# Lecture 18: Slow System Calls, Nonblocking I/O

- 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.
  - By this definition, there's no hard limit on the time they're allowed to take.
  - Even if a system call were to take 60s to complete, I'd consider it to be 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 stretch 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.
  - **`write`** calls are slow if data is being published to a socket and previously published data has congested internal buffers and not been pushed off the machine yet.

# Lecture 18: Slow System Calls, Nonblocking I/O

- Slow system calls are the ones capable of *blocking a thread of execution indefinitely*, rendering that thread inert until the system call returns.
  - We've relied on signals and signal handlers to lift calls to **`waitpid`** out of the normal flow of execution, and we've also relied on **`WNOHANG`** to ensure that **`waitpid`** never actually blocks.
    - That's what nonblocking means. We just didn't call it that back when we learned it.
  - We've relied on multithreading to get calls to **`read`** and **`write`**—calls to embedded within **`iosockstream`** operations—off the main thread.
    - Threading doesn't make the calls to **`read`** and **`write`** any faster, but it does parallelize the stall times free up the main thread so that it can work on other things.
    - You should be intimately familiar with these ideas based on your work with **`aggregate`** and **`proxy`**.
- **`accept`** and **`read`** are the I/O system calls that everyone always identifies as slow.

# Lecture 18: Slow System Calls, Nonblocking I/O

- Making Slow System Calls Fast
  - It's possible to configure a server socket to be *nonblocking*. When nonblocking server sockets are passed to **`accept`**, it'll always return as quickly as possible.
    - 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 **`errno`** 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.

# Lecture 18: Slow System Calls, Nonblocking I/O

- Making Slow System Calls Fast
  - It's possible to configure a traditional descriptor to be *nonblocking*. When nonblocking descriptors are passed to **`read`**, or **`write`**, it'll return as quickly as possible without waiting.
    - If one or more bytes of data are available when **`read`** is called, then some or all of those bytes are written into the supplied character buffer, and the number of bytes placed is returned.
    - If no data is available but the source of the data 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, **`errno`** 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.
    - Of course, if when **`read`** is called it's clear there'll never be any more data, **`read`** will return **`0`** as it has all along.
    - All of this applies to the **`write`** system call as well, though it's rare for **`write`** to return **`-1`** unless a genuine error occurred, even if the supplied descriptor is nonblocking.

# Lecture 18: Slow System Calls, Nonblocking I/O

- First example: consider the following **`slow-alphabet-server`** implementation:

```
static const string kAlphabet = "abcdefghijklmnopqrstuvwxyz";
static const useconds_t kDelay = 100000; // 100000 microseconds is 100 ms is 0.1 second
static void handleRequest(int client) {
    sockbuf sb(client);
    iosockstream ss(&sb);
    for (size_t i = 0; i < kAlphabet.size(); i++) {
        ss << kAlphabet[i] << flush;
        usleep(kDelay); // 100000 microseconds is 100 ms is 0.1 seconds
    }
}

static const unsigned short kSlowAlphabetServerPort = 41411;

int main(int argc, char *argv[]) {
    int server = createServerSocket(kSlowAlphabetServerPort);
    ThreadPool pool(128);
    while (true) {
        int client = accept(server, NULL, NULL);
        pool.schedule([client]() { handleRequest(client); });
    }
    return 0;
}
```

# Lecture 18: Slow System Calls, Nonblocking I/O

- **`slow-alphabet-server`** operates much like **`time-server-concurrent`** does.
- The protocol:
  - wait for an incoming connection
  - delegate responsibility to handle that connection to a worker within a **`ThreadPool`**
  - have worker very slowly spell out the alphabet over 2.6 seconds
  - close connection (which happens because the **`sockbuf`** is destroyed).
- There's nothing nonblocking about **`slow-alphabet-server`**, but it's intentionally slow to emulate the time a real server might take to synthesize a full response.
  - Many servers, like the one above, push out partial responses to the client. The accumulation of all partial responses becomes the full payload.
    - You should be familiar with the partial response concept if you've ever surfed YouTube or any other video content site like it, as you very well know that a video starts playing before the entire payload has been downloaded.

# Lecture 18: Slow System Calls, Nonblocking I/O

- Presented here is a traditional (i.e. blocking) client of the **`slow-alphabet-server`**:

```
int main(int argc, char *argv[]) {
    int client = createClientSocket("localhost", kSlowAlphabetServerPort);
    size_t numSuccessfulReads = 0;
    size_t numBytes = 0;
    while (true) {
        char ch;
        ssize_t count = read(client, &ch, 1);
        if (count == 0) break; // we are truly done
        numSuccessfulReads++;
        numBytes += count;
        cout << ch << flush;
    }
    close(client);
    cout << endl;
    cout << "Alphabet Length: " << numBytes << " bytes." << endl;
    cout << "Num reads: " << numSuccessfulReads << endl;
    return 0;
}
```

# Lecture 18: Slow System Calls, Nonblocking I/O

- Full implementation is right here:
  - Relies on traditional client socket as returned by **`createClientSocket`**.
  - Character buffer passed to **read** is of size 1, thereby constraining the range of legitimate return values to be [-1, 1].
  - Provided the **slow-alphabet-server** is running, a client run on the same machine would reliably behave as follows:

```
myth57:$ ./slow-alphabet-server &
[1] 7516
myth57:$ ./blocking-alphabet-client
abcdefghijklmnopqrstuvwxyz
Alphabet Length: 26 bytes.
Num reads: 26
myth57:$ time ./blocking-alphabet-client
abcdefghijklmnopqrstuvwxyz
Alphabet Length: 26 bytes.
Num reads: 26

real    0m2.609s
user    0m0.004s
sys     0m0.000s
myth57:$ kill -KILL 7516
[1] Killed              ./slow-alphabet-server
```

# Lecture 18: Slow System Calls, Nonblocking I/O

- Presented here is the nonblocking equivalent:

```
static const unsigned short kSlowAlphabetServerPort = 41411;
int main(int argc, char *argv[]) {
    int client = createClientSocket("localhost", kSlowAlphabetServerPort);
    setAsNonBlocking(client);
    size_t numReads = 0, numSuccessfulReads = 0, numUnsuccessfulReads = 0, numBytes = 0;
    while (true) {
        char ch;
        ssize_t count = read(client, &ch, 1);
        if (count == 0) break;
        numReads++;
        if (count > 0) {
            numSuccessfulReads++;
            numBytes += count;
            cout << ch << flush;
        } else {
            assert(errno == EWOULDBLOCK || errno == EAGAIN);
            numUnsuccessfulReads++;
        }
    }
    close(client);
    cout << endl;
    cout << "Alphabet Length: " << numBytes << " bytes." << endl;
    cout << "Num reads: " << numReads << " ("
         << numSuccessfulReads << " successful, "
         << numUnsuccessfulReads << " unsuccessful)." << endl;
    return 0;
}
```

# Lecture 18: Slow System Calls, Nonblocking I/O

- Full implementation is right here:
  - Because client socket is configured to be nonblocking, **`read`** is incapable of blocking.
    - Data available? Expect **`ch`** to be updated and for the **`read`** call to return 1.
    - No data available, ever? Expect **`read`** return a 0.
    - No data available right now, but possibly in the future? Expect a return value of -1 and **`errno`** to be set to **`EWOULDBLOCK`**

- Inspect the output of our nonblocking client.
  - Reasonable questions: Is this better? Why rely on nonblocking I/O other than because we can? We'll answer these questions very soon.
  - Interesting statistics, and just look at the number of **read** calls!

```
myth57:$ ./slow-alphabet-server &
[1] 9801
myth57:$ ./non-blocking-alphabet-client
abcdefghijklmnopqrstuvwxyz
Alphabet Length: 26 bytes.
Num reads: 11394589 (26 successful, 11394563 unsuccessful).
myth57:$ time ./non-blocking-alphabet-client
abcdefghijklmnopqrstuvwxyz
Alphabet Length: 26 bytes.
Num reads: 11268990 (26 successful, 11268964 unsuccessful).

real    0m2.607s
user    0m0.264s
sys     0m2.340s

myth57:$ kill -KILL 9801
[1] Killed          ./slow-alphabet-server myth57:$
```

# Lecture 18: Slow System Calls, Nonblocking I/O

- 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 interface file:

```
class OutboundFile {
    public:
        OutboundFile();
        void initialize(const std::string& source, int sink);
        bool sendMoreData();
    private:
        // implementation details omitted for the moment
}
```

- The constructor configures an instance of the **`OutboundFile`** class.
- **`initialize`** supplies the local file that should be used as a data source and the descriptor where that file's contents should be replicated.
- **`sendMoreData`** pushes as much data as possible to the supplied sink, without blocking. It returns **`true`** if it's at all possible there's more payload to be sent, and **`false`** if all data has been fully pushed out. The fully documented interface file is right here.

# Lecture 18: Slow System Calls, Nonblocking I/O

- Here's a simple program we can use to ensure the **OutboundFile** implementation is working:

```
/**
* 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"
#include <unistd.h>

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

- The above program prints its source to standard output.
- A full copy of the program can be found right here.

# Lecture 18: More Nonblocking I/O

- The **outbound-file-test.cc** presented earlier can be used to confirm the **`OutboundFile`** class implementation works as expected.
  - The nonblocking aspect of the code doesn't really buy us anything.
  - Only one copy of the source file is being syndicated, so no harm comes from blocking, since there's nothing else to do.
- To see why nonblocking I/O might be useful, consider the following nonblocking server implementation, presented over several slides.
  - Our server is a static file server and responds to every single request—no matter how that request is structured—with the same payload. That payload is drawn from an HTML called "**expensive-server.cc.html**".
- Presented below is the portion of the server implementation that establishes the executable as a server that listens to port 12345 and sets the server socket to be nonblocking.

```
// expensive-server.html is expensive because it's always using the CPU, even when there's nothing to do
static const unsigned short kDefaultPort = 12345;
static const string kFileToServe("expensive-server.cc.html");
int main(int argc, char **argv) {
    int server = createServerSocket(kDefaultPort);
    assert(server != kServerSocketFailure);
    setAsNonBlocking(server);
    cout << "Static file server listening on port " << kDefaultPort << "." << endl;
    // more code follows
```

# Lecture 18: More Nonblocking I/O

- As with all servers, our static file server loops interminably and aggressively accepts incoming connections as quickly as possible.
  - The first part of the while loop calls and immediately returns from **`accept`**. The return is immediate, because server has been configured to be nonblocking.
  - The code immediately following the accept call branches in one of two directions.
    - If **`accept`** returns a -1, we verify the -1 isn't something to be concerned about.
    - If **`accept`** surfaces a new connection , we create an new **`OutboundFile`** on its behalf and append it to the running **`outboundFiles`** list of clients currently being served.

```
list<OutboundFile> outboundFiles;
while (true) {
    // part 1: below
    int client = accept(server, NULL, NULL);
    if (client == -1) {
        assert(errno == EWOULDBLOCK); // confirm -1 isn't a true failure
    } else {
        OutboundFile obf;
        obf.initialize(kFileToServe, client);
        outboundFiles.push_back(obf);
    }
    // part 2: presented on next slide
```

# Lecture 18: More Nonblocking I/O

- As with all servers, our static file server loops interminably and aggressively accepts incoming connections as quickly as possible.
  - The second part executes whether or not part 1 produced a new client connection and extended the **`outboundFiles`** list.
  - It iterates over every single **`OutboundFile`** in the list and attempts to send some or all available data out to the client.
    - If **sendMoreData** returns **true**, the loop advances on to the next client via **`++iter`**.
    - If **sendMoreData** returns **false**, the relevant OutboundFile is removed from **`outboundFiles`** before advancing. (Fortunately, **`erase`** does precisely what we want, and it returns the iterator addressing the next **`OutboundFile`** in the list.)

```
    list<OutboundFile> outboundFiles;
    while (true) {
        // part 1: presented and discussed on previous slide
        // part 2: below
        auto iter = outboundFiles.begin();
        while (iter != outboundFiles.end()) {
            if (iter->sendMoreData()) ++iter;
            else iter = outboundFiles.erase(iter);
        }
    }
}
```

# Lecture 18: More Nonblocking I/O

- The code for **`setAsNonblocking`** is fairly low-level.
  - It relies on a function called **`fcntl`** to do surgery on the relevant file session in the open file table.
  - That surgery does little more than toggle some 0 bit to a 1, as can be inferred from the last line of the three line implementation.
- The code for **`setAsNonblocking`** and a few peer functions are presented below.

```
void setAsNonBlocking(int descriptor) {
    fcntl(descriptor, F_SETFL, fcntl(descriptor, F_GETFL) | O_NONBLOCK); // preserve other set flags
}

void setAsBlocking(int descriptor) {
    fcntl(descriptor, F_SETFL, fcntl(descriptor, F_GETFL) & ~O_NONBLOCK); // suppress blocking bit, preserve others
}

bool isNonBlocking(int descriptor) {
    return !isBlocking(descriptor);
}

bool isBlocking(int descriptor) {
    return (fcntl(descriptor, F_GETFL) & O_NONBLOCK) == 0;
}
```

# Lecture 18: More Nonblocking I/O

- We've been using the **`OutboundFile`** abstraction without understanding how it works behind the scenes.
  - We really should see the implementation (or at least part of it) so we have some sense how it works and can be implemented using nonblocking techniques.
  - 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 descriptor (which may be a local file, a console, or a remote client machine) isn't exactly pretty.
  - However, my (Jerry's) implementation is decomposed well enough that I think many of the methods—the ones that I'll show in lecture, anyway—are easy to follow and provide a clear narrative.
  - At the very least, I'll convince you that the **`OutboundFile`** implementation is accessible to someone just finishing up CS110.

# Lecture 18: More Nonblocking I/O

```
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, numBytesSent;
        bool isSending;
        // private helper methods discussed later
};
```

- Here's is the condensed interface file for the OutboundFile class.
- **`source`** and **`sink`** are nonblocking descriptors bound to the data source and recipient
- **`buffer`** is a reasonably sized character array that helps shovel bytes lifted from source via **`read`** calls over to the **`sink`** via **`write`** calls.
- **`numBytesAvailable`** stores the number of meaningful characters in **`buffer`**.
- **`numBytesSent`** tracks the portion of **`buffer`** that's been pushed to the recipient.
- **`isSending`** tracks whether all data has been pulled from **`source`** and pushed to **`sink`**.

# Lecture 18: More Nonblocking I/O

- The implementations of the constructor and initialize are straightforward:

```
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;
}
```

- **`source`** is a nonblocking file descriptor bound to some local file
  - Note that the source file is opened for reading (**`O_RDONLY`**), and the descriptor is configured to be nonblocking (**`O_NONBLOCK`**) right from the start.
  - For reasons we've discussed, it's not super important that 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 shouldn't expect very many (if any) -1's to come back from our **`read`** calls.
- **`sink`** is explicitly converted to be nonblocking, since it might be blocking, and **`sink`** will very often be a socket descriptor that really should be nonblocking.

# Lecture 18: More Nonblocking I/O

- The implementation of **sendMoreData** is less straightforward:

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

- The first line decides if all data has been read from **`source`** and written to **`sink`**, and if so, it returns **`true`** unless it further confirms all of data written to **`sink`** has arrived at final destination, in which case it returns **`false`** to state that syndication is complete.
- The first call to **`dataReadyToBeSent`** checks to see if **`buffer`** houses data yet to be pushed out. If not, then it attempts to **`readMoreData`**. If after reading more data the buffer is still empty—that is, a single 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, but come back later to see if that changes.
- The call to **`writeMoreData`** is an opportunity to push data out to sink.