...making Linux just a little more fun!

<-- prev | next -->

Issues In Concurrent Server Design on Linux Systems - Part I

By Amit Saha

What is a Concurrent Server

In the simplest terms, a *server* is a process which waits for incoming client connections. The server can be a simple echo server, echoing back what the client sends to it - or it can be a complex program such as a FTP or web server handling thousands of connection requests from clients. A concurrent server is one that can process more than one client connection at the same time.

Implementing a Concurrent Server

Stevens suggests the following choices for the type of process control to use while designing concurrent Linux servers:

- One Fork Per Client Request
- Prefork with child calling accept
- Prefork with file locking to protect accept
- Prefork with thread mutex locking to protect accept
- Prefork with parent passing socket descriptor to child
- Create one thread per client request
- Prethreaded with mutex locking to protect accept
- Prethreaded with main thread calling accept

We shall now take a brief look at each of the above techniques by implementing the ideas to design TCP-based concurrent servers.

A. **TCP Concurrent Server, One Child per Client**

This is by far the simplest method of concurrent server design. The server process forks a child for each client connecting to it. The child forked then takes care of the corresponding client connection while the server waits for other client connections. In this issue, we shall stress on this particular method using it in to design an "enhanced" file transfer application.

B. **TCP Preforked Server, No Locking Around Accept**

This technique of server design uses a technique called *preforking*. The server launches a number of child processes when it starts, and the children service the clients as each client connection arrives. A few things worth noticing here are:

- New clients can be handled without the cost of a fork by the server
- The server has to guess the number of children to prefork when it starts

If the number of clients being serviced at any time exceeds the number of children, the additional clients will experience a service time delay (they will not be ignored as long as the backlog in `listen()` is less than or equal to the additional clients).

The *preforking mechanism* in a preforked server has been reproduced from Stevens for your reference:

```
pid_t child_make(int i,int listenfd,int addrlen)
{
        pid_t pid;
        if((pid=fork())>0)
                rteurn(pid);    //parent
        child_main(i,listenfd,addrlen); //never returns

}

void child_main(int i,int listenfd,int addrlen)
{
        int connfd;
        socklen_t clilen;
        struct sockaddr *cliaddr;
        cliaddr=malloc(addrlen);
        printf("Child %ld starting\n",(long)getpid());
        for(;;)
                {
                clilen=addrlen;
                connfd=accept(listenfd,cliaddr,&clilen);
                /*
                perform
                processing jobs
                */
                }
}
```

A point of interest in this regard is that multiple processes are calling accept on the same listening descriptor. How this is done and other associated issues are going to be dealt in subsequent parts of this series.

C. **TCP Preforked Server, File Locking Around Accept**

The implementation just described works fine with Berkeley-derived kernels. However, System V kernels, which implement `accept()` as a library function may not allow this. The solution lies in placing some kind of a "lock" around the call to accept, so that only one process at a time is blocked in the call to accept. The remaining children will be blocked trying to obtain the lock.

There are various ways to provide this locking mechanism (we will address these in subsequent articles.) The only change to the code previously decsribed in section B is to call the locking function before the children are created, and to release the lock in the `child_main()` function before performing the processing tasks.

D. **TCP Preforked Server, Descriptor Passing**

Another possible modification to the preforking type servers is having only the parent call accept and the "pass" the connected socket to one child. A few points worth noting here are:

- This gets around the possible need of file locking around the call to accept in all the children, but requires a form of descriptor passing to the children
- This complicates the code somewhat as the parent has to keep track of which children are busy and which are free to pass a new socket to a free child

There are two more design mechanisms possible - *TCP Prethreaded Server, per-Thread Accept* and *TCP Prethreaded Server, Main Thread Accept* - where instead of forking new child processes, we use threads to achieve concurrency. The rest is similar to the design mechanism just described.

We shall now introduce ourselves to two famous problems associated with scalability of Linux servers.

The Thundering Herd Problem

Network servers that use TCP/IP to communicate with their clients are rapidly growing in capability. A service may elect to create multiple threads or processes to wait for increasing numbers of concurrent incoming connections. By pre-creating these multiple threads, a network server can handle connections and requests at a faster rate than with a single thread.

In Linux, when multiple threads call accept() on the same TCP socket, they get put on the same wait queue, waiting for an incoming connection to wake them up.

In the Linux 2.2.9 kernel and earlier, when an incoming TCP connection is accepted, the `wake_up_interruptible()` function is invoked to awaken waiting threads. This function walks the socket's wait queue and awakens everybody. All but one of the threads, however, will put themselves back on the wait queue to wait for the next connection. This unnecessary awakening is commonly referred to as a "thundering herd" problem and creates scalability problems for network server applications.

For more about the "Thundering Herd problem", see this document.

The C10K Problem

'C' stands for 'Client' and 'K' stands for 1000. Dan Kegel's contention is that the web is a big place now, and web servers have to be able to handle ten thousand clients simultaneously; the solution to this problem involves finding the optimal configuration, both at Operating System level and network application level, which helps minimize the latency while serving on the order of tens of thousands of clients. In his 'C10K' paper, Dan suggests the following optimizations:

- Serve many clients with each thread, and use nonblocking I/O and level-triggered readiness notification
- Serve many clients with each thread, and use nonblocking I/O and readiness change notification
- Serve many clients with each server thread, and use asynchronous I/O
- serve one client with each server thread, and use blocking I/O
- Build the server code into the kernel

For more details, see the above link.

Now that we've acquainted ourselves briefly with the various design alternatives, we are going to take up the simplest among the design techniques - that is, *TCP Concurrent Server, One Child per client*. Then we are going to see how we can design an "*enhanced*" *file transfer program* using this principle. The other design techniques shall be described with an effective application for each in subsequent issues.

Enhanced Server Design Method I

TCP Concurrent Server, One Child per client

This is by far the simplest method of concurrent server design; the server process forks a child for each client connecting to it. The child then takes care of the corresponding client connection while the server waits for other client connections. The complete source code can be found here. In this example, the server just displays any message sent by the client. Let us now go through a part of the server code where the client connection is accepted:

```
1. while(1){
2. if ((newfd = accept(listener, (struct sockaddr *)&remoteaddr,&addrlen)) == -1)
3.         {
4.       perror("accept");
5.         }
6. else
7.         {
8.      //fork a new child whenever we have a new client connection

9.      pid=fork();
10.     if(pid==0){

        //handle the client

11.     printf("new connection from %s on socket %d\n", inet_ntoa(remoteaddr.sin_addr),newfd);
12.     printf("Recieved %d bytes from the client",read(newfd,buf,sizeof(buf)));
13.     printf("%s",buf);

        //do any other tasks as needed
14.     exit(0);
        }
}
```

Examination of the server code reveals that if a client connection is succesfully accepted (line 6), then the server forks a new child (line 9). The child process then does the other jobs using the client connection (lines 11-13). After this the child exits.

Meanwhile the parent process goes back into the infinite loop (line 1) and waits for incoming client connections (line 2).

Let us now use this server design method to do something useful. First, we'll outline our goal - i.e., building an Enhanced File Transfer Application - clearly:

- **Design a simple File Transfer client & Server Program.** This effectively creates a single connection between the client and the server for each file transfer operation. It can be very easily observed that the file transfer operation consumes more time as the file size increases even if the client and the server is on the same machine.
- **Enhance the application to be less time consuming.** The enhancement method that we implement here is that instead of opening a single connection, we are going

to open multiple connections from the sender to the receiver. From the implementation point of view, we are creating multiple sockets (endpoints). This automatically calls for management issues like server concurrency (our issue here), since we are opening up multiple connections to the server, though from the same client. This will also introduce us to the issue of concurrency in client programs.

In this "simple" file transfer program, the server binds itself to a port and waits for the file that the receiver is sending. The client then connects to the server on the port opened and proceeds with the transmission. For simplicity, only the relevant code snippet is given here; you'll need to add the standard server and client program procedures like filling up the socket data structures, binding your server, etc. to test it if desired.

```
// The client
sock_descr=socket(AF_INET,SOCK_STREAM,6);  //create a TCP socket
c_status=connect(sock_descr,(struct sockaddr*)&remote_addr,sizeof(remote_addr));

if(c_status<0){
        printf("Can't connect");
        exit(0);
}
else {
        fd=fopen(argv[1],"r");   //the file to be transferred is supplied as a command line argument
        if(fd==NULL){
                perror("Error reading file");
        }

        while(!feof(fd){
                ch=fgetc(fd);
                write(sock_descr,&ch,1),sock_descr);
        }
}

// The server

//listener is the descriptor returned by listen()
if ((newfd = accept(listener, (struct sockaddr *)&remoteaddr,&addrlen)) == -1){
        perror("accept");
}
else {
        printf("Recieving file...\n");
        if(read(newfd,&ch,1) <= 0){
                fputc(ch,fp);
        fseek(fp,1,SEEK_CUR);
        }
}
```

The File Transfer Optimization Algorithm

The file transfer method just described suffers from a drawback: the time involved in the whole process increases drastically with file size. We will now enhance the process by introducing a degree of concurrency at the sender as well as the receiver. The terms sender/receiver and client/server, by the way, are used interchangeably in this text. What we seek to do at the sender's end is to logically divide the file using the algorithm mentioned below. The splitting is done at two levels: level 1 is the logical division of the file horizontally into *Num_chunks*. Then, each of the *Num_chunks* is divided into *Num_packets*. The number of simultaneous connections that the client will open up with the server is *num_sockets* which is equal to *Num_packets*. The various parameter values in the following pseudo-code are arbitrary, and can be replaced as appropriate for your application.

```
Check file size, f_size(in MB)
if f_size > 5  then call module SPLIT else proceed with normal("sequential transmission")
        Module SPLIT
                For 5 < f_size < 10
                Level 1 split
                        Num_chunks=[f_size/ sizeof-1-chunk]
                        Sizeof-1-chunk=2mb
                Level 2 split
                        Num_packets=[sizeof-level-1-chunk/ sizeof-1-level2 packet]
                        Sizeof-1-level2 packet=0.25 mb

                For 10 < f_size < 15
                Level 1 split
                        Num_chunks=[f_size/ sizeof-1-chunk]
                        Sizeof-1-chunk=4mb
                Level 2 split
                        Num_packets=[sizeof-level-1-chunk/ sizeof-1-level2 packet]
                        Sizeof-1-level2 packet=0.25 mb

                For 15 < f_size  < 20
                Level 1 split
                        Num_chunks=[f_size/ sizeof-1-chunk]
                        Sizeof-1-chunk=5mb
                Level 2 split
                        Num_packets=[sizeof-level-1-chunk/ sizeof-1-level2 packet]
                        Sizeof-1-level2 packet=0.25 mb

                No of open sockets=Num_packets

        End Module SPLIT

Proceed with transmission
```

Note that we have blissfully ignored the case where the file size may be greater than 20MB. Let's take it one step at a time!

That was the algorithm we intend to implement. This will be covered in the next issue, which will introduce us to concurrent clients and Interprocess Communication (IPC) mechanisms. Till then - happy coding and experimenting!

References:

1. Unix Network Programming(Vol I & II) - W.Richard Stevens
2. Dan Kegel's C10K paper
3. Accept() scalability on Linux, Steve Molloy, CITI - University of Michigan

Talkback: Discuss this article with The Answer Gang

The author is a 3rd year Computer Engineering Undergraduate at Haldia Institute of Technology, Haldia. His interests include Network Protocols, Network Security, Operating systems, and Microprocessors. He is a Linux fan and loves to hack the Linux kernel.

Published in Issue 129 of Linux Gazette, August 2006

<-- prev | next -->
Home FAQ Site Map Mirrors Translations Search Archives Authors Contact Us
Home > August 2006 (#129) > Article