

6.033 Spring 2018

Lecture #5

- **Threads**
- **Condition Variables**
- **Preemption**

operating systems enforce modularity on a single machine using **virtualization**

in order to enforce modularity + build an effective operating system

1. programs shouldn't be able to refer to (and corrupt) each others' **memory** → **virtual memory**
2. programs should be able to **communicate** → **bounded buffers**
(virtualize communication links)
3. programs should be able to **share a CPU** without one program halting the progress of the others → assume one program per CPU

operating systems enforce modularity on a single machine using **virtualization**

in order to enforce modularity + build an effective operating system

1. programs shouldn't be able to refer to (and corrupt) each others' **memory** → **virtual memory**
2. programs should be able to **communicate** → **bounded buffers**
(virtualize communication links)
3. programs should be able to **share a CPU** without one program halting the progress of the others → **threads**
(virtualize processors)

today's goal: use **threads** to allow multiple programs to share a CPU

thread: a virtual processor

thread API:

`suspend()` : save state of current thread
to memory

`resume()` : restore state from memory

```
send(bb, message):  
  acquire(bb.lock)  
  while bb.in - bb.out == N:  
    release(bb.lock)  
    acquire(bb.lock)  
  bb.buf[bb.in mod N] <- message  
  bb.in <- bb.in + 1  
  release(bb.lock)  
  return
```

```
send(bb, message):  
  acquire(bb.lock)  
  while bb.in - bb.out == N:  
    release(bb.lock)  
    yield()  
    acquire(bb.lock)  
  bb.buf[bb.in mod N] <- message  
  bb.in <- bb.in + 1  
  release(bb.lock)  
  return
```

```
yield():
```

```
    acquire(t_lock)
```

```
    id = cpus[CPU].thread
```

```
    threads[id].state = RUNNABLE
```

```
    threads[id].sp = SP
```

```
    threads[id].ptr = PTR
```

Suspend
current thread

```
do:
```

```
    id = (id + 1) mod N
```

```
while threads[id].state != RUNNABLE
```

Choose new
thread

```
SP = threads[id].sp
```

```
PTR = threads[id].ptr
```

```
threads[id].state = RUNNING
```

```
cpus[CPU].thread = id
```

Resume new
thread

```
release(t_lock)
```

condition variables: let threads wait for events, and get notified when they occur

condition variable API:

`wait(cv)`: yield processor and wait to be notified of `cv`

`notify(cv)`: notify waiting threads of `cv`


```
send(bb, message):  
    acquire(bb.lock)  
    while bb.in - bb.out == N:  
        release(bb.lock)  
        wait(bb.not_full)  
        acquire(bb.lock)  
    bb.buf[bb.in mod N] <- message  
    bb.in <- bb.in + 1  
    release(bb.lock)  
    notify(bb.not_empty)  
    return
```

(threads in receive() will wait on bb.not_empty and notify of bb.not_full)

problem: lost notify

condition variable API:

`wait(cv, lock)`: yield processor, release lock, wait to be notified of cv

`notify(cv)`: notify waiting threads of cv

```
send(bb, message):  
  acquire(bb.lock)  
  while bb.in - bb.out == N:  
    wait(bb.not_full, bb.lock)  
  bb.buf[bb.in mod N] <- message  
  bb.in <- bb.in + 1  
  release(bb.lock)  
  notify(bb.not_empty)  
  return
```

```
wait(cv, lock):
    acquire(t_lock)
    release(lock)
    id = cpus[CPU].thread
    threads[id].cv = cv
    threads[id].state = WAITING
    yield_wait()
    release(t_lock)
    acquire(lock)
```

will be different
than `yield()`

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

```
yield_wait(): // called by 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
```

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

```
release(t_lock)
```

problem: wait() holds **t_lock**

```
yield_wait(): // called by wait()
```

```
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
```

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

problem: current thread's state shouldn't be RUNNABLE

```
yield_wait(): // called by wait()
```

```
id = cpus[CPU].thread
```

```
threads[id].sp = SP
```

```
threads[id].ptr = PTR
```

```
do:
```

```
id = (id + 1) mod N
```

```
while threads[id].state != RUNNABLE
```

```
SP = threads[id].sp
```

```
PTR = threads[id].ptr
```

```
threads[id].state = RUNNING
```

```
cpus[CPU].thread = id
```

problem: deadlock (`wait()` holds `t_lock`)

```
yield_wait(): // called by wait()
```

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

```
do:
```

```
    id = (id + 1) mod N
```

```
    release(t_lock)
```

```
    acquire(t_lock)
```

```
while threads[id].state != RUNNABLE
```

```
SP = threads[id].sp
```

```
PTR = threads[id].ptr
```

```
threads[id].state = RUNNING
```

```
cpus[CPU].thread = id
```

problem: stack corruption


```
yield_wait(): // called by 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
```

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

preemption: forcibly interrupt threads

```
timer_interrupt():  
  push PC  
  push registers  
  yield()  
  pop registers  
  pop PC
```

problem: what if timer interrupt occurs while running
`yield()` or `yield_wait()`?

preemption: forcibly interrupt threads

```
timer_interrupt():  
  push PC  
  push registers  
  yield()  
  pop registers  
  pop PC
```

solution: hardware mechanism to disable interrupts

- **Threads** virtualize a processor so that we can share it among programs. **yield()** allows the kernel to suspend the current thread and resume another.
- **Condition Variables** provide a more efficient API for threads, where they **wait** for an event and are **notified** when it occurs. `wait()` requires a new version of `yield()`, **`yield_wait()`**.
- **Preemption** forces a thread to be interrupted so that we don't have to rely on programmers correctly using `yield()`. Requires a special **interrupt** and hardware support to disable other interrupts.