Announcements

- **Important Dates**
  - Assignment 6 due Wednesday night at 11:59 p.m.
    - You can hand the assignment in without penalty until this Sunday at 11:59 p.m.
    - You can use up to two late days (Monday is one late day, Tuesday is two late days) even on this last assignment.
  - Final Exam is June 10th at 8:30 a.m. in Hewlett 200.
    - Will cover all material taught in Lectures 1 through 25 in depth, and will expect surface understanding of the nonblocking I/O material I'm covering today and next week.
    - Exam is closed notes, closed book, closed computer, but you can bring and refer to two 8.5" x 11" sheets of paper with as much as you can cram onto their four sides, front and back.
    - Three practice final exams and solutions are available online now as Handout 12. Several students asked that I provide them early, so here I am doing that.
- **This week's topics:**
  - Non-blocking I/O and event-driven programming (**`epoll`**, **`kqueue`**, and **`libev`**/**`libuv`** packages), cross-language compilation, the Tornado web server, node.js and Google's V8 engine.
  - Will teach lecture today and Monday for sure, Wednesday if needed.

Fast and Slow System Calls

- **The first week of class, we learned about special functions called system calls.**
  - Fast system calls are those that return immediately, where "immediately" means they just need the processor and other local resources to get their work done.
    - There's no hard limit on the time they're allowed to take.
    - Even if a system call were to take 60 seconds to complete, it'd be considered fast if all 60 seconds were spent executing code (i.e. no idle time blocking on external resources.)
  - Slow system calls are those that wait for an indefinite period of time for something to finish (e.g. **`waitpid`**), for something to become available (e.g. **`read`** from a client socket that's not seen any data recently), or for some external event (e.g. network connection request from client via **`accept`**.)
    - Calls to **`read`** are considered fast if they're reading from a local file, because there aren't really any external resources to prevent read from doing it's work. It's true that some hardware needs to be accessed, but because that hardware is grafted into the machine, we can say with some confidence that the data being read from the local file will be available within a certain amount of time.
    - Calls to **`write`** are considered slow if data if being published to a socket and previously published data has congested internal buffers and not been pushed off the machine yet.
  - Slow system calls are the ones capable of blocking a thread of execution indefinitely, rendering that thread inert until the call is able to return.
    - We've relied on signals and signal handlers to take calls to **`waitpid`** off the normal flow of execution, and we've also relied on the **`WNOHANG`** flag to ensure that **`waitpid`** never actually blocks. That's an example of nonblocking. We just didn't call it that then.
    - We've relied on multithreading to get calls to **`read`** and **`write`** off the main thread. (Note: remember that the **`iosocsktream`** class implementation layers over calls to **`read`** and **`write`**.)
      - Threading doesn't make the calls to **`read`** and **`write`** any faster, but it does parallelize the stall times, and it also frees up the main thread so that it can, if it chooses, focus on CPU-bound computation.
      - You should be intimiately familiar with these ideas based on your work with **`news-aggregator`** and **`http-proxy`**.
  - **`accept`** and **`read`** are the system calls that everyone always identifies as slow.

Making Slow System Calls Fast

- ### Configuring a descriptor as nonblocking.
  - It's possible to configure a descriptor to be what's called **nonblocking**. When nonblocking descriptors are passed to **`accept`**, **`read`**, or **`write`**, the function will always return as quickly as possible without waiting for anything external.
  - **`accept`**
    - If a client connection is available when **`accept`** is called, it'll return immediately with a socket connection to that client.
    - Provided the server socket is configured to be nonblocking, **`accept`** will return -1 instead of blocking if there are no pending connection requests. The -1 normally denotes that some error occurred, but if the **`errno`** global is set to **`EWOULDBLOCK`**, the -1 isn't ***really*** identifying an error, but instead saying that **`accept`** would have blocked had the server socket passed to it been a traditional (i.e. blocking) socket descriptor.
  - **`read`**
    - If one or more bytes of data are immediately available when **`read`** is called, then those bytes (or at least some of them) are written into the supplied character buffer, and the number of bytes placed is returned.
    - If no data is available (and the descriptor is still open and the other end of the descriptor hasn't been shut down,) then **`read`** will return -1, provided the descriptor has been configured to be nonblocking. Again, this -1 normally denotes that some error occurred, but in this case, the **`errno`** global is set to **`EWOULDBLOCK`**. That's our clue that **`read`** didn't really fail. The -1/**`EWOULDBLOCK`** combination is just saying that the call to **`read`** ***would have*** blocked had the descriptor been a traditional (i.e. blocking) one.

Case study: `OutboundFile` class

- **The `OutboundFile` class is designed to read a local file and push its contents out over a supplied descriptor, and to do so without ever blocking.**
  - Here's an abbreviated version of the interface file:

```
class OutboundFile {
 public:
  OutboundFile();
  void initialize(const std::string& source, int sink);
  bool sendMoreData();

 private:
  // implementation details ommitted for the moment
```

  - The constructor just defaultly constructs an instance of the **`OutboundFile`** class.
  - The **`initialize`** method identifies what local file should be used as a source of data and the descriptor where that data should be written verbatim.
  - The **`sendMoreData`** method pushes as much data as possible to the supplied sink, without blocking. It returns **`true`** if it's possible there's more data to be sent, and **`false`** if all data has been fully pushed out.
  - The full interface file (which leaks some implementation details, because you see the private section) is right here.

Unit test for `OutboundFile`

- **Here's a simple program I use to ensure that the `OutboundFile` class is working.**
  - It's a simple program prints the source of the unit test to standard output. #meta
  - There's a copy of the code right here.

```
/**
 * File: outbound-file-test.cc
 * ---------------------------
 * Demonstrates how one should use the OutboundFile class
 * and can be used to confirm that it works properly.
 */

#include "outbound-file.h"

int main(int argc, char *argv[]) {
  OutboundFile obf;
  obf.initialize("outbound-file-test.cc", STDOUT_FILENO);
  while (obf.sendMoreData()) {;}
  return 0;
}
```

Static File Server

- **Now consider the following server.**
  - This is a program that implements a nonblocking server that happily serves up a copy of the server code itself to the client.
  - A full copy of the program is right here.
  - And here's a copy of the program right here:

```
static const unsigned short kDefaultPort = 12345;
static const int kDefaultBacklog = 128;
static const string kFileToServe("expensive-server.cc");
int main(int argc, char **argv) {
  int serverSocket = createServerSocket(kDefaultPort, kDefaultBacklog);
  if (serverSocket == kServerSocketFailure) {
    cerr << "Could not start server.  Port " << kDefaultPort << " is probably in use." << endl;
    return 0;
  }

  setAsNonBlocking(serverSocket);
  cout << "Static file server listening on port " << kDefaultPort << "." << endl;
  list<OutboundFile> outboundFiles;
  size_t numConnections = 0;
  size_t numActiveConnections = 0;

  while (true) {
    int clientSocket = accept(serverSocket, NULL, NULL);
    if (clientSocket == -1) {
      assert(errno == EWOULDBLOCK);
    } else {
      OutboundFile obf;
      obf.initialize(kFileToServe, clientSocket);
      outboundFiles.push_back(obf);
      cout << "Connection #" << ++numConnections << endl;
      cout << "Queue size: " << ++numActiveConnections << endl;
    }

    auto iter = outboundFiles.begin();
    while (iter != outboundFiles.end()) {
      if (iter->sendMoreData()) {
        ++iter;
      } else {
        iter = outboundFiles.erase(iter);
        cout << "Queue size: " << --numActiveConnections << endl;
      }
    }
  }
}
```

  - The implementation of **`setAsNonBlocking`** is UNIX gobbledygook, and it's right here:

```
void setAsNonBlocking(int descriptor) {
  int flags = fcntl(descriptor, F_GETFL);
  if (flags == -1) flags = 0; // if first call to fcntl fails, just go with 0
  fcntl(descriptor, F_SETFL, flags | O_NONBLOCK); // preserve other set flags
}
```