# Designing a Thread Pool

## Overview

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.

## Structure

### Queue

I simply build a queue based on an array and use. All basic operations on the queue is implemented as below.

```
struct task{
    void (*function)(void *p);
    void *data;
};

struct task queue[QUEUE_SIZE];

int head = 0;
int tail = 0;

pthread_mutex_t mutex;      // for the queue
sem_t sem;

bool isFull()
{
    return head == (tail + 1) % QUEUE_SIZE;
}

bool isEmpty()
{
    return head == tail;
}

int enqueue(struct task t)
{
    pthread_mutex_lock(&mutex);
    if(isFull())
    {
        // The queue is full
        pthread_mutex_unlock(&mutex);
        return 1;
    }
    queue[tail] = t;
    tail = (tail + 1) % QUEUE_SIZE;
    pthread_mutex_unlock(&mutex);
    return 0;
}
struct task* dequeue()
{
    pthread_mutex_lock(&mutex);
    if(isEmpty())
    {
        // The queue is empty
        pthread_mutex_unlock(&mutex);
        return NULL;
    }
    struct task *t = &queue[head];
    head = (head + 1) % QUEUE_SIZE;
    pthread_mutex_unlock(&mutex);
    return t;
}
```

## Thread Pool

Then we are going to implement a thread pool.

- `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

```
void pool_init(void)
{
    sem_init(&sem, 0, 0);
    pthread_mutex_init(&mutex, NULL);
    for(int i = 0; i < THREADS; i++)
    {
        pthread_create(&pool[i],NULL,worker,NULL);
    }
}
```

- `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.

```
int pool_submit(void (*somefunction)(void *p), void *p)
{
    struct task t;
    t.function = somefunction;
    t.data = p;

    sem_post(&sem);
    return enqueue(t);
}
```

- `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.

```
void *worker(void *param)
{
    while(1)
    {
        sem_wait(&sem);
        struct task* next = dequeue();
        if(next == NULL)    continue;
        execute(next->function, next->data);
    }
    pthread_exit(0);
}

void execute(void (*somefunction)(void *p), void *p)
{
    (*somefunction)(p);
}
```

- `pool shutdown()` : this function will cancel each worker thread and then wait for each thread to terminate.

```
void pool_shutdown(void)
{
    for(int i = 0; i < THREADS; i++)
    {
        pthread_cancel(pool[i]);
        pthread_join(pool[i],NULL);
    }
}
```

## Client

To test new thread pool, I design a `client.c` submitting 20 multiplication tasks to the thread pool.

```
/*
    A sample of the program that uses thread pool.
*/

#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 multiply 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;
    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);
        }

    }

    sleep(3);
    pool_shutdown();

    return 0;
}
```

## Result show

## How to the run the program

To compile the files, enter `make`

To run it, enter `./test`

# Producer-Consumer Problem

## Overview

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.

## Structure

### The Buffer

First I implement a queue similar to the one in the previous project(Thread Pool)

```
buffer_item buffer[BUFFER_SIZE];
int head = 0;
int tail = 0;

/*
    insert item into buffer
    return 0 if successful,
    otherwise return -1 indicating an error condition
*/
int insert_item(buffer_item item)
{
    if(head != (tail + 1) % BUFFER_SIZE){
        buffer[tail++] = item;
        tail %= BUFFER_SIZE;
        return 0;
    }
    else{
        return -1;
    }
}


/*
    remove item from buffer
    return 0 if successful,
    otherwise return -1 indicating an error condition
*/
int remove_item(buffer_item* item)
{
    if(head != tail){
        * item = buffer[head++];
        return 0;
    }
    else{
        return -1;
    }
}
```

## Producer and Consumer

To simulate the real world, here we use `sleep(rand() % MAX_SLEEP_PERIOD + 1)` to avoid producers or consumers running without a rest. Then I use standard counting semaphores for empty and full, which count the number of empty slots and full slots in the buffer and a mutex lock, which protects the actual insertion or removal of items in the buffer.

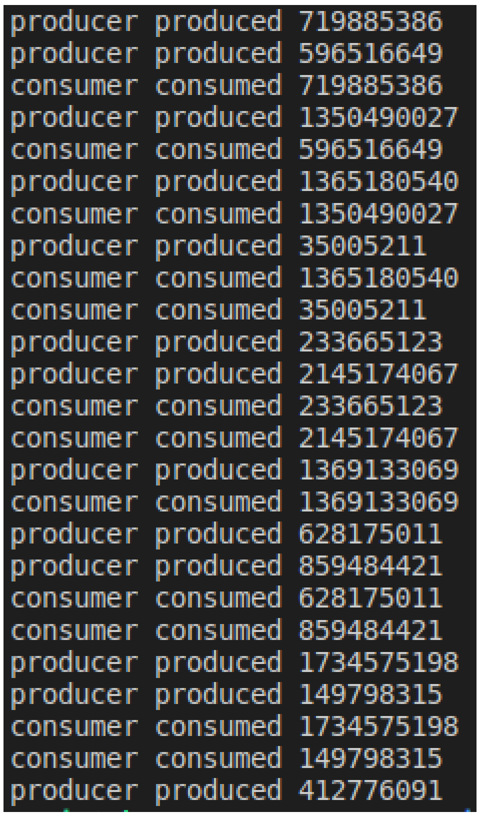
```
void *producer(void *param){
    buffer_item item;

    while(1){
        // sleep for a random period of time
        sleep(rand() % MAX_SLEEP_PERIOD + 1);

        item = rand();

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


        if(insert_item(item))
            fprintf(stderr, "error in producer");
        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() % MAX_SLEEP_PERIOD + 1);

        item = rand();

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

        if(remove_item(&item))
            fprintf(stderr, "error in consumer");
        else
            printf("consumer consumed %d\n", item);

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

    }
}
```

### Main function

To test this scenario, I implement `model.c` where the producer produce random number and the consumer consume it.

```c
int main()
{
    // initialization of mutex and semaphore
    pthread_mutex_init(&mutex, NULL);
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);


    pthread_t producers[NUM_OF_PRODUCER];
    pthread_t consumers[NUM_OF_CONSUMER];

    for(int i = 0; i < NUM_OF_PRODUCER; i++){
        pthread_create(&producers[i], NULL, producer, NULL);
    }
    for(int i = 0; i < NUM_OF_CONSUMER; i++){
        pthread_create(&consumers[i], NULL, consumer, NULL);
    }


    sleep(10);


     for(int i = 0; i < NUM_OF_PRODUCER; i++){
        pthread_cancel(producers[i]);
        pthread_join(producers[i], NULL);
    }
    for(int i = 0; i < NUM_OF_CONSUMER; i++){
        pthread_cancel(consumers[i]);
        pthread_join(consumers[i], NULL);
    }
    return 0;
}
```

## Result Show



```
producer produced 719885386
producer produced 596516649
consumer consumed 719885386
producer produced 1350490027
consumer consumed 596516649
producer produced 1365180540
consumer consumed 1350490027
producer produced 35005211
consumer consumed 1365180540
consumer consumed 35005211
producer produced 233665123
producer produced 2145174067
consumer consumed 233665123
consumer consumed 2145174067
producer produced 1369133069
consumer consumed 1369133069
producer produced 628175011
producer produced 859484421
consumer consumed 628175011
consumer consumed 859484421
producer produced 1734575198
producer produced 149798315
consumer consumed 1734575198
consumer consumed 149798315
producer produced 412776091
```

## How to run the program

To compile the files, enter `make`

To run it, enter `./test`