```
6.828 2014 Lecture 11: Coordination (sleep&wakeup)
=

# plan
  homework
  sequence coordination
    xv6: sleep & wakeup
  challenges
    lost wakeup problem
        signals

# homework: context switching
  user-level context switching
    same idea as kernel-level
  show solution
  Q: what address is 161
  Q: why is it on the stack?
  Q: what happens when uthread blocks in kernel?
  Q: do user-level uthreads run concurrently?

# big picture:
  Multiple threads executing in the kernel
  Sharing memory, devices, and various data structures.
  Locks to protect invariants
  One outstanding disk request
  One scheduler selecting a thread to run
  # Show context switching pattern in kernel

# sequence coordination:
  how to arrange for threads to wait for each other to do
    e.g., wait for disk interrupt to complete
    e.g., wait for pipe readers to make space in pipe
    e.g., wait for child to exit
    e.g., wait for block to use

# straw man solution: spin
  waste CPU cycles if need to spin for long time

# better solution: primitives for coordination
  sleep & wakeup (xv6)
  condition variables (homework)
  barriers (homework)
  etc.

# sleep&wakeup:

  sleep(chan, lock)
    sleeps on a "channel", an address to name the condition we are sleeping on

  wakeup(chan)
     wakeup wakes up all threads sleeping on chan
     this may wake up more than one thread

  threads may need to retest the condition they are waiting for
  to make sure that they don't proceed, if only one thread can proceed
  therefore sleep is typically called inside a loop

  case study: iderw()

# designing and implementing these primitives is difficult
  - why does sleep take the ide lock as argument?
    demo:
    switch acquire(ptable.lock) and release(lk)
      ie. sleep releases thread's lock before acquiring ptable.lock
    ideintr() runs before sleeper sets its thread state to SLEEPING
      it scans proctable but no thread is SLEEPING
    now sleep acquires ptable.lock and sets current thread to SLEEPING
    -> sleep misses wakeup; deadlock
   - signals

# problem: lost wakeup
  wakeup happens before sleeper goes to sleep
  wakeup is lost
  many solutions to this problem
   e.g., count wakeups (as semaphores do)
  all require some new semantics

# xv6 solution: sleep takes a lock as argument
  sleeper and wakeup acquires locks for shared data structure
  sleep holds the lock until after it has ptable.lock
  once it has ptable.lock, no wakeup can come in before it sets state to sleeping
    -> no lost wakeup problem
  requires that sleep takes a lock argument!

# iderw example
  first hold ptable lock
  set SLEEP
  then release the lock argument
  wakeup cannot get lock until sleeper is already to at sleep
  why a loop around sleep?

# Many primitives in literature to solve lost-wakeup problem
  counting wakeup&sleep calls in semaphores
  pass locks as an extra argument in condition variables (as in sleep)
  etc.

# Another example: pipe
  what is the race if sleep didn't take p->lock as argument?

# kill: how to kill a process?
  problem: target process may be running show you cannot clean it up yet
  solution: target commits suicide
    source sets flag
        target thread checks flag in trap and exits
  downside: it may take a while until process is really killed
   e.g., until timer interrupt goes off
  another complication: what if target thread is sleeping in the kernel

# signals and sleep
  goal: user wants to kill a process (ctrl-C), but process is at sleep in the kernel
  hard to get right:
    a process could sleep somewhere deep in the kernel
    a signal forces it out of sleep
    but when out of sleep,it is *not* because the condition it is waiting on is true

# xv6 solution
  some sleep loops check for p->killed (e.g., see pipe.c)
  BUT not always (see ide.c)
    Q: does iderw check for p->killed in sleep loop?
    Q: what goes wrong if we would modify xv6 to check for p->killed and return
       from iderw()?
    A: we might be in the create() system call, doing several disk writes.
           if iderw() returns, then may leave the on-disk fs in an inconsistent state
           e.g., block allocated but it doesn't show up in any inode
  larger problem:
    a thread may be sleeping deep in the kernel
        may break invariants if we just bail out
  xv6 doesn't allow this kill in iderw():
    target thread may not be killed until much later
```

```
# xv6 spec for kill
  target will never execute a user instruction
  target may be killed but with (long) delay

# Potential race between sleep & setting p->killed
  race:
    1. target may check p->killed
        2. source sets p->killed
        3. target calls sleep
  effect:
    target will be killed after reading from disk
      may take a long time before that happens
    or never
      see consoleread() in console.c
      target is waiting for input from console, but user never types again
  BUT race is *not* a violation of xv6 spec for kill

# Other "solutions" to dealing with sleep and signals
  A common approach is use longjmp (unwind the stack), and retry the system call.
  but also has corner cases that are difficult to get right
```