RUNNABLE;
2960 }
2961
```

Wakeup1函数将所有标记为chan的进程状态设置为RUNNABLE; 从而使得该进程可以被调度,唤醒了该进程



问题7: xv6的 wakeup 操作,为什么要拆分成 wakeup 和 wakeup1 两个函数,请举例说明。

```
2962 // Wake up all processes sleeping on chan.
2963 void
2964 wakeup(void *chan)
2965 {
2966     acquire(&ptable.lock);
2967     wakeup1(chan);
2968     release(&ptable.lock);
2969 }
```

如果调用wakeup的进程已经拥有的ptable.lock, 重新申请的时候就会出问题。Wakeup1讲所有的标记为函数唤醒所有等待条件变量chan的进程后, 在某一时刻, 调度器选择进程表中第一个就绪态的进程(假设为先前调用sleep函数进入等待状态的线程), 修改进程状态为运行态, 执行swtch进行上下文切换, 切换回sleep函数中继续执行。



问题7: xv6的 wakeup 操作, 为什么要拆分成 wakeup 和 wakeup1 两个函数, 请举例说明。

例子exit函数, 已经拥有了ptable.lock, 这么用时因为之后还要更改其他process的状态, 省的再acquire一次

```
2627 exit(void)
2628 {
2629     struct proc *curproc = myproc();
2630     struct proc *p;
2631     int fd;
2632
2633     if(curproc == initproc)
2634         panic("init exiting");
2635
2636     // Close all open files.
2637     for(fd = 0; fd < NOFILE; fd++){
2638         if(curproc->ofile[fd]){
2639             fileclose(curproc->ofile[fd]);
2640             curproc->ofile[fd] = 0;
2641         }
2642     }
2643
2644     begin_op();
2645     iput(curproc->cwd);
2646     end_op();
2647     curproc->cwd = 0;
2648
2649     acquire(&ptable.lock);
2650
2651     // Parent might be sleeping in wait().
2652     wakeup1(curproc->parent);
2653
2654     // Pass abandoned children to init.
2655     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2656         if(p->parent == curproc){
2657             p->parent = initproc;
2658             if(p->state == ZOMBIE)
2659                 wakeup1(initproc);
2660         }
2661     }
```

问题8: 假设 wakeup 操作唤醒了多个等待相同 channel 的进程, 此时这多个进程会如何执行? xv6 的 wakeup 是否符合 Mesa semantics?

会全部唤醒

如果数据已经被消费了, 被唤醒的程序就会发现没有数据可以处理, 由于 scheduler 进行上下文切换回到的是 sleep 调用 sched 的位置, 当 sleep 退出后, 会再次检测是否满足条件, 如果不满足进入睡眠, 因此保证了不发生错误。

Spurious wakeup

符合 Mesa semantics (条件不为真, 重新判断一次)

```
2807 void
2808 sched(void)
2809 {
2810     int intena;
2811     struct proc *p = myproc();
2812
2813     if(!holding(&ptable.lock))
2814         panic("sched ptable.lock");
2815     if(mycpu()->ncli != 1)
2816         panic("sched locks");
2817     if(p->state == RUNNING)
2818         panic("sched running");
2819     if(readeflags() & FL_IF)
2820         panic("sched interruptible");
2821     intena = mycpu()->intena;
2822     swtch(&p->context, mycpu()->scheduler);
2823     mycpu()->intena = intena;
2824 }
```

```
2757 void
2758 scheduler(void)
2759 {
2760     struct proc *p;
2761     struct cpu *c = mycpu();
2762     c->proc = 0;
2763
2764     for(;;){
2765         // Enable interrupts on this processor.
2766         sti();
2767
2768         // Loop over process table looking for process to run.
2769         acquire(&ptable.lock);
2770         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2771             if(p->state != RUNNABLE)
2772                 continue;
2773
2774             // Switch to chosen process. It is the process's job
2775             // to release ptable.lock and then reacquire it
2776             // before jumping back to us.
2777             c->proc = p;
2778             switchvm(p);
2779             p->state = RUNNING;
2780
2781             swtch(&(c->scheduler), p->context);
2782             switchkvm();
2783
2784             // Process is done running for now.
2785             // It should have changed its p->state before coming back.
2786             c->proc = 0;
2787         }
2788         release(&ptable.lock);
2789
2790     }
2791 }
```

问题9: wakeup 时如果没有 sleeping 的进程, wakeup 会阻塞吗?



不会阻塞, 在调度的过程中是顺序遍历, pcb在ptable中的顺序就是唤醒的顺序

```
2953 wakeup1(void *chan)
2954 {
2955     struct proc *p;
2956
2957     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
2958         if(p->state == SLEEPING && p->chan == chan)
2959             p->state = RUNNABLE;
2960 }
2961
```





Linux 信号量



问题1.1: 信号量的数据结构



如下:

```
struct semaphore {  
    raw_spinlock_t lock; //自旋锁, 用于count值的互斥访问  
    unsigned int count; //计数值, 能同时允许访问的数量  
    struct list_head wait_list; //等待列表, 加入不能立即获取到信号量的访问者  
};  
  
struct semaphore_waiter {  
    struct list_head list; //用于添加到信号量的等待列表中  
    struct task_struct *task; //用于指向等待的进程, 在实际实现中, 指向current  
    bool up; //用于标识是否已经释放  
};
```



问题1.2：信号量的PV操作



有以下6种：

基本2种	<code>void down(struct semaphore *sem)</code> <code>void up(struct semaphore *sem)</code>
4种 down 操作	<code>int down_interruptible(struct semaphore *sem)</code>
	<code>int down_killable(struct semaphore *sem)</code>
	<code>int down_trylock(struct semaphore *sem)</code>
	<code>int down_timeout(struct semaphore *sem, long jiffies)</code>



问题2：各类down操作的用途

- **down_interruptible:**

```
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;
}
```

用途：可以在无法获得信号量时，通过__down_interruptible函数调用__down_common函数让进程进入浅度睡眠状态（即可以被另外信号唤醒），并且给result赋值，通过result的值判断其是否被信号唤醒。即用于让进程浅度睡眠。



问题2：各类down操作的用途

- **down_killable:**

```
int down_killable(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_killable(sem);
    raw_spin_unlock_irqrestore(&sem->lock, flags);
    return result;
}
```

用途：与down_interruptible类似，但是用于让进程进入可杀死的深度睡眠状态（可被致命信号唤醒，如SIGKILL）。

问题2：各类down操作的用途

- **down_trylock:**

```
int down_trylock(struct semaphore *sem)
{
    unsigned long flags;
    int count;
    raw_spin_lock_irqsave(&sem->lock, flags);
    count = sem->count - 1;
    if (likely(count >= 0))
        sem->count = sem->count;
    raw_spin_unlock_irqrestore(&sem->lock, flags);
}
```

用途：可用于不需睡眠的进程，直接尝试获取信号量，如果无法获取会直接根据返回值处理，即不会让该进程睡眠。

问题2：各类down操作的用途

- **down_timeout:**

```
int down_timeout(struct semaphore *sem, long timeout)
{
    unsigned long flags;
    int result = 0;
    raw_spin_lock_irqsave(&sem->lock, flags);
    if (likely(sem->count > 0))
        sem->count--;
    else
        result = __down_timeout(sem, timeout);
    raw_spin_unlock_irqrestore(&sem->lock, flags);
    return result;
}
```

用途：可用于睡眠时间有限的进程，若该进程在超时时间（timeout）内没有获取信号量则会自行唤醒。

问题3：简要分析down和up的实现

- down:

```
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);
}
```

原理：和 `down_interruptible` 类似，先去尝试获取信号量（flag），如果无法获得信号量，则通过 `__down` 函数让进程进入深度睡眠（只有获取信号量才会唤醒）状态，没有返回值。

问题3：简要分析down和up的实现

- 4个__down:

```
void __sched __down(struct semaphore *sem){
    __down_common(sem, TASK_UNINTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
}
int __sched __down_interruptible(struct semaphore *sem){
    __down_common(sem, TASK_INTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
}
int __sched __down_killable(struct semaphore *sem){
    __down_common(sem, TASK_KILLABLE, MAX_SCHEDULE_TIMEOUT);
}
int __sched __down_timeout(struct semaphore *sem, long timeout){
    __down_common(sem, TASK_UNINTERRUPTIBLE, timeout);
}
```

原理：用于给与__down_common函数不同的参数，从而实现不同的睡眠功能。

问题3：简要分析down和up的实现

- **__down_common :**

```
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 = 0;
    for (;;) {
        if(signal_pending_state(state,task))
            goto interrupted;
        if (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;
    }
timed_out:
    list_del(&waiter.list);
    return -ETIME;
interrupted:
    list_del(&waiter.list);
    return -EINTR;
}
```

问题3：简要分析down和up的实现



- **__down_common** 分析：

- **for前**：将当前进程设置一个waiter节点加到wait_list管理的等待队列里，并且初始化数据。
- **signal_pending_state函数与第一个if-goto部分**：判断当前进程是否要进行浅度睡眠或可杀死的深度睡眠，且存在一个挂起的信号或致命信号，如果是，则将其从等待队列中除去（唤醒），并且返回特定错误代码。
- **第二个if-goto部分**：判断限时睡眠的进程是否超时，如果是，则将其从等待队列中除去（唤醒），并且返回特定错误代码。
- **schedule_timeout函数**：使进程进入相应的睡眠状态直到等待超时。
- 由于waiter.up设置为0，因此会无限循环，直到进程被该信号量的up操作所唤醒（waiter.up会为1），此时进程可以获得信号量。



问题3：简要分析down和up的实现

- up:

```
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);
}
```

原理：判断等待队列，如果为空则增加semaphore的计数。如果不为空则调用__up函数唤醒进程。



问题3：简要分析down和up的实现

- **__up:**

```
static ninline 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 = 1;
    wake_up_process(waiter->task);
}
```

原理：将该进程从等待队列中除去，waiter.up置1，唤醒该进程。





中国科学院大学
University of Chinese Academy of Sciences



THANKS

