Producer-consumer Model Based Thread Pool Design

Producer-consumer Model Based Thread Pool Design

Liangzhou Wang[1,a] and Chaobin Wang[2,b]

[1]School of computer science, China West Normal University, Nanchong, Sichuan, China
[2]Corresponding author, School of computer science, ChinaWest Normal University, Nanchong, Sichuan, China

[a]Mobile:13696228109 Email: 815937667@qq. com
[b]Mobile:13990777368 Email: cbwang@cwnu. edu. cn

Abstract: The concurrency of the server has a crucial impact on the efficiency of processing requests on the server side, so the thread pool technology has been widely applied to the server. Different thread pools play an important role in improving the performance of highly concurrent servers. This article first introduces socket technology, thread pools, etc. , which are high-concurrency server development technologies in Linux systems, then proposes a thread pool model based on the producer consumer model, and gives key codes. Finally, the performance is compared through experiments . Experimental results show that lightweight servers can achieve higher concurrency by using this thread pool model.

1. Introduction

With the development of information technology, countless users communicate on the Internet every day, and the server plays an important role in it. Therefore, the server receives more and more requests. And how to allow the server to handle a large number of concurrent connections in a short time while also ensuring the reliability of the server has become the first problem to be solved in the server development process.

Since the advent of the Internet, server technology has evolved along with the development of the Internet industry. The early server was a traditional Unix model, which is a single-process server, and a new process was created to process each new request. Today's mainstream multi-process servers derive a certain amount of processes in advance, and wait for the request to be directly allocated to the process. Typical multi-process servers include Apache and Microsoft IIS. With the advent of multi-core processors, multi-threaded server technology has also been developed. The Nginx server is a typical multi-threaded server. Domestic Taobao, NetEase and other companies have built a server framework based on Nginx.

For light-weight servers, multi-process servers consume more memory and have lower concurrent processing power. Frequent creation of multi-threads also consumes more light servers. Therefore, traditional multi-process / multi-thread based concurrent servers can no longer meet the requirements of high concurrency and high reliability. To deal with the shortcomings of traditional concurrent servers, this paper proposes a thread pool based on the ideas of producers and consumers, which can keep lightweight servers in a most efficient state.

2. High concurrent server development technology

2. 1. socket technology

2. 1. 1. The meaning of socket. Socket is an interface specification used to directly access the communication protocol, the program can directly access TCP, UDP and other protocols to receive and send data through the socket. According to the definition of RFC793, the socket is composed of the port number and IP.

2. 1. 2. Socket programming framework. Server communication process:(1) Call the socket () function to create a socket and return the file descriptor "listenfd".

(2) Call the bind (listenfd, server address port) function to bind the socket with the local address and port number.

(3) Call the listen (listenfd, connection queue length) function to declare the socket as listening mode.

(4) Call the accept (listenfd, client address port) function. After the three-way handshake, the server calls this function to block waiting for the client to connect, and returns a new socket "connfd" for communication with the client.

(5) Call the read (connfd, buf, size) and write (connfd, buf, size) functions to communicate with the client.

(6) Call the close (connfd) function to close the socket.

Client communication process:

(1) Call the socket () function to create a socket and return the file descriptor "fd".

(2) Call the connect (fd, server address port) function, send a request to the server, and establish a connection.

(3) Call write (fd, buf, size) and read (fd, buf, size) functions to communicate with the server.

(4) Call the close (fd) function to close the socket.

The specific flowchart is shown in Figure 1:

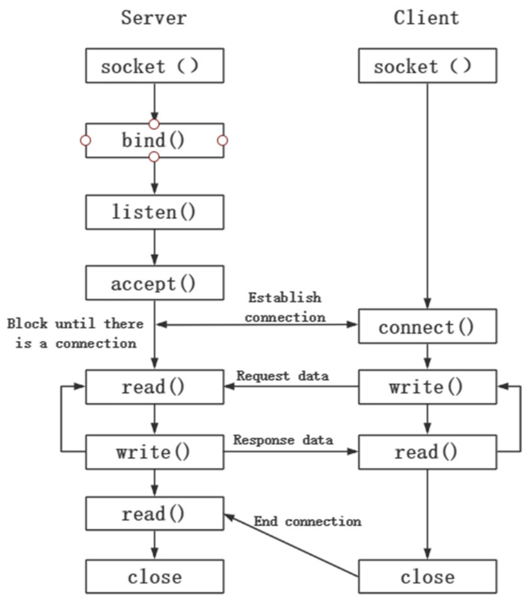

Figure 1. Flow chart of creating a socket model.

2. 2. Producer-consumer model

The producer-consumer model is a model that solves the problem of strong coupling between producers and consumers by using a blocking queue. There is no direct data communication between the two, but communication through blocking queues. After the producer produces the data, it is not submitted to the consumer, but directly placed in the blocking queue, and then the consumer directly fetches the data from the blocking queue. The blocking queue provides a buffer that can balance the processing power of producers and consumers, and can be used to decouple producers and consumers. In general, the producer-consumer model has the advantages of decoupling, support for concurrency, and support for uneven busyness.

2. 3. Linux multithreaded programming

2. 3. 1. Processes and threads. A process is a basic unit for resource allocation in an operating system, and a thread is an execution unit within a process, and is a basic unit that is independently scheduled and dispatched by the operating system. All resources owned by the process are shared among threads. Compared with processes, threads have the advantages of low resource consumption and fast creation speed. Therefore, in the process of creating highly concurrent servers, multi-threaded programming has gradually become the mainstream.

2. 3. 2. Thread pool technology. Although the resource consumption required to create a thread is greatly reduced compared to the creation process, if the server needs to process a task with a short execution time and frequent requests, the server will continuously create and destroy threads. In this process, a lot of resources are wasted. Therefore, it is not suitable for high concurrency situations. The thread pool technology creates a certain amount of threads in advance, which saves the resources for creating and destroying threads, making the server respond faster and more efficient.

3. Thread pool model based on producer-consumer model

3. 1. Creation of thread pool model

The thread pool model based on the producer-consumer model is shown in Figure 2. For the client and the task queue, the client is the producer and the task queue is the consumer, then the client puts a request in the task queue; for the thread pool and the task queue, the thread pool is the consumer and the task queue is the producer, then the thread pool takes the request sent by the client from the task queue.

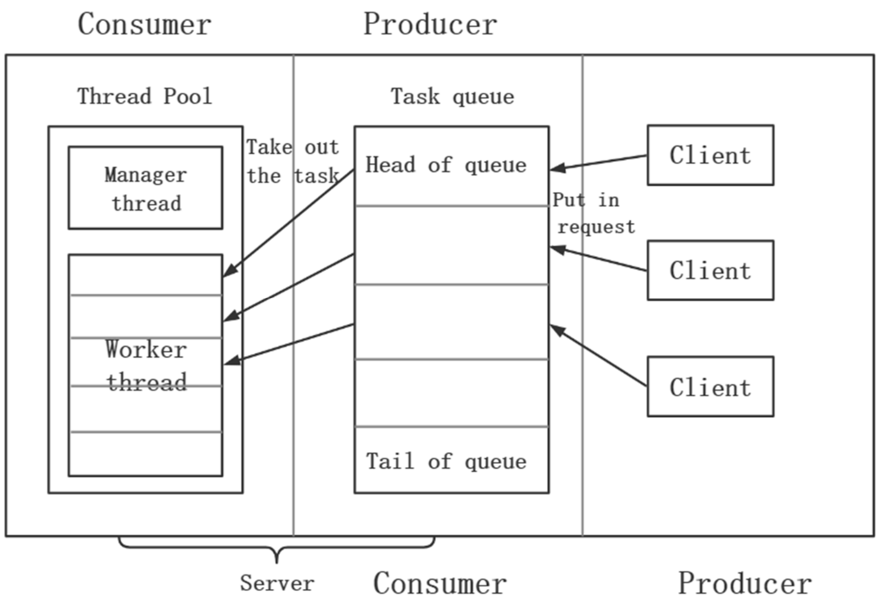

Figure 2. Thread pool model diagram.

(1) The task queue is responsible for accepting client connection requests and placing them in the task queue. When the task queue is full, it blocks clients to put requests into the task queue; when the

task queue is empty, it blocks clients to take the task from the task queue.

(2) The thread pool includes the manager thread and the worker thread. The manager thread is not responsible for processing tasks. Instead, it queries the task queue every time whether the task queue is empty. If there are tasks in the task queue and there are no idle threads or there are too many working threads, the manager thread will create a certain amount of threads, also it can destroy idle threads according to a certain ratio; the worker thread is only responsible for processing tasks. When there is a task in the task queue, the blocked worker thread is woken up and the worker thread takes the task from the task queue and removes the task from the task queue.

The specific workflow is shown in Figure 3. First, the worker thread and the manager thread are created by initializing the thread pool. The number of initial worker threads can be given a value according to actual needs. Then when the task queue is empty, the thread blocks and waits for the task queue to receive the request. When the task queue is not empty, the blocked thread is awakened, the worker thread takes the task from the head of the task queue, and finally the task processing function of the worker thread starts to process the obtained request.

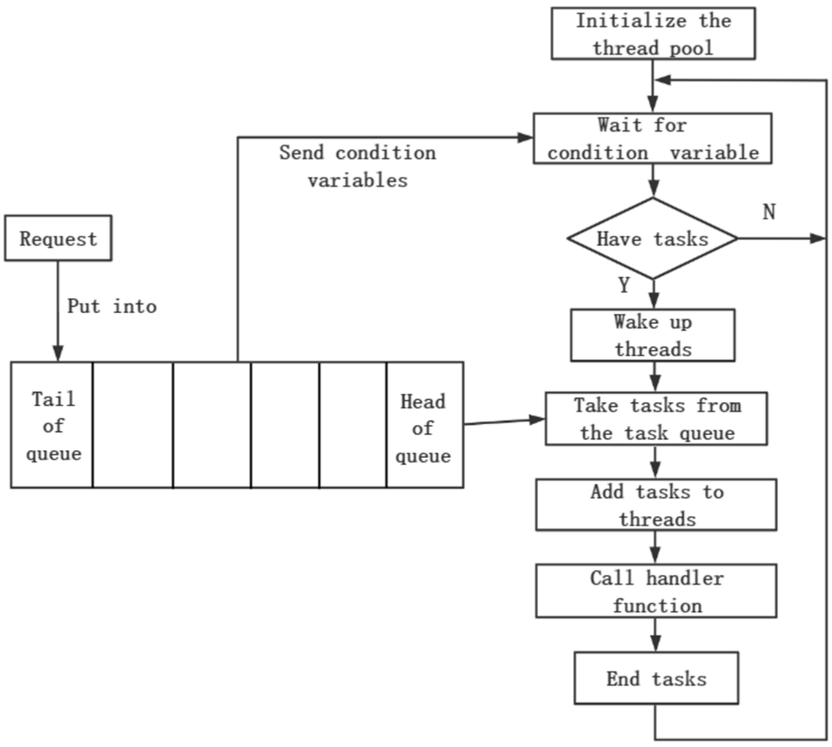

Figure 3. Workflow of thread pool.

3. 2. Thread pool code implementation

Some key codes of thread pool creation are as follows.

3. 2. 1. Task queue code. (1) The structure of the task queue is defined as follows:

```
struct taskQueueNode        // Structure of each node in the task queue
{
      void* (*function)(void*); // Callback function, client request stored by node
      void* arg;                          // Function parameter
};
struct taskQueue
{
   pthread_cond_t queue_full;     // Condition variable, task queue is full
```

```
    pthread_cond_t queue_empty;        // Condition variable, task queue is empty
    pthread_mutex_t lock;              // Mutual exclusion locks, blocking condition variables
    taskQueueNode* task_queue; // A queue consisting of nodes of type taskQueueNode*
    taskQueueNode* head;               //Head node
    taskQueueNode* tail;                //Tail node
    int size;                           // Number of nodes
    int maxsize;                        // Maximum number of nodes
};
```

(2) Initialize the task queue: initialize the relevant parameters of the task queue, the maximum number of nodes is determined according to actual needs.

```
taskQueue* taskqueue_init(int maxsize)
{
    taskQueue*    queue = (taskQueue*)malloc(sizeof(taskQueue));
    queue->size = 0;
    queue->maxsize = maxsize;
    queue->front = 0;
    queue->tail = 0;
    return queue;
}
```

(3) Add tasks to the task queue: call this function in the main function, through the callback function parameter void * (* function) (void * arg) of the function, you can directly insert the task into the task queue and use pthread_cond_wait () and pthread_cond_signal () function to block and wake up the thread.

```
int task_add(threadPool* queue, void* (*function)(void* arg), void* arg)
{
    while (queue->size == queue->maxsize)// When the task queue is full, block
    {
        pthread_cond_wait(&(queue->queue_full), &(queue->lock));
    }
    queue->tail->function = function;      // Add tasks
    queue->tail->arg = arg;
    queue->tail = queue->tail + 1;
    queue->size++;
    pthread_cond_signal(&(queue->queue_empty)); // When the task
                //queue is not empty, wake up the blocked thread
    return 0;
}
```

3. 2. 2. Thread pool code. (1) The structure of the thread pool is defined as follows:

```
struct threadPool
{
    pthread_mutex_t lock;        // Mutex
    pthread_t* workthr_tid;    // Worker thread tid
   pthread_t adminthr_tid;    //Manager thread tid
   pthread_t destroythr_tid; // Thread tid to be destroyed
    int min_num;                        // Minimum number of threads
    int max_num;                    // Maximum number of threads
    int live_num;                       // Number of live threads
    int work_num;                      // Number of worker threads
};
```

(2) Initialize the thread pool: initialize the relevant parameters of the thread pool, and use the Linux thread function pthread_create () to create and create manager threads and a certain number of worker threads. The minimum number of threads and the maximum number of threads are determined according to actual needs.

```
threadPool* threadpool_create(int min_num, int max_num)
{
    threadPool* poll = (threadPool*)malloc(sizeof(threadPool));
    poll->min_num = min_num;
    poll->max_num = max_num;
    poll->work_num = 0;
    poll->live_num = min_num;
    poll->size = 0;
    poll->workthr_tid = (pthread_t*)malloc(sizeof(pthread_t));
    for (int i=0; i<min_num; i++)      // Create worker thread
    {
        pthread_create(&(poll->workthr_tid[i]), NULL, work_thread, (void*)poll);
     }
    pthread_create(&(poll->adminthr_tid), NULL, admin_thread, (void*)poll);     // Create
manager
                                                                                 //thread
    return poll;
}
```

(3) Worker thread function: the implementation function of the worker thread task is called when the worker thread is created, and when it is awakened, the task is taken from the task queue and executed.

```
void* work_thread(void* arg)
{
     ...........
    if (queue->size==0)// When the task queue is empty, the thread is blocked
    {
         pthread_cond_wait(&(queue->queue_empty), &(queue->lock));
    }
    ...........
    taskQueueNode* node;
    node->function = queue->head->function; // Take tasks from the task queue
    node->arg = queue->head->arg;
    queue->head = queue->head + 1;
    queue->size--;
    pthread_cond_signal(&(queue->queue_full)); // Wake up threads
                                             //blocked when the task queue is full
    pool->work_num++;
    (*(node. function))(node. arg);                     // Perform tasks
...........
    }
}
```

(4) Manager thread function: the implementation function of the manager thread task, which is called when the manager thread is created, mainly uses pthread_create () and pthread_exit () to create new threads and destroy threads, and individual parameters can be modified as needed.

```
void* admin_thread(void* arg)
```

```
    {
        …………
        int size = queue->size;
        int live_num = pool->live_num;
        int work_num = pool->work_num;
/* If the number of tasks is greater than the minimum number of threads and the number of surviving threads is less than the maximum number of threads, create new threads*/
          if (size >= MIN_NUM & live_num < pool->max_num)
            {
               pthread_create(&(pool->workthr_tid[i]), NULL, work_thread, (void*)arg);
               pool->live_num++;
            }
 /* If twice the number of worker threads is less than the number of surviving threads and the number of surviving threads is greater than the minimum number of threads, destroy the thread */
   if ((work_num * 2)< live_num & live_num>pool->min_num)//
        {
                    pthread_exit(&(pool->destroythr_tid));
        }
    …………
        }
```

4. Thread pool test results and analysis

Siege is a high-performance stress testing tool that can be used for stress testing and performance evaluation of servers. Therefore, in order to test the performance of the thread pool proposed in this paper, a simple web server was written, one does not use optimization, and the other uses thread pool optimization. The siege tool simulates multiple concurrent access servers. The response time and the number of successful transmissions of the two are collected. Some performance data are shown in Table 1 and Table 2:

Table 1. Original server performance.

Concurrent number	Average response time /s	Number of successful transmissions	Transactions per second
100	0. 63	100	32. 68
200	0. 76	171	30. 13
300	0. 85	241	30. 01
400	1. 03	310	28. 98
500	1. 41	372	21. 76

Table 2. Thread pool server performance.

Concurrent number	Average response time /s	Number of successful transmissions	Transactions per second
100	0. 59	100	35. 71
200	0. 69	180	33. 48
300	0. 81	278	31. 25
400	0. 89	375	30. 24
500	1. 03	464	28. 25

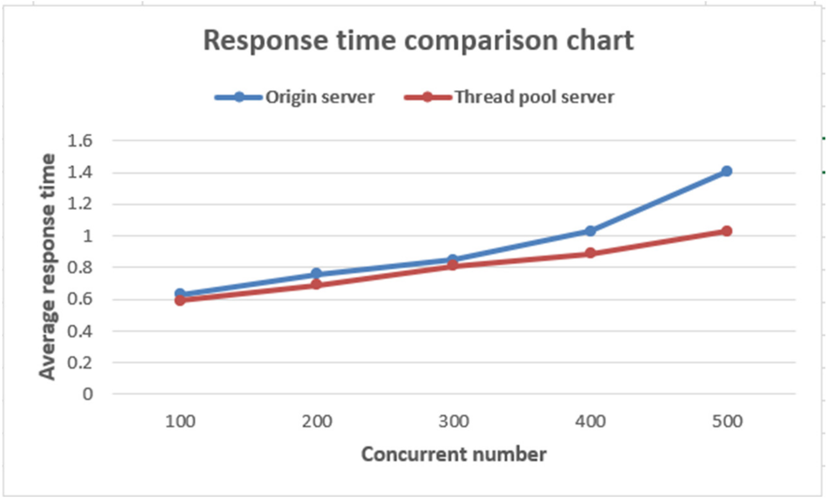

Figure 4. Comparison of response time.

It can be seen from Figure 4 that when the number of concurrent connections simulated by the siege tool is increasing, the response time of the server is also increasing. When the number of concurrency is not high, the difference between the response time of the original server and the thread pool server is not large, but once the number of concurrency is higher than about 400, the response time of the original server will rise rapidly, while the increment of the thread pool server remains unchanged, and the overall response time is lower than the original server.

Other performance changes, such as the number of successful transmissions and transaction volume per second, are generally similar to the response time. At low concurrent numbers, the performance difference between the two servers is not large, but when the concurrent number is too high, the original server 's Performance will drop dramatically, and the thread pool server will remain stable.

Based on the above experimental results, compared with the original server, the thread pool model studied in this paper not only improves the performance, but also improves the stability. Therefore, it can be applied to a lightweight and highly concurrent server.

5. Conclusion

This article reveals some of the problems with traditional servers by introducing modern high-concurrency server development techniques. In order to solve these problems, this paper proposes a thread pool model based on producer-consumer model. High concurrency is achieved as much as possible and the server's operating efficiency is optimized. Through the analysis of experimental results , the lightweight server based on this thread pool has good stability and high reliability.

Acknowledgments

In the process of writing this article, the tutor's guidance was given from the beginning to the end. I would like to thank my tutor for his guidance.

References

[1] Peng Hua . Effect of thread pool scheduling on server performance[J]. Communications Technology. 2019(09) (in Chinese).
[2] Liang Minggang, Chen Xiqu. The researchon realization of high concurrent server base on epoll plus threadpool in Linux[J]. Journal of Wuhan Polytechnic University. 2012(03) (in Chinese).
[3] Deng Ting. Communication design of remote monitoring system based on secondary C/S model[J] Microelectronics & Computer. 2015(06) (in Chinese).
[4] W. Richard, StevensBillFenner and AndrewM. Rudoff . UNIX Network Programming[M]. Beijing: Posts & Telecom Press. 2006
[5] Lu Xiangqian, Xie Chuiyi and Huo Yin . Parallel algorithm and simulation application on

randomness producer-consumer problem[J] . Computer Applications and Software. 2018(05) (in Chinese).
[6] You Shuang . High Performance Linux Server Programming[M]. Beijing: China Machine Press:2013 .
[7] Zhang Yao. The improvement and implement of high concurrency web server based on Nginx[D] Jilin University. 2016 (in Chinese).
[8] GulKSQ, WANG Peng, Luo Senlin and Pan Limin . A technical research on high-concurrency web application[J] . Netinfo Security. 2017(12):29-35 (in Chinese).
[9] Cao Wenbin, Tan Xinming, Liu Bei and Liu Chuanwen . Design and implementation of event-driven high performance websocket server[J] . Computer Applications and Software. 2018, 35(1):21-27 (in Chinese).
[10] Qiu Jie, Zhu Xiaoshu and Sun Xiaoyan . Research and implementation of message push based on Epoll model[J]. Journal of Hefei University of Technology(Natural Science) . 2016. 39(4):476-480. (in Chinese).