计科21-2 2021011587 吴维皓

实验四

题目1到4都在源码中

题目5

多线程模型

```
[linux1@bogon web]$ sudo /root/http_load-12mar2006/http_load -p 1000 -s 30 url.txt
http://192.168.88.132:8015/index.html: byte count wrong
http://192.168.88.132:8015/index.html: byte count wrong
http://192.168.88.132:8015/index.html: byte count wrong
233157 fetches, 1000 max parallel, 6.3651e+07 bytes, in 30 seconds
272.996 mean bytes/connection
7771.9 fetches/sec, 2.1217e+06 bytes/sec
msecs/connect: 106.915 mean, 7022.51 max, 0.015 min
msecs/first-response: 7.79382 mean, 6418.16 max, 0.045 min
3 bad byte counts
HTTP response codes:
  code 200 -- 233154
```

线程池模型

- （容量：100，并发度：1000 ---- 最优）

```
[linux1@bogon web]$ sudo /root/http_load-12mar2006/http_load -p 1000 -s 30 url.txt
285765 fetches, 1000 max parallel, 7.80138e+07 bytes, in 30 seconds
273 mean bytes/connection
9525.49 fetches/sec, 2.60046e+06 bytes/sec
msecs/connect: 1.6802 mean, 1003.74 max, 0.016 min
msecs/first-response: 91.4689 mean, 292.256 max, 0.611 min
HTTP response codes:
  code 200 -- 285765
```

- （200，1000）

```
[linux1@bogon web]$ sudo /root/http_load-12mar2006/http_load -p 1000 -s 30 url.txt
285743 fetches, 1000 max parallel, 7.80078e+07 bytes, in 30.0002 seconds
273 mean bytes/connection
9524.71 fetches/sec, 2.60025e+06 bytes/sec
msecs/connect: 1.5954 mean, 1004.53 max, 0.014 min
msecs/first-response: 91.5605 mean, 293.804 max, 0.392 min
HTTP response codes:
  code 200 -- 285743
```

- （50，1000）

```
[linux1@bogon web]$ sudo /root/http_load-12mar2006/http_load -p 1000 -s 30 url.txt
142750 fetches, 1000 max parallel, 3.89708e+07 bytes, in 30 seconds
273 mean bytes/connection
4758.33 fetches/sec, 1.29902e+06 bytes/sec
msecs/connect: 1.66435 mean, 1003.88 max, 0.016 min
msecs/first-response: 196.528 mean, 398.487 max, 11.759 min
HTTP response codes:
  code 200 -- 142750
```

- （50，500）

```
[linux1@bogon web]$ sudo /root/http_load-12mar2006/http_load -p 500 -s 20 url.txt
76200 fetches, 500 max parallel, 2.08026e+07 bytes, in 20 seconds
273 mean bytes/connection
3810 fetches/sec, 1.04013e+06 bytes/sec
msecs/connect: 0.994695 mean, 1003.03 max, 0.016 min
msecs/first-response: 119.053 mean, 303.532 max, 1.154 min
HTTP response codes:
  code 200 -- 76200
```

- (50, 300)

```
[linux1@bogon web]$ sudo /root/http_load-12mar2006/http_load -p 300 -s 10 url.txt
38080 fetches, 300 max parallel, 1.03958e+07 bytes, in 10 seconds
273 mean bytes/connection
3808 fetches/sec, 1.03958e+06 bytes/sec
msecs/connect: 1.77149 mean, 1000.95 max, 0.013 min
msecs/first-response: 66.0089 mean, 251.006 max, 0.555 min
HTTP response codes:
  code 200 -- 38080
```

- (60, 50)

```
[linux1@bogon web]$ sudo /root/http_load-12mar2006/http_load -p 50 -s 5 url.txt
19040 fetches, 50 max parallel, 5.19792e+06 bytes, in 5 seconds
273 mean bytes/connection
3808 fetches/sec, 1.03958e+06 bytes/sec
msecs/connect: 0.061181 mean, 0.51 max, 0.012 min
msecs/first-response: 2.70486 mean, 11.518 max, 0.06 min
HTTP response codes:
  code 200 -- 19040
```

- (50, 50)

```
[linux1@bogon web]$ sudo /root/http_load-12mar2006/http_load -p 50 -s 5 url.txt
23506 fetches, 50 max parallel, 6.41714e+06 bytes, in 5 seconds
273 mean bytes/connection
4701.2 fetches/sec, 1.28343e+06 bytes/sec
msecs/connect: 0.0515611 mean, 0.52 max, 0.013 min
msecs/first-response: 0.263503 mean, 10.737 max, 0.038 min
HTTP response codes:
  code 200 -- 23506
```

- (40, 50)

```
[linux1@bogon web]$ sudo /root/http_load-12mar2006/http_load -p 50 -s 5 url.txt
19040 fetches, 50 max parallel, 5.19792e+06 bytes, in 5 seconds
273 mean bytes/connection
3808 fetches/sec, 1.03958e+06 bytes/sec
msecs/connect: 0.061181 mean, 0.51 max, 0.012 min
msecs/first-response: 2.70486 mean, 11.518 max, 0.06 min
HTTP response codes:
  code 200 -- 19040
```

- (4, 50)

```
[linux1@bogon web]$ sudo /root/http_load-12mar2006/http_load -p 50 -s 5 url.txt
1904 fetches, 50 max parallel, 519792 bytes, in 5 seconds
273 mean bytes/connection
380.8 fetches/sec, 103958 bytes/sec
msecs/connect: 0.0586691 mean, 0.412 max, 0.013 min
msecs/first-response: 119.157 mean, 130.059 max, 0.259 min
HTTP response codes:
  code 200 -- 1904
```

过程与分析： 从测试结果来看，线程池模型的性能更优于多线程模型，并且多线程模型在测试时有发生字节数错误的可能。但这并不意味着线程池模型就是完美的，线程池模型的性能与“池子”容量有很大关系。开始在测试线程池模型性能时，我选择的并发度只有50，得到的测试结果让我误以为线程池容量的大小为50时，该模型性能最优。随着测试并发度的增加，线程池模型的效果反而不如多线程模型，开始我还安慰自己，是不是线程池模型中的锁太多了，甚至还有自旋锁，导致各线程间不太“自由”导致的，直到我抱着好奇心去扩大了“池子”的容量，才发现原来是我被骗到了，自己把自己限制住了。经过多次调试，我发现线程池模型的性能与测试并发度和池子容量有密切关系，一个优秀的线程池模型是通过并发度与池子容量调制出来的，不像多线程模型那般简单。经调试，并发度在30-100，池子容量为50时效果最好；并发度为1000，池子容量为100最好。

多线程

- 优点
  - 实现简单，代码结构清晰，每个请求分配给一个独立的线程
- 缺点
  - 每个线程都需要独立的内存空间，创建和销毁线程会占用系统资源
  - 大量线程可能导致内存消耗过大
  - 多线程模型可能引入竞态条件，比如上面的字节数错误的发生可能就是竞态引发的
  - 高并发环境下，多线程间切换会导致上下文切换开销过大

线程池

- 优点
  - 高并发环境下性能更优
  - 可以重复使用已创建的线程，避免了频繁创建和销毁线程的开销，提高了资源利用率
  - 线程池允许限制并发执行的任务数量，防止系统资源被过度占用，有助于控制系统的稳定性
  - 通过限制线程数量，可以减少线程之间的上下文切换开销，提高性能
- 缺点
  - 实现复杂
  - 对互斥锁和条件变量操作不当将严重影响模型性能
  - 模型性能依赖于线程池容量和任务并发度

```
//线程池代码

typedef struct
{
    int hit;
    int fd;
} webparam;

typedef struct staconv
```

```c
{
    pthread_mutex_t mutex;
    pthread_cond_t cond; // 用于阻塞和唤醒线程池中的线程
    int status;          // 表示任务队列状态，1表示有任务，0表示无任务
} staconv;

typedef struct task
{
    struct task *task;            // 指向下一个任务
    void *(*function)(void *arg); // 函数指针
    void *arg;                    // 函数参数指针
} task;

typedef struct taskqueue
{
    pthread_mutex_t mutex; // 互斥读写任务队列
    task *front;
    task *rear;
    staconv *has_jobs; // 根据状态，阻塞线程
    int len;           // 任务个数
} taskqueue;

typedef struct thread
{
    int id;                    // 线程id
    pthread_t pthread;         // 封装的POSIX线程
    struct threadpool *pool; // 与线程池绑定
} thread;

typedef struct threadpool
{
    thread **threads;                 // 线程指针数组
    volatile int num_threads;         // 线程池中线程数量
    volatile int num_working;         // 正在工作的线程数量
    pthread_mutex_t thcount_lock;     // 线程池锁，用于修改上面两个变量
    pthread_cond_t thread_all_idle; // 用于线程消耗的条件变量
    taskqueue queue;                  // 任务队列
    volatile bool is_alive;           // 表示线程池是否还存在
} threadpool;

void push_taskqueue(taskqueue *queue, task *newtask)
{
    task *node = (task *)malloc(sizeof(task));
    if (!node)
    {
        perror("Error allocating memory for new task");
        exit(EXIT_FAILURE);
    }

    // 将任务信息复制到新节点
    node->function = newtask->function;
    node->arg = newtask->arg;
    node->task = NULL;

    // 加锁，修改任务队列
```

```c
    pthread_mutex_lock(&queue->mutex);

    if (queue->len == 0)
    {
        // 队列为空，直接添加新任务
        queue->front = node;
        queue->rear = node;
    }
    else
    {
        // 否则将新任务添加到队列尾部
        queue->rear->task = node;
        queue->rear = node;
    }
    queue->len++;

    // 通知等待在队列上的线程，有新任务到来
    pthread_cond_signal(&queue->has_jobs->cond);

    pthread_mutex_unlock(&queue->mutex);
}

void init_taskqueue(taskqueue *queue)
{
    // 初始化互斥量(互斥访问任务队列)
    pthread_mutex_init(&queue->mutex, NULL);

    // 初始化条件变量(在队列为空时阻塞等待任务的到来)
    queue->has_jobs = (staconv *)malloc(sizeof(staconv));
    pthread_cond_init(&queue->has_jobs->cond, NULL);

    // 初始化任务队列
    queue->front = NULL;
    queue->rear = NULL;
    queue->len = 0;
}

task *take_taskqueue(taskqueue *queue) // 取出队首任务，并在队列中删除该任务
{
    // 加锁，访问任务队列
    pthread_mutex_lock(&queue->mutex);

    // 如果队列为空，等待任务到来
    while (queue->len == 0)
    {
        pthread_cond_wait(&queue->has_jobs->cond, &queue->mutex);
    }

    task *curtask = queue->front; // 取出队首任务
    queue->front = curtask->task; // 更新队列头指针,指向下一个任务
    if (queue->len == 1)          // 如果队列只有一个任务，更新尾指针
    {
        queue->rear = NULL;
    }
    queue->len--;
```

```
    // 解锁互斥量
    pthread_mutex_unlock(&queue->mutex);

    return curtask;
}

void destory_taskqueue(taskqueue *queue)
{

    pthread_mutex_lock(&queue->mutex); // 加锁，访问任务队列

    while (queue->front != NULL) // 释放队列中所有任务节点
    {
        task *curtask = queue->front;
        queue->front = curtask->task;
        free(curtask);
    }
    free(queue->has_jobs); // 释放条件变量

    pthread_mutex_unlock(&queue->mutex);

    pthread_mutex_destroy(&queue->mutex); // 销毁互斥量
}

struct threadpool *initThreadPool(int num_threads)
{
    threadpool *pool;
    pool = (threadpool *)malloc(sizeof(struct threadpool));
    pool->num_threads = 0;
    pool->num_working = 0;
    pool->is_alive = 1; // 一开始忘记置1了，我说怎么能跑但网页就是打不开!
    pthread_mutex_init(&(pool->thcount_lock), NULL); // 初始化互斥量
    pthread_cond_init(&(pool->thread_all_idle), NULL); // 初始化条件变量
    init_taskqueue(&pool->queue); // 初始化任务队列@
    pool->threads = (struct thread **)malloc(num_threads * sizeof(struct thread *)); // 创建线程数组

    int i;
    for (i = 0; i < num_threads; i++)
    {
        create_thread(pool, &pool->threads[i], i); // 在pool->threads[i]前加了个&
    }
    // 每个线程在创建时,运行函数都会进行pool->num_threads++操作
    while (pool->num_threads != num_threads) // 忙等待，等所有进程创建完毕才返回
    {
    }
    return pool;
}
```

```
void addTask2ThreadPool(threadpool *pool, task *curtask)
{
    push_taskqueue(&pool->queue, curtask); // 将任务加入队列@
}

void waitThreadPool(threadpool *pool)
{
    pthread_mutex_lock(&pool->thcount_lock);
    while (pool->queue.len || pool->num_working)
    {
        pthread_cond_wait(&pool->thread_all_idle, &pool->thcount_lock);
    }
    pthread_mutex_unlock(&pool->thcount_lock);
}

void destoryThreadPool(threadpool *pool)
{
    // 等待线程执行完任务队列中的所有任务，并且任务队列为空 @
    waitThreadPool(pool);
    pool->is_alive = 0;                                  // 关闭线程池运行
    pthread_cond_broadcast(&pool->queue.has_jobs->cond); // 唤醒所有等待在任务队列上的线程(让它们检查 is_alive 的状态并退出)

    destory_taskqueue(&pool->queue); // 销毁任务队列 @

    // 销毁线程指针数组，并释放为线程池分配的内存 @
    int i;
    for (i = 0; i < pool->num_threads; i++)
    {
        free(pool->threads[i]);
    }
    free(pool->threads);

    // 销毁线程池的互斥量和条件变量
    pthread_mutex_destroy(&pool->thcount_lock);
    pthread_cond_destroy(&pool->thread_all_idle);

    // 释放线程池结构体内存
    free(pool);
}

int getNumofThreadWorking(threadpool *pool)
{
    return pool->num_working;
}

void *thread_do(struct thread *pthread)
{
    // 设置线程名称
    char thread_name[128] = {0};
    sprintf(thread_name, "thread-pool-%d", pthread->id);

    prctl(PR_SET_NAME, thread_name);

    // 获得(绑定)线程池
```

```
    threadpool *pool = pthread->pool;

    pthread_mutex_lock(&pool->thcount_lock);
    pool->num_threads++; // 对创建线程数量进程统计@
    pthread_mutex_unlock(&pool->thcount_lock);

    while (pool->is_alive)
    {
        // 如果队列中还有任务，则继续运行；否则阻塞 @
        pthread_mutex_lock(&pool->queue.mutex);
        while (pool->queue.len == 0 && pool->is_alive)
        {
            pthread_cond_wait(&pool->queue.has_jobs->cond, &pool-
>queue.mutex);
        }
        pthread_mutex_unlock(&pool->queue.mutex);

        if (pool->is_alive)
        {
            pthread_mutex_lock(&pool->thcount_lock);
            pool->num_working++; // 对工作线程数量进行统计@
            pthread_mutex_unlock(&pool->thcount_lock);

            // 取任务队列队首，并执行
            void *(*func)(void *);
            void *arg;
            task *curtask = take_taskqueue(&pool->queue); // 取出队首任务，并
在队列中删除该任务@（自己实现take_taskqueue）
            if (curtask)
            {
                func = curtask->function;
                arg = curtask->arg;
                func(arg);     // 执行任务
                free(curtask); // 释放任务
            }

            pthread_mutex_lock(&pool->thcount_lock);
            pool->num_working--;
            pthread_mutex_unlock(&pool->thcount_lock);

            // 当工作线程数量为0时，表示任务全部完成，此时运行阻塞在waitThreadPool上
的线程 @
            if (pool->num_working == 0 && pool->queue.len == 0)
                pthread_cond_signal(&pool->thread_all_idle);
        }
    }

    // 线程执行完任务将要退出，需改变线程池中的线程数量 @
    pthread_mutex_unlock(&pool->thcount_lock);
    pool->num_threads--;
    pthread_mutex_unlock(&pool->thcount_lock);

    return NULL;
}
```

```c
int create_thread(struct threadpool *pool, struct thread **pthread, int id)
{
    *pthread = (struct thread *)malloc(sizeof(struct thread));
    if (*pthread == NULL)
    {
        perror("create_thread(): Could not allocate memory for thread\n");
        return -1;
    }

    // 设置该线程的属性
    (*pthread)->pool = pool;
    (*pthread)->id = id;

    pthread_create(&(*pthread)->pthread, NULL, (void *)thread_do, (*pthread)); // 创建线程
    pthread_detach((*pthread)->pthread);                                          // 线程分离
    return 0;
}

void logger(int type, char *s1, char *s2, int socket_fd)
{...}

void *web(void *data)
{...}

int main(int argc, char **argv)
{
    .
    .
    .

    threadpool *pool = initThreadPool(100);
    for (hit = 1;; hit++)
    {
        length = sizeof(cli_addr);
        if ((socketfd = accept(listenfd, (struct sockaddr *)&cli_addr, &length)) < 0)
            logger(ERROR, "system call", "accept", 0);

        task *curtask = (task *)malloc(sizeof(task));
        curtask->task = NULL;
        curtask->function = web;
        webparam *param = malloc(sizeof(webparam));
        param->hit = hit;
        param->fd = socketfd;
        curtask->arg = (void *)param;
        addTask2ThreadPool(pool, curtask);
    }
    destoryThreadPool(pool);
    return 0;
}
```

实验五