OS Project-5 Report

Designing a Thread Pool && The Producer-Consumer Problem

Zhou Yiyuan

516021910270
1025152261@qq.com

1 Designing a Thread Pool

1.1 Introduce

This project involves creating and managing a thread pool, and I completed it using either Pthreds and POSIX synchronization.

When thread pools are used, a task is submitted to the pool and executed by a thread from the pool. Work is submitted to the pool using a queue, and an available thread removes work from the queue. If there are no available threads, the work remains queued until one becomes available. If there is no work, threads await notification until a task becomes available.

1.2 The Client

Users of the thread pool will utilize the following API :

- **void pool_init()**
  - Initializes the thread pool.
- **int pool submit(void (*somefunction)(void *p), void *p)**
  - where somefunction is a pointer to the function that will be executed by a thread from the pool and p is a parameter passed to the function.
- **void pool shutdown(void)**
  - Shuts down the thread pool once all tasks have completed.

1.2.1 Code

- **sleep(3)** is used to wait for completion of **pool_init()**, and then tasks start to be submitted.
- 15 tasks are submitted continuously.
- **sleep(3)** is invoked again to ensure that all tasks have been submitted, or errors may occur.

```c
/* client.c */
/**
 * Example client program that uses thread pool.
 */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include "threadpool.h"

struct data
{
    int a;
    int b;
};
```

```c
void add(void *param)
{
    struct data *temp;
    temp = (struct data*)param;

    printf("I add two values %d and %d result = %d\n",temp->a, temp->b, temp->a
    ↪  + temp->b);
}

int main(void)
{
    // create some work to do
    struct data *work;

    // initialize the thread pool
    pool_init();
    sleep(3);

    // submit the work to the queue
    for(int i=0; i<15; ++i){
        work = (struct data *)malloc(sizeof(struct data));
        work->a = i;
        work->b = 2*i;
        pool_submit(&add, work);
    }

    // may be helpful
    sleep(3);
    pool_shutdown();

    return 0;
}
```

1.3 Implementation of the Thread Pool

We need to implement the functions that are called by client users, as well as several additional functions that support the internals of the thread pool. Implementation will involve the following activities:

1.3.1 pool_init()

The pool_init() function will create the threads at startup as well as initialize mutual-exclusion locks and semaphores.

```
// initialize the thread pool
void pool_init(void)
{
    pthread_mutex_init(&mutex,NULL);
    sem_unlink("SEM_QUEUE");
    sem_close(sem_queue);
    sem_queue = sem_open("SEM_QUEUE", O_CREAT, 0666, 0);

    taskListHead = NULL;
    run = 1;

    for(int i=0; i<NUMBER_OF_THREADS; ++i)
        pthread_create(&tid[i], NULL, worker, NULL);
}
```

1.3.2 pool_submit()

The pool_submit() function places the function to be executed, as well as its data into a task struct. The task struct represents work that will be completed by a thread in the pool. pool_submit() will add these tasks to the queue by invoking the enqueue() function, and worker threads will call dequeue() to retrieve work from the queue. The queue may be implemented dynamically (using a linked list).

```
int pool_submit(void (*somefunction)(void *p), void *p)
{
    task *newTask = (task *)malloc(sizeof(task));
    newTask->function = somefunction;
    newTask->data = p;

    pthread_mutex_lock(&mutex);
    enqueue(*newTask);
    sem_post(sem_queue);
    pthread_mutex_unlock(&mutex);

    return 0;
}
```

1.3.3 worker()

The worker() function is executed by each thread in the pool, where each thread will wait for available work. Once work becomes available, the thread will remove it from the queue and invoke execute() to run the specified function. A named semaphore is used for notifying a waiting thread when work is submitted to the thread pool.

```
void *worker(void *param)
{
    while(run){
        sem_wait(sem_queue);
        if(run == 0) break;
        pthread_mutex_lock(&mutex);
        worktodo = (task *)malloc(sizeof(task));
        *worktodo =  dequeue();
        execute(worktodo->function, worktodo->data);
        pthread_mutex_unlock(&mutex);
    }
    printf("exit worker\n");
    pthread_exit(0);
}
```

1.3.4 mutex

A mutex lock is necessary to avoid race conditions when accessing or modifying the queue.

1.3.5 pool_shutdown()

The pool_shutdown() function will cancel each worker thread and then wait for each thread to terminate by calling pthread_join().

```
// shutdown the thread pool
void pool_shutdown(void)
{
    run = 0;
    for(int i=0; i<NUMBER_OF_THREADS; ++i)
        sem_post(sem_queue);
    for(int i=0; i<NUMBER_OF_THREADS; ++i)
        pthread_join(tid[i],NULL);
    sem_unlink("SEM_QUEUE");
    sem_close(sem_queue);
}
```

1.3.6 enqueue()

The enqueue() function will insert the new task to the head of the linked list.

```
// insert a task into the queue
// returns 0 if successful or 1 otherwise,
int enqueue(task t)
{
    node *newNode = (node *)malloc(sizeof(node));
    newNode->Task = t;
```

```
    newNode->next = taskListHead;

    taskListHead = newNode;

    return 0;
}
```

1.3.7 dequeue()

The dequeue() function will remove the rail task from the linked list and return it.

```
// remove a task from the queue
task dequeue()
{
    node *temp = taskListHead;
    node *prev = taskListHead;

    while(temp->next != NULL){
        prev = temp;
        temp = temp->next;
    }

    if(prev == temp){
        taskListHead = NULL;
        return temp->Task;
    }else{
        prev->next = NULL;
        return temp->Task;
    }
}
```

1.4 Result

```
→  ThreadPool ./example
I add two values 0 and 0 result = 0
I add two values 1 and 2 result = 3
I add two values 2 and 4 result = 6
I add two values 3 and 6 result = 9
I add two values 4 and 8 result = 12
I add two values 5 and 10 result = 15
I add two values 6 and 12 result = 18
I add two values 7 and 14 result = 21
I add two values 8 and 16 result = 24
I add two values 9 and 18 result = 27
I add two values 10 and 20 result = 30
I add two values 11 and 22 result = 33
I add two values 12 and 24 result = 36
I add two values 13 and 26 result = 39
I add two values 14 and 28 result = 42
exit worker
exit worker
exit worker
```

Figure 1: The result of thread pool

2 The Producer-Consumer Problem

2.1 Introduce

In this project, you will design a programming solution to the bounded-buffer problem using the producer and consumer processes. The solution presented in Section 7.1.1 uses three semaphores: empty and full , which count the number of empty and full slots in the buffer, and mutex , which is a binary (or mutual-exclusion) semaphore that protects the actual insertion or removal of items in the buffer. For this project, you will use standard counting semaphores for empty and full and a mutex lock, rather than a binary semaphore, to represent mutex .

2.1.1 Buffer

The buffer will be manipulated with two functions, **insert_item()** and **remove_item()** , which are called by the producer and consumer threads, respectively. A circular array serves as the buffer. In addition, **display()** is implemented to print all numbers in the buffer.

```c
typedef int buffer_item;
#define BUFFER_SIZE 5
int head, rail;
/* the buffer */
buffer_item buffer[BUFFER_SIZE + 1];

void display(void){
    if(head == rail)
        printf("buffer is null\n");
    else{
```

```
        int i = (head + 1) % (BUFFER_SIZE + 1);
        while (true)
        {
            printf("%d ", buffer[i]);
            if(i == rail)
                break;
            i = (i + 1) % (BUFFER_SIZE + 1);
        }
        printf("\n");
    }
}

int insert_item(buffer_item item) {
    /* insert item into buffer
    return 0 if successful, otherwise
    return -1 indicating an error condition */

    rail = (rail + 1) % (BUFFER_SIZE + 1);
    buffer[rail] = item;

    printf("insert %d :", item);
    display();

    return 0;
}

int remove_item(buffer_item *item){
    /* remove an object from
    placing it in item
    return 0 if successful,
    return -1 indicating an error condition */


    head = (head + 1) % (BUFFER_SIZE + 1);
    *item = buffer[head];

    printf("remove %d :", *item);
    display();

    return 0;
}
```

2.2 Main()

The main() function will initialize the buffer and create the separate pro- ducer and consumer threads. Once it has created the producer and consumer threads, the main() function will sleep for

a period of time and, upon awaken- ing, will terminate the application. The main() function will be passed three parameters on the command line:

- How long to sleep before terminating
- The number of producer threads
- The number of consumer threads

```
int main(int argc, char *argv[]){
    /* 1. Get command line arguments argv[1],argv[2],argv[3] */
    if(argc != 4){
        printf("numbers of params are wrong!");
        exit(-1);
    }
    int sleep_time = atoi(argv[1]);
    int num_producer = atoi(argv[2]);
    int num_consumer = atoi(argv[3]);

    printf("sleep time = %d, num producer = %d, num consumer = %d\n",
    ↪  sleep_time, num_producer, num_consumer);

    /* 2. Initialize buffer */
    pthread_mutex_init(&mutex,NULL);
    head = rail = 0;

    sem_unlink("ZYY_SEM_FULL");
    sem_close(sem_full);
    sem_unlink("ZYY_SEM_EMPTY");
    sem_close(sem_empty);
    sem_full = sem_open("ZYY_SEM_FULL", O_CREAT, 066, 0);
    sem_empty = sem_open("ZYY_SEM_EMPTY", O_CREAT, 066, BUFFER_SIZE);

    // /* Create producer thread(s) */
    pthread_t *producer_tid = (pthread_t *)malloc(num_producer *
    ↪  sizeof(pthread_t));
    for(int i=0; i<num_producer; ++i)
        pthread_create(&producer_tid[i], NULL, producer, NULL);

    // /* 4. Create consumer thread(s) */
    pthread_t *consumer_tid = (pthread_t *)malloc(num_consumer *
    ↪  sizeof(pthread_t));
    for (int i=0; i<num_consumer; ++i)
        pthread_create(&consumer_tid[i], NULL, consumer, NULL);

    /* 5. Sleep */
    sleep(sleep_time);
```

```
    /* 6. Exit */
    sem_unlink("ZYY_SEM_FULL");
    sem_close(sem_full);

    sem_unlink("ZYY_SEM_EMPTY");
    sem_close(sem_empty);
    return 0;
}
```

2.3 The Producer and Consumer Threads

The producer thread will alternate between sleeping for a random period of time and inserting a random integer into the buffer. Random numbers will be produced using the rand() function, which produces random integers between 0 and RAND_MAX . The consumer will also sleep for a random period of time and, upon awakening, will attempt to remove an item from the buffer.

```
void *producer(void *param) {
    buffer_item item;
    while (true) {
        /* sleep for a random period of time */
        sleep(rand()%4 + 1);
        sem_wait(sem_empty);
        pthread_mutex_lock(&mutex);

        item = rand() % 10;
        if (insert_item(item))
            printf("report error condition");
        // else
        //     printf("producer produced %d\n", item);

        pthread_mutex_unlock(&mutex);
        sem_post(sem_full);
    }
}

void *consumer(void *param) {
    buffer_item item;
    while (true) {
        /* sleep for a random period of time */
        sleep(rand()%4 + 1);
        sem_wait(sem_full);
        pthread_mutex_lock(&mutex);

        if (remove_item(&item))
            printf("report error condition");
        // else
```

```
        //      printf("consumer consumed %d\n",item);

        pthread_mutex_unlock(&mutex);
        sem_post(sem_empty);
    }
}
```

3 Result

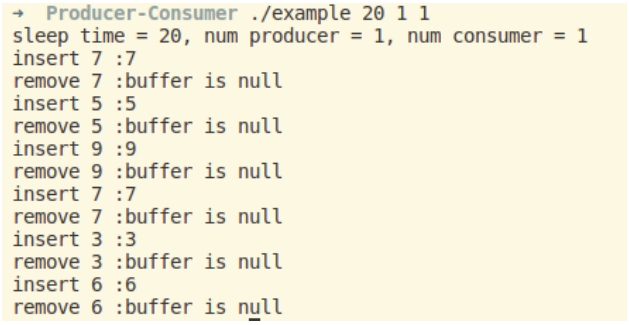

Figure 2: sleep_time=20, producer_num=1, consumer_num=1

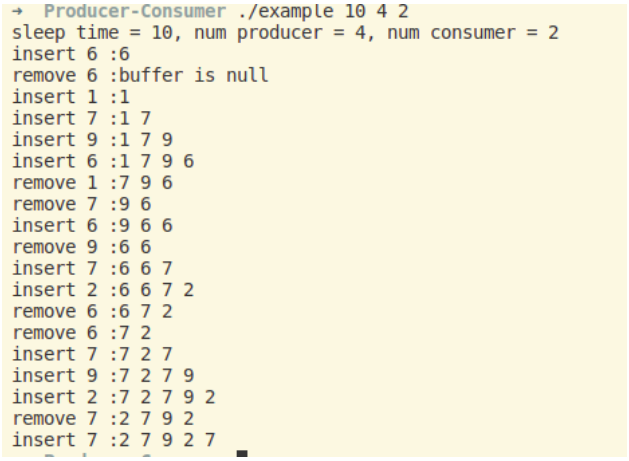

Figure 3: sleep_time=10, producer_num=4, consumer_num=2

```
→  Producer-Consumer ./example 5 10 10
sleep time = 5, num producer = 10, num consumer = 10
insert 1 :1
remove 1 :buffer is null
insert 9 :9
insert 2 :9 2
remove 9 :2
insert 7 :2 7
insert 5 :2 7 5
remove 2 :7 5
insert 2 :7 5 2
insert 7 :7 5 2 7
insert 6 :7 5 2 7 6
remove 7 :5 2 7 6
remove 5 :2 7 6
remove 2 :7 6
remove 7 :6
insert 9 :6 9
insert 7 :6 9 7
insert 4 :6 9 7 4
remove 6 :9 7 4
insert 3 :9 7 4 3
remove 9 :7 4 3
remove 7 :4 3
remove 4 :3
insert 3 :3 3
insert 0 :3 3 0
insert 1 :3 3 0 1
insert 5 :3 3 0 1 5
remove 3 :3 0 1 5
insert 6 :3 0 1 5 6
remove 3 :0 1 5 6
remove 0 :1 5 6
insert 9 :1 5 6 9
remove 1 :5 6 9
remove 5 :6 9
remove 6 :9
```

Figure 4: sleep_time=5, producer_num=10, consumer_num=10