6.033 Computer System Engineering
Spring 2009

6.033 Lecture 6: Client/server on one computer

Intro
  how to implement c/s on one computer
  valuable in itself
  involves concurrency, an independently interesting topic
    DP1 is all about concurrency

what do we want?
  [diagram: X client, X server, NO KERNEL YET]
  client wants to send e.g. image to server
  goal: arms-length, so X srvr not vulnerable to client bugs

idea: let kernel manage interaction
  client/srvr interact w/ trusted kernel, not each other
  [diagram: X client, X server, kernel]
  let's focus on one-way flow (can use two for RPC)
  buffer memory in kernel
    each entry holds a message pointer
  send(m) to add msg to buffer
  receive(m) to read msg out of buffer
  finite buffer, so send() may have to wait
  buffer may be empty, so receive() may have to wait
  why does buffer have multiple entries?
    sender / receiver rates may vary around an average
    let sender accumulate a backlog when it's faster
      so receiver has input when sender is slower
  very much like a UNIX pipe

problem: concurrency
  some data structure inside kernel, s() and r() modify it
  what if s() and r() active at same time? may interfere
  concurrency a giant problem, will come up many times
  let's start simple:
    each program gets its own CPU
    there is only one memory system
  [diagram: two CPUs, one memory]
  system calls run on both CPUs!
    i.e. if program A calls send(), send() runs on A's CPU
  send() and receive() interact via single shared memory system

data structure
  "bounded buffer"
  [diagram: BUFFER[5], IN, OUT]
  each array entry: pointer to message buffer
  IN: number of messages put into BB
  OUT: number of messages read out of BB
  IN mod 5 is next place for send() to write
  OUT mod 5 is next place for receive() to look
  example: in = 28, out = 26
    two messages waiting, slots 1 and 2
  in > out => BB not empty
  in - out < 5 => not full

send() code slide
  p is "port", points to instance of BB, so we can have many of them

```
    e.g. one per c/s pair, or one per UNIX pipe
  loop to wait until room ("busy-wait")
  write slot
  increment input count

receive() code slide
  loop to wait until more sends than recvs
  if there's a message
  increment p.out AFTER copying msg
    since p.out++ may signal send() to overwrite
    if send() is waiting for space

I believe this simple BB code works
  [show slide with both]
  even if send() and receive() called at same time
  concurrency rarely work out this well!

Assumptions for simple BB send/recv
  1. One sender, one receiver
  2. Each has its own CPU (otherwise loop prevents other from running)
  3. in and out don't overflow
  4. CPUs perform mem reads and writes in the order shown
     oops! this code probably won't work as shown!
     compiler might put in/out in regs, not see other's changes
     CPU might increment p.in before writing buffer[]
     I will assume memory R/W in program order

Suppose we want multiple senders
  e.g. so many clients can send msgs to X server
  would our send() work?

Concurrent send()
  A: send(p, m1) B: send(p, m2)
  what do we *want* to happen?
  what would be the most useful behavior?
  goal:
    two msgs in buf, in == 2
    we don't care about order

Example prob w/ concurrent send()
  on different cpus, at the same time, on the same p
  A          B
  r in, out
             r in, out
  w buf[0]
             w buf[0]
  r in=0
             r in=0
  w in=1
             w in=1
  result: in = 1, one message in bounded buffer, and one was lost!

This kind of bug is called a "race"
  once A puts data in buffer, it has to hurry to finish w/ incr of p.in!

Other races in this code
```

```
  suppose only one slot left
  A and B may both observe in - out < N
  put *two* items in buf, overwrite oldest entry

Races are a serious problem
  easy mistake to make -- send() looks perfectly reasonable!
  hard to find
    depends on timing, may arise infrequently
    e.g. Therac-25, only experienced operator typed fast enough

How to fix send()'s races?
  original code assumed no-one else messing w/ p.in &c
    only one CPU at a time in send()
    == isolated execution
  can we restore that isolation?

Locks
  a lock is a data type with two operations
    acquire(l)
    s1
    s2
    release(l)
  the lock contains state: locked or unlocked
  if you try to acquire a locked lock
    acquire will wait until it's released
  if two acquire()s try to get a lock at same time
    one will succeed, the other will wait

How to fix send() with locking?
  [locking send() slide]
  associate a lock w/ each BB
  acquire before using BB
    release only after done using BB
  high-level view:
    no interleaving of multiple send()s
    only one send() will be executing guts of send()
    likely to be correct if single-sender send() was correct

Does it matter how send() uses the lock?
  move acquire after IF? [slide]

Why separate lock per bounded buffer?
  rather than e.g. all BBs using same lock
    that would allow only one BB to be active
  but it's OK if send()s on different BBs are concurrent
  lock-per-BB improves performance / parallelism

Deadlock
  big program can have thousands of locks, some per module
  once you have more than one lock in your system,
    you have to worry about deadlock
  deadlock: two CPUs each have a lock, each waiting for other to release
  example:
    implementing a file system
    need to ensure two programs don't modify a directory at the same time
    have a lock per directory
```

```
      create(d, name):
        acquire d.lock
        if name exists:
          error
        else
          create dir ent for name
        release d.lock
    what about moving a file from one dir to another? like mv
      move(d1, name, d2):
        acquire d1.lock
        acquire d2.lock
        delete name from d1
        add name to d2
        release d2.lock
        release d1.lock
    what is the problem here?

Avoiding deadlock
  look for all places where multiple locks are held
  make sure, for every place, they are acquired in the same order
    then there can be no locking cycles, and no deadlock
  for move():
    sort directories by i-number
    lock lowere i-number first
    so:
      if d1.inum < d2.inum:
        acquire d1.lock
        acquire d2.lock
      else:
        acquire d2.lock
        acquire d1.lock
  this can be painful: requires global reasoning
    acquire l1
    print("...")
  does print() acquire a lock? could it deadlock w/ l1?
  the good news is that deadlocks are not subtle once they occur

Lock granularity
  how to decide how many locks to have, what they should protect?
  a spectrum, coarse vs fine
  coarse:
    just one lock, or one lock per module
    e.g. one lock for whole file system
    more likely correct
    but CPUs may wait/spin for lock, wasting CPU time
    "serial execution", performance no better than one CPU
  fine:
    split up data into many pieces, each with a separate lock
    different CPUs can use different data, different locks
      operate in parallel, more work gets done
    but harder to get correct
    more thought to be sure ops on different pieces don't interact
      e.g. deadlock when moving between directories
  always start as coarse as possible!
    use multiple locks only if you are forced to, by low parallel performance
```

```
How to implement acquire and release?
  Here's a plan that DOES NOT WORK:
    acquire(l)
      while l == 0
        do nothing
      l = 1
    release(l)
      l = 0
  Has a familiar race:
    A and B both see l = 0
    A and B both set l = 1
    A and B both hold the lock!

If only we could make l==0 test and l=1 indivisible...
  most CPUs provide the indivisible instruction we need!
  differs by CPU, but usually similar to:
    RSM(a)
      r <- mem[a]
      mem[a] <- 1
      return r
  sets memory to 1, returns old value
  RSM = Read and Set Memory

How does h/w make RSM indivisible?
  a simple plan
  two CPUs, a bus, memory
  only one bus, only one CPU can use it
  there's an arbiter that decides
  so CPU grabs bus, does read AND write, releases bus
  arbiter forces one RSM to finish before other can start

How to use RSM for locking?
  acquire(l)
    while RSM(l) == 1
      do nothing
  always sets lock to 1
    if already locked: harmless
    if not locked: locks
  RSM returns 0 iff not already locked
  only one of a set of concurrent CPUs will see 0
  tidbit: you can implement locks w/ ordinary LOADs and STOREs
    i.e. w/o hardware-supported RSM
    it's just awkward and slow
    look up Dekker's Algorithm

Summary
  BB is what you need for client/server on one computer
  Concurrent programming is tough
  Locks can help make concurrency look more sequential
  Watch out for deadlock
  Next: more than one program per CPU
```