Operating System Project 3

潘梓丞 517030910349

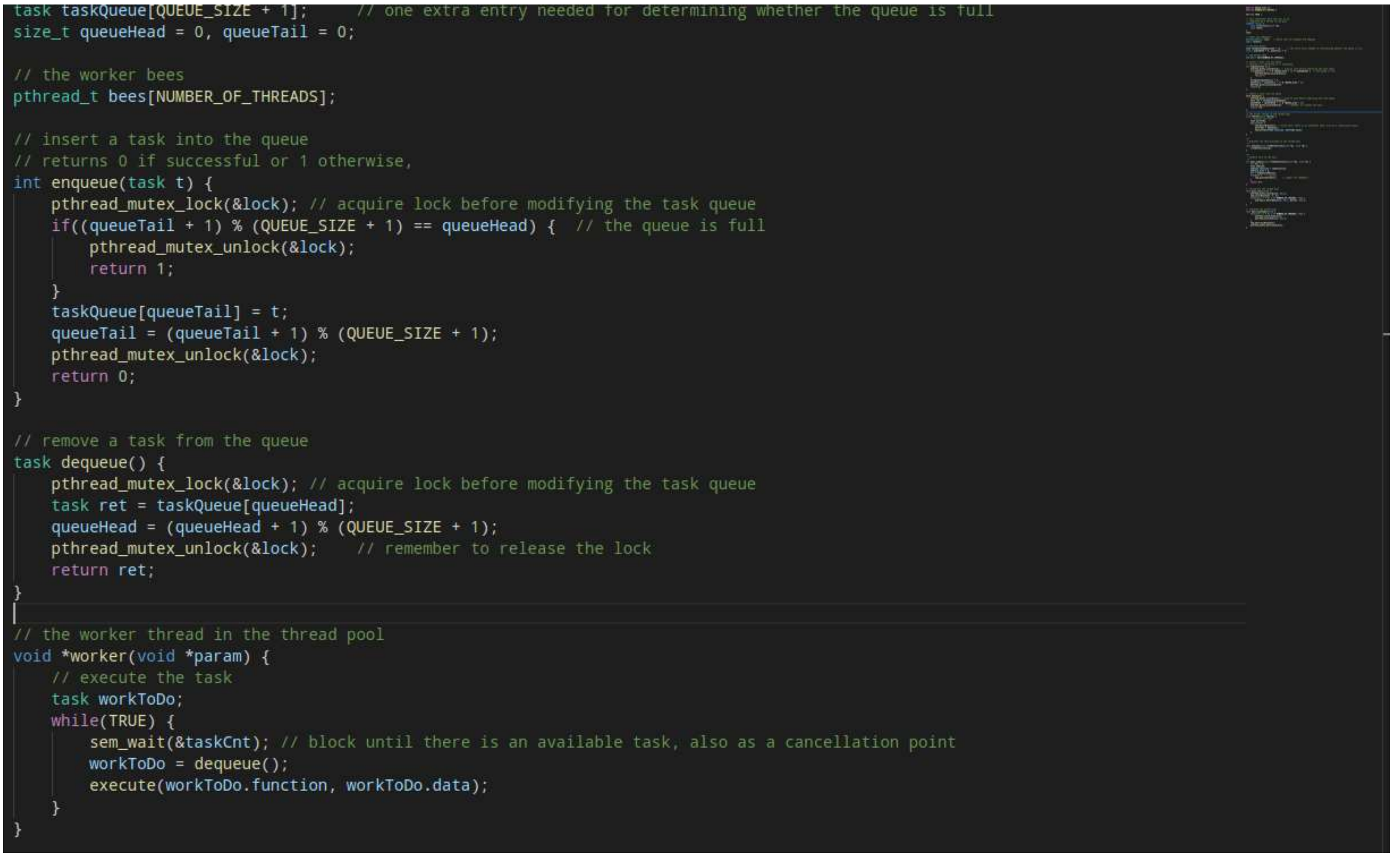
The goal of the project is to build a threadpool and simulate a producer-consumer model
The porject was done by VM VirtualBox 5.2.18
The code are written by C and the library needed will be shown in code

1.threadpool

idea

Define a work which output $i + i$, the porgram will accept a integer n and then using the threadpool to do the addition for n times. In the threadpool, using a $mutex\_lock$ to lock when enqueue or dequeue in order to avoid race condition, after a opration is done, unlock the $mutex\_lock$ to wait for next task

code



```
task taskQueue[QUEUE_SIZE + 1];     // one extra entry needed for determining whether the queue is full
size_t queueHead = 0, queueTail = 0;

// the worker bees
pthread_t bees[NUMBER_OF_THREADS];

// insert a task into the queue
// returns 0 if successful or 1 otherwise,
int enqueue(task t) {
    pthread_mutex_lock(&lock); // acquire lock before modifying the task queue
    if((queueTail + 1) % (QUEUE_SIZE + 1) == queueHead) {  // the queue is full
        pthread_mutex_unlock(&lock);
        return 1;
    }
    taskQueue[queueTail] = t;
    queueTail = (queueTail + 1) % (QUEUE_SIZE + 1);
    pthread_mutex_unlock(&lock);
    return 0;
}

// remove a task from the queue
task dequeue() {
    pthread_mutex_lock(&lock); // acquire lock before modifying the task queue
    task ret = taskQueue[queueHead];
    queueHead = (queueHead + 1) % (QUEUE_SIZE + 1);
    pthread_mutex_unlock(&lock);    // remember to release the lock
    return ret;
}

// the worker thread in the thread pool
void *worker(void *param) {
    // execute the task
    task workToDo;
    while(TRUE) {
        sem_wait(&taskCnt); // block until there is an available task, also as a cancellation point
        workToDo = dequeue();
        execute(workToDo.function, workToDo.data);
    }
}
```

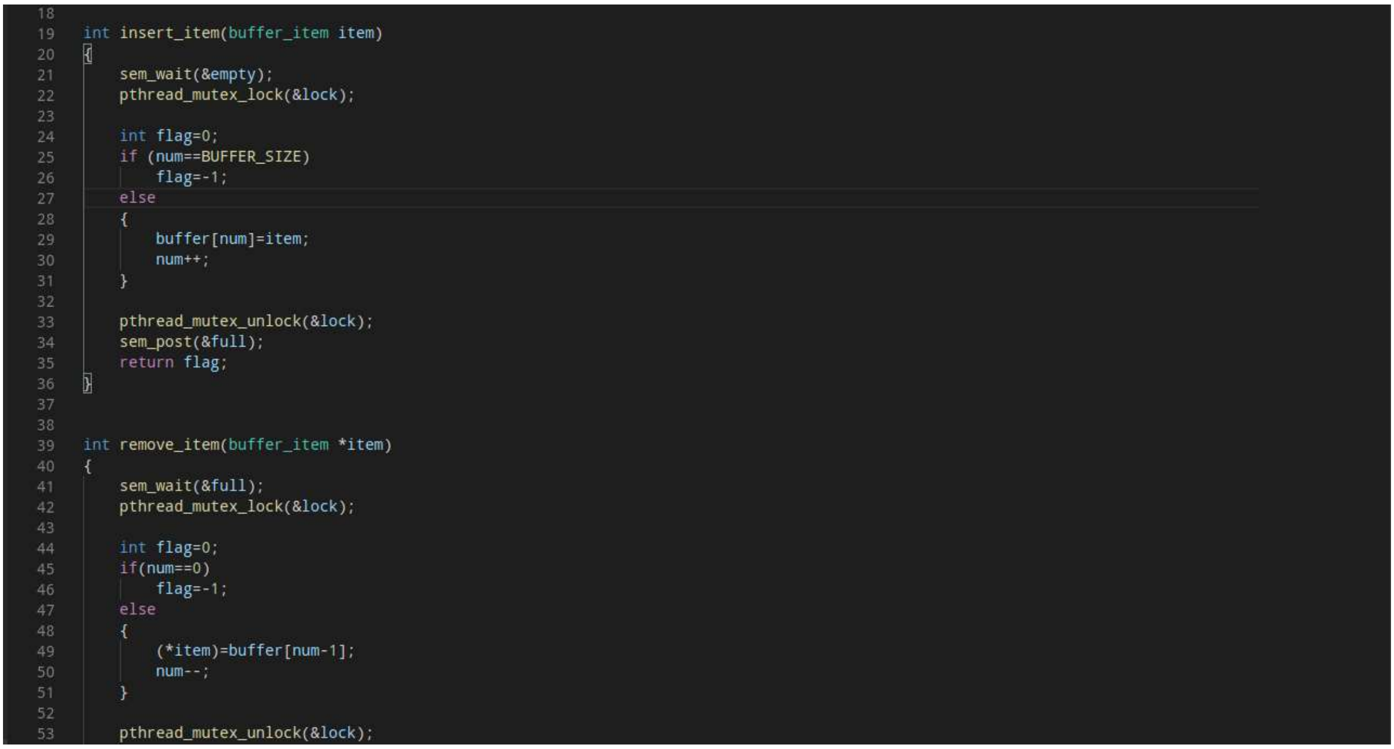
2.producer-consumer model

idea

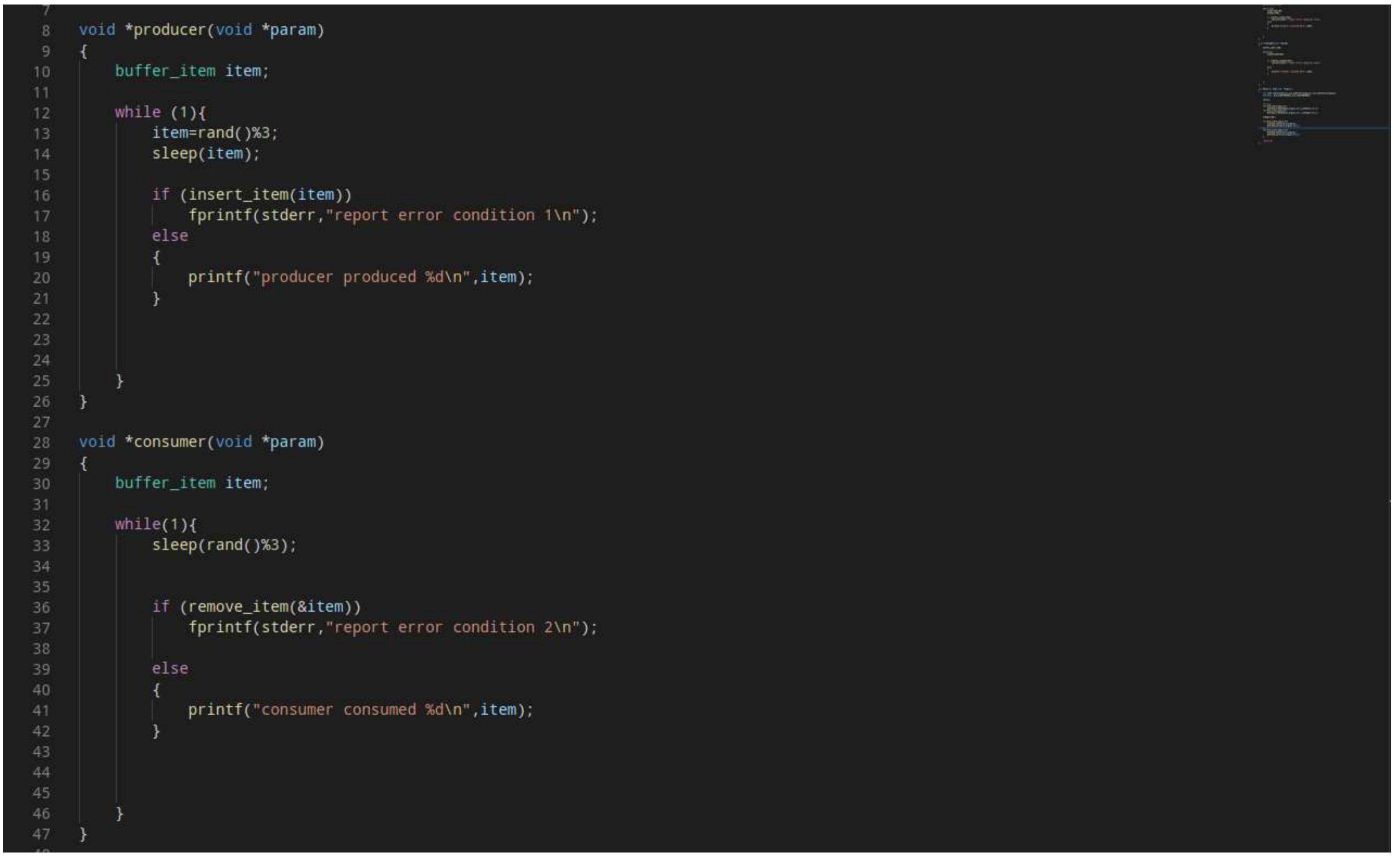
Just same as the threadpool, using a $mutex\_lock$ to lock the program when insert and remove. When the work is done, unlock the $mutex\_lock$ to wait for next task.

For producer and consumer, both of them will receive a flag from the function $insert$ and $remove$. If the flag is -1, report error condition, otherwise print the cargo produced and consumered.

code



```
int insert_item(buffer_item item)
{
    sem_wait(&empty);
    pthread_mutex_lock(&lock);

    int flag=0;
    if (num==BUFFER_SIZE)
        flag=-1;
    else
    {
        buffer[num]=item;
        num++;
    }

    pthread_mutex_unlock(&lock);
    sem_post(&full);
    return flag;
}


int remove_item(buffer_item *item)
{
    sem_wait(&full);
    pthread_mutex_lock(&lock);

    int flag=0;
    if(num==0)
        flag=-1;
    else
    {
        (*item)=buffer[num-1];
        num--;
    }

    pthread_mutex_unlock(&lock);
```



```
void *producer(void *param)
{
    buffer_item item;

    while (1){
        item=rand()%3;
        sleep(item);

        if (insert_item(item))
            fprintf(stderr,"report error condition 1\n");
        else
        {
            printf("producer produced %d\n",item);
        }



    }
}

void *consumer(void *param)
{
    buffer_item item;

    while(1){
        sleep(rand()%3);


        if (remove_item(&item))
            fprintf(stderr,"report error condition 2\n");

        else
        {
            printf("consumer consumed %d\n",item);
        }



    }
}
```