Report of OS Project 4 & 5

郭嘉宋 517030910374

Assignment

Project 4 — Scheduling Algorithms

This project involves implementing several different process scheduling algorithms. The scheduler will be assigned a predefined set of tasks and will schedule the tasks based on the selected scheduling algorithm. Each task is assigned a priority and CPU burst. The following scheduling algorithms will be implemented:

- First-come, first-served (FCFS), which schedules tasks in the order in which they request the CPU.
- Shortest-job-first (SJF), which schedules tasks in order of the length of the tasks' next CPU burst.
- Priority scheduling, which schedules tasks based on priority.
- Round-robin (RR) scheduling, where each task is run for a time quantum (or for the remainder of its CPU burst).
- Priority with round-robin, which schedules tasks in order of priority and uses round-robin scheduling for tasks with equal priority.

Priorities range from 1 to 10, where a higher numeric value indicates a higher relative priority. For round-robin scheduling, the length of a time quantum is 10 milliseconds.

Project 5:

5-1: Designing a Thread Pool

Thread pools were introduced in Section 4.5.1. 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 Pthreds and POSIX synchronization or Java. Below we provide the details relevant to each specific technology. In the source code download we provide the C source file threadpool.c as a partial implementation of the thread pool. You will need to implement the functions that are called by client users, as well as several additional functions that support the internals of the thread pool. Implementation will involve the following activities:

- The pool init() function will create the threads at startup as well as initialize mutual-exclusion locks and semaphores.
- The pool submit() 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. pool submit() will add these tasks to the queue by invoking the enqueue() function, and worker threads will call dequeue() to retrieve work from the queue. The queue may be

implemented statically (using arrays) or dynamically (using a linked list). The pool init() function has an int return value that is used to indicate if the task was successfully submitted to the pool (0 indicates success, 1 indicates failure). If the queue is implemented using arrays, pool init() will return 1 if there is an attempt to submit work and the queue is full. If the queue is implemented as a linked list, pool init() should always return 0 unless a memory allocation error occurs.

- The worker() 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. A semaphore can be used for notifying a waiting thread when work is submitted to the thread pool. Either named or unnamed semaphores may be used. Refer to Section 7.3.2 for further details on using POSIX semaphores.
- A mutex lock is necessary to avoid race conditions when accessing or modifying the queue. (Section 7.3.1 provides details on Pthreads mutex locks.)
- The pool shutdown() function will cancel each worker thread and then wait for each thread to terminate by calling pthread join(). Refer to Section 4.6.3 for details on POSIX thread cancellation. (The semaphore operation sem wait() is a cancellation point that allows a thread waiting on a semaphore to be cancelled.)

5-2: The Producer–Consumer Problem

In Section 7.1.1, 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 shown in Figures 5.9 and 5.10. 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 mutualexclusion) semaphore that protects the actual insertion or 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 this problem using either Pthreads or the Windows API.

Environment

- Virtual platform: VMware Workstation Pro 15.0
- Operating system: Ubuntu 18.04
- Kernel: Linux 5.3.1
- Local system: Windows 10
- IDE: Clion 18.2.3

Solution

Project 4 — Scheduling Algorithms

```
/**
 * Driver.c
 *
 * Schedule is in the format
```

```
 *
 *  [name] [priority] [CPU burst]
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "task.h"
#include "list.h"
#include "schedulers.h"

#define SIZE    100

int main(int argc, char *argv[])
{
    FILE *in;
    char *temp;
    char task[SIZE];

    char *name;
    int priority;
    int burst;
    int tid = 0;


    in = fopen(argv[1],"r");

    while (fgets(task,SIZE,in) != NULL) {
        temp = strdup(task);
        name = strsep(&temp,",");
        priority = atoi(strsep(&temp,","));
        burst = atoi(strsep(&temp,","));


        // add the task to the scheduler's list of tasks
        add(name,priority,burst, tid);

        __sync_fetch_and_add(&tid,1);

        free(temp);
    }

    fclose(in);

    // invoke the scheduler
    schedule();

    return 0;
}
```

- The head files are showed in source code.

- 'add' function is adding name, priority, burst time and tid to the schedule list.
- Using '__sync_fetch_and_add' to update the tid, it's an atomic addition.
- 'schedule' fuction is to schedule the list by 5 laws.

FCFS

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "schedulers.h"
#include "task.h"
#include "list.h"
#include "cpu.h"


static struct node *list = NULL;

// add a task to the list
void add(char *name, int priority, int burst, int tid){
    struct task *t = malloc(sizeof(struct task));
    t->name = name;
    t->priority = priority;
    t->burst = burst;
    t->tid = tid;
    insert(&list, t);
}


void reverse(struct node *temp){
    insert(&list, temp->task);
    free(temp);
}

void execute(struct node *temp){
    run(temp->task, temp->task->burst);
    free(temp->task);
    free(temp);
}

// invoke the scheduler
void schedule() {
    struct node *first_node  = list->next;
    list->next = NULL;
    traverse(first_node, reverse);
    traverse(list, execute);
}
```

- just reverse the list, because 'add' function is the reverse direction, like a stack.

SJF

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "schedulers.h"
#include "task.h"
#include "list.h"
#include "cpu.h"


static struct node *list = NULL;
static struct task *shortest_task;

// add a task to the list
void add(char *name, int priority, int burst, int tid){
    struct task *t = malloc(sizeof(struct task));
    t->name = name;
    t->priority = priority;
    t->burst = burst;
    t->tid = tid;
    insert(&list, t);
}


void reverse(struct node *temp){
    insert(&list, temp->task);
    free(temp);
}

void find_SJ(struct node *tmp)
{
    if (shortest_task == NULL)
        shortest_task = tmp -> task;
    else
    {
        if (shortest_task->burst > tmp->task->burst )
            shortest_task = tmp -> task;
    }
}


// invoke the scheduler
void schedule() {
    struct node *first_node  = list->next;
    list->next = NULL;
    traverse(first_node, reverse);
    while( list != NULL)
    {
    shortest_task = NULL;
```

```
    traverse(list, find_SJ);
    run (shortest_task, shortest_task -> burst);
    delete (&list, shortest_task);
    free (shortest_task);
    }
}
```

- Use 'find_SJ' fuction to find the shortest job.
- First reverse the list, then find the shortest every time.
- Remember to delete the shortest job in the queue when finished.

Priority schedule

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "schedulers.h"
#include "task.h"
#include "list.h"
#include "cpu.h"


static struct node *list = NULL;
static struct task *VIP_task;

// add a task to the list
void add(char *name, int priority, int burst, int tid){
    struct task *t = malloc(sizeof(struct task));
    t->name = name;
    t->priority = priority;
    t->burst = burst;
    t->tid = tid;
    insert(&list, t);
}


void reverse(struct node *temp){
    insert(&list, temp->task);
    free(temp);
}

void find_VIP(struct node *tmp)
{
    if (VIP_task == NULL)
        VIP_task = tmp -> task;
    else
    {
        if (VIP_task->priority < tmp->task->priority )
            VIP_task = tmp -> task;
```

```
    }
}


// invoke the scheduler
void schedule() {
    struct node *first_node  = list->next;
    list->next = NULL;
    traverse(first_node, reverse);
    while( list != NULL)
    {
    VIP_task = NULL;
    traverse(list, find_VIP);
    run (VIP_task, VIP_task -> burst);
    delete (&list, VIP_task);
    free (VIP_task);
    }
}
```

- 'find_VIP' function is to find the highest priority task.
- The structure is the same as the SJF, and the only difference is one is the shortest time and the other is the highest priority.

RR

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "schedulers.h"
#include "task.h"
#include "list.h"
#include "cpu.h"

static struct node *list = NULL;
static struct node *queue = NULL;

// add a task to the list
void add(char *name, int priority, int burst, int tid){
    struct task *t = malloc(sizeof(struct task));
    t->name = name;
    t->priority = priority;
    t->burst = burst;
    t->tid = tid;
    insert(&list, t);
}


void reverse(struct node *temp){
```

```
    insert(&list, temp->task);
    free(temp);
}

void execute(struct node *tmp)
{
    if (tmp->task->burst > QUANTUM)
    {
        run (tmp->task, QUANTUM);
        tmp->task->burst -= QUANTUM;
        insert(&queue ,tmp->task);
    }
    else
    {
        run(tmp ->task, tmp ->task->burst);
        free(tmp->task);
    }
    free(tmp);
}


// invoke the scheduler
void schedule() {
    struct node *head;
    struct node *first_node  = list->next;
    list->next = NULL;
    traverse(first_node, reverse);
    while( list != NULL)
    {
        head = list; // make head move forward
        list = list ->next;
        execute(head);
        if ( list == NULL) //the original list has traverse to end
        {
            list = NULL;
            traverse(queue, reverse);
//make the queue reverse and traverse function add the
//reverse queue to original pointer list
            queue = NULL;
        }

    }
}
```

- Note that using queue to store the rest of jobs
- Remember to add the reverse queue to the original list and then do while again.

Priority_rr

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "schedulers.h"
#include "task.h"
#include "list.h"
#include "cpu.h"

static struct node *list = NULL;
static struct node *queue = NULL;
static struct task *last = NULL;

// add a task to the list
void add(char *name, int priority, int burst, int tid){
    struct task *t = malloc(sizeof(struct task));
    t->name = name;
    t->priority = priority;
    t->burst = burst;
    t->tid = tid;
    insert(&list, t);
}


void reverse(struct node *temp){
    insert(&list, temp->task);
    free(temp);
}

void find_lowest_priority(struct node *tmp)
{
    if (last == NULL)
        last = tmp -> task;
    else
    {
        if (last->priority > tmp->task->priority )
            last = tmp -> task;
    }
}

void execute(struct node *tmp)
//execute current process and add the remaining process to queue
{
    if (tmp->task->burst > QUANTUM)
    {
        run (tmp->task, QUANTUM);
        tmp->task->burst -= QUANTUM;
        insert(&queue ,tmp->task);
    }
    else
    {
        run(tmp ->task, tmp ->task->burst);
        free(tmp->task);
```

```
    }
    free(tmp);
}


// invoke the scheduler
void schedule() {
    struct node *head;
    int current_priority = 0; //the current priority
    queue = list; //now queue is the list
    list = NULL;
    while( queue != NULL)
    {
        last = NULL;
        traverse(queue, find_lowest_priority);
        delete(&queue, last);
        insert(&list, last);
        // add the element in queue back to list, and with reverse arrangement.
    }
    while(list!=NULL) //same priority are in queue, lower are in list
    {
        head = list;
        if (queue != NULL && head->task->priority < current_priority)
            //vips are still doing!
        {
            traverse(queue, reverse);
            queue = NULL;
        }
        else{
            list = list->next;
            current_priority = head->task->priority;
            execute(head);
        }
        if(list == NULL) //the last priority are moved in the stack
        {
            traverse(queue, reverse);
            queue = NULL;
        }
    }

}
```

- Note that we need to record the current priority.
- By comparing the priority and the next job to judge whether higher priority jobs have been finished.
- The rest code is the same as rr schedule.

Project 2 — Linux Kernel Module for Task Information

```c
#define BUFFER_SIZE 128
#define PROC_NAME "pid"

static long l_pid;

static ssize_t proc_read(struct file *file, char *buf, size_t count, loff_t *pos);
static ssize_t proc_write(struct file *file, const char __user *usr_buf,
 size_t count, loff_t *pos);

static struct file_operations proc_ops = {
        .owner = THIS_MODULE,
        .read = proc_read,
        .write = proc_write
};

static int proc_init(void)
{
        proc_create(PROC_NAME, 0666, NULL, &proc_ops);

        printk(KERN_INFO "/proc/%s created\n", PROC_NAME);

        return 0;
}

static void proc_exit(void)
{
        remove_proc_entry(PROC_NAME, NULL);

        printk( KERN_INFO "/proc/%s removed\n", PROC_NAME);
}

static ssize_t proc_read(struct file *file,
char __user *usr_buf, size_t count, loff_t *pos)
{
        int rv = 0;
        char buffer[BUFFER_SIZE];
        static int completed = 0;
        struct task_struct *tsk = NULL;

        if (completed) {
                completed = 0;
                return 0;
        }

        tsk = pid_task(find_vpid(l_pid), PIDTYPE_PID);

        completed = 1;
        rv = sprintf(buffer, "command = [%s] pid = [%ld] state = [%ld]\n",
        (tsk->comm), l_pid, (tsk->state));
        copy_to_user(usr_buf,buffer,rv);

        return rv;
}
```

```
static ssize_t proc_write(struct file *file, const char __user *usr_buf,
 size_t count, loff_t *pos)
{
        char *k_mem;

        k_mem = kmalloc(count, GFP_KERNEL);

        copy_from_user(k_mem,usr_buf,count);
        k_mem[count-1] = 0;
        sscanf(k_mem, "%ld", &l_pid);
        kfree(k_mem);
        return count;
}

module_init( proc_init );
module_exit( proc_exit );

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Module");
MODULE_AUTHOR("GUO");
```

- Just recording data to /proc/pid and read data from it.
- The read and write function are done in lab1, so project2 of lab2 is quite easy.

Project 5:

5-1: Designing a Thread Pool

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

#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
static task queue[QUEUE_SIZE];
pthread_mutex_t mutex; //mutex lock
static int count = 0;
static int full = 0;
static int empty = 0;
sem_t sem; //semaphone

// the worker bee
pthread_t bee[NUMBER_OF_THREADS]; // thread pool

// insert a task into the queue
// returns 0 if successful or 1 otherwise,
int enqueue(task tmp)
{
    pthread_mutex_lock(&mutex);
    if (count == QUEUE_SIZE)
    {
        pthread_mutex_unlock(&mutex);
        return 1; // can't enter the queue
    }
    else
    {
        queue[full++] = tmp;
        full %= QUEUE_SIZE; //if full == QUEUE_SIZE, then turn it to 0
        count ++;
        pthread_mutex_unlock(&mutex);
    }
    return 0;
}

// remove a task from the queue
task dequeue()
{
    pthread_mutex_lock(&mutex);
    task worktodo = queue[empty++];
    empty %= QUEUE_SIZE; //if empty == QUEUE_SIZE, then turn it to 0
    count --;
    pthread_mutex_unlock(&mutex);
    return worktodo;
}

// the worker thread in the thread pool
void *worker(void *param)
{
    task worktodo;
    while (1)
    {
        sem_wait(&sem);
        worktodo = dequeue();
        execute(worktodo.function, worktodo.data);
```

```
    }
    // execute the task
    execute(worktodo.function, worktodo.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)
{
    int flag;
    task worktodo;
    worktodo.function = somefunction;
    worktodo.data = p;
    flag = enqueue(worktodo); // judge whether it is successful to enqueue
    if (flag)
        return flag; //error exists
    sem_post(&sem);
    return flag; //good

}

// initialize the thread pool
void pool_init(void)
{
    sem_init(&sem, 0, 0);
    pthread_mutex_init(&mutex, NULL);
    for (int i=0;i< NUMBER_OF_THREADS; i++)
        pthread_create(&bee[i],NULL,worker,NULL);
}

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

- I use an array 'queue' to store the threads.

- Note to use mutex lock when doing enqueue and dequeue.
- I use 'sem' to be a semaphore to ensure only one thread can enter the critical section.

5-2: The Producer – Consumer Problem

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

// the buffer
buffer_item buffer[BUFFER_SIZE];
static int tail = 0;

pthread_mutex_t mutex;
sem_t empty;
sem_t full;

// insert item into buffer
// return 0 if successful, otherwise
// return -1 indicating an error condition
int insert_item(buffer_item item){
    int flag;
    sem_wait(&empty);
    pthread_mutex_lock(&mutex);

    // insert item
    if(tail < BUFFER_SIZE){
        buffer[tail++] = item;
        flag = 0;
    }
    else{
        flag = -1;
    }

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

// remove an object from buffer
// placing it in item
// return 0 if successful, otherwise
// return -1 indicating an error condition
int remove_item(buffer_item *item){
    int flag;
    sem_wait(&full);
    pthread_mutex_lock(&mutex);

    // remove item
    if(tail > 0){
```

```
        *item = buffer[--tail];
        flag = 0;
    }
    else{
        flag = -1;
    }

    pthread_mutex_unlock(&mutex);
    sem_post(&empty);
    return flag;
}

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

- Most code is showed in the assignment, so just complete them.
- Note to use 'empty' and 'full' to control the buffer process.
- 'mutex' is required to ensure the atomic process.

Experiment Result

Project 4:

- FCFS

```
Running task = [T1] [4] [20] for 20 units.
Running task = [T2] [3] [25] for 25 units.
Running task = [T3] [3] [25] for 25 units.
Running task = [T4] [5] [15] for 15 units.
Running task = [T5] [5] [20] for 20 units.
Running task = [T6] [1] [10] for 10 units.
Running task = [T7] [3] [30] for 30 units.
Running task = [T8] [10] [25] for 25 units.
```

- SJF

```
os@ubuntu:~/GUO/Homework/4$ ./sjf schedule.txt
Running task = [T6] [1] [10] for 10 units.
Running task = [T4] [5] [15] for 15 units.
Running task = [T1] [4] [20] for 20 units.
Running task = [T5] [5] [20] for 20 units.
Running task = [T2] [3] [25] for 25 units.
Running task = [T3] [3] [25] for 25 units.
Running task = [T8] [10] [25] for 25 units.
Running task = [T7] [3] [30] for 30 units.
```

- Priority

```
Running task = [T8] [10] [25] for 25 units.
Running task = [T4] [5] [15] for 15 units.
Running task = [T5] [5] [20] for 20 units.
Running task = [T1] [4] [20] for 20 units.
Running task = [T2] [3] [25] for 25 units.
Running task = [T3] [3] [25] for 25 units.
Running task = [T7] [3] [30] for 30 units.
Running task = [T6] [1] [10] for 10 units.
```

- rr

```
Running task = [T1] [4] [20] for 10 units.
Running task = [T2] [3] [25] for 10 units.
Running task = [T3] [3] [25] for 10 units.
Running task = [T4] [5] [15] for 10 units.
Running task = [T5] [5] [20] for 10 units.
Running task = [T6] [1] [10] for 10 units.
Running task = [T7] [3] [30] for 10 units.
Running task = [T8] [10] [25] for 10 units.
Running task = [T1] [4] [10] for 10 units.
Running task = [T2] [3] [15] for 10 units.
Running task = [T3] [3] [15] for 10 units.
Running task = [T4] [5] [5] for 5 units.
Running task = [T5] [5] [10] for 10 units.
Running task = [T7] [3] [20] for 10 units.
Running task = [T8] [10] [15] for 10 units.
Running task = [T2] [3] [5] for 5 units.
Running task = [T3] [3] [5] for 5 units.
Running task = [T7] [3] [10] for 10 units.
Running task = [T8] [10] [5] for 5 units.
```

- Priority_rr

```
Running task = [T8] [10] [25] for 10 units.
Running task = [T8] [10] [15] for 10 units.
Running task = [T8] [10] [5] for 5 units.
Running task = [T4] [5] [15] for 10 units.
Running task = [T5] [5] [20] for 10 units.
Running task = [T4] [5] [5] for 5 units.
Running task = [T5] [5] [10] for 10 units.
Running task = [T1] [4] [20] for 10 units.
Running task = [T1] [4] [10] for 10 units.
Running task = [T2] [3] [25] for 10 units.
Running task = [T3] [3] [25] for 10 units.
Running task = [T7] [3] [30] for 10 units.
Running task = [T2] [3] [15] for 10 units.
Running task = [T3] [3] [15] for 10 units.
Running task = [T7] [3] [20] for 10 units.
Running task = [T2] [3] [5] for 5 units.
Running task = [T3] [3] [5] for 5 units.
Running task = [T7] [3] [10] for 10 units.
Running task = [T6] [1] [10] for 10 units.
```

Project 5

- 5-1

```
  /**
 * Example client 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;
    sleep(1);
    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;
    work.a = 5;
    work.b = 10;

    // initialize the thread pool
    pool_init();

    // submit the work to the queue
    pool_submit(&add,&work);
    pool_submit(&add,&work);
    pool_submit(&add,&work);
    pool_submit(&add,&work);
    pool_submit(&add,&work);
    pool_submit(&add,&work);
    pool_submit(&add,&work);
    pool_submit(&add,&work);
    pool_submit(&add,&work);
    pool_submit(&add,&work);

    // may be helpful
    sleep(5);

    pool_shutdown();
```

```
    return 0;
}
```

- sleep fuction is used to present the process of threads.

```
os@ubuntu:~/GUO/Homework/5_1$ ./example
I add two values 5 and 10 result = 15
I add two values 5 and 10 result = 15
I add two values 5 and 10 result = 15
I add two values 5 and 10 result = 15
I add two values 5 and 10 result = 15
I add two values 5 and 10 result = 15
I add two values 5 and 10 result = 15
I add two values 5 and 10 result = 15
I add two values 5 and 10 result = 15
I add two values 5 and 10 result = 15
```

5-2:

```
os@ubuntu:~/GUO/Homework/5_2$ make
gcc -Wall -c main.c -lpthread
gcc -Wall -c buffer.c -lpthread
gcc -Wall -o main main.o buffer.o -lpthread
os@ubuntu:~/GUO/Homework/5_2$ ./main 12 3 5
producer produced 596516649
consumer consumed 596516649
producer produced 1102520059
consumer consumed 1102520059
producer produced 1540383426
consumer consumed 1540383426
producer produced 1726956429
consumer consumed 1726956429
consumer consumed 278722862
producer produced 278722862
producer produced 468703135
consumer consumed 468703135
producer produced 1369133069
consumer consumed 1369133069
consumer consumed 1656478042
producer produced 1653377373
consumer consumed 1653377373
producer produced 1656478042
producer produced 1734575198
```

Logs

This is my first time to learn threads, mutex lock and semaphone. I learned how to use "pthread", "sem" and so many other commands. I felt satisfied and I hope I can learn more in further study. Finally, thanks teacher and TA for reading my report and code.

517030910374 郭嘉宋