

Project5 实验报告

余北辰 519030910245

1 实验概述

1.1 实验名称

Designing a Thread Pool & Producer-Consumer Problem

1.2 实验内容

1. 模拟线程池
2. 模拟解决生产者-消费者问题

2 实验环境

- Ubuntu 18.04.5 LTS
- Linux version 5.4.0-72-generic
- VirtualBox 6.1.18

3 实验过程与结果展示

3.1 Designing a Thread Pool

测试用函数

在 `client.c` 文件中设计用于测试的函数：

```
1  int main(void)
2  {
3      // create some work to do
4      pool_init();
5      struct data work[100];
```

```

6   for (int i = 0; i < 100; ++i)
7   {
8       work[i].a = i;
9       work[i].b = i;
10      // submit the work to the queue
11      pool_submit(&add, &work[i]);
12  }
13  sleep(3);
14
15  pool_shutdown();
16
17  return 0;
18  }
19

```

每一个线程都用来进行两个相同数字的加法，一共有100个线程。

变量定义与初始化

这里我们通过循环数组的形式实现线程池中的工作队列。

这里工作队列的 `enqueue()` 和 `dequeue()` 问题类似于生产者-消费者问题，因此引入 `mutex`，`full` 和 `empty` 三个信号量以解决同步问题。

```

1   typedef struct
2   {
3       void (*function)(void *p);
4       void *data;
5   } task;
6
7   // the work queue
8   task workqueue[QUEUE_SIZE];
9
10  sem_t mutex;
11  sem_t full;
12  sem_t empty;
13  int shutdown;
14
15  // the worker bee
16  pthread_t bee[NUMBER_OF_THREADS];
17  int rear, front;

```

在 `pool_init()` 函数中实现变量的初始化：

`mutex` 初始化为1。

初始时，工作队列为空，故将`full`初始化为0，而`empty`初始化为工作队列的大小。

使用 `pthread_create()` 函数，启动线程池里的所有的线程。

```
1 void pool_init(void)
2 {
3     shutdown = 0;
4     rear = front = 0;
5     sem_init(&mutex, 0, 1);
6     sem_init(&full, 0, 0);
7     sem_init(&empty, 0, QUEUE_SIZE);
8     for (int i = 0; i < NUMBER_OF_THREADS; ++i)
9     {
10         pthread_create(&bee[i], NULL, worker, NULL);
11     }
12 }
```

入队与出队

使用 `enqueue()` 和 `dequeue()` 实现新任务加入队尾和队首任务离开工作队列。

入队时，首先要检查工作队列中是否有空位(`empty`)；入队后，要维护工作队列中元素个数，令其加一(`full`)；

出队时，首先要检查工作队列中是否有元素(`full`)；出队后，要维护工作队列中空位个数，令其加一(`empty`)；

```
1 // insert a task into the queue
2 // returns 0 if successful or 1 otherwise,
3 void enqueue(task t)
4 {
5     sem_wait(&empty);
6     sem_wait(&mutex);
7     workqueue[rear] = t;
8     rear = (rear + 1) % QUEUE_SIZE;
9     //currentsize++;
10    sem_post(&mutex);
11    sem_post(&full);
12 }
13
14 // remove a task from the queue
15 task dequeue()
16 {
17     task work_to_dequeue;
18     sem_wait(&full);
19     sem_wait(&mutex);
20     work_to_dequeue = workqueue[front];
```

```

21     front = (front + 1) % QUEUE_SIZE;
22     //currentsize--;
23     sem_post(&mutex);
24     sem_post(&empty);
25     return work_to_dequeue;
26 }

```

线程的运行

`worker()` 函数不断调用 `dequeue()` 令工作队列的队首出队，再调用 `execute()` 执行相应进程；

```

1 // the worker thread in the thread pool
2 void *worker(void *param)
3 {
4     // execute the task
5     task work_to_do;
6     while (TRUE)
7     {
8         work_to_do = dequeue();
9         if (shutdown)
10             pthread_exit(0);
11         execute(work_to_do.function, work_to_do.data);
12     }
13 }
14
15 /**
16  * Executes the task provided to the thread pool
17  */
18 void execute(void (*somefunction)(void *p), void *p)
19 {
20     (*somefunction)(p);
21 }

```

新任务的提交

而 `pool_submit()` 的功能是提交新任务，使新任务加入工作队列：

```

1  int pool_submit(void (*somefunction)(void *p), void *p)
2  {
3      task work_to_submit;
4      work_to_submit.function = somefunction;
5      work_to_submit.data = p;
6      enqueue(work_to_submit);
7
8      return 0;
9  }

```

线程的释放

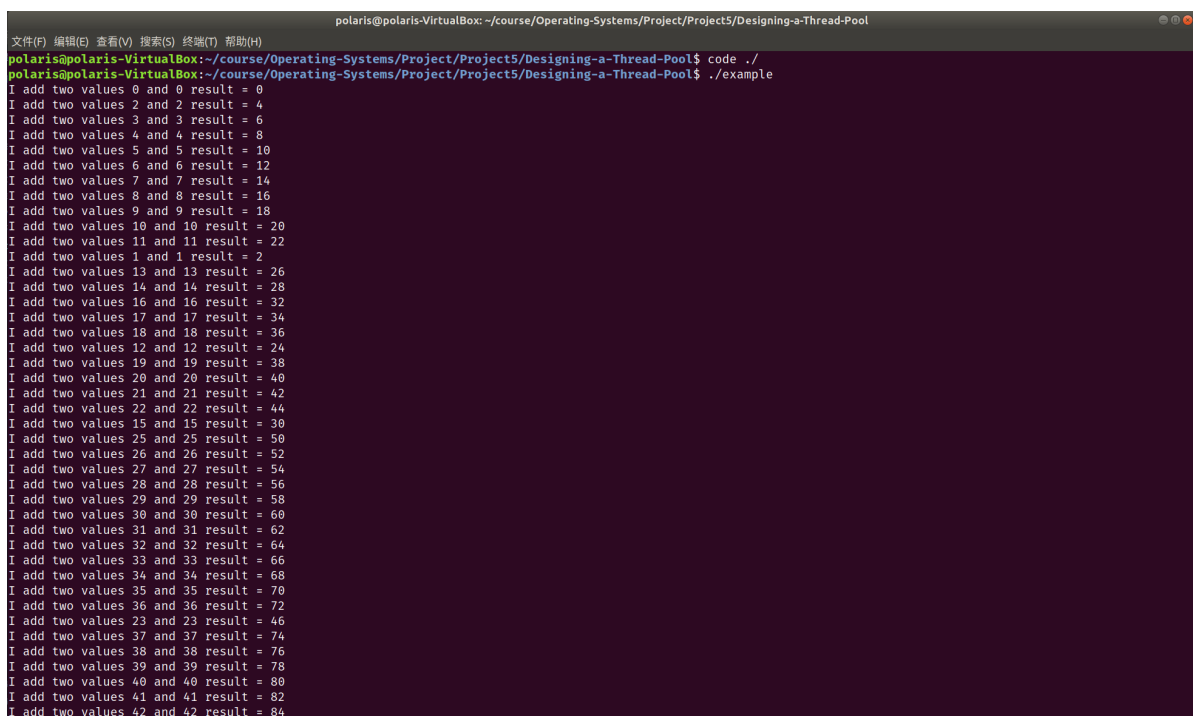
`pool_shutdown()` 的功能是将各个线程都释放掉。首先让因为信号量 `full` 而一直等待的线程解除阻塞，再调用 `pthread_join()` 函数。

```

1  // shutdown the thread pool
2  void pool_shutdown(void)
3  {
4      shutdown = 1;
5      for (int i = 0; i < NUMBER_OF_THREADS; ++i)
6          sem_post(&full);
7      for (int i = 0; i < NUMBER_OF_THREADS; ++i)
8          pthread_join(bee[i], NULL);
9  }

```

测试结果



```

polaris@polaris-VirtualBox: ~/course/Operating-Systems/Project/Project5/Designing-a-Thread-Pool
polaris@polaris-VirtualBox:~/course/Operating-Systems/Project/Project5/Designing-a-Thread-Pool$ code ./
polaris@polaris-VirtualBox:~/course/Operating-Systems/Project/Project5/Designing-a-Thread-Pool$ ./example
I add two values 0 and 0 result = 0
I add two values 2 and 2 result = 4
I add two values 3 and 3 result = 6
I add two values 4 and 4 result = 8
I add two values 5 and 5 result = 10
I add two values 6 and 6 result = 12
I add two values 7 and 7 result = 14
I add two values 8 and 8 result = 16
I add two values 9 and 9 result = 18
I add two values 10 and 10 result = 20
I add two values 11 and 11 result = 22
I add two values 1 and 1 result = 2
I add two values 13 and 13 result = 26
I add two values 14 and 14 result = 28
I add two values 16 and 16 result = 32
I add two values 17 and 17 result = 34
I add two values 18 and 18 result = 36
I add two values 12 and 12 result = 24
I add two values 19 and 19 result = 38
I add two values 20 and 20 result = 40
I add two values 21 and 21 result = 42
I add two values 22 and 22 result = 44
I add two values 15 and 15 result = 30
I add two values 25 and 25 result = 50
I add two values 26 and 26 result = 52
I add two values 27 and 27 result = 54
I add two values 28 and 28 result = 56
I add two values 29 and 29 result = 58
I add two values 30 and 30 result = 60
I add two values 31 and 31 result = 62
I add two values 32 and 32 result = 64
I add two values 33 and 33 result = 66
I add two values 34 and 34 result = 68
I add two values 35 and 35 result = 70
I add two values 36 and 36 result = 72
I add two values 23 and 23 result = 46
I add two values 37 and 37 result = 74
I add two values 38 and 38 result = 76
I add two values 39 and 39 result = 78
I add two values 40 and 40 result = 80
I add two values 41 and 41 result = 82
I add two values 42 and 42 result = 84

```

```
polaris@polaris-VirtualBox: ~/course/Operating-Systems/Project/Project5/Designing-a-Thread-Pool
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
I add two values 39 and 39 result = 78
I add two values 40 and 40 result = 80
I add two values 41 and 41 result = 82
I add two values 42 and 42 result = 84
I add two values 43 and 43 result = 86
I add two values 44 and 44 result = 88
I add two values 45 and 45 result = 90
I add two values 46 and 46 result = 92
I add two values 47 and 47 result = 94
I add two values 48 and 48 result = 96
I add two values 49 and 49 result = 98
I add two values 50 and 50 result = 100
I add two values 24 and 24 result = 48
I add two values 51 and 51 result = 102
I add two values 52 and 52 result = 104
I add two values 53 and 53 result = 106
I add two values 54 and 54 result = 108
I add two values 55 and 55 result = 110
I add two values 56 and 56 result = 112
I add two values 57 and 57 result = 114
I add two values 58 and 58 result = 116
I add two values 59 and 59 result = 118
I add two values 60 and 60 result = 120
I add two values 61 and 61 result = 122
I add two values 62 and 62 result = 124
I add two values 63 and 63 result = 126
I add two values 64 and 64 result = 128
I add two values 65 and 65 result = 130
I add two values 66 and 66 result = 132
I add two values 67 and 67 result = 134
I add two values 68 and 68 result = 136
I add two values 69 and 69 result = 138
I add two values 70 and 70 result = 140
I add two values 71 and 71 result = 142
I add two values 72 and 72 result = 144
I add two values 73 and 73 result = 146
I add two values 74 and 74 result = 148
I add two values 75 and 75 result = 150
I add two values 76 and 76 result = 152
I add two values 77 and 77 result = 154
I add two values 78 and 78 result = 156
I add two values 79 and 79 result = 158
I add two values 80 and 80 result = 160
I add two values 81 and 81 result = 162

polaris@polaris-VirtualBox: ~/course/Operating-Systems/Project/Project5/Designing-a-Thread-Pool
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
I add two values 57 and 57 result = 114
I add two values 58 and 58 result = 116
I add two values 59 and 59 result = 118
I add two values 60 and 60 result = 120
I add two values 61 and 61 result = 122
I add two values 62 and 62 result = 124
I add two values 63 and 63 result = 126
I add two values 64 and 64 result = 128
I add two values 65 and 65 result = 130
I add two values 66 and 66 result = 132
I add two values 67 and 67 result = 134
I add two values 68 and 68 result = 136
I add two values 69 and 69 result = 138
I add two values 70 and 70 result = 140
I add two values 71 and 71 result = 142
I add two values 72 and 72 result = 144
I add two values 73 and 73 result = 146
I add two values 74 and 74 result = 148
I add two values 75 and 75 result = 150
I add two values 76 and 76 result = 152
I add two values 77 and 77 result = 154
I add two values 78 and 78 result = 156
I add two values 79 and 79 result = 158
I add two values 80 and 80 result = 160
I add two values 81 and 81 result = 162
I add two values 82 and 82 result = 164
I add two values 83 and 83 result = 166
I add two values 84 and 84 result = 168
I add two values 85 and 85 result = 170
I add two values 86 and 86 result = 172
I add two values 87 and 87 result = 174
I add two values 88 and 88 result = 176
I add two values 89 and 89 result = 178
I add two values 90 and 90 result = 180
I add two values 91 and 91 result = 182
I add two values 92 and 92 result = 184
I add two values 93 and 93 result = 186
I add two values 94 and 94 result = 188
I add two values 95 and 95 result = 190
I add two values 96 and 96 result = 192
I add two values 97 and 97 result = 194
I add two values 98 and 98 result = 196
I add two values 99 and 99 result = 198
polaris@polaris-VirtualBox:~/course/Operating-Systems/Project/Project5/Designing-a-Thread-Pool$
```

测试结果正常。

3.2 Producer-Consumer Problem

buffer的实现

与Thread Pool中工作队列的实现类似，通过循环队列实现 **buffer**，定义 **empty** 和 **full** 两个信号量以及互斥锁 **mutex**，分别定义 **insert** 和 **remove** 操作：

```
1 | #include "buffer.h"
```

```

2  #include <semaphore.h>
3  #include <pthread.h>
4  #include <stdio.h>
5
6  buffer_item buffer[BUFFER_SIZE];
7  sem_t empty;
8  sem_t full;
9  pthread_mutex_t mutex;
10 int rear, front;
11
12
13 void init()
14 {
15     rear = front = 0;
16     sem_init(&empty, 0, BUFFER_SIZE);
17     sem_init(&full, 0, 0);
18     pthread_mutex_init(&mutex, 0);
19 }
20
21 int insert_item(buffer_item item)
22 {
23     sem_wait(&empty);
24     pthread_mutex_lock(&mutex);
25     buffer[rear] = item;
26     rear = (rear + 1) % BUFFER_SIZE;
27     pthread_mutex_unlock(&mutex);
28     sem_post(&full);
29     return 0;
30 }
31
32 int remove_item(buffer_item *item)
33 {
34     sem_wait(&full);
35     pthread_mutex_lock(&mutex);
36     *item = buffer[front];
37     front = (front + 1) % BUFFER_SIZE;
38     pthread_mutex_unlock(&mutex);
39     sem_post(&empty);
40     return 0;
41 }

```

主函数的实现

分别定义producer和consumer线程；

在运行 `sleep_time` 的时间后将所有进程依次释放；

按照课本要求，在producer和consumer线程中使用 `rand()` 函数在 `sleep_time` 的时间范围内随机得到进程睡眠的时间。

代码如下：

```
1  #include <stdlib.h>
2  #include <unistd.h>
3  #include <stdio.h>
4  #include <pthread.h>
5
6  #include "buffer.h"
7
8  int sleep_time, producer_num, consumer_num;
9  void *producer(void *param)
10 {
11     buffer_item item;
12     while (1)
13     {
14         sleep(rand() % sleep_time);
15         item = rand();
16         if (insert_item(item))
17             fprintf(stderr, "report error condition");
18         else
19             printf("producer produced %d\n", item);
20     }
21 }
22
23 void *consumer(void *param)
24 {
25     buffer_item item;
26     while (1)
27     {
28         sleep(rand() % sleep_time);
29         if (remove_item(&item))
30             fprintf(stderr, "report error condition");
31         else
32             printf("consumer consumed %d\n", item);
33     }
34 }
35
36 int main(int argc, char *argv[])
37 {
```



```
38     if (argc != 4)
39     {
40         fprintf(stderr, "report error condition");
41         return -1;
42     }
43
44     srand((int)time(0));
45
46     sleep_time = atoi(argv[1]);
47     producer_num = atoi(argv[2]);
48     consumer_num = atoi(argv[3]);
49
50     init();
51
52     pthread_t producers[producer_num];
53     pthread_t consumers[consumer_num];
54
55     for (int i = 0; i < producer_num; ++i)
56     {
57         pthread_create(&producers[i], 0, producer, 0);
58     }
59
60     for (int i = 0; i < consumer_num; ++i)
61     {
62         pthread_create(&consumers[i], 0, consumer, 0);
63     }
64
65     sleep(sleep_time);
66
67     for (int i = 0; i < producer_num; i++)
68         pthread_cancel(producers[i]);
69
70     for (int i = 0; i < consumer_num; i++)
71         pthread_cancel(consumers[i]);
72     return 0;
73 }
```

测试结果

```
polaris@polaris-VirtualBox: ~/course/Operating-Systems/Project/Project5/The-Producer-Consumer-Problem
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
polaris@polaris-VirtualBox:~/course/Operating-Systems/Project/Project5/The-Producer-Consumer-Problem$ ./main 3 2 2
producer produced 1980695337
producer produced 1096512255
consumer consumed 1980695337
consumer consumed 1096512255
producer produced 1540196328
consumer consumed 1540196328
producer produced 1221513013
producer produced 428914910
producer produced 1892166010
producer produced 378268344
consumer consumed 1221513013
producer produced 1326268861
consumer consumed 428914910
polaris@polaris-VirtualBox:~/course/Operating-Systems/Project/Project5/The-Producer-Consumer-Problem$ |
```

测试结果正常。

4 实验总结

1. 根据提示，在 `client.c` 文件中添加语句 `sleep(3);`，保证所有提交的任务都能正常执行完毕；
2. 无论是信号量还是互斥锁，无论是在初始化时还是在P/V操作时，在相对应函数中传递的参数都是该变量的地址。

5 实验参考资料

- 实验参考书籍：Operating System Concept, 10th edition
- 实验源代码网址：<https://github.com/greggagne/osc10e>