CS356 Project 5

Name : HaotianXue

Student ID : 518021910506

Email : xavihart@sjtu.edu.cn

CS356 Project 5
- Introduction
- Thread Pool
  - General Ideas
  - Implementation Details
    - Thread Pool
    - Task Queue
    - Mutex and Semaphore
    - Pool Submit
    - Code Structure
  - Evaluation
- Producer-Consumer Problem
  - General Ideas
  - Implementation Details
    - Mutex and Semaphore
    - Code Structure
  - Evaluation
- Conclusion

Introduction

This is the report for project5 for CS356, also for COS book chapter7.

The project is composed of two sections, both of which using synchronizing tools to solve critical section problems in real problems. To be specific, the first task encourages us to implement a thread pool and the second task encourage us to give a practical solution to the producer-consumer problem.

For both problems, we use POSIX system to implement using C in Linux.

Thread Pool

General Ideas

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.

Implementation Details

Thread Pool

The thread pool is a array a `pthread` which has a fixed size of three, which means we have at most three tasks working together. And each time a task is submitted, we check the pool repeatedly and to find a free pool, where we use a `work` array to note whether the pool is in work.

Task Queue

A queue is used to place the awaiting tasks, and the queue element is task structure which is composed to the task function and parameters.

Mutex and Semaphore

We have one mutex `QueueMutex` to make the operation to queue, including `enqueue` and `dequeue` , mutual exclusive to each other.

Also we have a semaphore `ThreadSem` which is initialized to the number of thread numbers to restrict the number of threads working in the pool simultaneously.

Pool Submit

This is the key function of the implementation, each time a task is submitted, we should first wait for the `ThreadSem` to find some thread which is free, after finding a free thread in the pool, we create a thread and dequeue a task to run it, lastly we signal the semaphore to make other task able to enter the working area.

Code Structure

The code for the thread pool is as follows:

```
#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
```

```c
// completed by a thread in the pool
typedef struct
{
    void (*function)(void *p);
    void *data;
}
task;

// the work queue
task worktodo;
pthread_mutex_t QueueMutex;
sem_t ThreadSem;
pthread_t pool[NUMBER_OF_THREADS];
int work[NUMBER_OF_THREADS];
task TaskQueue[QUEUE_SIZE];
int QueueSize;

// insert a task into the queue
// returns 0 if successful or 1 otherwise,
int enqueue(task t)
{
    if(QueueSize < QUEUE_SIZE){
        pthread_mutex_lock(&QueueMutex);
        TaskQueue[QueueSize] = t;
        QueueSize++;
        pthread_mutex_unlock(&QueueMutex);
    }else{
        return 1;
    }
    return 0;
}

// remove a task from the queue
task dequeue()
{
    if(QueueSize < 1){
        worktodo.data = NULL;
        worktodo.function = NULL;
    }else{
        pthread_mutex_lock(&QueueMutex);
        worktodo = TaskQueue[0];
        QueueSize--;
        for (int i = 0; i < QueueSize; ++i)
            TaskQueue[i] = TaskQueue[i + 1];
        TaskQueue[QueueSize].function = NULL;
        TaskQueue[QueueSize].data = NULL;
        pthread_mutex_unlock(&QueueMutex);
    }

    return worktodo;
}
```

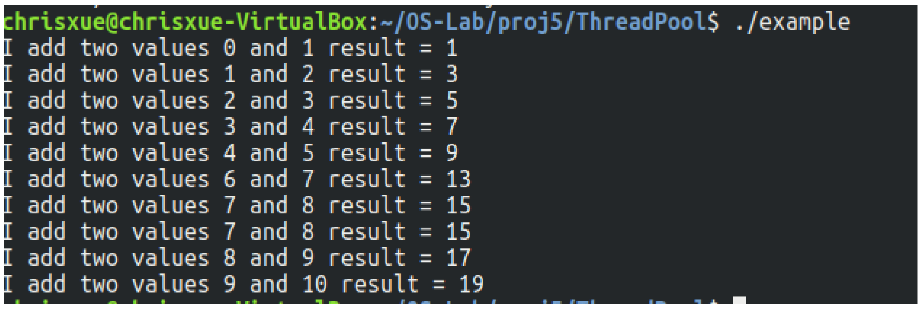
```c
// the worker thread in the thread pool
void *worker(void *param)
{
    // execute the task
    int *i = (int*) param;
    int id = *i;
    worktodo = dequeue();
    execute(worktodo.function, worktodo.data);
    work[id] = 0;
    sem_post(&ThreadSem);
    pthread_exit(0);
}

/**
 * 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)
{
    worktodo.function = somefunction;
    worktodo.data = p;
    int f = enqueue(worktodo);
    int find = 0;
    if(!f){
        sem_wait(&ThreadSem);
        while(1){
            //printf("%d\n", find);
            if(work[find] == 0){
                work[find] = 1;
                break;
            }else{
                if(find == NUMBER_OF_THREADS - 1) {find =
0;continue;}
                find ++;
            }
        }
    }

    pthread_create(&pool[find], NULL, worker, &find);

    return 0;
}

// initialize the thread pool
void pool_init(void)
```

```
{
    QueueSize = 0;
    for (int i = 0; i < NUMBER_OF_THREADS; ++i){
        work[i] = 0;
    }
    sem_init(&ThreadSem, 0, NUMBER_OF_THREADS);
    pthread_mutex_init(&QueueMutex, NULL);
}

// shutdown the thread pool
void pool_shutdown(void)
{
    for (int i = 0; i < NUMBER_OF_THREADS; ++i)
        pthread_join(pool[i], NULL);
    sem_destroy(&ThreadSem);
    pthread_mutex_destroy(&QueueMutex);
}
```

Code for the main function is as follows, where we summit tasks and test the correctness of them:

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

struct data
{
    int a;
    int b;
};

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[10];
    for(int i = 0;i < 10;++i){
        work[i].a = i;
        work[i].b = i + 1;
    }

    // initialize the thread pool
    pool_init();
    // submit the work to the queue
    for(int i = 0;i < 10;++i){
        pool_submit(&add, &work[i]);
    }
    pool_shutdown();
    return 0;
}
```

Evaluation



```
chrisxue@chrisxue-VirtualBox:~/OS-Lab/proj5/ThreadPool$ ./example
I add two values 0 and 1 result = 1
I add two values 1 and 2 result = 3
I add two values 2 and 3 result = 5
I add two values 3 and 4 result = 7
I add two values 4 and 5 result = 9
I add two values 6 and 7 result = 13
I add two values 7 and 8 result = 15
I add two values 7 and 8 result = 15
I add two values 8 and 9 result = 17
I add two values 9 and 10 result = 19
```

Producer-Consumer Problem

General Ideas

The Producer-Consumer problem is a typical problem in synchronizing programming. We have producer which add elements in the buffer and consumers which fetch elements from the buffer. We should use synchronizing tools to make the following pairs mutual exclusive:

- Producers and Producers
- Producers and Consumers
- Consumers and Consumers

Implementation Details

Mutex and Semaphore

We use two semaphores and one mutex lock in the implementation:

- `mutex`: a mutex lock used to gain entry to change the buffer
- `full` : a semaphore used to see if the buffer is full, initialized to 0,full also refer to the used space in the buffer
- `empty` : a semaphore used to see if the buffer is empty, initialized to size of buffer,empty also refer to the free space in the buffer

Code Structure

```
#include"buffer.h"
```

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <semaphore.h>

buffer_item buffer[BUFFER_SIZE];

pthread_mutex_t mutex;
sem_t full;     // set to 0
sem_t empty;    // set to BUFFER_SIZE
int size;

void init(){
    sem_init(&full, 0, 0);
    sem_init(&empty, 0, BUFFER_SIZE);
    pthread_mutex_init(&mutex, NULL);
    size = 0;
}

int Insert(buffer_item s){
    sem_wait(&empty);
    pthread_mutex_lock(&mutex);
    //printf("INSERT\n");
    size ++;
    buffer[size - 1] = s;
    pthread_mutex_unlock(&mutex);
    sem_post(&full);
    return 0;
}
int Remove(buffer_item *t){
    sem_wait(&full);
    pthread_mutex_lock(&mutex);
    //printf("REMOVE\n");
    *t = buffer[0];
    size --;
    for(int i = 0;i < size;++i){
        buffer[i] = buffer[i + 1];
    }
    pthread_mutex_unlock(&mutex);
    sem_post(&empty);
    return 0;
}
```

We also have a main code which are used to generate threads to run producer and consumer operations, and we output the operation to evaluate them:

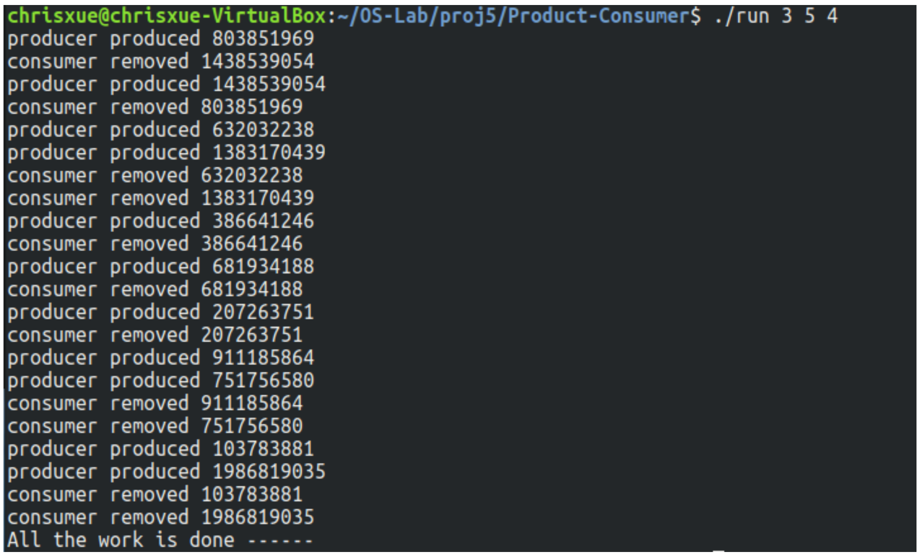
```
#include "buffer.h"
#include <pthread.h>
#include<stdlib.h>
#include <stdio.h>
#include <assert.h>
```

```
#include<unistd.h>

int SleepTime, NumCons, NumProd;
void* producer(void*param);
void* consumer(void*param);

int main(int argc, char*argv[]){
    assert(argc == 4);
    SleepTime = atoi(argv[1]);
    NumCons = atoi(argv[2]);
    NumProd = atoi(argv[3]);
    init();
    srand((int)time(0));
    pthread_t ConsThreads[NumCons];
    pthread_t ProdThreads[NumProd];
       for(int i = 0;i < NumProd;++i){
        //printf("%d\n", i);
        pthread_create(&ProdThreads[i], NULL, producer,
NULL);
    }
    for(int i = 0;i < NumCons;++i){
        //printf("%d\n", i);
        pthread_create(&ConsThreads[i], NULL, consumer,
NULL);
    }
    sleep(SleepTime) ;
    for(int i = 0;i < NumCons;++i){
        //printf("%d\n", i);
        pthread_cancel(ConsThreads[i]);
    }
    for(int i = 0;i < NumProd;++i){
       pthread_cancel(ProdThreads[i]);
    }
    printf("All the work is done ------\n");
    return 0;
}

void* producer(void*param){
    buffer_item item;
    while(1){
        int sleep_time = rand() % 3;
        sleep(sleep_time);
        item = rand();
        //printf("%d\n", item);
        if(Insert(item)){
            printf("producer error!\n");
        }else{
            printf("producer produced %d\n", item);
        }
    }
}
```

```
void* consumer(void*param){
    buffer_item item;
    while(1){
        int sleep_time = rand() % 3;
        sleep(sleep_time);
        if(Remove(&item)){
            printf("consumer error!\n");
        }else{
            printf("consumer removed %d\n", item);
        }
    }
}
```

Evaluation

We set 5 consumers and 4 producers in the evaluation, and the sleep time is set to three, which means the work will be terminated after three seconds so the change to the buffer will not be infinitely happening.



```
chrisxue@chrisxue-VirtualBox:~/OS-Lab/proj5/Product-Consumer$ ./run 3 5 4
producer produced 803851969
consumer removed 1438539054
producer produced 1438539054
consumer removed 803851969
producer produced 632032238
producer produced 1383170439
consumer removed 632032238
consumer removed 1383170439
producer produced 386641246
consumer removed 386641246
producer produced 681934188
consumer removed 681934188
producer produced 207263751
consumer removed 207263751
producer produced 911185864
producer produced 751756580
consumer removed 911185864
consumer removed 751756580
producer produced 103783881
producer produced 1986819035
consumer removed 103783881
consumer removed 1986819035
All the work is done ------
```

Conclusion

This project helps me review what we learnt in class about thread pool tech and the synchronizing tools. We also learnt to use POSIX synchronizing tools such as `pthread_mutex`_t and `sem_t` and there operation in C implementation in Linux. During the project work, I learnt how to use `gdb` and `core` to debug especially for segmentation fragment faults. And I also learn more about writing `Makefile` for C program.

Thanks for the instructions and useful help offered by Prof. Wu and all the TAs!