Fundamentos de Sistemas de Operação

MIEI 2016/2017

Homework Assignment 2 (V 2.0)

Overview

Client / Server (multithreaded) interaction via datagram sockets (UDP).

Deadline and Delivery

This assignment is to be conducted individually by each student. The code should be submitted via the Mooshak system using the individual account of the student. **The deadline is October 31st 2016, Friday, until 17h00 (We have pushed the date a few days).**

Description

The client/server architecture is a very common interaction mechanism in the Internet for providing a particular service (e.g. file transfer, access to web pages, remote execution in the Cloud, etc). The architecture's components include a server process (usually concurrent i.e, multi-threaded) running in a well-known location, and multiple clients whose requests are served by the (concurrent) server. The goal of this assignment is the familiarisation with this architecture and to exercise interprocess communication via datagram sockets (*UDP*) and concurrent programming via *pthreads*.

In this assignment it is required that you *implement a simple Server* that exposes to clients a set of pre-defined operations (already available in a library - *server_functions* - that you should link to your code) that may either be completely independent from each other or may share some resources. *The operations are to be executed concurrently* (i.e. each one in the context of a new thread) and upon a client's request. The client's requests are synchronous meaning that a client submits a request to an operation's execution to the server and waits for the reply.

The set of available functions are:

- `void functionA(int a, char *returnValue)`
  `This function receives as input an integer and produces as result a string in the parameter "returnValue".`
- `void functionB(int a, int b, char *returnValue)`
  `This function receives as input two integers, a and b, and produces as result a string in the parameter "returnValue".`
- `void functionC(char *a, char * returnValue)`
  `This function receives as input a string in parameter "a" and produces as result a string in the parameter "returnValue".`

Moreover, the following rules (pre-conditions) have to be satisfied concerning the above functions:

- `Multiple instances of functionB() cannot execute at the same time since they share the same resource named resourceB that is a critical region (e.g. incrementing a global counter).`
- `There are no execution restrictions for functionA() and functionC() meaning that several instances of either functionA() or functionC() may be run concurrently among them and concurrently with instances of functionB().`

A client communicates with the server program with the application protocol described below.

- A client sends a string to the server with

```
Operation_name argument1 [argument2]
```

In the case of the above three functions, we have
- `FunctionA number`
- `FunctionB number number`
- `FunctionC string`

- Upon receiving and processing a particular request, the server sends a string to the client with the result of the operation
  - `result`

The client is already implemented on the file *client.c* and receives as arguments the *port* where the server is waiting for incoming requests. It is assumed that both the clients and server are located on the same machine (address "localhost").

The implementation of the client has the following steps:

1) Creates a UDP socket bound to some port number, other than the server's.
2) Creates a variable of type struct *sockaddr_in* where the hostname is *"localhost"* and the port number is *port*
3) Executes an interaction loop with the server containing the following steps:
  a) Reads a line with the name of the function and its arguments
  b) Sends an ASCII message with this request to the server
  c) Waits for the reply
  d) Writes the result in the terminal
  e) Goes back to point 3a.

Your implementation of the server program has to include a function `void server( int port )` that receives and concurrently executes requests from several clients. This function performs the following actions (We already provide a skeleton of the server code with some useful functions and definitions):

1) Creates a UDP socket bound to `port_number`
2) Receives an ASCII message with the name of the function (mapping to one of the above) and the necessary arguments
3) Check if the operation's name is valid
  - If the name is not valid returns the string "Invalid operation"
4) If the action name is valid:
  a) The operation is executed concurrently using the *pthreads* libarary and obeying the execution requirements defined above
  b) The reply is sent to the client
5) Waits for a next client's request upon which actions from 2) forward will be repeated.

An initial skeleton of the program is given in file `server.c`.

Consider the code include in archive *tpc2.zip* available through the CLIP system. After completing the file `server.c` compile and launch it as follows:

Start by compiling the support libraries:

```
gcc –Wall –c udp_comm.c
gcc –Wall –c server_functions.c
```

Them compile the server and launch it:

```
gcc -Wall –o server server.c udp_comm.o server_functions.o
./server port_number
```

Compile the file `client.c` and launch it as follows:

```
gcc -Wall -o client client.c udp_comm.o server_functions.o
./client port_number
```

The file `udp_comm.c` comprises the implementation of the aforementioned UDP communication functions. Please refer to the slides of the lectures of 14th and 16th of October available in CLIP; see also chapter 47 of OSTEP book. The pthreads library is described in chapters 26, 27, and 28 of the OSTEP book. The file `server_functions.c` contains the definitions of the functions to be executed by the server.

Execution example:

Please find below an execution example of the provided client. In bold you can find the output of the client given the commands provided to it. Below you can also find some notes regarding the behaviour of the server given these commands

FunctionA 1
Function A executed with Sucess for 1 seconds.
FunctionB 45 23
Function B executed with Sucess. Result: 1035.
FunctionC ThIs Is A STRinG WiTH a MIX oF CAps and NON-CaPS ChARS!
Function C executed with Sucess. Result: this is a string with a mix of caps and non-caps chars!
.

Notice that all function identifiers start with a capital F. Also for the argument of FunctionC, everything in front of the function identifier is part of the argument to be passed to the function. You should not remove the '\n' from the line (that is what that dot appears in the final line of the output). In more detail the server upon receiving the last request should receive as an argument a string with the text: "ThIs Is A STRinG WiTH a MIX oF CAps and NON-CaPS ChARS!\n".

Additional Hints

- You might need to pass more than one argument to each thread in the server. We already provide a structure definition that you can use for this purpose (`struct requestInformation`). More than that, we already provide handy functions to create and free such structures. All of this can be find in the file `server.c`
- A new thread should handle each client request. Notice that you do not know how many client requests the server is going to receive.
- The server should not wait for a request to be processed before starting to process a new incoming request.
- You can safely assume that all requests and replies sent between client and server have a maximum size of 1024 bytes.

- You might want to look into the method `pthread_detach.` For your convenience we replicate part of the manual page below:

> **NAME**
> `pthread_detach` -- detach a thread
>
> **SYNOPSIS**
> ```
> #include <pthread.h>
>
> int
> pthread_detach(pthread_t thread);
> ```
>
> **DESCRIPTION**
> The `pthread_detach()` function is used to indicate to the implementation that storage for the thread thread can be reclaimed when the thread terminates. If thread has not terminated, `pthread_detach()` will not cause it to terminate. The effect of multiple `pthread_detach()` calls on the same target thread is unspecified.

- Another method that might be useful to use in conjunction with the previous one is the `pthread_self`. In a nutshell, this method allows a thread to obtain its own `pthread_t` identifier.