

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。

```
1  int threadpool_destroy(struct threadpool *pool)
2  {
3      assert(pool != NULL);
4      pthread_mutex_lock(&(pool->mutex));
5      if (pool->queue_close || pool->pool_close) //线程池已经退出了，就直接返回
6      {
7          pthread_mutex_unlock(&(pool->mutex));
8          return -1;
9      }
10
11     pool->queue_close = 1; //置队列关闭标志
12     while (pool->queue_cur_num != 0)
13     {
14         pthread_cond_wait(&(pool->queue_empty), &(pool->mutex)); //等待队列为
空
15     }
16
17     pool->pool_close = 1; //置线程池关闭标志
18     pthread_mutex_unlock(&(pool->mutex));
19     pthread_cond_broadcast(&(pool->queue_not_empty)); //唤醒线程池中正在阻塞的
线程
20     pthread_cond_broadcast(&(pool->queue_not_full)); //唤醒添加任务的
threadpool_add_job函数
21     int i;
22     for (i = 0; i < pool->thread_num; ++i)
23     {
24         pthread_join(pool->pthreads[i], NULL); //等待线程池的所有线程执行完毕
25     }
26
27     pthread_mutex_destroy(&(pool->mutex)); //清理资源
28     pthread_cond_destroy(&(pool->queue_empty));
29     pthread_cond_destroy(&(pool->queue_not_empty));
30     pthread_cond_destroy(&(pool->queue_not_full));
31     free(pool->pthreads);
32     struct job *p;
33     while (pool->head != NULL)
34     {
35         p = pool->head;
36         pool->head = p->next;
37         free(p);
38     }
39     free(pool);
40     return 0;
41 }
```

```

2  /*线程运行的逻辑函数*/
3  void* threadpool_function(void* arg)
4  {
5      struct threadpool *pool = (struct threadpool*)arg;
6      struct job *pjob = NULL;
7      while (1) //死循环
8      {
9          pthread_mutex_lock(&(pool->mutex));
10         while ((pool->queue_cur_num == 0) && !pool->pool_close) //队列为空
            时，就等待队列非空
11         {
12             pthread_cond_wait(&(pool->queue_not_empty), &(pool->mutex));
13         }
14         if (pool->pool_close) //线程池关闭，线程就退出
15         {
16             pthread_mutex_unlock(&(pool->mutex));
17             pthread_exit(NULL);
18         }
19         pool->queue_cur_num--;
20         pjob = pool->head;
21         if (pool->queue_cur_num == 0)
22         {
23             pool->head = pool->tail = NULL;
24         }
25         else
26         {
27             pool->head = pjob->next;
28         }
29         if (pool->queue_cur_num == 0)
30         {
31             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
32         }
33         if (pool->queue_cur_num == pool->queue_max_num - 1)
34         {
35             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
36         }
37         pthread_mutex_unlock(&(pool->mutex));
38
39         (*(pjob->callback_function))(pjob->arg); //线程真正要做的工作，回调函数
            的调用
40         free(pjob);
41         pjob = NULL;
42     }
43 }

```

题目2.

完成函数push_taskqueue, take_taskqueue, init_taskqueue和destory_taskqueue。

```

1  void push_taskquene(taskqueue *queue, task *curtask)
2  {

```

```

3     pthread_mutex_lock(&queue->mutex);
4     if (queue->len == 0)
5     {
6         queue->front = curtask;
7         queue->rear = curtask;
8     }
9     else
10    {
11        queue->rear->next = curtask;
12        queue->rear = curtask;
13    }
14    queue->len++;
15    pthread_mutex_unlock(&queue->mutex);
16 }
17 task *take_taskqueue(taskqueue *queue)
18 {
19     pthread_mutex_lock(&queue->mutex);
20     task *curtask;
21     if (queue->len == 0)
22     {
23         curtask = NULL;
24     }
25     else
26     {
27         curtask = queue->front;
28         queue->front = curtask->next;
29         queue->len--;
30     }
31     pthread_mutex_unlock(&queue->mutex);
32     return curtask;
33 }
34 void init_taskqueue(taskqueue *queue)
35 {
36     queue->front = NULL;
37     queue->rear = NULL;
38     queue->len = 0;
39     pthread_mutex_init(&queue->mutex, NULL);
40 }
41 void destroy_taskqueue(taskqueue *queue)
42 {
43     pthread_mutex_destroy(&queue->mutex);
44 }

```

题目3.

添加必要的程序代码，以最终完成线程池。

```

1 // threadpool
2 #include "threadpool.h"
3 #include <assert.h>
4
5 struct threadpool* threadpool_init(int thread_num, int queue_max_num)
6 {
7     struct threadpool *pool = NULL;
8     do

```

```

9      {
10         pool = malloc(sizeof(struct threadpool));
11         if (NULL == pool)
12         {
13             printf("failed to malloc threadpool!\n");
14             break;
15         }
16         pool->thread_num = thread_num;
17         pool->queue_max_num = queue_max_num;
18         pool->queue_cur_num = 0;
19         pool->head = NULL;
20         pool->tail = NULL;
21         if (pthread_mutex_init(&(pool->mutex), NULL))
22         {
23             printf("failed to init mutex!\n");
24             break;
25         }
26         if (pthread_cond_init(&(pool->queue_empty), NULL))
27         {
28             printf("failed to init queue_empty!\n");
29             break;
30         }
31         if (pthread_cond_init(&(pool->queue_not_empty), NULL))
32         {
33             printf("failed to init queue_not_empty!\n");
34             break;
35         }
36         if (pthread_cond_init(&(pool->queue_not_full), NULL))
37         {
38             printf("failed to init queue_not_full!\n");
39             break;
40         }
41         pool->pthreads = malloc(sizeof(pthread_t) * thread_num);
42         if (NULL == pool->pthreads)
43         {
44             printf("failed to malloc pthreads!\n");
45             break;
46         }
47         pool->queue_close = 0;
48         pool->pool_close = 0;
49         int i;
50         for (i = 0; i < pool->thread_num; ++i)
51         {
52             pthread_create(&(pool->pthreads[i]), NULL, threadpool_function,
53 (void *)pool);
54         }
55         return pool;
56     } while (0);
57
58     return NULL;
59 }
60
61 int threadpool_add_job(struct threadpool* pool, void* (*callback_function)
(void *arg), void *arg)

```

```

62 {
63     assert(pool != NULL);
64     assert(callback_function != NULL);
65     assert(arg != NULL);
66
67     pthread_mutex_lock(&(pool->mutex));
68     while ((pool->queue_cur_num == pool->queue_max_num) && !(pool->
    >queue_close || pool->pool_close))
69     {
70         pthread_cond_wait(&(pool->queue_not_full), &(pool->mutex)); //队
        列满的时候就等待
71     }
72     if (pool->queue_close || pool->pool_close) //队列关闭或者线程池关闭就退
        出
73     {
74         pthread_mutex_unlock(&(pool->mutex));
75         return -1;
76     }
77     struct job *pjob =(struct job*) malloc(sizeof(struct job));
78     if (NULL == pjob)
79     {
80         pthread_mutex_unlock(&(pool->mutex));
81         return -1;
82     }
83     pjob->callback_function = callback_function;
84     pjob->arg = arg;
85     pjob->next = NULL;
86     if (pool->head == NULL)
87     {
88         pool->head = pool->tail = pjob;
89         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
90     }
91     else
92     {
93         pool->tail->next = pjob;
94         pool->tail = pjob;
95     }
96     pool->queue_cur_num++;
97     pthread_mutex_unlock(&(pool->mutex));
98     return 0;
99 }
100
101 void* threadpool_function(void* arg)
102 {
103     struct threadpool *pool = (struct threadpool*)arg;
104     struct job *pjob = NULL;
105     while (1) //死循环
106     {
107         pthread_mutex_lock(&(pool->mutex));
108         while ((pool->queue_cur_num == 0) && !pool->pool_close) //队列为空
            时，就等待队列非空
109         {
110             pthread_cond_wait(&(pool->queue_not_empty), &(pool->mutex));
111         }

```

```

112     if (pool->pool_close)    //线程池关闭，线程就退出
113     {
114         pthread_mutex_unlock(&(pool->mutex));
115         pthread_exit(NULL);
116     }
117     pool->queue_cur_num--;
118     pjob = pool->head;
119     if (pool->queue_cur_num == 0)
120     {
121         pool->head = pool->tail = NULL;
122     }
123     else
124     {
125         pool->head = pjob->next;
126     }
127     if (pool->queue_cur_num == 0)
128     {
129         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
130     }
131     if (pool->queue_cur_num == pool->queue_max_num - 1)
132     {
133         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
134     }
135     pthread_mutex_unlock(&(pool->mutex));
136
137     (*(pjob->callback_function))(pjob->arg);    //线程真正要做的工作，回调函
数的调用
138     free(pjob);
139     pjob = NULL;
140 }
141 }
142 int threadpool_destroy(struct threadpool *pool)
143 {
144     assert(pool != NULL);
145     pthread_mutex_lock(&(pool->mutex));
146     if (pool->queue_close || pool->pool_close)    //线程池已经退出了，就直接返回
147     {
148         pthread_mutex_unlock(&(pool->mutex));
149         return -1;
150     }
151
152     pool->queue_close = 1;    //置队列关闭标志
153     while (pool->queue_cur_num != 0)
154     {
155         pthread_cond_wait(&(pool->queue_empty), &(pool->mutex));    //等待队列
为空
156     }
157
158     pool->pool_close = 1;    //置线程池关闭标志
159     pthread_mutex_unlock(&(pool->mutex));
160     pthread_cond_broadcast(&(pool->queue_not_empty));    //唤醒线程池中正在阻塞
的线程

```

```

161 pthread_cond_broadcast(&(pool->queue_not_full)); //唤醒添加任务的
threadpool_add_job函数
162 int i;
163 for (i = 0; i < pool->thread_num; ++i)
164 {
165     pthread_join(pool->pthreads[i], NULL); //等待线程池的所有线程执行完
毕
166 }
167
168 pthread_mutex_destroy(&(pool->mutex)); //清理资源
169 pthread_cond_destroy(&(pool->queue_empty));
170 pthread_cond_destroy(&(pool->queue_not_empty));
171 pthread_cond_destroy(&(pool->queue_not_full));
172 free(pool->pthreads);
173 struct job *p;
174 while (pool->head != NULL)
175 {
176     p = pool->head;
177     pool->head = p->next;
178     free(p);
179 }
180 free(pool);
181 return 0;
182 }
183

```

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。

```

1 //webserver.c
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <unistd.h>
6 #include <arpa/inet.h>
7 #include <sys/socket.h>
8 #include <netinet/in.h>
9 #include "threadpool.h"
10
11 #define BUFFER_SIZE 1024
12 #define THREAD_NUM 4
13
14 void handle_request(void *arg) {
15     int client_socket = *((int *)arg);
16     char buffer[BUFFER_SIZE];
17     ssize_t bytes_received;
18
19     // Receive the HTTP request
20     bytes_received = recv(client_socket, buffer, sizeof(buffer), 0);
21     if (bytes_received < 0) {
22         perror("Error receiving data from client");
23         close(client_socket);
24         return;
25     }
26 }
27

```

```

26     if (strstr(buffer, "GET") == NULL) {
27         // We only support GET requests in this example
28         close(client_socket);
29         return;
30     }
31
32     // Extract the requested file path from the request
33     char file_path[100];
34     sscanf(buffer, "GET /%s", file_path);
35
36     // If the requested path is empty, set it to "index.html"
37     if (strlen(file_path) == 0) {
38         strcpy(file_path, "index.html");
39     }
40
41     // Open and read the requested file
42     FILE *file = fopen(file_path, "rb");
43     if (file == NULL) {
44         // If the file is not found, send a 404 response
45         const char *not_found_response = "HTTP/1.1 404 Not Found\r\n\r\n";
46         send(client_socket, not_found_response, strlen(not_found_response),
0);
47     } else {
48         // If the file is found, send a 200 OK response followed by the
file content
49         const char *ok_response = "HTTP/1.1 200 OK\r\n\r\n";
50         send(client_socket, ok_response, strlen(ok_response), 0);
51
52         // Read and send the file content
53         size_t bytes_read;
54         while ((bytes_read = fread(buffer, 1, sizeof(buffer), file)) > 0) {
55             send(client_socket, buffer, bytes_read, 0);
56         }
57
58         fclose(file);
59     }
60
61     // Close the client socket
62     close(client_socket);
63 }
64
65 int main(int argc, char *argv[]) {
66     if (argc != 3) {
67         fprintf(stderr, "Usage: %s [port] [file_directory]\n", argv[0]);
68         exit(EXIT_FAILURE);
69     }
70
71     // Extract command line arguments
72     int port = atoi(argv[1]);
73     const char *file_directory = argv[2];
74
75     // Create a thread pool
76     struct threadpool *pool = threadpool_init(THREAD_NUM, 10);
77
78     // Create a socket

```



```

79     int server_socket = socket(AF_INET, SOCK_STREAM, 0);
80     if (server_socket < 0) {
81         perror("Error creating socket");
82         exit(EXIT_FAILURE);
83     }
84
85     // Set up server address
86     struct sockaddr_in server_addr;
87     server_addr.sin_family = AF_INET;
88     server_addr.sin_port = htons(port);
89     server_addr.sin_addr.s_addr = INADDR_ANY;
90
91     // Bind the socket
92     if (bind(server_socket, (struct sockaddr *)&server_addr,
93 sizeof(server_addr)) < 0) {
94         perror("Error binding socket");
95         exit(EXIT_FAILURE);
96     }
97
98     // Listen for incoming connections
99     if (listen(server_socket, 10) < 0) {
100         perror("Error listening for connections");
101         exit(EXIT_FAILURE);
102     }
103
104     printf("Web server listening on port %d...\n", port);
105
106     while (1) {
107         // Accept a connection
108         int client_socket = accept(server_socket, NULL, NULL);
109         if (client_socket < 0) {
110             perror("Error accepting connection");
111             continue;
112         }
113
114         // Enqueue the client socket to the thread pool for processing
115         int *client_socket_ptr = (int *)malloc(sizeof(int));
116         *client_socket_ptr = client_socket;
117         threadpool_add_job(pool, handle_request, (void
118 *)client_socket_ptr);
119     }
120
121     // Close the server socket and destroy the thread pool
122     close(server_socket);
123     threadpool_destroy(pool);
124
125     return 0;
126 }

```

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。

```
1 num_threads=1
2 → http_load-09Mar2016 ./http_load -parallel 10 -fetches 1000 urls.txt
3 1000 fetches, 10 max parallel, 260000 bytes, in 0.052216 seconds
4 260 mean bytes/connection
5 19151.2 fetches/sec, 4.97932e+06 bytes/sec
6 msecs/connect: 0.087026 mean, 0.7 max, 0.014 min
7 msecs/first-response: 0.22475 mean, 3.743 max, 0.042 min
8 HTTP response codes:
9   code 200 -- 1000
10
```

```
1 num_threads=2
2 → http_load-09Mar2016 ./http_load -parallel 10 -fetches 1000 urls.txt
3 1000 fetches, 10 max parallel, 260000 bytes, in 0.076752 seconds
4 260 mean bytes/connection
5 13029 fetches/sec, 3.38753e+06 bytes/sec
6 msecs/connect: 0.124987 mean, 3.992 max, 0.014 min
7 msecs/first-response: 0.289556 mean, 2.323 max, 0.045 min
8 HTTP response codes:
9   code 200 -- 1000
10 → http_load-09Mar20
```

```
1 num_threads=3
2 → http_load-09Mar2016 ./http_load -parallel 10 -fetches 1000 urls.txt
3 1000 fetches, 10 max parallel, 260000 bytes, in 0.071381 seconds
4 260 mean bytes/connection
5 14009.3 fetches/sec, 3.64243e+06 bytes/sec
6 msecs/connect: 0.134086 mean, 1.116 max, 0.013 min
7 msecs/first-response: 0.323018 mean, 1.884 max, 0.072 min
8 HTTP response codes:
9   code 200 -- 1000
10
```

```
1 num_threads=4
2 → http_load-09Mar2016 ./http_load -parallel 10 -fetches 1000 urls.txt
3 1000 fetches, 10 max parallel, 260000 bytes, in 0.078894 seconds
4 260 mean bytes/connection
5 12675.2 fetches/sec, 3.29556e+06 bytes/sec
6 msecs/connect: 0.142843 mean, 2.582 max, 0.013 min
7 msecs/first-response: 0.303359 mean, 2.441 max, 0.078 min
8 HTTP response codes:
9   code 200 -- 1000
10
```

```

1 num_threads=5
2 → http_load-09Mar2016 ./http_load -parallel 10 -fetches 1000 urls.txt
3 1000 fetches, 10 max parallel, 260000 bytes, in 0.071919 seconds
4 260 mean bytes/connection
5 13904.5 fetches/sec, 3.61518e+06 bytes/sec
6 msecs/connect: 0.129232 mean, 0.681 max, 0.012 min
7 msecs/first-response: 0.298131 mean, 6.035 max, 0.046 min
8 HTTP response codes:
9   code 200 -- 1000
10

```

```

1 num_threads=6
2 → http_load-09Mar2016 ./http_load -parallel 10 -fetches 1000 urls.txt
3 1000 fetches, 10 max parallel, 260000 bytes, in 0.075317 seconds
4 260 mean bytes/connection
5 13277.2 fetches/sec, 3.45208e+06 bytes/sec
6 msecs/connect: 0.144974 mean, 1.523 max, 0.013 min
7 msecs/first-response: 0.283606 mean, 2.968 max, 0.037 min
8 HTTP response codes:
9   code 200 -- 1000
10

```

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：

The screenshot shows the output of the `sudo iotop` command. At the top, it displays summary statistics for Total and Current Disk Read and Write speeds, all currently at 0.00 B/s. Below this is a table with columns: TID, PRIO, USER, DISK READ, DISK WRITE, SWAPIN, IO>, and COMMAND. The table lists 21 processes, mostly from the `root` user, with various priorities (be/4, rt/4). Most processes show 0.00 B/s for both disk read and write. The `COMMAND` column lists various system processes like `init auto noprompt`, `[kthreadd]`, `[rcu_gp]`, etc. At the bottom, there are instructions for using keyboard shortcuts (keys: any: refresh, q: quit, i: ionice, o: active, p: procs, a: accum, sort: r: asc, left: SWAPIN, right: COMMAND, home: TID, end: COMMAND) and a note: `CONFIG_TASK_DELAY_ACCT not enabled in kernel, cannot determine SWAPIN and IO %`.

TID	PRIO	USER	DISK READ	DISK WRITE	SWAPIN	IO>	COMMAND
1	be/4	root	0.00 B/s	0.00 B/s	?unavailable?		init auto noprompt
2	be/4	root	0.00 B/s	0.00 B/s	?unavailable?		[kthreadd]
3	be/0	root	0.00 B/s	0.00 B/s	?unavailable?		[rcu_gp]
4	be/0	root	0.00 B/s	0.00 B/s	?unavailable?		[rcu_par_gp]
5	be/0	root	0.00 B/s	0.00 B/s	?unavailable?		[slub_flushwq]
6	be/0	root	0.00 B/s	0.00 B/s	?unavailable?		[netns]
8	be/0	root	0.00 B/s	0.00 B/s	?unavailable?		[kworker~_highpri]
10	be/0	root	0.00 B/s	0.00 B/s	?unavailable?		[mm_percpu_wq]
11	be/4	root	0.00 B/s	0.00 B/s	?unavailable?		[rcu_tasks_rude_]
12	be/4	root	0.00 B/s	0.00 B/s	?unavailable?		[rcu_tasks_trace]
13	be/4	root	0.00 B/s	0.00 B/s	?unavailable?		[ksoftirqd/0]
14	be/4	root	0.00 B/s	0.00 B/s	?unavailable?		[rcu_sched]
15	rt/4	root	0.00 B/s	0.00 B/s	?unavailable?		[migration/0]
16	rt/4	root	0.00 B/s	0.00 B/s	?unavailable?		[idle_inject/0]
18	be/4	root	0.00 B/s	0.00 B/s	?unavailable?		[cpuhp/0]
19	be/4	root	0.00 B/s	0.00 B/s	?unavailable?		[cpuhp/1]
20	rt/4	root	0.00 B/s	0.00 B/s	?unavailable?		[idle_inject/1]
21	rt/4	root	0.00 B/s	0.00 B/s	?unavailable?		[migration/1]

keys: any: refresh q: quit i: ionice o: active p: procs a: accum
sort: r: asc left: SWAPIN right: COMMAND home: TID end: COMMAND
CONFIG_TASK_DELAY_ACCT not enabled in kernel, cannot determine SWAPIN and IO %

```
vmstat 2 10

procs -----memory----- --swap-- -----io---- -system-- -----cpu-----
-
r  b   swpd   free   buff   cache   si   so   bi   bo   in   cs  us  sy  id  wa  s
t
1  0       0 975692  89912 3972868    0    0   70   75  368 1348   2   4  93   0
0
7  0       0 977660  89912 3973020    0    0    0    0 24649 114668   4  77  19
0  0
4  0       0 976964  89920 3973204    0    0    0   38 19818 105402   3  68  29
0  0
9  0       0 892272  89920 3973128    0    0    0    0 22129 117249   6  80  14
0  0
1  0       0 864492  89920 3973432    0    0    0    0 18329  95617   5  71  24
0  0
3  0       0 864044  89928 3973264    0    0    0   26 22456 132814   3  77  20
0  0
7  0       0 866772  89928 3973224    0    0    0    0 25625 120842   4  82  15
0  0
3  0       0 865824  89928 3973208    0    0    0    0 24635 138838   3  81  16
0  0
```

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。