Distributed Programming I

Solution of laboratory 3

Exercise 3.1 (concurrent TCP server)

Use the functions Socket(), Bind() and Listen(). Then, put the Accept() function call in an infinite loop, and soon after the Accept() call the Fork(). In the 'child section' of the code it is enough to proceed as in exercise 2.3 (infinite loop that reads commands; break the cycle if a QUIT command is received or if using the socket returns some error).
In order to avoid zombie processes, add a signal handler for SIGCHLD.
In order to limit the number of processes to three, a possible solution is the following: let the father process maintain a child counter, that must be incremented in the father branch of the if(Fork()) test. Before performing the Accept() in the infinite loop, test the value of the child counter. If this value equals to three, call the wait() function in blocking mode (the default mode). When the function returns, decrement the child counter by 1.
The described solution, however, has the risk to miss some terminated clients: if the father process is blocked, as example, in an Accept(), and two children terminate when the counter value is 3, the wait() in the previous if reduces the counter by 1 but for the second child no wait() is performed, which may leave zombies in the system.
A possible solution to this problem is to perform, after the children counter test but before the Accept() function call, a loop of non blocking waitpid() (by using the WNOHANG constant) while there are children waiting for a wait(): this can be checked by looking at the waitpid() return value.
This trick allows to solve the problem of correctly updating the active children number, but until the program flow does not newly reach the wait() cycle, terminated children remain active in the system as zombie processes (for example, when the father is blocked in the Accept()). To avoid having zombie children in the system it is necessary to register a SIGCHLD signal handler: this signal is sent to the father every time a process terminates.
In this handler perform the loop of nonblocking waitpid(). It is necessary to perform a loop, and not a single waitpid(), because it is not guaranteed that a single signal will be sent for every terminated children (for example, if the father execution stream is in the signal handler, it has already performed the waitpid() loop, and in the meantime two children terminate, the handler will be newly called, but just once).
Note that handling the SIGCHLD signal in this way makes unnecessary the waitpid() loop before the Accept().
NOTE: it is possible to experience a "race condition" if the SIGCHLD signal is sent when the father has just entered the if that tests the children numbers, but before the blocking wait(), as the wait() would be blocking but the number of children waiting for the wait() would have been already reduced by the signal handler. To avoid this problem, it is possible to adopt one of these two solutions:

1. By using the sigprocmask(), disable the SIGCHLD signal receipt before entering in the if and re-enable it after exiting the if

2 Install the SIGCHLD signal handler just before calling the Accept() and uninstall it soon after. In this case it is necessary, after the if that tests the children number, to perform a while loop with a non-blocking wait(), as in the signal handler, in order to call the wait() for the children that have terminated when the signal handler was not installed.

NOTE: to verify the existence of this “race condition” it is possible to put a sleep() call inside the if branch but before the wait(). Note, however, that the sleep is also interrupted by the SIGCHLD signal, thus it is necessary that the sleep is put in a while loop to effectively obtain the desired waiting time, before calling the wait(). This while loop could exploit the return value of the sleep() function: in fact, it always returns the number of seconds that still needs to pass before reaching the target waiting time it was initially set to. For example:

```
int to = SECONDS_TO_WAIT;
while (to) { to = sleep(to); }
```

Thanks to Davide Colombatto for having signaled the existence of this “race condition”.

Exercise 3.2 (TCP client with multiplexing)

In order to simultaneously handle the user input (standard input from keyboard) and the socket communication from the server it is possible to proceed in this way. After opening the socket and connecting to the server, enter an infinite loop: the first operation to perform in the loop is a Select(), specifying that we want to wait for readiness for reading of two file descriptors: the connected socket and the file descriptor corresponding to the standard input (it corresponds to the int value 0, or it can be obtained as fileno(stdin)). The bits corresponding to the socket and to the file descriptor in the file descriptor set must be set with two FD_SET() calls. The Select() will have NULL as timeout: this means that it will wait until one of the two file descriptors is available for reading.

After the select, two independent if statements will verify, by using FD_ISSET(), if the socket has data available for reading and/or if the standard input has data available for reading. If the socket is ready, process it by reading it. If the standard input is ready, read a line from stdin and parse the command, by executing the proper instruction. In the case of a GET command directly write the command to the socekt (an operation that is always possible).

Note that it is not possible to perform more than one read from the socket in the first if, because there is no guarantee that the second socket read will not be blocking, and this would ruin the multiplexing of the application. In the case of a file read, then, before each read, it is necessary that the execution stream will come back again to the Select() instruction. It is possible to separate the receipt of Server response (“+OK”) from the file by properly using a flag to keep track of where the received data must be stored. Please note that also the read of the server response must be performed in a non blocking way, and it is not guaranteed that the entire “+OK” message is received in a single read. It is then important to maintain updated the reading state of the socket in order to understand if it is necessary to terminate the reading of the command or if the received data already belongs to the file that must be transmitted.

Note, finally, that the user standard input is buffered, thus the Select() returns only when the ENTER key is pressed. Character input is never blocked but, as it happens in C, this is handled just when the ENTER key is pressed.

An alternative solution is to use a separate child to receive files while the father process handles the input. However, in this case, handling the “A” command is more complicated, as the father must interrupt its child. This functionality can be implemented only by sending a signal (kill()) to the child process, to terminate it.

Exercise 3.3 (TCP server with pre-forking)

Use the Socket(), Bind() and Listen() functions, as usual. Perform a loop of 10 iterations, each one performing a Fork(). In the child portion of code put the Accept() function inside an infinite loop, followed by the server command handling. All the Accept() will initially be blocking. As a connection attempt occurs, one of the Accept() will return, while the other ones will continue to be blocked. The O.S. kernel will automatically select one of the blocked Accept() every time a new client atempts to perform a connection.
To terminate the children when the father is terminated, capture the SIGINT signal and, in the signal handling code, perform a kill() for each child, then a series of wait().

Exercise 3.4 (XDR-based file transfer)

Proceed in a way similar to exercise 2.3, but with the following differences.
On the client side, use the specific xdr function generated by rpcgen for encoding the message. The socket can be connected directly to the xdr stream by means of xdrstdio_create. Make sure to free the memory that was allocated by the XDR functions (in particular for the sequence of bytes that correspond to the file contents) and to flush the output by means of fflush().When the XDR structures have been used they have to be freed by means of xdr_destroy().
In the server side it is possible to use an analogous procedure but this means allocating memory for storing the whole file content, which may be too heavy. As this is a server, it is better to adopt a different solution, i.e. to send the file size using xdr_int and then sending the file bytes by means of normal send or write calls. If the number of bytes of the file is not a multiple of 4, it is necessary to add a number of padding bytes in order to make the total number of bytes a multiple of 4. In order to avoid memory leaks it is necessary to free all the memory allocated in the XDR structures, including the file name (use the valgrind program in order to check the absence of memory leaks, as explained in the lab).