6.033 Lecture 5 2/19/2014

Threads

<u>Topics:</u>
Threads
Condition Variables
Pre-emption

Last two lectures: multiple programs on a computer, each with it's own CPU.
Last time: bounded buffer, and locking to preserve atomcity and protect data structures from concurrent modifications

First lets look at how to implement an atomic lock (slides)

```
 acquire(l):    ## strawman design
   while l != 0:
     do nothing
   l <- 1

 release(l):
   l <- 0
```

 Race condition similar to send(), with two CPUs:
   Both CPUs run acquire(l).
   Both see l==0.
   Both set l<-1.
   Both start executing code after the acquire().
 Checking l!=0 and setting l=1 should be atomic!
   Chicken-and-egg problem?

Hardware support: atomic instruction that combines both operations!
 Luckily, most processors provide a solution (often many possible solutions).
 One such atomic instruction on x86:

```
   XCHG reg, addr   ## atomic exchange
     temp <- Mem[addr]
     Mem[addr] <- reg
     reg <- temp
```

 If XCHG is atomic, we can implement acquire.

```
 acquire(l):
   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
when we succeed we will have atomically installed 1 into l, so others
will block)

Ok, now to the main topic: how to handle more than one program per CPU.

Basic idea: create a virtual processor called a "*thread*"
A key OS service, like virtual memory

Basic plan for for creating a thread is encapsulate state of running program on a CPU, and swap state when need to switch threads

<u>What is the state of a running program?</u>

- value of all of the registers, including the stack and program counter
- all of the program's memory (heap, stack, etc.)
    - captured by the page table (register)

Need an thread API:
    suspend() : save the state of a running program to memory (on a given CPU)
    resume() : restore the state from memory (on a given CPU)

Can now multiple a single CPU between two programs:
    suspend A
    resume B
    suspend B
    resume A

<u>Ok, so when do we suspend/resume a running program?</u>
    periodically (let's come back to that idea)
    when program is busy doing something else

    (when might that be?)
Saw an example yesterday

(Show send() code)

Spin-loop uses lots of CPU cycles
If we only have one CPU, the receive() function might never run.

Idea: let's switch to a different thread (virtual CPU) if send is spinning
(show send() w/ yield code)

Here yield() releases the physical CPU from the current thread, and gives to another.

The call to yield() will return some time later (after one or more other threads have run)

<u>So what happens inside yield?</u>

Suspend running thread
Choose new thread to run
Resume new thread

This is how threading is implemented —so let's dive in!

data structures
threads table:
——————

thread id	stack pointer	page table register	state
			RUNNABLE/RUNNING

cpus table
———————

cpu id	thread id

t_lock: to make concurrent suspend/resume operations atomic

All we need to do is save stack pointer and page table register, since all registers send() is using will be saved to stack! (PC is saved to stack by call to yield — more on that in a minute)

Need to swap page table register, since stack pointer is a virtual address in the address space of the currently running program! Swapping page table register also effectively "swaps" from one memory to another.

Ok, so lets look at some code:

[slide: step 1, suspend current thread]
How does yield() know which thread is currently executing?
  Some per-CPU register (call it CPU) contains the current CPU number.
  cpus[] array contains the thread running on each CPU.

Change state from RUNNING (currently executing) to RUNNABLE (can execute).
  Allows some CPU (either this one or another one) to run this thread later.
Save the current value of the SP register to the current thread.

[slide: step 2, find another thread]
Look through the threads array for a RUNNABLE thread.
Skip ones that are RUNNING on other CPUs.

[slide: step 3, resume the chosen thread]
First, mark the chosen thread as RUNNING, so no other CPU runs it.
Restore the CPU's SP register from the thread's saved value.
Return, which starts running the chosen thread.

<u>Why does the chosen thread start running when yield returns?</u>

When we change the value of the stack pointer, we are causing yield() to return to a different thread!!!

How? Think about what stack pointer has on it after calling yield:

Suppose P1 call's send, which yields to P2

```
P1 stack                                P2 stack
bb                                      # saved vars
m #sends args                           return addr into somewhere  <—
… # saved regs, etc
return addr into send  <— top of stack
```

When we set SP to P2, the return address that will be popped off is for where-ever P2 last yielded from

Similarly, if P2 yields back to P1, we will set SP to P1's stack, and return will go back to send()

Setting the SP is effectively the point at which we have switched threads

yield() is called by one thread, but returns into a different thread.

Second half of yield (after SP switch) actually finishes a previous yield.

What does t_lock protect?
Atomically set thread[].state and thread[].sp.
Atomically find a RUNNABLE thread and mark it RUNNING.

[show yield code again]

yield lets us switch to another thread
but send() and receive() are still inelegant — lots of checking and yielding, may be inefficient.

<u>Wouldn't it be better if send/receiver just got woken up when buffer had space / message was ready?</u>

Condition Variables

Turns out most OSes provide an abstraction for this: *condition variables*

A CV names a condition a thread can wait for

wait(cv,lock); go to sleep (releasing lock, re-acquiring after wakeup)
notify(cv): wake up any threads waiting on the cv

note that sequencing of wait/notify is important
wait() and then notify(): wait returns
notify() and then wait(): wait does not return

(cv does not keep any state about how many notifies have been done to it)

Lock argument will help — will come back later

<u>So how do we use this in our send/receive?</u>

two condition variables:
bb.full: send waits if full, receive notifies
bb.empty: receive waits if empty, send notifies

(show send code)

<u>why do we still need while loop?</u>

because many senders may be waiting, and notify wakes up all of them, can't assume there is space after a notify

(other possible implementations of wait and notify are possible, e.g., where notify just notifies one waiter. but problem is then what if that waiter doesn't want to "use" the notify, e.g., doesn't want to write to slot.)

<u>What happens if send notifies but no one is waiting?</u>

(that's ok, a later receiver will not wait if there is space, i.e., in > out)

Ok, so now lets try to understand why we need this lock argument to wait()

Suppose we just release the lock before calling wait and acquire it afterwards

(show demo code)

<u>Program freezes? why?</u>

Thread T0 calls release(&lck), but before it can call wait_unlocked(), another thread T1 runs, calls notify, so the notify is lost (this is bad)

Need to mark thread as waiting before a concurrent notify can run, so that we don't miss notify!

So we pass lock into wait so that it can release the lock at the appropriate time.

This is called the "lost notify" problem

<u>What if we didn't use a lock at all?</u> (same problem — might miss notify)
<u>What if we held lock through wait?</u> (bad idea — notifier needs to acquire lock to update
data structure!)

Show fixed version

(Note that wait re-acquires the lock before it returns, but this isn't necessary for correctness, just for symmetry.)

Here wait atomically releases lock and sets itself as ready to receive a notify

<u>How does this avoid lost notify?</u>

Either notify completes before sender checks (in which case send will find space)

or

Sender waits before notifier runs (in which case sender will receive notify)

"Before or after atomicity" A before B or B before A but not interleaved!

In practice, we associate a condition variable with a lock that protects concurrent data structure

To really understand how this works, lets look at code for wait / notify:
[slide: wait]

Changes to the thread table:
- new state called WAITING.
- new variable threads[].cvar, to help notify() find threads.

Once we've acquired t_lock, we can be sure that concurrent notifies won't run and see our thread until we've had a chance to both update thread state and release the lock

Need to modify yield() a bit — come back to that

  [slide: wait + notify]
  notify() iterates over threads, changes WAITING to RUNNABLE if matching cvar.
  CPUs that subsequently call yield() will find these RUNNABLE threads, run them.

<u>Ok, so what is this "yield_wait" thing?</u>

[slide: revisiting yield -- recall original yield]
  Easy problem: wait() already acquired t_lock and changed state.
    Easy fix: just delete that code.
    [slide: yield_wait(), first attempt]

  Hard problem: what if nothing is RUNNABLE?
    yield_wait() will keep looping forever, while holding t_lock.
    Other CPUs cannot acquire t_lock, even if they want to do a notify!
      ==> yet another kind of deadlock!

  Solution: release the lock inside of the loop.
    [slide: yield_wait()]
    Allows another CPU's notify to acquire t_lock and make some thread RUNNABLE.

  Why do we need to also switch SP twice now?
    Once yield_wait() releases t_lock, another CPU can run our current thread.
    So now two CPUs have their SP pointing to the same thread's stack!
    If both access the stack, can result in the stack being corrupted.
    Fix: allocate a separate stack for use inside of yield_wait(), for each CPU.
    Switch to this special stack before releasing t_lock.

Thus far, seen how to yield CPU to another thread.

But what if if a thread never calls yield?

So far: that thread will keep running forever, no other thread will run.
Next goal: force every thread to yield periodically.

<u>Preemption:</u>
Use special hardware ("timer") to periodically generate an interrupt.
Interrupt handler will forcibly call yield().

(show code)

```
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
```

As before, yield() switches stacks, and interrupt returns to another thread.
Need to save registers, because interrupt might not be at function boundary.
  But we can save them on the stack, and then just switch stacks as before.

<u>What happens if timer interrupt occurs when CPU is running yield / yield_wait?</u>
Problem: t_lock held!
  When timer tries to call yield(), acquire(t_lock) will hang forever.
Solution: hardware mechanism to disable interrupts.
  Disable interrupts before acquiring t_lock, re-enable after releasing it.

[summary slide]