Project #1
Command Server

Assigned: Feb 12, 2024 Document version: 1.0
Due date: Feb 28, 2024

- This project will be done in groups of two students. You can do it individually as well. You will program in C/Linux. Programs will be tested in Ubuntu 22.04 Linux 64-bit.
- **Objectives/keywords**: Practice with process creation, program execution, IPC, signals, fork(), exec(), low-level file I/O, message queue, named and unnamed pipes, experimantation.

---

1. Project Specification (75 pts)

In this project, you will implement a command server program and a respective client program. The server will execute the commands (programs) whose names are submitted by a client and return the results (outputs) to the client. The server should be able to communicate with more than one client concurrently. There will be a message queue through which clients will be able to send connection requests to the server and get connected. Named pipes will be used between a client and the server to send commands and receive results. The server will use a separate child process to serve a client. Hence, if there are N clients connected, there will be N child servers running concurrently.

Between a client and a server child process, two **named pipes** will be used; one for sending data from the client to the child server (**cs pipe**), the other for sending data from child server to the client (**sc pipe**). A client will send a command line over the cs pipe. The server child will send the result over the sc pipe. These named pipes will be created by the client, by using the `mkfifo()` system call and their names will be sent to the server when the client connects to the server. A client can interact with a user to get command lines, or it can get the command lines from a file.

The server program will be named as `comserver` and will have the following parameters.

```
comserver MQNAME
```

The `MQNAME` parameter will be used to specify a name for the message queue that will be used by the server to receive the incoming connection requests from the clients.

The client program will be named as `comcli` and will have the following parameters.

```
comcli MQNAME [-b COMFILE] [-s WSIZE]
```

The `MQNAME` argument is used to specify the name of the message queue on which the server is waiting for connection requests. By default, the client program is an *interactive* client. It will interact with a user and will receive the commands from the user. If `-b COMFILE` option is specified, however, the client will be a *batch* client; it will receive the commands from an input ascii text file. The name of the file is the value of the `COMFILE` argument. The `-s WSIZE` option will be used to specify the maximum number of bytes to write to an sc pipe with a single `write()` system call (on the server side).

When started, **the server** will create a message queue (via `mq_open()` call with `O_CREAT` flag) and will wait on it (by calling `mq_receive()`) to receive a connection request from a client. When such a request is received, the server will parse the incoming connection request message and will obtain the ID of the client, the names for the two named pipes created by the client, and the `WSIZE` value. Then the server will create a child process (server child) with the `fork()` system call. The server child will handle the client. Then the server (main server process) will loop and wait on the message queue for another connection request. It will continue operating like this.

A server child process that is created to handle a client will first open the respective named pipes. Then it will send a response message over the sc pipe to the client that the connection is successfully established. Then it will wait on the cs pipe to receive a command line. When a command line arrives, the server child will parse it and execute it. To execute the command line, the server child will create one or two child processes. The execution result of a command line will be first written to a file. Then the server child will send the content of the file back to the client over the sc pipe. Then the server child will loop and will wait on the cs pipe to receive and process another command line. The server child will continue running like this until it receives a **quit request** from the respective client over the cs pipe. Upon such a request, the server child will send a quit-ack message back to the client and will terminate.

When started, **a client** will first create two named pipes and then will send a **connection request** message to the server over the message queue. The names of

the pipes, an ID for the client, and the maximum message size (`WSIZE`) will be included in the message. Then, the client will wait on the sc pipe for a response from the server. When it receives a response that indicates the connection is accepted and established, the client will be ready to take command lines from a user. It will ask the user to enter a command line. After getting the command line from the user, it will send it to the server child by using the cs pipe. Then it will wait on the sc pipe for the result. When the result arrives, the client will print the result to the screen and will wait for another user command line. The client will loop and continue operating like this until the user types **quit** command.

If the client is a batch client (invoked with `-b` option), then it will read one command line from the `COMFILE` and send it to the server child over the cs pipe and will wait on the sc pipe for the result. After getting and printing the result, the child will loop and read another command line from the command file. The client will continue operating like this until it comes to the end of the command file.

When the user types the quit command (in interactive mode) or when the end of `COMFILE` is reached (in batch mode), the client will send a quit-request message to the server child. When it gets the quit-ack message back, it will remove the named pipes and will terminate.

A **command line** can include a single command name or two command names that are piped (combined) together with a | sign (compound command). A command name can include zero or more arguments separated by white space characters (SPACE or TAB). The number of arguments, including the command name, can be at most `MAXARGS`. Some command line examples are: ”ls”, ”ps aux”, ”ls -al”, ”sort”, ”wc”, ”cat afile.txt”. A command line that includes two command names can also include arguments for each command after the command name. In a compound command the | sign indicates that the output of first command will be the input of the second command. Example compound command lines are: ”ps | wc”, ”ps aux | sort”, ”cat afile.txt | wc”.

A child server, when it receives a command line from a client, will parse the command line first and will then create a **temporary output file** to store the result of the command execution. Then, if the command line includes a single command name (program name), the child server will create a child process with the `fork()` system call to execute the related command (program). We will call this created child process as a **runner child** process. Then the child server will wait until the runner child runs and terminates. For that purpose, the child server can use the `wait()` or `waitpid()` system call. The runner child will first redirect its standard output to go to the output file, instead of the screen. It can use the `dup()`, or `dup2()` call for this purpose. This is because we want the result of a command execution at the server not to go to the screen, but to go to a file from where it can

be sent to the client. After redirecting the standard output, the runner child will execute the command (program) by invoking the `exec()` system call. When the command (program) is executed, the output (i.e., result) goes to the output file specified. When the execution finishes, the running child will terminate. This will cause the server child to return from its wait call. Then the server child will access the output file and will send the content of it to the client via the sc pipe.

If the command line includes two commands with a | symbol in between, the child server will first create an **unnamed pipe** via the `pipe()` call. Then it will create two runner child processes by issuing two fork() calls. The runner processes will have the file descriptors of the unnamed pipe. Then, the first runner process, that will execute the first command (the command before the | symbol), will first redirect the standard output of itself to go to the pipe, and will execute the command by invoking the `exec()` system call. The second runner process will redirect its standard input to read from the pipe (not from the keyboard/console), and will redirect its standard output to go to the output file. After redirection, the second runner child will execute the second command by invoking the `exec` system call. In this way, the compound command will be executed and the result (output) will go the output file. Meanwhile the server child will wait for the termination of these two runner processes by using the `wait()` or `waitpid()` calls. When the runner child processes finish execution (terminate), the server child will return from waiting and will be able to continue. It will read the content of the output file and will send it to the client via the sc pipe. After that, the server child will delete the output file and will wait for another command line to arrive from the cs pipe.

A client should also support a **quitall** command. When the user types this command, a **quitall request** message will be sent to the server over the cs pipe. Then the server will terminate all child processes (if any) and cleanup the message queue. It will also cause the termination of all client processes. A process can request the termination of another process by sending a signal to it. **A signal** can be sent by invoking the `kill()` system call. To get terminated upon a signal, a process (client or server child) should implement a signal handler function and should register it with the operating system kernel by using the `signal()` system call. The signal handler function needs to call the `exit()` system call to terminate the process. After such a termination, all pipes and the message queue should have been removed, so that we can have a clean system for the next run of the client-server program.

The project requires **messages** to be exchanged between a client and server process. There are different kinds of messages that can be exchanged, such as connection request message, command result message, etc. We will have the following format for all types of messages:

[Length (4 bytes), Type (1 byte), Padding (3 bytes) bytes, Data (0 or more bytes)].

That means, a byte-stream corresponding to a message will have the first 4 bytes (bytes 0, 1, 2, 3) indicating total message size (length). At the receiver side, this length can be used to collect the incoming bytes from a pipe to form a message and detect the end of the message. The length will be encoded using little-ending format: the byte 0 (in the message) is the least significant byte of the length (a 32-bit integer), the byte 3 is the most significant byte. Then the next byte (byte 4) will indicate the type of the message. Then we will have 3 bytes of padding (empty bytes - unused area) (bytes 5, 6, 7). Then we will have 0 or more bytes of data. You can consider the length, type and padding fields as the message header; the remaining part as the message data. We should have at least the following message types:

- Connection request message (`CONREQUEST`).
- Connection reply message (`CONREPLY`).
- Command line message (`COMLINE`). This message will include a command line.
- Command result (output data) message (`COMRESULT`). The message will include the result (output) of a command line execution at the server. The data part of the message can be 0 or more bytes. For some commands, such as "ps", the result can be small; for some commands, such as "cat afile.txt", the result can be quite large number of bytes. Note that a `write()` operation to an sc pipe will send this message in units of at most `WSIZE` bytes.
- Quit request message (`QUITREQ`)
- Quit response (ack) message (`QUITREPLY`)
- Quitall request message (`QUITALL`).

2. Experiments and Report (25 pts)

Run your program with different number of clients. Measure the time elapsed. Run also with different values of `WSIZE`. Put your results into tables (or plots). Are the results different? What is the tendency? Try to explain the results and the reasons. Put all your experimental data, analysis and interpretations into a report file and upload a `report.pdf` file in your submission.

3. Submission

Put all your files into a directory named with your Student ID. If the project is done as a group, the IDs of both students will be written, separated by a dash '–'. In a `README.txt` file, write your name, ID, etc. (if done as a group, all names and IDs will be included). The set of files in the directory will include

`README.txt`, `Makefile`, and program source file(s). We should be able to compile your program(s) by just typing `make`. No binary files (executable files or .o files) will be included in your submission. Then **tar** and **gzip** the directory, and submit it to **Moodle**.

For example, a project group with student IDs `21404312` and `214052104` will create a directory named `21404312-214052104` and will put their files there. Then, they will tar the directory (package the directory) as follows:

```
tar cvf 21404312-214052104.tar 21404312-214052104
```

Then they will gzip (compress) the tar file as follows:

```
gzip 21404312-214052104.tar
```

In this way they will obtain a file called `21404312-214052104.tar.gz`. Then they will upload this file to **Moodle**. For a project done individually, just the ID of the student will be used as a file or directory name.

4. Tips and Clarifications

- In this project, the client and server programs are to run on the same machine. Therefore we can use pipes and message queues. With TCP/IP network socket programming, we could do the communication via sockets. Then the client and server programs could run in different machines. The body of the client program and server program that you developed in this project, however, could stay as it is for the most part, except the communication part.
- **Tips and clarifications** about the project will be added to a *web-page* that will be linked from course's Moodle page (under project specification document). Please check this project web-page regularly to follow the clarifications. Further *useful information*, *tips*, and *explanations* that will be put on this page will help you for the developing the project. It will be a dynamic, living page, that may be updated (based on your questions as well) until the project deadline.