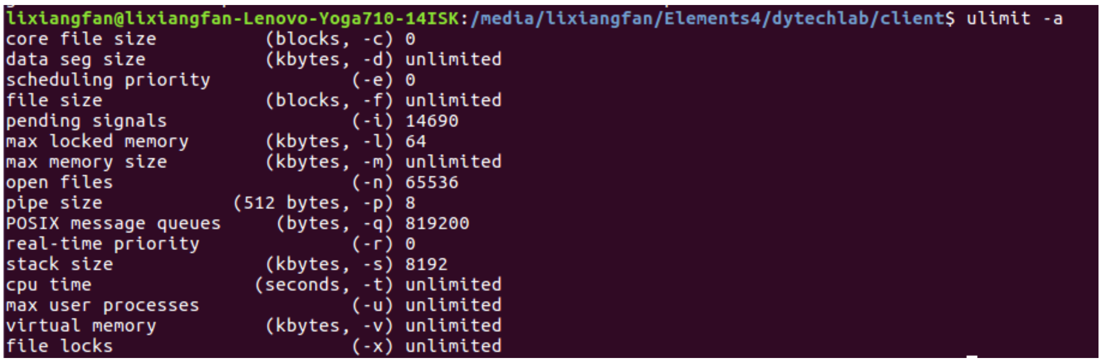
Implement a simple http server that uses epoll and threadpool to ensure high concurrency and fast response. It's just a viable framework and does not provide any useful service. All I care about is the concurrency and response frequency.

How to run the programme

- the programme should be run on a Linux OS
- under the server/client folder, use `make` command to compile and link the server/client, then you will obtain an executable file `testserver` / `testclient`.
- first, start the server using command `./testserver` under the server folder.
- then, start the clients using command `./testclient` under the client folder. The `testclient` programme will start a number of clients and send certain numbers of requests to the server and get response message from the server.
- To run the test programme successfully, you should:
    - make sure that you start the server before the clients.
    - the value of `SERVER_IP` in `testclient.cpp` should be the ip address of the computer on which you run the `testserver` programme, and the value of `SERVER_PORT` in `testclient.cpp` should be the same as `PORT` in `testserver.cpp`, and it should be an available interface.
    - the `testclient` programme will launch `CONCURRENT_CLIENTS` number of clients, each will send `REQUESTS_PER_CLIENT` requests to the server, if you set `CONCURRENT_CLIENTS` to a large value, then you must use `ulimit -u/-s/-n` to change user limits to a larger value:

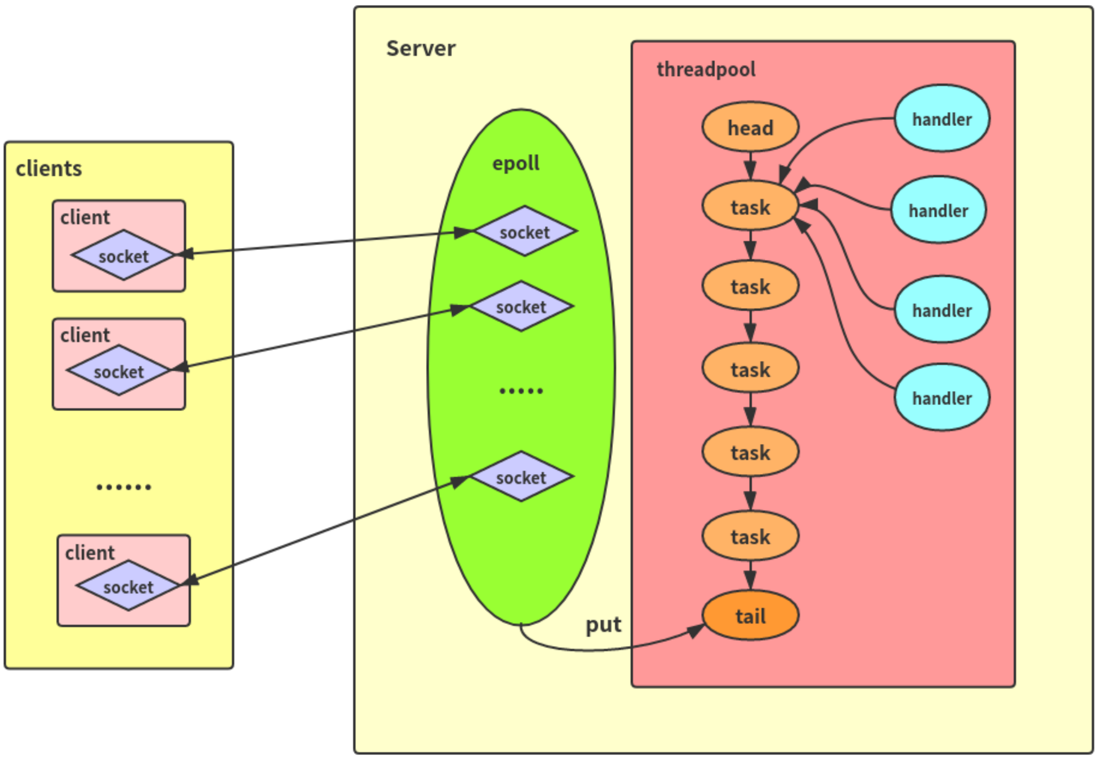


```
lixiangfan@lixiangfan-Lenovo-Yoga710-14ISK:/media/lixiangfan/Elements4/dytechlab/client$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 14690
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 65536
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) unlimited
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

    - `CONCURRENT_CLIENTS` in `testclient.cpp` could not be greater than `MAX_CLIENTS` in `testserver.cpp`
    - set `MEASURE_TIME` to true if you want to measure programming time. If `MEASURE_TIME` was set to true, the programme would avoid printing request/response message to the console which will slow down the programming time.

Implementation details

The following image shows the basic framework of the server:



- Http1.1 default support Persistent Connections, so I use tcp socket to implement http connections. The client would establish connection with the server only once.
- The httpserver does not provide any useful service, in my test programme, the clients just send request messages to say hello to the server. Everytime the server receives a "Hello" message, it would send back a response message to say hello to the client.
- In the main thread of the server, I use a `master_socket` to listen the connect events of the clients, and for every client that is connected to he server, I use a `client_socket` to listen the request/disconnect events of it.
- Epoll and threadpool are the key points of the server framework, they ensure the efficient performance of the server under high concurrency.
- To avoid sticky packets, I pad every request/response message to 1024 bytes.

epoll

- epoll is an improved version of poll, it's used for handle large number of file descriptors in Linux system and it is more efficient than select/poll
- In my server, I use epoll to listen the events that happen on all the client sockets, epoll can notify me of the event type and the socket descriptor on which the event happened in just O(1) time, compared with select/poll which needs O(n) time to locate the socket. Under high concurrency, this would improve the response performance greatly.

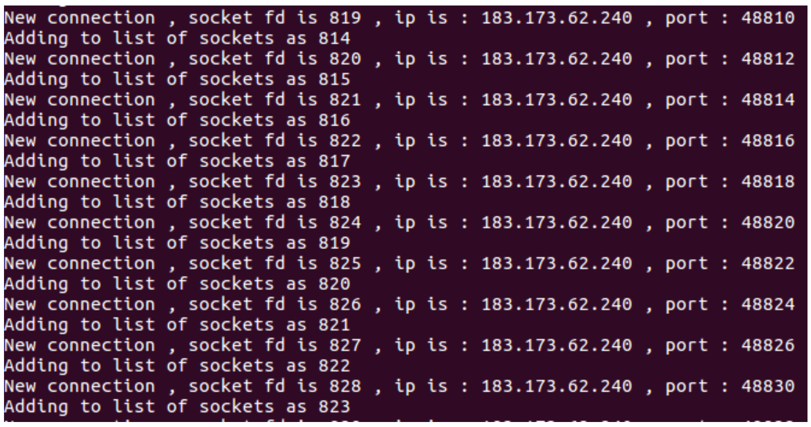
threadpool

- In order to handle the requests that the server received using multi-thread, I design a structure called threadpool. It maintains a task queue and ensure these tasks can be safely handled by multiple threads.
- Each time the server receives a request, the task(mainly the socket descriptor of the request client) would be put to the tail of the task queue.

- `MAX_THREADS` number of threads would continuously take the first task out of the task queue and handles it. To be more specific, these threads would get the corresponding socket descriptor of the first task, read message from that socket, and send back response message.
- Threadpool uses `pthread_mutex_t` and `pthread_cond_t` to ensure mutually exclusive access to the task queue.

Concurrency

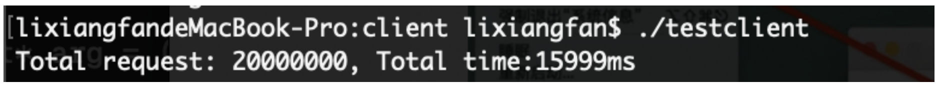
- Set `MAX_CLINETS` in `testserver.cpp` to the maximum number of clients the server can serve concurrently. This number can be large, only restricted by your machine limit.
- I set `CONCURRENT_CLIENTS` to 10000 in `testclient.cpp`, `MAX_CLIENTS` to 10000 in `testserver.cpp`, and run the programme, the server can handle clients' requests efficiently:



```
New connection , socket fd is 819 , ip is : 183.173.62.240 , port : 48810
Adding to list of sockets as 814
New connection , socket fd is 820 , ip is : 183.173.62.240 , port : 48812
Adding to list of sockets as 815
New connection , socket fd is 821 , ip is : 183.173.62.240 , port : 48814
Adding to list of sockets as 816
New connection , socket fd is 822 , ip is : 183.173.62.240 , port : 48816
Adding to list of sockets as 817
New connection , socket fd is 823 , ip is : 183.173.62.240 , port : 48818
Adding to list of sockets as 818
New connection , socket fd is 824 , ip is : 183.173.62.240 , port : 48820
Adding to list of sockets as 819
New connection , socket fd is 825 , ip is : 183.173.62.240 , port : 48822
Adding to list of sockets as 820
New connection , socket fd is 826 , ip is : 183.173.62.240 , port : 48824
Adding to list of sockets as 821
New connection , socket fd is 827 , ip is : 183.173.62.240 , port : 48826
Adding to list of sockets as 822
New connection , socket fd is 828 , ip is : 183.173.62.240 , port : 48830
Adding to list of sockets as 823
```

Response frequency

- Now I would measure the response frequency of the server. Of course I don't have 10000 computers to run the clients. But if I use 10000 threads in one computer to simulate 10000 clients, the total request frequency would by slow down because of large numbers of thread switches. So my plan was: because I have two personal computers, one MAC OS and another Linux, I would start 2 clients on the MAC and run the server on the Linux, the clients would send large numbers of requests simultaneously to ensure high requests frequency. Then I can measure the response frequency of the server.
- set `MAX_THREADS` in `testserver.cpp` to 4 (4 threads handle requests), and set `REQUESTS_PER_CLIENT` to 10000000, `CONCURRENT_CLIENTS` to 2, run the programme



```
[lixiangfandeMacBook-Pro:client lixiangfan$ ./testclient
 Total request: 20000000, Total time:15999ms
```

$$\text{response\_frequency} = \frac{20000000}{15.999\text{s}} = 12500781 \text{ response / s}$$

References

- Socket Programming in C/C++:

  https://www.geeksforgeeks.org/socket-programming-cc/
- NIO network model:

  https://blog.csdn.net/qq_28303495/article/details/89514690

- Epoll usage:

  https://blog.csdn.net/qq_17308321/article/details/85254165

- Thread pool:

  https://en.wikipedia.org/wiki/Thread_pool