

计算机系统体系结构 Project5

517030910326 王孝诚

2019.11.17

实验环境

Windows 10 下使用 VMWare Workstation 15 Player 创建和运行虚拟机，虚拟机环境是 Linux 发行版 Ubuntu16.04.6 LTS。

1 线程池

这部分要实现一个线程池，提供给用户几个 API: 1) void pool_init(), 2) int pool_submit(void (*some-function)(void *p), void *p), 3) void pool_shutdown(void)。按照书上的描述，实现各自功能，并使用 semaphore 和 mutex 来实现同步和避免 race condition。

1.1 设计思路

1. **client.c.** 需要对原始代码进行一些修改，原本只调用了一次 pool_submit()，这里需要添加多个任务到线程池中展示其功能。
2. **threadpool.c.** 这里按照用户调用的顺序说明其需要实现的功能。
 - (a) 需要一个队列来保存提交的任务，这里使用数组实现。
 - (b) 首先用户初始化线程池，需要初始化信号量（这里使用了命名的信号量）、互斥锁以及创建特定数量个线程。创建的线程传入 worker 函数，worker 在线程的生命周期中一直循环，每次循环等待表示有任务可执行的信号量，然后执行一次任务。这里取出任务的 dequeue 操作需要加锁，防止多个线程同时更改。
 - (c) 接下来用户多次提交任务，在添加任务到队列的 enqueue 操作中，要对队列加锁，防止多个线程同时对其更改；若提交成功了（队列有位置可以添加）则调用 sem_post 以通知等待的线程有任务可取。然后 worker 会立即取出任务并执行。
 - (d) 最后用户要关闭线程池，对每个线程调用 pthread_cancel。

1.2 核心代码解释

1. 声明的全局变量

```
1 // the work queue
2 task worktodo[QUEUE_SIZE];
3
4 // the worker bee
5 pthread_t bee[NUMBER_OF_THREADS];
6
7 int numTask = 0;
8 pthread_mutex_t mutex;
9 sem_t *sem;
```

2. 初始化线程池

```
1 // initialize the thread pool
2 void pool_init(void)
3 {
4     sem = sem_open("SEM", O_CREAT, 0666, 0);
5     pthread_mutex_init(&mutex, NULL);
6     for (int i=0; i<NUMBER_OF_THREADS; ++i){
7         pthread_create(&bee[i],NULL,worker,NULL);
8     }
9 }
```

3. 入队列操作 enqueue

```
1 // insert a task into the queue
2 // returns 0 if successful or 1 otherwise,
3 int enqueue(task t)
4 {
5     pthread_mutex_lock(&mutex);
6     if(numTask >= QUEUE_SIZE){
7         pthread_mutex_unlock(&mutex);
8         return 1;
9     }
10    worktodo[numTask++]=t;
11    pthread_mutex_unlock(&mutex);
12    return 0;
13 }
```

4. 出队列操作 dequeue

```
1 // remove a task from the queue
2 task *dequeue()
3 {
4     pthread_mutex_lock(&mutex);
5     if(numTask<=0){
6         pthread_mutex_unlock(&mutex);
7         return NULL;
8     }
9     task *next_work = &worktodo[--numTask];
10    pthread_mutex_unlock(&mutex);
11    return next_work;
12 }
```

5. 工作线程

```
1 // the worker thread in the thread pool
2 void *worker(void *param)
3 {
4     // execute the task
5     while(TRUE){
6         sem_wait(sem);
```

```

7      task *newTask = dequeue();
8      if (newTask == NULL) continue;
9      execute(newTask->function, newTask->data);
10     }
11     pthread_exit(0);
12 }

```

6. 提交任务

```

1  int pool_submit(void (*somefunction)(void *p), void *p)
2  {
3      task newTask;
4      newTask.function = somefunction;
5      newTask.data = p;
6      int res = enqueue(newTask);
7      if (!res) sem_post(sem);
8      return res;
9  }

```

7. 关闭线程池

```

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

```

2 生产者-消费者问题

这里要使用两个信号量 empty 和 full 以及一个互斥锁 mutex 来解决生产者消费者问题。

2.1 设计思路

主函数的内容：首先获得命令行输入的 sleep time，生产者数量，消费者数量；然后初始化 buffer；然后分别创建若干个生产者线程和消费者线程，并让他们运行生产者/消费者的操作；然后主进程 sleep 输入的时间后终止所有线程。

对于生产者线程，将一直循环等待 buffer 有无空位，即 `sem_wait(&empty)`；若有空，则加锁，然后向 buffer 插入一条 item；随后解锁，并且对 full 信号量 `sem_post(&full)`。对于消费者线程，则相反，一直等待 buffer 有无 item，即 `sem_wait(&full)`；然后取出一条 item，随后解锁、`sem_post(&empty)`；。

另外还要实现 buffer 的 insert 和 remove 操作，这里用了数组实现的循环队列。

2.2 核心代码解释

1. 一些变量的声明。

```

1  buffer_item buffer[BUFFER_SIZE]; //manipulated as a circular queue
2  int rear = 0;
3  int front = 0;
4  pthread_mutex_t mutex;
5  sem_t empty;
6  sem_t full;

```

2. 首先是 buffer 的插入和删除。

```

1  int insert_item(buffer_item item) {
2      /* insert item into buffer
3       return 0 if successful, otherwise
4       return -1 indicating an error condition */
5      if ((rear+1)%BUFFER_SIZE==front){
6          return -1;
7      }
8      rear = (rear+1)%BUFFER_SIZE;
9      buffer[rear] = item;
10     return 0;
11 }
12
13 int remove_item(buffer_item *item) {
14     /* remove an object from buffer
15     placing it in item
16     return 0 if successful, otherwise
17     return -1 indicating an error condition */
18     if (front==rear){
19         return -1;
20     }
21     front = (front+1)%BUFFER_SIZE;
22     item = &buffer[front];
23     return 0;
24 }

```

3. 然后是生产者工作函数。消费者类似，这里就不展示了。

```

1  void *producer(void *param) {
2      buffer_item item;
3      while (1) {
4          /* sleep for a random period of time */
5          sleep(rand()%3);
6          /* generate a random number */
7          item = rand();
8          sem_wait(&empty);
9          pthread_mutex_lock(&mutex);
10         if (insert_item(item))
11             printf("insert_error_condition\n");
12         else
13             printf("producer_produced_%d\n",item);
14         pthread_mutex_unlock(&mutex);
15         sem_post(&full);

```

```

16     }
17 }

```

4. 最后是主函数。注意 empty 初始值应为 BUFFER_SIZE-1，因为循环队列需要区别空和满两种情况的 front 和 rear 关系，有一个空存储单元不能使用。

```

1  int main(int argc, char *argv[]) {
2  /* 1. Get command line arguments argv[1],argv[2],argv[3] */
3      int sleep_time, producer_threads, consumer_threads;
4      if (argc != 4)
5      {
6          fprintf(stderr, "Input form:<sleep_time> <producer_threads_number> <consumer_threads_number>\n");
7          return -1;
8      }
9      sleep_time = atoi(argv[1]);
10     producer_threads = atoi(argv[2]);
11     consumer_threads = atoi(argv[3]);
12
13     /* 2. Initialize buffer */
14     pthread_mutex_init(&mutex, NULL);
15     sem_init(&full, 0, 0);
16     sem_init(&empty, 0, BUFFER_SIZE-1);
17
18     /* 3. Create producer thread(s) */
19     pthread_t producer_ids[producer_threads];
20     for (int i=0;i<producer_threads;++i){
21         pthread_create(&producer_ids[i], NULL, &producer, NULL);
22     }
23
24     /* 4. Create consumer thread(s) */
25     pthread_t consumer_ids[consumer_threads];
26     for (int i=0;i<consumer_threads;++i){
27         pthread_create(&consumer_ids[i], NULL, &consumer, NULL);
28     }
29
30     /* 5. Sleep */
31     sleep(sleep_time);
32
33     /* 6. Exit */
34     return 0;
35 }
36

```

3 实验结果

```
xcwang@ubuntu:~/Documents/project5/ThreadPool$ ./example
I add two values 0 and 1 result = 1
I add two values 2 and 3 result = 5
I add two values 1 and 2 result = 3
I add two values 4 and 5 result = 9
I add two values 3 and 4 result = 7
```

Figure 1: ThreadPool

```
xcwang@ubuntu:~/Documents/project5/producer_consumer$ ./pcp 4 3 3
producer produced 719885386
producer produced 596516649
producer produced 1025202362
producer produced 783368690
consumer consumed 0
consumer consumed 0
producer produced 1365180540
producer produced 304089172
consumer consumed 0
producer produced 35005211
consumer consumed 0
producer produced 521595368
consumer consumed 0
producer produced 278722862
consumer consumed 0
producer produced 468703135
consumer consumed 0
producer produced 1315634022
consumer consumed 0
producer produced 1059961393
consumer consumed 0
producer produced 1656478042
producer produced 1656478042
```

Figure 2: Producer consumer