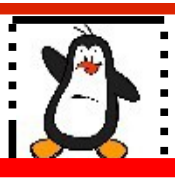


Video Lecture # 37
Socket Programming - IV
Design of Concurrent Servers

Course: SYSTEM PROGRAMMING

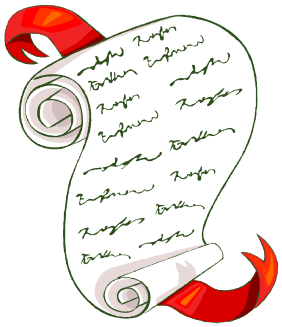
Instructor: Arif Butt

Punjab University College of Information Technology (PUCIT)
University of the Punjab

Source Code files available at: `https://bitbucket.org/arifpucit/spvl-repo/src`
Lecture Slides available at: `http://arifbutt.me`

Today's Agenda

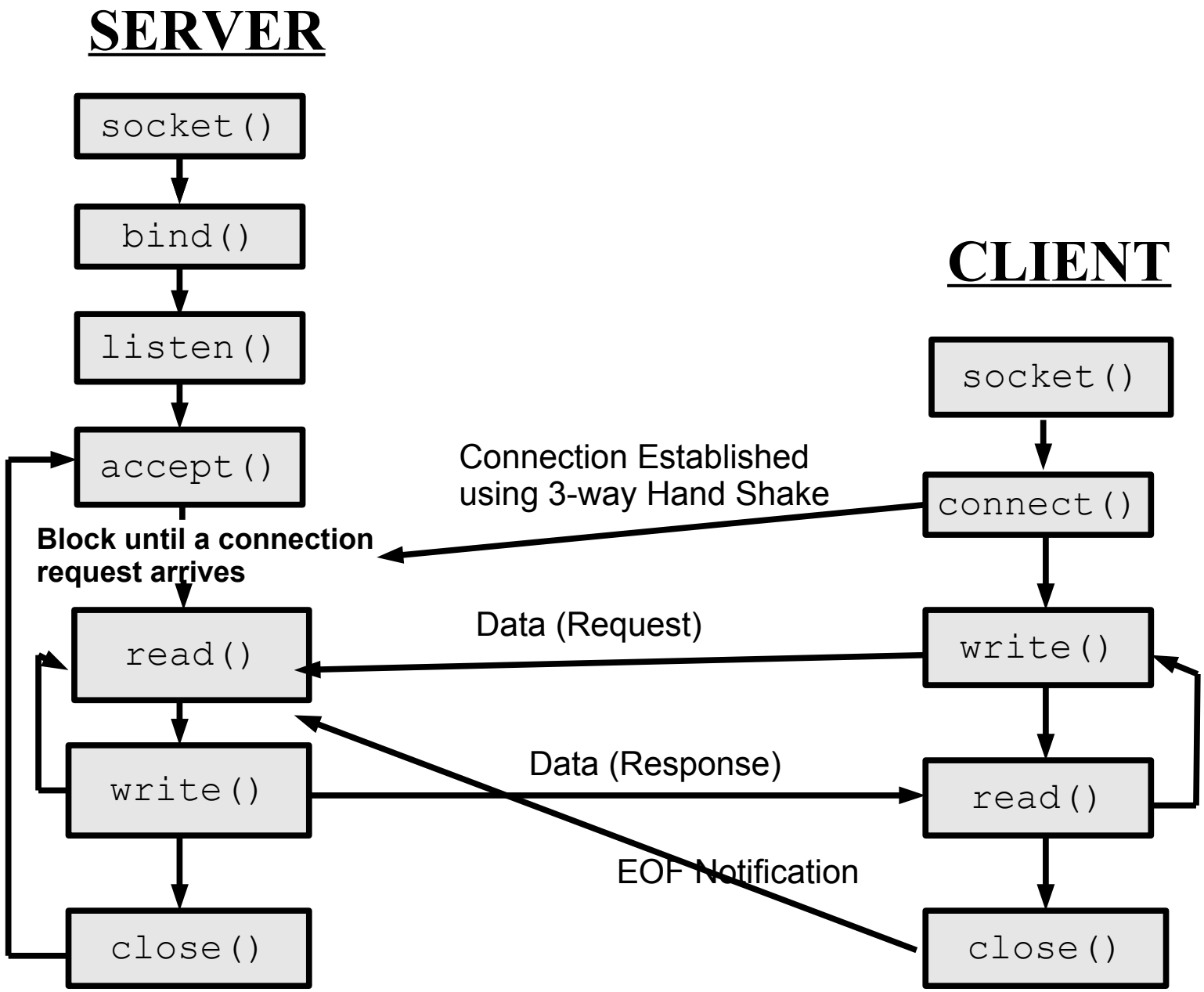
- Iterative vs Concurrent Servers
- Multiple clients accessing a Iterative `echo` Server
- Design and Code of Concurrent `echo` Server
  - Using `fork()`
  - Using `pthread_create()`
  - Using `select()`
- Concurrent Clients
- Assignment: Extending the `nc(1)` command to follow the concurrent model



Internetworking with Linux:
https://www.youtube.com/playlist?list=PL7B2bn3G_wfD6_mhy-eLdn_mFgQ_mOyLl

What happens when an iterative server get requests from multiple clients?

System Call Graph: TCP Sockets



Iterative TCP echo Server

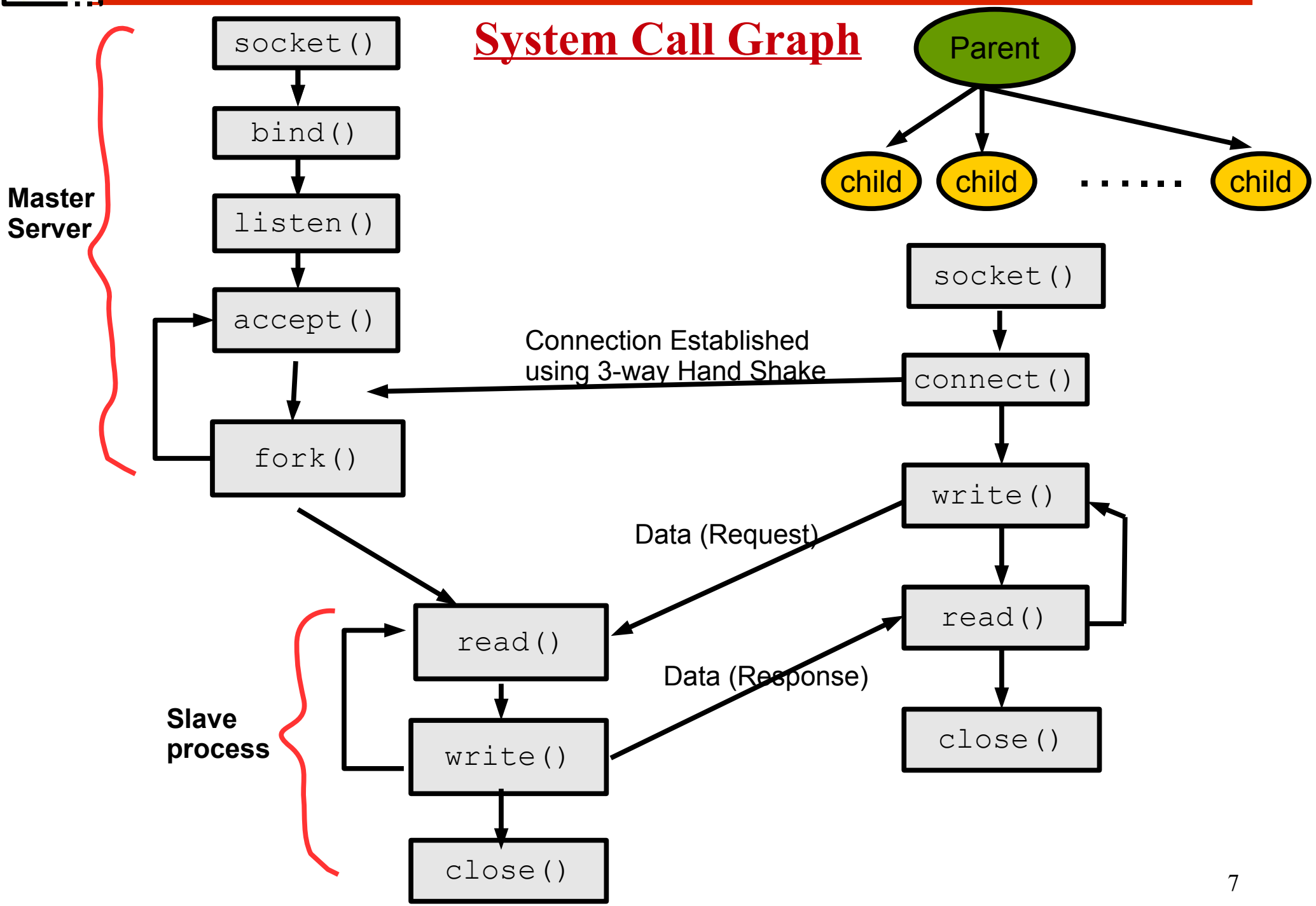
`tcpechoserver.c, tcpechoclient.c`

Concurrent TCP Server

- Concurrent models mostly use Stream sockets and can be implemented using following three techniques:
- **Multi-process Model:** Multiple single threaded processes (using `fork()` system call to cater for every new client). This is done if each slave can operate in isolation. To achieve maximal concurrency in case of multiprocessors
- **Multi-threaded Model:** Multiple threads within a single process. Use `pthread_create()` library call to cater for every new client. This is done if each slave need to share data with the parent or with each other
- **Non-Blocking Multiplexed I/O:** Single thread within a single process using asynchronous I/O. Server create a non-blocking socket and use `select()` system call to cater for reading and writing on multiple clients using socket multiplexing

Concurrent Connection-oriented Model

System Call Graph



Pseudocode: Concurrent Connection Oriented Server

Multiprocess: using `fork()`

```
socket();
bind();
listen();
while(1){
   accept();
    fork();      // create a child process
    if (child){
       Close master socket
       Communicate with data socket
       Close data socket
       exit();
    }
   else
   {
       Close data socket
       Move up to accept a new client connection
   }
}
```

Concurrent TCP echo Server Using `fork()`

`tcpechoserver_forked.c`

Pseudocode: Concurrent Connection Oriented Server

Multi-Threaded: using pthread_create()

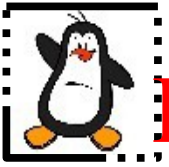
```
socket();
bind();
listen();
while(1){
   accept();
    pthreadcreate(); /*create a detached thread and pass
                              data socket to it as argument
    Move up to accept a new client connection
}


void* threadfunction(void* arg){
       Communicate using data socket
       Close data socket
       pthread_exit();
    }
```

Concurrent TCP echo Server Using `pthread_create()`

`tcpechoserver_threaded.c`

I/O Multiplexing using `select()`



Introduction to `select()`

- A process sometimes, expects input from two different sources, but it doesn't know which input will be available first. For example, consider a process trying to read from source A, but input is only available from source B, and the process blocks
- One solution of monitoring multiple file descriptors is to use a separate process/thread for each descriptor
- The other method is perform I/O multiplexing, which can be done using
  - The `select()` system call (appeared in BSD UNIX)
  - The `poll()` system call (appeared in System V UNIX)
- Now a days both are required by SUSv3

select() System call

```
int select(int nfds, fd_set* rfds, fd_set* wfds,
           fd_set* efds, struct timeval* timeout);
```

- The `select()` call allows a program to monitor multiple file descriptors, until one or more of the file descriptors become ready for some class of I/O operation. A file descriptor is considered ready if it is possible to perform a corresponding I/O operation w/o blocking
- The `select()` call takes three descriptor sets of type `fd_set` as arguments:
  - `rfds` is the set of file descriptors to be tested to see if input is possible
  - `wfds` is the set of file descriptors to be tested to see if output is possible
  - `efds` is the set of file descriptors to be tested to see if exceptional condition has occurred
- The first argument `nfds` must be at least one greater than the largest file descriptor in all three descriptor sets (to achieve performance gains)
- The last argument `timeout` is a value that forces `select()` to return after a certain time has elapsed, even if no descriptors are ready. A NULL over here means block indefinitely

select() System call (cont...)

```
int select(int nfds, fd_set* rfds, fd_set* wfds,
           fd_set* efds, struct timeval* timeout);
```

- The `select()` call returns one of the following:
  - A -1 indicating that an error occurred
  - A 0 indicating that the call timed out before any file descriptor became ready
  - A value greater than zero, indicates the total number of ready file descriptors in all three descriptor sets
- If a bit in a descriptor set is 0, it means that the corresponding descriptor is not ready and a value of 1 means the corresponding descriptor is ready
- The `fd_set` data type is implemented as bit fields in arrays of integers and can be manipulated via four macros:

FD_ZERO(&fds)	Initialize descriptor set, `fds`, to all zeros
FD_SET(fd, &fds)	Include the given descriptor `fd` in the descriptor set `fds`
FD_CLR(fd, &fds)	Exclude the given descriptor `fd` from the descriptor set `fds`
FD_ISSET(fd, &fds)	Test if descriptor `fd` in descriptor set `fds` has a value of 0 or 1. Returns 0 if `fd` is not in the set `fds` and nonzero otherwise

Concurrent TCP echo Server Using `select()`

`tcpechoserver_select.c`

Concept of Concurrent Clients

- We all know that concurrent servers are used to reduce the average response time experienced by a client. However, the need of a concurrent client is not so obvious
- Consider a TCP `echo` client, which expect input from two descriptors, one from the user via keyboard and other from the client-side socket that is connected to the server-side socket
- A user may type some text using the keyboard any time and the server response for a previous client message may also arrive at the client-side socket descriptor at any instant of time (depending on the load on the server process and network traffic)
- Since the two descriptors become ready asynchronously, so you can implement such concurrent clients by using the `select()` system call

Assignment: Design and Code Concurrent version of `nc(1)`

Things To Do

If you have problems visit me in counseling hours. . . .