523454

Computer Network Programming

Multiplexing TCP connections

Dr. Parin Sornlertlamvanich,

parin.s@sut.ac.th

Multiplexing TCP connections

- The socket APIs are blocking by default
  - **accept()** to wait for an incoming connection, your program's execution is blocked until a new incoming connection
  - **recv()** to read incoming data, your program's execution blocks until new data is available
- Blocking I/O can be a significant problem
  - Lab2's program needed to serve multiple clients

Multiplexing TCP connections (cont.)

- Then, imagine that one slow client connected to it
  - this slow client takes a minute before sending its first data
  - our server would simply be waiting on the **recv()** call to return
  - If other clients were trying to connect, they would have to wait it out
- Blocking also isn't usually acceptable on the client side either
  - In web browser, it has a tab feature where many whole web pages can be loaded in parallel
  - a technique for handling many separate connections simultaneously

(1) Polling non-blocking sockets

- To configure sockets to use a non-blocking operation
  - calling **fcntl()** with the O_NONBLOCK flag

- Once in non-blocking mode
  - a call to **recv()** with no data will return immediately
  - simply check each of its active sockets in turn
  - It would handle any socket that returned data and ignore any socket that didn’t – called Polling

(1) Polling non-blocking sockets (cont.)

- Polling can be a waste of computer resources since most of the time
  - there will be no data to read
  - It also complicates the program (the programmer is required to manually track which sockets are active and which state, they are in)
  - Return values from **recv()** must also be handled differently than with blocking sockets

(2) Forking and multithreading

- To start a new thread or process for each connection
- In this case, blocking sockets are fine
  - they block only their servicing thread/process, and they do not block other threads/processes
- The fork() function splits the executing program into two separate processes

(2) Forking and multithreading (cont.)

```
while(1) {
    socket_client = accept(socket_listen, &new_client, &new_client_length);
    int pid = fork();
    if (pid == 0) { //child process
        close(socket_listen);
        recv(socket_client, ...);
        send(socket_client, ...);
        close(socket_client);
        exit(0);
    }
    //parent process
    close(socket_client);
}
```

- A multi-process TCP server may accept connections like this:
  - the program blocks on accept()
  - When a new connection is established, the program calls **fork()** to split into two processes
  - The child process, where pid == 0, only services this one connection
    - Using **recv()** freely without worrying about blocking
  - The parent process calls close() and returns to listening for more conns

(3) Synchronous multiplexing with select()

- **select() <-> a set of sockets**
  - It tells us which ones are ready to be read
  - It can also tell us which sockets are ready to write to
- int select (int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);

(3) Synchronous multiplexing with select()

- `int select (maxDescPlus1, &readDescs, &writeDescs, &exceptionDescs, &timeout);`
  - maxDescsPlus1: integer, hint of the maximum number of descriptors
  - readDescs: `fd_set`, checked for immediate input availability
  - writeDescs: `fd_set`, checked for the ability to immediately write data
  - exceptionDescs: `fd_set`, checked for pending exceptions
  - timeout: `struct timeval`, how long it blocks (NULL → forever)

  \- **returns** the total number of ready descriptors, -1 in case of error

  \- **changes** the descriptor lists so that only the corresponding positions are set

```
int FD_ZERO  (fd_set *descriptorVector);                  /* removes all descriptors from vector */
int FD_CLR   (int descriptor, fd_set *descriptorVector);  /* remove descriptor from vector */
int FD_SET   (int descriptor, fd_set *descriptorVector);  /* add descriptor to vector */
int FD_ISSET (int descriptor, fd_set *descriptorVector);  /* vector membership check */
```

```
struct timeval {
    time_t tv_sec;   /* seconds */
    time_t tv_usec;  /* microseconds */
};
```

(3) Synchronous multiplexing with select()

- Before calling select(), we must first add our sockets into an fd_set

- If we have three sockets, socket_listen, socket_a, and socket_b
- To add them to an fd_set

```
fd_set our_sockets;
FD_ZERO(&our_sockets);
FD_SET(socket_listen, &our_sockets);
FD_SET(socket_a, &our_sockets);
FD_SET(socket_b, &our_sockets);
```

It is important to zero-out the fd_set using FD_ZERO() before use

(3) Synchronous multiplexing with select()

- select() also requires that we pass a number that's larger than the largest socket descriptor

```
int max_socket;
max_socket = socket_listen;
if (socket_a > max_socket) max_socket = socket_a;
if (socket_b > max_socket) max_socket = socket_b;
```

- When we call select(), it modifies our fd_set of sockets to indicate which sockets are ready
  - Therefore, we want to copy our socket set before calling it

```
fd_set copy;
copy = our_sockets;

select(max_socket+1, &copy, 0, 0, 0);
```

(3) Synchronous multiplexing with select()

- This call blocks until at least one of the sockets is ready to be read from
- When select() returns
  - **copy** is modified so that it only contains the sockets that are ready to be read from
  - To check which sockets are still in copy using **FD_ISSET()**

```
if (FD_ISSET(socket_listen, &copy)) {
        //socket_listen has a new connection
        accept(socket_listen...
}
if (FD_ISSET(socket_a, &copy)) {
        //socket_a is ready to be read from
        recv(socket_a...
}
if (FD_ISSET(socket_b, &copy)) {
        //socket_b is ready to be read from
        recv(socket_b...
}
```

(3) Synchronous multiplexing with select()

- **If we wanted to monitor an** fd_set **for writability instead of readability**
  - pass our fd_set as the <u>third</u> argument to **select()**

- **We can monitor a set of sockets for exceptions by passing it as**
  - the <u>fourth</u> argument to **select()**

select() timeout

- The last argument allows us to specify a timeout
  - **tv_sec** holds the number of seconds, and **tv_usec** holds the number of microseconds

```
struct timeval {
    long tv_sec;
    long tv_usec;
}
```

- **If we want** select() **to wait a maximum of 1.5 seconds**
  - **select()** returns after a socket in **fd_set copy** is ready to read or after 1.5 seconds has elapsed, whichever is sooner

```
struct timeval timeout;
timeout.tv_sec = 1;
timeout.tv_usec = 500000;
select(max_socket+1, &copy, 0, 0, &timeout);
```

(3) Synchronous multiplexing with select()

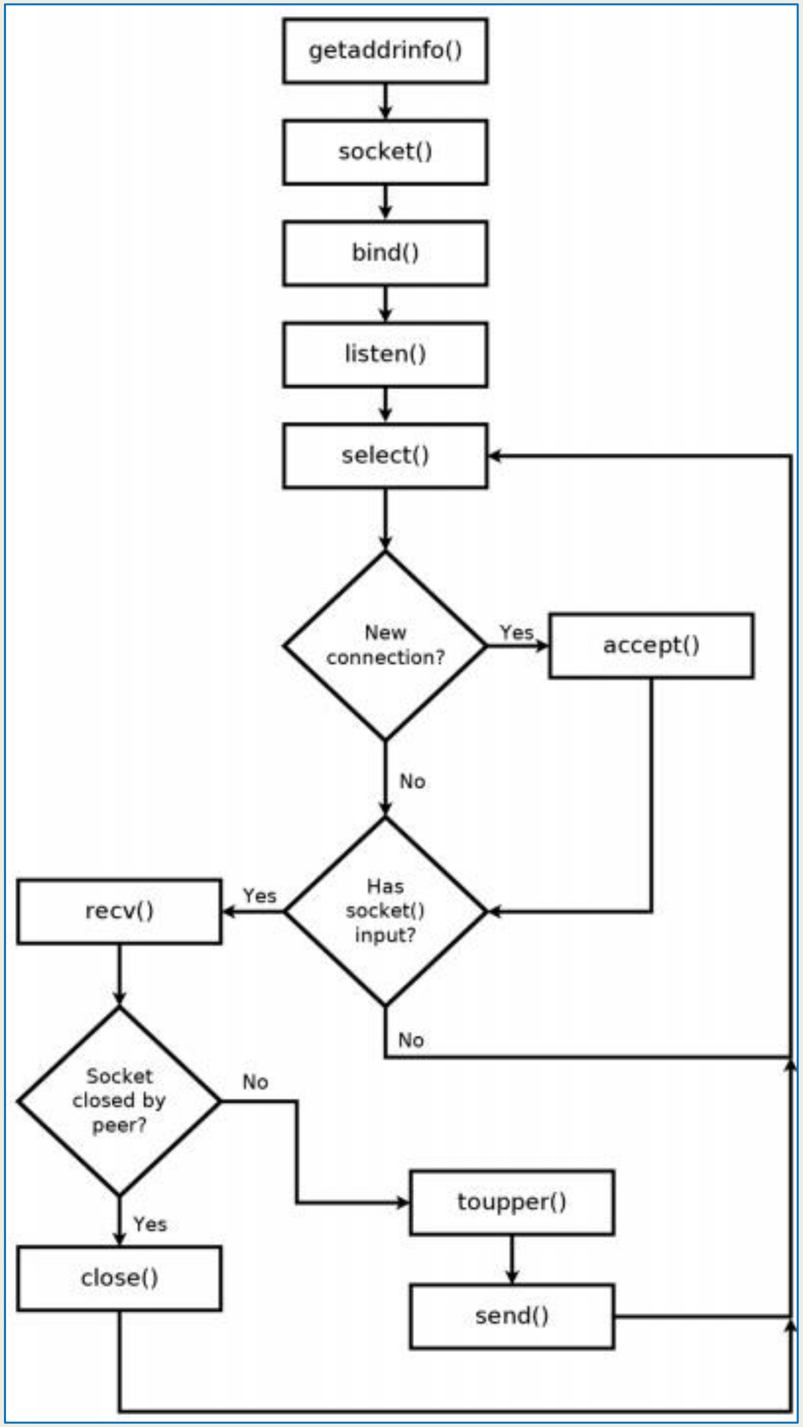
- select() can also be used to monitor for writeable sockets, and sockets with exceptions
  - sockets where we could call **send()** without blocking

```
select(max_sockets+1, &ready_to_read, &ready_to_write,
&excepted, &timeout);
```

- **On success,** select() **itself returns the number of socket descriptors**
  - The return value is zero if it timed out before
  - returns -1 to indicate an error

Example: TCP server

- TCP server that converts strings into uppercase
- If a client connects and sends Hello
  - The server will send **HELLO** back
- **To call** select(), **which alerts us if a new connection is available**
  - a new connection is waiting do we call accept()
- **When data is received by** recv(), **we run it through** toupper()
  - return it to the client using **send()**



getaddrinfo() -> socket() -> bind() -> listen()

```
struct addrinfo *bind_address;
getaddrinfo(0, "7777", &hints, &bind_address);

printf("Creating socket...\n");
int socket_listen;
socket_listen = socket(bind_address->ai_family,
            bind_address->ai_socktype, bind_address->ai_protocol);

printf("Binding socket to local address...\n");
if (bind(socket_listen,
            bind_address->ai_addr, bind_address->ai_addrlen)) {
    fprintf(stderr, "bind() failed. (%d)\n", GETSOCKETERRNO());
    return 1;
}
freeaddrinfo(bind_address);

printf("Listening...\n");
if (listen(socket_listen, 10) < 0) {
    fprintf(stderr, "listen() failed. (%d)\n", GETSOCKETERRNO());
    return 1;
}
```

Example: TCP server

- getaddrinfo() -> socket() -> bind() -> listen()
- **To define an** fd_set **structure that stores all of the active sockets**
  - For now, we add only our **listening socket ->** **max_socket** (it's the only socket)

```
fd_set master;
FD_ZERO(&master);
FD_SET(socket_listen, &master);
SOCKET max_socket = socket_listen;
```

Example: TCP server

- To add new connections to master
  - To enter the **main loop**, and set up our call to **select()**

```
printf("Waiting for connections...\n");

while(1) {
	fd_set reads;
	reads = master;
	if (select(max_socket+1, &reads, 0, 0, 0) < 0) {
		fprintf(stderr, "select() failed. (%d)\n", GETSOCKETERRNO());
		return 1;
	}
...
```

- This works by first <u>copying</u> our fd_set **master** into **reads.**
- Recall that **select()** modifies the set given to it. If we didn't copy master, we would <u>lose its data</u>
- Timeout value of 0 -> it <u>doesn't return</u> until a socket in the master set is <u>ready</u> to be read from

Example: TCP server

- Whether it was flagged by select() as being ready. If a socket, X, was flagged by select()
  - then FD_ISSET(X, &reads) is true
- Socket descriptors are positive integers, so we can try every possible socket descriptor up to max_socket

```
int i;
for(i = 1; i <= max_socket; ++i) {
	if (FD_ISSET(i, &reads)) {
	//Handle socket – Next slide
	}
}
```

Example: TCP server

- FD_ISSET() is only true for sockets that are ready to be read

- In the case of socket_listen, this means that a new connection is ready to be established with accept()
  - We should first determine whether the current socket is the listening one or not. If it is, we call **accept()**

- For all other sockets, it means that data is ready to be read with recv()

```
if (i == socket_listen) {
        struct sockaddr_storage client_address;
        socklen_t client_len = sizeof(client_address);
        int socket_client = accept(socket_listen,
                (struct sockaddr*) &client_address,
                &client_len);
        if (socket_client == -1) {
                fprintf(stderr, "accept() failed. (%d)\n",
                        GETSOCKETERRNO());
                return 1;
        }

        FD_SET(socket_client, &master);
        if (socket_client > max_socket)
                max_socket = socket_client;

        char address_buffer[100];
        getnameinfo((struct sockaddr*)&client_address,
                        client_len,
                        address_buffer, sizeof(address_buffer), 0, 0,
                        NI_NUMERICHOST);
        printf("New connection from %s\n", address_buffer);
} else {
...
```

- **If the** socket i **is not** socket_listen
  - then it is instead a request for an established connection (**existing connection**)
  - to read it with **recv()**, convert it into **uppercase**

```
} else {
        char read[1024];
        int bytes_received = recv(i, read, 1024, 0);
        if (bytes_received < 1) {
                FD_CLR(i, &master);
                close(i);
                continue;
        }
        int j;
        for (j = 0; j < bytes_received; ++j)
                read[j] = toupper(read[j]);
        send(i, read, bytes_received, 0);
}
```

Example: TCP server

```
                } //if FD_ISSET
            } //for i to max_socket
        } //while(1)

        printf("Closing listening socket...\n");
        close(socket_listen);

        printf("Finished.\n");
        return 0;
}
```

Example: TCP server

```
┌──(kali㉿kali)-[~/lab_netPro/my_lab/lab3/temp]
└─$ ./serv
Configuring local address...
Creating socket...
Binding socket to local address...
Listening...
Waiting for connections...
New connection from 127.0.0.1
New connection from 127.0.0.1
New connection from 127.0.0.1
```

```
┌──(kali㉿kali)-[~/lab_netPro/my_lab/lab3/temp]
└─$ ./client 127.0.0.1 7777
Configuring remote address...
Remote address is: 127.0.0.1 7777
Creating socket...
Connecting...
Connected.
To send data, enter text followed by enter.
CPE2
Sending: CPE2
Sent 5 bytes.
Received (5 bytes): CPE2
```

```
┌──(kali㉿kali)-[~/lab_netPro/my_lab/lab3/temp]
└─$ ./client 127.0.0.1 7777
Configuring remote address...
Remote address is: 127.0.0.1 7777
Creating socket...
Connecting...
Connected.
To send data, enter text followed by enter.
CPE1
Sending: CPE1
Sent 5 bytes.
Received (5 bytes): CPE1
```

```
┌──(kali㉿kali)-[~/lab_netPro/my_lab/lab3/temp]
└─$ ./client 127.0.0.1 7777
Configuring remote address...
Remote address is: 127.0.0.1 7777
Creating socket...
Connecting...
Connected.
To send data, enter text followed by enter.
CPE3
Sending: CPE3
Sent 5 bytes.
Received (5 bytes): CPE3
SUT3
Sending: SUT3
Sent 5 bytes.
Received (5 bytes): SUT3
```