操作系统课程设计 Project 5
Designing a Thread Pool
& Producer-Consumer Problem

姓名：郭倩昀
班级：F1903303
学号：519021910095
Email：guoqianyun@sjtu.edu.cn

2021 年 5 月 22 日

目录

1 Designing a Thread Pool **2**
1.1 实验内容与目标 2
1.2 实验过程及步骤 2
1.3 实验代码 2
1.4 实验测试 7

2 Producer-Consumer Problem **7**
2.1 实验内容与目标 7
2.2 实验过程及步骤 7
2.3 实验代码 8
2.4 实验测试 13

3 Conclusion **13**
3.1 问题与解决方案 13
3.2 实验心得 13

1 Designing a Thread Pool

1.1 实验内容与目标

本实验需要利用 C 语言创建并管理 thread pool，根据提供的源文件 Makefile 以及 client，thread-pool 文件，要求我们完善 threadpool.c 文件中的 enqueue，dequeue，worker，pool_submit，pool_init，pool_shutdown 等函数。

1.2 实验过程及步骤

- 将 work queue 设计为链接表
  设计队列结点结构体并创建头结点和尾结点。
- 利用同步工具
  创建 mutex 为访问 work queue 的时候避免竞争条件，创建信号量用于挂起等待线程。
- 完善 enqueue 与 dequeue 函数
  enqueue 根据传入的 task 创建 queue node 加入到任务列表，dequeue 将头结点指向的任务删除并返回。
- 完善 worker 函数
  work thread 的工作函数，当有等待线程的时候将任务取出并执行，注意 mutex 和信号量的使用。结束后用 pthread_exit(0) 返回。
- 完善 pool_submit 函数
  根据参数所给的函数信息和 data 信息创建 task，加入到任务列表，注意 mutex 和信号量的使用。
- 完善 pool_init 函数
  线程池初始化时候，给任务列表头结点分配空间，初始化头结点和尾结点，初始化 mutex 和信号量，为每个线程调用 pthread_create 创建线程并输出创建成功的信息。过程中用 err 记录关键步骤的执行情况，当出现错误时候及时报错并退出执行。
- 完善 pool_shutdown 函数
  将全局变量 shutdown 设为 1，让线程不再继续接受新的任务。然后取消所有的信号量，等待线程执行完毕并输出执行完毕的信息，最后破坏 mutex 和信号量结束工作。
- 修改 client.c 文件便于测试
  原来的 client.c 文件只有一个待完成的 work，经修改后设定为随机生成 10 个 work 并递交给线程池工作。

1.3 实验代码

threadpool.c

```
/**
 * Implementation of thread pool.
 */

```

```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;

// the work queue
struct queue_node{  //linked list
    task worktodo;
    struct queue_node *next;
};
struct queue_node *head,*tail;


pthread_t bee[NUMBER_OF_THREADS];   // the worker bee
pthread_mutex_t queue_mutex;        //mutex
sem_t sem;              //semaphore
int shutdown;

// insert a task into the queue
// returns 0 if successful or 1 otherwise,
int enqueue(task t)
{
    tail->next=(struct queue_node *) malloc (sizeof(struct queue_node));
    if(tail->next==NULL) return 1;  //allocation error occurs
    tail=tail->next;                //add to the end
    tail->worktodo=t;
    return 0;
}

// remove a task from the queue
task dequeue()
{
    if(head==tail)                  //queue is empty
    {
        fprintf(stderr,"ERROR:no work to do\n");
        exit(1);
    }
    else
    {
        struct queue_node *tmp;     //remove from the head
        tmp=head;
        head=head->next;
```

```
        free(tmp);
    }
    return head->worktodo;
}

// the worker thread in the thread pool
void *worker(void *param)           //exe by each thread in the pool
{
    task tsk;
    while(TRUE)
    {
        sem_wait(&sem);             //notifying a waiting thread
        if(shutdown) break;
        pthread_mutex_lock(&queue_mutex);//avoid race conditions when accessing queue
        tsk=dequeue();
        pthread_mutex_unlock(&queue_mutex);
        execute(tsk.function, tsk.data);// execute the task
    }
    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)
{
    task tsk;       //place function and data
    tsk.function=somefunction;
    tsk.data=p;
    pthread_mutex_lock(&queue_mutex);//avoid race conditions when accessing queue
    int rst=enqueue(tsk);
    pthread_mutex_unlock(&queue_mutex);
    if(rst==0) sem_post(&sem);
    return rst;
}

// initialize the thread pool
void pool_init(void)
{
    int err;
    shutdown=0;
    head = (struct queue_node *) malloc (sizeof(struct queue_node));
    if(head==NULL)
    {
        fprintf(stderr,"ERROR:queue init error\n");
        exit(1);
    }
    head->next=NULL;
    tail=head;
```

```
119
120     err=pthread_mutex_init(&queue_mutex,NULL);  //initialize mutex
121     if(err)
122     {
123         fprintf(stderr,"ERROR:pthread mutex init error\n");
124         exit(1);
125     }
126     err=sem_init(&sem,0,0);                     //initialize semaphore
127     if(err)
128     {
129         fprintf(stderr,"ERROR:semaphore init error\n");
130         exit(1);
131     }
132     for(int i=0;i<NUMBER_OF_THREADS;i++)        //pthread create
133     {
134         err=pthread_create(&bee[i],NULL,worker,NULL);
135         if(err)
136         {
137             fprintf(stderr,"ERROR:pthread create error\n");
138             exit(1);
139         }
140     }
141     fprintf(stdout,"Pthread create successfully\n");
142 }
143
144 // shutdown the thread pool
145 void pool_shutdown(void)
146 {
147     int err;
148     shutdown=1;
149     for(int i=0;i<NUMBER_OF_THREADS;i++)        //cancel each worker thread
150     {
151         sem_post(&sem);
152     }
153     for(int i=0;i<NUMBER_OF_THREADS;i++)        //wait for each thread to terminate
154     {
155         err=pthread_join(bee[i],NULL);
156         if(err)
157         {
158             fprintf(stderr,"ERROR:pthread join error\n");
159             exit(1);
160         }
161     }
162     fprintf(stdout,"Pthread join successfully\n");
163     err=pthread_mutex_destroy(&queue_mutex);    //destroy mutex
164     if(err)
165     {
166         fprintf(stderr,"ERROR:pthread mutex destroy error\n");
167         exit(1);
168     }
169     err=sem_destroy(&sem);                      //destroy semaphore
170     if(err)
171     {
172         fprintf(stderr,"ERROR:semaphore destroy error\n");
173         exit(1);
174     }
175 }
```

client.c

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

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <time.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
    srand((unsigned)time(NULL));
    struct data work[10];
    for (int i = 0; i < 10; ++ i) {
        work[i].a = rand() % 100;
        work[i].b = rand() % 100;
    }


    // initialize the thread pool
    pool_init();

    // submit the work to the queue
    for (int i = 0; i < 10; ++ i) {
        static int err;
        err = pool_submit(&add, &work[i]);
        if (err)
        fprintf(stderr, "Submit work %d error because the task queue is full.\n", i);
    }

    sleep(3);

    pool_shutdown();

    return 0;
}
```

1.4 实验测试

- thread pool 测试 (图 1)

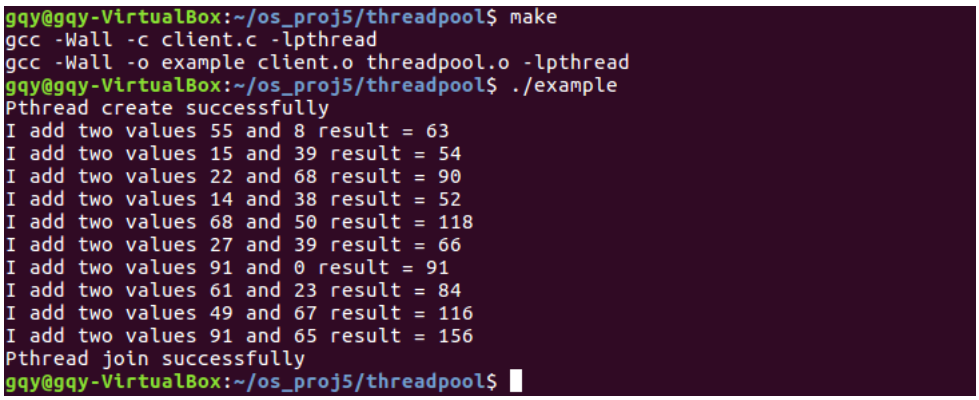

图 1: thread pool 测试

测试指令如下

```
make
./example
```

首先用 Makefile 文件编译，生成 example 可执行文件，输入./example 并执行，测试生成的 10 个 work 递交给线程池工作，测试结果如图 1。

2 Producer-Consumer Problem

2.1 实验内容与目标

本实验需要利用 C 语言结合 mutex，信号量和线程使用应对生产者-消费者问题。需要设计 buffer 以及 producer_consumer.c 文件实现，参考书本 7.1.1 的方式利用同步工具，针对每一个生产者和消费者创建相应的线程工作。

2.2 实验过程及步骤

- buffer 设计
  根据书本指导，设计的 buffer 包含函数 insert_item()，remove_item()，另外添加了一个初始化函数 buffer_init()。buffer 的数组大小设为比规定大小多一位，头指针和尾指针分别指向 buffer 中第一个元素的前一位以及最后一个元素。初始化函数将头指针和尾指针设为 0。insert_item 将元素插入到尾指针后并更新尾指针，remove_item 将头指针的元素传出，如果遇到 buffer 满或空导致不能进行加减元素则返回 -1 表示异常。

- 同步工具的使用
  producer_consumer.c 中创建 mutex 和信号量 empty 和 full。mutex 用于在访问 buffer 的时候避免竞争条件。empty 和 full 用于挂起等待的消费者或者生产者线程。empty 初始化为 BUFFER_SIZE，full 初始化为 0。

- 为生产者和消费者设计线程工作函数
  每个生产者每次间隔随机时间，然后生成随机 item 并通过 insert_item 加入到 buffer 中；每个消费者每次间隔随机时间，然后通过 remove_item 从 buffer 中消费 item。注意中间的 mutex 和信号量的使用，工作时如果终止信号 terminate 为 1 则停止工作退出，如果出现错误情况及时报错。结束工作后调用 pthread_exit(0) 退出。

- main() 函数设计
  首先通过命令行读入三个参数，分别为终止前运行时间，生产者个数和消费者个数。然后分别初始化 buffer，mutex，full，empty（注意初始值设置），为每一个生产者和消费者线程分配空间并调用 pthread_create 创建并输出创建成功的信息，传入工作函数。运行时间结束后将终止信号 terminate 设为 1，让所有线程停止工作并退出，调用 pthread_join 等待线程结束工作并输出结束信息，最后破坏 mutex 和信号量结束工作。

2.3 实验代码

buffer.h

```
/* buffer.h */
typedef int buffer_item;
#define BUFFER_SIZE 5
int insert_item(buffer_item item);
int remove_item(buffer_item *item);
void buffer_init();
```

buffer.c

```
# include "buffer.h"

buffer_item buffer[BUFFER_SIZE+1];
int head;//before the first one in the buffer
int tail;//last one in the buffer

void buffer_init()              //buffer initialize
{
    head=0;
    tail=0;
}

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

int remove_item(buffer_item *item)
{
    if(head==tail)              //empty
        return -1;
```

```
    head=(head+1)%(BUFFER_SIZE+1);
    *item=buffer[head];
    return 0;
}
```

producer_consumer.c

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

#define TRUE 1
#define RANDOM_TIME_BASE 5
int terminate;
pthread_mutex_t mutex;//for accessing buffer
sem_t empty, full;

void *producer(void *param);

void *consumer(void *param);

int main(int argc, char *argv[]){
    srand((unsigned)time(NULL));

    terminate=0;
    pthread_t *producer_t, *consumer_t;
    int time, producer_number, consumer_number;
    static int err;

//get command line
    if(argc!=4)
    {
        fprintf(stderr,"ERROR:invalid arguments\n");
        exit(1);
    }
    time=atoi(argv[1]);
    producer_number=atoi(argv[2]);
    consumer_number=atoi(argv[3]);

//initialize the buffer
    buffer_init();

    //initialize the mutex lock
    err=pthread_mutex_init(&mutex, NULL);
    if(err)
    {
        fprintf(stderr,"ERROR:pthread mutex init create error\n");
        exit(1);
    }

    //initialize the semaphore
    err=sem_init(&empty, 0, BUFFER_SIZE);//empty initialize to n
```

```
    if(err)
    {
        fprintf(stderr,"ERROR:semaphore init create error\n");
        exit(1);
    }
    err=sem_init(&full, 0, 0);//full initialize to 0
    if(err)
    {
        fprintf(stderr,"ERROR:semaphore init create error\n");
        exit(1);
    }

//create producer threads
    producer_t = (pthread_t*)malloc(sizeof(pthread_t)*producer_number);
    for(int i=0;i<producer_number;i++)
    {
        pthread_create(&producer_t[i],NULL,&producer,NULL);
    }

//create consumer threads
    consumer_t = (pthread_t*)malloc(sizeof(pthread_t)*consumer_number);
    for(int i=0;i<consumer_number;i++)
    {
        pthread_create(&consumer_t[i],NULL,&consumer,NULL);
    }
    fprintf(stdout,"Pthread create successfully\n");
//sleep
    sleep(time);

//terminate
    terminate=1;
    //cancel each thread
    for(int i=0;i<producer_number;i++)
    {
        sem_post(&empty);
    }
    for(int i=0;i<consumer_number;i++)
    {
        sem_post(&full);
    }
    //pthread join
    for(int i=0;i<producer_number;i++)
    {
        err=pthread_join(producer_t[i],NULL);
        if(err)
        {
            fprintf(stderr,"ERROR:pthread join error\n");
            exit(1);
        }
    }
    for(int i=0;i<consumer_number;i++)
    {
        err=pthread_join(consumer_t[i],NULL);
        if(err)
        {
            fprintf(stderr,"ERROR:pthread join error\n");
            exit(1);
```

```
        }
    }
    fprintf(stdout,"Pthread join successfully\n");

    //destroy mutex
    err=pthread_mutex_destroy(&mutex);
    if(err)
    {
        fprintf(stderr,"ERROR:pthread mutex destroy error\n");
        exit(1);
    }

    //destroy semaphore
    err=sem_destroy(&empty);
    if(err)
    {
        fprintf(stderr,"ERROR:semaphore destroy error\n");
        exit(1);
    }
    err=sem_destroy(&full);
    if(err)
    {
        fprintf(stderr,"ERROR:semaphore destroy error\n");
        exit(1);
    }
    free(producer_t);
    free(consumer_t);

    return 0;
}

void *producer(void *param)
{
    buffer_item item;
    while(TRUE)
    {
        //sleep for a random period of time
        int random_time;
        random_time=rand()%RANDOM_TIME_BASE;
        sleep(random_time);

        //generate a random number
        item=rand();

        sem_wait(&empty);
        pthread_mutex_lock(&mutex);
        if(terminate) break;
        if(insert_item(item))
            fprintf(stderr,"ERROR: cannot insert item %d\n", item);
        else
            fprintf(stdout,"Producer produced item %d\n", item);
        pthread_mutex_unlock(&mutex);
        sem_post(&full);
    }
    pthread_mutex_unlock(&mutex);
    pthread_exit(0);
}
```

```
164
165 void *consumer(void *param)
166 {
167     buffer_item item;
168     while(TRUE)
169     {
170         //sleep for a random period of time
171         int random_time;
172         random_time=rand()%RANDOM_TIME_BASE;
173         sleep(random_time);
174
175         sem_wait(&full);
176         pthread_mutex_lock(&mutex);
177         if(terminate) break;
178         if(remove_item(&item))
179             fprintf(stderr,"ERROR: cannot remove item %d\n", item);
180         else
181             fprintf(stdout,"Consumer consumed item %d\n", item);
182         pthread_mutex_unlock(&mutex);
183         sem_post(&empty);
184     }
185     pthread_mutex_unlock(&mutex);
186     pthread_exit(0);
187 }
```

```
gqy@gqy-VirtualBox:~/os_proj5$ make
gcc -Wall -c producer_consumer.c -lpthread
gcc -Wall -o producer_consumer producer_consumer.o buffer.o -lpthread
gqy@gqy-VirtualBox:~/os_proj5$ ./producer_consumer 9 5 3
Pthread create successfully
Producer produced item 975417950
Producer produced item 1312890156
Consumer consumed item 975417950
Producer produced item 682110775
Producer produced item 331429345
Producer produced item 1567444831
Producer produced item 1567059540
Consumer consumed item 1312890156
Consumer consumed item 682110775
Producer produced item 397591093
Producer produced item 202316346
Consumer consumed item 331429345
Consumer consumed item 1567444831
Producer produced item 476086270
Consumer consumed item 1567059540
Producer produced item 1765865338
Producer produced item 134915866
Consumer consumed item 397591093
Consumer consumed item 202316346
Producer produced item 1340493675
Producer produced item 549164725
Consumer consumed item 476086270
Producer produced item 1140283042
Consumer consumed item 1765865338
Producer produced item 1743652646
Consumer consumed item 134915866
Producer produced item 1471712388
Consumer consumed item 1340493675
Producer produced item 1893291665
Pthread join successfully
gqy@gqy-VirtualBox:~/os_proj5$
```

图 2: producer consumer 测试

2.4 实验测试

- producer consumer 测试 (图 2) 测试指令如下

```
make
./producer_consumer 9 5 3
```

首先用 Makefile 文件编译，生成可执行文件 producer_consumer，输入运行时间 9s，生产者 5 个，消费者 3 个并执行，测试结果如图 2。

3 Conclusion

3.1 问题与解决方案

由于经过 project4 后对线程应用比较熟悉，本次 project 对线程的应用难度不大，主要困难点就在于同步工具的使用，包括 mutex 和信号量的使用，需要对同步工具有透彻的理解并清楚了解程序中每一个创建的同步工具的具体用处，并正确运用。原本理论学习的时候对这些同步工具的理解并没有那么深入，在本次 project 设计中重新仔细阅读书上的示例并经过多次尝试后成功正确应用了同步工具完成了实验内容。

3.2 实验心得

本次 project5 让我进一步熟悉了线程的运用，与此同时增加了同步工具的使用，让我对 mutex，信号量等工具有了更清晰的认识。虽然由于运用不熟练过程中出现了一些错误，但是最终多次尝试后调试成功，可以见得除了理论学习外，最终还是要多动手亲自来实现才可以真正掌握新的知识和工具。总的来说本次 project 很好地锻炼了代码能力并加深了对进程同步工具的理解，让我受益匪浅。