Announcements

- **Important Dates**
  - Assignment 6 officially due Wednesday at 11:59 p.m., although we'll accept it without penalty until Sunday night at 11:59 p.m.
    - Beyond Sunday, you can use up to two late days as you would on any assignment.
    - We won't be able to grade Assignment 6 submissions prior to the final exam.
  - Final Exam is June 10th at 8:30 a.m. in Hewlett 200.
    - Will cover all material taught in all lectures, though only a surface understanding of nonblocking I/O will be expected of you.
    - 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 already up online as Handout 12.
- **Today's topics:**
  - Finish up non-blocking I/O and event-driven programming (**`epoll`**, **`kqueue`**, and **`libev`**/**`libuv`** packages).
    - Review high-level implementation of **`OutboundFile`** class. The interface file is right here, and the implementation is right here.
    - Introduce the **`epoll`** package, which is a suite of three functions and a few data structures that help identify which nonblocking file descriptors are capable of surfacing and/or accepting actual data and not having **`accept`** or **`read`** return -1. The final example—an efficient, nonblocking web server—is presented right here.
  - Lecture today, we'll see about Wednesday.

Overview of `OutboundFile` class implementation

- **The full implementation**
  - The full implementation includes lots of spaghetti code.
  - In particular, true file descriptors and socket descriptors need to be treated differently in a few places—in particular, detecting when all data has been flushed out to the sink desciptor (which may be a local file, a console, or a remote client machine) isn't exactly pretty.
  - However, my implementation is decomposed well enough that I think many of the methods—the ones I'll show in lecture—are easy to follow and provide a clear narrative. At the very least, I'll surely convince you that the **`OutboundFile`** implementation is accessible to someone just finishing up CS110.

Overview of `OutboundFile` class implementation

- **The full implementation**
  - Here's is the condensed interface file for the **`OutboundFile`** class.

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

 private:
  int source, sink;
  static const size_t kBufferSize = 128;
  char buffer[kBufferSize];
  size_t numBytesAvailable;
  size_t numBytesSent;
  bool isSending;

  bool dataReadyToBeSent() const;
  void readMoreData();
  void writeMoreData();
  bool allDataFlushed();
};
```

  - This was presented in the last slide deck, except now I'm exposing the private data members.
    - **`source`** and **`sink`** are descriptors bound to the data source and the data recipient, and both are nonblocking.
    - **`buffer`** is a reasonably sized character array that helps shovel bytes lifted from the **`source`** via calls to **`read`** over to the **`sink`** via calls to **`write`**. We shouldn't be surprised that **`read`** and **`write`** come in to play.
    - **`numBytesAvailable`** stores the number of meaningful characters residing in **`buffer`**.
    - **`numBytesSent`** tracks the number of bytes residing in **`buffer`** that have been written to the recipient.
      - When **`numBytesAvailable`** and **`numBytesSent`** are equal, we know that **`buffer`** is effectively empty, and that perhaps another call to **`read`** is in order.
    - **`isSending`** tracks whether all data has been pulled from the **`source`** and pushed to the recipient **`sink`**.

Overview of `OutboundFile` class implementation

- **The full implementation**
  - Here's is enough of the **`OutboundFile`** implementation to make it clear how it works.

```
OutboundFile::OutboundFile() : isSending(false) {}

void OutboundFile::initialize(const string& source, int sink) {
  this->source = open(source.c_str(), O_RDONLY | O_NONBLOCK);
  this->sink = sink;
  setAsNonBlocking(this->sink);
  numBytesAvailable = numBytesSent = 0;
  isSending = true;
}
```

  - The implementations of the constructor and **`initialize`** are complete.
    - The **`source`** is always a file descriptor bound to some local file.
      - Note that the file is opened for reading (**`O_RDONLY`**), and the descriptor is configured to be nonblocking (**`O_NONBLOCK`**).
      - For reasons we discussed last time, it's not super important that the **`source`** be nonblocking, since it's bound to a local file. But in the spirit of a nonblocking example, it's fine to make it nonblocking anyway. We just should expect very many (if any) -1's to come back from the **`read`** calls.
    - The **`sink`** is explicitly marked as nonblocking, since we shouldn't require the client to convert it to nonblocking ahead of time.

Overview of `OutboundFile` class implementation

- **The full implementation**
  - Here's is enough of the **`OutboundFile`** implementation to make it clear how it works.

```
bool OutboundFile::sendMoreData() {
  if (!isSending) return !allDataFlushed();
  if (!dataReadyToBeSent()) {
    readMoreData();
    if (!dataReadyToBeSent()) return true;
  }
  writeMoreData();
  return true;
}
```

  - The implementation of **`sendMoreData`** is incomplete, but it's enough that you can understand the fully store. (Again, full implementation is right here).
    - Recall that **`sendMoreData`** returns **`false`** when no more calls to **`sendMoreData`** are needed, and **`true`** if it's unclear.
    - The first line detects the situation where are data has been read from the **`source`** and written to the **`sink`**, and it returns **`true`** unless it's able to further confirm that all of the data written to **`sink`** has actally arrived (i.e. we've confirmed that it's been flushed out to the final destination.)
    - The first call to **`dataReadyToBeSent`** checks to see if the **`buffer`** includes any characters that have yet to be pushed out. If not, then it attempts to **`readMoreData`**. If after reading more data the buffer is still empty—after all, we have have **`read`** in everything, or the call to **`read`** resulted in a -1/**`EWOULDBLOCK`** pair, then we return **`true`** as a statement that there's no data to be written, no need to try writing, but come back later to see if that changes.
    - The call to **`writeMoreData`** is an opportunity to push data out to the **`sink`**.
    - We return **`true`** at the end, because we need to come back to, see if there's more data to be read in, or if we're done reading and writing and we've also managed to flush everything out to the ultimate destination.

I/O-Event Driven Programming

- **Introducing the `epoll` I/O notification utilities.**
  - **`epoll`** is a package of Linux routines that help nonblocking servers yield the processor until it knows there's work to be done with one or more of the open client connections.
  - There are three functions in the **`epoll`** suite that I'll briefly introduce in lecture today.
    - **`epoll_create`**, which creates something called a watch set, which itself is a set of file descriptors which we'd like to monitor. The return value is itself a file descriptor used to identify a watch set. Because it's a file descriptor, watch sets can contain other watch sets. #deep
    - **`epoll_ctl`** is a control function that allows us to add descriptors to a watch set, remove descriptors from a watch set, and reconfigure file descriptors already in the watch set.
    - **`epoll_wait`** waits for I/O events, blocking the calling thread until one or more events are detected.
  - The example I'll work with in class is too large to put in the slide deck, so you should refer to an online copy. I'll run the new server in class, explain what it does, and cover key parts of the overall design and how it uses the **`epoll`** suite to efficiently operate as a nonblocking web server capable of managing tens of thousands of open connections at once.