

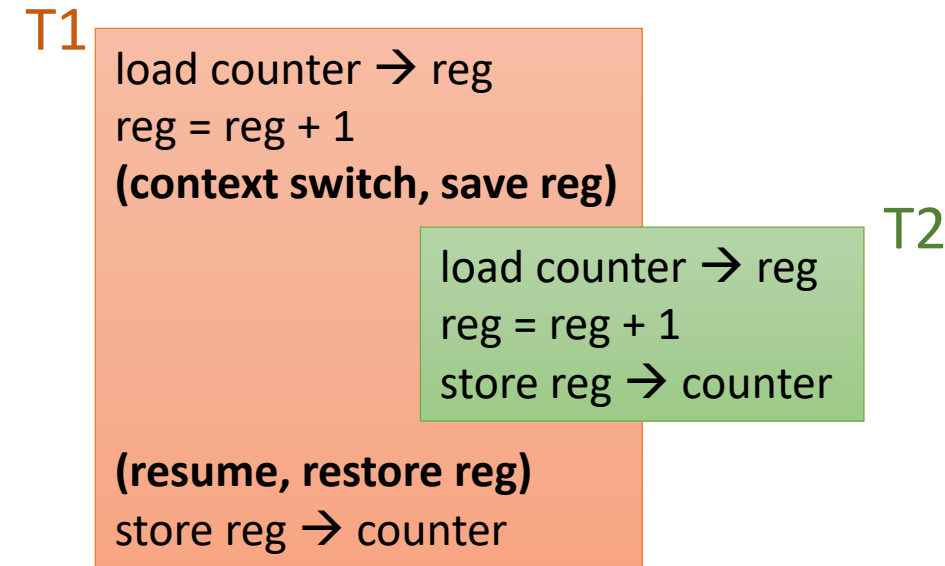
Locking

Mythili Vutukuru
CSE, IIT Bombay

Recap: Shared data access in threads

```
load counter → reg  
reg = reg + 1  
store reg → counter
```

- The C code “counter = counter + 1” is compiled into multiple instructions
 - Load counter variable from memory into register
 - Increment register
 - Store register back into memory of counter variable
- What happens when two threads run this line of code concurrently?
 - Counter is 0 initially
 - T1 loads counter into register, increment reg
 - Context switch, register (value 1) saved
 - T2 runs, loads counter 0 from memory
 - T2 increments register, stores to memory
 - T1 resumes, stores register value to counter
 - Counter value rewritten to 1 again
 - Final counter value is 1, expected value is 2



Recap: Race conditions, critical sections

- Incorrect execution of code due to concurrency is called **race condition**
 - Due to unfortunate timing of context switches, atomicity of data update violated
- Race conditions happen when we have **concurrent execution on shared data**
 - **Threads** sharing common data in memory image of user processes
 - Processes in kernel mode sharing **OS data structures**
- We require **mutual exclusion** on some parts of user or OS code
 - Concurrent execution by multiple threads/processes should not be permitted
- Parts of program that need to be executed with mutual exclusion for correct operation are called **critical sections**
 - Present in multi-threaded programs, OS code
- How to access critical sections with mutual exclusion? Using locks

Using locks

- Locks are special variables that provide mutual exclusion
 - Provided by threading libraries
 - Can call **lock/acquire** and **unlock/release** functions on a lock
- When a thread T1 acquires a lock, another thread T2 cannot acquire same lock
 - Execution of T2 stops at the lock statement
 - T2 can proceed only after T1 releases the lock
- Acquire lock → critical section → release lock ensures mutual exclusion in critical section

```
int counter;
pthread_mutex_t m;

void start_fn() {

    for(int i=0; i < 1000; i++) {
        pthread_mutex_lock(&m)
        counter = counter + 1
        pthread_mutex_unlock(&m)
    }

main() {
    counter = 0

    pthread_t t1, t2
    pthread_create(&t1,.., start_fn, ..)
    pthread_create(&t2, .., start_fn,..)

    pthread_join(t1, ..)
    pthread_join(t2, ..)

    print counter
}
```

How to implement a lock?

- Goals of a lock implementation
 - Mutual exclusion (obviously!)
 - Fairness: all threads should eventually get the lock, and no thread should starve
 - Low overhead: acquiring, releasing, and waiting for lock should not consume too many resources
- Implementation of locks are needed for both userspace programs (e.g., pthreads library) and kernel code
 - Separate implementations in user libraries and OS

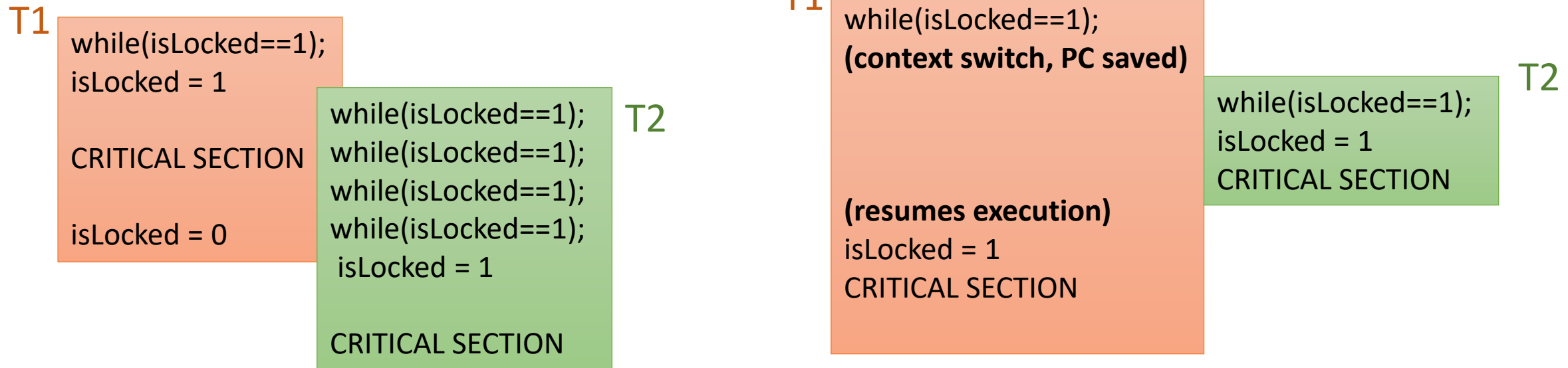
Incorrect lock implementation

- Example of incorrect lock implementation
 - Use variable `isLocked` to indicate lock status (0 means lock is free, 1 indicates it is acquired)
 - To acquire lock, a thread waits as long as lock is busy, and then sets it to 1 (acquired)
 - One interleaving of executions (left) works while another (right) may not work

```
int isLocked = 0

void acquire_lock() {
    while(isLocked == 1); //wait
    isLocked = 1
}

void release_lock() {
    isLocked = 0
}
```



Hardware atomic instructions

- Need a way to check a variable and set its value atomically
 - No context switch between checking lock variable and setting it
 - But user programs have no control over context switches
- Solution: use **hardware atomic instructions**
- Example: **test-and-set** hardware atomic instruction
 - Two arguments: address of variable and new value to set
 - Writes new value into a variable and returns old value in one single step
 - Entire logic implemented in hardware, runs in one single step

```
1      int TestAndSet(int *old_ptr, int new) {  
2          int old = *old_ptr; // fetch old value at old_ptr  
3          *old_ptr = new;      // store 'new' into old_ptr  
4          return old;          // return the old value  
5      }
```

Lock implementation using test-and-set

- Simple lock can be implemented using test-and-set instruction
 - isLocked variable indicates lock status (0=free, 1=acquired)
 - If test-and-set(&isLocked, 1) returns 1, it means lock is not free, wait
 - If test-and-set(&isLocked, 1) returns 0, lock was free and was acquired, done!
- No further race conditions possible with this lock implementation
 - All modern lock implementations based on such hardware instructions
 - Software based locking algorithms do not work well in modern systems

```
int isLocked = 0
```

```
void acquire_lock() {  
    while(test-and-set(&isLocked, 1) == 1); //wait  
    //return, lock is acquired  
}
```



```
1  typedef struct __lock_t {
2      int flag;
3  } lock_t;
4
5  void init(lock_t *lock) {
6      // 0 indicates that lock is available, 1 that it is held
7      lock->flag = 0;
8  }
9
10 void lock(lock_t *lock) {
11     while (TestAndSet(&lock->flag, 1) == 1)
12         ; // spin-wait (do nothing)
13 }
14
15 void unlock(lock_t *lock) {
16     lock->flag = 0;
17 }
```

Figure 28.3: A Simple Spin Lock Using Test-and-set

Another instruction: compare-and-swap

- Another example: **compare-and-swap** (CAS) hardware atomic instruction
 - Three arguments: address of variable, expected old value, new value
 - If variable has expected old value, then write new value and return true; else do not change variable and return false

```
1  int CompareAndSwap(int *ptr, int expected, int new) {  
2      int actual = *ptr;  
3      if (actual == expected)  
4          *ptr = new;  
5      return actual;  
6  }
```

Figure 28.4: **Compare-and-swap**

Lock using CAS

- Lock implementation using compare-and-swap
 - If compare-and-swap(&isLocked, 0, 1) returns false, it means lock is busy, wait
 - If compare-and-swap(&isLocked, 0, 1) returns true, it means old value of lock was 0 and was changed to 1, so lock has been acquired, done!

```
int isLocked = 0

void acquire_lock() {
    while(compare-and-swap(&isLocked, 0, 1) == false); //wait
}
```

Evaluating spinlock implementations

- Correctness: does it lead to mutual exclusion correctly?
- Fairness: are all waiting threads treated fairly? Can we guarantee that every waiting thread will get its turn?
 - The implementations we saw here do not guarantee it
- Performance: overheads of having threads spin for lock
 - Single core system: what happens when thread holding lock is context switched out and other threads that are scheduled continue to spin for lock?
 - Problem less severe in multicore system. Why? (Thread holding lock can finish while other threads are spinning)

Spinlock vs. sleeping mutex

- Simple lock implementation seen here is a **spinlock**
 - If thread T1 has acquired lock, and thread T2 also wants lock, then T2 will keep spinning in a while loop till lock is free
- Another implementation option: thread can go to sleep (be blocked) while waiting for lock, saving CPU cycles
 - OS blocks waiting thread, context switch to another thread/process
 - Such locks are called **(sleeping) mutex**
- Threading libraries provide APIs for both spinlocks and sleeping mutex
 - Better to use spinlock if locks are expected to be held for short time, avoid context switch overhead
 - Better to use sleeping mutex if critical sections are long

Guidelines for using locks

- When writing multithreaded programs, careful **locking discipline**
 - Protect each shared data structure with one lock
 - Locks can be **coarse-grained** (one big fat lock) or **fine-grained** (many smaller locks)
 - Any thread wanting to access shared data must acquire corresponding lock before access, release lock after access
- If using third-party libraries in multi-threaded programs, check the documentation to see if the library is **thread-safe**
 - Thread-safe implementations work correctly with concurrent access

Guidelines for using locks

- Good practice to acquire locks for both **reading and writing data**
 - Why locks for reading? We do not want to read incorrect data while another thread is concurrently updating the data
 - Some libraries provide separate locks for reading and writing, allowing multiple threads to concurrently read data if no other thread is writing
- Good practice to minimize use of locks, use only when needed
 - Why? Use of locks serializes thread access, removes gains due to parallelism
 - Example of minimizing lock usage: instead of each thread updating shared global counter, let each thread update a local counter, and periodically update global counter

```
1  typedef struct __counter_t {
2      int          value;
3      pthread_mutex_t lock;
4  } counter_t;
5
6  void init(counter_t *c) {
7      c->value = 0;
8      Pthread_mutex_init(&c->lock, NULL);
9  }
10
11 void increment(counter_t *c) {
12     Pthread_mutex_lock(&c->lock);
13     c->value++;
14     Pthread_mutex_unlock(&c->lock);
15 }
16
17 void decrement(counter_t *c) {
18     Pthread_mutex_lock(&c->lock);
19     c->value--;
20     Pthread_mutex_unlock(&c->lock);
21 }
22
23 int get(counter_t *c) {
24     Pthread_mutex_lock(&c->lock);
25     int rc = c->value;
26     Pthread_mutex_unlock(&c->lock);
27     return rc;
28 }
```

Figure 29.2: A Counter With Locks

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

```
1500 // Mutual exclusion lock.
1501 struct spinlock {
1502     uint locked;           // Is the lock held?
1503
1504     // For debugging:
1505     char *name;            // Name of lock.
1506     struct cpu *cpu;       // The cpu holding the lock.
1507     uint pcs[10];          // The call stack (an array of program
1508                             // that locked the lock.
1509 };
```

```
1573 void
1574 acquire(struct spinlock *lk)
1575 {
1576     pushcli(); // disable interrupts to avoid deadlock.
1577     if(holding(lk))
1578         panic("acquire");
1579
1580     // The xchg is atomic.
1581     while(xchg(&lk->locked, 1) != 0)
1582         ;
1583
1584     // Tell the C compiler and the processor to not move loads or stores
1585     // past this point, to ensure that the critical section's memory
1586     // references happen after the lock is acquired.
1587     __sync_synchronize();
1588
1589     // Record info about lock acquisition for debugging.
1590     lk->cpu = mycpu();
1591     getcallerpcs(&lk, lk->pcs);
1592 }
```

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
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

```
1662 // Pushcli/popcli are like cli/sti except that they are matched:
1663 // it takes two popcli to undo two pushcli. Also, if interrupts
1664 // are off, then pushcli, popcli leaves them off.
1665
1666 void
1667 pushcli(void)
1668 {
1669     int eflags;
1670
1671     eflags = readeflags();
1672     cli();
1673     if(mycpu()->ncli == 0)
1674         mycpu()->intena = eflags & FL_IF;
1675     mycpu()->ncli += 1;
1676 }
1677
1678 void
1679 popcli(void)
1680 {
1681     if(readeflags() & FL_IF)
1682         panic("popcli - interruptible");
1683     if(--mycpu()->ncli < 0)
1684         panic("popcli");
1685     if(mycpu()->ncli == 0 && mycpu()->intena)
1686         sti();
1687 }
```

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

```
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 }
```

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

```
2662 // Jump into the scheduler, never to return.
2663 curproc->state = ZOMBIE;
2664 sched();
2665 panic("zombie exit");
2666 }
```

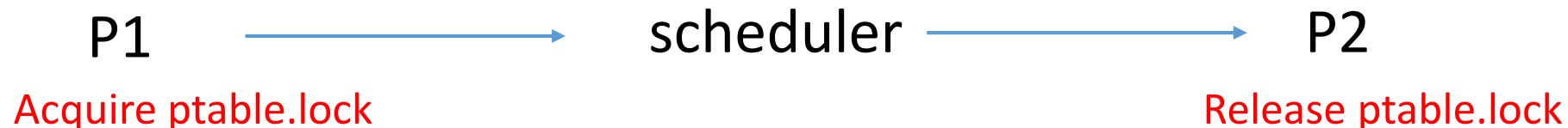
```
2894 // Go to sleep.
2895 p->chan = chan;
2896 p->state = SLEEPING;
2897
2898 sched();
2899
```

```
2826 // Give up the CPU for one scheduling round.
2827 void
2828 yield(void)
2829 {
2830     acquire(&ptable.lock);
2831     myproc()->state = RUNNABLE;
2832     sched();
2833     release(&ptable.lock);
2834 }
```

ptable.lock (1)

```
2409 struct {  
2410     struct spinlock lock;  
2411     struct proc proc[NPROC];  
2412 } 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



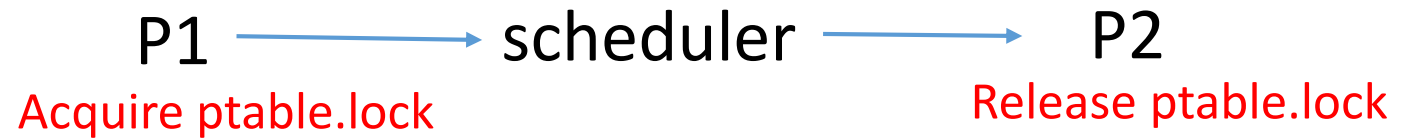
ptable.lock (2)

- 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 switch?
 - 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

```
2826 // Give up the CPU for one scheduling round.
2827 void
2828 yield(void)
2829 {
2830     acquire(&ptable.lock);
2831     myproc()->state = RUNNABLE;
2832     sched();
2833     release(&ptable.lock);
2834 }
```

```
2852 void
2853 forkret(void)
2854 {
2855     static int first = 1;
2856     // Still holding ptable.lock from scheduler.
2857     release(&ptable.lock);
2858
2859     if (first) {
2860         // Some initialization functions must be run i
2861         // of a regular process (e.g., they call sleep
2862         // be run from main()).
2863         first = 0;
2864         iinit(ROOTDEV);
2865         initlog(ROOTDEV);
2866     }
```


ptable.lock (3)



- 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.

```
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 }
```