**Course: CSNC 2411**

**Computer Communications and Networks**

**(Lab)**



**Lab 4**

Socket Programming:

TCP Client Server Communication

(TCP Concurrent Server)

Lab Manual 04

|  |
| --- |
| Objectives  * TCP concurrent server * Understanding the fork() call, ppid and pid * TCP concurrent server and client communication |

# Reference Material

**TCP Concurrent Server**

TCP concurrent server is a server process, which can handle multiple clients simultaneously, i.e. multiple clients can talk to a single server at any time.

Previously, we have designed a TCP iterative server, i.e. only one client talks to the server at any time; another client can talk if the previous client closes the connection; e.g. many telnet clients from our lab (each user has a client) can be connected to one telnet server, and all can simultaneously execute their commands.

To design a concurrent server the **fork()** system call of UNIX is used.

**fork() System Call**

fork() is a UNIX function which can create a duplicate of a process called the child process. The details of fork() system call are:-

* 1. takes no arguments
  2. returns an integer value
  3. called once but returns twice
* once in the calling process, called parent process, with an integer value called as ppid (parent process ID)
* and once in the newly created process, called child process, with a value 0.

The child process may call getppid() to get the parent process ID.

Child is an exact copy of the parent process; all descriptors opened in the parent are shared by child and the program execution resumes from the statement after fork().

**Using fork(), an Example**

/\* Demo of simple use of fork() \*/

int main() {

pid\_t pid = fork();

if (pid == -1)

exit (-1);

if (pid == 0)

printf (“\nChild process\n");

else if (pid> 0)

printf (“\nParent process\n");

}

Output: Parent process, Child process

**Parent Process id (ppid)**

* **getppid():** returns the process ID of the parent of the calling process. If the calling process was created by the fork() function and the parent process still exists at the time of calling getppid(), this function returns the process ID of the parent process. Otherwise, this function returns a value of 1, which is the process id for init() process.

**Syntax:**

pid\_t getppid (void);

**Return type**: getppid() returns the process ID of the parent of the current process. It never throws any error, and is therefore always successful.

**Current Process id (pid)**

* **getpid() :** returns the process ID of the calling process. This is often used by routines that generate unique temporary filenames.

**Syntax:**

pid\_t getpid (void);

**Return type:** getpid() returns the process ID of the current process. It never throws any error and is therefore always successful.

**Sample Code for ppid and pid**

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

int main(void)

{

//variable to store calling function's process id

pid\_t process\_id;

//variable to store parent function's process id

pid\_t p\_process\_id;

//getpid() - will return process id of calling function

process\_id = getpid();

//getppid() - will return process id of parent function

p\_process\_id = getppid();

//printing the process ids

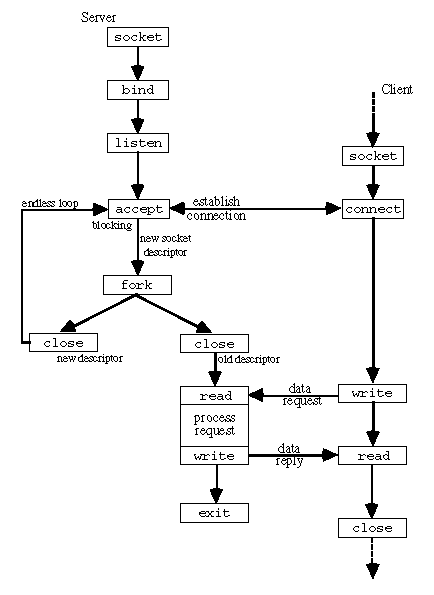
printf("The process id: %d\n", process\_id);

printf("The process id of parent function: %d\n", p\_process\_id);

return 0;

}

**Flow diagram for TCP Concurrent Server & Client communication**

****



**Understanding how Concurrent server works**

listenfd = socket(....);

…../\* other source codes \*/

bind ( listenfd,..... );

listen ( listenfd, qlen );

for ( ; ; ) {

connfd = accept ( listenfd,..... );

if ( ( pid = fork() ) == 0 ) { /\* true for child process \*/

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

/\* process the request with help of connfd \*/

close ( connfd ); /\* done with this client \*/

exit ( 0 ); /\* child terminates \*/

}

close ( connfd ); /\* parent process \*/

}

# Lab Tasks

1. **Your task is to create two client processes and show that both the clients are communicating with the TCP concurrent server at the same time and server is responding to both of them accordingly.**

**[10 Marks]**

**For example**

**Server side:**

**Messages:**

Client with Port 1234: Hello Server

Client with Port 1235: Hello Server

**Client side:**

**Client with Port 1234**

**Message:**

Server with Port 8080: Hello Client

**Client with Port 1235**

**Message:**

Server with Port 8080: Hello Client