```
6.S081 2020 Lecture 13: Coordination (sleep&wakeup)

# plan
  re-emphasize a few points about xv6 thread switching
  sequence coordination
    sleep & wakeup
    lost wakeup problem
  termination

why hold p->lock across swtch()?
   this is an important point and affects many situations in xv6
   [diagram: P1, STACK1, swtch, STACK_SCHED]
   yield:
     acquire(&p->lock);
     p->state = RUNNABLE;
     swtch();
   scheduler:
     swtch();
     release(&p->lock);

the main point of holding p->lock across swtch():
  prevent another core's scheduler from seeing p->state == RUNNABLE
  until after the original core has stopped executing the thread
  and after the original core has stopped using the thread's stack

why does sched() forbid spinlocks from being held when yielding the CPU?
   (other than p->lock)
   i.e. sched() checks that noff == 1
   on a single-core machine, imagine this:
     P1           P2
     acq(L)
     sched()
                  acq(L)
   this is a deadlock:
     P2 will spin until P1 releases -- and P2 won't yield the CPU
     but P1 won't release until it runs again
   with multiple cores, deadlock can also arise; more spinlocks must be involved
   solution: do not hold spinlocks and yield the CPU!

# topic: sequence coordination
  threads need to wait for specific events or conditions:
    wait for disk read to complete (event is from an interrupt)
    wait for pipe writer to produce data (event is from a thread)
    wait for any child to exit

# coordination is a fundamental building-block for thread programming.
  often straightforward to use.
  but (like locks) subject to rules that sometimes present difficult puzzles.

# why not just have a while-loop that spins until event happens?
  pipe read:
    while buffer is empty {
    }
  pipe write:
    put data in buffer

# better solution: coordination primitives that yield the CPU
  there are a bunch e.g. barriers, semaphores, event queues.
  xv6 uses sleep & wakeup

# example: uartwrite() and uartintr() in uart.c
  I have modified these functions!
    not the same as default xv6
    see "code" link on schedule page
```

```
  the basic idea:
    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
  write() calls uartwrite()
    uartwrite() writes first byte (if it can)
    uartwrite() calls sleep() to wait for the UART's interrupt
    uartintr() calls wakeup()
    the "&tx_chan" argument serves to link the sleep and wakeup
  simple and flexible:
    sleep/wakeup don't need to understand what you're waiting for
    no need to allocate explicit coordination objects

# why the lock argument to sleep()?
  sadly you cannot design sleep() as cleanly as you might hope
  sleep() cannot simply be "wait for this event"
  the problem is called "lost wakeups"
    it lurks behind all sequence coordination schemes, and is a pain
  here's the story

# suppose just sleep(chan); how would we implement?
  here's a BROKEN sleep/wakeup
  broken_sleep(chan)
    sleeps on a "channel", a number/address
      identifies the condition/event we are waiting for
    p->state = SLEEPING;
    p->chan = chan;
    sched();
  wakeup(chan)
    wakeup wakes up all threads sleeping on chan
    may wake up more than one thread
    for each p:
      if p->state == SLEEPING && p->chan == chan:
        p->state = RUNNABLE

# how would uart code use this (broken) sleep/wakeup?
  int done
  uartwrite(buf):
    for each char c:
      while not done:
        sleep(&done)
      send c
      done = false
  uartintr():
    done = true
    wakeup(&done)
  done==true is the condition we're waiting for
  &done is the sleep channel (not really related to the condition)

# but what about locking?
  - driver's data structures e.g. done
  - UART hardware
  both uartwrite()() and uartintr() need to lock
  should uartwrite()() hold a lock for the whole sequence?
    no: then uartintr() can't get lock and set done
  maybe uartwrite() could release the lock before sleep()?
  let's try it -- modify uart.c to call broken_sleep()
    release(&uart_tx_lock);
    broken_sleep(&tx_chan);
    acquire(&uart_tx_lock);

what goes wrong when uartwrite() releases the lock before broken_sleep()?
  uartwrite() saw that the previous character wasn't yet done being sent
```

```
  interrupt occurred after release(), before broken_sleep()
  uartwrite() went to sleep EVEN THOUGH UART TX WAS DONE
  now there is nothing to wake up uartwrite(), it will sleep forever

this is the "lost wakeup" problem.

we need to eliminate the window between uartwrite()'s check of
  the condition, and sleep() marking the process as asleep.
  we'll use locks to prevent wakeup() from running during the entire window.

we'll change the sleep() interface and the way it's used.
  we'll require that there be a lock that protects the
    condition, and that the callers of both sleep() and
    wakeup() hold the "condition lock"
  sleep(chan, lock)
    caller must hold lock
    sleep releases lock, re-acquires before returning
  wakeup(chan)
    caller must hold lock
  (repair uart.c)

let's look at wakeup(chan) in proc.c
  it scans the process table, looking for SLEEPING and chan
  it grabs each p->lock
  remember also that caller acquired condition lock before calling wakeup()
  so wakeup() holds BOTH the condition lock and each p->lock

let's look at sleep() in proc.c
  sleep *must* release the condition lock
    since we can't hold locks when calling swtch(), other than p->lock
  Q: how can sleep() prevent wakeup() from running after it releases the condition lock?
  A: acquire p->lock before releasing condition lock
  since wakeup() holds *both* locks, it's enough for sleep() to hold *either*
    in order to force wakeup() to spin rather than look at this process
  now wakeup() can't proceed until after swtch() completes
    so wakeup() is guaranteed to see p->state==SLEEPING and p->chan==chan
  thus: no lost wakeups!

note that uartwrite() wraps the sleep() in a loop
  i.e. re-checks the condition after sleep() returns, may sleep again
  two reasons:
    maybe multiple waiters, another thread might have consumed the event
    kill() wakes up processes even when condition isn't true
  all uses of sleep are wrapped in a loop, so they re-check

# Another example: piperead()
  the condition is data waiting to be read (nread != nwrite)
  pipewrite() calls wakeup() at the end
  what is the race if piperead() used broken_sleep()?
  note the the loop around sleep()
    multiple processes may be reading the same pipe
  why the wakeup() at the end of piperead()?

# the sleep/wakeup interface/rules are a little complex
  sleep() doesn't need to understand the condition, but it needs the condition lock
  sleep/wakeup is pretty flexible, though low-level
  there are other schemes that are cleaner but perhaps less general-purpose
    e.g. the counting semaphore in today's reading
  all have to cope with lost wakeups, one way or another

*************

# another coordination challenge -- how to terminate a thread?
  a puzzle: we want need to free resources that might still be in use
```

```
# problem: thread X cannot just destroy thread Y
  what if Y is executing on another core?
  what if Y holds locks?
  what if Y is in the middle of a complex update to important data structures?

# problem: a thread cannot free all of its own resources
  e.g. its own stack, which it is still using
  e.g. its struct context, which it may need to call swtch()

# xv6 has two ways to get rid of processes: exit() and kill()

# ordinary case: process voluntarily quits with exit() system call
  some freeing in exit(), some in parent's wait()
  exit() in proc.c:
    close open files
    change parent of children to PID 1 (init)
    wake up wait()ing parent
    p->state = ZOMBIE
      dying but not yet dead
      won't run again
      won't (yet) be re-allocated by fork(), either
    (note stack and proc[] entry are still allocated...)
    swtch() to scheduler
  wait() in proc.c (parent, or init, will eventually call):
    sleep()s waiting for any child exit()
    scans proc[] table for children with p->state==ZOMBIE
    calls freeproc()
      (p->lock held...)
      trapframe, pagetable, ..., p->state=UNUSED
  thus: wait() is not just for app convenience, but for O/S as well
    *every* process must be wait()ed for
    thus the re-parenting of children of an exiting process
  some complexity due to
     child exits concurrently with its own parent
     parent-then-child locking order to avoid deadlock

# what about kill(pid)?
  problem: may not be safe to forcibly terminate a process
    it might be executing in the kernel
      using its kernel stack, page table, proc[] entry, trapframe
    it might hold locks
      e.g. in the middle of fork()ing a new process
      and must finish to restore invariants
    so: kill() can't directly destroy the target!
  solution:
    kill() sets p->killed flag, nothing else
    the target process itself checks for p->killed
      and calls exit() itself
    look for "if(p->killed) exit(-1);" in usertrap()
      no locks are held at that point
      so it's safe to exit()

# what if kill() target is sleep()ing?
  in that case it doesn't hold locks, and isn't executing!
  is it OK for kill() destroy the target right away?
  might be OK:
    waiting for console input
  might not be OK:
    waiting for disk midway through file creation

# xv6 solution to kill() of sleep()ing process
  see kill() in proc.c
    changes SLEEPING to RUNNABLE -- like wakeup()
    so sleep() will return, probably before condition is true
  some sleep loops check for p->killed
```

```
    e.g. piperead(), consoleread()
    otherwise read could hang indefinitely for a killed process
  some sleep loops don't check p->killed
    e.g. virtio_disk.c
    OK not to check p->killed since disk reads are pretty quick
  so a kill()ed process may continue for a while
    but usertrap() will exit() after the system call finishes

# xv6 spec for kill
  if target is in user space
    will die next time it makes a system call or takes a timer interrupt
  if target is in the kernel
    target will never execute another user instruction
    but may spend quite a while yet in the kernel

# Summary
  sleep/wakeup let threads wait for specific events
  concurrency means we have to worry about lost wakeups
  termination is a pain in threading systems
```