Lecture 14: Nonblocking I/O

Note: Reading these lecture notes is not a substitute for watching the lecture. I frequently go off script, and you are responsible for understanding everything I talk about in lecture unless I specify otherwise.

Warmup: overload characteristics

Consider the two server implementations below, where the sequential `handleRequest` function always takes exactly 1.500 seconds to execute. The two servers would respond very differently if 1000 clients were to connect – one per 1.000 seconds – over a 1000 second window. What would the 500th client experience when it tried to connect to the first server? What would the 500th client experience when it tried to connect to the second?

```
// Server Implementation 1
int main(int argc, char *argv[]) {
   int server = createServerSocket(12345); // sets the backlog to 128
   while (true) {
      int client = accept(server, NULL, NULL);
      handleRequest(client);
   }
}

// Server Implementation 2
int main(int argc, char *argv[]) {
   ThreadPool pool(1);
   int server = createServerSocket(12346); // sets the backlog to 128
   while (true) {
      int client = accept(server, NULL, NULL);
      pool.schedule([client] { handleRequest(client); });
   }
}
```

Limitations of threading

So far, we have been using threads to overcome latency in network connections. However, threading has some limitations:

- Threads are expensive. If we want to handle 100 simultaneous connections, we can theoretically do it with threads, but the scheduler is going to be sad.
- There is a very real limit on the number of threads we can create. We simply can't handle 500 simultaneous connections with threads, even if each connection uses virtually no CPU time. (This is a very real situation, and happens with servers that handle HTTP long polling.)

Nonblocking I/O is a technique that allows us to juggle many connections within a single thread. (We can also share the work to multiple threads, if we'd like.)

Usually, if you call read() or write() on a file descriptor, they block until the operation is complete. We can instead configure a file descriptor to be *nonblocking,* so that read() and write() always return immediately (sort of like calling waitpid with WNOHANG). If there is nothing to be read, `read` and `write` will return `-1` with `errno` `EAGAIN`.

Basic nonblocking client

This client reads one character at a time from a server until it has received the entire alphabet. It's written using paradigms we're already used to:

```
int main(int argc, char *argv[]) {
    int client = createClientSocket("localhost", 12345);

    size_t numSuccessfulReads = 0;
    size_t numBytes = 0;
    while (true) {
        char ch;
        ssize_t count = read(client, &ch, 1);
        assert(count != -1); // simple sanity check, would be more robust in pract
        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;
}
```

If we configure the `client` file descriptor to be nonblocking, then `read` will return `-1` with errno `EAGAIN` if there's nothing ready for us to read. We can handle this:

```
int main(int argc, char *argv[]) {
    int client = createClientSocket("localhost", 12345);
    setAsNonBlocking(client);

    size_t numReads = 0;
    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
        numReads++;
        if (count > 0) {
            numSuccessfulReads++;
            numBytes += count;
            cout << ch << flush;
        } else {
            assert(errno == EWOULDBLOCK);
        }
    }

    close(client);
    cout << endl;
    cout << "Alphabet Length: " << numBytes << " bytes." << endl;
    cout << "Num reads: " << numSuccessfulReads << " of " << numReads << endl;
    return 0;
}
```

This implementation is actually *worse* than the first one, because it has 100% CPU utilization

during its execution. (Even though there's nothing to read, it keeps calling `read` over and over again.) We want file descriptors to be nonblocking so that we can juggle many of them and only read from file descriptors that have new bytes for us to read (just like we want to be able to call `waitpid(-1, ...)` to get updates only on children that have changed state), but we *do* want to block when there is truly nothing useful for us to be doing at the moment.

We can use the `epoll` set of functions for this, which is very similar to `sigsuspend`ing until SIGCHLD comes in.

- `epoll_create` creates a "watch set" of file descriptors. We can add a file descriptor to this set, then receive a notification when new bytes come in via that descriptor.
  - `int ws = epoll_create(1);`
  - This returns a file descriptor, which should be `close`d when we are finished.
- `epoll_ctl` modifies a watch set, either adding or removing or modifying descriptors in the set. (Think `sigaddset`.)
  - `int epoll_ctl(int epfd, int operation, int fd, struct epoll_event *event);`
- `epoll_wait` waits until there is activity on a file descriptor in the watch set. (Think `sigsuspend`.) It also returns a list of events, so that you can see exactly which file descriptors have updates.
  - `int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);`

Here's an updated client that wastes no `read` calls:

```
int main(int argc, char *argv[]) {
    int client = createClientSocket("localhost", 12345);
    setAsNonBlocking(client);
    int ws = epoll_create(1);
    struct epoll_event target;
    target.events = EPOLLIN;
    target.data.fd = client;
    epoll_ctl(ws, EPOLL_CTL_ADD, client, &target);

    size_t numReads = 0;
    size_t numSuccessfulReads = 0;
    size_t numBytes = 0;
    while (true) {
```

```
        struct epoll_event events[1];
        epoll_wait(ws, events, 1, -1);
        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);
        }
    }

    close(client);
    cout << endl;
    cout << "Alphabet Length: " << numBytes << " bytes." << endl;
    cout << "Num reads: " << numSuccessfulReads << " of " << numReads << endl;
    return 0;
}
```

Nonblocking echo server

Truth be told, nonblocking I/O is much more useful in building servers than it is in building clients. (Servers generally need to juggle more connections than clients do.) This is a nonblocking version of the “echo” server that we wrote together in the networking section of the class. The server that we wrote previously could only handle 16 connections at a time (limited by a semaphore), but this can handle as many file descriptors as we’re able to create.

```
// This describes the state of a connection, and is returned by
// EchoConnection::doReadWrite
enum conn_state_t {
    // The conn is open, but we're waiting for the client to send us stuff to
    // read
    CONNECTION_READING,
```

```
    // We're in the middle of writing to the network (echoing bytes back), but
    // the network is too congested and we're going to have to try again later.
    CONNECTION_WRITING,

    // The network connection has been closed
    CONNECTION_CLOSED,
};

class EchoConnection {
public:
    EchoConnection(int clientSocket);
    conn_state_t doReadWrite();
private:
    int clientSocket;
    char buffer[1024];
    size_t numBytesAvailable;
    size_t numBytesSent;
};

EchoConnection::EchoConnection(int clientSocket)
        : clientSocket(clientSocket), numBytesAvailable(0),
        numBytesSent(0) {
    setAsNonBlocking(clientSocket);
}

conn_state_t EchoConnection::doReadWrite() {
    // If we don't have any unsent data in `buffer` to reflect back at the
    // client, let's read some more data to echo. (We might have unsent data in
    // `buffer` if we were previously in the middle of echoing stuff, but the
    // network became too congested, so we had to try again later.)
    if (numBytesSent == numBytesAvailable) {
        ssize_t incomingCount = read(clientSocket, buffer, 1024);
        if (incomingCount == -1) {
            // There is no data to read right now, but the connection is still
            // open.
            assert(errno == EAGAIN);
            return CONNECTION_READING;
        } else if (incomingCount == 0) {
```

```
            // The client has gone away
            return CONNECTION_CLOSED;
        }
        cout << "Read " << incomingCount << " bytes from fd " << clientSocket
            << endl;
        numBytesAvailable = incomingCount;
        numBytesSent = 0;
    }

    // By this point, we have bytes to send. Let's keep trying to send stuff
    // until (1) we finish, or (2) we find out that the network is too
    // congested to send right now
    while (numBytesSent < numBytesAvailable) {
        // Ignore SIGPIPE for the duration of the write. In the event that the
        // client hung up on us here, we don't want our process to get killed
        // (which is the default behavior)
        auto old = signal(SIGPIPE, SIG_IGN);
        ssize_t outgoingCount = write(clientSocket,
                buffer + numBytesSent, numBytesAvailable - numBytesSent);
        signal(SIGPIPE, old);
        if (outgoingCount >= 0) {
            cout << "Wrote " << outgoingCount << " bytes to fd "
                << clientSocket << endl;
            numBytesSent += outgoingCount;
        } else if (errno == EPIPE) {
            // The client hung up before we could write
            return CONNECTION_CLOSED;
        } else {
            // The network is too congested right now. We need to try again
            // later
            assert(errno == EAGAIN);
            return CONNECTION_WRITING;
        }
    }
    // When we get to this point (because the `while` loop has exited), we've
    // completely finished sending any bytes in `buffer`. Time to read more
    // bytes
    return CONNECTION_READING;
```

```
}

class EchoServer {
public:
    void run();
private:
    int watchset;
    int serverSocket;
    unordered_map<int, EchoConnection> connections;
    void acceptNewConnections();
    void handleClientActivity(int clientSocket);
};

void EchoServer::run() {
    serverSocket = createServerSocket(12345);
    watchset = epoll_create(1);

    setAsNonBlocking(serverSocket);
    struct epoll_event target;
    target.events = EPOLLIN | EPOLLET;
    target.data.fd = serverSocket;
    epoll_ctl(watchset, EPOLL_CTL_ADD, serverSocket, &target);

    while (true) {
        struct epoll_event events[64];
        int numEvents = epoll_wait(watchset, events, 64, -1);
        for (int i = 0; i < numEvents; i++) {
            int eventFd = events[i].data.fd;
            if (eventFd == serverSocket) {
                acceptNewConnections();
            } else {
                handleClientActivity(eventFd);
            }
        }
    }

    close(serverSocket);
    close(watchset);
```

```
}

void EchoServer::acceptNewConnections() {
    // We configured our server socket to be edge-triggered, so once we receive
    // a notification that there is at least one client, we have to loop until
    // we process *all* waiting clients. (This is very similar to why you need
    // a while loop in a SIGCHLD handler.)
    while (true) {
        int clientSocket = accept(serverSocket, NULL, NULL);
        if (clientSocket == -1) {
            break;
        }
        connections.insert({clientSocket, EchoConnection(clientSocket)});
        struct epoll_event target;
        target.events = EPOLLIN;
        target.data.fd = clientSocket;
        epoll_ctl(watchset, EPOLL_CTL_ADD, clientSocket, &target);
        cout << "New connection accepted!" << endl;
    }
}

void EchoServer::handleClientActivity(int clientSocket) {
    EchoConnection& conn = (*connections.find(clientSocket)).second;
    // Attempt to read or write bytes on this connection
    conn_state_t state = conn.doReadWrite();
    if (state == CONNECTION_CLOSED) {
        // Remove the connection from the unordered_map, and close the file
        // descriptor. (Note that closing the file descriptor will also remove
        // this client from our epoll watch set.)
        connections.erase(clientSocket);
        close(clientSocket);
        cout << "Socket " << clientSocket << " closed" << endl;
    } else if (state == CONNECTION_WRITING) {
        // We were in the process of writing, but the network buffers got too
        // congested, so we're going to have to try again later. Add EPOLLOUT
        // so that we get notified when things clear up and we're able to send
        // bytes again
        struct epoll_event updated;
```

```
            updated.events = EPOLLIN | EPOLLOUT;
            updated.data.fd = clientSocket;
            epoll_ctl(watchset, EPOLL_CTL_MOD, clientSocket, &updated);
        } else {
            // State is CONNECTION_READING, so we're waiting for the client to send
            // us bytes to read. Remove EPOLLOUT (if it was present), so that we
            // don't receive constant notifications that we can write to the
            // network (because we aren't trying to write right now).
            struct epoll_event updated;
            updated.events = EPOLLIN;
            updated.data.fd = clientSocket;
            epoll_ctl(watchset, EPOLL_CTL_MOD, clientSocket, &updated);
        }
    }

    int main(int argc, char *argv[]) {
        EchoServer().run();
        return 0;
    }
```

Correction to lecture

The code that I originally wrote in lecture burned a lot more CPU than it was supposed to. This was because of a combination of two things: I configured each client socket to be level-triggered *and* I configured each client socket using `EPOLLOUT`.

The `EPOLLIN` flag waits for data to be available for us to read; the `EPOLLOUT` flag waits until we're able to send data. (We might be momentarily unable to send data because the OS internal buffers are too full, or the network is too congested.) Level-triggering with `EPOLLIN` means that if any data is available to read, we'll be notified, even if we already received a prior notification. Level-triggering with `EPOLLOUT` means that if it is possible to write to the network, we'll be notified, even if we already received a prior notification. That's bad news for us – the vast majority of the time, the network is *not* congested, so it's possible for us to write... so `epoll_wait` will give us a notification about every client socket pretty much all the time.

Given this, there are two possible fixes. We can use `EPOLLOUT` only when we know the network is clogged (and we need to get a notification when it becomes un-clogged), or we

can use edge triggering.

Using `EPOLLOUT` sparingly

In `acceptNewConnections`, we can remove the `EPOLLOUT` flag to receive notifications only about incoming data by default:

```
void EchoServer::acceptNewConnections() {
    while (true) {
        int clientSocket = accept(serverSocket, NULL, NULL);
        if (clientSocket == -1) {
            break;
        }
        connections.insert({clientSocket,
                EchoConnection(clientSocket)});
        struct epoll_event target;
        target.events = EPOLLIN;
        target.data.fd = clientSocket;
        epoll_ctl(watchset, EPOLL_CTL_ADD, clientSocket, &target);
        cout << "New connection accepted!" << endl;
    }
}
```

We can use `epoll_ctl` with `EPOLL_CTL_MOD` (instead of `EPOLL_CTL_ADD`) to add the `EPOLLOUT` flag once we discover that the network buffers are congested. Unfortunately, this requires some changes to the way I wrote my code, because we discover network congestion in `doReadWrite` (by receving `-1` with `errno EAGAIN` when we call `write()`), but we don't have access to the watch set file descriptor there, so calling `epoll_ctl` is tricky.

A quick-and-dirty fix simply adds a `watchset` parameter to `doReadWrite` so that we can call `epoll_ctl` from there. It's not great decomposition.

```
bool EchoConnection::doReadWrite(int watchset) {
    if (numBytesSent == numBytesAvailable) {
        ssize_t incomingCount = read(clientSocket, buffer, 1024);
        if (incomingCount == -1) {
            assert(errno == EAGAIN);
```

```
            return true;
        } else if (incomingCount == 0) {
            return false;
        }
        numBytesAvailable = incomingCount;
        numBytesSent = 0;
    }

    // Keep writing bytes until we've written everything to the network, or
    // until we find out that the network is too congested and we'll have to
    // try again in a little bit
    while (numBytesSent < numBytesAvailable) {
        auto old = signal(SIGPIPE, SIG_IGN);
        ssize_t outgoingCount = write(clientSocket,
                buffer + numBytesSent, numBytesAvailable - numBytesSent);
        signal(SIGPIPE, old);
        if (outgoingCount >= 0) {
            numBytesSent += outgoingCount;
            // We successfully wrote to the network. Remove EPOLLOUT so that we
            // don't keep receiving notifications that the network is clear
            struct epoll_event target;
            target.events = EPOLLIN;
            target.data.fd = clientSocket;
            epoll_ctl(watchset, EPOLL_CTL_MOD, clientSocket, &target);
        } else if (errno == EPIPE) {
            return false;
        } else {
            assert(errno == EAGAIN);
            // The network is congested... Let's ask for a notification when it
            // clears up
            struct epoll_event target;
            target.events = EPOLLIN | EPOLLOUT;
            target.data.fd = clientSocket;
            epoll_ctl(watchset, EPOLL_CTL_MOD, clientSocket, &target);
            return true;
        }
    }
    return true;
```

```
}
```

A better fix is to have `EchoConnection::doReadWrite` tell its caller when the network is clogged, and then have `EchoServer::run` call `epoll_ctl`. This is the solution that I have opted for, and I replaced the code in the "Nonblocking echo server" section above with this code. It's worth reading through, and I added a lot of comments to make it digestible.

Using edge triggering

Note: You don't have to understand this section thoroughly, but it would be good to read.

Level triggering is causing problems with `EPOLLOUT` because we get nonstop notifications whenever it's possible to write to the network. Using edge triggering solves this problem, because we receive a *single* notification when it *becomes* possible to write to the network, and then nothing more.

However, edge triggering can be a little trickier to handle well. In the above code (using level triggering), note that we only read up to 1024 bytes from the client at a time. If the client sends us 2048 bytes, `epoll_wait` will notify us, we'll read 1024 bytes and return from `EchoConnection::doReadWrite()`, then `epoll_wait` will notify us a second time (because it's level triggering and there are still bytes left that we need to read from the client), and finally we'll read the last 1024 bytes. If we're using level triggering, we'll only receive a single notification about those 2048 bytes, and we'll have to be sure to ingest all of it.

My quick-and-dirty solution is going to use a C++ `stringstream` to buffer a variable number of bytes. (If the client sent us 2048 bytes, we'll try to read all of it). This is easy to implement, but you should note that it's a very imperfect solution. If one client sends us 1GB of data to echo back, we're going to get stuck trying to load the whole 1GB into memory (which could cause us to run out of memory), and then we're going to try to write the whole 1GB back to the client, and in the meantime, we won't be talking to any of the other connected clients because we're so tied up with this one. (This is known as a "starvation problem.") The ideal solution does a lot of work to fairly juggle the connections we have, but it's beyond the scope of this class.

In `acceptNewConnections`, we add the `EPOLLET` flag for edge triggering:

```
void EchoServer::acceptNewConnections() {
    while (true) {
        int clientSocket = accept(serverSocket, NULL, NULL);
```

```
        if (clientSocket == -1) {
            break;
        }
        connections.insert({clientSocket, EchoConnection(clientSocket)});
        struct epoll_event target;
        target.events = EPOLLIN | EPOLLOUT | EPOLLET;
        target.data.fd = clientSocket;
        epoll_ctl(watchset, EPOLL_CTL_ADD, clientSocket, &target);
        cout << "New connection accepted!" << endl;
    }
}
```

In `EchoConnection::doReadWrite()`, we attempt to read as many bytes as possible into a `stringstream`. Then, we `write` that data back to the client in 1024-byte chunks. The full code is in `nonblocking-echo-server-et.cc`.

```
/* Attempt to read bytes from the client. Returns true if the connection is
 * still open, false if closed. */
bool EchoConnection::doRead() {
    // Clear out the buffer, in case any previous data is already there
    buffer.str("");
    buffer.clear();
    bool connectionOpen = true;

    // Keep reading data into our buffer until we run out of stuff to read
    while (true) {
        char smallBuf[1024];
        ssize_t incomingCount = read(clientSocket, smallBuf, sizeof(smallBuf));
        if (incomingCount == -1) {
            // There's nothing more to read right now
            assert(errno == EAGAIN);
            break;
        } else if (incomingCount == 0) {
            // Client has closed connection
            connectionOpen = false;
            break;
        } else {
            buffer.write(smallBuf, incomingCount);
```

```
        }
    }

    buffer.seekg(0, ios::end);
    numBytesAvailable = buffer.tellg();
    // Seek buffer to beginning, so that when we start reading from this
    // buffer, we start from the beginning
    buffer.seekg(0, ios::beg);
    numBytesSent = 0;
    cout << "Read " << numBytesAvailable << " bytes from fd " << clientSocket
        << endl;
    return connectionOpen;
}

bool EchoConnection::doReadWrite() {
    // If we don't have any unsent data in `buffer` to reflect back at the
    // client, let's read some more data to echo. (We might have unsent data in
    // `buffer` if we were previously in the middle of echoing stuff, but the
    // network became too congested, so we had to try again later.)
    if (numBytesSent == numBytesAvailable) {
        if (!doRead()) return false;
    }

    // By this point, we have bytes to send. Let's keep trying to send stuff
    // until (1) we finish, or (2) we find out that the network is too
    // congested to send right now
    while (numBytesSent < numBytesAvailable) {
        // Ignore SIGPIPE for the duration of the write. In the event that the
        // client hung up on us here, we don't want our process to get killed
        // (which is the default behavior)
        char smallBuf[1024];
        buffer.seekg(numBytesSent, ios::beg);
        buffer.read(smallBuf, sizeof(smallBuf));

        auto old = signal(SIGPIPE, SIG_IGN);
        ssize_t outgoingCount = write(clientSocket, smallBuf, buffer.gcount());
        signal(SIGPIPE, old);
        if (outgoingCount >= 0) {
```

```
            cout << "Wrote " << outgoingCount << " bytes to fd "
                << clientSocket << endl;
            numBytesSent += outgoingCount;
        } else if (errno == EPIPE) {
            // The client hung up before we could write
            return false;
        } else {
            // The network is too congested right now. We need to try again
            // later
            assert(errno == EAGAIN);
            return true;
        }
    }
    // When we get to this point (because the `while` loop has exited), we've
    // completely finished sending any bytes in `buffer`. Time to read more
    // bytes
    return true;
}
```