Today's plan:
- Any questions about lock server lab?
- Reviewing event driven programming
- Outline structure of the remaining labs
- Common libasync/libarpc/nfsloop programming idioms:
  - writing rpc client code
  - writing async functions that call RPCs
  - writing rpc server code
- Flash

Event driven programming
- Achieve I/O concurrency for communication efficiently
- Threads give cpu *and* i/o concurrency
- Never quote clear when you'll context switch: cpu+i/o concurrency
- State machine style execution
  - Lots of "threads": request handling state machines in parallel
  - Single address space: no context switch overhead ==> efficient
  - Have kernel notify us of I/O events that we can handle w/o blocking
- The point: this preserves the serial natures of the events
  - Programmer sees events/functions occuring one at a time
  - Simplifies locking (but when do you still need it?)

- libasync handles most of the busywork
  - [draw amain/select on board again]
  - e.g. write-ability events are usually boring
- libarpc translates to events that the programmer might care about:
  - rpcs

ccfs architecture:
- [draw block diagram on the board:]
  - OS [app, ccfs] --> blockserver <-- [ccfs, app] OS
  - \--> lockserver <--/
- ccfs communicate through RPC: you'll be writing clients and servers
- [include names of RPCs on the little lines]
- real apps can be structured just like this: okws, chord/dhash

Synchronous RPC:
- [Example 1]
  - [Sketch this on the board and use it to show evolution]

Making RPCs
- Already saw basic framework in Lab 1
- libarpc provides an rpc compiler: protocol.x -> .C and .h
  - Provides (un)marshalling of structs into strings
  - External Data Representation, XDR (rfc1832)
- [Example 2]
  - libraries to help:
    - handle the network (axprt: asynchronous transport)
    - write clients (aclnt),
      - aclnt handles all bookkeeping/formatting/etc for us:
        - e.g. which cb gets called
    - write servers (asrv/svccb)

Asynchronous RPC: needs a callback!
- [Example 3]
  - Note:

Cite as: Robert Morris, course materials for 6.824 Distributed Computer Systems Engineering, Spring 2006. MIT OpenCourseWare (http://ocw.mit.edu/), Massachusetts Institute of Technology. Downloaded on [DD Month YYYY].
1. Need to split code into separate functions: need to declare prototypes
2. "return values" passed in by aclnt as arguments: e.g. clnt_stat
3. cb must keep track of where results will be stored.
4. Actually must split everything that uses an async function!

How do we translate this into a stub function?
   Need to provide our own callback....
   [Example 4]
...translate RPC results/error into something the app can use.

Server side:
   Setup involves listening on a socket, allocating a server with dispatch cb

   [Example 5]
dispatch (svccb *sbp):
   switch to dispatch on sbp->proc ();
call sbp->reply (res);

   You must not block when handling proc ()
   you don't need to reply right away but blocking would be bad

Managing memory with svccb:
   Use getarg<type> to get pointer to argument, svccb managed
   Use getres<type> to get a pointer to a reply struct, svccb managed
   sbp->reply causes the sbp to get deleted.

Writing user-level NFS servers:
   classfsd code will allow you to mount a local NFS server w/o root
   nfsserv_udp handles tedious work, we register a dispatch function
   Similar to generic RPC server but use nfscall *, instead of svccb.
   Adds features like nc->error ()

You'll need to do multiple operations to handle each RPC
   [draw RPC issue timeline os->kernel->ccfs->lockserver/blockserver]
Not unlike how we might operate:
   get an e-mail from friend: can you make it to my wedding?
   check class calendar on web, check research deadlines
   send IM to wife, research ticket prices, reply
   Or Amazon.com login...
   [Example 6]

An aside on locking:
   No locking etc needed usually: e.g. to increment a variable
   When do you need locking?
   When an operation involving multiple stages
   Be careful about callbacks that are supposed to happen "later"
   e.g. delaycb (send_grant);

Parallelism and loops
   [Example 7a]: synchronous code
   [Example 7b]: serialized and async
   [Example 7c]: parallelism but yet...
   [Example 7d]: better parallelism?

Summary
Cite as: Robert Morris, course materials for 6.824 Distributed Computer Systems Engineering, Spring 2006. MIT OpenCourseWare (http://ocw.mit.edu/), Massachusetts Institute of Technology. Downloaded on [DD Month YYYY].
Events programming gives programmer a view that is roughly consistent with what happens.
Can build abstractions to handle app level events
Need to break up state and program flow
  but always know when there's a wait,
  and have good control over parallelism