Simple Remote Procedure Call Service

Check My Courses for Due Date

High-Level Description

Consider a simple calculator. It has a read-eval-print-loop (REPL), where you enter something like the following.

```
>> add 6 10
16
>> multiply 4 5
20
>>
```

The calculator is structured as two programs: frontend and backend. The frontend does not do the actual calculations. It just prints the prompt, get the input from the user, parses the input into a message. The frontend passes the message to the backend that does the actual calculations and passes the result back to the frontend, where it is printed to the user. We write two programs `backend.c` and `frontend.c`. The backend has some functions it wants to expose for the frontend. The frontend needs to call those functions that are exposed by the backend. The remote procedure call service (RPCServ) is responsible for linking the two. You are expected to develop such a RPCServ as part of this assignment. We start with a very simple implementation. The frontend issues an execution command and backend runs it and returns the result. This simple structure would work even with multiple frontends if the commands issued by the frontends can be executed very quickly. If some commands can hold the backend for a long time (like seconds), we have a problem. When a frontend is running a long command, other frontends will find the backend unavailable. That is, you run sleep 5 in the frontend and the backend is held by that frontend for 5 seconds. If other frontends try to run a calculation, they will not get any results. To solve this problem, you will use multi-processing. That is the backend will create a serving process for each frontend. In this configuration, as soon as the frontend connects to the backend, we create a new serving process that is dedicated to the frontend and let it serve the frontend's requests. Even if the frontend does a sleep 5, it would not cause problems for other frontends. The backend is still available for requests from other frontends. To keep things simple, we limit the number of concurrent frontends to 5.

Frontend Requirements

The pseudo code shown below for the frontend is not complete. It shows the bare essentials. You need to add the missing functions to make the frontend meet all the requirements and make it work with RPCServ and the backend.

```
backend = RPC_Connect(backendIP, backendPort)
while(no_exit) {
      print_prompt()
      line = read_line()
```

```
        cmd = parse_line(line)
        RPC_Call(backend, cmd.name, cmd.args)
    }
    RPC_Close(backend)
```

The frontend does not implement any of the commands the user enters into the shell. It simply relays them to the backend. You will notice that the command entered by the user looks like the following: command (string) and parameters. We will restrict the parameters to 2 or less. You can have commands with no parameters. The parameters can be integers or floating-point numbers. You need to have an RPCServ interface to send the command and parameters to the backend. In the pseudo code, we show such an interface – `RPC_Call()`. The frontend will check if the user has entered the `exit` command. If that is the case, the frontend will stop reading the next command and terminate the association with the backend. The backend is a separate process, so it keeps running even after the frontend has stopped running. The user can enter the `shutdown` command in the frontend to terminate the backend. With multiple frontends connecting to the backend, the shutdown can be tricky. More on this in the backend requirements. Some commands entered by the user in the frontend may not be recognized by the backend, in that case the backend will send the NOT_FOUND error message. The front needs to display this to the user. We can also have error message for certain operations such as division by zero errors. These error messages need to be displayed as well.

Backend Requirements

The pseudo code shown below for the backend is not complete. It shows the bare essentials. You need to add the missing functions to make the backend meet all the requirements and make it work with RPCServ and the frontend.

```
serv = RPC_Init(myIP, myPort)
for_all_functions(name)
        RPC_Register(serv, name, function)
while(no_shutdown) {
        client = accept_on_server_socket(serv)
        serv_client(client)
}
```

The backend sets up a server at `myPort` in the current machine (you can use "`127.0.0.1`" to point to the current machine). The server should be set up, so it is bound to the given port and is listening for incoming connections from the frontend. Before you start accepting the connections, you need to register all the functions that the RPCServ is willing to offer as a service. For example, if you have the following function to add two integers that you want to expose, you need to register it with the RPCServ as shown below.

```
int addInts(int x, int y) {
        return x + y;
}
```

```
RPC_Register("add", addInts)
```

The RPCServ is responsible for invoking (that is calling) the `addInts` function with the appropriate parameters when a request comes from the frontend.

The RPCServ keeps running until a shutdown command is issued by the frontend. With a single frontend, this is quite simple. You simply exit the program after closing the sockets in an orderly manner. With multiple frontends, things can get little bit tricky.

With multiple frontends, the pseudo code shown above needs some revision. Soon after accepting a connection, you need to create a child process and let that child process handle the new connection. That is the socket connection (client) is passed to the child process and it will be doing the calculations and sending the results or error back to the client. The server is free to loop back and accept another connection without waiting for the previous frontend's service to complete.

With multiple frontends, let's consider the situation where the backend received the shutdown command. The command is received in a child process. For the backend to terminate, we must terminate the parent process that started running the backend. The child that received the shutdown must notify the parent about the reception of the shutdown. For this purpose, the child can use the return value in an `exit()` system call. The parents gets the return value of the child using the `waitpid()` system call. The example code below shows how you can return a value from a child to parent.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
  int pid;
  int rval;

  if ((pid = fork()) == 0) {
    sleep(10);
    return 10;
  }

  while (1) {
    sleep(1);
    int res = waitpid(pid, &rval, WNOHANG);
    printf("Returned value %d\n", WEXITSTATUS(rval));

  }
}
```

On the server socket, the parent is going to block. It is waiting for new connections from the frontends. Once a connection comes in, the parent is going to unblock and proceed to create a child process to handle the frontend. To handle the shutdown properly, we need to check whether any child processes have already issued a shutdown. If so, we close the client connection that just arrived and stop accepting any more connections. When all the child processes that are running have completed their execution, the parent terminates. Because the parent is blocking on the server socket, it would not be able to shutdown or even know about the shutdown when it arrives. The

parent would only detect the shutdown at the arrival of the next frontend processing request. Therefore, the backend would keep going even after shutdown has been sent until the next frontend request.

NOTE: There is a better way of doing the same activity as the above that will use epoll() or select(). If you are following the advanced tutorials and are already familiar with C/Linux socket programming, you are strongly encouraged to use those system calls. The design is left to you, but your design needs to meet or exceed the above functionality. For example, with epoll() or select() you could terminate the backend without waiting until the next frontend request.

Backend Functions: You need to provide the following functions in the backend implementation.

```
1. int addInts(int a, int b);
   // add two integers
2. int multiplyInts(int a, int b);
   // multiple two integers
3. float divideFloats(float a, float b);
   // divide float numbers (report divide by zero error)
4. int sleep(int x);
   // make the calculator sleep for x seconds – this is blocking
5. uint64_t factorial(int x);
   // return factorial x
```

RPCServ Requirements

We are not going to test your RPCServ implementation with applications other than the calculator. Therefore, we are not standardizing on the RPCServ interface. However, you are strongly encouraged to provide at least the following.

```
rpc_t *RPC_Init(char *host, int port)
// rpc_t is a type defined by you and it holds
// all necessary state, config about the RPC connection
RPC_Register(rpc_t *r, char *name, callback_t fn)
// callback_t is type defined by you
rpc_t *RPC_Connect(char *name, int port)
RPC_Close(rpc_t *r)
RPC_Call(rpc_t *r, char *name, args..)
// You can have different variations to
// handle different number of parameters and types
```

This is a guide for you to organize the RPCServ implementation. You can change the function signatures and have more functions in your RPCServ implementation.

How the Assignment will be Graded?

We will use a shell script to grade your assignment. The shell script will start the backend and start the frontend and inject different inputs. The script checks the output from your frontend against an expected value and reports an error.

Grade distribution will be notified very soon by the TAs.

What You Need to Handin?

What needs to be handed in?

Source files, Makefile or CMakeLists.txt?

Can I Collaborate with My Friends?

This is an individual assignment. You can brainstorm with your friends in developing RPCServ and other components. The final implementation must be yours only. You cannot do group coding.