Lecture 19: Nonblocking I/O Example

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 19: Nonblocking I/O Example

- The implementations of the constructor and initialize are straightforward:

```cpp
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 19: Nonblocking I/O Example

- The implementation of `sendMoreData` is less straightforward:

```cpp
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.
Lecture 19: A Tale of Two Servers

- We are now going to look at two servers:
  - `expensive-server.cc`
  - `efficient-server.cc`

- The `expensive-server.cc` "Illustrates how nonblocking IO can be used to implement a single-threaded web server. This particular example wastes a huge amount of CPU time (as it while loops forever without blocking), but it does demonstrate how nonblocking IO can be used to very easily serve multiple client requests at a time without threads or multiple processes."

- In other words: we can run a non-blocking server in a single thread and have a responsive system that does not miss connections.

- The entire purpose of the server is to serve a single file (it's own source code, in .html form), and to handle multiple requests fast (we will test to see if it can handle a lot of requests quickly!)

- The full source code for `expensive-server.cc` can be found here.
Lecture 19: A Tale of Two Servers: `expensive-server.cc`

- The server has a single function, `main`, and the server setup proceeds as normal. We tell the server to not block on `accept`:

```cpp
static const unsigned short kDefaultPort = 12345;
static const string kFileToServe("expensive-server.cc.html");
int main(int argc, char **argv) {
    int serverSocket = createServerSocket(kDefaultPort);
    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;
    size_t numAcceptCalls = 0;
}
```

- We also set up a `list<OutboundFile>` to keep track of all of our outbound file connections, and a couple of variables to track the connections.
Lecture 19: A Tale of Two Servers: `expensive-server.cc`

- We set up a canonical `while (true)` loop and look for a connection via the `accept` system call. The call will return immediately, and if the return value is `-1` and `errno` has been set to `EWOULDBLOCK`, then we know that there were not any connections ready to be accepted:

```c++
while (true) {
    // right here!
    int clientSocket = accept(serverSocket, NULL, NULL);
    cout << "Num calls to accept: " << ++numAcceptCalls << "." << endl;
    if (clientSocket == -1) {
        assert(errno == EWOULDBLOCK);
    } else {
```

- If we do have a ready connection, then we handle it by initializing an `OutboundFile` that we push onto our list of connections we are handling:

```c++
OutboundFile obf;
obf.initialize(kFileToServe, clientSocket);
outboundFiles.push_back(obf);
cout << "Connection #" << ++numConnections << endl;
cout << "Queue size: " << ++numActiveConnections << endl;
}
```
Lecture 19: A Tale of Two Servers: expensive-server.cc

- Once we are done checking for and handling new connections, we need to make progress on any outstanding connections, so we loop through them and tell each to `sendMoreData()` once, erasing any connections that have finished:
- Remember: `sendMoreData` is non-blocking, so we progress through the loop quickly.

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

Lecture 19: A Tale of Two Servers: expensive-server.cc

- There is a reason this server is called **expensive-server**: it spends a lot of time busy waiting (spinning), particularly when there are no files to serve. If we run the server as is for one second, we get a continuous stream of `cout` statements like this:

  ```bash
  $ cat oneSecond.sh
  #!/bin/bash
  ./expensive-server &
  pid=$!
  sleep 1
  kill $pid
  $ ./oneSecond
  Static file server listening on port 12345.
  Num calls to accept: 1.
  Num calls to accept: 2.
  Num calls to accept: 3.
  Num calls to accept: 4.
  Num calls to accept: 5.
  Num calls to accept: 6.
  ... lots of lines removed
  Num calls to accept: 196716.
  Num calls to accept: 196717.
  Num calls to accept: 196718.
  Num calls to accept: 196719.
  Num calls to accept: 196720.
  Num calls to accept: 196721.
  $
  ```

- Although we are successful in running a responsive server in a single thread, we don't like the idea of a spinning process, because it wastes resources that the operating system could dedicate to other processes (and it makes the fans on our computer sound like jet engines).

- In truth, we do want to put our process to sleep unless:

  1. We have an incoming connection, or
  2. We have to send data on an already-open connection.

- So, we need to get the OS involved, and we will now take a look at how we can do this.
Lecture 19: The epoll Family of System Calls

- Linux has a scalable I/O event notification mechanism \(^1\) called `epoll` that can monitor a set of file descriptors to see whether there is any I/O ready for them. There are three system calls, as described below, that form the api for `epoll`.

  ```c
  int epoll_create1(int flags);
  ```

  - This function creates an epoll object and returns a file descriptor. The only valid flag is `EPOLL_CLOEXEC`, which closes the descriptor on exec as you might expect.

  ```c
  int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
  ```

  - This function configures which descriptors are watched by the object, and `op` can be `EPOLL_CTL_ADD`, `EPOLL_CTL_MOD`, or `EPOLL_CTL_DEL`. We will investigate `struct epoll_event` on the next slide.

  ```c
  int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
  ```

  - This function waits for any of the events being monitored, until there is a `timeout`. It returns up to `maxevents` at once and populates the `events` array with each event that has occurred.

\(^1\) https://en.wikipedia.org/wiki/Epoll
Lecture 19: The epoll Family of System Calls

- The `struct epoll_event` is defined as follows:

```c
struct epoll_event {
    uint32_t events;  /* Epoll events */
    epoll_data_t data; /* User data variable */
};
```

- `epoll_data_t` is a `typedef`'d `union`, defined as follows:

```c
typedef union epoll_data {
    void            *ptr;
    int              fd;
    uint32_t         u32;
    uint64_t         u64;
} epoll_data_t;
```

- A `union` is a data structure that can hold a single data type out of a set of data types, and it does so in a single memory location. The actual memory size of the union is that of the largest data type that can be stored.

- The `events` member is a bit mask, and for our purposes, we care about three values:
  - **EPOLLIN** : the file is available for reading
  - **EPOLLOUT** : the file is available for writing
  - **EPOLLET** : This sets the file descriptor to be "edge triggered", meaning that events are delivered when there is a change on the descriptor (e.g., there is data to be read).
Lecture 19: A Tale of Two Servers: efficient-server.cc

- The efficient server we will set up uses **epoll** to call functions when file descriptors are able to input or output data.
- Let's start with **main**:

```c
static const unsigned short kDefaultPort = 33333;
int main(int argc, char **argv) {
    int server = createServerSocket(kDefaultPort);
    if (server == kServerSocketFailure) {
        cerr << "Failed to start server. Port " << kDefaultPort << " is probably already in use." << endl;
        return 1;
    }
    cout << "Server listening on port " << kDefaultPort << endl;
    runServer(server);
    return 0;
}
```

- **main** simply sets up a server, and then calls the **runServer** function, which we will look at next.
The `runServer` function first converts the server socket to be nonblocking, and sets up the `epoll` watch around the socket:

```c
static void runServer(int server) {
    setAsNonBlocking(server);
    int ws = buildInitialWatchSet(server);
}
```

Let's jump to the `buildInitialWatchSet` function:

```c
static const int kMaxEvents = 64;
static int buildInitialWatchSet(int server) {
    int ws = epoll_create1(0);
    struct epoll_event info = {.events = EPOLLIN | EPOLLET, .data = {.fd = server}};
    epoll_ctl(ws, EPOLL_CTL_ADD, server, &info);
    return ws;
}
```

This function creates an epoll watch set around the supplied server socket. We register an event to show our interest in being notified when the server socket is available for read (and accept) operations via `EPOLLIN`, and we also note that the event notifications should be edge triggered (`EPOLLET`) which means that we'd only like to be notified that data is available to be read when the kernel is certain there is data.
Lecture 19: A Tale of Two Servers: `efficient-server.cc`

- Continuing where we left off with `runServer`, the function next creates an array of `struct epoll_event` objects to hold the events we may encounter.
- Then it sets up a `while (true)` loop and sets up the only blocking system call in the server, `epoll_wait()`.

```c
struct epoll_event events[kMaxEvents];
while (true) {
    int numEvents = epoll_wait(ws, events, kMaxEvents, /* timeout = */ -1);
}
```

- We never want to time out on the call, and when nothing interesting is happening with our watch set, our process is put to sleep in a similar fashion to `waits` we have seen previously in class.
- Multiple events can trigger at the same time, and we get a count (`numEvents`) of the number of events put into the `events` array.
- (continued on next slide)
Lecture 19: A Tale of Two Servers: efficient-server.cc

- When one or more of our file descriptors in the watch set trigger, we handle the events in the events array, one at a time. For our server, there could be three different events:
  - If the event was a connection request, events[i].data.fd will be the server's file descriptor, and we accept a new connection (we will look at that function shortly):

    ```c
    for (int i = 0; i < numEvents; i++) {
        if (events[i].data.fd == server) {
            acceptNewConnections(ws, server);
        }
    }
    ```

  - If the event indicates that it has incoming data (EPOLLIN), then we need to consume the data in the request:

    ```c
    } else if (events[i].events & EPOLLIN) { // we're still reading the client's request
        consumeAvailableData(ws, events[i].data.fd);
    }
    ```

  - If the event indicates that it has outgoing data (EPOLLOUT), then we publish data to that file descriptor:

    ```c
    } else { // events[i].events & EPOLLOUT
        publishResponse(events[i].data.fd);
    }
    ```
Let’s look at the `acceptNewConnections` function next.

We may have multiple connections that have come in at once, so we need to accept all of them. Therefore, we have a `while(true)` loop that continues until there are no more connections to be made:

```cpp
static void acceptNewConnections(int ws, int server) {
    while (true) {
        int clientSocket = accept4(server, NULL, NULL, SOCK_NONBLOCK);
        if (clientSocket == -1) return;
        struct epoll_event info = {
            .events = EPOLLIN | EPOLLET,
            .data = {.fd = clientSocket};
        }
        epoll_ctl(ws, EPOLL_CTL_ADD, clientSocket, &info);
    }
}
```

When we make a connection, we update our epoll watch list to include our client socket and the request to monitor it for input (again, as an edge-triggered input).

We use the `epoll_ctl` system call to register the new addition to our watch list:
Lecture 19: A Tale of Two Servers: efficient-server.cc

- We have two more functions to look at for our server: `consumeAvailableData` and `publishResponse`. The first is more complicated, but also happens first, so let's look at it now.

- The `consumeAvailableData` function attempts to read in as much data as it can from the server, until either there isn't data to be read (meaning we have to read it later), or until we get enough information in the header to respond. The second condition is met when we receive two newlines, or "\r\n\r\n":

```cpp
static const size_t kBufferSize = 1;
static const string kRequestHeaderEnding("\r\n\r\n");
static void consumeAvailableData(int ws, int client) {
    static map<int, string> requests; // tracks what's been read in thus far over each client socket
    size_t pos = string::npos;
    while (pos == string::npos) {
        char buffer[kBufferSize];
        ssize_t count = read(client, buffer, kBufferSize);
        if (count == -1 && errno == EWOULDBLOCK) return; // not done reading everything yet, so return
        if (count <= 0) { close(client); break; } // passes? then bail on connection, as it's borked
        requests[client] += string(buffer, buffer + count);
        pos = requests[client].find(kRequestHeaderEnding);
        if (pos == string::npos) continue;
    }
}
```

- Notice the `static map<>` variable inside the function. This map persists across all calls to the function, and tracks the data we have read, per client.
- If we still have data to read, but we have not yet gotten to our header ending, we keep reading data (because it is available). (continued on next slide)
Lecture 19: A Tale of Two Servers: `efficient-server.cc`

- Once we receive the header ending, we can log how many active connections we have, and then we also print out the header we've received.
- Next, we modify our epoll watch event to also trigger when data needs to be written on the client (this will happen when we publish our response).

```cpp
cout << "Num Active Connections: " << requests.size() << endl;
cout << requests[client].substr(0, pos + kRequestHeaderEnding.size()) << flush;
struct epoll_event info = {.events = EPOLLOUT | EPOLLET, .data = {.fd = client}};
epoll_ctl(ws, EPOLL_CTL_MOD, client, &info); // MOD == modify existing event
requests.erase(client);
```

- Notice that we don't break out of the while loop at this point! We continue looping until we have read all of the available data; otherwise, `epoll_wait` will not trigger again, because there is still data waiting for us (e.g., the rest of the response). The only time we exit the loop (see the previous slide) is when we have no more data to read, at which point we also close the connection.
- Also notice (previous slide) that we return when we encounter a potential block -- we don't close the connection, and we don't erase the client entry in our `requests` map. Recall that as a static variable, the map persists, as does the `requests` map entry.
- Once we exit the loop because there is no more data, we erase the client entry in our `requests` map, because it is no longer needed.
Finally, let's turn our attention to `publishResponse`.

Our response needs to be a proper HTTP response, and we supplement this with our HTML code for the website we will push to the client.

```cpp
static const string kResponseString("HTTP/1.1 200 OK\r\n\r
"<b>Thank you for your request! We're working on it! No, really!</b><br/>
"<br/>
"<img src="http://vignette3.wikia.nocookie.net/p__/images/e/e0/
"Agnes_Unicorn.png/revision/latest?cb=20160221214120&path-prefix=protagonist"/>");
```

As we saw in `consumeAvailableData`, we have a `static` map, this time populated with the file descriptor of the client we are responding to, with the values corresponding to the number of bytes we have sent. Remember, no blocking allowed!

We attempt to write all of the data in the response, but if we can't, we don't block and we return, knowing that the `responses` map will persist until the next time we call the function to push data. We erase the entry from the map and close the connection once we have sent all the data for the response (which may be after multiple calls to the function).

```cpp
static void publishResponse(int client) {
  static map<int, size_t> responses;
  responses[client]; // insert a 0 if key isn't present
  while (responses[client] < kResponseString.size()) {
    ssize_t count = write(client, kResponseString.c_str() + responses[client],
                          kResponseString.size() - responses[client]);
    if (count == -1 && errno == EAGAIN) return;
    if (count == -1) break;
    assert(count > 0);
    responses[client] += count;
  }
  responses.erase(client);
  close(client);
}
```
Lecture 19: A Tale of Two Servers: Conclusion

- As we have seen over the past few lectures, there are many ways to build a server.
- The best way depends on your own goals, your system setup (OS, computer type, network, etc.), but there are two key things that your servers should prioritize:
  - Accept and respond to as many connections as they can
  - Respond quickly to client requests
- Both of these issues are realizable using threads (or processes), or by using non-blocking I/O and OS-savvy waiting. You should probably not write a server that busy-waits (and you should almost always avoid busy-waiting, in general)
- Whichever way you decide to write your servers, you do need to understand some lower-level ideas, such as those we covered in class this quarter.
- Networking comes with a number of challenges, particularly when you prioritize as above. There are more details than we've seen so far (see below). For some of those details (and even lower-level concerns), consider taking CS 144.

https://xkcd.com/869/