Concurrent Servers with the sockets API

Support Materials

- On-line man pages: socket(2), socket(7), send(2), recv(2), read(2), write(2), setsockopt(2), fcntl(2), select(2), tcp(7), ip(7).
- Guide to using sockets by Brian "Beej" Hall
- Tcpdump manual
- Chapters 6, 7 y 8 of "Linux Socket Programming" by Sean Walton, Sams Publishing Co. 2001
- Cheat Sheet file within this project

Problem statement

The sockets assignment is divided into the following three parts:

1. Sequential servers (echo client and server, socket options, analysis with tcpdump, file servers)
2. **Concurrent servers (processes, threads).**
3. Input/output (signal handlers, poll, select)

Concurrent Servers

The fundamental difference with respect to sequential servers is that concurrent servers start a new process (using `fork()`) or thread (using `pthread_create()`) to provide a service to incoming connections.

A process has its own memory area. So if I use a process per client, each process will have its own variables with its values. However, threads share the same memory area and resources. The election of one or the other depends on the advantages and disadvantages of each of them, but both of them can be used (for example, in this assignment we use processes, but we could have used threads).

The source code we will use during the entire lab assignment can be cloned from this repo in this way:

```
git  clone https://gitlab.gast.it.uc3m.es/aptel/sockets2_concurrent_servers.git
```

> If you need a quick refresh on Threads and Processes:

> Since in this assignment we are using processes instead of threads, it is key for you to find out how it works. Go to the `processes` folder, compile it (make) and execute `./processes` Look what happens to the variable test 😉

> When it comes to Threads the most important functions are:

- pthread_create() creates a new thread
- pthread_join() makes a thread to wait for other thread completion
- pthread_mutex_lock() (and similar ones) allows two or more threads to synchronize their access to a resource
- pthread_exit() finish a threads execution

> In the `threads` folder you can find a small program to test threads, compile and execute it `./threads` Analyze the code.

One of the strategies used by high-performance servers involves distributing incoming requests among a pool of previously started servers, thus eliminating the time needed to create a thread or process. In general, TCP servers are concurrent, in order to be able to handle several clients simultaneously (and UDP servers are sequential). In the following, we will use the concurrent echo server (`TCPechod`) and the echo client that we compiled in the first part of the sockets assignment, that can be found in the folder `psockets2`.

To compile the file we will proceed in the same way as in the sequential servers assignment (make).

From examination of the code of the concurrent server, it can be observed that after receiving a connection (whereupon the accept function returns), the TCPechod server starts a new process using the fork instruction.

The child process (`fork()==0`) closes its reference to the passive socket descriptor inherited from the parent process (`msock`), and executes the service (`TCPecho()`). The parent process, (`fork()!=0`) closes the socket descriptor ssock returned by `accept()` (to which the client is connected), and executes accept again to wait for new connections.

Remember to clone the code using:

```
git clone https://gitlab.gast.it.uc3m.es/aptel/sockets2_concurrent_servers.git
```

1. Compile:

```
make clean
make
```

Execute the server on port 8xxx:

```
./TCPechod 8xxx
```

Execute the client in another window:

```
./TCPecho <server host> 8xxx
```

and observe their behaviour.

2. Examine the traffic whose origin/destination port is 8xxx with tcpdump, while executing two clients on two different machines.

3. With this version of the server, do you observe the problem that the original sequential server had in section 3 of the first part of the sockets assignment (sequential sockets)? Why?

Use the netstat command to observe the connections that are established and to see their ephemeral states and ports. For example:

```
netstat -tn
netstat -putan
```

The second shows also process information. Both show the TCP connections in numerical format.

4. With this version of the server, do you observe the problem that the original sequential server had in section 5 of the first part of the sockets assignment (sequential sockets)? Why?

Signals

When a process is initiated, the course of its execution can be changed (pause, restart, cancel, etc.) or a parent process can be notified of the finalization of children processes using SIGNALS. Signals allow processes to communicate with each other, and the kernel can also communicate with them (chapters 7 and 10 (pages 225-229) of "Linux Socket Programming", Sean Walton). Signals are denoted by `SIG{NAME}`, where NAME is the name given to the signal

There is a set of signals for which all processes show the same behaviour, and other signals for which different processes will behave differently, and there are even processes that ignore certain signals, because the probability of these signals happening is insignificant. However, there are signals that cannot be ignored, as they are common to all processes. In this assignment, we will see some of the most important (or used) signals.

5. Look at the code of the new server, above, and notice the instruction immediately following the creation of the socket:

```
 (void) signal(SIGCHLD, reaper);
```

and also the function to which it refers:

```
void reaper(int sig){
   int status;
   while (wait3(&status, WNOHANG, (struct rusage *)0) >= 0)
}
```

This is a handler for the `SIGCHILD` signal (sent by the system to indicate to the parent process that one of its child processes has terminated). The signal function registers the handler so that when the `SIGCHILD` signal is received the handler is executed. The handler enables the system to free all the resources that the child process was using. In the case where a parent process does not catch a terminated child process' signal, the child process remains in the zombie state. Inside the reaper function we see the system call wait3 (it is a variant of the wait call), that allows to block the server until the child process dies, but WNOHANG allows to specify that wait3 should not block waiting for a process to finish. In this way, it avoids that a wrong call

blocks the server ("UNIX Network Programming, Volume 1, Second Edition: Networking APIs: Sockets and XTI").

Make the following test: compile two concurrent servers, one with the call to signal and the other without it, and observe the differences between the two versions by examining the process table (with the ps x command) after a client has closed its connection. The ps command shows the processes in execution. The x option allows to show all the processes of the current user.

6. Another very important handler is the SIGPIPE signal, which, in the case of sockets, indicates that the connection has been broken (it receives an RST) . The SIGPIPE handler includes the code needed to treat this exception.

Include the SIGPIPE handler in the echo server, following the example of the SIGCHILD handler. Experiment with this modified code and note the occasions on which the execution of the handler can be observed. Note that:

```
kill -l
```

shows the available signals and

```
kill -signal pid
```

sends the specified signal to the specified process.

You can use either `netstat -putan` or `ps aux` to find the process number (pid).