Operating Systems (0368-2162)

Printable Characters Counting Server Assignment

Due date (via moodle): **July 10th, 2025, 23:59**

Individual work policy

The work you submit in this course is required to be the result of your individual effort only. You may discuss concepts and ideas with others, but **you must program individually**. **You should never observe another student's code**, from this or previous semesters.

Students violating this policy will receive a **course grade of 250** ("did not complete course requirements").

1 Introduction

The goal of this assignment is to gain experience with sockets and network programming. In this assignment, you will implement a toy client/server architecture: a printable characters counting (PCC) server. Clients connect to the server and send it a stream of bytes. The server counts how many of the bytes are printable and returns that number to the client. The server also maintains overall statistics on the number of printable characters it has received from all clients. When the server terminates, it prints these statistics to standard output.

2 Assignment description

You need to implement two programs, a server and a client:

1. **Server** (`pcc_server`): The server accepts TCP connections from clients. A client that connects sends the server a stream of N bytes (N is determined by the client and is not a global constant). The server counts the number of printable characters in the stream (a *printable character* is a byte b whose value is $32 \leq b \leq 126$). Once the stream ends, the server sends the count back to the client over the same connection. In addition, the server maintains a data structure in which it counts the number of times each printable character was observed in all the connections. When the server receives a `SIGINT`, it prints these counts and exits.

2. **Client** (`pcc_client`): The client creates a TCP connection to the server and sends it the contents of a user-supplied file. The client then reads back the count of printable characters from the server, prints it, and exits.

2.1 Client specification

Implement the following program in a file named `pcc_client.c`. The following details the specification of the program.

Command line arguments:

- `argv[1]`: server's IP address (assume a valid IP address).
- `argv[2]`: server's port (assume a 16-bit unsigned integer).
- `argv[3]`: path of the file to send.

You should validate that the correct number of command line arguments is passed, and detect errors while opening the file (e.g., if it doesn't exist) and reading it.

The flow:

1. Open the specified file for reading.
2. Create a TCP connection to the specified server port on the specified server IP.
3. Transfer the contents of the file to the server over the TCP connection and receive the count of printable characters computed by the server, using the following protocol:
   (a) The client sends the server N, the number of bytes that will be transferred (i.e., the file size). The value N is a 32-bit unsigned integer in **network byte order**.
   (b) The client sends the server N bytes (the file's content).
   (c) The server sends the client C, the number of printable characters. The value C is a 32-bit unsigned integer in **network byte order**.
4. Print the number of printable characters obtained to `stdout` using **exactly** the following `printf()` format string:
   `"# of printable characters: %u\n"`
5. Exit with exit code 0.

Guidelines:

- Use the `inet_pton()` function for converting a string containing an IP address to binary representation.
- Use only system calls to access files. Don't use C library functions such as `fopen()`.
- You can assume that the size of the file can be represented with a 32-bit unsigned integer.
- Do **not** assume that you can allocate a buffer whose size is equal to the file size, because the file size might be huge. Allocations of up to 1 MB are OK.
- On any error condition, print an error message to `stderr` containing the `errno` string (i.e., with `perror()` or `strerror()`) and exit with exit code 1.
- There's no need to clean up file descriptors or free memory when exiting.

2.2 Server specification

Implement the following program in a file named `pcc_server.c`. The following details the specification of the program.

Command line arguments:

- `argv[1]`: server's port (assume a 16-bit unsigned integer).

You should validate that the correct number of command line arguments is passed.

The flow:

1. Initialize a data structure `pcc_total` that will count how many times each printable character was observed in all client connections. The counts are 32-bits unsigned integers.
2. Listen to incoming TCP connections on the specified server port. Use a `listen()` queue of size 10.
3. Enter a loop, in which each iteration:
   (a) Accepts a TCP connection.
   (b) When a connection is accepted, reads a stream of bytes from the client, computes its printable character count and writes the result to the client over the TCP connection (using the protocol described above). After sending the result to the client, updates the `pcc_total` global data structure. You don't need to handle overflow of the `pcc_total` counters.
4. If the server receives a `SIGINT` signal (for example, user hits Ctrl-C) perform the following actions:
   (a) If the server is processing a client when `SIGINT` is delivered, finish handling this client (including updating the `pcc_total` global data structure).
   (b) For every printable character, print the number of times it was observed to standard output. Use **exactly** the following `printf()` format string to print the count of each character:
   `"char '%c' : %u times\n"`
   - Only print characters that appear at least once; do not print characters with zero occurrences.
   - Print the characters in ascending ASCII order.
   (c) Exit with exit code 0.
   (d) Handling of `SIGINT` must be **atomic** with respect to processing of client requests. We define "processing a client" as the period of time starting when `accept()` returns the client's socket and until closing its socket. Therefore:
   - If `SIGINT` is delivered when no client is being processed, **no new client connection may be `accept()`ed and processed**.
   - If `SIGINT` is delivered while a client is being processed, that client connection must be processed to completion.
   - There's no requirement for `accept()`ing and/or handling pending connections in the `listen` queue after a `SIGINT`. For example, if `SIGINT` is delivered while a client is being processed, the program should print its statistics and exit after finishing processing that client, and not worry about any pending connections that have not been `accept()`ed yet.

Guidelines:

- You define the data structures required to implement the above specification. All that is required is for the program to behave as specified above.
- Use `bind()` to the address `INADDR_ANY` to accept connections on all network interface. See the `ip(7)` manual page for more details.
- Use the `SO_REUSEADDR` socket option to enable reusing the port quickly after the server terminates. See the `socket(7)` manual page for more details. (To see why, try omitting this option and then killing and quickly restarting the server with the same port argument.)
- Do **not** assume that you can allocate a buffer whose size is equal to the file size, because the file size might be huge. Allocations of up to 1 MB are OK.
- There's no need to clean up file descriptors or free memory when exiting.

Error handling:

1. Client connections may terminate unexpectedly due to TCP errors. You can assume that a TCP error occurs if and only if a system call sending/receiving data to/from the client returns an error with `errno` being one of `ETIMEDOUT`, `ECONNRESET`, or `EPIPE`. Client connections may also terminate if the client process is killed unexpectedly; this case is indicated by a system call receiving data from the client returning 0 (i.e., EOF). If a client connection terminates due to such TCP errors or unexpected connection close:
    - **Do not** exit the server. Just print an error message to `stderr` and keep accepting new client connections.
    - The `pcc_global` statistics **must not** reflect the data received on the (terminated) connection.
2. If a system call fails as part of the program's design (for example, `EINTR` after receiving a signal), you do not have to treat it as an error that requires exiting the program (since it's part of your intended flow).
3. On any other error condition, print an error message to `stderr` containing the `errno` string (i.e., with `perror()` or `strerror()`) and exit with exit code 1.

3 Relevant system calls and functions

1. Learn about and use the following: `socket()`, `connect()`, `bind()`, `listen()`, `accept()`, `htonl()`, `ntohl()`, `htons()`, `ntohs()`, and `setsockopt()`.
2. Read the manual page `ip(7)` for more information.
3. Read the manual page `signal(7)` and the `sigaction()` documentation.

4 Useful information & tips

- `/dev/urandom` is a pseudo device file that returns random bytes. You can `read()` from it repeatedly and keep getting random bytes forever. You can use this device to generate files with non-printable characters, for testing.

- The IP address `127.0.0.1` specifies the local host (it is called a loopback address). If you don't know the IP address of your machine, or are working on a VM without an Internet connection, you can still connect to `127.0.0.1`.
- Notice that TCP ports less than 1024 are *reserved* and cannot be used by non-root processes; you will get an error if the server tries to `bind()` to such a port.
- You can use the `nc` (netcat) program to simulate a server or client.
- You can use the `ngrep` program to monitor the traffic on a specific port. (You will first need to run `sudo apt install ngrep` to install this program.) Running `ngrep` requires root permissions, so you'll need to run it using `sudo`.

5 Submission instructions

1. Submit two files, `pcc_server.c` and `pcc_client.c`. (Don't submit them in one ZIP file!)
2. Document your code with explanations for every non-trivial part of your code. Help the grader understand your solution and the flow of your code.
3. The programs must compile cleanly on the course VM (no errors or warnings) when the following command is run in a directory containing the relevant source code file:

```
gcc -O3 -D_POSIX_C_SOURCE=200809 -Wall -std=c11 pcc_server.c (or pcc_client.c)
```