Lab Report of Project 5
Designing a Thread Pool & Producer-Consumer Problem

Name: Li Dongyue Student Number: 516030910502
Computer System Engineering (Undergraduate), Fall 2018

December 1, 2018

1 Designing a Thread Pool

1.1 Introduction

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. I complete it using `Pthreads` and `POSIX` synchronization. Below I provide the details about implementing a thread pool.

1.2 Thread Pool Implementation

Basically we implement the thread in the following ways. We create a queue to store submitted tasks. Threads in the thread pool are taking tasks out from the queue and execute the task independently.

To implement a queue, I use a static queue by creating a array to store tasks. A index number `curTaskNum` indicates the queue head. Followed basic static queue structure, the `enqueue()` and `dequeue()` are implemented as follows:

```
// insert a task into the queue
// returns 0 if successful or 1 otherwise,
int enqueue(task t)
{
    pthread_mutex_lock(&mutex);
    if (curTaskNum == QUEUE_SIZE)
    {
        pthread_mutex_unlock(&mutex);
        return 1;
    }
    worktodo[curTaskNum++] = t;
    pthread_mutex_unlock(&mutex);
    return 0;
}

// remove a task from the queue
task* dequeue()
{
    pthread_mutex_lock(&mutex);
    if (curTaskNum==0)
    {
        pthread_mutex_unlock(&mutex);
        return NULL;
    }
    task* curTask = &worktodo[--curTaskNum];
    pthread_mutex_unlock(&mutex);
    return curTask;
}
```

threadpool.c

Then we are going to build up a thread pool. Given the source code, we are actually going to implement the following functions:

- `pool_init()`: this function will **create the threads** at start-up as well as **initialize mutual-exclusion locks and semaphores**. It is implemented as follows:

```c
// initialize the thread pool
void pool_init(void)
{
    curTaskNum = 0;

    sem = sem_open("SEM", O_CREAT, 0666, 1);

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

threadpool.c

- `pool_submit()`: this 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. This function uses the `enqueue()` function to submit task to the queue. It is implemented as follows:

```c
/**
 * Submits work to the pool.
 */
int pool_submit(void (*somefunction)(void *p), void *p)
{
    task tmpTask;

    tmpTask.function = somefunction;
    tmpTask.data = p;

    sem_post(sem);
    return enqueue(tmpTask);
}
```

threadpool.c

- `worker()`: this 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. This function uses the `dequeue()` function to get task from the queue. It is implemented as follows:

```c
// the worker thread in the thread pool
void *worker(void *param)
{
    // execute the task
    while(TRUE)
    {
        task* toDoTask = dequeue();
        if (toDoTask == NULL) continue;

        execute(toDoTask->function, toDoTask->data);
    }

    pthread_exit(0);
}
```

threadpool.c

- `pool_shutdown()`: this function will cancel each worker thread and then wait for each thread to terminate. It is implemented as follows:

```c
// shutdown the thread pool
void pool_shutdown(void)
{
    int i;
    for(i = 0; i < NUMBER_OF_THREADS; i++)
    {
      pthread_cancel(bee[i]);
    }
}
```

threadpool.c

In a client file, to use a thread pool. We need to the following:

- Initialize the thread pool.
- Submit tasks to the thread pool.
- After finishing, shut down the thread pool.

In my work, I let a client submit 100 addition works to the thread pool. It works out just fine:

```c
int main(void)
{
    // create some work to do
    struct data work;
    work.a = 5;
    work.b = 10;

    // initialize the thread pool
    pool_init();

    // submit the work to the queue
    for (int i=0; i<20; i++)
    {
        int s = pool_submit(&add,&work);
        while (s)
        {
           s = pool_submit(&add,&work);
        }
    }

    // may be helpful
    sleep(3);

    pool_shutdown();

    return 0;
}
```

client.c

To compile my files, it is needed to type `make` under the directory path. To run this file, it is needed to type `./example` to run.

2 Producer-Consumer Problem

2.1 Introduction

In this project, you will design a programming solution to the bounded-buffer problem using the producer and consumer processes

The solution presented textbook 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.

In this project, I 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.

2.2 Producer Consumer Implementation

In this project, we need to implement two elements: **The Buffer** and **The Producer & Consumer Thread**.

The Buffer

First, we implement the buffer. We use a **circular queue** as a buffer. The array of buffer item objects will be manipulated in a circular queue. The buffer will be manipulated with two functions, `insert_item()` and `remove_item()`, which are called by the producer and consumer threads, respectively.

The buffer will also require an initialization function that initializes the mutual-exclusion object mutex along with the empty and full semaphores. We use `buffer_init()` to initialize the buffer and create the separate producer and consumer threads. The buffer is implemented as follows:

```c
int insert_item(buffer_item item) {
/* insert item into buffer
return 0 if successful, otherwise
```

```
return -1 indicating an error condition */
    if ((rear+1)%BUFFER_SIZE == front)
        return -1;
    buffer[rear] = item;
    rear = (rear+1)%BUFFER_SIZE;
    return 0;
}


int remove_item(buffer_item *item) {
/* remove an object from buffer
placing it in item
return 0 if successful, otherwise
return -1 indicating an error condition */
    if (front == rear)
    {
        return -1;
    }
    item = &buffer[front];
    front = (front+1)%BUFFER_SIZE;
    return 0;
}

void buffer_init()
{
    rear = front = 0;
    pthread_mutex_init(&mutex, NULL);
  sem_init(&full, 0, 0);
  sem_init(&empty, 0, BUFFER_SIZE);
}
```

ProCon.c

Producer & 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.

It is implemented as follows:

```
void *producer(void *param) {
    buffer_item item;
    while (1) {
    /* sleep for a random period of time */
    sleep(rand()%5);
    /* generate a random number */
    item = rand();

    sem_wait(&empty);
    pthread_mutex_lock(&mutex);

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

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

void *consumer(void *param) {
    buffer_item item;
    while (1) {


    /* sleep for a random period of time */
    sleep(rand()%5);

    sem_wait(&full);
    pthread_mutex_lock(&mutex);

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

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

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

ProCon.c

To test this, we use `main()` function. In main function, we let user input the running time, producer threads number, and the consumer threads number. Then in the main, the producers produce random numbers, and the consumers consume random numbers. In details, it is implemented like this:

```
int main(int argc, char const *argv[])
{
    /* 1. Get command line arguments argv[1],argv[2],argv[3] */
    int sleepTime, producerThreads, consumerThreads;
    if(argc != 4)
  {
    fprintf(stderr, "Useage: <sleep time> <producer threads> <consumer threads>\n");
    return -1;
  }
    sleepTime = atoi(argv[1]);
  producerThreads = atoi(argv[2]);
  consumerThreads = atoi(argv[3]);

    /* 2. Initialize buffer */
    buffer_init();

    /* 3. Create producer thread(s) */
    pthread_t pid[producerThreads], cid[consumerThreads];

  for(int i = 0; i < producerThreads; i++){
    pthread_create(&pid[i],NULL,&producer,NULL);
  }

    /* 4. Create consumer thread(s) */

  for(int j = 0; j < consumerThreads; j++){
    pthread_create(&cid[j],NULL,&consumer,NULL);
  }

    /* 5. Sleep */
  sleep(sleepTime);

    /* 6. Exit */
    return 0;
}
```

ProCon.c

To compile this file, we can use `gcc ProCon.c -pthread -o -output`.
To run it, we can input `./output`.