expt.4 2021011615 田天成 计算机21-3

题目1.

添补相应的程序代码到上面函数中“……”位置处。

```c
int threadpool_destroy(struct threadpool *pool)
{
    assert(pool != NULL);
    pthread_mutex_lock(&(pool->mutex));
    if (pool->queue_close || pool->pool_close)   //线程池已经退出了，就直接返回
    {
        pthread_mutex_unlock(&(pool->mutex));
        return -1;
    }

    pool->queue_close = 1;        //置队列关闭标志
    while (pool->queue_cur_num != 0)
    {
        pthread_cond_wait(&(pool->queue_empty), &(pool->mutex));  //等待队列为空
    }

    pool->pool_close = 1;      //置线程池关闭标志
    pthread_mutex_unlock(&(pool->mutex));
    pthread_cond_broadcast(&(pool->queue_not_empty));  //唤醒线程池中正在阻塞的线程
    pthread_cond_broadcast(&(pool->queue_not_full));   //唤醒添加任务的threadpool_add_job函数
    int i;
    for (i = 0; i < pool->thread_num; ++i)
    {
        pthread_join(pool->pthreads[i], NULL);    //等待线程池的所有线程执行完毕
    }

    pthread_mutex_destroy(&(pool->mutex));          //清理资源
    pthread_cond_destroy(&(pool->queue_empty));
    pthread_cond_destroy(&(pool->queue_not_empty));
    pthread_cond_destroy(&(pool->queue_not_full));
    free(pool->pthreads);
    struct job *p;
    while (pool->head != NULL)
    {
        p = pool->head;
        pool->head = p->next;
        free(p);
    }
    free(pool);
    return 0;
}
```

```

```

```
/*线程运行的逻辑函数*/
void* threadpool_function(void* arg)
{
    struct threadpool *pool = (struct threadpool*)arg;
    struct job *pjob = NULL;
    while (1)  //死循环
    {
        pthread_mutex_lock(&(pool->mutex));
        while ((pool->queue_cur_num == 0) && !pool->pool_close)   //队列为空时，就等待队列非空
        {
            pthread_cond_wait(&(pool->queue_not_empty), &(pool->mutex));
        }
        if (pool->pool_close)   //线程池关闭，线程就退出
        {
            pthread_mutex_unlock(&(pool->mutex));
            pthread_exit(NULL);
        }
        pool->queue_cur_num--;
        pjob = pool->head;
        if (pool->queue_cur_num == 0)
        {
            pool->head = pool->tail = NULL;
        }
        else
        {
            pool->head = pjob->next;
        }
        if (pool->queue_cur_num == 0)
        {
            pthread_cond_signal(&(pool->queue_empty));        //队列为空，就可以通知threadpool_destroy函数，销毁线程函数
        }
        if (pool->queue_cur_num == pool->queue_max_num - 1)
        {
            pthread_cond_broadcast(&(pool->queue_not_full));  //队列非满，就可以通知threadpool_add_job函数，添加新任务
        }
        pthread_mutex_unlock(&(pool->mutex));

        (*(pjob->callback_function))(pjob->arg);   //线程真正要做的工作，回调函数的调用
        free(pjob);
        pjob = NULL;
    }
}
```

题目2.

完成函数push_taskqueue，take_taskqueue，init_taskqueue和 destory_taskqueue。

```
void push_taskquene(taskqueue *queue, task *curtask)
{
```

```
    pthread_mutex_lock(&queue->mutex);
    if (queue->len == 0)
    {
        queue->front = curtask;
        queue->rear = curtask;
    }
    else
    {
        queue->rear->next = curtask;
        queue->rear = curtask;
    }
    queue->len++;
    pthread_mutex_unlock(&queue->mutex);
}
task *take_taskqueue(taskqueue *queue)
{
    pthread_mutex_lock(&queue->mutex);
    task *curtask;
    if (queue->len == 0)
    {
        curtask = NULL;
    }
    else
    {
        curtask = queue->front;
        queue->front = curtask->next;
        queue->len--;
    }
    pthread_mutex_unlock(&queue->mutex);
    return curtask;
}
void init_taskqueue(taskqueue *queue)
{
    queue->front = NULL;
    queue->rear = NULL;
    queue->len = 0;
    pthread_mutex_init(&queue->mutex, NULL);
}
void destory_taskqueue(taskqueue *queue)
{
    pthread_mutex_destroy(&queue->mutex);
}
```

题目3.

添加必要的程序代码，以最终完成线程池。

```
// threadpool
#include "threadpool.h"
#include <assert.h>

struct threadpool* threadpool_init(int thread_num, int queue_max_num)
{
    struct threadpool *pool = NULL;
    do
```

```
    {
        pool = malloc(sizeof(struct threadpool));
        if (NULL == pool)
        {
            printf("failed to malloc threadpool!\n");
            break;
        }
        pool->thread_num = thread_num;
        pool->queue_max_num = queue_max_num;
        pool->queue_cur_num = 0;
        pool->head = NULL;
        pool->tail = NULL;
        if (pthread_mutex_init(&(pool->mutex), NULL))
        {
            printf("failed to init mutex!\n");
            break;
        }
        if (pthread_cond_init(&(pool->queue_empty), NULL))
        {
            printf("failed to init queue_empty!\n");
            break;
        }
        if (pthread_cond_init(&(pool->queue_not_empty), NULL))
        {
            printf("failed to init queue_not_empty!\n");
            break;
        }
        if (pthread_cond_init(&(pool->queue_not_full), NULL))
        {
            printf("failed to init queue_not_full!\n");
            break;
        }
        pool->pthreads = malloc(sizeof(pthread_t) * thread_num);
        if (NULL == pool->pthreads)
        {
            printf("failed to malloc pthreads!\n");
            break;
        }
        pool->queue_close = 0;
        pool->pool_close = 0;
        int i;
        for (i = 0; i < pool->thread_num; ++i)
        {
            pthread_create(&(pool->pthreads[i]), NULL, threadpool_function, (void *)pool);
        }

        return pool;
    } while (0);

    return NULL;
}

int threadpool_add_job(struct threadpool* pool, void* (*callback_function)(void *arg), void *arg)
```

```
{
    assert(pool != NULL);
    assert(callback_function != NULL);
    assert(arg != NULL);

    pthread_mutex_lock(&(pool->mutex));
    while ((pool->queue_cur_num == pool->queue_max_num) && !(pool->queue_close || pool->pool_close))
    {
        pthread_cond_wait(&(pool->queue_not_full), &(pool->mutex));   //队列满的时候就等待
    }
    if (pool->queue_close || pool->pool_close)    //队列关闭或者线程池关闭就退出
    {
        pthread_mutex_unlock(&(pool->mutex));
        return -1;
    }
    struct job *pjob =(struct job*) malloc(sizeof(struct job));
    if (NULL == pjob)
    {
        pthread_mutex_unlock(&(pool->mutex));
        return -1;
    }
    pjob->callback_function = callback_function;
    pjob->arg = arg;
    pjob->next = NULL;
    if (pool->head == NULL)
    {
        pool->head = pool->tail = pjob;
        pthread_cond_broadcast(&(pool->queue_not_empty));  //队列空的时候，有任务来时就通知线程池中的线程：队列非空
    }
    else
    {
        pool->tail->next = pjob;
        pool->tail = pjob;
    }
    pool->queue_cur_num++;
    pthread_mutex_unlock(&(pool->mutex));
    return 0;
}

void* threadpool_function(void* arg)
{
    struct threadpool *pool = (struct threadpool*)arg;
    struct job *pjob = NULL;
    while (1)  //死循环
    {
        pthread_mutex_lock(&(pool->mutex));
        while ((pool->queue_cur_num == 0) && !pool->pool_close)   //队列为空时，就等待队列非空
        {
            pthread_cond_wait(&(pool->queue_not_empty), &(pool->mutex));
        }
```

```
        if (pool->pool_close)   //线程池关闭，线程就退出
        {
            pthread_mutex_unlock(&(pool->mutex));
            pthread_exit(NULL);
        }
        pool->queue_cur_num--;
        pjob = pool->head;
        if (pool->queue_cur_num == 0)
        {
            pool->head = pool->tail = NULL;
        }
        else
        {
            pool->head = pjob->next;
        }
        if (pool->queue_cur_num == 0)
        {
            pthread_cond_signal(&(pool->queue_empty));        //队列为空，就可以通知threadpool_destroy函数，销毁线程函数
        }
        if (pool->queue_cur_num == pool->queue_max_num - 1)
        {
            pthread_cond_broadcast(&(pool->queue_not_full));  //队列非满，就可以通知threadpool_add_job函数，添加新任务
        }
        pthread_mutex_unlock(&(pool->mutex));

        (*(pjob->callback_function))(pjob->arg);   //线程真正要做的工作，回调函数的调用
        free(pjob);
        pjob = NULL;
    }
}
int threadpool_destroy(struct threadpool *pool)
{
    assert(pool != NULL);
    pthread_mutex_lock(&(pool->mutex));
    if (pool->queue_close || pool->pool_close)   //线程池已经退出了，就直接返回
    {
        pthread_mutex_unlock(&(pool->mutex));
        return -1;
    }

    pool->queue_close = 1;        //置队列关闭标志
    while (pool->queue_cur_num != 0)
    {
        pthread_cond_wait(&(pool->queue_empty), &(pool->mutex));  //等待队列为空
    }

    pool->pool_close = 1;      //置线程池关闭标志
    pthread_mutex_unlock(&(pool->mutex));
    pthread_cond_broadcast(&(pool->queue_not_empty));  //唤醒线程池中正在阻塞的线程
```

```
    pthread_cond_broadcast(&(pool->queue_not_full));    //唤醒添加任务的threadpool_add_job函数
    int i;
    for (i = 0; i < pool->thread_num; ++i)
    {
        pthread_join(pool->pthreads[i], NULL);     //等待线程池的所有线程执行完毕
    }

    pthread_mutex_destroy(&(pool->mutex));          //清理资源
    pthread_cond_destroy(&(pool->queue_empty));
    pthread_cond_destroy(&(pool->queue_not_empty));
    pthread_cond_destroy(&(pool->queue_not_full));
    free(pool->pthreads);
    struct job *p;
    while (pool->head != NULL)
    {
        p = pool->head;
        pool->head = p->next;
        free(p);
    }
    free(pool);
    return 0;
}

```

题目4.

利用实现的线程池，替换实验3中Web服务的多线程模型。

```
//webserver.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include "threadpool.h"

#define BUFFER_SIZE 1024
#define THREAD_NUM 4

void handle_request(void *arg) {
    int client_socket = *((int *)arg);
    char buffer[BUFFER_SIZE];
    ssize_t bytes_received;

    // Receive the HTTP request
    bytes_received = recv(client_socket, buffer, sizeof(buffer), 0);
    if (bytes_received < 0) {
        perror("Error receiving data from client");
        close(client_socket);
        return;
    }
```

```c
    if (strstr(buffer, "GET") == NULL) {
        // We only support GET requests in this example
        close(client_socket);
        return;
    }

    // Extract the requested file path from the request
    char file_path[100];
    sscanf(buffer, "GET /%s", file_path);

    // If the requested path is empty, set it to "index.html"
    if (strlen(file_path) == 0) {
        strcpy(file_path, "index.html");
    }

    // Open and read the requested file
    FILE *file = fopen(file_path, "rb");
    if (file == NULL) {
        // If the file is not found, send a 404 response
        const char *not_found_response = "HTTP/1.1 404 Not Found\r\n\r\n";
        send(client_socket, not_found_response, strlen(not_found_response), 0);
    } else {
        // If the file is found, send a 200 OK response followed by the file content
        const char *ok_response = "HTTP/1.1 200 OK\r\n\r\n";
        send(client_socket, ok_response, strlen(ok_response), 0);

        // Read and send the file content
        size_t bytes_read;
        while ((bytes_read = fread(buffer, 1, sizeof(buffer), file)) > 0) {
            send(client_socket, buffer, bytes_read, 0);
        }

        fclose(file);
    }

    // Close the client socket
    close(client_socket);
}

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s [port] [file_directory]\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    // Extract command line arguments
    int port = atoi(argv[1]);
    const char *file_directory = argv[2];

    // Create a thread pool
    struct threadpool *pool = threadpool_init(THREAD_NUM, 10);

    // Create a socket
```

```c
    int server_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (server_socket < 0) {
        perror("Error creating socket");
        exit(EXIT_FAILURE);
    }

    // Set up server address
    struct sockaddr_in server_addr;
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(port);
    server_addr.sin_addr.s_addr = INADDR_ANY;

    // Bind the socket
    if (bind(server_socket, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
        perror("Error binding socket");
        exit(EXIT_FAILURE);
    }

    // Listen for incoming connections
    if (listen(server_socket, 10) < 0) {
        perror("Error listening for connections");
        exit(EXIT_FAILURE);
    }

    printf("Web server listening on port %d...\n", port);

    while (1) {
        // Accept a connection
        int client_socket = accept(server_socket, NULL, NULL);
        if (client_socket < 0) {
            perror("Error accepting connection");
            continue;
        }

        // Enqueue the client socket to the thread pool for processing
        int *client_socket_ptr = (int *)malloc(sizeof(int));
        *client_socket_ptr = client_socket;
        threadpool_add_job(pool, handle_request, (void *)client_socket_ptr);
    }

    // Close the server socket and destroy the thread pool
    close(server_socket);
    threadpool_destroy(pool);

    return 0;
}
```

题目5.

调整线程池中线程个数参数，以达到Web服务并发性能最优。利用 http_load及其它性能参数，分析和对比多线程模型与线程池模型在Web服务进程中的优点和缺点。

```
num_threads=1
➜  http_load-09Mar2016 ./http_load -parallel 10 -fetches 1000 urls.txt
1000 fetches, 10 max parallel, 260000 bytes, in 0.052216 seconds
260 mean bytes/connection
19151.2 fetches/sec, 4.97932e+06 bytes/sec
msecs/connect: 0.087026 mean, 0.7 max, 0.014 min
msecs/first-response: 0.22475 mean, 3.743 max, 0.042 min
HTTP response codes:
  code 200 -- 1000

```

```
num_threads=2
➜  http_load-09Mar2016 ./http_load -parallel 10 -fetches 1000 urls.txt
1000 fetches, 10 max parallel, 260000 bytes, in 0.076752 seconds
260 mean bytes/connection
13029 fetches/sec, 3.38753e+06 bytes/sec
msecs/connect: 0.124987 mean, 3.992 max, 0.014 min
msecs/first-response: 0.289556 mean, 2.323 max, 0.045 min
HTTP response codes:
  code 200 -- 1000
➜  http_load-09Mar20
```

```
num_threads=3
➜  http_load-09Mar2016 ./http_load -parallel 10 -fetches 1000 urls.txt
1000 fetches, 10 max parallel, 260000 bytes, in 0.071381 seconds
260 mean bytes/connection
14009.3 fetches/sec, 3.64243e+06 bytes/sec
msecs/connect: 0.134086 mean, 1.116 max, 0.013 min
msecs/first-response: 0.323018 mean, 1.884 max, 0.072 min
HTTP response codes:
  code 200 -- 1000

```

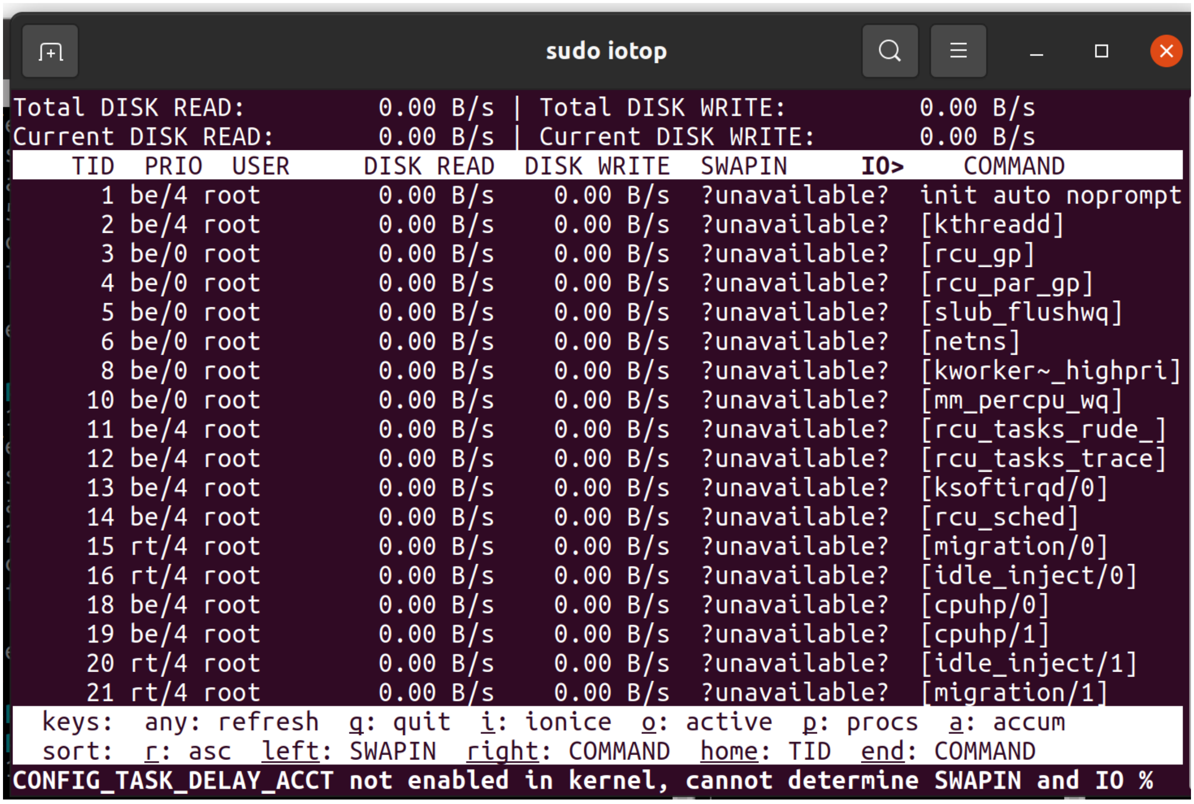
```
num_threads=4
➜  http_load-09Mar2016 ./http_load -parallel 10 -fetches 1000 urls.txt
1000 fetches, 10 max parallel, 260000 bytes, in 0.078894 seconds
260 mean bytes/connection
12675.2 fetches/sec, 3.29556e+06 bytes/sec
msecs/connect: 0.142843 mean, 2.582 max, 0.013 min
msecs/first-response: 0.303359 mean, 2.441 max, 0.078 min
HTTP response codes:
  code 200 -- 1000

```

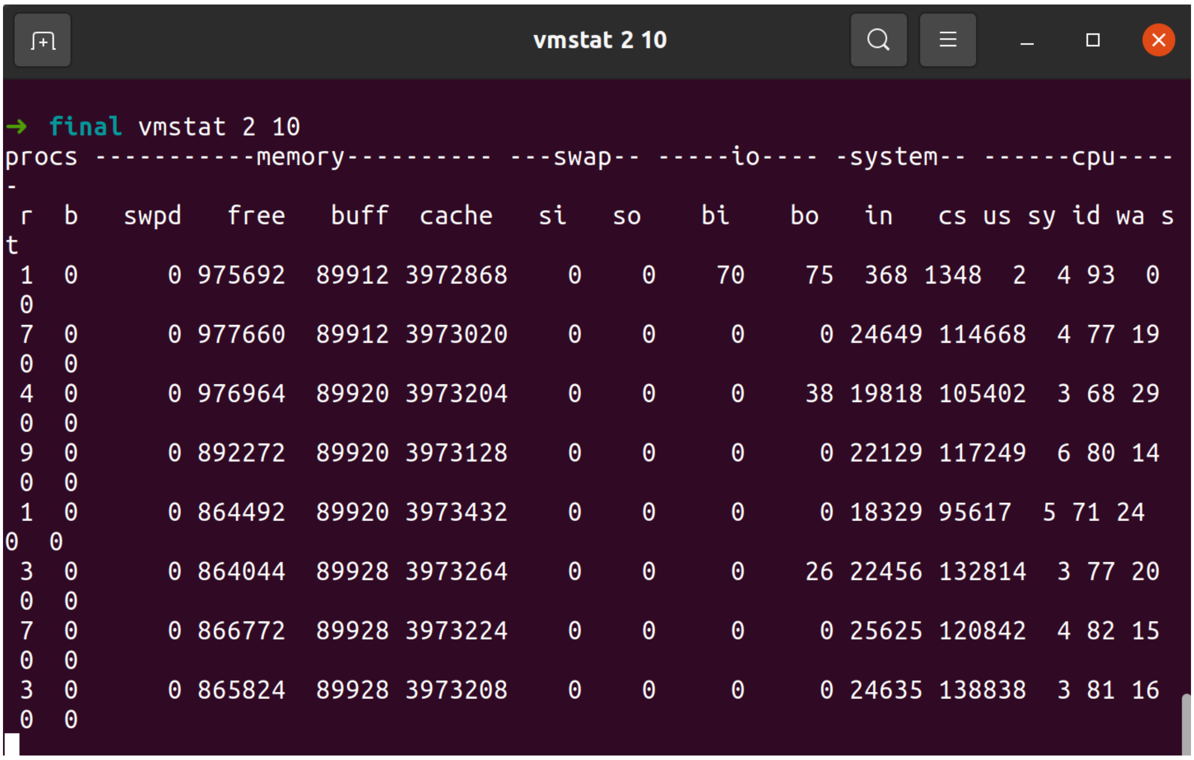
```
num_threads=5
➜  http_load-09Mar2016 ./http_load -parallel 10 -fetches 1000 urls.txt
1000 fetches, 10 max parallel, 260000 bytes, in 0.071919 seconds
260 mean bytes/connection
13904.5 fetches/sec, 3.61518e+06 bytes/sec
msecs/connect: 0.129232 mean, 0.681 max, 0.012 min
msecs/first-response: 0.298131 mean, 6.035 max, 0.046 min
HTTP response codes:
  code 200 -- 1000

```

```
num_threads=6
➜  http_load-09Mar2016 ./http_load -parallel 10 -fetches 1000 urls.txt
1000 fetches, 10 max parallel, 260000 bytes, in 0.075317 seconds
260 mean bytes/connection
13277.2 fetches/sec, 3.45208e+06 bytes/sec
msecs/connect: 0.144974 mean, 1.523 max, 0.013 min
msecs/first-response: 0.283606 mean, 2.968 max, 0.037 min
HTTP response codes:
  code 200 -- 1000

```

从数据来看，`` `num_threads=1 ``时，运行的性能最佳，在单位时间内能够处理最多的请求。

对这两种模型的优点和缺点的分析和对比：





多线程模型:

优点:

1. 简单直观： 多线程模型相对较简单，易于理解和实现。可以根据需要创建和管理线程，使其更具灵活性。
2. 适用于短任务： 对于一些相对短暂的任务，创建线程的开销相对较小。

缺点:

1. 资源开销： 创建和销毁线程会带来额外的系统资源开销，包括内存和处理器时间。
2. 可伸缩性： 在高负载情况下，线程的数量可能会增加，但随着线程数量的增加，管理和调度线程的开销也会增加，可能导致性能瓶颈。
3. 死锁和竞争条件： 多线程模型容易引发死锁和竞争条件，需要谨慎处理共享资源。

线程池模型:

优点:

1. 资源重用： 线程池重用线程，减少了线程创建和销毁的开销。
2. 控制并发度： 线程池可以限制并发执行的线程数量，防止系统资源被过度占用。
3. 任务队列： 可以使用任务队列管理待执行的任务，提高系统的可管理性。
4. 线程生命周期管理： 线程池提供了对线程生命周期的管理，包括线程的创建、销毁和异常处理。

缺点:

1. 复杂性： 相对于简单的多线程模型，线程池模型的实现可能更为复杂。
2. 不适合短任务： 对于一些短暂的任务，线程池的维护可能会带来一些额外的开销。