L5: Threads

Topics
Multi-threading
Condition Variables
Pre-emption

Sam Madden
6.033 Spring 2014
**Strawman Acquire/Release**

\[
\text{acquire}(L): \quad \#\# \text{strawman design} \\
\text{while } L \neq 0: \\
\quad \text{do nothing} \\
L \leftarrow 1
\]

\[
\text{release}(L): \quad \text{ } \\
L \leftarrow 0
\]

*Race condition!*  

<table>
<thead>
<tr>
<th>CPU1</th>
<th>CPU2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acquire(L)</td>
<td>Acquire(L)</td>
</tr>
</tbody>
</table>

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

```python
def 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
    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
  } choose new thread
  } resume new thread
  } 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

```python
def 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)
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_wait():

\[
\begin{align*}
    \text{id} &= \text{cpus[CPU].thread} \\
    \text{threads[id].sp} &= \text{SP} \\
    \text{threads[id].ptr} &= \text{PTR} \\
    \text{SP} &= \text{cpus[CPU].stack}
\end{align*}
\]

do:

\[
\begin{align*}
    \text{id} &= (\text{id} + 1) \mod N \\
    \text{release(t_lock)} \\
    \text{acquire(t_lock)}
\end{align*}
\]

while \( \text{threads[id].state} \neq \text{RUNNABLE} \)

\[
\begin{align*}
    \text{threads[id].state} &= \text{RUNNING} \\
    \text{PTR} &= \text{threads[id].ptr} \\
    \text{SP} &= \text{threads[id].sp} \\
    \text{cpus[CPU].thread} &= \text{id}
\end{align*}
\]
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, …