Lecture 32: I/O multiplexing

David Hovemeyer

April 27, 2020

601.229 Computer Systems Fundamentals

Example code

Example code for today is on course website in `iomux.zip`

Blocking operations

Why we need concurrency for server programs

Server main loop:

```
int server_fd = open_listenfd(port);

while (1) {
  int client_fd =
    Accept(server_fd, NULL, NULL);
  chat_with_client(client_fd);
  close(client_fd);
}
```

Why we need concurrency for server programs

Server main loop:

```
int server_fd = open_listenfd(port);

while (1) {
  int client_fd =
    Accept(server_fd, NULL, NULL); // Indefinite wait
  chat_with_client(client_fd);
  close(client_fd);
}
```

Why we need concurrency for server programs

Server main loop:

```
int server_fd = open_listenfd(port);

while (1) {
  int client_fd =
    Accept(server_fd, NULL, NULL);
  chat_with_client(client_fd);     // Indefinite wait
  close(client_fd);
}
```

Why we need concurrency for server programs

Server main loop:

```
int server_fd = open_listenfd(port);

while (1) {
  int client_fd =
    Accept(server_fd, NULL, NULL);
  chat_with_client(client_fd);
  close(client_fd);
}
```

The server is not responsive while

Why we need concurrency for server programs

Server main loop:

```
int server_fd = open_listenfd(port);

while (1) {
  int client_fd =
    Accept(server_fd, NULL, NULL);
  chat_with_client(client_fd);
  close(client_fd);
}
```

The server is not responsive while

1. Waiting for client connection to arrive

Why we need concurrency for server programs

Server main loop:

```
int server_fd = open_listenfd(port);

while (1) {
  int client_fd =
    Accept(server_fd, NULL, NULL);
  chat_with_client(client_fd);
  close(client_fd);
}
```

The server is not responsive while

1. Waiting for client connection to arrive
2. Waiting to receive data from client

Why we need concurrency for server programs

Server main loop:

```
int server_fd = open_listenfd(port);

while (1) {
  int client_fd =
    Accept(server_fd, NULL, NULL);
  chat_with_client(client_fd);
  close(client_fd);
}
```

The server is not responsive while

1. Waiting for client connection to arrive
2. Waiting to receive data from client
3. Waiting to send data to client (sometimes required by TCP protocol)

Blocking operations

- Operation such as `accept`, `read`, and `write` can *block*

Blocking operations

- Operation such as `accept`, `read`, and `write` can *block*
- “Blocking” means that the OS kernel suspends the calling thread until the operation has completed

Blocking operations

- Operation such as `accept`, `read`, and `write` can *block*
- "Blocking" means that the OS kernel suspends the calling thread until the operation has completed
- E.g., when calling `accept`, the calling thread is blocked until a request for a new client connection

Blocking operations

- Operation such as `accept`, `read`, and `write` can *block*
- "Blocking" means that the OS kernel suspends the calling thread until the operation has completed
- E.g., when calling `accept`, the calling thread is blocked until a request for a new client connection
- Problem: while a thread is blocked, it can't do anything else

Blocking operations

- Operation such as `accept`, `read`, and `write` can *block*
- "Blocking" means that the OS kernel suspends the calling thread until the operation has completed
- E.g., when calling `accept`, the calling thread is blocked until a request for a new client connection
- Problem: while a thread is blocked, it can't do anything else
- So, there is no way to support multiple simultaneous clients, and have the server be responsive, using a single thread

Blocking operations

- Operation such as `accept`, `read`, and `write` can *block*
- "Blocking" means that the OS kernel suspends the calling thread until the operation has completed
- E.g., when calling `accept`, the calling thread is blocked until a request for a new client connection
- Problem: while a thread is blocked, it can't do anything else
- So, there is no way to support multiple simultaneous clients, and have the server be responsive, using a single thread
  - Or is there?

Nonblocking I/O

- Modern operating systems support *nonblocking* I/O
- In Unix/Linux, a file descriptor can be made nonblocking
- All operations that would normally block are guaranteed not to block if the filed descriptor is nonblocking
- If a blocking operation (`accept`, `read`, `write`) is invoked, but it can't be completed immediately:
  - Operation returns an error
  - `errno` is set to `EWOULDBLOCK` error code

Aside: errno, error codes

- When a C library or system call function fails, `errno` is set to an integer error code to indicate the reason for the failure
- Available using `#include <errno.h>`
- It's not actually a global variable (because that wouldn't work in a multithreaded program)
- Actual definition in the Linux C library (glibc):

```
extern int *__errno_location (void) __THROW __attribute_const__;
# define errno (*__errno_location ())
```

- `__errno_location` function returns a pointer to an integer variable allocated in thread-local storage
  - So, each thread has its own `errno`

An idea

Could we handle multiple client connections simultaneously as long as the server avoids doing any blocking I/O?

An idea

Could we handle multiple client connections simultaneously as long as the server avoids doing any blocking I/O?

Challenge: how do we know which file descriptors are ready to perform I/O?

I/O multiplexing

I/O multiplexing

Alternative approach for supporting multiple simultaneous client connections

Basic idea: server maintains sets of active file descriptors (mostly client connections, but also for file I/O)

I/O multiplexing

Alternative approach for supporting multiple simultaneous client connections

Basic idea: server maintains sets of active file descriptors (mostly client connections, but also for file I/O)

Main server loop uses `select` or `poll` system call to check which file descriptors are *ready*, meaning that a read or write can be performed without blocking

I/O multiplexing

Alternative approach for supporting multiple simultaneous client connections

Basic idea: server maintains sets of active file descriptors (mostly client connections, but also for file I/O)

Main server loop uses `select` or `poll` system call to check which file descriptors are *ready*, meaning that a read or write can be performed without blocking

Compared to using processes or threads for concurrency:

- Advantage: less overhead (CPU, memory) per client connection than processes or threads
- Disadvantage: higher code complexity

select system call

The `select` system call:

```
#include <sys/select.h>

int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

`readfds`, `writefds`, and `exceptfds` are sets of file descriptors

`select` waits until at least one file descriptor has become ready for reading or writing, or has an exceptional condition

- `readfds`, `writefds`, and/or `exceptfds` are *modified* to indicate the specific file descriptors that are ready
- `timeout` specifies maximum amount of time to wait, NULL means indefinitely

fd_set

An `fd_set` represents a set of file descriptors

Operations (where `set` is an `fd_set` variable):

- `FD_ZERO(&set)`: make `set` empty
- `FD_SET(fd, &set)`: add `fd` to `set`
- `FD_CLR(fd, &set)`: remove `fd` from `set`
- `FD_ISSET(fd, &set)`: true if `fd` is in `set`, false otherwise

I/O multiplexing main loop

Pseudo-code:

```
create server socket, add to active fd set

while (1) {
  wait for fd to become ready (select or poll)

  if server socket ready
    accept a connection, add it to set

  for fd in client connections
    if fd is ready for reading, read and update connection state
    if fs is ready for writing, write and update connection state
}
```

Updating connection state

The main difficulty of using I/O multiplexing is that communication with clients is *event-driven*

Updating connection state

The main difficulty of using I/O multiplexing is that communication with clients is *event-driven*

When data is read from the client, event-processing code must figure out what to do with it

- Data read might be a partial message

Updating connection state

The main difficulty of using I/O multiplexing is that communication with clients is *event-driven*

When data is read from the client, event-processing code must figure out what to do with it

- Data read might be a partial message

Similar issue when sending data to client: data might need to be sent in chunks

Maintaining and updating state of client connections is more complicated compared to code for process- or thread-based concurrency

- With these approaches, we can just use normal loops and control flow

Example: echo server

- Example: `echoserv.c`
- Protocol: read one line of text from client, send same line back, repeat until `quit` is received

Connection data structure

Per-connection data structure:

```
#define CONN_READING  0
#define CONN_WRITING  1
#define CONN_DONE     2

struct Connection {
  char in_buf[BUFFER_SIZE];
  char out_buf[BUFFER_SIZE];
  int in_count, out_pos, out_count;
  int state;
};
```

`in_buf`, `in_count`: data received from client

`out_buf`, `out_pos`, `out_count`: data to be sent to client

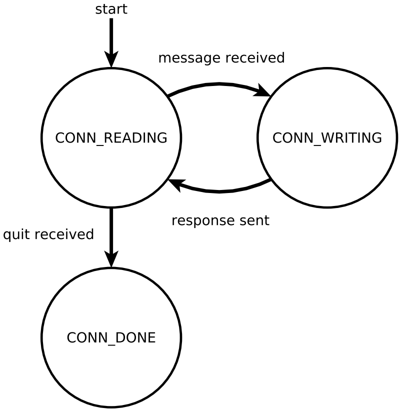
`state`: client state (`CONN_READING`, `CONN_WRITING`, or `CONN_DONE`)

State machines

A synchronous network protocol can be modeled as a *state machine*

In a protocol implementation using threads or processes for concurrency, state is implicit

When implementing a protocol with I/O multiplexing, state must be explicit



Making a file descriptor nonblocking

Even when using `select` or `poll` to determine when file descriptors are ready, it is still a good idea to make them nonblocking

Avoids situations where an I/O operation might block

Making a file descriptor nonblocking:

```
void make_nonblocking(int fd) {
  int flags = fcntl(fd, F_GETFL, 0);
  if (flags < 0) {
    fatal("fcntl failed: could not get flags");
  }
  flags |= O_NONBLOCK;
  if (fcntl(fd, F_SETFL, flags) < 0) {
    fatal("fcntl failed: could not set flags");
  }
}
```

readfds and writefds

- Server has two `fd_sets`, `readfds` and `writefds`
- These specify the file descriptors that the server wants to check for being ready to read (`readfds`) or write (`writefds`)
- The server socket and the client file descriptors of all connections in the `CONN_READING` state are placed in `readfds`
- The client file descriptors of all connections in the `CONN_WRITING` state are placed in `writefds`
- Each call to `select` determines which file descriptors in `readfds` are ready for reading, and which file descriptors in `writefds` are ready for writing
  - If the server socket file descriptor is ready for reading, it means that a connection request has arrived (and a call to `accept` will not block)

Building readfds and writefds

```
// Code executed for each iteration of server main loop

// Place client socket fds in readfds and writefds as appropriate
for (int fd = 0; fd <= maxfd; fd++) {
  struct Connection *conn = client_conn[fd];
  if (conn) {
    if (conn->state == CONN_READING) {
      FD_SET(fd, &readfds);
    } else if (conn->state == CONN_WRITING) {
      FD_SET(fd, &writefds);
    }
  }
}

// Server socket is always in readfds
FD_SET(serverfd, &readfds);
```

Determine which file descriptors are ready

```
int rc = select(maxfd + 1, &readfds, &writefds, NULL, NULL);
if (rc < 0) {
  fatal("select failed");
}
```

The `maxfd` variable keeps track of the maximum file descriptor value: `select` is more efficient when it checks fewer file descriptors for readiness

Accept a client connection

```
if (FD_ISSET(serverfd, &readfds)) {
  int clientfd = Accept(serverfd, NULL, NULL);
  make_nonblocking(clientfd);
  if (clientfd > maxfd) {
    maxfd = clientfd;
  }
  client_conn[clientfd] = create_client_conn();
}
```

Service client connections

```
for (int fd = 0; fd <= maxfd; fd++) {
  if (client_conn[fd] != NULL) {
    struct Connection *conn = client_conn[fd];
    if (FD_ISSET(fd, &readfds)) {
      client_do_read(fd, conn);
    }
    if (FD_ISSET(fd, &writefds)) {
      client_do_write(fd, conn);
    }
    if (conn->state == CONN_DONE) {
      close(fd);
      free(conn);
      client_conn[fd] = NULL;
    }
  }
}
```

client_do_read

```
void client_do_read(int fd, struct Connection *conn) {
  int remaining = BUFFER_SIZE - conn->in_count - 1;

  ssize_t rc = read(fd, conn->in_buf + conn->in_count, remaining);
  if (rc < 0) {
    fatal("read failed");
  }
  conn->in_count += rc;

  // process the data that was read
  ...40+ lines of code omitted...
}
```

client_do_read

```
void client_do_read(int fd, struct Connection *conn) {
  int remaining = BUFFER_SIZE - conn->in_count - 1;

  ssize_t rc = read(fd, conn->in_buf + conn->in_count, remaining);
  if (rc < 0) {
    fatal("read failed");
  }
  conn->in_count += rc;

  // process the data that was read
  ...40+ lines of code omitted...
}
```

Code is fairly complicated because it must

client_do_read

```
void client_do_read(int fd, struct Connection *conn) {
  int remaining = BUFFER_SIZE - conn->in_count - 1;

  ssize_t rc = read(fd, conn->in_buf + conn->in_count, remaining);
  if (rc < 0) {
    fatal("read failed");
  }
  conn->in_count += rc;

  // process the data that was read
  ...40+ lines of code omitted...
}
```

Code is fairly complicated because it must

- Determine if a complete message was received

client_do_read

```
void client_do_read(int fd, struct Connection *conn) {
  int remaining = BUFFER_SIZE - conn->in_count - 1;

  ssize_t rc = read(fd, conn->in_buf + conn->in_count, remaining);
  if (rc < 0) {
    fatal("read failed");
  }
  conn->in_count += rc;

  // process the data that was read
  ...40+ lines of code omitted...
}
```

Code is fairly complicated because it must

- Determine if a complete message was received
- If so, copy it to out_buf, deal with leftover data, update connection state

client_do_write

```
void client_do_write(int fd, struct Connection *conn) {
  int remaining = conn->out_count - conn->out_pos;
  ssize_t rc = write(fd, conn->out_buf + conn->out_pos, (size_t) remaining);
  if (rc < 0) {
    fatal("write failed");
  }

  conn->out_pos += rc;
  if (conn->out_pos == conn->out_count) {
    conn->state = CONN_READING;
  }
}
```

client_do_write

```
void client_do_write(int fd, struct Connection *conn) {
  int remaining = conn->out_count - conn->out_pos;
  ssize_t rc = write(fd, conn->out_buf + conn->out_pos, (size_t) remaining);
  if (rc < 0) {
    fatal("write failed");
  }

  conn->out_pos += rc;
  if (conn->out_pos == conn->out_count) {
    conn->state = CONN_READING;
  }
}
```

Fairly straightforward: just try to copy data from `out_buf` to the client socket

That wasn't so bad?

- The I/O multiplexing echo server implementation not terribly complex (a little over 200 lines of code)

That wasn't so bad?

- The I/O multiplexing echo server implementation not terribly complex (a little over 200 lines of code)
- **However**: the protocol was *very* simple

That wasn't so bad?

- The I/O multiplexing echo server implementation not terribly complex (a little over 200 lines of code)
- **However**: the protocol was *very* simple
  - and even so, `client_do_read` was quite complicated!

That wasn't so bad?

- The I/O multiplexing echo server implementation not terribly complex (a little over 200 lines of code)
- **However**: the protocol was *very* simple
  - and even so, `client_do_read` was quite complicated!
- Real protocols (e.g., HTTP) would be *much* more complicated to implement

That wasn't so bad?

- The I/O multiplexing echo server implementation not terribly complex (a little over 200 lines of code)
- **However**: the protocol was *very* simple
  - and even so, `client_do_read` was quite complicated!
- Real protocols (e.g., HTTP) would be *much* more complicated to implement
- It would be nice if there were a way to get the benefits of I/O multiplexing, but write our code in a "threaded" style rather than an "event-driven" style

I/O multiplexing with coroutines

Coroutines

One way to reduce the complexity of I/O multiplexing is to implement communication with clients using *coroutines*

Coroutines are, essentially, a lightweight way of implementing threads

- But with runtime cost closer to function call overhead

Each client connection is implemented as a coroutine

When a client file descriptor finds that a client fd is ready for reading or writing, it *yields* to the client coroutine

Client coroutine will do I/O, and then yield back to the main routine

Echo server implementation with coroutines

- `echoserv_co.c` is an echo server implementation using coroutines
- Similar number of lines of code as `echoserv.c`
- However, 30 lines of code are coroutine-aware versions of `read` and `write`
  - They check for `EWOULDBLOCK` and yield back to the main routine if a call to `read` or `write` would block
- Server main loop is very similar
- Actual protocol implementation is much simpler!

Echo server client coroutine

```
void chat_with_client(void) {
  struct Connection *conn = (struct Connection *) aco_get_arg();
  for (;;) {
    // read a line
    conn->state = CONN_READING;
    co_readline(conn);

    // if line was "quit", we're done
    if (strcmp(conn->out_buf, "quit") == 0) {
      break;
    }

    // echo line back to client
    conn->state = CONN_WRITING;
    co_write_fully(conn->fd, conn->out_buf, strlen(conn->out_buf));
    co_write_fully(conn->fd, "\r\n", 2);
  }
  aco_exit();
}
```

Observations

- The `chat_with_client` function looks almost exactly like a thread start function
- The assignments to `conn->state` help the main routine know when to schedule the coroutine (based on the readiness of its file descriptor for reading or writing)
- The `co_readline` and `co_write_fully` functions are "coroutine-aware" I/O functions which yield back to the main routine if a call to `read` or `write` would block
- See complete code for details