```
a2ps -A fill -o x --line-numbers=1 events.c webclient.c
webclient_libasync.c

Office hours
Discussion mailing list
(Wiki?)

****
Recall simple web-like server?
  Now we'll look at a client for the server
  Show how to write some asynchronous code
  Build up to libasync which you will and have been using for labs

  [webclient.c]

Where does this block?
  connect: makes a tcp connection
  write(s): is remote side willing to take data?
  read(s): has data come back from the remote side?
  write(1): is terminal ready for output?

How to program in event style?
  Identify events and appropriate responses: state machine
    programmer has to know when something might block!
  Write a loop that handles incoming events (I/O events)
  [events.c Example 1]

select()
  Need a way to multiplex sockets
  Program must then interleave operations

  [write prototype on the board: nfds, reads, writes, excepts, time]
  Condition is actually "read() would not block"; might be EOF.
  select() blocks (if timeout > 0) to avoid wasteful polling.
    this is important; you *do* want to block in select().

Translate low-level system events into application level events
  Buffer net I/O, maintain individual application state
  Writing this event loop for each program is tedious
  [sketch implementation on the board]
  What if your program does the one thing in parallel?
    Have to partition up state for each client
    Need to maintain sets of file descriptors
  What if your program does many things?
    e.g. let's add DNS resolution
  Hard to be modular if event loop knows about all activities.
  And knows how to consult all state.

We would prefer abstraction...
  Use a library to provide main loop (e.g. libasync)
  Programmer provides "callbacks" to handle events
  [events.c: Example 2]

Break up code into functions with non-blocking ops
  let the library handle the boring async stuff
  [prototypes in webclient_libasync.c]
```

It's unfortunately hard for async programs to maintain state
  [draw logical diagram of select loop and function calls]
  Ordinary programs, and threads, use variables.
    Which persist across function calls, and blocking operations.
    Since they are stored on the stack.
  Async programs can't keep state on the stack.
    Since each callback must return immediately.

How can they maintain state across calls?
  Use global variables
  Use the heap:
    Programmers package up state in a struct, malloc struct
    Each callback could take a void * (libevent)
    (In C++, can do this somewhat implicitly using an object.)
  This turns out to be hard to program
    No type safety
    Must declare structs for every set of state transfer
    User has to manage memory in potentially tricky cases
  libasync provides a form of closures

cb = wrap(fn, a, b) generates a closure.
  That is, a function pointer plus an environment of saved values.
      cb() calls fn(a, b)
  Also provides something like function currying.
  useful later on when callbacks do different things based on input
  Given a function with signature "R fn (A, B)":
    cb = wrap (fn) -> callback<R, A, B>::ref
  use it like this:
    cb (a, b)
  Or:
    wrap (fn, a) -> callback<R, B>::ref

Limited compared to Scheme closures:
  You must explicitly indicate what variables to keep around.
  Can only pass a certain number of arguments

How are callbacks implemented?
  See callback.h: one of the few commented files in libasync.
  templates to generate dynamic structs holding values
  templates provide type safety:

```
    R fn (A, B);
    cb = wrap (fn) -> callback<R, A, B>::ref  cb (a, b)
    cb = wrap (fn, a) -> callback<R, B>::ref  cb (b);
    cb = wrap (fn, a, b) -> callback<R>::ref  cb ();
```

  callbacks are reference counted to simplify mem mgmt
    normally, arguments in the wrap would have been on stack
    now, values are stored in closures created by wrap().
    How do we know when we've used a callback the last time?
    That's why they're reference counted.

What is the result?
  [webclient_libasync.c]
  what's the difference between filename and buf?

This is still somewhat tedious...
  Must handle memory allocation for strings

Must manually buffer data to and from client
Have to translate network read/writes into application level events

libasync provides some solutions:
suio and aios handle raw and line-oriented i/o
reference counted data (strings and general dynamic structs)
asynchronous RPC library
but you still have to do work like splitting your code up into functions
loops can still be a pain