Asynchronous I/O

Ryan Eberhardt
August 18, 2021

Today

- Threads work great — but where do they fall short?
- Introducing an alternative way to write programs called “asynchronous I/O”
- Where do asynchronous I/O models fall short? What can we do about it?

Concurrency with threads

Concurrency with threads

- Here's our basic echo server from Lecture 15:

```
int main(int argc, char *argv[]) {
    int serverSocket = createServerSocket(12345);
    if (serverSocket < 0) {
        cout << "Error: could not start server" << endl;
        return 1;
    }
    size_t connCount = 0;
    while (true) {
        int clientSocket = accept(serverSocket, NULL, NULL);
        size_t connId = connCount++;
        echo(clientSocket, connId);
    }
    return 0;
}
```

🥳 *main thread*

- This code works great… but can only handle one client at a time

Concurrency with threads

- Here's our basic echo server from Lecture 15:

```
int main(int argc, char *argv[]) {
    int serverSocket = createServerSocket(12345);
    if (serverSocket < 0) {
        cout << "Error: could not start server" << endl;
        return 1;
    }
    size_t connCount = 0;
    while (true) {
        int clientSocket = accept(serverSocket, NULL, NULL);
        size_t connId = connCount++;
        echo(clientSocket, connId);
    }
    return 0;
}
```

🥳 *main thread*

- This code works great… but can only handle one client at a time

Concurrency with threads

- Here's our basic echo server from Lecture 15:

```
int main(int argc, char *argv[]) {
    int serverSocket = createServerSocket(12345);
    if (serverSocket < 0) {
        cout << "Error: could not start server" << endl;
        return 1;
    }
    size_t connCount = 0; 🥳 main thread
    while (true) {
        int clientSocket = accept(serverSocket, NULL, NULL);
        size_t connId = connCount++;
        echo(clientSocket, connId);
    }
    return 0;
}
```

- This code works great… but can only handle one client at a time

Concurrency with threads

- Here's our basic echo server from Lecture 15:

```
int main(int argc, char *argv[]) {
    int serverSocket = createServerSocket(12345);
    if (serverSocket < 0) {
        cout << "Error: could not start server" << endl;
        return 1;
    }
    size_t connCount = 0;
    while (true) {  🥳 main thread
        int clientSocket = accept(serverSocket, NULL, NULL);
        size_t connId = connCount++;
        echo(clientSocket, connId);
    }
    return 0;
}
```

- This code works great… but can only handle one client at a time

Concurrency with threads

- Here's our basic echo server from Lecture 15:

```
int main(int argc, char *argv[]) {
    int serverSocket = createServerSocket(12345);
    if (serverSocket < 0) {
        cout << "Error: could not start server" << endl;
        return 1;
    }
    size_t connCount = 0;
    while (true) {
        int clientSocket = accept(serverSocket, NULL, NULL);  🥳 main thread
        size_t connId = connCount++;
        echo(clientSocket, connId);
    }
    return 0;
}
```

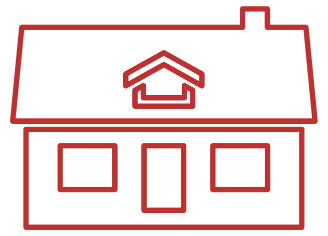
- This code works great… but can only handle one client at a time

Concurrency with threads

- Here's our basic echo server from Lecture 15:

```
int main(int argc, char *argv[]) {
    int serverSocket = createServerSocket(12345);
    if (serverSocket < 0) {
        cout << "Error: could not start server" << endl;
        return 1;
    }
    size_t connCount = 0;
    while (true) {
        int clientSocket = accept(serverSocket, NULL, NULL);
        size_t connId = connCount++;
        echo(clientSocket, connId);
    }
    return 0;
}
```

- This code works great… but can only handle one client at a time



✊🧐 *client 1 connects*

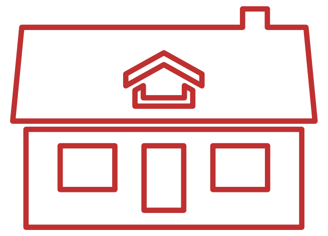
😴 *main thread*

Concurrency with threads

- Here's our basic echo server from Lecture 15:

```
int main(int argc, char *argv[]) {
    int serverSocket = createServerSocket(12345);
    if (serverSocket < 0) {
        cout << "Error: could not start server" << endl;
        return 1;
    }
    size_t connCount = 0;
    while (true) {
        int clientSocket = accept(serverSocket, NULL, NULL);
        size_t connId = connCount++;
        echo(clientSocket, connId);
    }
    return 0;
}
```

🥳 *main thread*



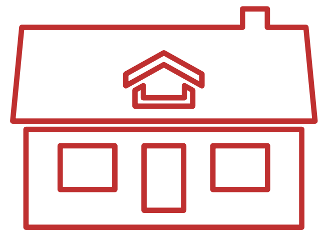
😄 *client 1 connected*

- This code works great… but can only handle one client at a time

Concurrency with threads

- Here's our basic echo server from Lecture 15:

```
int main(int argc, char *argv[]) {
    int serverSocket = createServerSocket(12345);
    if (serverSocket < 0) {
        cout << "Error: could not start server" << endl;
        return 1;
    }
    size_t connCount = 0;
    while (true) {
        int clientSocket = accept(serverSocket, NULL, NULL);
        size_t connId = connCount++;
        echo(clientSocket, connId);   🥳 main thread
    }
    return 0;
}
```



😄 *client 1 connected*

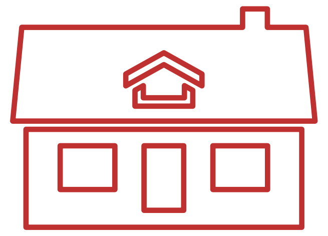
- This code works great… but can only handle one client at a time

Concurrency with threads

- Here's our basic echo server from Lecture 15:

```
int main(int argc, char *argv[]) {
    int serverSocket = createServerSocket(12345);
    if (serverSocket < 0) {
        cout << "Error: could not start server" << endl;
        return 1;
    }
    size_t connCount = 0;
    while (true) {
        int clientSocket = accept(serverSocket, NULL, NULL);
        size_t connId = connCount++;
        echo(clientSocket, connId);
    }
    return 0;
}
```

😴 *main thread*



😄 *client 1 connected*

✊🧐 *client 2 connects…*

- This code works great… but can only handle one client at a time

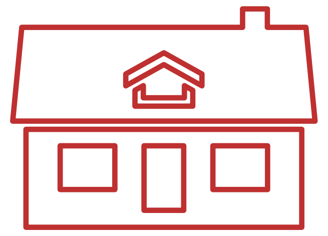
Concurrency with threads

- Here's our basic echo server from Lecture 15:

```
int main(int argc, char *argv[]) {
    int serverSocket = createServerSocket(12345);
    if (serverSocket < 0) {
        cout << "Error: could not start server" << endl;
        return 1;
    }
    size_t connCount = 0;
    while (true) {
        int clientSocket = accept(serverSocket, NULL, NULL);
        size_t connId = connCount++;
        echo(clientSocket, connId);
    }
    return 0;
}
```

😴 *main thread*

- This code works great… but can only handle one client at a time



😄 *client 1 connected*

✊🧐 *client 2 connects…*

- When waiting for a client to connect or when waiting for the client to send data, the main thread is *blocked*. Thread is pulled off the processor so that other threads can do things
  - Usually a good thing
  - But this prevents the thread from doing other things in the meantime

Concurrency with threads

- No problem! We add a ThreadPool:

```
int main(int argc, char *argv[]) {
    ...
    while (true) {
        int clientSocket = accept(serverSocket, NULL, NULL);  😴 main thread
        size_t connId = connCount++;
        pool.schedule([clientFd, connId]{
            echo(clientSocket, connId);  😴 TP thread 1   😴 TP thread 2
        });
    }
}
```

- Now the main thread waits for new incoming connections, while ThreadPool threads wait for clients to send stuff. Implications:
  - Number of simultaneous clients is bounded by the number of threads we can have
  - We switch between talking to clients by switching threads on/off the CPU (context switching)

The Problem with Threads

- Memory overhead: If we have many threads, we consume a lot of memory. This places an upper bound on how many threads we can have
  - Each thread has its own stack space that needs to get managed by the OS. Trying to have 5000 concurrent connections? 5000 threads = 5000 stack segments = 40GB at 8MB/stack! (yike)
- Context switching cost: When we use **blocking** functions within a thread, we discard the rest of the CPU time slice and incur a cost on switching the thread to be *blocked.*
  - Each switch is expensive! Virtual address space needs to get switched, registers need to get restored, cache gets stepped on, etc
  - This is a big cost for high-performance situations (servers). If we have to block on a client, maybe that thread could've done some other work instead.

The Problem with Threads

```
static void echo(int clientFd, size_t connId) {
    sockbuf sb(clientFd);
    iosockstream ss(&sb);
    while (true) {
        string line;
        getline(ss, line);
        if (ss.eof() || ss.fail()) {
            break;
        }
        ss << "\t" << line << endl;
    }
}
```

thread 1

The Problem with Threads

```
static void echo(int clientFd, size_t connId) {
    sockbuf sb(clientFd);
    iosockstream ss(&sb);
    while (true) {
        string line;
        getline(ss, line);
        if (ss.eof() || ss.fail()) {
            break;
        }
        ss << "\t" << line << endl;
    }
}
```

thread 1

The Problem with Threads

```
static void echo(int clientFd, size_t connId) {
    sockbuf sb(clientFd);
    iosockstream ss(&sb);
    while (true) {    thread 1
        string line;
        getline(ss, line);
        if (ss.eof() || ss.fail()) {
            break;
        }
        ss << "\t" << line << endl;
    }
}
```

The Problem with Threads

```
static void echo(int clientFd, size_t connId) {
    sockbuf sb(clientFd);
    iosockstream ss(&sb);
    while (true) {
        string line;
        getline(ss, line); 🥳 thread 1
        if (ss.eof() || ss.fail()) {
            break;
        }
        ss << "\t" << line << endl;
    }
}
```

The Problem with Threads

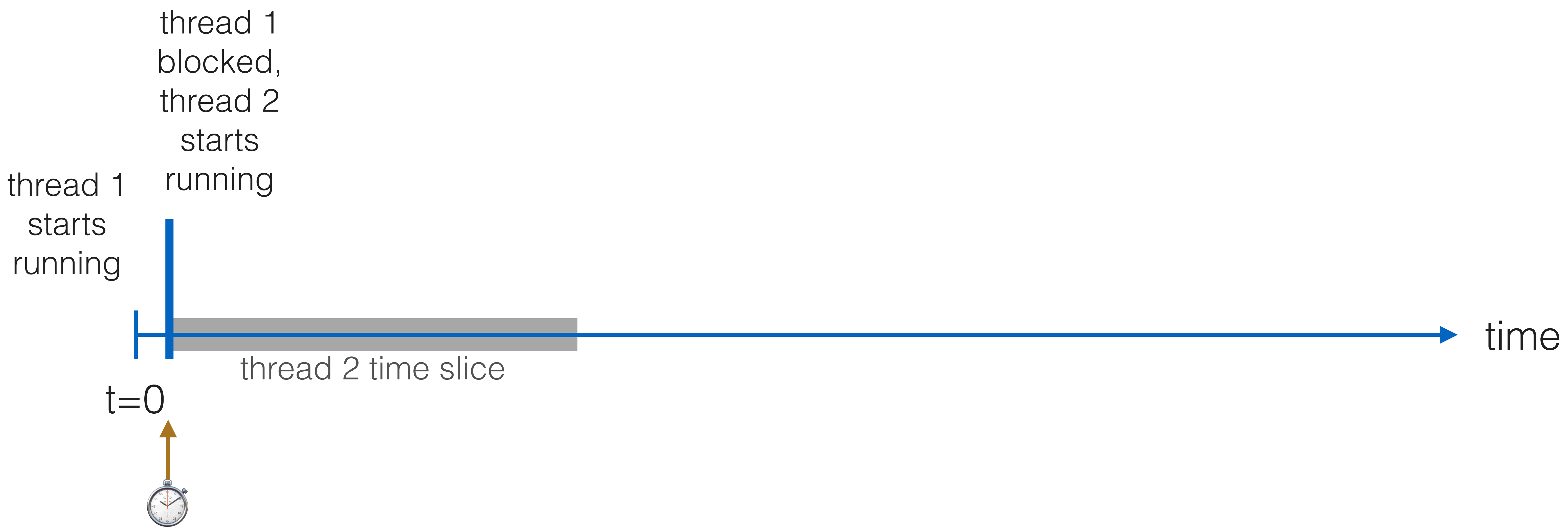
```
static void echo(int clientFd, size_t connId) {
    sockbuf sb(clientFd);
    iosockstream ss(&sb);
    while (true) {
        string line;
        getline(ss, line); 😴 thread 1
        if (ss.eof() || ss.fail()) {
            break;
        }
        ss << "\t" << line << endl;
    }
}
```

The Problem with Threads

```
static void echo(int clientFd, size_t connId) {
    sockbuf sb(clientFd);
    iosockstream ss(&sb);
    while (true) {
        string line;
        getline(ss, line); 😴 thread 1
        if (ss.eof() || ss.fail()) {
            break;
        }
        ss << "\t" << line << endl;
    }
}
```

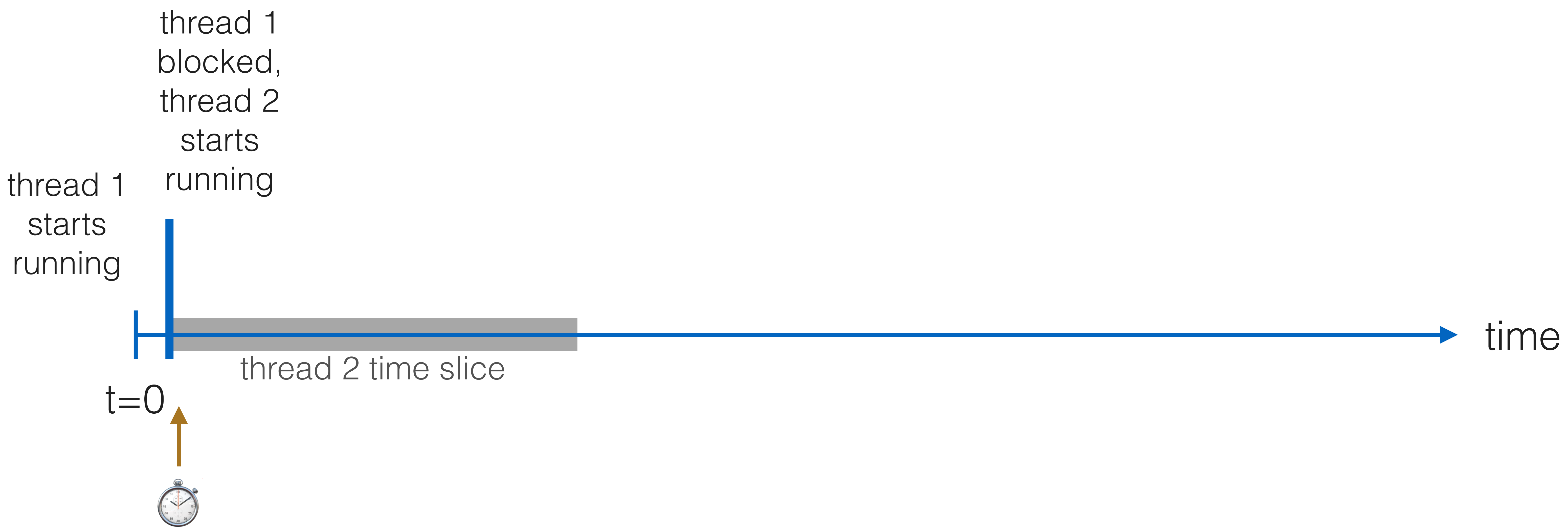
The Problem with Threads

```
static void echo(int clientFd, size_t connId) {
    sockbuf sb(clientFd); 🥳 thread 2
    iosockstream ss(&sb);
    while (true) {
        string line;
        getline(ss, line); 😴 thread 1
        if (ss.eof() || ss.fail()) {
            break;
        }
        ss << "\t" << line << endl;
    }
}
```



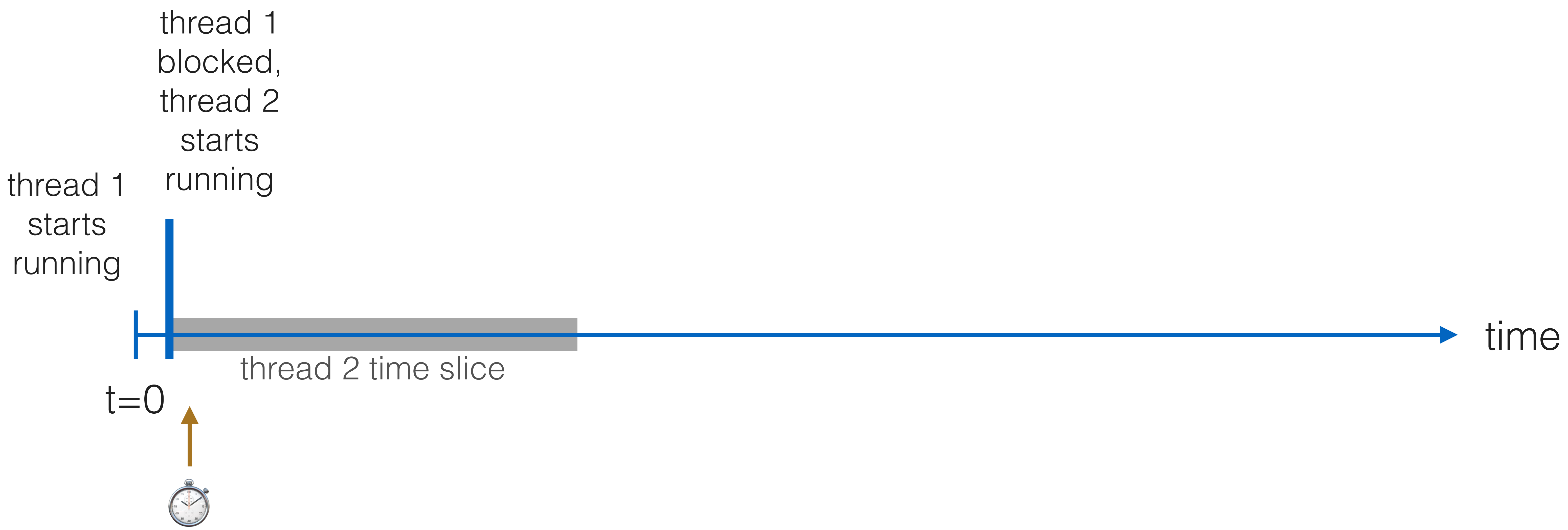
The Problem with Threads

```
static void echo(int clientFd, size_t connId) {
    sockbuf sb(clientFd);
    iosockstream ss(&sb);          // thread 2
    while (true) {
        string line;
        getline(ss, line);         // thread 1
        if (ss.eof() || ss.fail()) {
            break;
        }
        ss << "\t" << line << endl;
    }
}
```



The Problem with Threads

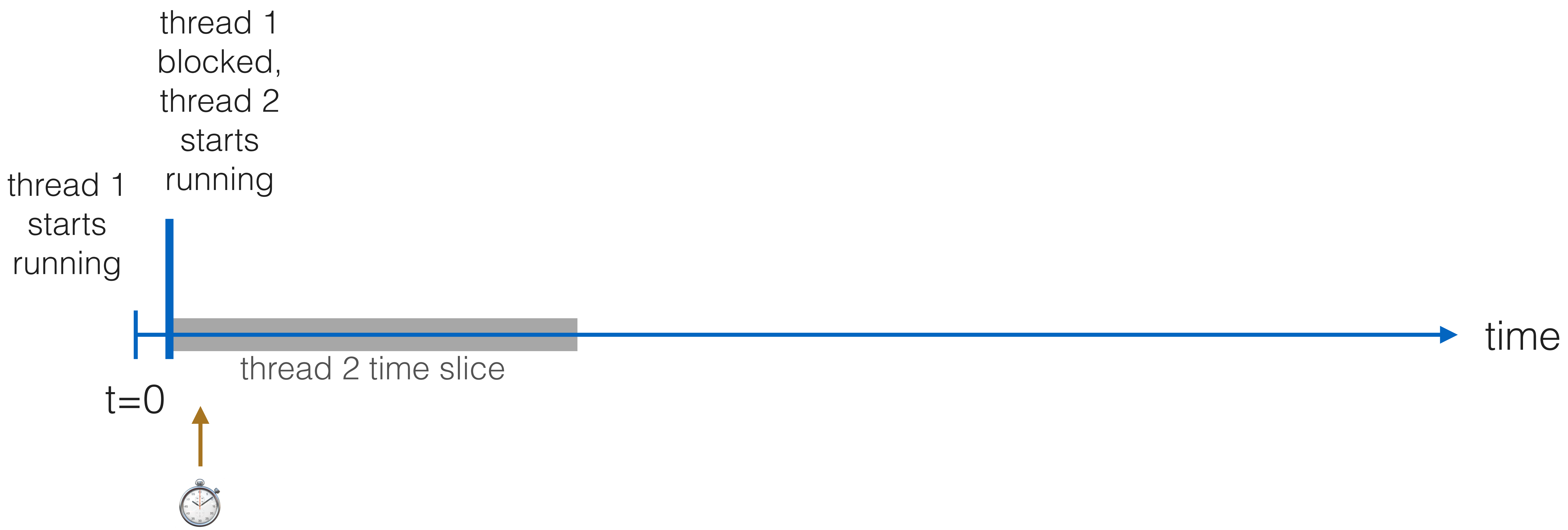
```
static void echo(int clientFd, size_t connId) {
    sockbuf sb(clientFd);
    iosockstream ss(&sb);
    while (true) { 🥳 thread 2
        string line;
        getline(ss, line); 😴 thread 1
        if (ss.eof() || ss.fail()) {
            break;
        }
        ss << "\t" << line << endl;
    }
}
```



The Problem with Threads

```
static void echo(int clientFd, size_t connId) {
    sockbuf sb(clientFd);
    iosockstream ss(&sb);
    while (true) {
        string line;
        getline(ss, line);
        if (ss.eof() || ss.fail()) {
            break;
        }
        ss << "\t" << line << endl;
    }
}
```

thread 1 *thread 2* (at the `getline(ss, line);` line)

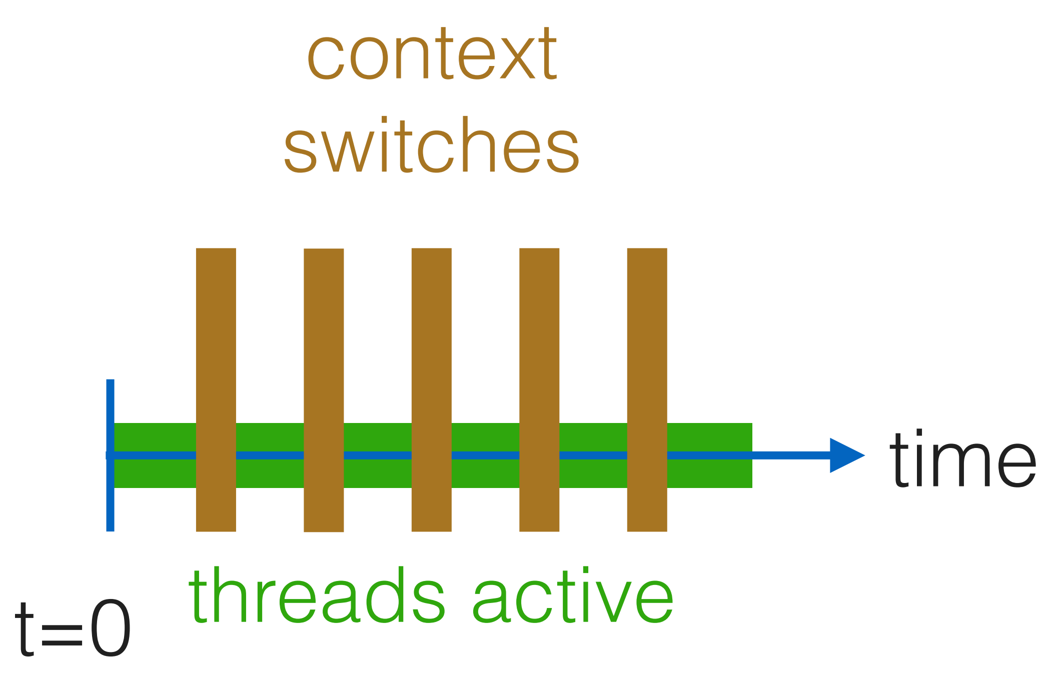


The Problem with Threads

```
static void echo(int clientFd, size_t connId) {
    sockbuf sb(clientFd);
    iosockstream ss(&sb);
    while (true) {
        string line;
        getline(ss, line);
        if (ss.eof() || ss.fail()) {
            break;
        }
        ss << "\t" << line << endl;
    }
}
```

😴 *thread 1* 😴 *thread 2*

The Problem with Threads



- In an I/O-bound application such as a web server, very little time is spent on the CPU before the thread gets blocked and we incur the cost of a context switch
- When a huge number of threads are performing I/O with little computation, context switching represents a significant fraction of CPU time

Roadmap

Threads are great!

But we can't have too many of them, and context switches are expensive

Is there a way we can have concurrency with less penalties?

Non-blocking I/O

- Traditionally, the read() sys call would block if there is more data to be read but not available.
  - This causes the thread to get pulled off the CPU. It can't do anything else in the meantime.
- Instead, we could have read() return a special error value instead of blocking
  - If we see that a client hasn't sent us anything yet, we can do other useful work on this thread e.g. reading from other descriptors we're managing.
- **This allows us to have concurrent I/O with one thread!**

Demo program: receive-two

- Let's implement a basic program that receives data from *two* clients and prints received data to the terminal as it comes in, without any threads
    - First: does it make sense why this is difficult without threads?
- Wait for two clients to connect, then pass their file descriptors to receiveTwoConnections:

```
int main(int argc, char *argv[]) {
    int waitingListFd = createServerSocket(12345);
    if (waitingListFd == -1) {
        cerr << "Failed to bind to port 12345" << endl;
        return 1;
    }

    receiveTwoConnections(
        accept(waitingListFd, NULL, NULL),
        accept(waitingListFd, NULL, NULL));
    return 0;
}
```

```cpp
static void receiveTwoConnections(int client1, int client2) {
    cout << "Printing from two incoming connections" << endl;



    bool client1StillSending = true;
    bool client2StillSending = true;
    while (client1StillSending || client2StillSending) {
        if (client1StillSending) {
            client1StillSending = receiveFromFd(client1, "CLIENT 1");
            if (!client1StillSending) {
                close(client1);
            }
        }
        if (client2StillSending) {
            client2StillSending = receiveFromFd(client2, "CLIENT 2");
            if (!client2StillSending) {
                close(client2);
            }
        }
    }
    cout << "Connections closed" << endl;
}
```

😴 *thread 1*

If client 2 sends data right now, we won't see it!

```cpp
/**
 * Trys reading from the specified file descriptor, printing out the received
 * data if there is any. Returns `true` if the connection is still open, or
 * `false` if the connection has been closed.
 */
static bool receiveFromFd(int fd, const char *clientName) {
    while (true) {
        char buf[512];
        size_t numRead = read(fd, buf, sizeof(buf)); 😴
        if (numRead == 0) {
            // client closed the connection
            return false;
        } else if (numRead == -1) {
            // read() failed
            perror("read");
            return false;
        }

        // If we get here, numRead must be greater than 0, so we actually
        // received something
        cout << clientName << ": " << string(buf, numRead) << endl;
    }
}
```

```
static void configureAsNonblocking(int fd) {
    fcntl(fd, F_SETFL, fcntl(fd, F_GETFL, 0) | O_NONBLOCK);
}
```

```
static void receiveTwoConnections(int client1, int client2) {
    cout << "Printing from two incoming connections" << endl;

    configureAsNonblocking(client1);
    configureAsNonblocking(client2);

    bool client1StillSending = true;
    bool client2StillSending = true;
    while (client1StillSending || client2StillSending) {
        if (client1StillSending) {
            client1StillSending = receiveFromFd(client1, "CLIENT 1");
            if (!client1StillSending) {
                close(client1);
            }
        }
        if (client2StillSending) {
            client2StillSending = receiveFromFd(client2, "CLIENT 2");
            if (!client2StillSending) {
                close(client2);
            }
        }
    }
    cout << "Connections closed" << endl;
}
```

Set O_NONBLOCK on the socket

```
/**
 * Trys reading from the specified file descriptor, printing out the received
 * data if there is any. Returns `true` if the connection is still open, or
 * `false` if the connection has been closed.
 */
static bool receiveFromFd(int fd, const char *clientName) {
    while (true) {
        char buf[512];
        size_t numRead = read(fd, buf, sizeof(buf));
        if (numRead == 0) {
            // client closed the connection
            return false;
        } else if (numRead == -1) {



            // read() failed
            perror("read");
            return false;

        }

        // If we get here, numRead must be greater than 0, so we actually
        // received something
        cout << clientName << ": " << string(buf, numRead) << endl;
    }
}
```

If there is no more data to read at the moment, but the connection is still open, read() will return -1 with errno=EWOULDBLOCK

```
static void configureAsNonblocking(int fd) {
    fcntl(fd, F_SETFL, fcntl(fd, F_GETFL, 0) | O_NONBLOCK);
}
```

```
static void receiveTwoConnections(int client1, int client2) {
    cout << "Printing from two incoming connections" << endl;

    configureAsNonblocking(client1);
    configureAsNonblocking(client2);

    bool client1StillSending = true;
    bool client2StillSending = true;
    while (client1StillSending || client2StillSending) {
        if (client1StillSending) {
            client1StillSending = receiveFromFd(client1, "CLIENT 1");
            if (!client1StillSending) {
                close(client1);
            }
        }
        if (client2StillSending) {
            client2StillSending = receiveFromFd(client2, "CLIENT 2");
            if (!client2StillSending) {
                close(client2);
            }
        }
    }
    cout << "Connections closed" << endl;
}
```

```
/**
 * Trys reading from the specified file descriptor, printing out the received
 * data if there is any. Returns `true` if the connection is still open, or
 * `false` if the connection has been closed.
 */
static bool receiveFromFd(int fd, const char *clientName) {
    while (true) {
        char buf[512];
        size_t numRead = read(fd, buf, sizeof(buf));
        if (numRead == 0) {
            // client closed the connection
            return false;
        } else if (numRead == -1) {
            if (errno == EAGAIN || errno == EWOULDBLOCK) {
                // client is still connected, but there is nothing to read
                // right now. read() would have normally blocked, but we
                // configured the fd to be non-blocking, so we see EAGAIN
                // instead
                return true;
            } else {
                // read() failed
                perror("read");
                return false;
            }
        }

        // If we get here, numRead must be greater than 0, so we actually
        // received something
        cout << clientName << ": " << string(buf, numRead) << endl;
    }
}
```

Demo: /usr/class/cs110/samples/aio/receive-two

```
top
top - 16:42:37 up 25 days, 14:30, 13 users,  load average: 0.22, 0.07, 0.02
Tasks: 328 total,   2 running, 319 sleeping,   7 stopped,   0 zombie
%Cpu(s):  6.5 us,  6.1 sy,  0.0 ni, 87.4 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  31990.7 total,  16001.7 free,   1259.3 used,  14729.7 buff/cache
MiB Swap:  32641.0 total,  32641.0 free,      0.0 used.  30232.4 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
4144528 rebs      20   0    8252   1764   1580 R 100.0   0.0   0:14.07 receive-two
      1 root      20   0  171436  15676   8380 S   0.0   0.0   4:38.69 systemd
      2 root      20   0       0      0      0 S   0.0   0.0   0:01.53 kthreadd
      3 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_gp
      4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_par_gp
      6 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/0:0H-kblockd
      9 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 mm_percpu_wq
     10 root      20   0       0      0      0 S   0.0   0.0   0:12.14 ksoftirqd/0
```

```
top - 16:42:37 up 25 days, 14:30, 13 users,  load average: 0.22, 0.07, 0.02
Tasks: 328 total,   2 running, 319 sleeping,   7 stopped,   0 zombie
%Cpu(s):  6.5 us,  6.1 sy,  0.0 ni, 87.4 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :  31990.7 total,  16001.7 free,   1259.3 used,  14729.7 buff/cache
MiB Swap:  32641.0 total,  32641.0 free,      0.0 used.  30232.4 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
4144528 rebs      20   0    8252   1764   1580 R 100.0   0.0   0:14.07 receive-two
      1 root      20   0  171436  15676   8380 S   0.0   0.0   4:38.69 systemd
      2 root      20   0       0      0      0 S   0.0   0.0   0:01.53 kthreadd
      3 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_gp
      4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_par_gp
      6 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/0:0H-kblockd
      9 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 mm_percpu_wq
     10 root      20   0       0      0      0 S   0.0   0.0   0:12.14 ksoftirqd/0
```

```cpp
static void receiveTwoConnections(int client1, int client2) {
    cout << "Printing from two incoming connections" << endl;

    configureAsNonblocking(client1);
    configureAsNonblocking(client2);

    bool client1StillSending = true;
    bool client2StillSending = true;
    while (client1StillSending || client2StillSending) {
        if (client1StillSending) {
            client1StillSending = receiveFromFd(client1, "CLIENT 1");
            if (!client1StillSending) {
                close(client1StillSending);
            }
        }
        if (client2StillSending) {
            client2StillSending = receiveFromFd(client2, "CLIENT 2");
            if (!client1StillSending) {
                close(client1StillSending);
            }
        }
    }
    cout << "Connections closed" << endl;
}
```

This loop doesn't block…
… at all…
… even when there is *no* data to process

epoll: wait until a file descriptor is ready

- The epoll API allows us to register a set of file descriptors to watch
- epoll_wait puts us to sleep until a file descriptor is ready for reading/writing
  - Not unlike assignment 4: sigwait() to wait until there is an update with a child process, then call waitpid() with WNOHANG in a loop to get all the updates
  - Here: epoll_wait() to wait until there is new data coming in on a file descriptor. Then read() with O_NONBLOCK to get all the received data

```
static void receiveTwoConnections(int client1, int client2) {
    cout << "Printing from two incoming connections" << endl;

    configureAsNonblocking(client1);
    configureAsNonblocking(client2);

    int epollFd = epoll_create1(0);
    addToWatchSet(epollFd, client1);
    addToWatchSet(epollFd, client2);

    size_t numConnections = 2;
    while (numConnections > 0) {
        struct epoll_event event;
        epoll_wait(epollFd, &event, 1, -1);
        const char *clientName = event.data.fd == client1 ? "CLIENT 1" : "CLIENT 2";
        bool clientStillSending = receiveFromFd(event.data.fd, clientName);
        if (!clientStillSending) {
            removeFromWatchSet(epollFd, event.data.fd);
            close(event.data.fd);
            numConnections--;
            cout << clientName << " closed" << endl;
        }
    }
    cout << "All connections closed" << endl;

    close(epollFd);
}
```

Create set of file descriptors we want to watch

event.data.fd now has the fd number that is ready for reading

^ now receive all the data that client sent

```
static void addToWatchSet(int epollFd, int fd) {
    struct epoll_event event;
    event.events = EPOLLIN | EPOLLET;
    event.data.fd = fd;
    epoll_ctl(epollFd, EPOLL_CTL_ADD, fd, &event);
}
```

event.data is an epoll_data *union* that allows us to store 8 bytes of data, which will be returned to us by epoll_wait when this fd is ready for reading

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

```
static void removeFromWatchSet(int epollFd, int fd) {
    epoll_ctl(epollFd, EPOLL_CTL_DEL, fd, NULL);
}
```

Fully asynchronous I/O

- epoll can be used with any number of file descriptors, of any type

```
int main(int argc, char *argv[]) {
    int waitingListFd = createServerSocket(12345);
    if (waitingListFd == -1) {
        cerr << "Failed to bind to port 12345" << endl;
        return 1;
    }

    configureAsNonblocking(waitingListFd);

    int epollFd = epoll_create1(0);
    addToWatchSet(epollFd, waitingListFd);

    while (true) {
        struct epoll_event event;
        epoll_wait(epollFd, &event, 1, -1);
        if (event.data.fd == waitingListFd) {
            acceptSomeClients(epollFd, event.data.fd);
        } else {
            receiveFromClient(epollFd, event.data.fd);
        }
    }
    return 0;
}
```

Commonly called the *event loop*: Waits for something to happen, then does something quick in response. No threading, no blocking!

```
static void acceptSomeClients(int epollFd, int waitingListFd) {
    while (true) {
        int clientFd = accept(waitingListFd, NULL, NULL);
        if (clientFd == -1) {
            // Let's assume for this example that this is because
            // EAGAIN/EWOULDBLOCK
            break;
        }
        cout << "Received new connection! Client fd " << clientFd << endl;
        configureAsNonblocking(clientFd);
        addToWatchSet(epollFd, clientFd);
    }
}
```

- We could add other types of file descriptors and other actions to the event loop, e.g. responding to keyboard input on stdin, responding to a signal, etc.

New benefits

- With only one thread, context switching overhead is eliminated
- Overhead of each connection is miniscule
  - We don’t need to create a whole stack every time we want to support a new connection
- This model can easily support 10k+ simultaneous connections with just a single thread
  - In fact, this is your *only* practical option for applications looking to support 10-100k+ concurrent connections. Standard for HTTP servers nowadays
  - Another example: scanning the internet (see guest talk next Wednesday!)

The dark side of epoll

The dark side of epoll

- State management is hard. Real world applications get very messy, very quickly
- Asynchronous I/O interfaces are usually platform-specific
  - epoll is Linux only. Mac and other BSD derivatives have kqueue, Solaris has /dev/poll, Windows has I/O completion ports
- There are so many small details that need to be perfectly correct in order for async I/O applications to work correctly

State management is hard

- Our sample asynchronous I/O server is pretty simple…
  - because it does almost nothing useful
  - Key point: it maintains almost no state per connection. Just prints out whatever incoming data is received
- Real life is much more complicated. When an fd is ready, what are we supposed to do with it?
- Painful code: https://web.archive.org/web/20120504033548/https://banu.com/blog/2/how-to-use-epoll-a-complete-example-in-c/
- Actual applications:
  - The client just sent me the last part of an HTTP request. What were all of the earlier parts of the request it sent me before?
  - Was I waiting for the client to send me something, or was I in the middle of sending something to the client?
  - Alice the Client asked me for her emails, but I needed to get them from Bob the Database. Now Bob the Database responded with some info, but I can’t remember what I was supposed to do with it

State management with threads: Solved by the stack

- When an event happens (e.g. data comes in), how do we remember what we were in the middle of doing? How do we remember previous data?
  - Multithreading: the saved %rip register tells us what we were doing
  - the thread's stack stores any previous data we still need

```
static void handleRequest(iosockstream &ss) {
    Request request = readRequest(ss);          ⟵ %rip 👀
    size_t emailId = parseRequestedEmailId(request);
    Email email = getEmail(emailId);
    sendEmail(ss, email);
}

static void getEmail(size_t emailId) {
    Connection conn = openDatabaseConnection();
    return conn.queryEmail(emailId);
}
```

```
main:
sockbuf sb
iosockstream ss
---
handleRequest:
iosockstream &ss
```

State management with threads: Solved by the stack

- When an event happens (e.g. data comes in), how do we remember what we were in the middle of doing? How do we remember previous data?
  - Multithreading: the saved %rip register tells us what we were doing
  - the thread's stack stores any previous data we still need

```
static void handleRequest(iosockstream &ss) {
    Request request = readRequest(ss);
    size_t emailId = parseRequestedEmailId(request);   ← %rip 👀
    Email email = getEmail(emailId);
    sendEmail(ss, email);
}

static void getEmail(size_t emailId) {
    Connection conn = openDatabaseConnection();
    return conn.queryEmail(emailId);
}
```

```
main:
sockbuf sb
iosockstream ss
---
handleRequest:
iosockstream &ss
Request request
```

State management with threads: Solved by the stack

- When an event happens (e.g. data comes in), how do we remember what we were in the middle of doing? How do we remember previous data?
  - Multithreading: the saved %rip register tells us what we were doing
  - the thread's stack stores any previous data we still need

```
static void handleRequest(iosockstream &ss) {
    Request request = readRequest(ss);
    size_t emailId = parseRequestedEmailId(request);
    Email email = getEmail(emailId);           ← %rip 👀
    sendEmail(ss, email);
}

static void getEmail(size_t emailId) {
    Connection conn = openDatabaseConnection();
    return conn.queryEmail(emailId);
}
```

```
main:
sockbuf sb
iosockstream ss
```

```
handleRequest:
iosockstream &ss
Request request
size_t emailId
```

State management with threads: Solved by the stack

- When an event happens (e.g. data comes in), how do we remember what we were in the middle of doing? How do we remember previous data?
  - Multithreading: the saved %rip register tells us what we were doing
  - the thread's stack stores any previous data we still need

```
static void handleRequest(iosockstream &ss) {
    Request request = readRequest(ss);
    size_t emailId = parseRequestedEmailId(request);
    Email email = getEmail(emailId);   ←
    sendEmail(ss, email);
}

static void getEmail(size_t emailId) {   ← %rip 👀
    Connection conn = openDatabaseConnection();
    return conn.queryEmail(emailId);
}
```

```
main:
sockbuf sb
iosockstream ss
-------------------
handleRequest:
iosockstream &ss
Request request
size_t emailId
-------------------
getEmail:
size_t emailId
```

State management with threads: Solved by the stack

- When an event happens (e.g. data comes in), how do we remember what we were in the middle of doing? How do we remember previous data?
  - Multithreading: the saved %rip register tells us what we were doing
  - the thread's stack stores any previous data we still need

```
static void handleRequest(iosockstream &ss) {
    Request request = readRequest(ss);
    size_t emailId = parseRequestedEmailId(request);
    Email email = getEmail(emailId);  ←
    sendEmail(ss, email);
}

static void getEmail(size_t emailId) {
    Connection conn = openDatabaseConnection();
    return conn.queryEmail(emailId);  ← %rip 👀
}
```

If we block on querying the email, the stack still stores all context we need to eventually send this email back to the client

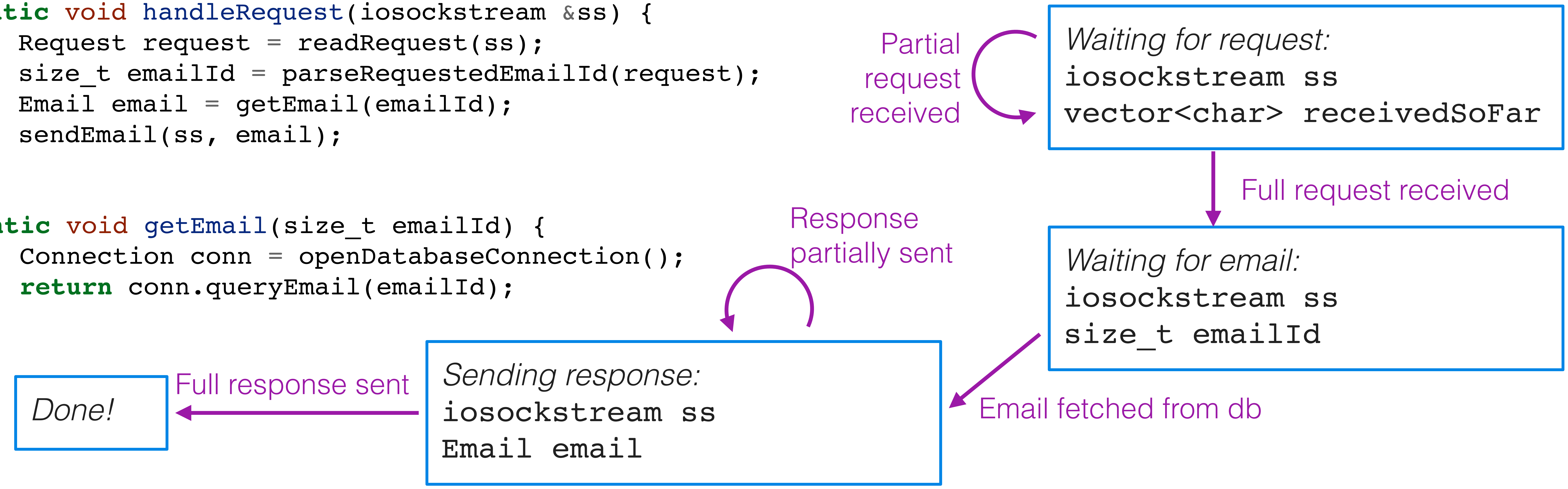
```
main:
sockbuf sb
iosockstream ss
---
handleRequest:
iosockstream &ss
Request request
size_t emailId
---
getEmail:
size_t emailId
Connection conn
```

State management with AIO: what to do?

- Somehow need to store a snapshot of everything that is happening right now and what needs to happen next
- State machines (CS 103):
  - Define each state we can be waiting in
  - Define transitions, driven by something that happens (e.g. new data comes in)

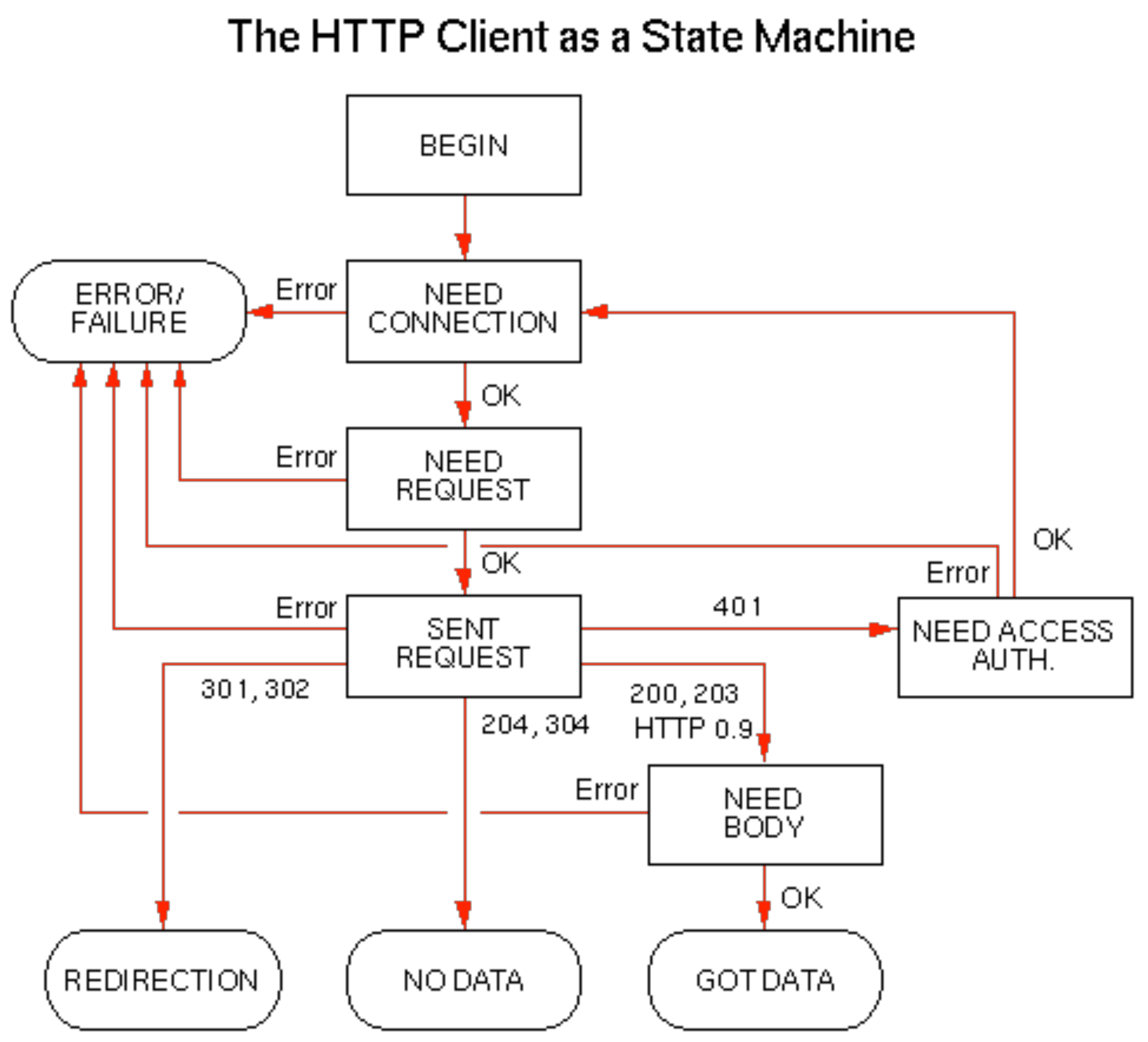
```
static void handleRequest(iosockstream &ss) {
    Request request = readRequest(ss);
    size_t emailId = parseRequestedEmailId(request);
    Email email = getEmail(emailId);
    sendEmail(ss, email);
}

static void getEmail(size_t emailId) {
    Connection conn = openDatabaseConnection();
    return conn.queryEmail(emailId);
}
```



State management with AIO: what to do?

- State machine for implementing HTTP client:



https://www.w3.org/Library/User/Architecture/HTTPFeatures.html

State management with AIO: what to do?

- When something happens on a file descriptor, we can get the state machine associated with that file descriptor, see what state it's in, and follow the state transition associated with whatever just happened (e.g. incoming data)
- Manually specifying/implementing state machines is still really hard and complicated… But it's a lot better than nothing

AIO interfaces are generally platform-specific

- epoll is Linux only. Mac and other BSD derivatives have kqueue, Solaris has /dev/poll, Windows has I/O completion ports
- Each of these has totally different semantics for the small (but important) details
- How to implement portable async programs??

AIO: The devil is in the details 😈

- Epoll, and IO interfaces in general, are extremely hard to use correctly. Many small details need to be *just perfect*
- Small selection of problems:
  - Fairness/starvation: if we keep getting a massive amount of data coming in on one fd, that is likely to prevent us from processing data on other fds
  - With multiprocessing/multithreading: Thundering herd problem: "a large number of processes or threads waiting for an event are awoken when that event occurs, but only one process is able to handle the event. When the processes wake up, they will each try to handle the event, but only one will win. All processes will compete for resources, possibly freezing the computer, until the herd is calmed down again."
- Further reading:
  - https://idea.popcount.org/2017-02-20-epoll-is-fundamentally-broken-12/
  - https://blog.cloudflare.com/the-sad-state-of-linux-socket-balancing/

Better solutions: Better abstractions

Better abstractions

- Key point: we need simpler abstractions so you can focus on solving your problem without having to think about all the details of async I/O
- Over the years, people have developed a few ways to cope
- Most promising idea ("coroutines"/"async/await"):
  - Write code that looks almost like normal threaded code
  - The compiler or interpreter will compile your code into a state machine ("promise" or "future")
  - Submit the promise/future to the event loop ("executor") and the runtime will take care of all the messy business

Better abstractions

- Example (Rust):

```
async fn addToInbox(email_id: u64, recipient_id: u64)
    -> Result<(), Error>
{
    let message = loadMessage(email_id).await?;
    let recipient = get_recipient(recipient_id).await?;
    recipient.verifyHasSpace(&message)?;
    recipient.addToInbox(message).await
}
```

- Javascript:

```
async function addToInbox(email_id, recipient_id) {
    let message = await loadMessage(email_id);
    let recipient = await get_recipient(recipient_id);
    recipient.verifyHasSpace(message);
    await recipient.addToInbox(message);
}
```

- C++20 finally got coroutines, but they aren't very usable yet. Maybe wait another 2 years
  Relevant: https://www.scs.stanford.edu/~dm/blog/c++-coroutines.html

Takeaways

Takeaways

- Async I/O is a must for extremely high concurrency
- Not useful for CPU-bound work, when you're using the time slice and have less concurrency
- Avoid manual nonblocking + epoll if at all possible. Use stackless coroutines/promises/futures if possible
- If you need to use it, take the time to explore the many pitfalls first