L5: Threads

Nickolai Zeldovich
6.033 Spring 2013

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: …
        release(bb.lock)
        yield()
        acquire(bb.lock)

receive(bb):
    acquire(bb.lock)
    while True:
        if bb.in > bb.out: …
        release(bb.lock)
        yield()
        acquire(bb.lock)
```

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

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

    threads[id].state = RUNNING
    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

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

    threads[id].state = RUNNING
    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

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

    threads[id].state = RUNNING
    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
```

suspend current thread

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

choose new thread

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

resume new thread

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

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

    threads[id].state = RUNNING
    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

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

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

Yield for wait

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

switch to this CPU's kernel stack

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

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

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

resume new thread

Yield for preemption

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

switch to this CPU's kernel stack

```
    cpus[CPU].thread = None
    do:
        id = (id + 1) mod N
        release(t_lock)
        acquire(t_lock)
    while threads[id].state ≠ RUNNABLE
```

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

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

resume new thread

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, …