Proxy Lab Report

Informations

Name: CUI HANGQI (최항기)
Student-ID: 2016-27542
Lab: Proxy Lab

Process Introduction

This lab program start with the main function which initialized the openfile, semaphore, variables and waiting for connections from any other client, constantly.

When a client connect to the proxy, then it will create a new thread to deal the service, and wait for next client's connection.

The new thread will invoke the start routine — process_request(), and do the following tasks.

1. copy the arguments and free them to avoid memory leak.
2. Initialize the file descriptor and waiting for the request form the client by Rio_readlineb_w(), repeatedly.
3. When get the request buffer, firstly, parse the buffer into hostname, port and content (also error handling).
4. Create file descriptor & initialize, and forward the request to the server, then read the response forward it back to client.
5. When got the response from server, print log informations to log file.
6. Finally, close all the file descriptor.

Implementation

In Main Function

1. Open the proxy.log file and get the fd.
2. Initialize the semaphore.
3. Do a easy loop to wait for any client connect, the code in loop is same as the code in echo server.
4. create a new thread, and then the thread will start routine by invoking the process_request()

process_request()

This is the function that the new thread will going to do.

1. detach the thread by Pthread_detach(Pthread_self()); function.
2. copy the socket address and free arguments to avoid memory leak.
3. Initialize the file descriptor of client, then start a big loop to start read the line from client.

The while loop will read buffer from client constantly. Every time it reads some thing not 0 length.

1. Parse the buffer into hostname, port and content. If there is no arguments or arguments not in format, write error message back to client and continue.
2. Then create a file descriptor of server and initialize it. Send the content to the target server, read the response, deal the format of log as the pdf required, at last send the response back to the client.
3. Finally, when a client terminated, close both client and server file descriptor.

deal_log_format()

This function will generate the formatted log string.

1. Passing the content buffer, socket address, and the log buffer.
2. Use time system function get current time string.
3. Use ntohl() get the IP address, and parse it into 4 segment which we usually seen.
4. Use ntohs() get the client port.
5. Finally, use sprintf() concat the all parts into one string and give it to the log buffer.

open_clientfd_ts()

Because there is a gethostbyname() function in this function, so it can be thread unsafe.

And I use the semaphore P/V locked the function. Then return the result.

Rio_readn_w() / Rio_readlineb_w() / Rio_writen_w()

In these function, when the error occur it will terminate the program.

So rewrite the function by return 0 instead of exit.

print_log()

print the fog string into proxy.log file and output the log content.

To avoid concurrent requests or context switch occur in inexplicable place, use semaphore P/V lock write action.

Notice Points

I. gethostbyname() or gethostbyaddr() are Class-3 thread unsafe function, so use mutex lock the function, so at any time only one thread can call these functions.
II. Use Robust I/O
III. Call Signal(SIGPIPE, SIG_IGN) to ignore SIGPIPE signal.
IV. detach the thread.
V. free the memory, close the file descriptor.
VI. Error handling.

Difficulties

1. The first confusing problem is when I send message to server by writing buffer into Rio_writen() function, the server never received. This problem had plagued me for a while.

   - Reason: The reason is that I didn't add the newline character at the end of the buffer.
   - Solution: add '\n' or '\r\n' at the end of the buffer.

2. To test my proxy concurrency, I made a script and test it, but when I increase the request number to a value, Then I occurred an error: "Pthread_create error: Resource temporarily unavailable." and terminated, the proxy couldn't connect to the server any more and occurred an error: Open_clientfd Unix error: Connection refused.

   - Reason: Put the open_clientfd() in the loop, server created too many thread. By default, the stack size limited 8MB, so when the number of threads become more than 450~550, It can't create new thread any more.
   - Solution: Do the server file descriptor creation once, by a flag variable.

3. In order to test my proxy concurrency, I made a script by nodejs which I used to use, because I never used python.

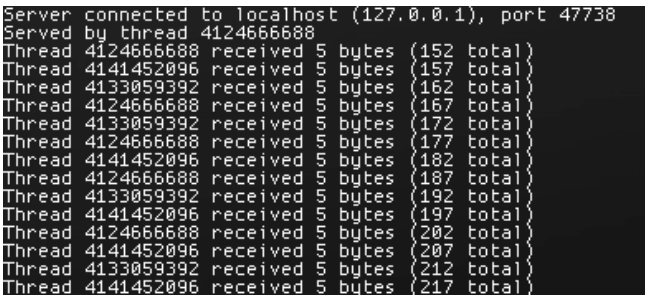
The script create 4 process running echoclient in 4 cpu core, respectively, which my virtual machine has the maximum number. Each client will then send multiple requests to the proxy server.

But I don't know why there always create 5 thread, and got close error.

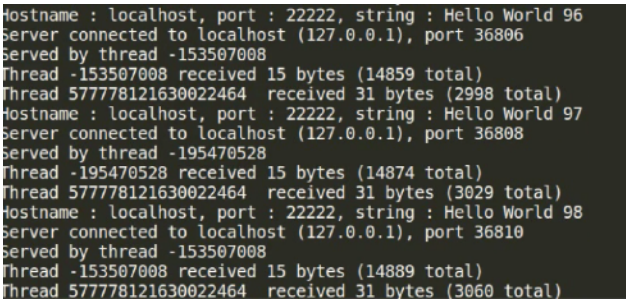
So every time if the server file descriptor didn't initialized, I skip close action.

Additional Finding

After second lab session, I found that my output is different from the video which updated in eTL. And I try to find the reason:



My proxy



eTL proxy

Reason:

- In proxy of mine, the process request function will create a file descriptor of server, and won't close it until the connection of client disconnected. So it can send request again and again with no more reconnections.
- The example proxy of eTL will create a new file descriptor and close it after get response, every time.

Then I asked TA for this finding, they say both implementation are correct. So I won't modify it to what showed in the video.

Script

To test my proxy concurrency, we need to create a script that can send concurrent requests. In the example, TA showed us a python script, but I never used python. As a matter of convenience, I decided use the script what I used to use — Nodejs.

I Nodejs there is a native module named Cluster which can take advantage of multi-core systems.

Firstly, it need to require the cluster module and get the number of CPUs.

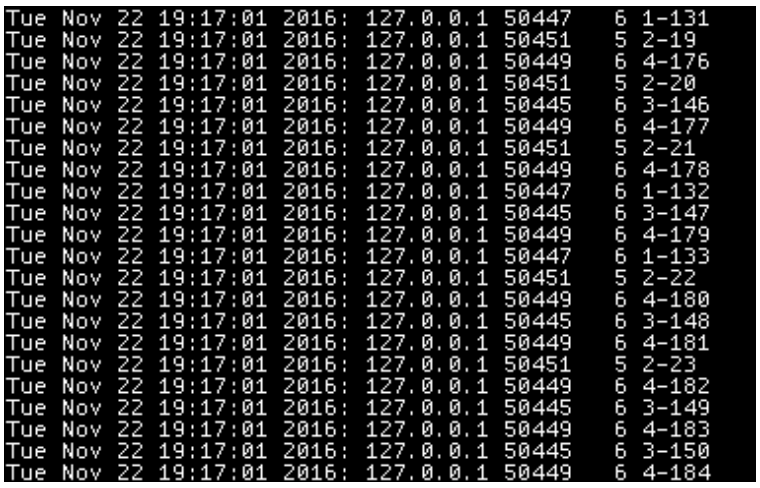
```
const cluster = require('cluster');
const numCPUs = require('os').cpus().length;
```

In the master process, it will fork 4 process which is the maximum number of my virtual machine has. and handling the exit event.

```
if (cluster.isMaster) {
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('exit', (worker, code, signal) => {
    …
      process.exit(0);
    …
  });
}
```

Every process will balancedly use each cores, and start to do the requests contiguously.



proxy.log

From the result of proxy.log we can see that the 4 processes associated well, and there is no line jump into any other line.