Project 5

Name: 刘 涵 之
ID : 519021910102

Project 5: Designing a Thread Pool The Producer–Consumer Problem

1 Designing a Thread Pool

Thread pools were introduced in the OS book. 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.

This project involves creating and managing a thread pool, and it may be completed using either Pthreds and POSIX synchronization or Java.

In the source code download we provide the C source file `threadpool.c` as a partial implementation of the thread pool. You will 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:

- The **pool_init()** function will create the threads at startup as well as initialize mutual-exclusion locks and semaphores.
- The **pool_submit()** function is partially implemented and currently 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 statically (using arrays) or dynamically (using a linked list).

  The **pool_init()** function has an int return value that is used to indicate if the task was successfully submitted to the pool (0 indicates success, 1 indicates failure). If the queue is implemented using arrays, **pool_init()** will return 1 if there is an attempt to submit work and the queue is full. If the queue is implemented as a linked list, **pool_init()** should always return 0 unless a memory allocation error occurs.
- 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 semaphore can be used for notifying a waiting thread when work is submitted to the thread pool. Either named or unnamed semaphores may be used.
- A mutex lock is necessary to avoid race conditions when accessing or modifying the queue.
- The pool shutdown() function will cancel each worker thread and then wait for each thread to terminate by calling **pthread_join()**. The semaphore operation **sem_wait()** is a cancellation point that allows a thread waiting on a semaphore to be cancelled.

Design: My design for this task is shown as follows:

- To track all threads on the thread pool, the linked list structure `work_queue` is implemented.
- A semaphore **thread_sem** and a mutex **thread_mut** is used to solve the critical section.
- A **worker** should wait the semaphore **thread_sem**, and if **running** is equal to 1 (which means the thread pool is closed), the worker will stop its job and exit.
- The critical section here is the **dequeue()** function. The mutex **thread_mut** is used to solve it.

Implementation of thread pool is shown as follows. (thread_pool.c)

```c
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <semaphore.h>
#include "threadpool.h"

#define QUEUE_SIZE 10
#define NUMBER_OF_THREADS 3

#define TRUE 1

// this represents work that has to be
// completed by a thread in the pool
typedef struct
{
    void (*function)(void *p);
    void *data;
}task;

struct work_queue
{
    task work;
    struct work_queue *next;
};

// task
task task_to_do;

// the work queue
struct work_queue worktodo;
struct work_queue *head, *tail;
// the worker bee
pthread_t bee[NUMBER_OF_THREADS];

//mutex
pthread_mutex_t thread_mut;

//sem
sem_t thread_sem;

```

```c
//todo or not todo
int running;

// insert a task into the queue
// returns 0 if successful or 1 otherwise,
int enqueue(task t)
{
    tail -> next = (struct work_queue *) malloc (sizeof (struct work_queue));
    if (tail -> next == NULL) {
        fprintf(stderr, "[Error] cannot malloc memory!\n");
                exit(1);
    }

    tail = tail -> next;
    tail -> work = t;

    return 0;
}

// remove a task from the queue
task dequeue()
{
    if (head == tail) {
        fprintf(stderr, "[Error] No work remains!\n");
                exit(1);
    }

    head = head -> next;

    return head -> work;
}

// the worker thread in the thread pool
void *worker(void *param)
{
    while (TRUE) {
        sem_wait(&thread_sem);

        if (running == 1) break;

        pthread_mutex_lock(&thread_mut);
        task_to_do = dequeue();
        pthread_mutex_unlock(&thread_mut);

        // execute the task
        execute(task_to_do.function, task_to_do.data);
    }

    pthread_exit(0);
}

```

```c
/**
 * Executes the task provided to the thread pool
 */
void execute(void (*somefunction)(void *p), void *p)
{
    (*somefunction)(p);
}

/**
 * Submits work to the pool.
 */
int pool_submit(void (*somefunction)(void *p), void *p)
{
    task_to_do.function = somefunction;
    task_to_do.data = p;

    pthread_mutex_lock(&thread_mut);
    int result = enqueue(task_to_do);
    pthread_mutex_unlock(&thread_mut);

    if (result == 0) {
        sem_post(&thread_sem);
    }
    return result;
}

// initialize the thread pool
void pool_init(void)
{
    running = 0;
    head = (struct work_queue *) malloc (sizeof (struct work_queue));
    if (head == NULL) {
        fprintf(stderr, "[Error] cannot malloc memory!\n");
                exit(1);
    }
    tail = head;
    head->next = NULL;

    // create mutex
    if (pthread_mutex_init(&thread_mut, NULL)){
        fprintf(stderr, "[Error] cannot create mutex!\n");
                exit(1);
    }

    // create semaphore
    if (sem_init(&thread_sem, 0, 0)) {
        fprintf(stderr, "[Error] cannot create semaphore!\n");
                exit(1);
    }

    // create threads
```

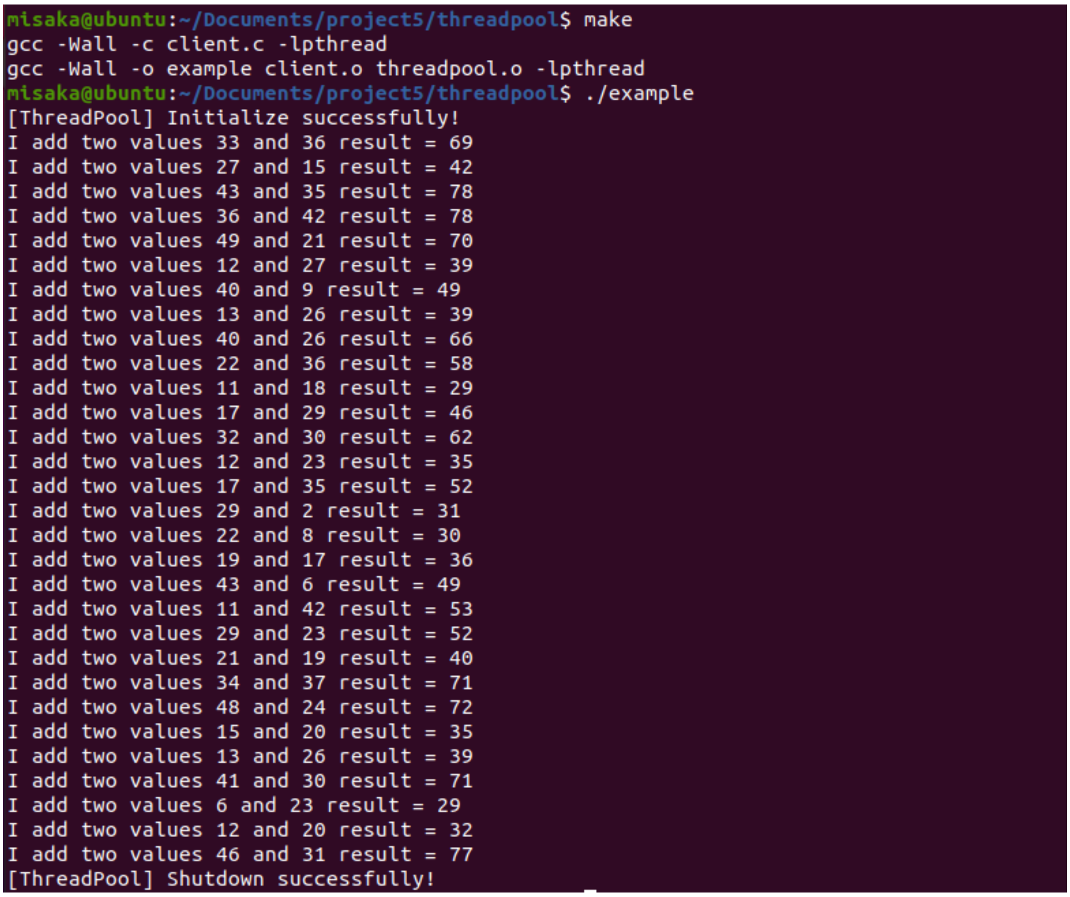
```c
    for (int i = 0; i < NUMBER_OF_THREADS; i++) {
        if(pthread_create(&bee[i],NULL,worker,NULL)) {
            fprintf(stderr, "[Error] cannot create thread!\n");
                        exit(1);
        }
    }

    fprintf(stdout, "[ThreadPool] Initialize successfully!\n");
}

// shutdown the thread pool
void pool_shutdown(void)
{
    running = 1;

    // set semaphore
    for (int i = 0; i < NUMBER_OF_THREADS; i++) {
        sem_post(&thread_sem);
    }

    // join
    for (int i = 0; i < NUMBER_OF_THREADS; i++) {
        if (pthread_join(bee[i],NULL)) {
            fprintf(stderr, "[Error] cannot join thread!\n");
                        exit(1);
        }
    }

    // destroy
    if (pthread_mutex_destroy(&thread_mut) || sem_destroy(&thread_sem)) {
        fprintf(stderr, "[Error] cannot destroy semaphore or mutex!\n");
                exit(1);
    }

    fprintf(stdout, "[ThreadPool] Shutdown successfully!\n");
}
```

Makefile for thread-pool task is shown as follow.

```makefile
CC=gcc
CFLAGS=-Wall
PTHREADS=-lpthread

all: client.o threadpool.o
	$(CC) $(CFLAGS) -o example client.o threadpool.o $(PTHREADS)

client.o: client.c
	$(CC) $(CFLAGS) -c client.c $(PTHREADS)

threadpool.o: threadpool.c threadpool.h
	$(CC) $(CFLAGS) -c threadpool.c $(PTHREADS)

```

```
clean:
	rm -rf *.o
	rm -rf example
```

The execution result is shown as follow.



```
misaka@ubuntu:~/Documents/project5/threadpool$ make
gcc -Wall -c client.c -lpthread
gcc -Wall -o example client.o threadpool.o -lpthread
misaka@ubuntu:~/Documents/project5/threadpool$ ./example
[ThreadPool] Initialize successfully!
I add two values 33 and 36 result = 69
I add two values 27 and 15 result = 42
I add two values 43 and 35 result = 78
I add two values 36 and 42 result = 78
I add two values 49 and 21 result = 70
I add two values 12 and 27 result = 39
I add two values 40 and 9 result = 49
I add two values 13 and 26 result = 39
I add two values 40 and 26 result = 66
I add two values 22 and 36 result = 58
I add two values 11 and 18 result = 29
I add two values 17 and 29 result = 46
I add two values 32 and 30 result = 62
I add two values 12 and 23 result = 35
I add two values 17 and 35 result = 52
I add two values 29 and 2 result = 31
I add two values 22 and 8 result = 30
I add two values 19 and 17 result = 36
I add two values 43 and 6 result = 49
I add two values 11 and 42 result = 53
I add two values 29 and 23 result = 52
I add two values 21 and 19 result = 40
I add two values 34 and 37 result = 71
I add two values 48 and 24 result = 72
I add two values 15 and 20 result = 35
I add two values 13 and 26 result = 39
I add two values 41 and 30 result = 71
I add two values 6 and 23 result = 29
I add two values 12 and 20 result = 32
I add two values 46 and 31 result = 77
[ThreadPool] Shutdown successfully!
```

图 1: Thread Pool

2 The Producer–Consumer Problem

In this project, you will design a programming solution to the bounded-buffer problem using the producer and consumer processes shown in Figures 5.9 and 5.10. 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. The producer and consumer—running as separate threads—will move items to and from a buffer that is synchronized with the empty, full, and mutex structures. You can solve this problem using either Pthreads or the Windows API.

Design: My design for this task is shown as follows:

- Semaphore `empty` and `full` is used to track the items in the buffer. `empty` represents the number of empty units in the buffer. `full` represents the number of full units in the buffer.
- The buffer is implemented using a circular queue.
- The producer will produce an item after a random time (from 0 to 4 seconds). The consumer will consume an item after a random time (from 0 to 4 seconds).

- The whole program will run for **time_all** seconds. There will be **producer_all** producer and **consumer_all** consumer.

The code of `buffer.c` is shown as follows.

```
#include "buffer.h"

buffer_item buffer[max_buf + 1];
int head, tail;

int insert_item(buffer_item item) {
    if (head == (tail + 1) % (max_buf + 1)) return -1; // buffer is full
    tail = (tail + 1) % (max_buf + 1);
    buffer[tail] = item;
    return 0;
}

int remove_item(buffer_item *item) {
    if (head == tail) return -1;
    head = (head + 1) % (max_buf + 1);
    *item = buffer[head];
    return 0;
}

void initial_buffer() {
    head = 0;
    tail = 0;
}
```

The code of `producer-consumer.c` is shown as follows.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#include "buffer.h"


int running = 0; // 0 -> running; 1 -> not running;

pthread_mutex_t mutex;
sem_t empty;
sem_t full;


void *producer(void *arg) {
    buffer_item item;

    while (1) {
        sleep(rand() % 5);
        sem_wait(&empty);
        pthread_mutex_lock(&mutex);
```

```c

        if (running == 1) break;
        item = rand();
        if (insert_item(item) != -1) {
            fprintf(stdout, "[Producer] %d is produced.\n", item);
        } else {
            fprintf(stderr, "[Error] unreachable!\n");
                        exit(1);
        }

        pthread_mutex_unlock(&mutex);
                sem_post(&full);
    }
    pthread_mutex_unlock(&mutex); // for terminating the program
        pthread_exit(0);
}

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

    while (1) {
        sleep(rand() % 5);
        sem_wait(&full);
        pthread_mutex_lock(&mutex);

        if (running == 1) break;
        if (remove_item(&item) != -1) {
            fprintf(stdout, "[Consumer] %d is consumed.\n", item);
        } else {
            fprintf(stderr, "[Error] unreachable!\n");
                        exit(1);
        }

        pthread_mutex_unlock(&mutex);
                sem_post(&empty);
    }
    pthread_mutex_unlock(&mutex); // for terminating the program
        pthread_exit(0);
}

int main(int argc, char *argv[]) {
    pthread_t *producer_thread, *consumer_thread;

    if (argc != 4) {
        fprintf(stderr, "[Error] Input should be like 'producer-consumer 10 5 5'.\n");
                exit(1);
    }

    int time_all = atoi(argv[1]);
    int producer_all = atoi(argv[2]);
    int consumer_all = atoi(argv[3]);
```

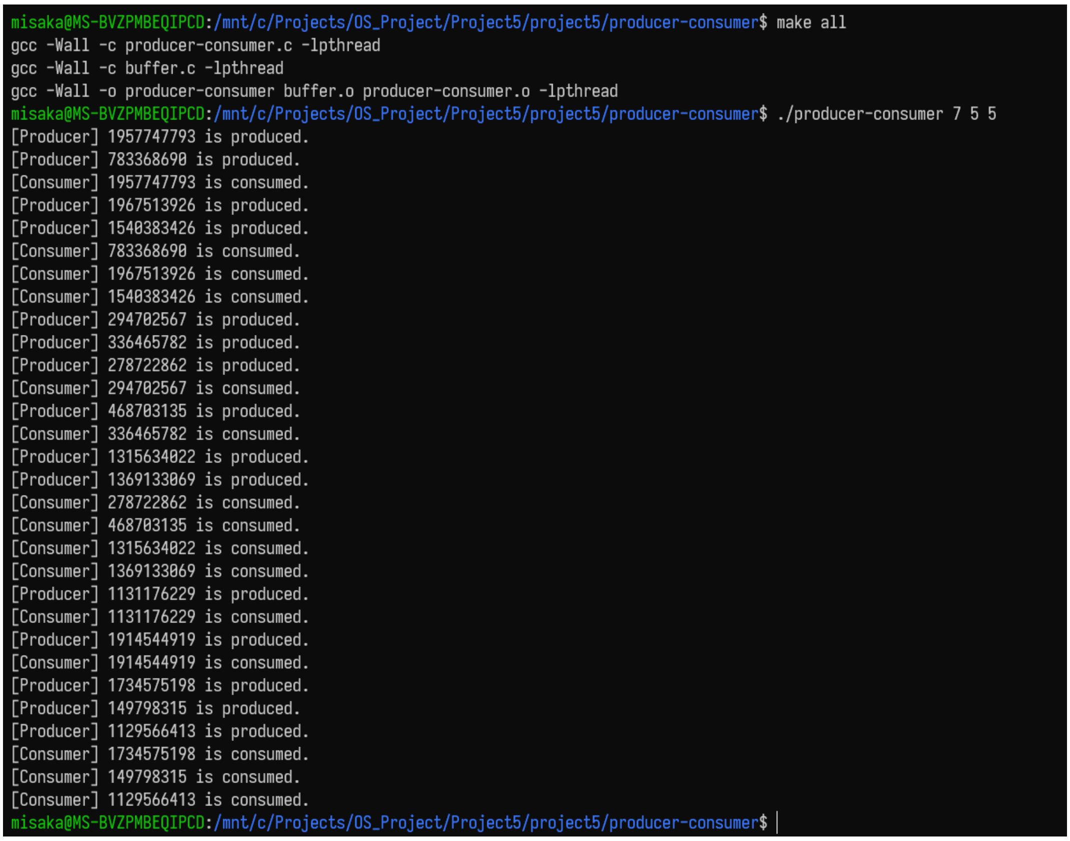
```

    initial_buffer();
    pthread_mutex_init(&mutex, NULL);
    sem_init(&empty, 0, max_buf);
    sem_init(&full, 0, 0);

    producer_thread = (pthread_t *) malloc (sizeof(pthread_t) * producer_all);
    consumer_thread = (pthread_t *) malloc (sizeof(pthread_t) * consumer_all);

    for (int i = 0; i < producer_all; i++)
                pthread_create(&producer_thread[i], NULL, &producer, NULL);
        for (int i = 0; i < consumer_all; i++)
                pthread_create(&consumer_thread[i], NULL, &consumer, NULL);

    sleep(time_all);

    running = 1;

    for (int i = 0; i < producer_all; i++)
                sem_post(&empty);
        for (int i = 0; i < consumer_all; i++)
                sem_post(&full);

    for (int i = 0; i < producer_all; i++)
                pthread_join(producer_thread[i], NULL);
        for (int i = 0; i < consumer_all; i++)
                pthread_join(consumer_thread[i], NULL);

    sem_destroy(&empty);
    sem_destroy(&full);
    pthread_mutex_destroy(&mutex);

    free(producer_thread);
    free(consumer_thread);
    return 0;
}
```

Makefile for producer consumer Problem is shown as follow.

```
CC=gcc
CFLAGS=-Wall
PTHREADS=-lpthread

all: producer-consumer.o buffer.o
	$(CC) $(CFLAGS) -o producer-consumer buffer.o producer-consumer.o $(PTHREADS)

buffer.o: buffer.c
	$(CC) $(CFLAGS) -c buffer.c $(PTHREADS)

producer-consumer.o: producer-consumer.c
	$(CC) $(CFLAGS) -c producer-consumer.c $(PTHREADS)
```

```

clean:
	rm -rf *.o
	rm -rf producer-consumer
```

The execution result is shown as follow.



```
misaka@MS-BVZPMBEQIPCD:/mnt/c/Projects/OS_Project/Project5/project5/producer-consumer$ make all
gcc -Wall -c producer-consumer.c -lpthread
gcc -Wall -c buffer.c -lpthread
gcc -Wall -o producer-consumer buffer.o producer-consumer.o -lpthread
misaka@MS-BVZPMBEQIPCD:/mnt/c/Projects/OS_Project/Project5/project5/producer-consumer$ ./producer-consumer 7 5 5
[Producer] 1957747793 is produced.
[Producer] 783368690 is produced.
[Consumer] 1957747793 is consumed.
[Producer] 1967513926 is produced.
[Producer] 1540383426 is produced.
[Consumer] 783368690 is consumed.
[Consumer] 1967513926 is consumed.
[Consumer] 1540383426 is consumed.
[Producer] 294702567 is produced.
[Producer] 336465782 is produced.
[Producer] 278722862 is produced.
[Consumer] 294702567 is consumed.
[Producer] 468703135 is produced.
[Consumer] 336465782 is consumed.
[Producer] 1315634022 is produced.
[Producer] 1369133069 is produced.
[Consumer] 278722862 is consumed.
[Consumer] 468703135 is consumed.
[Consumer] 1315634022 is consumed.
[Consumer] 1369133069 is consumed.
[Producer] 1131176229 is produced.
[Consumer] 1131176229 is consumed.
[Producer] 1914544919 is produced.
[Consumer] 1914544919 is consumed.
[Producer] 1734575198 is produced.
[Producer] 149798315 is produced.
[Producer] 1129566413 is produced.
[Consumer] 1734575198 is consumed.
[Consumer] 149798315 is consumed.
[Consumer] 1129566413 is consumed.
misaka@MS-BVZPMBEQIPCD:/mnt/c/Projects/OS_Project/Project5/project5/producer-consumer$
```

图 2: Producer Consumer Problem