Semaphore P() implementation

```
P(struct semaphore* sem) {
   spinlock_acquire(&sem->sem_lock);
   while (sem->sem_count == 0) {
       wchan_lock(sem->sem_wchan);
       spinlock_release(&sem->sem_lock);
       wchan_sleep(sem->sem_wchan);
       spinlock_acquire(&sem->sem_lock);
   }
   sem->sem_count--;
   spinlock_release(&sem->sem_lock);
}
```

What if we swapped lines 4 and 5?

```
1   P(struct semaphore* sem) {
2      spinlock_acquire(&sem->sem_lock);
3      while (sem->sem_count == 0) {
5          spinlock_release(&sem->sem_lock);
4          wchan_lock(sem->sem_wchan);
6          wchan_sleep(sem->sem_wchan);
7          spinlock_acquire(&sem->sem_lock);
8      }
9      sem->sem_count--;
10     spinlock_release(&sem->sem_lock);
11  }
```

Incorrect implementation

Semaphore count is initially 0

```
1   P(struct semaphore* sem) {
2      spinlock_acquire(&sem->sem_lock);
3      while (sem->sem_count == 0) {
5          spinlock_release(&sem->sem_lock);
4          wchan_lock(sem->sem_wchan);
6          wchan_sleep(sem->sem_wchan);
7          spinlock_acquire(&sem->sem_lock);
8      }
9      sem->sem_count--;
10     spinlock_release(&sem->sem_lock);
11  }

1   V(struct semaphore* sem) {
2       spinlock_acquire(&sem->sem_lock);
3       sem->sem_count++;
4       wchan_wakeone(sem->sem_wchan);
5       spinlock_release(&sem->sem_lock);
6   }
```

Thread 1

Calls P()

spinlock_release(…)

Thread 2

Incorrect implementation

Semaphore count is now 1

```
1   P(struct semaphore* sem) {
2      spinlock_acquire(&sem->sem_lock);
3      while (sem->sem_count == 0) {
5          spinlock_release(&sem->sem_lock);
4          wchan_lock(sem->sem_wchan);
6          wchan_sleep(sem->sem_wchan);
7          spinlock_acquire(&sem->sem_lock);
8      }
9      sem->sem_count--;
10     spinlock_release(&sem->sem_lock);
11  }

1   V(struct semaphore* sem) {
2       spinlock_acquire(&sem->sem_lock);
3       sem->sem_count++;
4       wchan_wakeone(sem->sem_wchan);
5       spinlock_release(&sem->sem_lock);
6   }
```

Thread 1

Calls P()

`spinlock_release(…)`

Either context switch or just very slow execution (e.g., pipeline stall, cache miss, etc.)

Thread 2

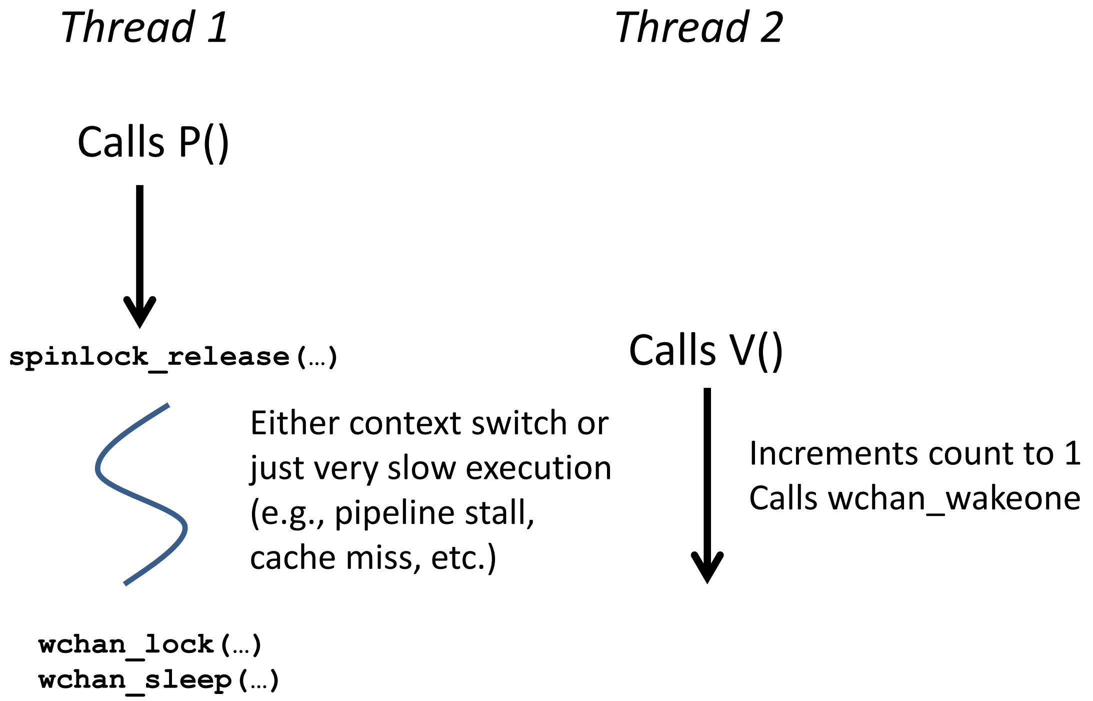
Calls V()

Increments count to 1
Calls wchan_wakeone

Incorrect implementation

Semaphore count is now 1

```
1  P(struct semaphore* sem) {
2     spinlock_acquire(&sem->sem_lock);
3     while (sem->sem_count == 0) {
5         spinlock_release(&sem->sem_lock);
4         wchan_lock(sem->sem_wchan);
6         wchan_sleep(sem->sem_wchan);
7         spinlock_acquire(&sem->sem_lock);
8     }
9     sem->sem_count--;
10    spinlock_release(&sem->sem_lock);
11 }

1  V(struct semaphore* sem) {
2      spinlock_acquire(&sem->sem_lock);
3      sem->sem_count++;
4      wchan_wakeone(sem->sem_wchan);
5      spinlock_release(&sem->sem_lock);
6  }
```



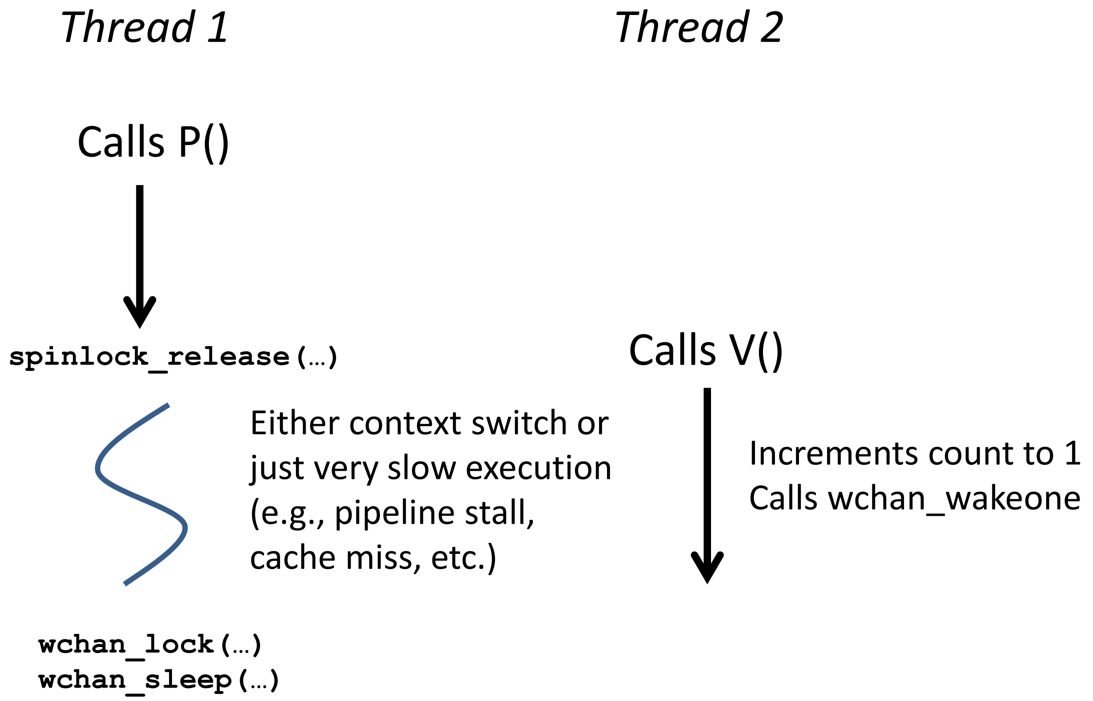
Either context switch or just very slow execution (e.g., pipeline stall, cache miss, etc.)

Increments count to 1
Calls wchan_wakeone

Incorrect implementation

Semaphore count is now 1

```
1   P(struct semaphore* sem) {
2      spinlock_acquire(&sem->sem_lock);
3      while (sem->sem_count == 0) {
5          spinlock_release(&sem->sem_lock);
4          wchan_lock(sem->sem_wchan);
6          wchan_sleep(sem->sem_wchan);
7          spinlock_acquire(&sem->sem_lock);
8      }
9      sem->sem_count--;
10     spinlock_release(&sem->sem_lock);
11  }

1   V(struct semaphore* sem) {
2       spinlock_acquire(&sem->sem_lock);
3       sem->sem_count++;
4       wchan_wakeone(sem->sem_wchan);
5       spinlock_release(&sem->sem_lock);
6   }
```



Thread 1 misses the wake up signal and blocks on P() even though count is 1.

Correct implementation

Semaphore count is initially 0

```
P(struct semaphore* sem) {
   spinlock_acquire(&sem->sem_lock);
   while (sem->sem_count == 0) {
       wchan_lock(sem->sem_wchan);
       spinlock_release(&sem->sem_lock);
       wchan_sleep(sem->sem_wchan);
       spinlock_acquire(&sem->sem_lock);
   }
   sem->sem_count--;
   spinlock_release(&sem->sem_lock);
}
```

```
V(struct semaphore* sem) {
    spinlock_acquire(&sem->sem_lock);
    sem->sem_count++;
    wchan_wakeone(sem->sem_wchan);
    spinlock_release(&sem->sem_lock);
}
```

Thread 1

Calls P()

↓

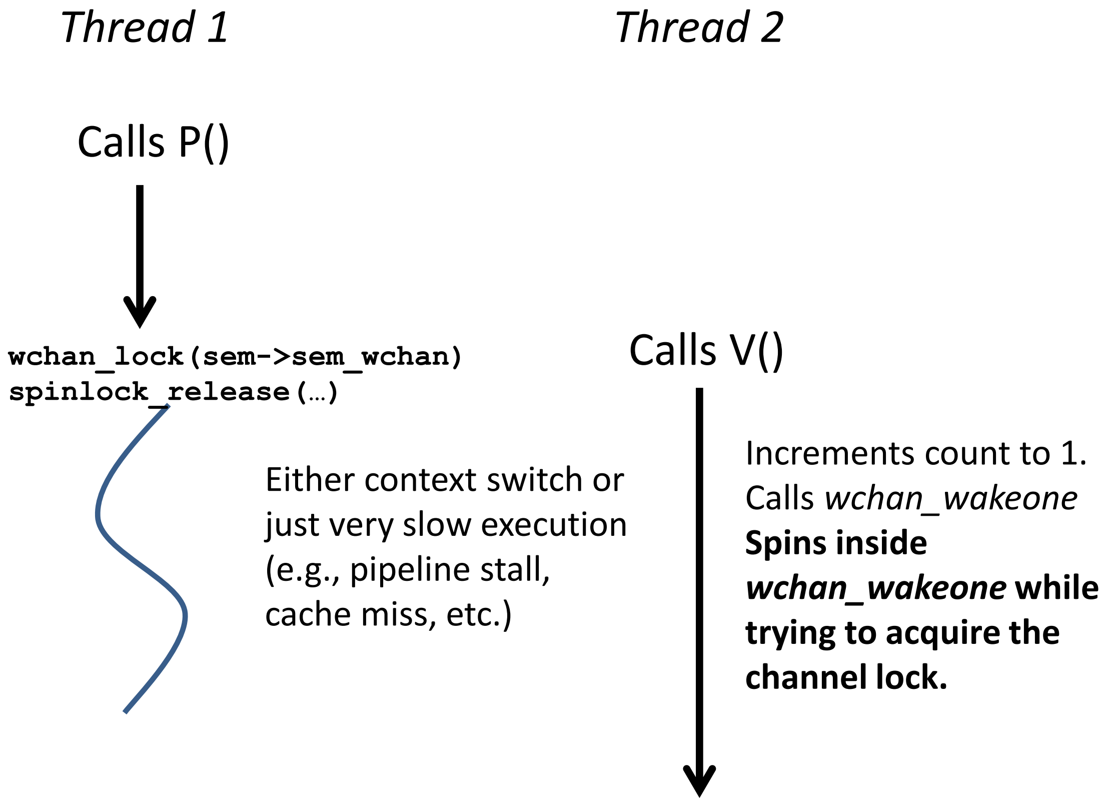
```
wchan_lock(sem->sem_wchan)
spinlock_release(…)
```

Thread 2

Correct implementation

Semaphore count is now 1

```
P(struct semaphore* sem) {
   spinlock_acquire(&sem->sem_lock);
   while (sem->sem_count == 0) {
       wchan_lock(sem->sem_wchan);
       spinlock_release(&sem->sem_lock);
       wchan_sleep(sem->sem_wchan);
       spinlock_acquire(&sem->sem_lock);
   }
   sem->sem_count--;
   spinlock_release(&sem->sem_lock);
}

V(struct semaphore* sem) {
    spinlock_acquire(&sem->sem_lock);
    sem->sem_count++;
    wchan_wakeone(sem->sem_wchan);
    spinlock_release(&sem->sem_lock);
}
```



Thread 1

Calls P()

`wchan_lock(sem->sem_wchan)`
`spinlock_release(…)`

Either context switch or just very slow execution (e.g., pipeline stall, cache miss, etc.)

Thread 2

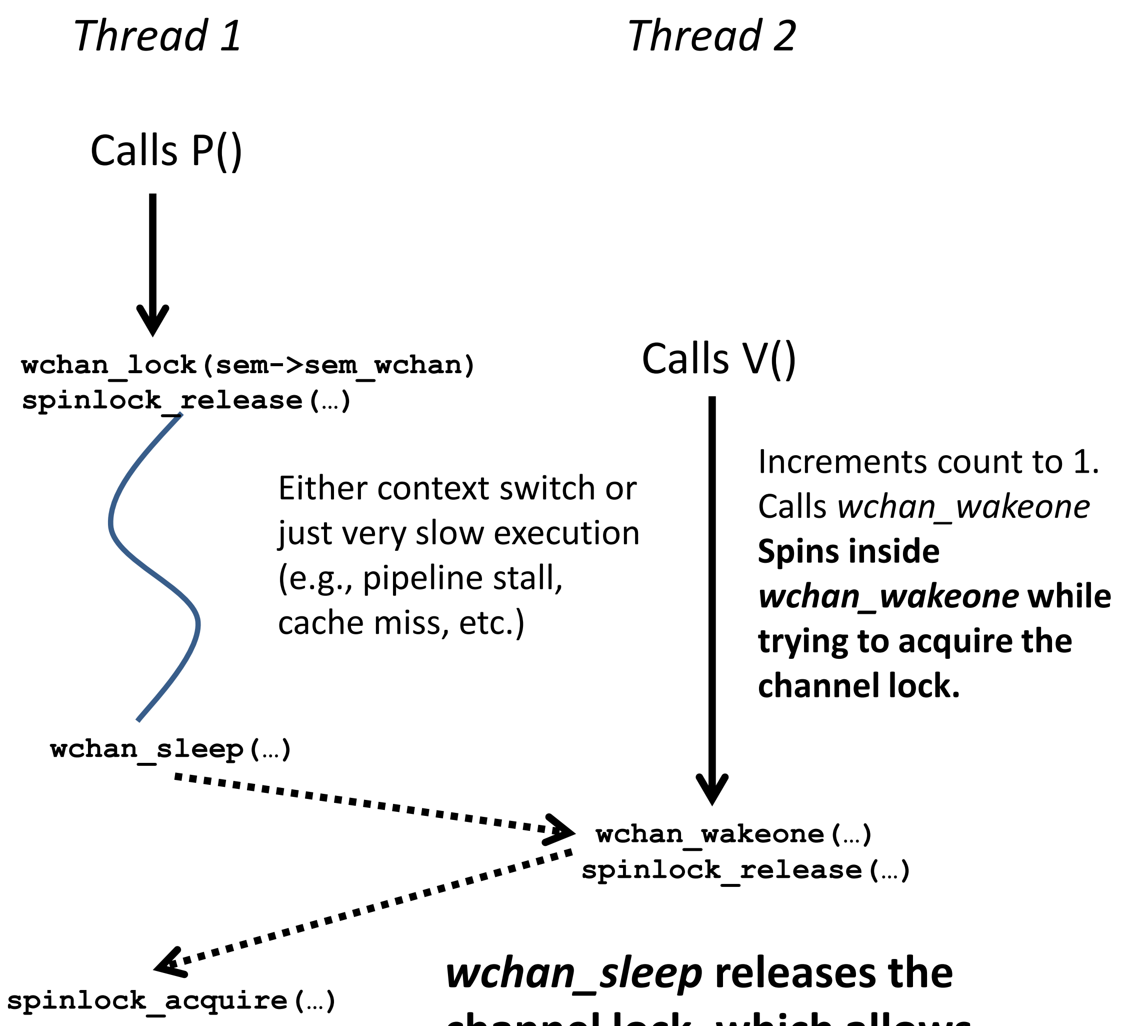
Calls V()

Increments count to 1.
Calls *wchan_wakeone*
Spins inside *wchan_wakeone* while trying to acquire the channel lock.

Correct implementation

Semaphore count is now 1

```
P(struct semaphore* sem) {
   spinlock_acquire(&sem->sem_lock);
   while (sem->sem_count == 0) {
       wchan_lock(sem->sem_wchan);
       spinlock_release(&sem->sem_lock);
       wchan_sleep(sem->sem_wchan);
       spinlock_acquire(&sem->sem_lock);
   }
   sem->sem_count--;
   spinlock_release(&sem->sem_lock);
}

V(struct semaphore* sem) {
    spinlock_acquire(&sem->sem_lock);
    sem->sem_count++;
    wchan_wakeone(sem->sem_wchan);
    spinlock_release(&sem->sem_lock);
}
```



Either context switch or just very slow execution (e.g., pipeline stall, cache miss, etc.)

Increments count to 1. Calls *wchan_wakeone* **Spins inside *wchan_wakeone* while trying to acquire the channel lock.**

***wchan_sleep* releases the channel lock, which allows *wchan_wakeone* to complete. This in turn wakes up thread 1.**

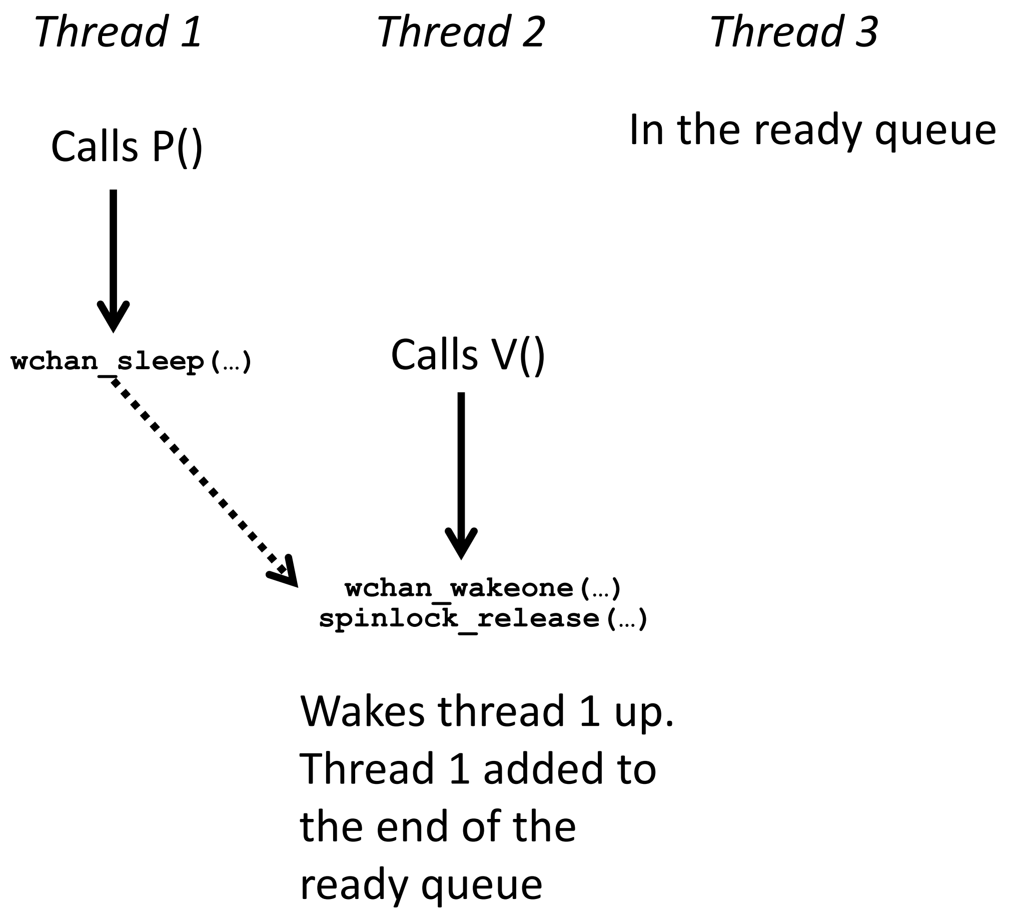
However, our implementation does not provide FIFO ordering.

```
P(struct semaphore* sem) {
   spinlock_acquire(&sem->sem_lock);
   while (sem->sem_count == 0) {
       wchan_lock(sem->sem_wchan);
       spinlock_release(&sem->sem_lock);
       wchan_sleep(sem->sem_wchan);
       spinlock_acquire(&sem->sem_lock);
   }
   sem->sem_count--;
   spinlock_release(&sem->sem_lock);
}

V(struct semaphore* sem) {
    spinlock_acquire(&sem->sem_lock);
    sem->sem_count++;
    wchan_wakeone(sem->sem_wchan);
    spinlock_release(&sem->sem_lock);
}
```

Thread 1	*Thread 2*	*Thread 3*
Calls P()		In the ready queue
↓		
`wchan_sleep(…)`		

However, our implementation does not provide FIFO ordering.

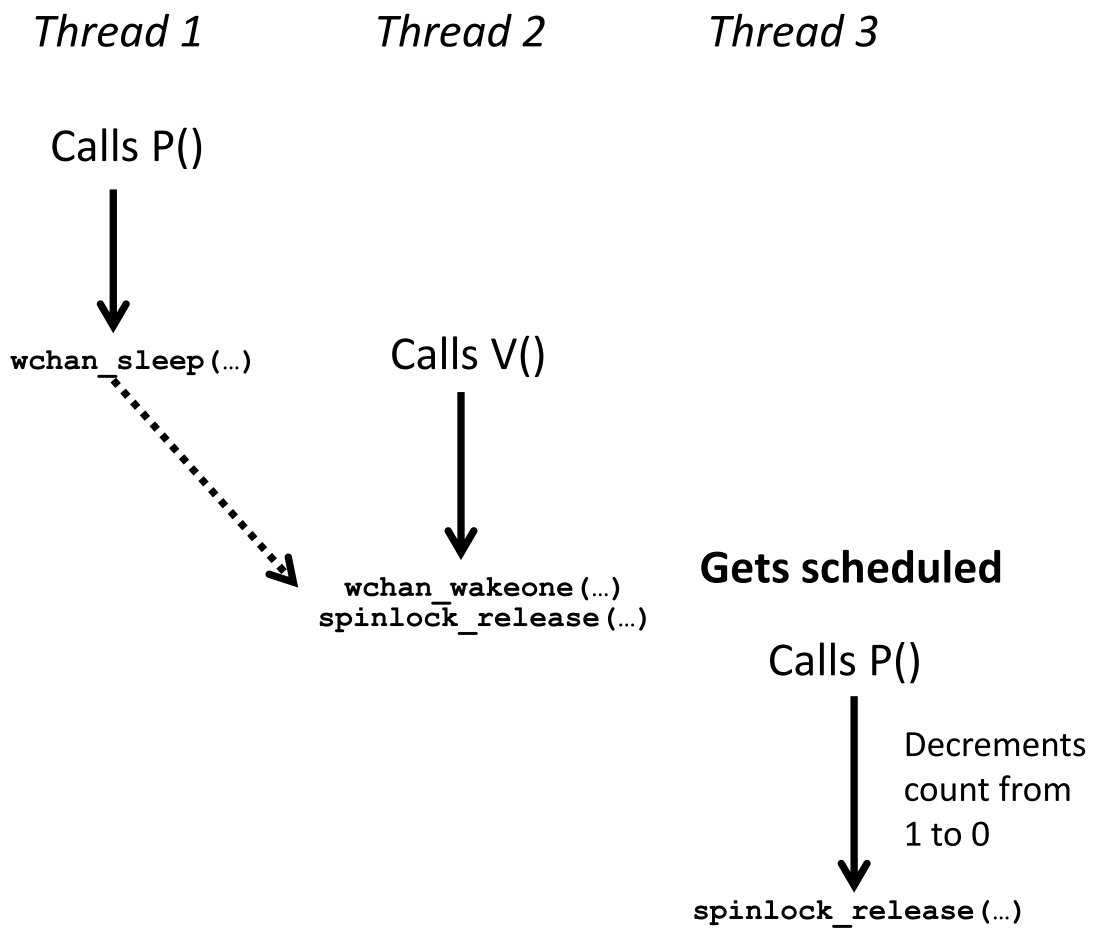
```
P(struct semaphore* sem) {
   spinlock_acquire(&sem->sem_lock);
   while (sem->sem_count == 0) {
       wchan_lock(sem->sem_wchan);
       spinlock_release(&sem->sem_lock);
       wchan_sleep(sem->sem_wchan);
       spinlock_acquire(&sem->sem_lock);
   }
   sem->sem_count--;
   spinlock_release(&sem->sem_lock);
}

V(struct semaphore* sem) {
    spinlock_acquire(&sem->sem_lock);
    sem->sem_count++;
    wchan_wakeone(sem->sem_wchan);
    spinlock_release(&sem->sem_lock);
}
```



However, our implementation does not provide FIFO ordering.

```
P(struct semaphore* sem) {
   spinlock_acquire(&sem->sem_lock);
   while (sem->sem_count == 0) {
       wchan_lock(sem->sem_wchan);
       spinlock_release(&sem->sem_lock);
       wchan_sleep(sem->sem_wchan);
       spinlock_acquire(&sem->sem_lock);
   }
   sem->sem_count--;
   spinlock_release(&sem->sem_lock);
}
```

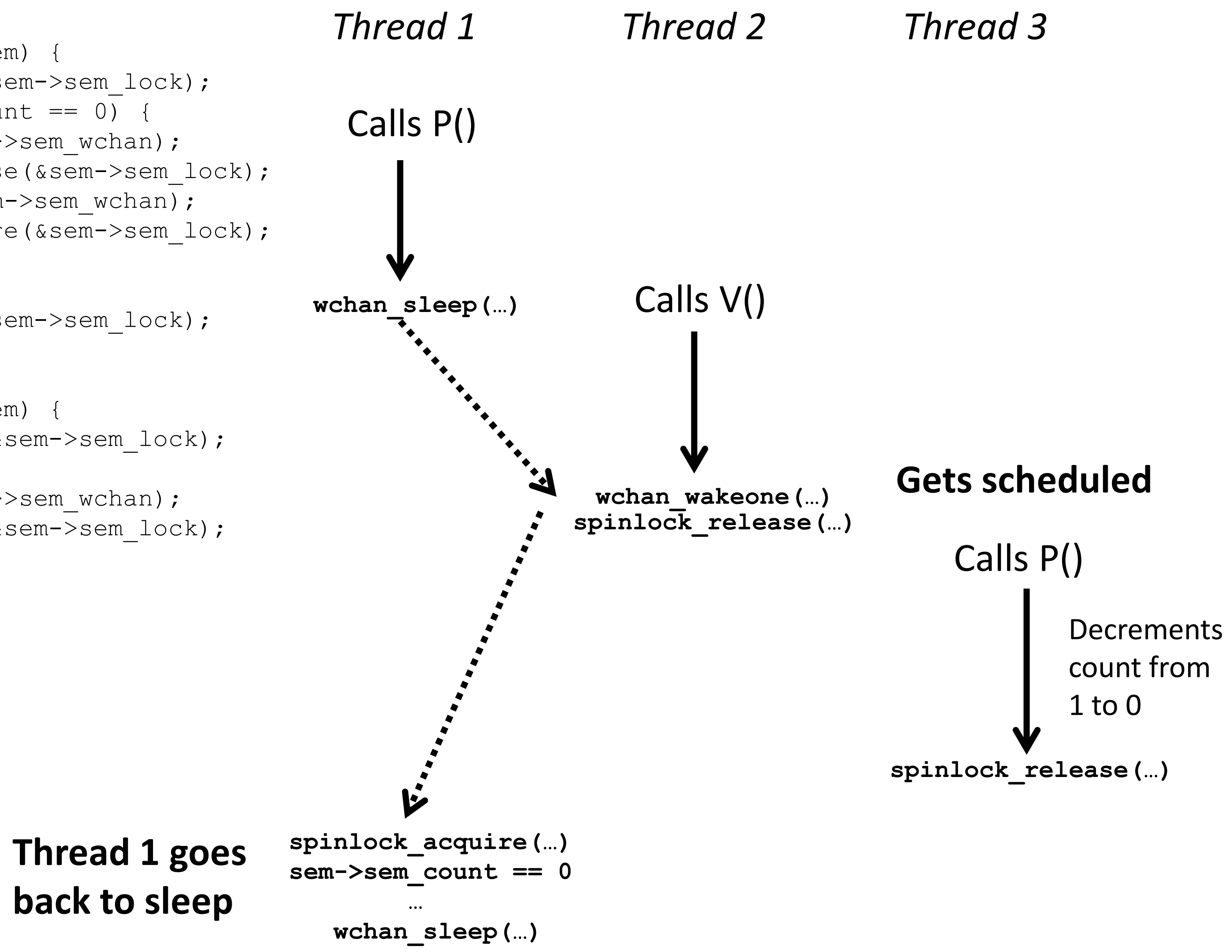
```
V(struct semaphore* sem) {
    spinlock_acquire(&sem->sem_lock);
    sem->sem_count++;
    wchan_wakeone(sem->sem_wchan);
    spinlock_release(&sem->sem_lock);
}
```



However, our implementation does not provide FIFO ordering.

```
P(struct semaphore* sem) {
   spinlock_acquire(&sem->sem_lock);
   while (sem->sem_count == 0) {
       wchan_lock(sem->sem_wchan);
       spinlock_release(&sem->sem_lock);
       wchan_sleep(sem->sem_wchan);
       spinlock_acquire(&sem->sem_lock);
   }
   sem->sem_count--;
   spinlock_release(&sem->sem_lock);
}
```

```
V(struct semaphore* sem) {
    spinlock_acquire(&sem->sem_lock);
    sem->sem_count++;
    wchan_wakeone(sem->sem_wchan);
    spinlock_release(&sem->sem_lock);
}
```



Thread 1
Thread 2
Thread 3
Calls P()
wchan_sleep(…)
Calls V()
wchan_wakeone(…)
spinlock_release(…)
Gets scheduled
Calls P()
Decrements count from 1 to 0
spinlock_release(…)
Thread 1 goes back to sleep
spinlock_acquire(…)
sem->sem_count == 0
…
wchan_sleep(…)