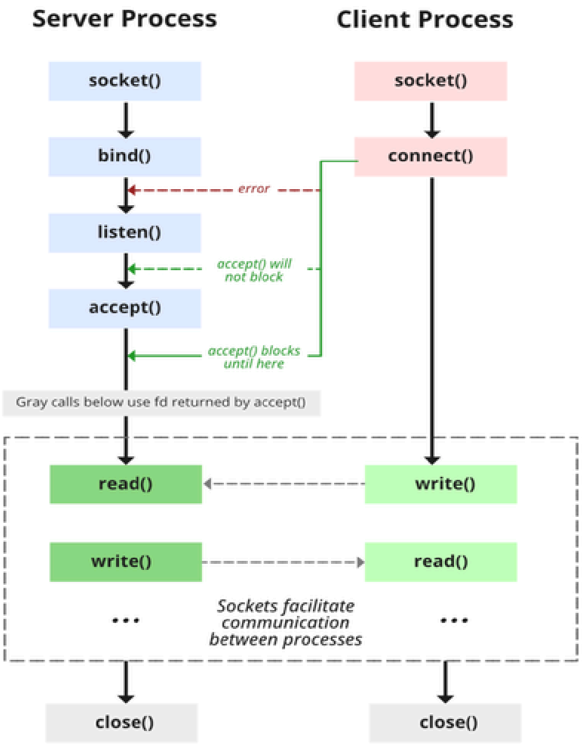
National University of Computer and Emerging Sciences (NUCES) Islamabad

Computer Networks Lab
Spring 2024
Week 08

Socket Programming (TCP Concurrent Servers)

Socket Function Calls

1. **Socket():** To create a socket
2. **Bind()**: It’s a socket identification like a telephone number to contact
3. **Listen()**: Ready to receive a connection
4. **Connect():** Ready to act as a sender
5. **Accept()**: Confirmation, it is like accepting to receive a call from a sender
6. **Send()**: To send data (write)
7. **Recv()**: To receive data (read)
8. **Close()**: To close a connection



State diagram for server and client model of Socket

Concurrent Servers

There are two main classes of servers, iterative and concurrent. An iterative server iterates through each client, handling it one at a time. A concurrent server handles multiple clients at the same time. The simplest technique for a concurrent server is to call the fork function, creating one child process for each client. An alternative technique is to use threads instead (i.e., light-weight processes).

The fork() function

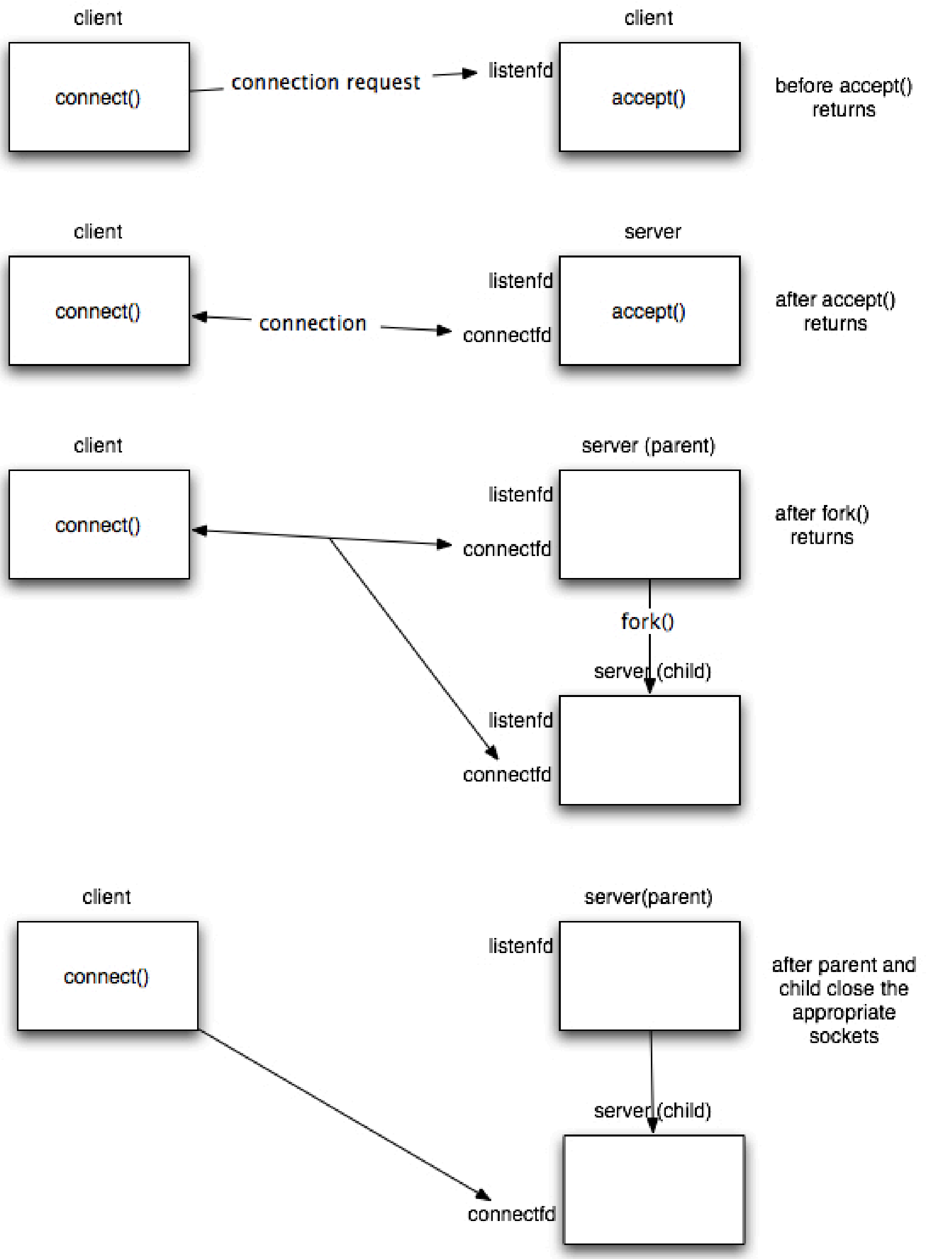
The fork() function is the only way in Unix to create a new process. It is defined as follows:

```
#include <unist.h>
pid_t fork(void);
```

The function returns 0 if in child and the process ID of the child in parent; otherwise, -1 on error.
The function fork() is called once but returns twice. It returns once in the calling process (called the parent) with the process ID of the newly created process (its child). It also returns in the child, with a return value of 0. The return value tells whether the current process is the parent or the child.
Example
A typical concurrent server has the following structure:

```
pid_t pid;
int listenfd, connfd;
listenfd = socket(...);

/***fill the socket address with server’s well known port***/

bind(listenfd, ...);
listen(listenfd, ...);

for ( ; ; ) {

  connfd = accept(listenfd, ...); /* blocking call */

  if ( (pid = fork()) == 0 ) {

    close(listenfd); /* child closes listening socket */

    /***process the request doing something using connfd ***/
    /* ................. */

    close(connfd);
    exit(0);  /* child terminates
    }
    close(connfd);  /*parent closes connected socket*/
  }
  }
```

When a connection is established, accept returns, the server calls fork, and the child process services the client (on the connected socket connfd). The parent process waits for another connection (on the listening socket listenfd. The parent closes the connected socket since the child handles the new client.



Sample Code:

Server Side

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/wait.h>
#include <string.h>


void handle_client(int client_socket);

int main() {

        int server_socket, client_socket;
        struct sockaddr_in server_address, client_address;
        socklen_t client_address_len = sizeof(client_address);
        pid_t pid;

        // Create the server socket
        server_socket = socket(AF_INET, SOCK_STREAM, 0);
        if (server_socket == -1) {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
        }

        // Configure server address
        server_address.sin_family = AF_INET;
        server_address.sin_addr.s_addr = INADDR_ANY;
        server_address.sin_port = htons(3001);

        // Bind the socket to the specified IP and port
        if (bind(server_socket, (struct sockaddr *) &server_address, sizeof(server_address)) == -1) {
        perror("Bind failed");
        exit(EXIT_FAILURE);
        }
```

```
    // Listen for incoming connections
    if (listen(server_socket, 2 ) == -1) {
    perror("Listen failed");
    exit(EXIT_FAILURE);
    }

    printf("Server started. Listening on port %d...\n", 3001);

    while (1) {
    // Accept incoming connection
    client_socket = accept(server_socket, (struct sockaddr *) &client_address, &client_address_len);
    if (client_socket == -1) {
    perror("Accept failed");
    continue;
    }

    // Fork a new process to handle the client
    pid = fork();
    if (pid == -1) {
    perror("Fork failed");
    close(client_socket);
    continue;
    } else if (pid == 0) {  // Child process
    close(server_socket);  // Close the server socket in child process
    handle_client(client_socket);
    close(client_socket);
    exit(EXIT_SUCCESS);
    } else {  // Parent process
    close(client_socket);  // Close the client socket in parent process
    // Clean up terminated child processes to avoid zombie processes
    while (waitpid(-1, NULL, WNOHANG) > 0);
    }
    }

    // Close the server socket
    close(server_socket);

    return 0;
}
```

```
void handle_client(int client_socket) {
        char buf[200];
        int num1, num2, result;
        char operator[2]; // Changed to string to accommodate operator as string

        // Receive the first message (number) from the client
        recv(client_socket, &buf, sizeof(buf), 0);
        num1 = atoi(buf);

        // Receive the third message (number) from the client
        recv(client_socket, &buf, sizeof(buf), 0);
        num2 = atoi(buf);

        result = num1 + num2;

        // Send the result back to the client
        sprintf(buf, "%d", result);
        send(client_socket, buf, sizeof(buf), 0);
}
```

Client Side

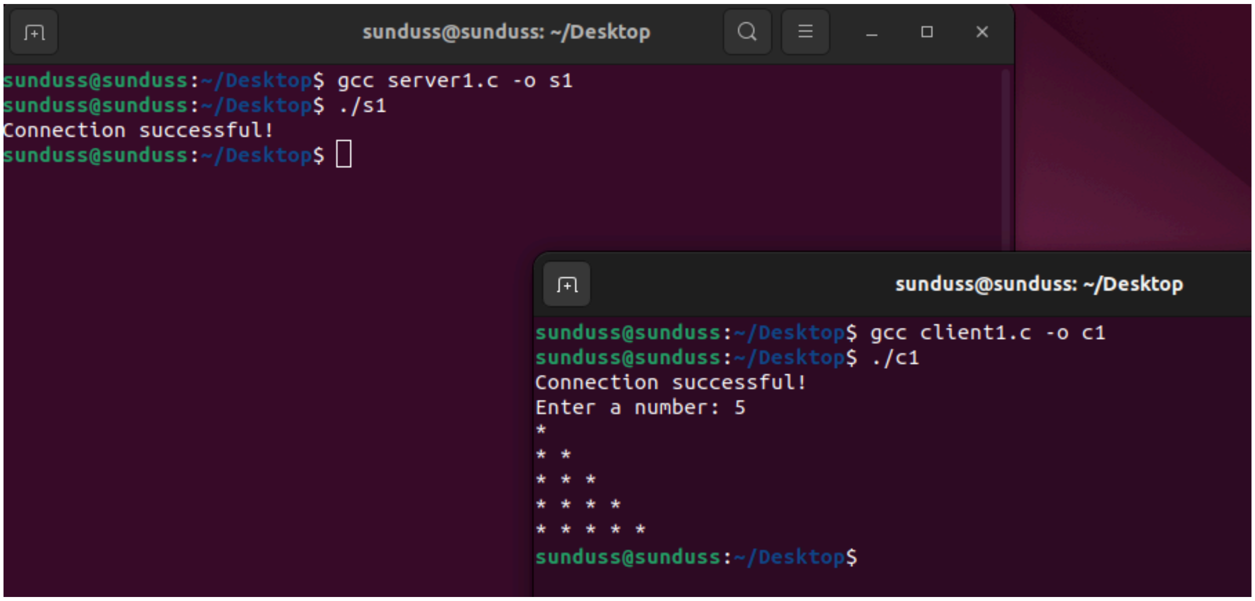
```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main() {
        char request[256];
        char buf[200];

        // create the socket
        int sock;
        sock = socket(AF_INET, SOCK_STREAM, 0);

        // setup an address
```

```
    struct sockaddr_in server_address;
    server_address.sin_family = AF_INET;
    server_address.sin_addr.s_addr = INADDR_ANY;
    server_address.sin_port = htons(3001);

    connect(sock, (struct sockaddr *) &server_address, sizeof(server_address));

 // Send the first message (number) to the server
    printf("Enter a number: ");
    fgets(request, sizeof(request), stdin);
    send(sock, request, sizeof(request), 0);


    // Send the third message (number) to the server
    printf("Enter another number: ");
    fgets(request, sizeof(request), stdin);
    send(sock, request, sizeof(request), 0);

    // Receive the result from the server
    recv(sock, &buf, sizeof(buf), 0);
    printf("\nServer result: %s\n", buf);

    // close the socket
    close(sock);

    return 0;
}
```

How to Run the code?

Open two separate terminals, each accessing the folder you have your code in. Compile both the files using the command : ***gcc filename.c -o exefilenameTobeassigned***

Once compiled without errors, access the server exe file, and then the client exe file. For reference see the image attached below.



```
sunduss@sunduss:~/Desktop$ gcc server1.c -o s1
sunduss@sunduss:~/Desktop$ ./s1
Connection successful!
sunduss@sunduss:~/Desktop$ 
```

```
sunduss@sunduss:~/Desktop$ gcc client1.c -o c1
sunduss@sunduss:~/Desktop$ ./c1
Connection successful!
Enter a number: 5
*
* *
* * *
* * * *
* * * * *
sunduss@sunduss:~/Desktop$
```

Practice tasks

Task 1:

Create a TCP client-server chat program (two-way) to see how the server connects to two clients and handles them simultaneously using fork() system call.

Note: Run your program on multiple systems

Submission Guidelines:

- Rename the code as rollNumber_server1.c for task 1, and the code for the client module as rollNumber_client1.c for task 1.
- Submit the screenshot of the terminal with message transfer along with the source code for both client and server of both of the tasks.