web-server

an event-driven web server and client

qzane
qzane@live.com

1. Content

1. Content **1**
2. What is this **1**
2.1. Install 1
2.2. server & client 2
2.3. server_loop 2
2.4. client_multi_process 3
2.5. server_multi_process 4
3. Result **3**
3.1. test 3
3.2. result 3
4. design **3**
4.1. server & client & server_loop 3
4.2. client_multi_process 3
4.3. server_multi_process 4

2. What is this

This web-server project contains the following files:

- server.c
- server_loop.c
- server_multi_process.c
- client.c
- client_multi_process.c

2.1. Install

```
gcc -o client client.c
gcc -o client_multi_process client_multi_process.c
gcc -o server server.c
```

```
gcc -o server_loop server_loop.c
gcc -o server_multi_process server_multi_process.c
```

2.2. server & client

on server machine

```
>>>./server <port>
```

on client machine

```
>>>./client <server's ip> <port>
Please enter the message: num=10-
result=29
```

explain: The server listen to the specified prot. Once it receive a message like "num=x-", it will reply "result=y", and the value of y equals $x * 3 - 1$. Also, it will wait for about 0.32s for each request simulating the IO delay. The client connect to the server with its ip address and port. Once the connection is set, the client'll ask you to input a message. Then, the client will sent the message to the server and display its reply.

2.3. server_loop

```
>>>./server_loop <port>
```

Very similar to **server** except that it will wait for another connection after each work.

2.4. client_multi_process

```
>>>./client_multi_process <server's ip> <port>
```

The main process of this program generate 20 sub-process, each of which will send a message like "num=count-" to the server. the values of count are 0,1,2...19.

2.5. server_multi_process

```
>>>./server_multi_process <port>
```

The real event-driven web server. The behavior is similar to **server_loop**, but once this program get a new request, it will generate a new process to deal with that request, so that the main process can keep on listening.

3. Result

3.1. test

run the following commands on server machine:

```
>>>./server_loop <port>          # test 1
>>>./server_multi_process <port> # test 2
```

for each test, run this commands on client machine:

```
>>>time ./client_multi_process <server's ip> <port>
```

3.2. result

test 1:

```
real    0m5.151s
user    0m0.001s
sys     0m0.002s
```

test 2:

```
real    0m0.363s
user    0m0.000s
sys     0m0.002s
```

PS: the client detects some errors when using server_loop but this thing didn't happened when using server_multi_process. So, I guess that the *backlog* argument of *listen*(2) only limit programs with single thread.

4. design

4.1. server & client & server_loop

I learnt sockets programming from the web-site:

http://www.linuxhowtos.org/C_C++/socket.htm

4.2. client_multi_process

```
for(count=0;count<PROCESS_NUM && fork()!=0;++count)
    ;//make PROCESS_NUM copies

if(count==PROCESS_NUM){//root process
    while(wait(NULL)!=-1)
        ;
    printf("\nAll finished!\n");
    exit(0);
}
```

4.3. server_multi_process

```
newsockfd = accept(sockfd,(struct sockaddr *)
                   &cli_addr,&clilen);
if(fork()==0){
    process(newsockfd);
    close(newsockfd);
    printf("Work%4d done!\n",count);
    exit(0);
}
close(newsockfd);
++count;
```