L5: Threads

<u>Topics</u>
Multi-threading
Condition Variables
Pre-emption

Sam Madden
6.033 Spring 2014

Strawman Acquire/Release

```
acquire(L):     ## strawman design
  while L != 0:
    do nothing
  L ← 1

release(L):
  L ← 0
```

Race condition!

CPU1	CPU2
Acquire(L)	Acquire(L)

Both see L == 0, both think they have acquired

Atomic Instructions to The Rescue

XCHG reg,addr:

tmp ← Mem[addr]
Mem[addr] ← reg
reg ← temp

Processor does this atomically

```
acquire(L):  ## correct design
  do:
    r ← 1
    XCHG r, L
  while r==1
```

If L is 0, after XCHG, r will be 0, and loop will terminate

Otherwise, someone else is holding L, and need to keep trying

Recall: send with locking

```
send(bb, m):
    acquire(bb.send_lock)
    while True:
        if bb.in – bb.out < N:
            bb.buf[bb.in mod N] ← m
            bb.in ← bb.in + 1
            release(bb.send_lock)
            return
```

Send and receive with yield

```
send(bb, m):
    acquire(bb.lock)
    while True:
        if bb.in – bb.out < N:
            … // enqueue message & return
        release(bb.lock)
        yield()
        acquire(bb.lock)

receive(bb):
    acquire(bb.lock)
    while True:
        if bb.in > bb.out:
            … // dequeue message & return
        release(bb.lock)
        yield()
        acquire(bb.lock)
```

```
yield():
    acquire(t_lock)
    id = cpus[CPU].thread
    threads[id].state = RUNNABLE
    threads[id].sp = SP
    threads[id].ptr = PTR

    do:
        id = (id + 1) mod N
    while threads[id].state ≠ RUNNABLE

    threads[id].state = RUNNING
    PTR = threads[id].ptr
    SP = threads[id].sp
    cpus[CPU].thread = id
    release(t_lock)
```

```
yield():
    acquire(t_lock)
    id = cpus[CPU].thread
    threads[id].state = RUNNABLE
    threads[id].sp = SP
    threads[id].ptr = PTR

    do:
        id = (id + 1) mod N
    while threads[id].state ≠ RUNNABLE

    threads[id].state = RUNNING
    PTR = threads[id].ptr
    SP = threads[id].sp
    cpus[CPU].thread = id
    release(t_lock)
```

} suspend current thread

```
yield():
    acquire(t_lock)
    id = cpus[CPU].thread                    } suspend
    threads[id].state = RUNNABLE             } current
    threads[id].sp = SP                      } thread
    threads[id].ptr = PTR

    do:                                      } choose
        id = (id + 1) mod N                  } new
    while threads[id].state ≠ RUNNABLE       } thread

    threads[id].state = RUNNING
    PTR = threads[id].ptr
    SP = threads[id].sp
    cpus[CPU].thread = id
    release(t_lock)
```

```
yield():
    acquire(t_lock)
    id = cpus[CPU].thread                      } suspend
    threads[id].state = RUNNABLE               } current
    threads[id].sp = SP                        } thread
    threads[id].ptr = PTR

    do:                                        } choose
        id = (id + 1) mod N                    } new
    while threads[id].state ≠ RUNNABLE         } thread

    threads[id].state = RUNNING                } resume
    PTR = threads[id].ptr                      } new
    SP = threads[id].sp                        } thread
    cpus[CPU].thread = id
    release(t_lock)
```

Send with yield, again

```
send(bb, m):
    acquire(bb.lock)
    while True:
        if bb.in – bb.out < N:
            bb.buf[bb.in mod N] ← m
            bb.in ← bb.in + 1
            release(bb.lock)
            return
        release(bb.lock)
        yield()
        acquire(bb.lock)
```

Send with wait / notify

```
send(bb, m):
    acquire(bb.lock)
    while True:
        if bb.in – bb.out < N:
            bb.buf[bb.in mod N] ← m
            bb.in ← bb.in + 1
            release(bb.lock)
            notify(bb.empty)
            return
        release(bb.lock)
        yield()
        acquire(bb.lock)
        wait(bb.full, bb.lock)
```

Wait and notify

```
wait(cvar, lock):
    acquire(t_lock)
    release(lock)
    threads[id].cvar = cvar
    threads[id].state = WAITING
    yield_wait()            # will be a little different than yield
    release(t_lock)
    acquire(lock)
```

Wait and notify

```
wait(cvar, lock):
        acquire(t_lock)
        release(lock)
        threads[id].cvar = cvar
        threads[id].state = WAITING
        yield_wait()            # will be a little different than yield
        release(t_lock)
        acquire(lock)

notify(cvar):
        acquire(t_lock)
        for i = 0 to N-1:
                if threads[i].cvar == cvar && threads[i].state == WAITING:
                        threads[i].state = RUNNABLE
        release(t_lock)
```

Recall: original yield

```
yield():
    acquire(t_lock)
    id = cpus[CPU].thread
    threads[id].state = RUNNABLE
    threads[id].sp = SP
    threads[id].ptr = PTR

    do:
        id = (id + 1) mod N
    while threads[id].state ≠ RUNNABLE

    threads[id].state = RUNNING
    PTR = threads[id].ptr
    SP = threads[id].sp
    cpus[CPU].thread = id
    release(t_lock)
```

} suspend current thread

} choose new thread

} resume new thread

Yield for wait, first attempt

```
yield_wait():
    ~~acquire(t_lock)~~
    id = cpus[CPU].thread
    ~~threads[id].state = RUNNABLE~~
    threads[id].sp = SP
    threads[id].ptr = PTR

    do:
        id = (id + 1) mod N
    while threads[id].state ≠ RUNNABLE

    threads[id].state = RUNNING
    PTR = threads[id].ptr
    SP = threads[id].sp
    cpus[CPU].thread = id
    ~~release(t_lock)~~
```

Yield for wait

```
yield_wait():
    id = cpus[CPU].thread
    threads[id].sp = SP
    threads[id].ptr = PTR
    SP = cpus[CPU].stack

    do:
        id = (id + 1) mod N
        release(t_lock)
        acquire(t_lock)
    while threads[id].state ≠ RUNNABLE

    threads[id].state = RUNNING
    PTR = threads[id].ptr
    SP = threads[id].sp
    cpus[CPU].thread = id
```

} Switch to this CPUs kernel stack

} choose new thread, but allow other CPUs to notify()

} resume new thread

timer interrupt

Timer interrupt:

```
  push PC             ## done by CPU
  push registers  ## not a function call, so can't assume
                         ##compiler saves them
  yield()
  pop registers
  pop PC
```

What happens if timer interrupt occurs when CPU is running yield / yield_wait?

t_lock held ➔ thread blocks forever

Solution: Disable timer interrupts before entering/exiting yield

Summary

- Threads allow running many concurrent activities on few CPUs
- Threads are at the core of most OS designs
- Explored some of the subtle issues with threads
  - yield, condition variables, preemption, …