Lecture 19: Events, Threads, and Asynchronous I/O

Principles of Computer Systems

Autumn 2019

Stanford University

Computer Science Department

Instructors: Chris Gregg

Philip Levis

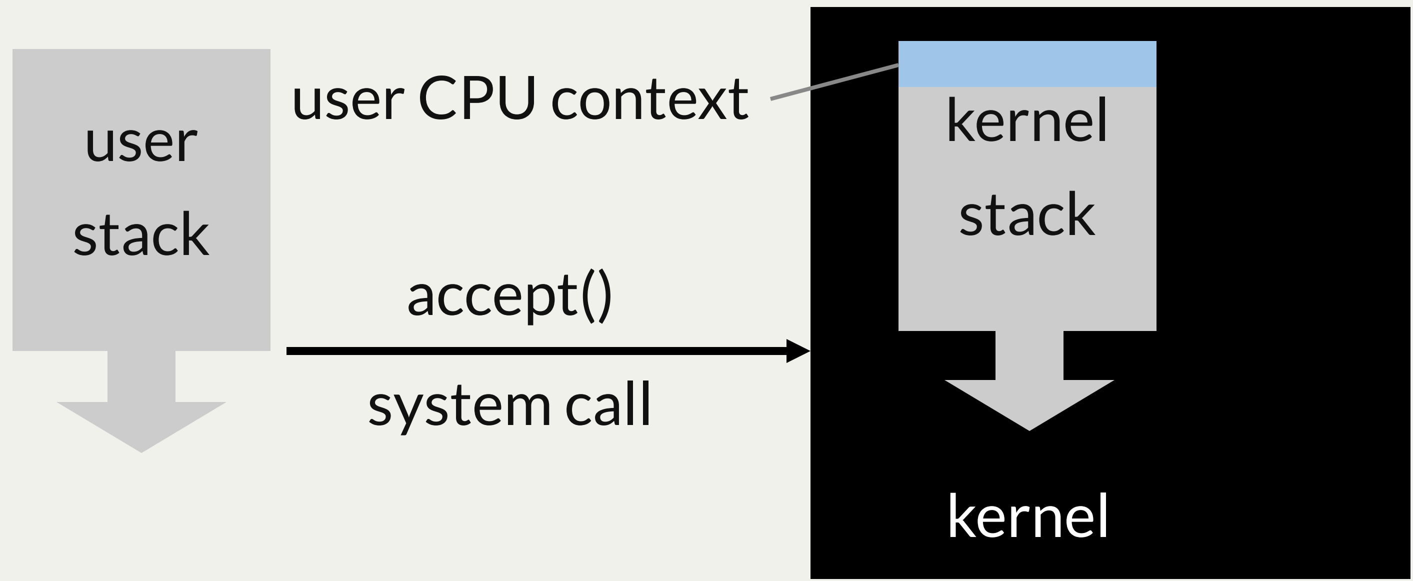
PDF of this presentation

Blocking

- We've seen a lot of system calls that *block*, yielding the CPU and causing a thread to sleep until the system call completes
  - Reading a file from a disk: **`read(2)`**
  - Writing data to a network socket: **`write(2)`**
  - Waiting for a process to complete: **`waitpid(2)`**
  - Locking a mutex: **`pthread_mutex_lock(2)`**
- Some system calls can (if nothing's broken) block for a little while:
  - Reading a file from a disk: **`read(2)`**
  - Opening a network connection: **`connect(2)`**
- Others (if nothing's broken) can block **forever**:
  - Reading from a network socket: **`read(2)`**
  - Accepting a network connection: **`accept(2)`**
  - Waiting for a process to complete: **`waitpid(2)`**
  - Locking a mutex: **`pthread_mutex_lock(2)`**

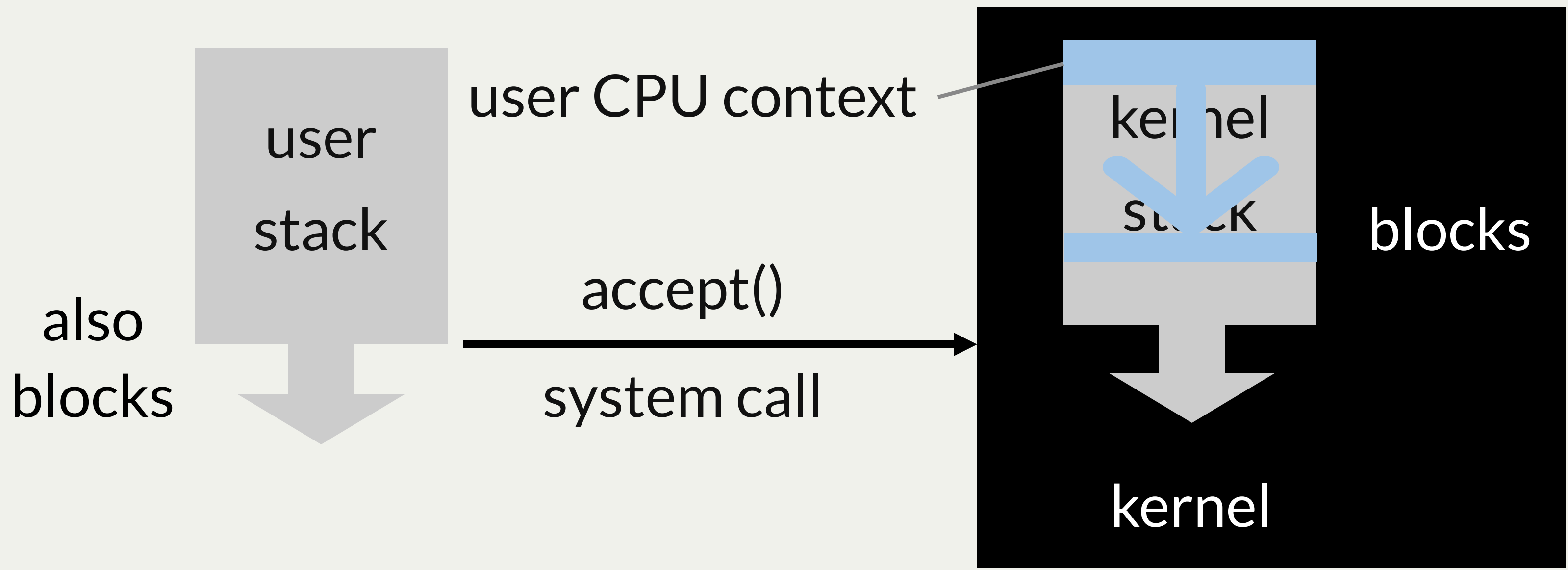
What Blocking Does

- Recall from when we discussed signals, when a user thread calls a system call, it traps into the operating system and starts using its kernel stack
- When the operations on the kernel stack complete, the OS restores the user CPU context and resumes the user thread
- If the operations on the kernel stack block, then the user stack is blocked as well: it will not resume until the kernel stack completely unwinds



What Blocking Does

- Recall from when we discussed signals, when a user thread calls a system call, it traps into the operating system and starts using its kernel stack
- When the operations on the kernel stack complete, the OS restores the user CPU context and resumes the user thread
- If the operations on the kernel stack block, then the user stack is blocked as well: it will not resume until the kernel stack completely unwinds



Blocking Limits Concurrency and Performance

- While a thread is blocked, it cannot do any useful work
- When all threads are blocked, the service can't do anything at all
  - Number of outstanding operations <= number of threads
- This can be a performance problem
- Example: your web proxy cache (some made up, approximate numbers)
  - Average time to download a remote page: 50ms
  - Time to serve a page from local cache: 1ms
  - Time spent on CPU serving a page from local cache: 100 microseconds
  - Even when always hitting cache, each thread is only 10% active (other 90% is spent writing to network socket, 100us/1ms)
  - When cache is cold, threads are 0.2% active (other 99.8% is spent in network I/O)
  - Maximum cache throughput on an 8 core machine needs up to 4,000 threads!
    - 500 threads/core * 8 cores

Why 4,000 Threads Can Be a Problem

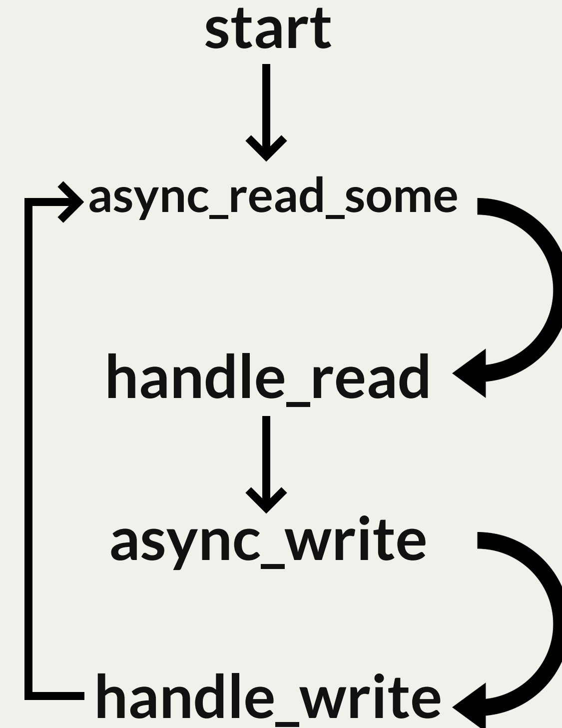
- By default, Linux allows up to 8MB of memory for a thread stack
  - 4,000 threads * 8MB = 32GB!
  - This memory isn't physically allocated, it's only allocated as it's used
    - Don't allocate large objects on the stack!
  - The kernel won't let you allocate more threads than can be stored in physical memory
- Any algorithm that iterates across a list of threads will be slow
  - Linux uses very fine-grained queues, e.g. per-socket, so it can scale
    - Imagine if, instead, all threads waiting on I/O were on a single queue, and the kernel had to iterate across it to figure out which one to resume
  - Be careful with `notify_all()`
- To allow parallelism across 4,000 threads, you need fine-grained locks (more than the 997 mutexes you implemented)
- 4,000 stacks can lead to a lot of cache misses
  - Each thread will put part of its stack into the cache, evicting cache lines for other thread stacks
- You can design systems that use 4,000 threads, but doing so is very hard and you have to be very careful in your implementation

Events: High Concurrency with One Thread

- An alternative concurrency model is to use *events* rather than threads
- Event-driven systems have no blocking calls
  - A call starts an operation (e.g., start a read)
  - The system invokes an event callback when the operation completes
- Archetypical examples of event-driven systems
  - GUIs: every button/menu item invokes a callback function on your application
  - Web pages/Javascript: events from user interactions (e.g., mouse movement, clicks, etc.) invoke callbacks
- Because thread never blocks, it can fully use a core by starting operations and handling completion callbacks
- This is more than non-blocking operations, which just says it won't block: non-blocking by itself has to use spin loops
  - Example of non-blocking: **`waitpid`** with **`WNOHANG`**
    - Code must enter a spin loop (wastes CPU cycles) to wait for work to do
  - Example of event-driven execution: signals with **`sigsuspend`**
    - Code can suspend until there is work to do (an event)

Example Event-Driven Boost Code: Echo Server

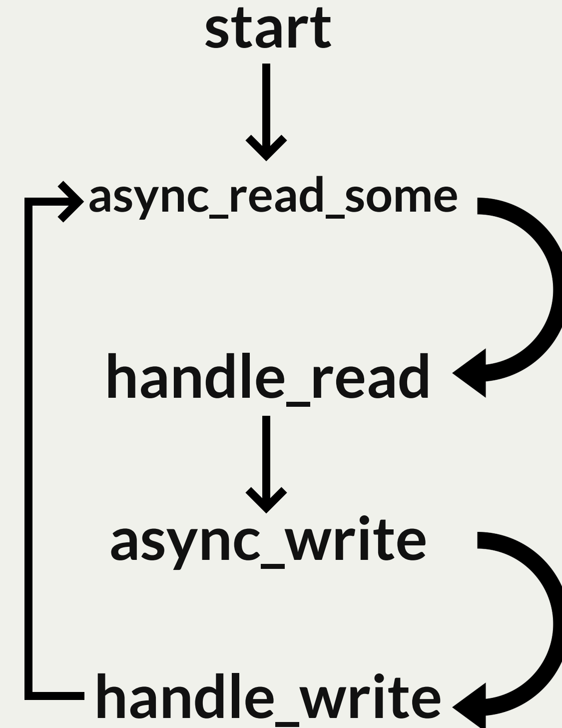
```
1  class session {
2  public:
3      session(boost::asio::io_service& io_service) :
4            socket_(io_service)   {} // construct a TCP-socket from io_service
5      tcp::socket& socket(){return socket_;}
6
7      void start(){
8          // initiate asynchronous read; handle_read() is callback-function
9          socket_.async_read_some(boost::asio::buffer(data_,max_length),
10             boost::bind(&session::handle_read,this,
11                 boost::asio::placeholders::error,
12                 boost::asio::placeholders::bytes_transferred));
13     }
14
15 private:
16     void handle_read(const boost::system::error_code& error,
17                      size_t bytes_transferred){
18         if (!error)
19             // initiate asynchronous write; handle_write() is callback-function
20             boost::asio::async_write(socket_,
21                 boost::asio::buffer(data_,bytes_transferred),
22                 boost::bind(&session::handle_write,this,
23                     boost::asio::placeholders::error));
24         else
25             delete this;
26     }
27
28     void handle_write(const boost::system::error_code& error){
29         if (!error)
30             // initiate asynchronous read; handle_read() is callback-function
31             socket_.async_read_some(boost::asio::buffer(data_,max_length),
32                 boost::bind(&session::handle_read,this,
33                     boost::asio::placeholders::error,
34                     boost::asio::placeholders::bytes_transferred));
35         else
36             delete this;
37     }
38
39     boost::asio::ip::tcp::socket socket_;
40     enum { max_length=1024 };
41     char data_[max_length];
42 };
```



Example Event-Driven Boost Code: Echo Server

```
 1 class session {
 2 public:
 3     session(boost::asio::io_service& io_service) :
 4            socket_(io_service)   {} // construct a TCP-s
 5     tcp::socket& socket(){return socket_;}
 6
 7     void start(){
 8         // initiate asynchronous read; handle_read() is callback-function
 9         socket_.async_read_some(boost::asio::buffer(data_,max_length),
10             boost::bind(&session::handle_read,this,
11                 boost::asio::placeholders::error,
12                 boost::asio::placeholders::bytes_transferred));
13     }
14
15 private:
16     void handle_read(const boost::system::error_code& error,
17                      size_t bytes_transferred){
18         if (!error)
19             // initiate asynchronous write; handle_write() is callback-function
20             boost::asio::async_write(socket_,
21                 boost::asio::buffer(data_,bytes_transferred),
22                 boost::bind(&session::handle_write,this,
23                     boost::asio::placeholders::error));
24         else
25             delete this;
26     }
27
28     void handle_write(const boost::system::error_code& error){
29         if (!error)
30             // initiate asynchronous read; handle_read() is callback-function
31             socket_.async_read_some(boost::asio::buffer(data_,max_length),
32                 boost::bind(&session::handle_read,this,
33                     boost::asio::placeholders::error,
34                     boost::asio::placeholders::bytes_transferred));
35         else
36             delete this;
37     }
38
39     boost::asio::ip::tcp::socket socket_;
40     enum { max_length=1024 };
41     char data_[max_length];
42 };
```

code returns here, can do other work



The Debate

- Huge debate in 1995-2004, with birth of Internet services: threads or events?

1995

Why Threads Are A Bad Idea (for most purposes)

John Ousterhout

Sun Microsystems Laboratories

The Debate

- Huge debate in 1995-2004, with birth of Internet services: threads or events?

1995

Why Threads Are A Bad Idea (for most purposes)

2002

Event-driven Programming for Robust Software

Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazières, Robert Morris
MIT Laboratory for Computer Science
{fdabek,kolya,kaashoek,rtm}@lcs.mit.edu, dm@scs.cs.nyu.edu

The Debate

- Huge debate in 1995-2004, with birth of Internet services: threads or events?

1995

Why Threads Are A Bad Idea
(for most purposes)

2002 **Event-driven Programming for Robust Software**

2003

Why Events Are A Bad Idea
(for high-concurrency servers)

Rob von Behren, Jeremy Condit and Eric Brewer
Computer Science Division, University of California at Berkeley
{jrvb, jcondit, brewer}@cs.berkeley.edu
http://capriccio.cs.berkeley.edu/

Tradeoffs

- Problems with threads
  - Synchronization bugs (data races, deadlock, etc.)
  - Limits concurrency to number of threads
  - Stacks are expensive
  - Possible interleaving of threads is hard to reason about
- Problems with events
  - State across events can't be stored on stack ("stack ripping")
  - Possible interleaving of events is hard to reason about
  - Sequential execution is lost, has to be manually traced across code

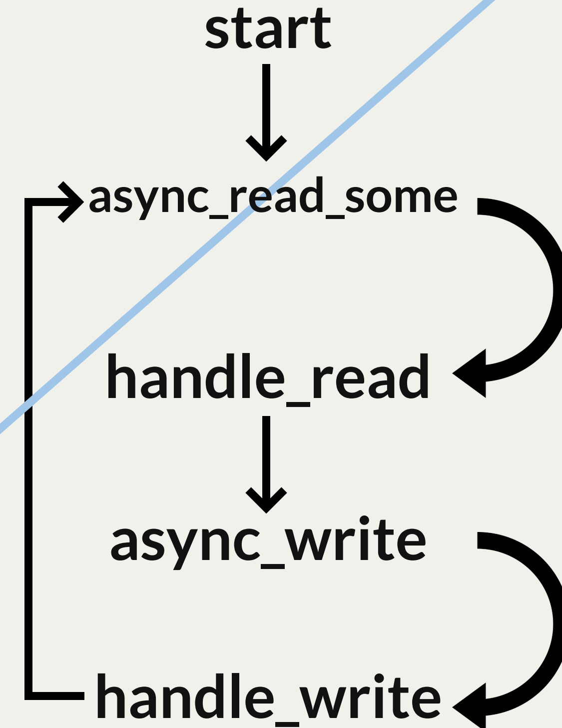
Tradeoffs

- Problems with threads
  - Synchronization bugs (data races, deadlock, etc.)
  - Limits concurrency to number of threads
  - Stacks are expensive
  - Possible interleaving of threads is hard to reason about
- Problems with events
  - **State across events can't be stored on stack ("stack ripping")**
  - Possible interleaving of events is hard to reason about
  - Sequential execution is lost, has to be manually traced across code

Stack Ripping

```
1  class session {
2  public:
3      session(boost::asio::io_service& io_service) :
4            socket_(io_service)   {} // construct a TCP-soc
5      tcp::socket& socket(){return socket_;}
6
7      void start(){
8          // initiate asynchronous read; handle_read() is callback-function
9          socket_.async_read_some(boost::asio::buffer(data_,max_length),
10             boost::bind(&session::handle_read,this,
11                 boost::asio::placeholders::error,
12                 boost::asio::placeholders::bytes_transferred));
13     }
14
15 private:
16     void handle_read(const boost::system::error_code& error,
17                      size_t bytes_transferred){
18         if (!error)
19             // initiate asynchronous write; handle_write() is callback-function
20             boost::asio::async_write(socket_,
21                 boost::asio::buffer(data_,bytes_transferred),
22                 boost::bind(&session::handle_write,this,
23                     boost::asio::placeholders::error));
24         else
25             delete this;
26     }
27
28     void handle_write(const boost::system::error_code& error){
29         if (!error)
30             // initiate asynchronous read; handle_read() is callback-function
31             socket_.async_read_some(boost::asio::buffer(data_,max_length),
32                 boost::bind(&session::handle_read,this,
33                     boost::asio::placeholders::error,
34                     boost::asio::placeholders::bytes_transferred));
35         else
36             delete this;
37     }
38
39     boost::asio::ip::tcp::socket socket_;
40     enum { max_length=1024 };
41     char data_[max_length];
42 };
```

Any state needed across asynchronou
calls must be stored in object (e.g., data



Tradeoffs

- Problems with threads
  - Synchronization bugs (data races, deadlock, etc.)
  - **Limits concurrency to number of threads**
  - **Stacks are expensive**
  - Possible interleaving of threads is hard to reason about
- Problems with events
  - State across events can't be stored on stack ("stack ripping")
  - Possible interleaving of events is hard to reason about
  - Sequential execution is lost, has to be manually traced across code

Of these seven tradeoff points, 5 of them (greyed out) relate to programming challenges and programmer reasoning: they can be ameliorated through software design, testing, and methodology. They're questions of taste.

2 of them (bold) are unavoidable performance issues.

Asynchronous I/O

- Need concurrency at a finger granularity than threads
- Requires asynchronous I/O: system calls do not block
  - Other system calls allow code to check when outstanding operations have completed
- Can be used as the underlying API for event driven execution
- Can be used as it is for synchronously checking for completion

Asynchronous I/O: 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:

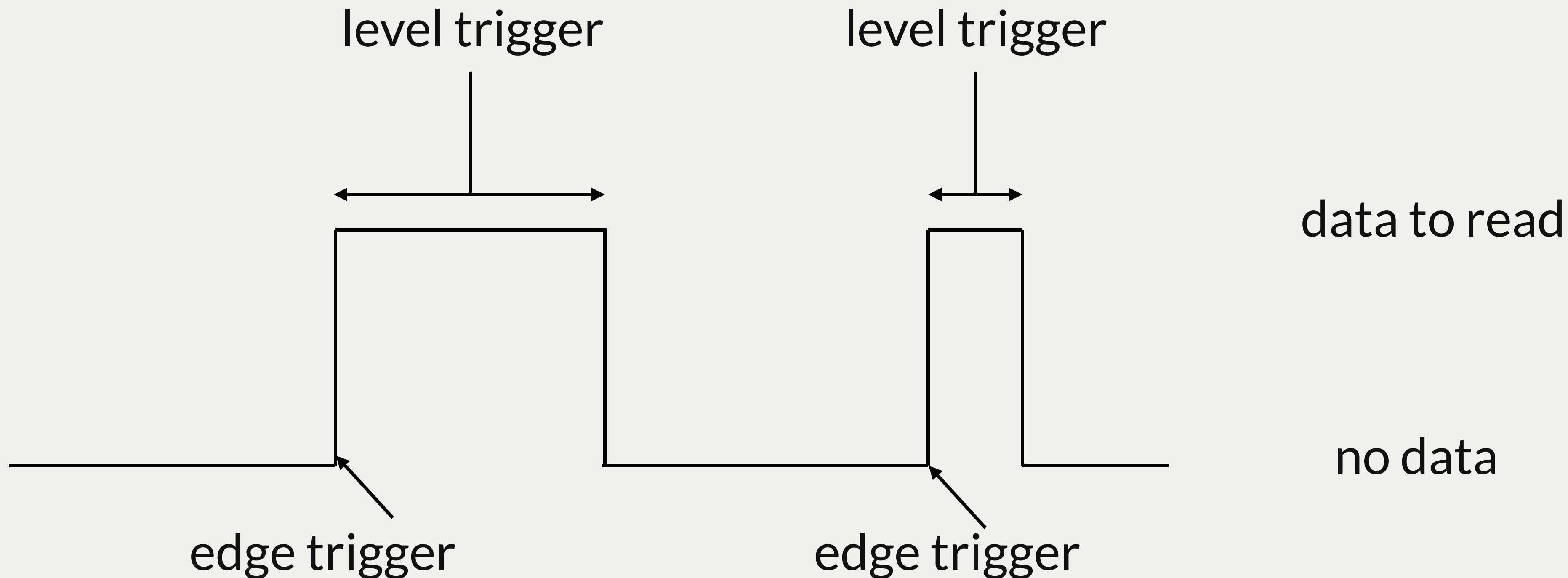
```
typedef union epoll_data {
    void        *ptr;
    int          fd;
    uint32_t     u32;
    uint64_t     u64;
} epoll_data_t;
```

- 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).

Level Triggered (LT) vs. Edge Triggered (ET)

- A watch set is by default *level-triggered*: epoll notifies while the condition is true
- You can also set it to be *edge-triggered*: epoll notifies when the condition becomes true



epoll Example: An Asynchronous Web Server

- Small example of using **`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 socket, and then calls the **`runServer`** function, which we will look at next.

Server Architecture Overview

- Use asynchronous I/O to handle many connections and requests from a single thread of control
- A core loop calls **`epoll_wait()`** to receive events from kernel
- An event on the server socket means there's a new connection: accept it
- To read requests, use a static map of strings to store requests, keyed on file descriptor
  - As parts of requests are read, append them to the string
  - When a request is complete, erase it and wait for write events
- To write responses, use a static map of integers to store write position, keyed on file descriptor
  - As parts of response is written, update write position
  - When a response is complete, erase it and close the socket

Setting Up Asynchronous I/O

- 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);  // fcntl(descriptor, F_SETFL, fcntl(descriptor, F_GETFL) | O_NONBLOCK)
  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 becomes available to be read: we'll need to be sure to read all of it.

Waiting for Asynchronous I/O Events

- 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.
- Note how this is different than event-driven execution with callbacks: your code is responsible for the core loop that waits on events, and determining how to respond to them. Event-driven libraries generally build on top of **`epoll_wait()`**.
- (continued on next slide)

Core Event Handling Loop

- 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);
      }
    }
  }
}
```

Handling a Connection Request: Add Socket to Watch List

- 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);
  }
}
```

Handling Data Reception: Read Until Request is Complete

- 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 so it tracks partially read data for each client (stack ripping).
- 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)

Finishing Reading a Request: Write a Response

- 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.

Sending a Response

- 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);
}
```

Server Architecture Overview

- Use asynchronous I/O to handle many connections and requests from a single thread of control
- A core loop calls `epoll_wait()` to receive events from kernel
- An event on the server socket means there's a new connection: accept it
- To read requests, use a static map of strings to store requests, keyed on file descriptor
  - As parts of requests are read, append them to the string
  - When a request is complete, erase it and wait for write events
- To write responses, use a static map of integers to store write position, keyed on file descriptor
  - As parts of response is written, update write position
  - When a response is complete, erase it and close the socket
- This very simple server allows for such a simple implementation (e.g., static maps in the read and write event handlers): real (larger, more complex) systems are usually tricker.

Threads vs. Events, Revisited

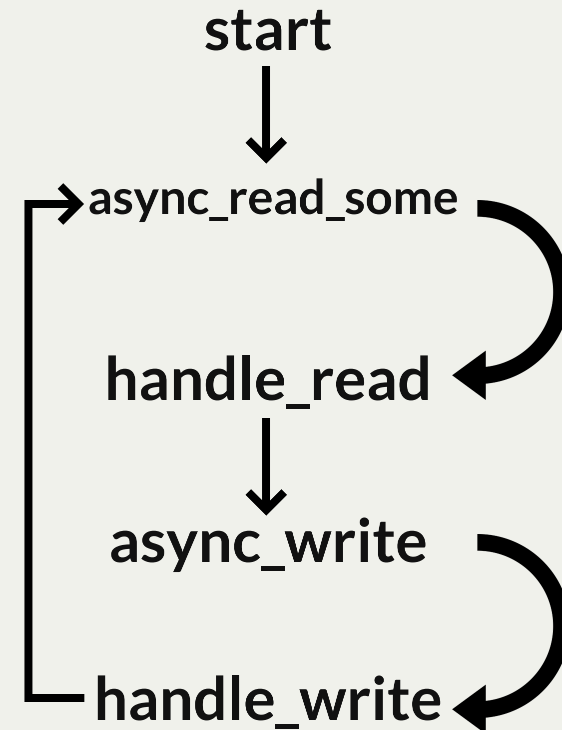
- Using `epoll_wait()` allows a single thread to handle hundreds or thousands of requests
  - Limited to what a single core can do
- Forking multiple processes to run `epoll_wait()` loops can use all the cores in a system: this is what many systems do today
- Recall the problems with events:
  - State across events can't be stored on stack ("stack ripping")
  - Sequential execution is lost, has to be manually traced across code
  - Possible interleaving of events is hard to reason about
- Let's look at how each problem is dealt with today.

State across events can't be stored on stack ("stack ripping")

- Usually, there is a tradeoff between small, short term state (stored on the stack), and larger or longer-term state, stored on the heap
  - You don't want to read a 4MB web page onto the stack
  - A session may persist for a long time (30+ seconds)
    - Having it in a global table allows a server to track and manage current sessions
    - Having it on the stack makes it private and more difficult to monitor/manage
  - You don't want to store temporary variables (e.g., current position) in the heap: you either have to allocate the union of all such possible variables (a lot of dead space), or are malloc'ing small values (e.g., a size_t)
- Standard approach today: *coroutines*, or small, dynamically allocated, per-session stacks
  - Coroutines allow a session to store local variables across asynchronous I/O calls
  - Coroutines make asynchronous I/O calls look blocking: they block in user space, rather than the kernel
  - Store larger, long-lived state on heap
  - Not preemptible: a coroutine in a long computation can hog a thread

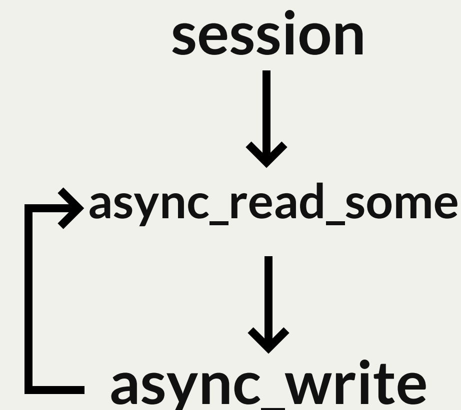
Example Event-Driven Boost Code: Echo Server

```
 1 class session {
 2 public:
 3     session(boost::asio::io_service& io_service) :
 4           socket_(io_service)   {} // construct a TCP-socket from io_service
 5     tcp::socket& socket(){return socket_;}
 6
 7     void start(){
 8         // initiate asynchronous read; handle_read() is callback-function
 9         socket_.async_read_some(boost::asio::buffer(data_,max_length),
10             boost::bind(&session::handle_read,this,
11                 boost::asio::placeholders::error,
12                 boost::asio::placeholders::bytes_transferred));
13     }
14
15 private:
16     void handle_read(const boost::system::error_code& error,
17                      size_t bytes_transferred){
18         if (!error)
19             // initiate asynchronous write; handle_write() is callback-function
20             boost::asio::async_write(socket_,
21                 boost::asio::buffer(data_,bytes_transferred),
22                 boost::bind(&session::handle_write,this,
23                     boost::asio::placeholders::error));
24         else
25             delete this;
26     }
27
28     void handle_write(const boost::system::error_code& error){
29         if (!error)
30             // initiate asynchronous read; handle_read() is callback-function
31             socket_.async_read_some(boost::asio::buffer(data_,max_length),
32                 boost::bind(&session::handle_read,this,
33                     boost::asio::placeholders::error,
34                     boost::asio::placeholders::bytes_transferred));
35         else
36             delete this;
37     }
38
39     boost::asio::ip::tcp::socket socket_;
40     enum { max_length=1024 };
41     char data_[max_length];
42 };
```



Example Coroutine Boost Code: Echo Server

```
 1 boost::asio::spawn(my_strand, do_echo);
 2
 3 void do_echo(boost::asio::yield_context yield) {
 4   try {
 5     char data[128];
 6     for (;;) {
 7       std::size_t length =
 8         my_socket.async_read_some(boost::asio::buffer(data), yield);
 9       if (ec == boost::asio::error::eof) {
10         break; //connection closed cleanly by peer
11       else if (ec) {
12         throw boost::system::system_error(ec); //some other error
13       }
14
15       boost::asio::async_write(my_socket, boost::asio::buffer(data, length), yield);
16       f (ec == boost::asio::error::eof) {
17         break; //connection closed cleanly by peer
18       else if (ec) {
19         throw boost::system::system_error(ec); //some other error
20       }
21     }
22   }
23   catch (std::exception& e) {
24     std::cerr<<"Exception: "<<e.what()<<"\n";
25   }
26 }
```



- Coroutines also restore sequential code: a linear code sequence appears sequential

Origin of C++ Coroutines (end of Threads vs. Events, 2003)

Capriccio: Scalable Threads for Internet Services

Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer
Computer Science Division
University of California, Berkeley
{jrvb,jcondit,zf,necula,brewer}@cs.berkeley.edu

ABSTRACT

This paper presents Capriccio, a scalable thread package for use with high-concurrency servers. While recent work has advocated event-based systems, we believe that thread-based systems can provide a simpler programming model that achieves equivalent or superior performance.

By implementing Capriccio as a user-level thread package, we have decoupled the thread package implementation from the underlying operating system. As a result, we can take advantage of cooperative threading, new asynchronous I/O mechanisms, and compiler support. Using this approach, we are able to provide three key features: (1) scalability to 100,000 threads, (2) efficient stack management, and (3) resource-aware scheduling.

We introduce *linked stack management*, which minimizes the amount of wasted stack space by providing safe, small, and non-contiguous stacks that can grow or shrink at run time. A compiler analysis makes our stack implementation

1. INTRODUCTION

Today's Internet services have ever-increasing scalability demands. Modern servers must be capable of handling tens or hundreds of thousands of simultaneous connections without significant performance degradation. Current commodity hardware is capable of meeting these demands, but software has lagged behind. In particular, there is a pressing need for a programming model that allows programmers to design efficient and robust servers with ease.

Thread packages provide a natural abstraction for high-concurrency programming, but in recent years, they have been supplanted by event-based systems such as SEDA [41]. These event-based systems handle requests using a pipeline of stages. Each request is represented by an event, and each stage is implemented as an event handler. These systems allow precise control over batch processing, state management, and admission control; in addition, they provide benefits such as atomicity within each event handler.

Basic Capriccio Idea

- Write sequential code with synchronous I/O
- I/O blocks in user space, not the kernel: an underlying asynchronous I/O loop resumes code when its I/O operation completes
- Rather than allocate the stack as a contiguous region of memory, allocate it as a linked list of heap-allocated structures
  - A "thread" uses only the stack that it needs
  - Compiler support sizes these allocations
  - A lot of the systems magic is about making this stack efficient
    - Don't allocate on each function call: allocate small chunks and allocate more when current calls need more than is left

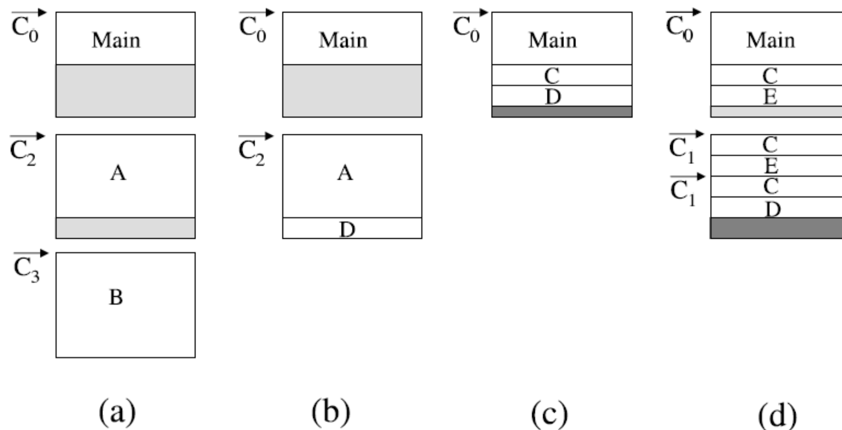


Figure 6: Examples of dynamic allocation and deallocation of stack chunks.

- A thread at 4 points in execution
- C0-C3 are "checkpoints", little bits of added code that check if a new block of stack needs to be allocated
- A, B, C, D, E are function calls that use up stack

Possible Interleaving of Events is Hard to Reason About

- Coroutines allow sequential code to appear sequential while storing temporary variables within a heap-allocated coroutine
  - Sequential code snippets look like threaded code
  - Many sequential code snippets can be multiplexed onto a single thread of control
  - Great use case: web requests, each of which is a stand-alone request/response
- But not all code is sequential
  - GUIs can have complex interleaving of user events
  - Network protocols can have complex interleaving of messages
- There is no silver bullet
  - Lots of modeling approaches: finite state machines
  - Lots of error checking approaches: code analysis, software verification
  - This problem is analogous to reasoning about thread interleaving: it is a fundamental challenge of concurrent software

Other Modern Techniques: Promises in JavaScript (for Node.js)

- Languages like Javascript allow you to easily write callbacks inline (like C++ lambdas): this can become really messy with many levels of callbacks: sequential code becomes nested

```
1  dboper.insertDocument(db, { name: "Test", description: "Test"},
2      "test", (result) => {
3          console.log("Insert Document:\n", result.ops);
4
5          dboper.findDocuments(db, "test", (docs) => {
6              console.log("Found Documents:\n", docs);
7
8              dboper.updateDocument(db, { name: "Test" },
9                  { description: "Updated Test" }, "test",
10                 (result) => {
11                     console.log("Updated Document:\n", result.result);
12
13                     dboper.findDocuments(db, "test", (docs) => {
14                         console.log("Found Updated Documents:\n", docs);
15
16                         db.dropCollection("test", (result) => {
17                             console.log("Dropped Collection: ", result);
18
19                             client.close();
20                         });
21                     });
22                 });
23         });
24     });
```

Other Modern Techniques: Promises in JavaScript (for Node.js)

- A JavaScript Promise is the result of a future computation: the **`then()`** method allows you to specify what to do when it completes.

```
1  database.insertDocument(db, { name: "Test",
2          description: "Chill Out! Its just a test program!"},
3          "test")
4          .then((result) => {
5              return database.findDocuments(db, "test");
6          })
7          .then((documents) => {
8              console.log("Found Documents:\n", documents);
9              return database.updateDocument(db, { name: "Test" },
10                     { description: "Updated Test" }, "test");
11         })
12         .then((result) => {
13             console.log("Updated Documents Found:\n", result.result);
14             return database.findDocuments(db, "test");
15         })
16         .then((docs) => {
17             console.log("The Updated Documents are:\n", docs);
18             return db.dropCollection("test");
19         })
20         .then((result) => {
21             return client.close();
22         })
23         .catch((err) => alert(err));
24 })
25 .catch((err) => alert(err));
```

Other Modern Techniques: Web Workers in JavaScript (for Browser)

- Browser JavaScript engines have a single thread of control
- A long computation causes the entire page to hang
- Solution: Web Workers for background processing (HTML5)

demo_workers.js

```
1 var i = 0;
2
3 function timedCount() {
4   i = i + 1;
5   postMessage(i);
6   setTimeout("timedCount()",500);
7 }
8
9 timedCount();
```

web page

```
1 <!DOCTYPE html>
2 <html>
3 <body>
4
5 <p>Count numbers: <output id="result"></output></p>
6 <button onclick="startWorker()">Start Worker</button>
7 <button onclick="stopWorker()">Stop Worker</button>
8
9 <script>
10 var w;
11
12 function startWorker() {
13   if (typeof(Worker) !== "undefined") {
14     if (typeof(w) == "undefined") {
15       w = new Worker("demo_workers.js");
16     }
17     w.onmessage = function(event) {
18       document.getElementById("result").innerHTML = event.data;
19     };
20   } else {
21     document.getElementById("result").innerHTML = "Sorry! No Web Worker
22   }
23 }
24
25 function stopWorker() {
26   w.terminate();
27   w = undefined;
28 }
29 </script>
30
31 </body>
32 </html>
```

starts a background worker and registers an event handler for its messages

Other Modern Techniques: Futures in Rust

- A Rust Future allows code to block on the generation of a result, not the start of the operation.
- For example, you can issue two requests in parallel, then wait for their completion.
  - Both invocations of **`new_example_future()`** will start executing
  - Execution will block on **`future1.join(future2)`**: we can add arbitrary code at line 11

```
1  extern crate futures;
2  extern crate future_by_example;
3
4  fn main() {
5      use futures::Future;
6      use futures::future::ok;
7      use future_by_example::new_example_future;
8
9      let future1 = new_example_future();
10     let future2 = new_example_future();
11     // Can do more computation here
12     let joined = future1.join(future2);
13     let (value1, value2) = joined.wait().unwrap();
14     assert_eq!(value1, value2);
15 }
```

Asynchronous I/O and Event-Driven Code

- Blocking I/O with threads is simple to write but doesn't scale well
  - Number of outstanding operations <= number of threads
- Asynchronous I/O allows a thread to have many I/O operations in parallel
  - Asynchronous because code handles completion at some later time, such that it is not synchronized with the start of the request
- Asynchronous I/O leads to event-driven programming
- Event-driven programming complicates sequential code
  - A linear series of I/O calls is spread across multiple functions, which a programmer must manually string together in their head
- Coroutines are the standard C++ middle ground: looks like threads, but many coroutines can execute on a single thread
- Other languages have different approaches
- The tension between simplicity and performance has been an open challenge in systems code for the past 25 years