Locking in xv6

Mythili Vutukuru

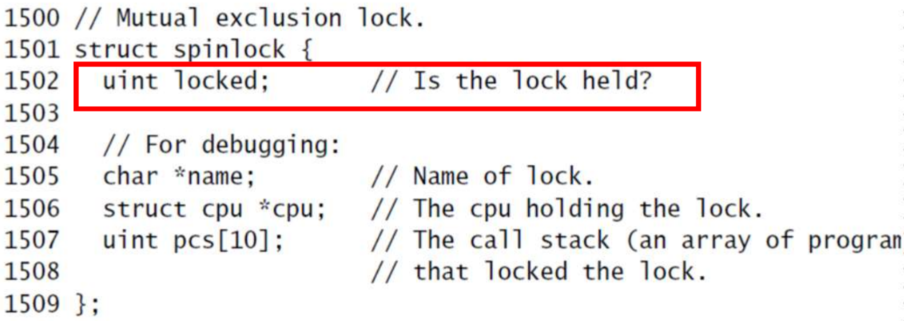
CSE, IIT Bombay

Locking in xv6

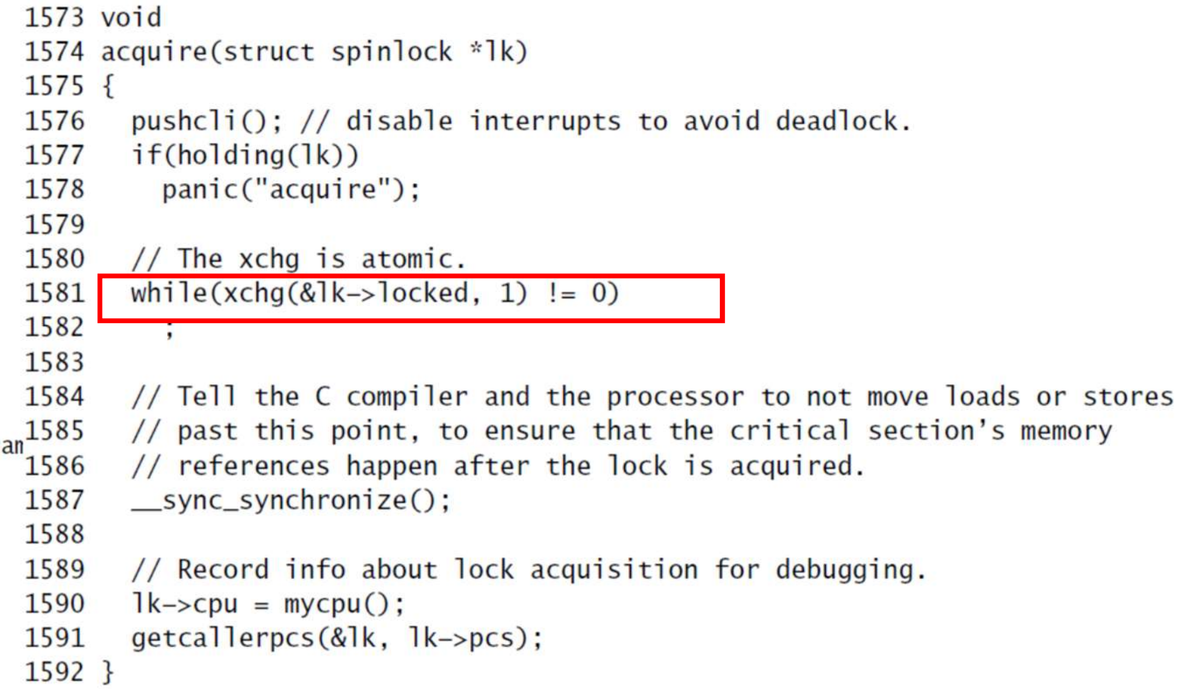
- No threads in xv6, no two user programs can access same memory image
  - No need for userspace locks like pthreads mutex
- However, scope for concurrency in xv6 kernel
  - Two processes in kernel mode in different CPUs can access same kernel data structures like ptable
  - Even in single core, when a process is running in kernel mode, another trap occurs, trap handler can access data that was being accessed by previous kernel code
- Solution: spinlocks used to protect critical sections
  - Limit concurrent access to kernel data structures that can result in race conditions
- xv6 also has a sleeping lock (built on spinlock, not discussed)

Spinlocks in xv6

- Acquiring lock: uses xchg x86 atomic instruction (test and set)
  - Atomically set lock variable to new value and returns previous value
  - If previous value is 0, it means free lock has been acquired, success!
  - If previous value is 1, it means lock is held by someone, continue to spin in a busy while loop till success



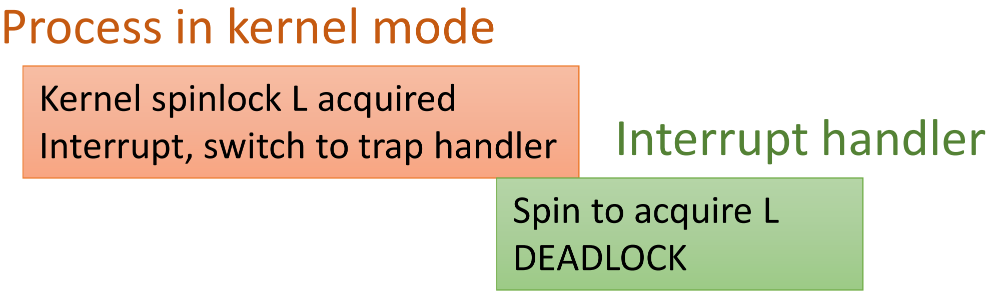
```
// Mutual exclusion lock.
struct spinlock {
  uint locked;       // Is the lock held?

  // For debugging:
  char *name;        // Name of lock.
  struct cpu *cpu;   // The cpu holding the lock.
  uint pcs[10];      // The call stack (an array of program
                     // that locked the lock.
};
```



```
void
acquire(struct spinlock *lk)
{
  pushcli(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");

  // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0)
    ;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that the critical section's memory
  // references happen after the lock is acquired.
  __sync_synchronize();

  // Record info about lock acquisition for debugging.
  lk->cpu = mycpu();
  getcallerpcs(&lk, lk->pcs);
}
```

Disabling interrupts for kernel spinlocks (1)

- When acquiring kernel spinlock, disables interrupts on CPU core: why?
  - What if interrupt and handler requests same lock: deadlock
  - Interrupts disabled only on local core, OK to spin for lock on another core
  - Why disable interrupts before even acquiring lock? (otherwise, vulnerable window after lock acquired and before interrupts disabled)
- Disabling interrupts not needed for userspace locks like pthread mutex
  - Kernel interrupt handlers will not deadlock for userspace locks

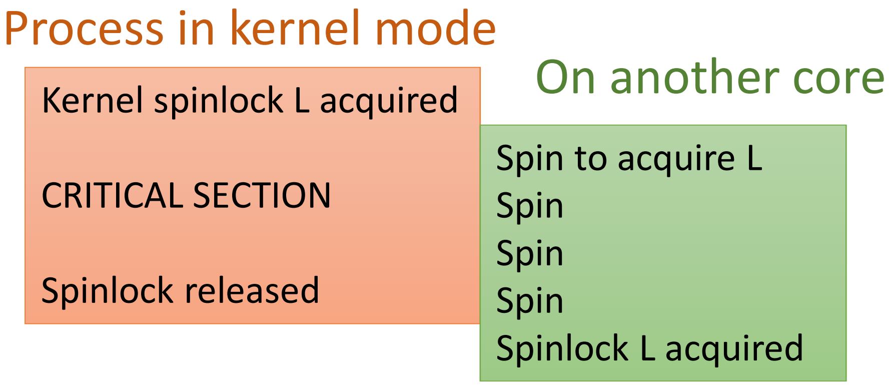


Process in kernel mode

Kernel spinlock L acquired
Interrupt, switch to trap handler

Interrupt handler

Spin to acquire L
DEADLOCK



Process in kernel mode

Kernel spinlock L acquired

CRITICAL SECTION

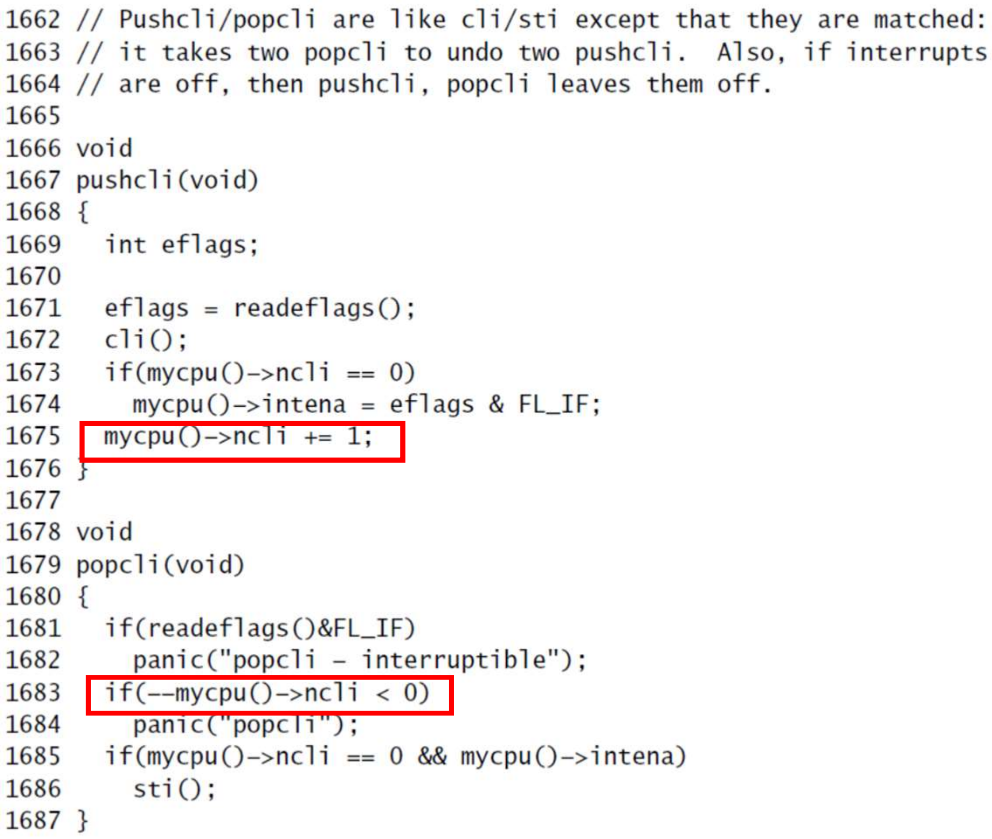
Spinlock released

On another core

Spin to acquire L
Spin
Spin
Spin
Spinlock L acquired

Disabling interrupts for kernel spinlocks (2)

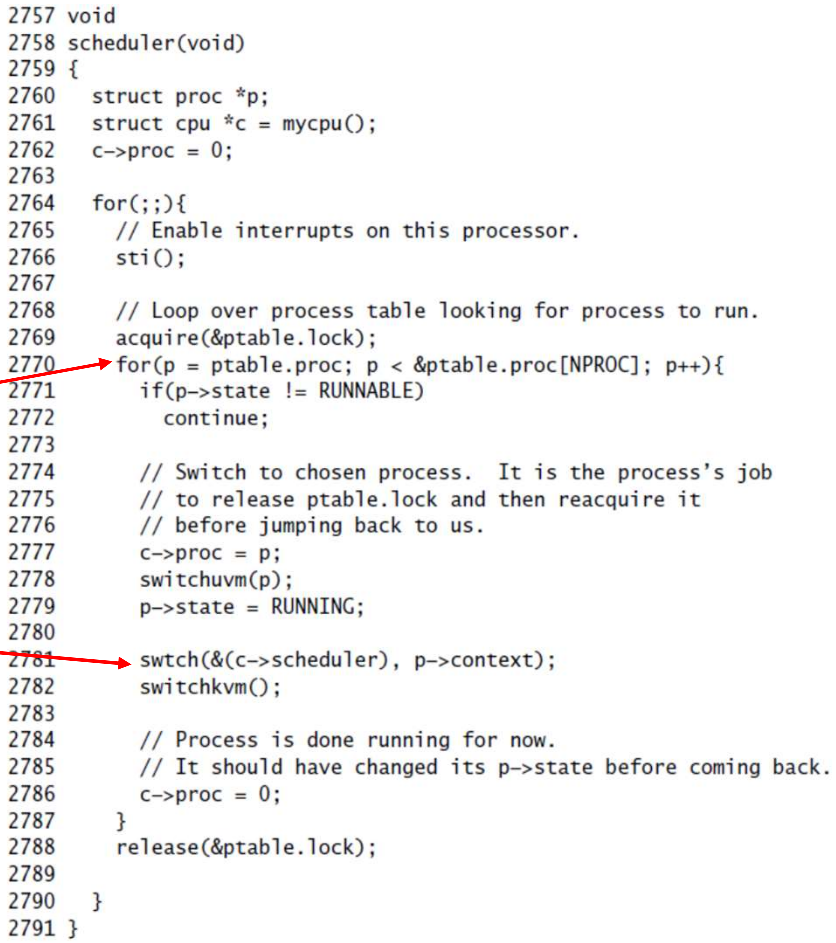
- Function pushcli: disables interrupts on CPU core before spinning for lock
  - Interrupts stay disabled until lock is released
- What if multiple spinlocks are acquired?
  - Interrupts must stay disabled until all locks are released
- Disabling/enabling interrupts:
  - pushcli disables interrupts on first lock acquire, increments count for future locks
  - popcli decrements count, renables interrupts only when all locks released



```
// Pushcli/popcli are like cli/sti except that they are matched:
// it takes two popcli to undo two pushcli.  Also, if interrupts
// are off, then pushcli, popcli leaves them off.

void
pushcli(void)
{
  int eflags;

  eflags = readeflags();
  cli();
  if(mycpu()->ncli == 0)
    mycpu()->intena = eflags & FL_IF;
  mycpu()->ncli += 1;
}

void
popcli(void)
{
  if(readeflags()&FL_IF)
    panic("popcli - interruptible");
  if(--mycpu()->ncli < 0)
    panic("popcli");
  if(mycpu()->ncli == 0 && mycpu()->intena)
    sti();
}
```

Recap: Context switching in xv6 (1)

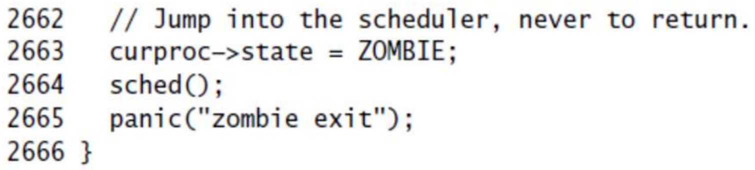
- Every CPU has a scheduler thread (special process that runs scheduler code)
- Scheduler goes over list of processes and switches to one of the runnable ones
- The special function “swtch” performs the actual context switch
  - Save context on kernel stack of old process
  - Restore context from kernel stack of new process



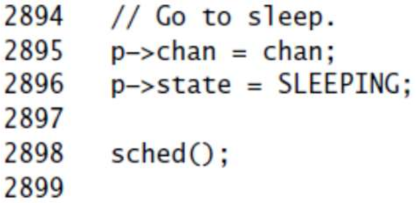
```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

      swtch(&(c->scheduler), p->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
    }
    release(&ptable.lock);

  }
}
```

Recap: Context switching in xv6 (2)

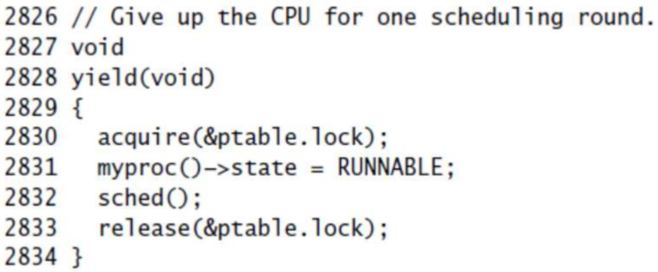
- After running for some time, the process switches back to the scheduler thread, when:
  - Process has terminated (exit system call)
  - Process needs to sleep (e.g., blocking read system call)
  - Process yields after running for long (timer interrupt)
- Process calls “sched” which calls “swtch” to switch to scheduler thread again
- Scheduler thread runs its loop and picks next process to run, and the story repeats



```
  // Jump into the scheduler, never to return.
  curproc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

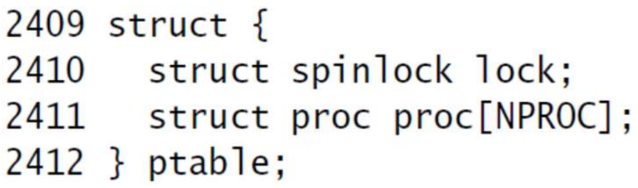


```
  // Go to sleep.
  p->chan = chan;
  p->state = SLEEPING;

  sched();

```



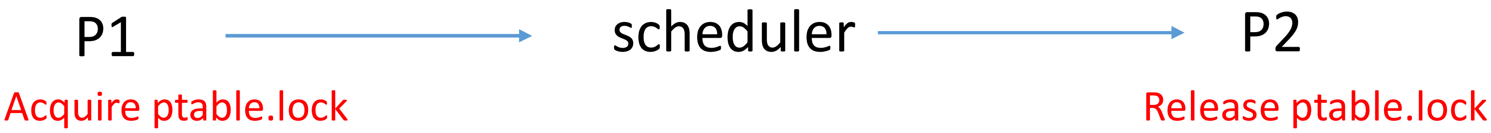
```
// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

ptable.lock (1)



```
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

- The process table protected by a lock, any access to ptable must be done with ptable.lock held
- Normally, a process in kernel mode acquires ptable.lock, changes ptable in some way, releases lock
  - Example: when allocproc allocates new struct proc
- But during context switch from process P1 to P2, ptable structure is being changed all through context switch, so when to release lock?
  - P1 acquires lock, switches to scheduler, switches to P2, P2 releases lock



P1

scheduler

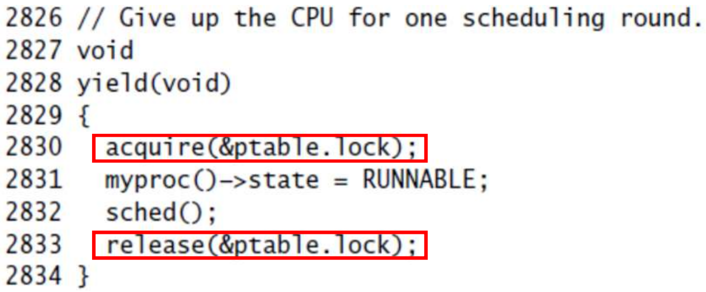
P2

Acquire ptable.lock

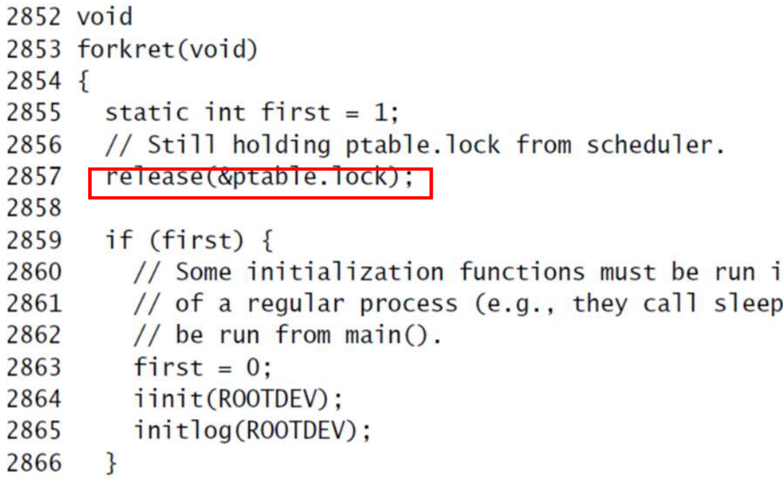
Release ptable.lock

ptable.lock (2)

- Every function that calls sched() to give up CPU will do so with ptable.lock held
- Which functions invoke sched() to give up CPU?
  - Yield: process gives up CPU due to timer interrupt
  - Sleep: when process wishes to block
  - Exit: when process terminates
- Every function where a process resumes after being scheduled release ptable.lock
- What functions does a process resume after swtch?
  - Yield: resuming process after yield is done
  - Sleep: resuming process that is waking up after sleep
  - Forkret: for newly created processes
- Purpose of forkret: to release ptable.lock
  - New process then returns from trap like its parent

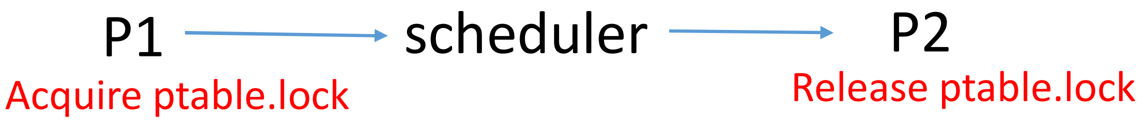


```
// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```



```
void
forkret(void)
{
  static int first = 1;
  // Still holding ptable.lock from scheduler.
  release(&ptable.lock);

  if (first) {
    // Some initialization functions must be run i
    // of a regular process (e.g., they call sleep
    // be run from main().
    first = 0;
    iinit(ROOTDEV);
    initlog(ROOTDEV);
  }
```

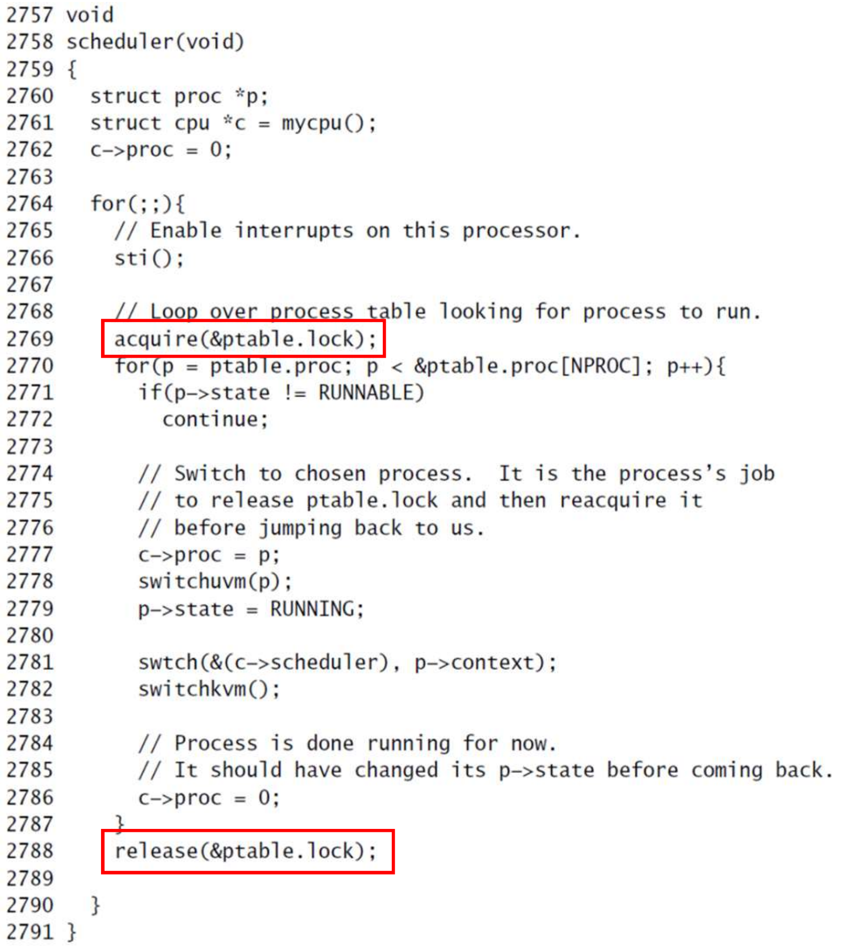
ptable.lock (3)


P1 → scheduler → P2

Acquire ptable.lock

Release ptable.lock

- Scheduler goes into loop with lock held
- Acquire ptable.lock in P1 → scheduler picks P2 → release in P2
- Later, acquire ptable.lock in P2 → scheduler picks P3 → release in P3
- Periodically, end of looping over all processes, releases lock temporarily
  - What if no runnable process found due to interrupts being disabled? Release lock, enable interrupts, allow processes to become runnable.


```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

      swtch(&(c->scheduler), p->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
    }
    release(&ptable.lock);

  }
}
```