Lecture 20: Slow System Calls and Non-Blocking I/O

Principles of Computer Systems

Spring 2019

Stanford University

Computer Science Department

Lecturer: Chris Gregg

PDF of this presentation

Lecture 20: Slow System Calls and Non-Blocking 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 20: Slow System Calls and Non-Blocking 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 20: Slow System Calls and Non-Blocking 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 20: Slow System Calls and Non-Blocking 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 20: Slow System Calls and Non-Blocking 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 20: Slow System Calls and Non-Blocking 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 20: Slow System Calls and Non-Blocking 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 20: Slow System Calls and Non-Blocking 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 20: Slow System Calls and Non-Blocking 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 << " unsuccessfu
    return 0;
}
```

Lecture 20: Slow System Calls and Non-Blocking 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
```

Lecture 20: Slow System Calls and Non-Blocking 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 20: Slow System Calls and Non-Blocking 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 20: Slow System Calls and Non-Blocking 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;
```

Lecture 20: Slow System Calls and Non-Blocking 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); // sanitycheck to confirm -1 doesn't represent a true failure
    } else {
        OutboundFile obf;
        obf.initialize(kFileToServe, client);
        outboundFiles.push_back(obf);
    }
    // part 2: presented on next slide
```

Lecture 20: Slow System Calls and Non-Blocking 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 20: Slow System Calls and Non-Blocking 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 20: Slow System Calls and Non-Blocking 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 20: Slow System Calls and Non-Blocking 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 20: Slow System Calls and Non-Blocking 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 20: Slow System Calls and Non-Blocking 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.

Lecture 20: 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 20: 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`**:

```
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 20: 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 **`EOULDBLOCK`**, then we know that there were not any connections ready to be accepted:

```
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:

```
    OutboundFile obf;
    obf.initialize(kFileToServe, clientSocket);
    outboundFiles.push_back(obf);
    cout << "Connection #" << ++numConnections << endl;
    cout << "Queue size: " << ++numActiveConnections << endl;
  }
```

Lecture 20: 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.

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

Lecture 20: 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:

```
$ 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 20: 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`**.
- `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.
- `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.
- `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 20: The epoll Family of System Calls

- The **`struct epoll_event`** is defined as follows:

```
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:

```
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 20: 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`**:

```
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.

Lecture 20: A Tale of Two Servers: efficient-server.cc

- The **`runServer`** function first converts the server socket to be nonblocking, and sets up the **`epoll`** watch around the socket:

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

- Let's jump to the **`buildInitialWatchSet`** function:

```
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 20: 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()`**.

```
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 **`wait`**s 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 20: 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):

```
    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:

```
      } 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:

```
      } else { // events[i].events & EPOLLOUT
        publishResponse(events[i].data.fd);
      }
    }
  }
}
```

Lecture 20: A Tale of Two Servers: efficient-server.cc

- 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:

```
static void acceptNewConnections(int ws, int server) {
  while (true) {
    int clientSocket = accept4(server, NULL, NULL, SOCK_NONBLOCK);
    if (clientSocket == -1) return;
```

- 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:

```
    struct epoll_event info = {.events = EPOLLIN | EPOLLET, .data = {.fd = clientSocket}};
    epoll_ctl(ws, EPOLL_CTL_ADD, clientSocket, &info);
  }
}
```

Lecture 20: 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"**:

```
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 20: 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).

```
    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.

Lecture 20: A Tale of Two Servers: efficient-server.cc

- 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.

```
static const string kResponseString("HTTP/1.1 200 OK\r\n\r\n"
        "<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\"/>");
```

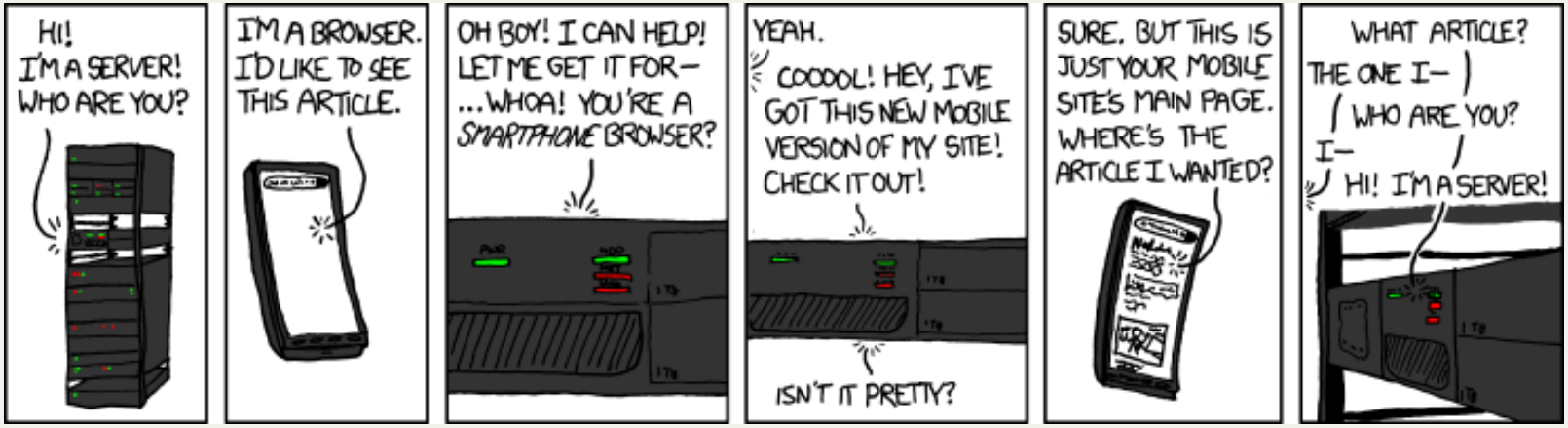
- As we saw in **`consumeAvailableData`**, we have a **`static`** map, this time populated with the file descriptor of the client we are responding to, with the values corresponding to the number of bytes we have sent. Remember, no blocking allowed!
- We attempt to write all of the data in the response, 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).

```
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 20: A Tale of Two Servers: efficient-server.cc

- 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/