XV6 条件变量

报告人：高亦钊

小组成员：高亦钊，索玉玺，王子铭，范楷元

Background

考虑一个最简单的例子，一对发送者和接受者（消费者和生产者）

Send 进程将一个数据的指针发送给Recv

```
100     struct q {
101       void *ptr;
102     };
103
104     void*
105     send(struct q *q, void *p)
106     {
107       while(q->ptr != 0)
108         ;
109       q->ptr = p;
110     }
111
```

```
112     void*
113     recv(struct q *q)
114     {
115       void *p;
116
117       while((p = q->ptr) == 0)
118         ;
119       q->ptr = 0;
120       return p;
121     }
```

Ver 1.0

优点: 没有数据丢失
缺点: 如果很久发一次数据，recv一直在判断是否有新数据，浪费CPU资源

```
100     struct q {
101       void *ptr;
102     };
103
104     void*
105     send(struct q *q, void *p)
106     {
107       while(q->ptr != 0)
108         ;
109       q->ptr = p;
110     }
111
```

```
112     void*
113     recv(struct q *q)
114     {
115       void *p;
116
117       while((p = q->ptr) == 0)
118         ;
119       q->ptr = 0;
120       return p;
121     }
```

Ver 2.0

假象我们有一组函数wakeup和sleep,让recv函数在没有新数据的时候放弃CPU资源，当有新数据的时候send将recv唤醒

新的问题: lost wake-up

```
201     void*
202     send(struct q *q, void *p)
203     {
204       while(q->ptr != 0)
205         ;
206       q->ptr = p;
207       wakeup(q);  /* wake recv */
208     }
209
```

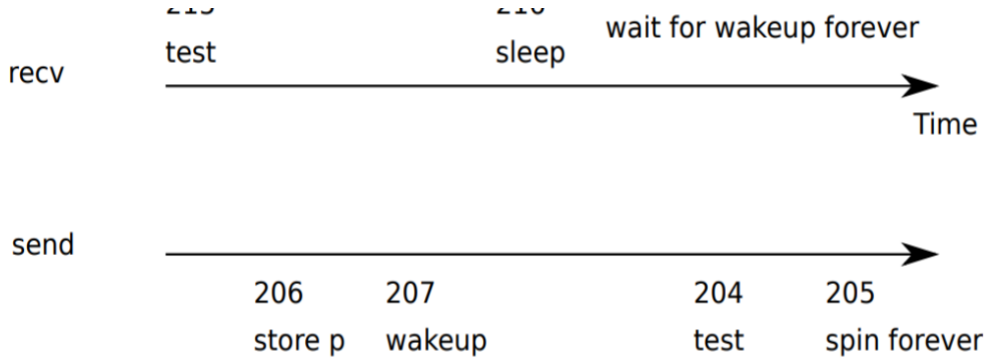
```
210     void*
211     recv(struct q *q)
212     {
213       void *p;
214
215       while((p = q->ptr) == 0)
216         sleep(q);
217       q->ptr = 0;
218       return p;
219     }
```

Lost wake-up

如果recv函数在结束判断决定执行sleep,但是sleep函数还没有彻底执行完时，send将新数据写入并且调用wakeup,此时发现并没有进程在sleep

```
201     void*
202     send(struct q *q, void *p)
203     {
204       while(q->ptr != 0)
205         ;
206       q->ptr = p;
207       wakeup(q);  /* wake recv */
208     }
209
```

```
210     void*
211     recv(struct q *q)
212     {
213       void *p;
214
215       while((p = q->ptr) == 0)
216         sleep(q);
217       q->ptr = 0;
218       return p;
219     }
```



原因总结：没有保证sleep成功前没有其他进程来发送新的数据

Ver 3.0: 加锁（在进入临界区的时候加锁）
新的问题：产生死锁（sleep结束后锁没有释放）

```
300     struct q {
301       struct spinlock lock;
302       void *ptr;
303     };
304
305     void*
306     send(struct q *q, void *p)
307     {
308       acquire(&q->lock);
309       while(q->ptr != 0)
310         ;
311       q->ptr = p;
312       wakeup(q);
313       release(&q->lock);
314     }
315
```

```
316     void*
317     recv(struct q *q)
318     {
319       void *p;
320
321       acquire(&q->lock);
322       while((p = q->ptr) == 0)
323         sleep(q);
324       q->ptr = 0;
325       release(&q->lock);
326       return p;
327     }
```

Ver 4.0:我们需要让sleep函数在结束的时候将锁释放，因此需要将锁作为参数，让sleep成功之后释放锁

```
400     struct q {
401       struct spinlock lock;
402       void *ptr;
403     };
404
405     void*
406     send(struct q *q, void *p)
407     {
408       acquire(&q->lock);
409       while(q->ptr != 0)
410         ;
411       q->ptr = p;
412       wakeup(q);
413       release(&q->lock);
414     }
```

```
415
416     void*
417     recv(struct q *q)
418     {
419       void *p;
420
421       acquire(&q->lock);
422       while((p = q->ptr) == 0)
423         sleep(q, &q->lock);
424       q->ptr = 0;
425       release(&q->lock);
426       return p;
427     }
```

Sleep
参数：*chan
(Callers of sleep and wakeup can use any mutually convenient number as the channel. Xv6 often uses the address of a kernel data structure involved in the waiting.)

参数：*lk（问题2）

为什么需要锁前面已经提及：
1.锁用来保证lost wake-up, 让sleep能安全的进入睡眠
2.Sleep函数需要将锁释放，否则这把锁不会被释放，wakeup函数无法执行

```
415  // Atomically release lock and sleep on chan.
416  // Reacquires lock when awakened.
417  void
418  sleep(void *chan, struct spinlock *lk)
419  {
420    struct proc *p = myproc();
421
422    if(p == 0)
423      panic("sleep");
424
425    if(lk == 0)
426      panic("sleep without lk");
427
428    // Must acquire ptable.lock in order to
429    // change p->state and then call sched.
430    // Once we hold ptable.lock, we can be
431    // guaranteed that we won't miss any wakeup
432    // (wakeup runs with ptable.lock locked),
433    // so it's okay to release lk.
434    if(lk != &ptable.lock){  //DOC: sleeplock0
435      acquire(&ptable.lock);  //DOC: sleeplock1
436      release(lk);
437    }
438    // Go to sleep.
439    p->chan = chan;
440    p->state = SLEEPING;
441
442    sched();
```

```
444    // Tidy up.
445    p->chan = 0;
446
447    // Reacquire original lock.
448    if(lk != &ptable.lock){  //DOC: sleeplock2
449      release(&ptable.lock);
450      acquire(lk);
451    }
452  }
```

```
467  // Wake up all processes sleeping on chan.
468  void
469  wakeup(void *chan)
470  {
471    acquire(&ptable.lock);
472    wakeup1(chan);
473    release(&ptable.lock);
474  }
475
454  //PAGEBREAK!
455  // Wake up all processes sleeping on chan.
456  // The ptable lock must be held.
457  static void
458  wakeup1(void *chan)
459  {
460    struct proc *p;
461
462    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
463      if(p->state == SLEEPING && p->chan == chan)
464        p->state = RUNNABLE;
465  }
```

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 wake-up问题？

为什么要使用ptable.lock:
因为涉及到线程状态的变化

```
415  // Atomically release lock and sleep on chan.
416  // Reacquires lock when awakened.
417  void
418  sleep(void *chan, struct spinlock *lk)
419  {
420    struct proc *p = myproc();
421
422    if(p == 0)
423      panic("sleep");
424
425    if(lk == 0)
426      panic("sleep without lk");
427
428    // Must acquire ptable.lock in order to
429    // change p->state and then call sched.
430    // Once we hold ptable.lock, we can be
431    // guaranteed that we won't miss any wakeup
432    // (wakeup runs with ptable.lock locked),
433    // so it's okay to release lk.
434    if(lk != &ptable.lock){  //DOC: sleeplock0
435      acquire(&ptable.lock);  //DOC: sleeplock1
436      release(lk);
437    }
438    // Go to sleep.
439    p->chan = chan;
440    p->state = SLEEPING;
441
442    sched();
```

```
444    // Tidy up.
445    p->chan = 0;
446
447    // Reacquire original lock.
448    if(lk != &ptable.lock){  //DOC: sleeplock2
449      release(&ptable.lock);
450      acquire(lk);
451    }
452  }

467  // Wake up all processes sleeping on chan.
468  void
469  wakeup(void *chan)
470  {
471    acquire(&ptable.lock);
472    wakeup1(chan);
473    release(&ptable.lock);
474  }
475
454  //PAGEBREAK!
455  // Wake up all processes sleeping on chan.
456  // The ptable lock must be held.
457  static void
458  wakeup1(void *chan)
459  {
460    struct proc *p;
461
462    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
463      if(p->state == SLEEPING && p->chan == chan)
464        p->state = RUNNABLE;
465  }
```

问题4：sleep 函数中，2890行if语句的作用是什么？
如果lk是ptable.lock，防止再申请一次，并且保证sleep不会当成正常的一个lock并释放掉

```
415  // Atomically release lock and sleep on chan.
416  // Reacquires lock when awakened.
417  void
418  sleep(void *chan, struct spinlock *lk)
419  {
420    struct proc *p = myproc();
421
422    if(p == 0)
423      panic("sleep");
424
425    if(lk == 0)
426      panic("sleep without lk");
427
428    // Must acquire ptable.lock in order to
429    // change p->state and then call sched.
430    // Once we hold ptable.lock, we can be
431    // guaranteed that we won't miss any wakeup
432    // (wakeup runs with ptable.lock locked),
433    // so it's okay to release lk.
434    if(lk != &ptable.lock){  //DOC: sleeplock0
435      acquire(&ptable.lock);  //DOC: sleeplock1
436      release(lk);
437    }
438    // Go to sleep.
439    p->chan = chan;
440    p->state = SLEEPING;
441
442    sched();
```

```
444    // Tidy up.
445    p->chan = 0;
446
447    // Reacquire original lock.
448    if(lk != &ptable.lock){  //DOC: sleeplock2
449      release(&ptable.lock);
450      acquire(lk);
451    }
452  }
```

```
467  // Wake up all processes sleeping on chan.
468  void
469  wakeup(void *chan)
470  {
471    acquire(&ptable.lock);
472    wakeup1(chan);
473    release(&ptable.lock);
474  }
475
454  //PAGEBREAK!
455  // Wake up all processes sleeping on chan.
456  // The ptable lock must be held.
457  static void
458  wakeup1(void *chan)
459  {
460    struct proc *p;
461
462    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
463      if(p->state == SLEEPING && p->chan == chan)
464        p->state = RUNNABLE;
465  }
```

问题5：sleep 函数中，2891行和2892行能不能交换顺序？为什么？

不能！在现在这种情况下sleep在释放lk前会获得ptable.lock，同时拥有两把锁，保证没有进程能执行wakeup
如果被调换位置，先释放lk,再申请ptable.lock，就会存在一个时刻sleep还没拿到ptable.lock，但是释放了lk,这时如果lk和ptable.lock都被一个进程得到，那么就可以执行wakeup,造成lost wakeup

```
415  // Atomically release lock and sleep on chan.
416  // Reacquires lock when awakened.
417  void
418  sleep(void *chan, struct spinlock *lk)
419  {
420    struct proc *p = myproc();
421
422    if(p == 0)
423      panic("sleep");
424
425    if(lk == 0)
426      panic("sleep without lk");
427
428    // Must acquire ptable.lock in order to
429    // change p->state and then call sched.
430    // Once we hold ptable.lock, we can be
431    // guaranteed that we won't miss any wakeup
432    // (wakeup runs with ptable.lock locked),
433    // so it's okay to release lk.
434    if(lk != &ptable.lock){  //DOC: sleeplock0
435      acquire(&ptable.lock);  //DOC: sleeplock1
436      release(lk);
437    }
438    // Go to sleep.
439    p->chan = chan;
440    p->state = SLEEPING;
441
442    sched();
```

```
444    // Tidy up.
445    p->chan = 0;
446
447    // Reacquire original lock.
448    if(lk != &ptable.lock){  //DOC: sleeplock2
449      release(&ptable.lock);
450      acquire(lk);
451    }
452  }
```

```
467  // Wake up all processes sleeping on chan.
468  void
469  wakeup(void *chan)
470  {
471    acquire(&ptable.lock);
472    wakeup1(chan);
473    release(&ptable.lock);
474  }
475
454  //PAGEBREAK!
455  // Wake up all processes sleeping on chan.
456  // The ptable lock must be held.
457  static void
458  wakeup1(void *chan)
459  {
460    struct proc *p;
461
462    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
463      if(p->state == SLEEPING && p->chan == chan)
464        p->state = RUNNABLE;
465  }
```

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?

进入就绪态:
1.将p->state 设为sleeping；
p->chan 设为chan标记睡再哪个通道上
2.调用sche()，switch context, 进入调度程序，交出CPU资源

```
415  // Atomically release lock and sleep on chan.
416  // Reacquires lock when awakened.
417  void
418  sleep(void *chan, struct spinlock *lk)
419  {
420    struct proc *p = myproc();
421
422    if(p == 0)
423      panic("sleep");
424
425    if(lk == 0)
426      panic("sleep without lk");
427
428    // Must acquire ptable.lock in order to
429    // change p->state and then call sched.
430    // Once we hold ptable.lock, we can be
431    // guaranteed that we won't miss any wakeup
432    // (wakeup runs with ptable.lock locked),
433    // so it's okay to release lk.
434    if(lk != &ptable.lock){  //DOC: sleeplock0
435      acquire(&ptable.lock);  //DOC: sleeplock1
436      release(lk);
437    }
438    // Go to sleep.
439    p->chan = chan;
440    p->state = SLEEPING;
441
442    sched();
```

```
444    // Tidy up.
445    p->chan = 0;
446
447    // Reacquire original lock.
448    if(lk != &ptable.lock){  //DOC: sleeplock2
449      release(&ptable.lock);
450      acquire(lk);
451    }
452  }
```

```
467  // Wake up all processes sleeping on chan.
468  void
469  wakeup(void *chan)
470  {
471    acquire(&ptable.lock);
472    wakeup1(chan);
473    release(&ptable.lock);
474  }
475
454  //PAGEBREAK!
455  // Wake up all processes sleeping on chan.
456  // The ptable lock must be held.
457  static void
458  wakeup1(void *chan)
459  {
460    struct proc *p;
461
462    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
463      if(p->state == SLEEPING && p->chan == chan)
464        p->state = RUNNABLE;
465  }
```

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?

如何进入就绪态和运行态:
1.wakeup1函数将睡再chan上所有的进程都设为runnable
2.等CPU接下来进行调度的时候就能将某个进程唤醒，变为running，并回到sleep调用sched时的context

Context switch from 380 to 346 and from 346 to 380

```
365  void
366  sched(void)
367  {
368    int intena;
369    struct proc *p = myproc();
370
371    if(!holding(&ptable.lock))
372      panic("sched ptable.lock");
373    if(mycpu()->ncli != 1)
374      panic("sched locks");
375    if(p->state == RUNNING)
376      panic("sched running");
377    if(readeflags()&FL_IF)
378      panic("sched interruptible");
379    intena = mycpu()->intena;
380    swtch(&p->context, mycpu()->scheduler);
381    mycpu()->intena = intena;
382  }
---
```

```
322  void
323  scheduler(void)
324  {
325    struct proc *p;
326    struct cpu *c = mycpu();
327    c->proc = 0;
328
329    for(;;){
330      // Enable interrupts on this processor.
331      sti();
332
333      // Loop over process table looking for process to run.
334      acquire(&ptable.lock);
335      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
336        if(p->state != RUNNABLE)
337          continue;
338
339        // Switch to chosen process.  It is the process's job
340        // to release ptable.lock and then reacquire it
341        // before jumping back to us.
342        c->proc = p;
343        switchuvm(p);
344        p->state = RUNNING;
345
346        swtch(&(c->scheduler), p->context);
347        switchkvm();
348
349        // Process is done running for now.
350        // It should have changed its p->state before coming back.
351        c->proc = 0;
352      }
353      release(&ptable.lock);
354
355    }
356  }
```

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。

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。
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

```
467  // Wake up all processes sleeping on chan.
468  void
469  wakeup(void *chan)
470  {
471    acquire(&ptable.lock);
472    wakeup1(chan);
473    release(&ptable.lock);
474  }
475
```

```
454  //PAGEBREAK!
455  // Wake up all processes sleeping on chan.
456  // The ptable lock must be held.
457  static void
458  wakeup1(void *chan)
459  {
460    struct proc *p;
461
462    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
463      if(p->state == SLEEPING && p->chan == chan)
464        p->state = RUNNABLE;
465  }
```

```
172  // Exit the current process.  Does not return.
173  // An exited process remains in the zombie state
174  // until its parent calls wait() to find out it exited.
```

```
175  void
176  exit(void)
177  {
178    struct proc *p;
179    int fd;
180
181    if(proc == initproc){
182      panic("init exiting");
183    }
184
185
186    // Close all open files.
187    for(fd = 0; fd < NOFILE; fd++){
188      if(proc->ofile[fd]){
189        fileclose(proc->ofile[fd]);
190        proc->ofile[fd] = 0;
191      }
192    }
193
194    iput(proc->cwd);
195    proc->cwd = 0;
196
197    acquire(&ptable.lock);
198
199    // Parent might be sleeping in wait().
200    wakeup1(proc->parent);
201
202    // Pass abandoned children to init.
203    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
204      if(p->parent == proc){
205        p->parent = initproc;
206        if(p->state == ZOMBIE)
207          wakeup1(initproc);
208      }
209    }
```

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?

1.会全部唤醒
2.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)

```
while((p = q->ptr) == 0)
  sleep(q, &q->lock);
```

```
365  void
366  sched(void)
367  {
368    int intena;
369    struct proc *p = myproc();
370
371    if(!holding(&ptable.lock))
372      panic("sched ptable.lock");
373    if(mycpu()->ncli != 1)
374      panic("sched locks");
375    if(p->state == RUNNING)
376      panic("sched running");
377    if(readeflags()&FL_IF)
378      panic("sched interruptible");
379    intena = mycpu()->intena;
380    swtch(&p->context, mycpu()->scheduler
381    mycpu()->intena = intena;
382  }
```

```
322  void
323  scheduler(void)
324  {
325    struct proc *p;
326    struct cpu *c = mycpu();
 7    c->proc = 0;
 8
 9    for(;;){
 0      // Enable interrupts on this processor.
331      sti();
332
333      // Loop over process table looking for process to run.
334      acquire(&ptable.lock);
335      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
336        if(p->state != RUNNABLE)
337          continue;
338
339        // Switch to chosen process.  It is the process's job
340        // to release ptable.lock and then reacquire it
341        // before jumping back to us.
342        c->proc = p;
343        switchuvm(p);
344        p->state = RUNNING;
345
346        swtch(&(c->scheduler), p->context);
347        switchkvm();
348
349        // Process is done running for now.
350        // It should have changed its p->state before coming back.
351        c->proc = 0;
352      }
353      release(&ptable.lock);
354
355    }
356  }
```

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

不会

```
467  // Wake up all processes sleeping on chan.
468  void
469  wakeup(void *chan)
470  {
471    acquire(&ptable.lock);
472    wakeup1(chan);
473    release(&ptable.lock);
474  }
475
```

```
454  //PAGEBREAK!
455  // Wake up all processes sleeping on chan.
456  // The ptable lock must be held.
457  static void
458  wakeup1(void *chan)
459  {
460    struct proc *p;
461
462    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
463      if(p->state == SLEEPING && p->chan == chan)
464        p->state = RUNNABLE;
465  }
```

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 操作比 较。

1. 初始化条件变量 pthread_cond_init
   int pthread_cond_init(pthread_cond_t *cv, const pthread_condattr_t *cattr);

2. 阻塞 pthread_cond_wait
   int pthread_cond_wait(pthread_cond_t *cv, pthread_mutex_t *mutex);

3. 解除条件变量上的阻塞 pthread_cond_signal
   int pthread_cond_signal(pthread_cond_t *cv);

4. 定时阻塞 pthread_cond_timewait
   int pthread_cond_timedwait(pthread_cond_t *cv, pthread_mutex_t *mp, const structtimespec * abstime);

5. 释放阻塞的所有进程 pthread_cond_broadcast
   int pthread_cond_broadcast(pthread_cond_t *cv);

6. 释放条件变量 pthread_cond_destroy
   int pthread_cond_destroy(pthread_cond_t *cv);

对比：
1.pthread条件变量有自己的数据结构，xv6是自行定义的，只需要满足有锁。

2.pthread 条件变量分为signal和broadcast,但是XV6只有“broadcast”. 使用signal的时候只唤醒一个进程，哪一个又调度方法决定

3. Pthread还支持定时阻塞和释放条件变量

```
typedef struct {
        struct pthread_queue queue;
        int flags;
        int waiters;
        pthread_mutex_t *mutex;
} pthread_cond_t;
```

1. Condition Variable Data Structure

```
typedef struct {
        struct pthread_queue queue;
        int flags;
        int waiters;
        pthread_mutex_t *mutex;
} pthread_cond_t;
```

2.Initialization

PTHREAD_COND_INITIALIZER

int pthread_cond_init(pthread_cond_t *cv, const pthread_condattr_t *cattr);

3. Wait
int pthread_cond_wait(pthread_cond_t *cv, pthread_mutex_t *mutex);

4.Signal
int pthread_cond_signal(pthread_cond_t *cv);
int pthread_cond_timedwait(pthread_cond_t *cv, pthread_mutex_t *mp, const structtimespec * abstime);
int pthread_cond_broadcast(pthread_cond_t *cv);
int pthread_cond_destroy(pthread_cond_t *cv);

Pthread提供的条件变量操作

1. 初始化条件变量 pthread_cond_init
   int pthread_cond_init(pthread_cond_t *cv, const pthread_condattr_t *cattr);

2. 阻塞 pthread_cond_wait
   int pthread_cond_wait(pthread_cond_t *cv, pthread_mutex_t *mutex);

3. 解除条件变量上的阻塞 pthread_cond_signal
   int pthread_cond_signal(pthread_cond_t *cv);

4. 定时阻塞 pthread_cond_timewait
   int pthread_cond_timedwait(pthread_cond_t *cv, pthread_mutex_t *mp, const structtimespec * abstime);

5. 释放阻塞的所有进程 pthread_cond_broadcast
   int pthread_cond_broadcast(pthread_cond_t *cv);

6. 释放条件变量 pthread_cond_destroy
   int pthread_cond_destroy(pthread_cond_t *cv);

Linux 信号量

问题1：Linux中信号量的数据结构及其对应的PV操作

1、Linux中信号量的数据结构：

结构体名：

struct semaphore

自旋锁：raw_spinlock_t lock;

信号量长度：unsigned int count;

等待进程的链表：struct list_head wait_list;

2、对应的PV操作：

（1）P操作

上锁：raw_spin_lock_irqsave(&sem->lock, flags);

信号量大于0，获得信号量：if (likely(sem->count > 0))

sem->count--;

信号量小于等于0，休眠：else

__down(sem);

释放锁：raw_spin_unlock_irqrestore(&sem->lock, flags);

```
struct semaphore {
        raw_spinlock_t          lock;
        unsigned int            count;
        struct list_head        wait_list;
};

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);
```

问题1：Linux中信号量的数据结构及其对应的PV操作

- 2、对应的PV操作
- （2）V操作
- 上锁：raw_spin_lock_irqsave(&sem->lock, flags);
- 等待队列为空，信号量加1：if (likely(list_empty(&sem->wait_list)))
- sem->count++;
- 等待队列非空，唤醒：else
- __up(sem);
- 释放锁：raw_spin_unlock_irqrestore(&sem->lock, flags);

```
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);
```

问题2：说明Linux扩展的各类down操作的用途

- 1、down：在此函数中首先进行信号量资源数的查看，如果信号量数据(count)不为0，则把其减1，并返回，调用成功；否则调用__down进行等待，调用者进行睡眠；该函数的调用不允许中断。
- 2、down_interruptible：该函数功能和down类似，不同之处为，down不会被信号（signal）打断，但down_interruptible能被信号打断，因此该函数有返回值来区分是正常返回还是被信号中断，如果返回0，表示获得信号量正常返回，如果被信号打断，返回-EINTR。
- 3、down_killable：down_killable与down_interruptible相近，最终传入的__down_common的实参有所不同(TASK_KILLABLE和TASK_INTERRUPTIBLE)，睡眠的进程可以因为受到致命信号而被唤醒，中断获取信号量的操作。

```
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);
```

```
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;
}
EXPORT_SYMBOL(down_interruptible);
```

```
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;
}
```

问题2：说明Linux扩展的各类down操作的用途

- 4、down_trylock：试图获取资源，若无法获得则直接返回1而不睡眠。返回0则 表示获取到了资源
- 5、down_timeout：可以自定义超时时间，也就是如果在超时间内不能得到资源，调用者会因为超时而自行唤醒。其实现过程如下，请注意超时参数的传入。其中TASK_UNINTERRUPTIBLE，如down一样，不可被致死信号和信号打断。

```
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 = count;
        raw_spin_unlock_irqrestore(&sem->lock, flags);

        return (count < 0);
}
EXPORT_SYMBOL(down_trylock);
```

```
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;
}
EXPORT_SYMBOL(down_timeout);
```

问题三：简要分析down和up操作的实现

- **down操作的实现**

上锁：raw_spin_lock_irqsave(&sem->lock, flags);

信号量大于0，获得信号量：if (likely(sem->count > 0))

sem->count--;

信号量小于等于0，休眠：else

__down(sem);

释放锁：raw_spin_unlock_irqrestore(&sem->lock, flags)

```
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);
```

问题三：简要分析down和up操作的实现

- **down操作的实现**
- 2、__down函数
- 调用__down_common，传入参量state = TASK_UNINTERRUPTIBLE，不可被信号申请中断：
- __down_common(sem, TASK_UNINTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
- MAX_SHEDULE_TIMEOUT 表示无限期睡眠

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

问题三：简要分析down和up操作的实现

- **up操作的实现**
- 1、__up函数
- 唤醒最早等待的沉睡进程，并从等待链表中删除此节点
- 2、up函数
- 上锁：raw_spin_lock_irqsave(&sem->lock, flags);
- 等待队列为空，信号量加1：if (likely(list_empty(&sem->wait_list)))
- sem->count++;
- 等待队列非空，唤醒：else
- __up(sem);
- 释放锁：raw_spin_unlock_irqrestore(&sem->lock, flags);

```
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);
}
```

```
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);
```

问题三：简要分析down和up操作的实现

- **down操作的实现**
- 1、__down_common函数
- （1）创建一个等待进程的表并接到信号量结构体的等待进程表尾：
- list_add_tail(&waiter.list, &sem->wait_list);
- （2）将up设置为false，当进程被正确唤醒时up = true：waiter.up = false;
- （3）休眠：for (;;)
- （4）根据state参量的设置，被信号中断：if (signal_pending_state(state, current))
- goto interrupted;
- （5）超时中断：if (unlikely(timeout <= 0))
- goto timed_out;
- （6）让出CPU：timeout = schedule_timeout(timeout);
- （7）正常唤醒：if (waiter.up)
- return 0;
- （8）超时中断处理：timed_out:
- list_del(&waiter.list);
- return -ETIME;
- （9）信号打断处理：interrupted:
- list_del(&waiter.list);
- return -EINTR;

```
tatic 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;
}
```