University of BRISTOL

Computer Systems B

COMS20012

Introduction to Operating Systems and Security

bristol.ac.uk

Implementing semaphores

OS161 code

Semaphores (kern/thread/synch.c)

```
void P(struct semaphore *sem)
{
        KASSERT(sem != NULL);
        KASSERT(curthread->t_in_interrupt == false);
        spinlock_acquire(&sem->sem_lock);
        while (sem->sem_count == 0) {
                // do something if we need to wait
        }
        KASSERT(sem->sem_count > 0);
        sem->sem_count--;
        spinlock_release(&sem->sem_lock);
}
```

1.Acquire the spin lock
2.Check if there are some resources available (counter > 0)
3.If yes, we're lucky. Happily go to step 8.
4.If no, then we first grab the lock of the wait channel, since the wait channel is also shared.
5.Release the spin lock, and wait on the wait channel by calling wchan_sleep
6.We're sleeping...
7.After wake up, first grab the spin lock, and go to step 2
8.At this point, the counter should be positive, decrement it by 1
9.Release the spin lock, and return

Semaphores (kern/thread/synch.c)

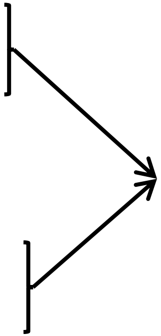
```
void P(struct semaphore *sem)
{
        KASSERT(sem != NULL);
        KASSERT(curthread->t_in_interrupt == false);
        spinlock_acquire(&sem->sem_lock);
        while (sem->sem_count == 0) {
                // do something if we need to wait
        }
        KASSERT(sem->sem_count > 0);
        sem->sem_count--;
        spinlock_release(&sem->sem_lock);
}
```

Conditions MUST be true

Conditions MUST be true

Semaphores (kern/thread/synch.c)

```
void P(struct semaphore *sem)
{
        KASSERT(sem != NULL);
        KASSERT(curthread->t_in_interrupt == false);
        spinlock_acquire(&sem->sem_lock);
        while (sem->sem_count == 0) {
                // do something if we need to wait
        }
        KASSERT(sem->sem_count > 0);
        sem->sem_count--;
        spinlock_release(&sem->sem_lock);
}
```



Check and Decrement counter under lock

Semaphores (kern/thread/synch.c)

```
void P(struct semaphore *sem)
{
        KASSERT(sem != NULL);
        KASSERT(curthread->t_in_interrupt == false);
        spinlock_acquire(&sem->sem_lock);
        while (sem->sem_count == 0) {
                // do something if we need to wait
        }
        KASSERT(sem->sem_count > 0);
        sem->sem_count--;
        spinlock_release(&sem->sem_lock);
}
```

Semaphores (kern/thread/synch.c)

```
void P(struct semaphore *sem)
{
        KASSERT(sem != NULL);
        KASSERT(curthread->t_in_interrupt == false);
        spinlock_acquire(&sem->sem_lock);
        while (sem->sem_count == 0) {
                release lock
                sleep
                acquire lock
        }
        KASSERT(sem->sem_count > 0);
        sem->sem_count--;
        spinlock_release(&sem->sem_lock);
}
```

Semaphores (kern/thread/synch.c)

```
void P(struct semaphore *sem)
{
        KASSERT(sem != NULL);
        KASSERT(curthread->t_in_interrupt == false);
        spinlock_acquire(&sem->sem_lock);
        while (sem->sem_count == 0) {
                wchan_sleep(sem->sem_wchan, &sem->sem_lock);
        }
        KASSERT(sem->sem_count > 0);
        sem->sem_count--;
        spinlock_release(&sem->sem_lock);
}
```

At a high level this is what this function does.

(see kern/thread/thread.c)

Semaphores (kern/thread/synch.c)

```
void P(struct semaphore *sem)
{
        KASSERT(sem != NULL);
        KASSERT(curthread->t_in_interrupt == false);
        spinlock_acquire(&sem->sem_lock);
        while (sem->sem_count == 0) {
                wchan_sleep(sem->sem_wchan, &sem->sem_lock);
        }
        KASSERT(sem->sem_count > 0);
        sem->sem_count--;
        spinlock_release(&sem->sem_lock);
}
```

Semaphores (kern/thread/synch.c)

```
void V(struct semaphore *sem)
{
        KASSERT(sem != NULL);

        spinlock_acquire(&sem->sem_lock);

        sem->sem_count++;
        KASSERT(sem->sem_count > 0);
        wchan_wakeone(sem->sem_wchan, &sem->sem_lock);

        spinlock_release(&sem->sem_lock);
}
```

1.Acquire the spin lock
2.Increment the counter by 1
3.Wake up some poor guy in the wait channel by calling wchan_wakeone)
4.Release the spin lock and return

Semaphores (kern/thread/synch.c)

```
void V(struct semaphore *sem)
{
        KASSERT(sem != NULL);

        spinlock_acquire(&sem->sem_lock);

        sem->sem_count++;
        KASSERT(sem->sem_count > 0);
        wchan_wakeone(sem->sem_wchan, &sem->sem_lock);

        spinlock_release(&sem->sem_lock);
}
```

Conditions MUST be true

Conditions MUST be true

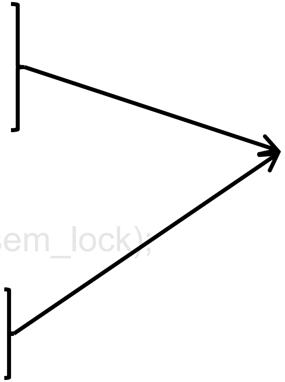
Semaphores (kern/thread/synch.c)

```
void V(struct semaphore *sem)
{
        KASSERT(sem != NULL);

        spinlock_acquire(&sem->sem_lock);

        sem->sem_count++;
        KASSERT(sem->sem_count > 0);
        wchan_wakeone(sem->sem_wchan, &sem->sem_lock);

        spinlock_release(&sem->sem_lock);
}
```



Increment counter under lock

Semaphores (kern/thread/synch.c)

```
void V(struct semaphore *sem)
{
        KASSERT(sem != NULL);

        spinlock_acquire(&sem->sem_lock);

        sem->sem_count++;
        KASSERT(sem->sem_count > 0);
        wchan_wakeone(sem->sem_wchan, &sem->sem_lock);

        spinlock_release(&sem->sem_lock);
}
```

Wake one sleeping thread

Wait channel

- wchan (we have seen it in action)
- Let’s threads wait on a certain event
- Include a lock and a queue
- Does this sound familiar?

Wait channel

- wchan (we have seen it in action)
- Let’s threads wait on a certain event
- Include a lock and a queue
- Does this sound familiar?

May be useful to help you build the condition variable primitive in lab 6

University of BRISTOL

Thank you

bristol.ac.uk