Project 5: Designing a Thread Pool and Producer-Consumer Problem

CS307-Operating System (D), CS356-Operating System Course Design, Chentao Wu, Spring 2020.

Name: 方泓杰(Hongjie Fang) Student ID: 518030910150 Email: galaxies@sjtu.edu.cn

1 Designing a Thread Pool

Thread pools were introduced in Section 4.5.1 in textbook. 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 Pthreads and POSIX synchronization or Java.

Solution. I use Pthread and POSIX synchronization in C programming language to complete the project. Here are my detailed solutions.

- I implement a linked queue to store the waiting work, so the length of the queue is basically unlimited and we can allow more work submitted in the same time.
- We use a mutex lock `queue_mutex` in every queue operation to prevent racing conditions in the queue.
- We use a semaphore `wait_sem` to represent the number of waiting work in the waiting queue.
- Every time we submit a work, we will first put it in the waiting queue. And then we use the semaphore `wait_sem` to awake a idle thread to execute it. If the threads are all busy, then the work will be waiting in the queue.
- For every thread (worker), we use the semaphore to let it wait for an available task. If it get an available task, it will update the queue and execute the task. The routine of every thread is basically **an infinite loop** repeating checking available work and executing it.
- When shutting down the thread pool, I use a flag `shutdown_flag` to notify every thread, and we will unlock the semaphore to let every thread exit from the infinite loop. And we use `pthread_join` to gather all threads together in the end.

Here is the specific implementation of the thread pool (`threadpool.c`).

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

#define QUEUE_SIZE 20
#define NUMBER_OF_THREADS 3

#define TRUE 1

```

```c
// 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 {
  task worktodo;
  struct queue_node *nxt;
} *head, *tail;

// the worker bee
pthread_t bee[NUMBER_OF_THREADS];

// the mutex lock
pthread_mutex_t queue_mutex;

// semaphore
sem_t wait_sem;

// shutdown flag
int shutdown_flag;

// Insert a task into the queue.
// returns 0 if successful or 1 otherwise.
int enqueue(task t) {
  tail -> nxt = (struct queue_node *) malloc (sizeof(struct queue_node));

  // fail to allocate space
  if (tail -> nxt == NULL) return 1;

  tail = tail -> nxt;
  tail -> worktodo = t;

  return 0;
}

// Remove a task from the queue.
task dequeue() {
  if (head == tail) {
    fprintf(stderr, "Error: No more work to do!\n");
    exit(1);
  }

  static struct queue_node *tmp;
  tmp = head;
  head = head -> nxt;
  free(tmp);
```

```
  return head -> worktodo;
}

// the worker thread in the thread pool.
void *worker(void *param) {
  static task tsk;
  while(TRUE) {
    // wait for available task
    sem_wait(&wait_sem);

    if (shutdown_flag) break;

    pthread_mutex_lock(&queue_mutex);
    tsk = dequeue();
    pthread_mutex_unlock(&queue_mutex);

    // execute the task
    execute(tsk.function, tsk.data);
  }
  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) {
  static task tsk;
  tsk.function = somefunction;
  tsk.data = p;

  pthread_mutex_lock(&queue_mutex);
  int res = enqueue(tsk);
  pthread_mutex_unlock(&queue_mutex);

  // success, update the semaphore.
  if (res == 0)
    sem_post(&wait_sem);
  return res;
}

// Initialize the thread pool.
void pool_init(void) {
  static int err;

  shutdown_flag = 0;

```

```
  // initialize the queue
  head = (struct queue_node *) malloc (sizeof(struct queue_node));
  if (head == NULL) {
    fprintf(stderr, "Error: queue initialization error!\n");
    exit(1);
  }
  head -> nxt = NULL;
  tail = head;

  // create the mutex lock
  err = pthread_mutex_init(&queue_mutex, NULL);
  if (err) {
    fprintf(stderr, "Error: pthread mutex initialization error!\n");
    exit(1);
  }

  // create the semaphore
  err = sem_init(&wait_sem, 0, 0);
  if (err) {
    fprintf(stderr, "Error: semaphore initialization error!\n");
    exit(1);
  }

  // create the threads
  for (int i = 0; i < NUMBER_OF_THREADS; ++ i) {
    err = pthread_create(&bee[i], NULL, worker, NULL);
    if (err) {
      fprintf(stderr, "Error: pthread create error!\n");
      exit(1);
    }
  }
  fprintf(stdout, "[Log] Create the threads successfully!\n");
}

// shutdown the thread pool
void pool_shutdown(void) {
  static int err;

  shutdown_flag = 1;

  // release the semaphore
  for (int i = 0; i < NUMBER_OF_THREADS; ++ i)
    sem_post(&wait_sem);

  // join the threads
  for (int i = 0; i < NUMBER_OF_THREADS; ++ i) {
    err = pthread_join(bee[i], NULL);
    if (err) {
      fprintf(stderr, "Error: pthread join error!\n");
      exit(1);
```

```
    }
  }
  fprintf(stdout, "[Log] Join the threads successfully!\n");

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

  // destroy the semaphore
  err = sem_destroy(&wait_sem);
  if (err) {
    fprintf(stderr, "Error: semaphore destroy error!\n");
    exit(1);
  }
}
```

I write a `client.c` program to help testing the thread pool. In the testing program, we construct 50 pieces of work, and each of them is adding two random values together. We will send them to the thread pool and wait some time in order to let all the work finish executing. You can refer to the code src/threadpool/client.c to see the details.

I also write a `Makefile` file to help testing. We only need to enter the following instructions in the terminal and we can test the program.

```
make
./example
```

Here are the executing result (Fig. 1 and Fig. 2). Here we only show the beginning logs and ending logs of the testing program, since the full version containing all executing logs of the 50 pieces of work is too long.

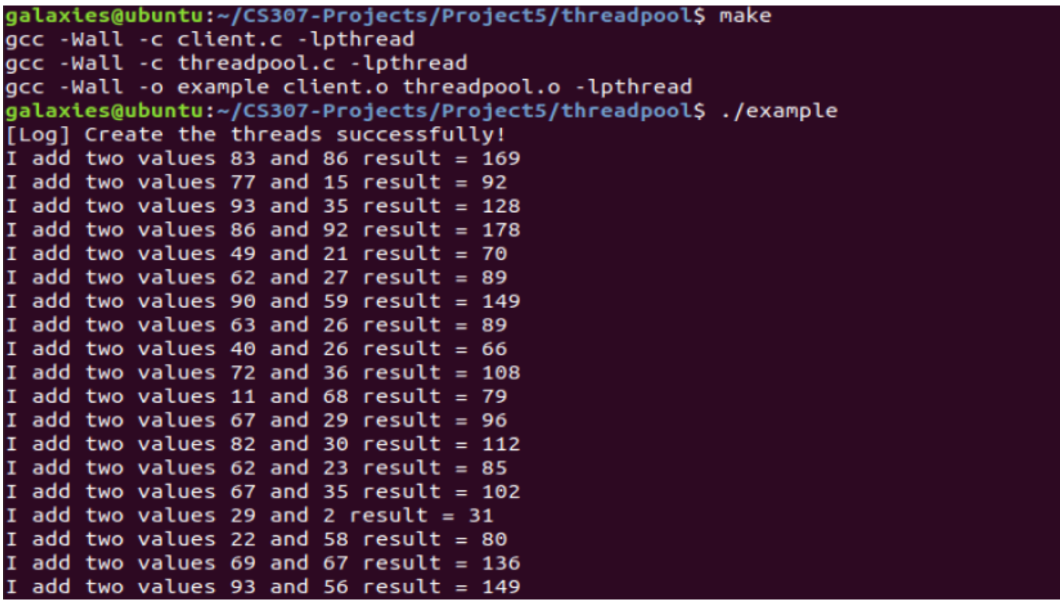

Figure 1: The beginning logs of the testing program

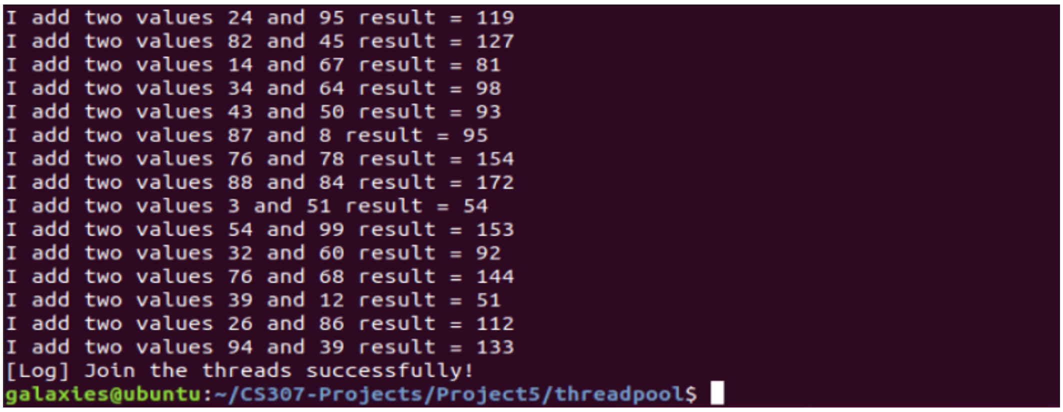

Figure 2: The ending logs of the testing program

In summary, that's the full C implementation and the testing process of the thread pool. □

2 Producer-Consumer Problem

In Section 7.1.1 in textbook, we presented a semaphore-based solution to the producer-consumer problem using a bounded buffer. In this project, you will design a programming solution to the bounded-buffer problem using the producer and consumer processes. 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 of 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 the problem using either Pthreads or the Windows API.

Solution. I use Pthread in C programming language in Linux system to complete the project. Here are my detailed solutions.

- We use a semaphore `empty` to represent the number of empty blocks in the buffer. We use a semaphore `full` to represent the number of full blocks in the buffer. And we also use a mutex lock `mutex` to prevent racing conditions among producers and consumers.
- We let the producers to produce a piece of data in random time period (randomly from 1 second to 3 seconds); we also let the consumers to consume a piece of data in random time period (randomly from 1 second to 3 seconds).
- We use a circular queue taught in the data structure class to implement the buffer module and we use the bounded-buffer solution in textbook to implement the producers and consumers.
- The program will run for T seconds with n producers and m consumers, where T, n and m are provided as arguments. We will read them from the arguments to the program and create $(n + m)$ threads representing each producer and each consumer. After T seconds, we will terminate all the threads and gather them together and exit the program.

Here is the specific implementation of the buffer module (`buffer.h` and `buffer.c`).

```
# define BUFFER_SIZE 5
typedef int buffer_item;
```

```
int insert_item(buffer_item item);
int remove_item(buffer_item *item);
void buffer_initialization();
```

```
# include "buffer.h"

// buffer implementation: circular queue
buffer_item buf[BUFFER_SIZE + 1];
int head, tail;

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

// remove an item from the buffer
int remove_item(buffer_item *item) {
  if (head == tail) return -1;
  head = (head + 1) % (BUFFER_SIZE + 1);
  *item = buf[head];
  return 0;
}

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

Here is the specific implementation of the producer-consumer program (`producer_consumer.c`).

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

# define TRUE 1

// semaphore empty, full
sem_t empty, full;
// mutex lock mutex
pthread_mutex_t mutex;
// terminate flag
int terminate_flag;

// producer thread
void *producer(void *param) {
```

```c
  buffer_item item;

  while(TRUE) {
    static int time_period;
    time_period = rand() % 3;
    sleep(time_period);

    // generate an item
    item = rand();

    sem_wait(&empty);
    pthread_mutex_lock(&mutex);
    if (terminate_flag) break;

    if (insert_item(item)) {
      fprintf(stderr, "Error: producer can not insert the item!\n");
      exit(1);
    }
    else fprintf(stdout, "[Log] Producer produced item %d.\n", item);

    pthread_mutex_unlock(&mutex);
    sem_post(&full);
  }
  pthread_mutex_unlock(&mutex);
  pthread_exit(0);
}

// consumer thread
void *consumer(void *param) {
  buffer_item item;

  while(TRUE) {
    static int time_period;
    time_period = rand() % 3;
    sleep(time_period);

    sem_wait(&full);
    pthread_mutex_lock(&mutex);
    if (terminate_flag) break;

    if (remove_item(&item)) {
      fprintf(stderr, "Error: consumer can not remove the item!\n");
      exit(1);
    }
    else fprintf(stdout, "[Log] Consumer consumed item %d.\n", item);

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

```
  pthread_exit(0);
}

int main(int argc, char *argv[]) {
  pthread_t *producer_t, *consumer_t;
  int total_time, producer_number, consumer_number;
  static int err;

  // extract the arguments
  if(argc != 4) {
    fprintf(stderr, "Error: invalid arguments.\n");
    exit(1);
  }
  total_time = atoi(argv[1]);
  producer_number = atoi(argv[2]);
  consumer_number = atoi(argv[3]);

  // initialization
  buffer_initialization();
  terminate_flag = 0;
  // |-- create the mutex lock
  err = pthread_mutex_init(&mutex, NULL);
  if (err) {
    fprintf(stderr, "Error: pthread mutex initialization error!\n");
    exit(1);
  }
  // |-- create the semaphore
  err = sem_init(&empty, 0, BUFFER_SIZE);
  if (err) {
    fprintf(stderr, "Error: semaphore initialization error!\n");
    exit(1);
  }
  err = sem_init(&full, 0, 0);
  if(err) {
    fprintf(stderr, "Error: semaphore initialization error!\n");
    exit(1);
  }

  // create threads
  producer_t = (pthread_t *) malloc (sizeof(pthread_t) * producer_number);
  consumer_t = (pthread_t *) malloc (sizeof(pthread_t) * consumer_number);

  for (int i = 0; i < producer_number; ++ i)
    pthread_create(&producer_t[i], NULL, &producer, NULL);
  for (int i = 0; i < consumer_number; ++ i)
    pthread_create(&consumer_t[i], NULL, &consumer, NULL);

  // sleep some time
  sleep(total_time);

```

```c
  // terminate
  terminate_flag = 1;
  // |-- terminate the threads
  for (int i = 0; i < producer_number; ++ i)
    sem_post(&empty);
  for (int i = 0; i < consumer_number; ++ i)
    sem_post(&full);
  // |-- join the threads
  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, "[Log] Join the threads successfully!\n");
  // |-- destroy the mutex lock
  err = pthread_mutex_destroy(&mutex);
  if (err) {
    fprintf(stderr, "Error: pthread mutex destroy error!\n");
    exit(1);
  }
  // |-- destroy the semaphores
  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 the spaces
  free(producer_t);
  free(consumer_t);

  return 0;
}
```

I write a `Makefile` file to help testing the program. We only need to enter the following instructions in the terminal and we can test the program with arguments $T = 8$, $n = 3$ and $m = 4$, that is, 3 producers and 4 consumers in 8 seconds. You can also try your own arguments yourselves.

```
make
./producer_consumer 8 3 4
```

Here is the execution result of the producer-consumer program (Fig. 3).

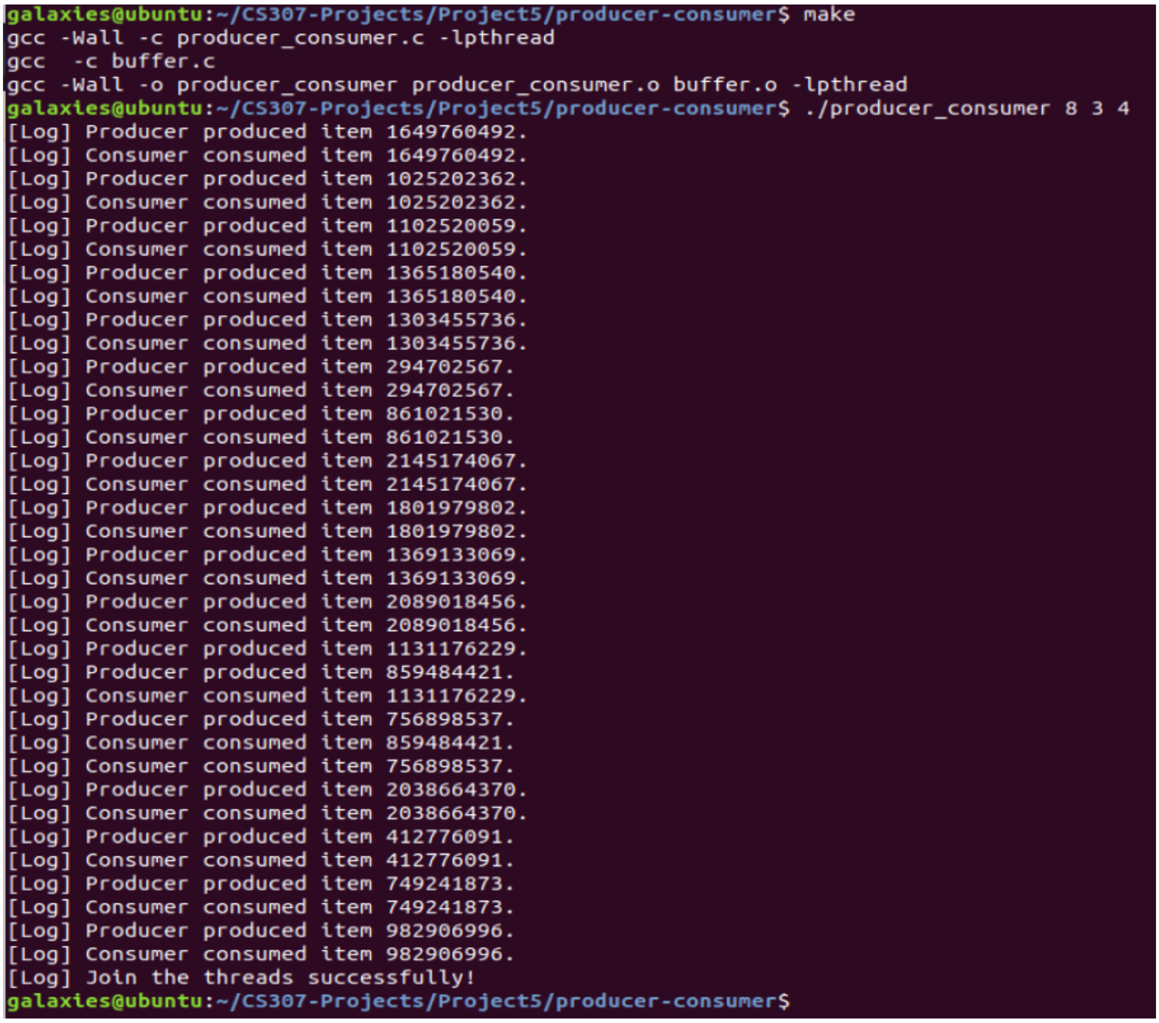

Figure 3: The execution result of the producer-consumer program

In summary, that's the full C implementation of the producer-consumer problem.

□

3 Personal Thoughts

The first project makes me understand the thread pool better, and the second one improves my understanding about some classical process synchronization methods. Both of them allow me to understand the basic tools such as Pthreads, mutex lock and semaphores in POSIX implementation better. I also implement the thread pool and the producer-consumer problem we learnt in the class myself, which benefits me a lot in programming field.

By the way, you can find all the source codes in the "src" folder. You can also refer to my github to see my codes of this project, and they are in the `Project5` folder.