6.S081: Scheduling Pt. 2

Adam Belay <abelay@mit.edu>

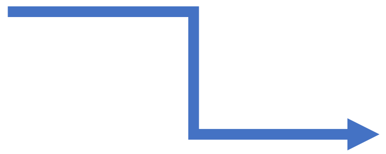
Today's agenda

- Recap xv6 thread scheduling
- Sequence coordination
  - Sleep & wakeup
  - Lost wakeup problem
- Termination

p->lock held across swtch()

On **kthread** stack:

```
acquire(&p->lock);
p->state = RUNNABLE;
swtch();
```



On **scheduler** stack:

```
swtch();
release(&p->lock);
```

p's STATE:
- lock
- kstack
- state
- context

swtch() both saves registers to context and gets off the kernel thread's stack

Why p->lock is held?

- Prevents another core’s scheduler thread from seeing p->state==RUNNABLE
  - Until original core has saved registers into context
  - And until original core is done executing on p’s stack

Q: Why does sched() prohibit other spinlocks from being held?

i.e., why does sched() check that noff==1?

Rescheduling with a lock held could deadlock!

On a single core machine, imagine the following:

```
P1:                     P2:
acquire(&l);
sched();
                        acquire(&l);
                        hangs forever!
```

*Possible on multiple cores too w/ more spinlocks

Solution: Never hold a lock when rescheduling

Next: Sequence coordination

- Threads need a way to wait for specific events or conditions
    - e.g., did a disk finish (complete) a read?
    - e.g., did one process insert data into a pipe that another process was waiting to read?
    - e.g., did a timeout happen?
    - e.g., did a child process finish and exit?

Coordination abstractions

- Allows one thread (or interrupt handler) to wake another thread
- Often: The bottom half wakes the top half
- Fundamental building block for threaded programming
- Many plans: mutexes, condition variables, waitgroups, barriers, semaphores
- xv6 has a simple plan… (shown next)

Strawman example: Pipes

Why not spin until the next event happens?

```
Pipe read:
  while buffer is empty {
  }
Pipe write:
  put data in buffer
```

Better plan: Block

- Enter the scheduler instead of spinning
- Allows other work to be processed while waiting for the event
- More efficient use of CPU resources

Coordination in xv6

- **sleep(chan, lock)**: blocks, waiting for an event; a lock must be held and passed as an argument
- **wakup(chan)**: wakes up a thread in sleep()
- chan is an opaque number or pointer
- lock prevents lost wakeups (next)

UART example

- the UART can only accept one (really a few) bytes of output at a time takes a long time to send each byte, perhaps millisecond.
- processes writing the console must wait until UART sends prev char the UART interrupts after it has sent each character writing thread should give up the CPU until then

Code example: UART

Q: Why the lock arg to sleep?

Q: Why the lock arg to sleep?

- Sleep cannot simply wait for the next event
- Problem: Lost wakeups

Suppose no lock passed to sleep

sleep(chan):

- Sleeps on a “channel”, a number/address that identifies the condition we are waiting for

```
p->state = SLEEPING;
p->chan = chan;
sched()
```

wakeup(chan):

- Wakes up all the threads sleeping on chan; May wake more than one thread

```
for each p:
  if p->state = SLEEPING && p->chan == chan:
    p->state = RUNNABLE
```

How would UART use this?

```
int done;
uartwrite(buf):
  for each char c:
    while not done:
      sleep(&done);
    send c; done = false;
uartintr():
  done = true;
  wakeup(&done);
```

Problem: Race condition

```
int done;
uartwrite(buf):
  for each char c:
    while not done:
      sleep(&done);
    send c; done = false;
uartintr():
  done = true;
  wakeup(&done);
```

Suppose interrupt fires here:
Sleeps waiting for done forever

This is the lost wakeup problem

Lost wakeups

- Need to eliminate window between

1. uartwrite()’s checking of the condition done
2. sleep() marking the thread as asleep

Solution to lost wakeups

- Change interface to sleep() and the way it is used
- A lock must protect the condition
- Callers of both sleep() and wakeup() must ”hold” the condition lock

New API:

sleep(chan, lock)

- Caller must hold the lock
- Sleep releases and reacquires lock internally

wakeup(chan), caller must now hold lock

How does xv6 implement sleep() and wakeup()?

Sleep/wakeup rules are complex

- sleep() doesn’t understand the condition, but it needs a lock that protects the condition
- Flexible but low-level
- Other schemes are cleaner, but less general purpose
    - E.g., the counting semaphore from the reading
- All schemes must cope with lost wakeups

Another challenge: How to terminate threads

- Need to free resources that are still in use
- Problem thread X cannot just destroy thread Y
  - What if Y executing on a different core?
  - What if Y holds a lock?
- Problem hard to free the resources inside a thread
  - Can’t free stack if still using it
  - Has a context that it needs to call swtch()

Stopping processes in xv6

- kill(): allows one process to stop another
    - Hard part: Need to cleanly stop using resources
    - Plan: set flag, let process stop itself at a clean point
- exit(): allows a process to stop itself
    - Set process to ZOMBIE state
    - Don’t free proc until wait() finishes
    - Why? Need to copy out the exit state first

How xv6 implements exit() and kill()

Summary

- sleep()/wakeup() let threads wait for conditions
- Concurrency means hazard of lost wakeups
- Termination is a pain in threading systems