CS307 Project5: Designing A Thread-Pool & Producer-Consumer Algorithm

Junjie Wang 517021910093

May 23, 2019

1 Programming Thoughts

1.1 Part1: Designing A Thread-Pool

The main advantage of a thread pool is that it saves the time of creating new threads(it will only create all the threads once at the beginning). In the implementation logic, we need to organize the tasks into a queue(with FIFO property) and then submit the tasks to the thread pool. Once receiving the signal(using semaphores here), an available thread will take over a task from the queue and start execution. Once it finishes execution, it will wait in the pool for new tasks to come.
An important part of this part is using semaphores to inform the threads of coming tasks and using mutex to prevent simultaneous operations on the task queue.

1.2 Part2: Producer-Consumer Algorithm

In this part we will create a number of producers and consumers. Producers will sleep for random time and then insert a random item into the buffer. Consumers will sleep for random time and then remove an random item from the buffer. The main logic here is to use semaphores **full** and **empty** to indicate whether the buffer is full or empty. Also mutex is needed for fear of simultaneous operations on the same buffer region. All this things will happen permanently, so we need to set a hyperparameter about how long it will take before the termination of the program.

2 Execution Results And Snapshots

The execution results are listed as follows:

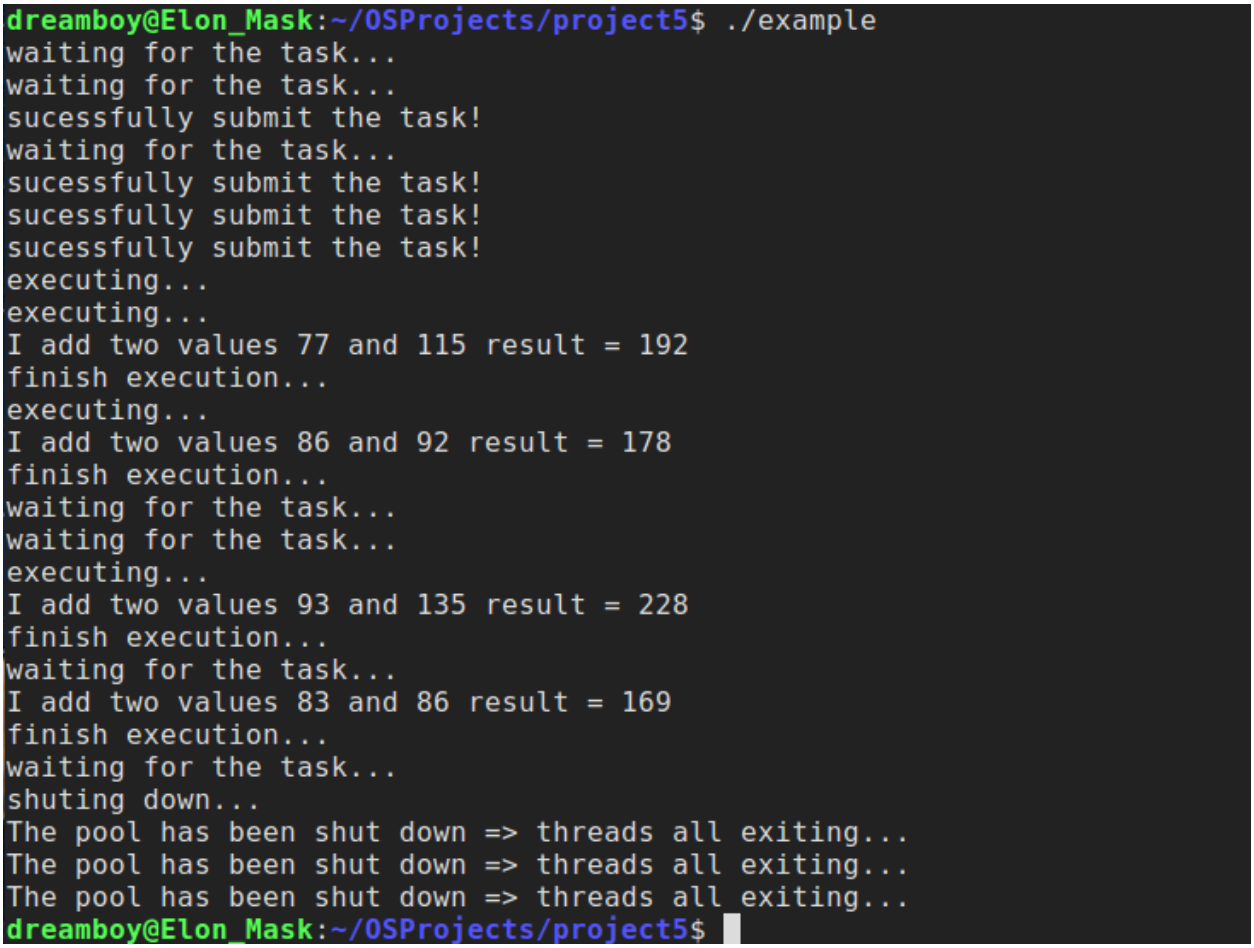

Figure 1: thread pool

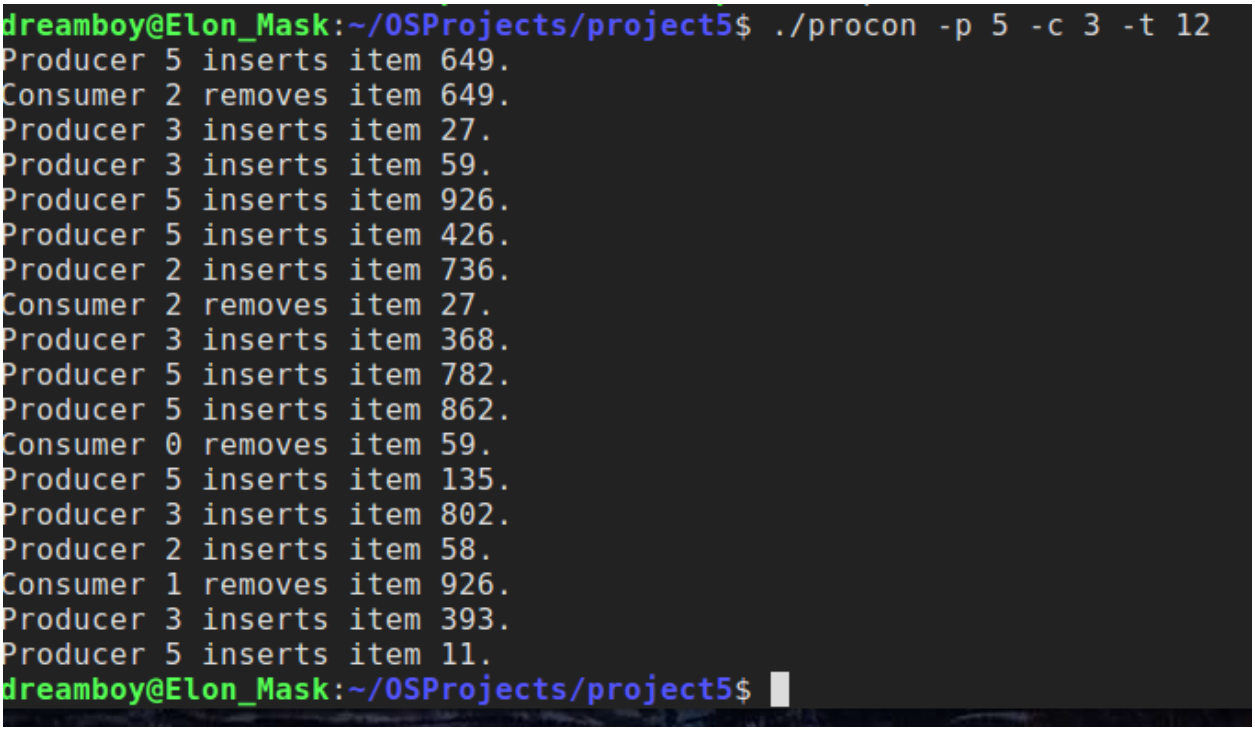

Figure 2: Producer-Consumer Algorithm

3 Code Explanation

First there are several major functions we will use about the semaphores and mutex.

```
/* if the value of semaphore is larger than 0, we decrease by 1 and
    continues.
   otherwise, we will wait until sem > 0 */
int sem_wait(sem_t *sem);
/* increase the value of sem by 1 and invoke threads which are waiting on
    this semaphore. */
int sem_post(sem_t *sem);
/* lock the mutex */
int pthread_mutex_lock(pthread_mutex_t *mutex);
/* unlock the mutex */
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

For part1, we show the code of the *worker* function, which is the core of the algorithm.

```
void *worker(void *param){
while(1){
   /* must watch on whether the pool has been shutdown */
   while(num_tasks == 0 && !is_shutdown){
   printf("waiting for the task...\n");
   sem_wait(&sem);
```

```
    }
    if(is_shutdown){
       /* if the pool has been shut down, all the threads must exit */
       printf("The pool has been shut down => threads all exiting...\n");
       pthread_exit(0);
    }
    /* lock the mutex to avoid race condition */
    pthread_mutex_lock(&mutex);
    task = dequeue();
    pthread_mutex_unlock(&mutex);
    num_tasks--;
    execute(task.function, task.data);
    }
}
```

For part2, we take the code of the consumer as example to show the logic.

```
void *consumer(void *param){
    /* sleep for random time
    * remove an item from the buffer */
    int sleep_time;
    int id = *(int *)param;
    bufferItem item;
    while(1){
    /* avoid endless sleeping */
    sleep_time = rand() % 10;
    sleep(sleep_time);
    item = removeItem();
    printf("Consumer %d removes item %d.\n", id, item);
    }
}
```