Concurrent Server:

The server can be *iterative*, i.e. it iterates through each client and serves one request at a time. Alternatively, a server can handle multiple clients at the same time in parallel, and this type of a server is called a *concurrent server.*

We have already seen an iterative connection-oriented (TCP-implemented) server in the echo-client, so now we will focus our attention on the *concurrent connection-oriented server*. There are several ways we can implement this server: The simplest technique is to call the Unix **fork()** function. Other techniques are to use threads or to pre-fork a fixed number of children when the function starts.

When the server *receive*s and *accept*s the client's connection, it *fork*s a copy of itself and lets the child handle the client as shown in the figure below:

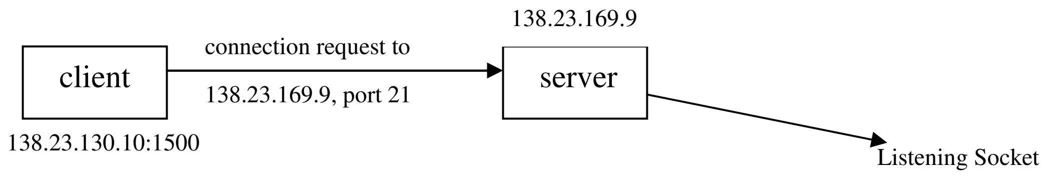

Figure 1: Client with IP address 138.23.130.10 requests to connect to Server (with IP 138.23.169.9 listening from port 21) from its local port number 1500.

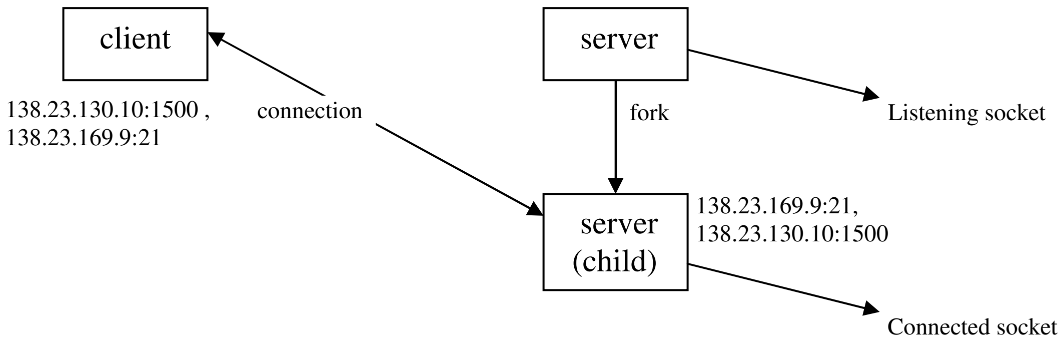

Figure 2: Server fork()s a copy of itself and lets the child handle this client from port 21, but it concurrently keeps listening (by the parent process) from port 21 as well.

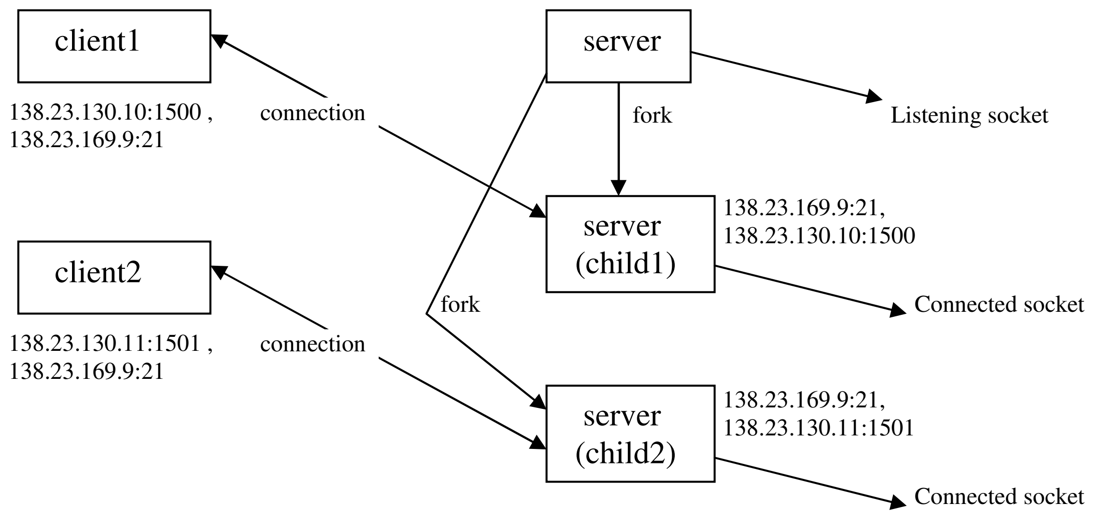

Figure 3: Another client requests to connect to this server. The server fork()s another copy of itself and lets this second child handle this client, whereas it still concurrently keeps listening from port 21 (by the parent process) and keeps serving client1 from port1 (by the first child process).

The listening socket must be distinguished from the connected socket on the server host. Although both sockets use the same local port on the server machine, they are indicated by distinct socket file descriptors, returned server's call of functions socket() and accept() respectively for listening and connected sockets.

Notice from figure 3 above that TCP must look at all four segments in the socket pair to determine which endpoint receives an arriving segment. In this figure, there are 3 sockets with the same local port 21 on the server. If a segment arrives from 138.23.130.10 port 1500 destined for 138.23.169.9 port 21, it is delivered to the first child. If a segment arrives from 138.23.130.11 port 1501 destined for 138.23.169.9 port 21, it is delivered to the second child. All other TCP segments destined for port 21 are delivered to the original listening socket.

Function Description: fork()

The **fork** command <u>creates a new separate process</u> for each client. The fork() command splits the current process into two processes: a parent and a child. The new process (child process) is an <u>almost exact</u> copy of the process that calls it (the parent process).

The fork() command returns 0 when called in the child process, returns the *process ID* of the newly created (child) process when called in the parent process, and –1 on error. Therefore, the return value of the function call to *fork()* tells the process whether it is the parent or the child. For a parent to keep track of its children, it should record the return values from call to *fork()*. (*Note:*

If it is desired to get the process ID of the parent, the child can obtain it by calling **getppid** command.)

From this point one can easily program the child process to serve the client's request while the parent can keep accepting other requests. *However, when a child finishes and exits it needs to notify the parent that it is done.* This is where the ***waitpid()*** command comes to screen.(Please do a man page on this function to learn more about it.) The server needs to harvest ***zombies*** (to be explained later) created via child process termination, which can be performed by calling the waitpid() command.

<u>Some of the attributes the child process **inherits** from the parent process:</u>

- Environment
- Set user ID mode bit
- Set group ID mode bit
- All attached shared libraries
- Process group ID
- Current directory
- Root directory
- File size limit (used in the **ulimit** subroutine)
- Attached shared memory segments (used in the **shmat** subroutine)
- Debugger process ID and multiprocess flag if the parent process has multiprocess debugging enabled (used in the **ptrace** subroutine).

<u>Some of the ways in which the child process differs from the parent process:</u>

- The child process has only one user thread; it is the one that called the **fork** subroutine.
- The child process has a unique process ID.
- The child process ID does not match any active process group ID.
- The child process has a different parent process ID.
- The child process has its own copy of the file descriptors for the parent process. However, each file descriptor of the child process shares a common file pointer with the corresponding file descriptor of the parent process.
- Process locks, text locks, and data locks are not inherited by the child process. (For information about locks, see the **plock** subroutine.)
- Any pending alarms are cleared in the child process.
- The set of signals pending for the child process is initialized to the empty set.
- The child process can have its own copy of the message catalogue for the parent process.
- The set of signals pending for the child process is initialized as an empty set.

<u>zombies:</u>

When a child exits, it sends a signal to the parent to inform it has finished execution. The Linux operating system will <u>not completely get rid of</u> this process when it exits, but keeps important information as to if the child exited normally, how much cpu time was used, etc. The parent is required to clean up after its children and if it does not, these children hang around, they do nothing because they are finished, and so they are called zombies. Zombies take up space in the kernel and are useless and at some point might be dangerous because we might run out of processes.

A parent cleans up after a child process using the **wait()** family of commands.

Format:

#include <sys/wait.h>

```
pid_t wait(int *statlock);
pid_t waitpid(pid_t pid, int *statlock, int options);
```

Both functions above return process ID if OK and 0 or –1 on error.