Concurrent Servers.

Aims.

The aim of this worksheet is to investigate the use of child processes for servers to enable them to apparently cope with several clients 'at the same time'. Use is made of the system **fork()** and **wait()** functions and signals. The socket communication functions **send()** and **recv()** are used.

Using fork().

The simple server we have produced so far sits looping forever waiting to accept a client request. If several requests come 'at once' the server will deal with the first one and the others are queued while this is done. Having finished with the first client the next is dealt with. If any more client requests arrive these are queued up to the maximum specified in the listening queue. If the queue is full then any extra requests will be 'lost'. If the processing being done by the server takes a long time for each request then some clients may have to wait a long time. If we could have a server for each client request then each client would probably get a better response time! This may be achieved by having the original server produce (**fork()**) a child process to deal with each accepted client connection.

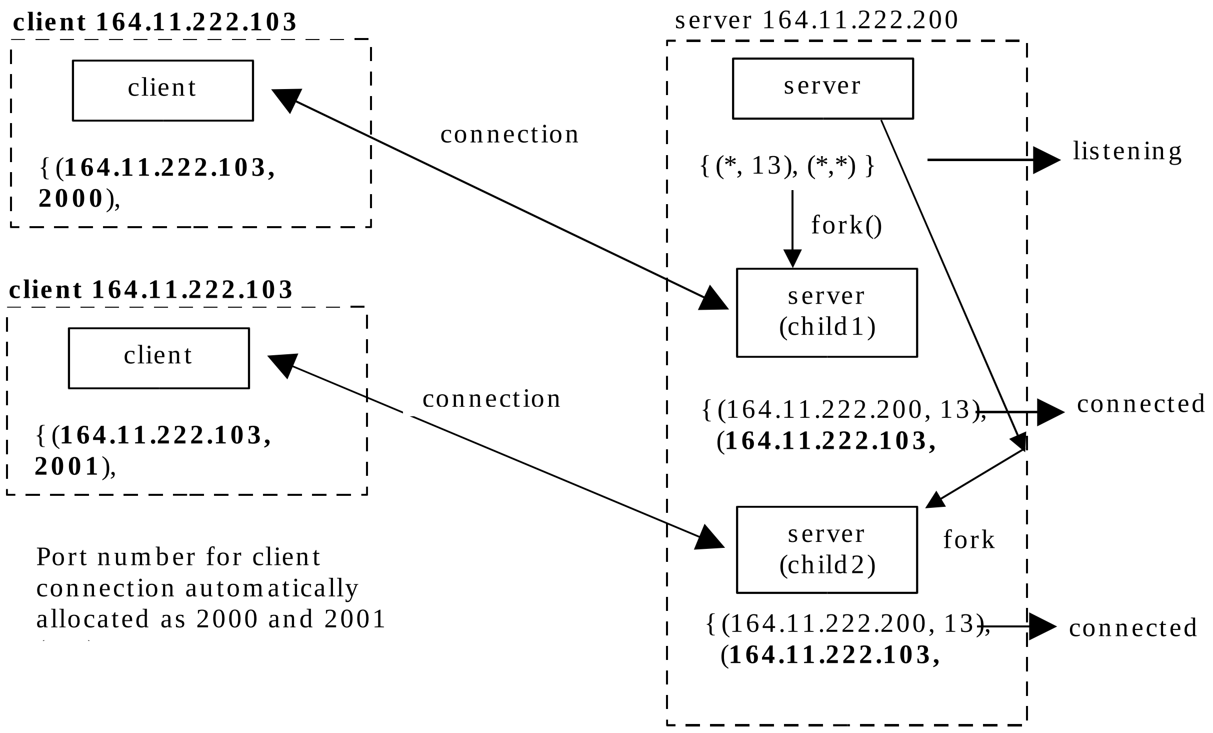
The **fork()** function creates an exact copy of the executing process. It also returns an integer value. In the parent version of the copy this value is the pid of its child. In the child copy the integer value is zero. By checking the returned value we can decide what course of action to follow.

The **accept**() function is used to get a connection request on the listening socket (*listenfd*). If a connection is accepted the **accept()** function supplies a new connection socket (*connfd*) on which to perform the connection activity. If the server **fork()**s after a successful **accept()** the parent server can close the new connection socket (*connfd*) and continue to loop waiting for a new client request on *listenfd*. Meanwhile the child copy can close the listening socket (*listenfd*) and carry out the connection request on the connection socket (*connfd*).

A problem now occurs – How do we identify each of the server processes so that incoming and outgoing messages reach their correct endpoints? The transport layer of our layered system must be able to identify the correct process from the header information supplied with each packet of information (the source port and the destination port number). The port number together with the IP address of the *client* forms a unique pair to identify the child process. If the server host has more than one network interface card then a second pair is needed to identify the server host IP and port number.

Of course all this is done automatically for us. We do not worry about it just make the necessary function calls as we need them!

Assuming that the server host has more than one possible network interface card so that it listens on any IP address, the initial (IP, port) pairs will have wild characters (asterisks, *) to denote that no connection is made. As a diagram we have:



Let us write a server that produces children to interact with any clients.

A fork()ing Server!

Let us create a server that accepts a connection from a client and spawns a child process to receive a message. The child server then simply responds with the greeting *'Hello, World!'*. The server's child remains actively listening on the connected socket until the client sends the message *'quit'*.

In a suitable directory start up **emacs** by

```
emacs serverFork.c &
```

and enter the following server code:

```
/* serverFork.c sends a message to any connected client. */

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

```
#include <arpa/inet.h>
#include <sys/wait.h>
#include <signal.h>

#define MYPORT 3490 /* the port users connect to */
#define BACKLOG 10 /* max no. of pending connections in server queue */
#define MAXDATASIZE 200

void sigchld_handler( int s) {
  while( wait( NULL) > 0);  /* wait for any child to finish */
}

int main( void) {
  int listenfd;                 /* listening socket */
  int connfd;                   /* connection socket */
  struct sockaddr_in server_addr; /* info for my addr i.e. server */
  struct sockaddr_in client_addr; /* client's address info */
  int sin_size;                 /* size of address structure */

  struct sigaction sa;  /* deal with signals from dying children! */
  int yes = 1;
  char clientAddr[ 20]; /* holds ascii dotted quad address */

  if( (listenfd = socket( AF_INET, SOCK_STREAM, 0)) == -1) {
    perror( "Server socket");
    exit( 1);
  }
  /* Set Unix socket level to allow address reuse */
  if( setsockopt( listenfd, SOL_SOCKET, SO_REUSEADDR,
                                  &yes, sizeof( int)) == -1) {
    perror( "Server setsockopt");
    exit( 1);
  }

  sin_size = sizeof( server_addr);
  memset( &server_addr, 0, sin_size);            /* zero struct */
  server_addr.sin_family = AF_INET;          /* host byte order ... */
  server_addr.sin_port = htons( MYPORT); /* . short, network byte order */
  server_addr.sin_addr.s_addr = INADDR_ANY; /* any server IP addr */

  if( bind( listenfd, (struct sockaddr *)&server_addr,
                           sizeof( struct sockaddr)) == -1) {
        perror( "Server bind");
        exit( 1);
  }

  if( listen( listenfd, BACKLOG) == -1) {
    perror( "Server listen");
    exit( 1);
  }
  /* Signal handler stuff */
  sa.sa_handler = sigchld_handler; /* reap all dead processes */
  sigemptyset( &sa.sa_mask);
  sa.sa_flags = SA_RESTART;
  if( sigaction( SIGCHLD, &sa, NULL) == -1) {
    perror( "Server sigaction");
    exit( 1);
  }

  while( 1) {                           /* main accept() loop */
```

```
    sin_size = sizeof( struct sockaddr_in);
    if( (connfd = accept( listenfd,
                (struct sockaddr *)&client_addr, &sin_size)) == -1) {
      perror( "Server accept");
      continue;
    }
    strcpy( clientAddr, inet_ntoa( client_addr.sin_addr));
    printf( "Server: got connection from %s\n", clientAddr);

    if( !fork()) {       /* the child process dealing with a client */
      char msg[ MAXDATASIZE];
      int numbytes;

      close( listenfd); /* child does not need the listener */
      msg[ 0] = '\0';   /* no message yet! */
      do {
        if( (numbytes =recv( connfd, msg, MAXDATASIZE -1, 0)) == -1) {
          perror( "Server recv");
          exit( 1);                    /* error end of child */
        }
        msg[ numbytes] = '\0';               /* end of string */
        fprintf( stderr, "Message received: %s\n", msg);

        if( send( connfd, "Hello, World!\n", 14, 0) == -1) {
          perror( "Server send");
          exit( 1);                   /* error end of child */
        }

      } while( strcmp( msg, "quit") != 0);
      close( connfd);
      exit( 0);                           /* end of child! */
    } /* fork() */
    close( connfd); /* parent does not need the connection socket */
  }  /* while( 1) */
  return 0;
}
```

Compile the server by:

```
gcc –Wall serverFork.c –o serverFork
```

Run the server in the background by:

```
serverFork &
```

Check that it is running by using **ps**.

We can test out this server by using **telnet** and supplying the hostname and portnumber. So, in a different window, type:

```
telnet localhost 3490
```

The server should display a message indicting that a connection has occurred. In the server window check what processes are running by using **ps**. In the telnet session type *'hello'*, the response *'Hello, World!'* should be displayed. Meanwhile the server should display the message it received, i.e. *hello*! Try entering the message *'How are you'* in the telnet session. What happens at the server end? How does this affect the client end? Use *'quit'* to close down the spawned server, Ctrl=] to end the **telnet** session then quit **telnet**.

Note the use of **send()** and **recv()**. These functions are used specifically for TCP socket communication, unlike **write()** and **read()** which may be used with any file and therefore may be used with sockets because all devices etc are treated as files in UNIX.
The **setsockopt()** function is used to enable us to reuse the socket address if the server fails (or more likely killed by us) without having to wait for any garbage collection! Use **man** to get more information.

We will discuss the signal handling later.

A Message Client.

In a suitable directory use **emacs** to create the client:

```
emacs msgclient.c &
```

Enter the following client code:

```
/* msgclient.c
 * needs to be supplied with host name or IP address of server
 * e.g. msgclient localhost or msgclient 127.0.0.1.
 * simply receives a message from the server then dies!
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>         /* for gethostbyname() */

#define PORT 3490       /* server port the client connects to */
#define MAXDATASIZE 100 /* max bytes to be received at once */

int main( int argc, char * argv[]) {
  int sockfd, numbytes;
  char buf[ MAXDATASIZE];
  struct hostent *he;
  struct sockaddr_in their_addr; /* server address info */
  char msg[ MAXDATASIZE];

  if( argc != 2) {
    fprintf( stderr, "usage: client hostname\n");
    exit( 1);
  }
  /* resolve server host name or IP address */
  if( (he = gethostbyname( argv[ 1])) == NULL) { /* host server info */
```

```
    perror( "Client gethostbyname");
    exit( 1);
  }

  if( (sockfd = socket( AF_INET, SOCK_STREAM, 0)) == -1) {
    perror( "Client socket");
    exit( 1);
  }

  memset( &their_addr, 0, sizeof( their_addr));   /* zero all */
  their_addr.sin_family = AF_INET;                /* host byte order .. */
  their_addr.sin_port = htons( PORT);   /* .. short, network byte order */
  their_addr.sin_addr = *((struct in_addr *)he -> h_addr);

  if( connect( sockfd, (struct sockaddr *)&their_addr,
                           sizeof( struct sockaddr)) == -1) {
    perror( "Client connect");
    exit( 1);
  }

  do {
    printf( "Message to send: ");
    scanf( "%s", msg);
    if( (numbytes = send( sockfd, msg, strlen( msg), 0)) == -1) {
      perror( "Client send");
      continue;
    }

    if( (numbytes = recv( sockfd, buf, MAXDATASIZE - 1, 0)) == -1) {
      perror( "Client recv");
      continue;
    }

    buf[ numbytes] = '\0';          /* end of string char */
    printf( "Received: %s\n", buf);
  } while( strcmp( msg, "quit") != 0);

  close( sockfd);
  return 0;
}
```

Compile this in the usual manner:

```
    gcc –Wall msgclient.c –o msgclient
```

Make sure that the server is still running (use **ps**) then open two or three new windows and start a client in each one `msgclient localhost`). After each client connects the server should indicate that a connection has occurred. How many servers appear to be running? Use each client to send some messages. Terminate one of the clients (quit) determine how many servers are now running. Kill the original server and try sending more messages with one of the running clients. What happens? Try starting a new client. What happens? Try starting a new server, etc., etc. When you have finished experimenting quit all the clients and kill the server.

<u>Catching Signals.</u>

When a child process dies it tries to inform its parent what has happened by sending an exit code. Hence the use of *exit(1)* for a child error and *exit(0)* for a normal end of the child. The parent needs to catch the signal and examine the exit code to determine if any action needs to be taken. The **wait()** function is used to accept the death of a child and get its exit code. In our case we just set up a signal handler to respond to the death of a child and use **wait()** to get the exit status but throw it away! To see the effect of not dealing with dying children comment out the signal handler stuff in the server and re-compile it. Test this new version with several clients as before checking the process status of the server (by **ps**) as clients die. What happens when the server itself is eventually killed?

Some alterations to try.

Modify the server so that the child process responds with *'Goodbye, clientaddr'* when the client *'quits'*. Further modify it so that it echoes the message it was sent rather than *'Hello, World!'*.

Change the MAXDATASIZE in the server to just 20 and send messages longer than 20 characters. What happens?

Try any modifications that you can think of to attempt to force error messages to check that error trapping works.

That just about ends our look at TCP sockets next we will experiment with UDP sockets.

References.

Beej's Guide to Network Programming, Brian "Beej" Hall, www.ecst.csuchico.edu/~beej/guide/net.

Unix Network Programming,Vol. 1. 2nd Edn., W. Richard Stevens, Prentice-Hall Pearson Education.

Linux Socket Programming by Example, Warren W. Gay, Que.