A simple concurrent server design

Simplest way to implement a concurrent server is to create a new child process to handle each client

```c
lfd = socket(...);
bind(lfd, ...);
listen(lfd, backlog);
for (;;) {
    cfd = accept(lfd, ...);
    switch (fork()) {
    case -1:
        errExit("fork");
    case 0: /* CHILD */
        close(lfd); /* Not needed in child */
        handleRequest(cfd);
        exit(EXIT_SUCCESS); /* Closes cfd */
    default: /* PARENT */
        break; /* Falls through */
    }
    close(cfd); /* Parent doesn't need cfd */
}
```

Also need a SIGCHLD handler to reap terminated children

Exercises

1. Implement the following server [template: sockets/ex.is_shell_sv.c]:

   ```
is_shell_sv <port>
   ```

   The server creates a socket that listens on the specified port and accepts client requests containing shell commands. (⚠️ Each client sends just one command to the server.) The server concurrently handles clients, executing each client’s command, and passing the results back across the client’s socket.

   Some hints:
   - To keep things simple, the server should obtain the client command by doing a single `read()` (not my `readLine()` function!) with a large buffer, and assume that whatever is read is the complete command.
   - A more sophisticated solution would involve the use of `shutdown(fd, SHUT_WR)` (covered later) in the client, and a loop in the server which reads until end-of-file.
   - Remember that `read()` does not null-terminate the returned buffer!
   - Easy execution of a shell command:
     ```c
     exec1("/bin/sh", "sh", "-c", cmd, (char *) NULL);
     ```
   - To have the command send `stdout` and `stderr` to the socket, use `dup2()`.

   ```c
   ```
Exercises

- Even without writing a client (which is a following exercise), you can test the server using `ncat`:

  ```bash
  $ ncat <host> <port-number> <<< "cmd"
  ```

- The "<<<" syntax (which is specific to `bash` and `zsh`) means take standard input from the following command-line argument.
- For `<host>`, you could use `localhost` (or perhaps `ip6-localhost`).

Once you have a working server, check the following test cases:

1. `while true; do ncat <host> <port> <<< 'false'; done`
   - If we create lots of children, is the server reaping the zombies? (Check the output from `ps axl | grep "defunct"`.)
   - See `sockets/is_echo_sv.c` for an example of a `SIGCHLD` handler and how to install it with `sigaction()`.

2. `ncat <host> <port> <<< 'sleep 1'`
   - Does this cause `accept()` in the server to fail with an error?

3. `ncat <host> <port> <<< 'rubbish'`
   - Does a suitable error message appear for the client?

4. `ncat <host> <port> <<< 'ls nonexistent-file'`
   - Does the error message from `ls` appear for the client?

5. Does your server handle the possibility that `fork()` may fail, by sending a suitable error message back to the client? Test this by running the server from a shell with a reduced process limit, such as:

  ```bash
  $ ulimit -u 1000 # Per-UID process limit of 1000
  $ ./ex.is_shell_sv <port>
  ```

  And then from another shell, attempt to start multiple concurrent clients:

  ```bash
  $ for p in $(seq 1 1000) ; do
    (ncat localhost <port> <<< "sleep 10" &)
  done
  ```

  On the client side, do you see error messages sent by the server?
Exercises

2. Write a client for the preceding server:

```
is_shell_cl <server-host> <server-port> 'shell command'
```

The client connects to the shell server, sends it a single shell command, reads the results sent back across the socket by the server, and displays the results on `stdout`.

[template: sockets/ex.is_shell_cl.c]

3. Write a UDP client and server with the following command-line syntax:

```
id_sysquery_cl <server-host> <server-port> <query>
id_sysquery_sv <server-port>
```

- The client sends a datagram to the server at the specified host and port. The datagram contains the word given in `query`, which should be either of the strings “uptime” or “version”. The client waits for the server to send a datagram in response, and prints the contents of that datagram on standard output.

- The server binds its socket to the specified port and receives datagrams from clients, and, depending on the content of the datagram, constructs a datagram containing the contents of either `/proc/uptime` or `/proc/version`, which it sends back to the client. If the client sends a datagram containing an unexpected word, the server should send back a datagram containing a suitable error message.