Announcements

- **Assignment 4 Due Tonight**
- **Assignment 5 Out Today, Due Tuesday, May 26th**
  - Assignment 5 has you implement a bona fide HTTP web proxy and cache, using the networking material you've learned over the course of the past several lectures.
  - Assignment 5 is nontrivial, but made much easier because you paid it forward by implementing your **`ThreadPool`** these past ten days.
- **Today's Agenda**
  - Applaud Michael Chang
  - Finish discussion of socket descriptors, socket addresses, and how to write a simple network client. We'll rely on examples presented in this past Wednesday's slide deck for those examples.
  - Advance on to the material presented in this slide deck, which discusses server sockets, talks about how to implement a server, and defends why CS110 teaches threading before I teach networking.

Writing Servers

- **Operative Metaphor**
  - Colloquially, server-side applications wait by the phone (at a particular extension), praying that someone—no, anyone!!—will call.
  - Formally, server-side applications...
    - create a socket, just as client applications do.
      - we'll stick with the **`AF_INET`** socket family and confine ourselves to IPv4 Internet addresses.
      - we'll stick with the **`SOCK_STREAM`** socket type so we can expect all incoming and outgoing data to be reliably delivered without loss. (**`SOCK_DGRAM`** was another option, but that allows for occasional packet loss, and we don't want that).
    - bind the server socket to a specific port, generally (but not always) to any of the host's associated IP addresses.
    - listen for incoming connection requests, and accept them as they come in.

Writing Servers (continued)

Creating a Server Socket

- Provided a server is prepared to listen to a specified port on any of its IP addresses, the following function returns a properly configured server socket:

```
static const int kReuseAddresses = 1; // 1 means true here
int createServerSocket(unsigned short port, int backlog) {
  int serverSocket = socket(AF_INET, SOCK_STREAM, 0);
  if (serverSocket < 0) return kServerSocketFailure;
  if (setsockopt(serverSocket, SOL_SOCKET, SO_REUSEADDR,
                 &kReuseAddresses, sizeof(int)) < 0) {
    close(serverSocket);
    return kServerSocketFailure;
  } // setsockopt used here so port becomes available even is server crashes and reboots

  struct sockaddr_in serverAddress; // IPv4-style socket address
  memset(&serverAddress, 0, sizeof(serverAddress));
  serverAddress.sin_family = AF_INET; // sin_family field used to self-identify sockaddr type
  serverAddress.sin_addr.s_addr = htonl(INADDR_ANY);
  serverAddress.sin_port = htons(port);

  if (bind(serverSocket, (struct sockaddr *) &serverAddress,
           sizeof(struct sockaddr_in)) < 0) {
    close(serverSocket);
    return kServerSocketFailure;
  } // bind the unbound socket to the (address, port) pair

  if (listen(serverSocket, backlog) < 0) {
    close(serverSocket);
    return kServerSocketFailure;
  } // listen through the server socket, allowing a backlog of the specified size

  return serverSocket;
}
```

- Note that **`createServerSocket`** returns a socket we listen to for incoming connections.

Baby's First Server

- **A time server!**
  - Full version of the server code (minus the **`createServerSocket`** interface and implementation) can be found right here.
  - Our time server accepts all incoming connection requests and quickly publishes the time without so much as listening to a word that the client has to say.

```
static const int kMaxQueueSize = 10;
int main(int argc, char *argv[]) {
  // oodles of error checking omitted for brevity (real version has it all, though)
  unsigned short port = extractPort(argv[1]);
  int serverSocket = createServerSocket(port, kMaxQueueSize);
  cout << "Server listening on port " << port << "." << endl;
  while (true) {
    struct sockaddr_in clientAddress;
    socklen_t clientAddressSize = sizeof(clientAddress);
    memset(&clientAddress, 0, clientAddressSize);
    int clientSocket =
      accept(serverSocket, (struct sockaddr *) &clientAddress,
          &clientAddressSize);
    cout << "Received a connection request from "
         << inet_ntoa(clientAddress.sin_addr) << "." << endl;
    publishTime(clientSocket);
  }

  return 0; // execution never gets here, but compiler might not know that
}
```

Baby's First Server (continued)

- **The implementation of `publishTime` is straightforward.**
  - The implementation itself, however, isn't the focus. Strictly speaking, it's generating dynamic content—uninteresting, but dynamic, nonetheless—and publishing it back to the client over the socket descriptor.

```
static void publishTime(int clientSocket) {
  time_t rawtime;
  time(&rawtime);
  struct tm *ptm = gmtime(&rawtime);
  char timeString[64]; // more than big enough
  /* size_t len = */ strftime(timeString, sizeof(timeString), "%c", ptm);
  sockbuf sb(clientSocket); // destructor closes socket
  iosockstream ss(&sb);
  ss << timeString << endl;
}
```

  - We rely on some C library functions to generate a time string, and we insert that string into an **`iosockstream`** that itself layers over the client socket.
    - Note that the intermediary **`sockbuf`** class takes ownership of the socket and closes it when its destructor is called.
    - Of note, the **`gmtime`** function returns a pointer to statically allocated data, which means that we need to extract the time information before **`gmtime`** is called again. Restated, because statically allocated data is involved, **`gmtime`** is not threadsafe. That's not a problem here, because we don't have any threads in this particular example. #adumbrationale

Server-Side Networking and Threading

- **Networking and threading belong together like peanut butter and jelly.**
  - The work a server needs to do in order to meet the client's request might be time-consuming—so time consuming that a sequential implementation might interfere with the server's ability to accept future requests.
  - The **`listen`** command can be configured to allow pending connection requests to queue up, but they can only queue up to a maximum amount. If the pending connection queue fills up—that is, it becomes congested—the server is then authorized to refuse connection requests it doesn't foresee handling any time soon.
  - One solution: as soon as **`accept`** returns a socket descriptor, spawn a child thread to get any intense, pseudo-blocking computation off of the main thread. The child thread can make use of a second processor or a second core, and the main thread can quickly move on to its next **`accept`** call.

Server-Side Networking and Threading (continued)

- **Networking and threading belong together like Simon and Paula.**
  - Here's the same time server example, save for its decision to use *threading* to compute and publish the time back to the client.

```
static const int kMaxQueueSize = 10;
int main(int argc, char *argv[]) {
  // oodles of error checking omitted for brevity (real version has it all, though)
  unsigned short port = extractPort(argv[1]);
  int serverSocket = createServerSocket(port, kMaxQueueSize);
  cout << "Server listening on port " << port << "." << endl;
  ThreadPool pool(4);
  while (true) {
    struct sockaddr_in clientAddress;
    socklen_t clientAddressSize = sizeof(clientAddress);
    memset(&clientAddress, 0, clientAddressSize);
    int clientSocket =
      accept(serverSocket, (struct sockaddr *) &clientAddress,
             &clientAddressSize);
    cout << "Received a connection request from "
         << inet_ntoa(clientAddress.sin_addr) << "." << endl;
    pool.schedule([clientSocket] {
      publishTime(clientSocket)
    });
  }

  return 0; // execution never gets here, but compiler might know now that
}
```

  - Note the use of a **`ThreadPool`** to get the server-side computation off of the the main thread. This way, the main thread can rotate around and more immediately advance on to other **`accept`** requests.

Server-Side Networking and Threading (continued)

- **Networking and threading belong together like Pepper and Iron Man.**
  - The implementation of **`publishTime` must** change to be threadsafe. The change is simple but important: we need to call a reentrant, threadsafe version of **`gmtime`** called **`gmtime_r`**.
    - **`gmtime`** returns a pointer to a single, statically allocated record of time information that's used by all calls to it. If two threads make competing calls to it, then both threads will race to pull time information from the shared, statically allocated record.
    - One solution is to use a **`mutex`** to ensure that a thread can call **`gmtime`** without competition and subsequently copy the data out of the static record into a local character buffer.
    - Another solution—one that doesn't require locking and one I think is better—makes use of a second version of the same function called **`gmtime_r`**. The second, reentrant version just requires that space for a dedicated return value be passed in.

```
static void publishTime(int clientSocket) {
  time_t rawtime;
  time(&rawtime);
  struct tm ptm;
  gmtime_r(&rawtime, &ptm);
  char timeString[64]; // more than big enough
  /* size_t len = */ strftime(timeString, sizeof(timeString), "%c", &ptm);
  sockbuf sb(clientSocket); // destructor closes socket
  iosockstream ss(&sb);
  ss << timeString << endl;
}
```