6.033 Computer System Engineering
Spring 2009

```
6.003 Lecture 7: Threads and Condition Variables

topic: virtual processors / threads
  monday: client/server / bounded buffer w/ one CPU per program
  today: more programs than CPUs
    only one CPU (no busy looping!)
    or a few CPUs, many more programs
  also fewer programs than CPUs (CPUs may need to be idle!)
  goal: virtualize the processor
    multiplex CPU among many "threads"

thread abstraction
  state of a runnable program
  so CPU multiplexing == suspend X, resume Y, suspend Y, resume X
  other abstractions for multiplexing CPU are possible
    this is a useful and traditional one
  controlled by "thread manager" or "scheduler"
  what is the required state?
  how to save state for suspend?
  how to resume from saved state?

send() from previous lecture
  illustrates why we want threads and multiplexing
  [send slide]
  loops waiting for BB to be non-full
  burns up a lot of CPU time
  if one CPU, maybe receive will never run!
  we'd like to let receive() run...

send w/ yield
  [send/receive slide]
  yield() gives up the CPU
  lets other threads run
    e.g. a receive() may have been waiting and called yield()
  someday yield() will return
    after other thread yield()s
    e.g. it tries to receive() but BB is empty

how to implement yield()?
  yield() is the guts of the thread implementation
  suspend one, resume another

data:
  threads[] table: state, sp
  thread stack 0, stack 1, ...
  cpus[] table: thread
  t_lock (coarse granularity...)

[yield version 1 slide]

what happens in yield?
  send calls yield
  how does it know what thread is running?
    per-CPU register CPU() contains cpu #
    cpus[] says what's happening on each CPU
  RUNNING -> RUNNABLE
```

```
    RUNNING means some CPU executing it now
    RUNNABLE means not executing, but could
  save SP (the CPU register)
  look for a different thread to run
    ignore the RUNNING ones
  mark "new" thread as RUNNING, so no other CPU runs it
  restore saved SP of new thread
    that is, load it into CPU's SP register
  return

questions:
  what state does yield save?
    just SP
    what is on the stack? local vars, RA, send()'s saved registers &c
    we might need to save/restore callee-saved registers too
  what happens in return after restoring?
  this use of SP might not work, depends on compiler
    I'm assuming compiled code does not change SP in body of yield()
    and that return basically just pops RA off stack
    more complex in real life
  what does t_lock protect?
    indivisible set of .state and .sp
    indivisible find RUNNABLE and mark it RUNNING
    don't let another CPU grab current stack until we've switched

Questions?

motivate notify / wait
  [send with yield slide]
  send() and receive() still chew up CPU time
    e.g. send() waits for receive to free up a slot in BB
    e.g. receive() waits for BB to be non-empty
    repeated yield() expensive if many threads waiting
  want send to suspend itself
    have receive wake it up when there is space
    do it in a general way
    don't want receive to have to know abt all threads waiting in send

"condition variable"
  object that acts as a rendezvous
  two methods:
    wait(cvar, lock) -- release(lock), yield, return after notify(cvar)
    notify(cvar) -- wake up all threads currently in wait(cvar)
  notify has no memory: if no threads wait()ing, no effect at all
    wait() and then notify(): wait returns
    notify() and then wait(): wait does not return

each BB has two condition variables:
  notfull (send waits on this if full)
  notempty (recv waits on this if empty)

[send with wait/notify]
  if full, waits, receive will someday free up a slot and notify(p.notfull)
  waits in a loop, re-checks after wait returns
    maybe multiple senders waiting, but only one slot freed up
    that is, wait() returning is only a hint
```

```
    you always ought to explicitly check the condition
  notifies notempty in case one or more receives are waiting
    no harm if no-one is waiting
  holds lock across while test and use of buffer
    so no other send() can sneak in and steal buffer[] slot

why does wait() release p.lock? why not have send() release it?
  i.e. why not
    while p.in - p.out == N:
      release
      wait(p.notfull)
      acquire
  notify might occur between release and wait
    no effect, since no threads waiting at that point
    then send()'s wait() won't return, even though there's a msg!
  this is the "lost notify" problem

avoiding lost notifies
  wait(cvar, lock)
    caller must hold the lock
    wait() atomically releases lock and marks thread as waiting
      so no notify can intervene
    re-acquires lock before returning
  notify(cvar)
    caller must hold the lock
  so, implicitly, condition variable always associated with a particular lock

implementing wait
  thread table additions:
    new state: WAITING
    threads[].cvar (so notify can find us)
  big Q: where to release the lock?
  [wait() slide]
  acquire t_lock first, then release the lock, then WAITING
    ensure that notify() holds both!
  b.t.w. need to modify yield()
  [wait+notify() slide]
  notify() caller holds lock, notify() acquires t_lock
    so receive's notify() holds both locks
    either executes before send acquires lock
      or after sending thread suspends
      (but NOT between send's check and suspension)
    if before:
      send() acquire waits until receive is done
      send() will see empty slot and not wait
    if after:
      notify() will see WAITING send thread, and mark it RUNNABLE

but now we must revisit yield()
  [yield v1 again]
  t_lock already held, not need to set state (easy)
  yield might find there is nothing RUNNABLE!!! (harder)
    this thread WAITING, but receive() running on another CPU
  loops forever while holding t_lock
    so no other CPU can execute notify()
    so no thread will ever be RUNNABLE
```

```
    system will hang

how to fix yield()?
  [yield version 2 slide]
  don't acquire t_lock, don't set to RUNNABLE
  release+acquire in "idle loop"
    still spins indefinitely while no runnable threads
    BUT lets other CPUs execute notify()
  t_lock held on return, but wait() releases it
  note I've also set the SP to a per-CPU stack, before idle loop
    why?
    yield() v1 runs idle loop on calling thread's stack
    someone might notify() it
    some other CPU in idle loop might run the thread
    now two CPUs are executing on the same stack
      e.g. calling functions like acquire, which modify the stack
    thus per-CPU stack for yield() to use when not in any thread

pre-emptive scheduling
  what if a thread never calls yield()?
    we are in trouble, no way to multiplex that CPU
    compute-bound, or long code paths, or broken user programs
    too annoying to require programmer to insert yield()s
  we want forcible periodic yield

how to pre-empt?
  timer h/w, generates an interrupt 10 times per second
  interrupt saves state, jumps to handler in kernel
  timer():
    yield()
    return

will the resulting stack resume correctly?
  interrupt pushes PC + regs on current thread's stack
  when not running, stack looks like:
    ...
    RA to thread at time of interrupt
    registers
    RA to timer()
  so yield() returns to interrupt handler, which returns to interrupted code

what if timer interrupt while you are in yield already?
  would call yield recursively
  deadlock: already holding t_lock
  acquire should disable interrupts
    release should re-enable
  not just for here, but all uses of locks

what if timer interrupt after idle loop releases t_lock?
  again, recursive yield()
  but invalid cpus[][CPU()].thread
  so fix yield() to null out .thread
  and fix timer interrupt to yield only if valid .thread

Summary
  closing thought: how to kill a thread? might be running...
```

```
threads are virtual processors
  allow many threads, few CPUs
  the foundation of time-sharing
we had to integrate:
  yield()
  condition variables
  interrupts for pre-emption
missing: creation (easy), exit (harder)
```