# Project5: **Designing a Thread Pool & The Producer – Consumer Problem**

Yanjie Ze 519021910706

# Project5-1: **Designing a Thread Pool**

## 1 Introduction

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.

In this project, I use **Pthread** and **POSIX**.

## 2 POSIX details

The POSIX version of this project will involve creating a number of threads using the Pthreads API as well as using POSIX mutex locks and semaphores for synchronization.

### 2.1 The Client

Users of the thread pool will utilize the following API:

- void pool init()— Initializes the thread pool.
- int pool submit(void (*somefunction)(void *p), void *p)— where somefunction is a

pointer to the function that will be executed by a thread from the pool and p is a parameter passed to the function.

- void pool shutdown(void)—Shuts down the thread pool once all tasks have completed.

Here's an example of using the API:

```
/**
 * 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;

    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);

    // may be helpful
    //sleep(3);

    pool_shutdown();

    return 0;
}
```

## 2.2 Implementation of the Thread Pool

In the source code download we provide the C source file threadpool.c as a partial implementation of the thread pool. We 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:

1. The `pool_init()` function will create the threads at startup as well as **initialize mutual-exclusion locks and semaphores**.
2. 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.
3. 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.

4. 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.)
5. 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.)

# 3 Implemenation Details and Methods

In this section, we explain how we finish the project in detail.

## 3.1 Define Global Variables

Here are several gloabl variables we may use in the following parts. We define them in **threadpool.h** .

The task queue of $size = QUEUE\_SIZE$ use the basic data structure: **array**.

```
// the work queue
task task_queue[QUEUE_SIZE];
task worktodo;
```

Also, define an array of threads:

```
// the thread queue
pthread_t thread_queue[NUMBER_OF_THREADS];
```

Define mutex lock, semaphore and the array **thread_working** to show which thread is working .

```
// the current size
int queue_current_size;

// the mutex lock for queue
pthread_mutex_t queue_mutex;

// work signal for threads;
int thread_working[NUMBER_OF_THREADS];

// sem for thread
sem_t thread_sem;
```

## 3.2 pool_init()

To initialize the thread pool, we are required to intialize **mutex lock** and **semaphore**, and we set all values of **thread_working** to 0, which is the thread state of not working currently.

```
// initialize the thread pool
void pool_init(void)
{
    queue_current_size = 0;
    sem_init(&thread_sem,0, NUMBER_OF_THREADS);
    pthread_mutex_init(&queue_mutex, NULL);
    for(int i=0 ; i<NUMBER_OF_THREADS; i++)
        thread_working[i]=0;// state of not working
}
```

## 3.3 enqueue()

Since the usage of the array is much simpler than the usage of the link list, we select the **array** as our basic data structure.

To implement `enqueue()`, we need to consider **exclusion**. We use **mutex lock**.

The **critical section** here is where we use the content of the queue and change the value of **queue_current_size**.

Each time we enter this **critical section**, we lock the mutex lock. After we exit the critical section, we unlock the mutex lock.

```
// insert a task into the queue
// returns 0 if successful or 1 otherwise,
int enqueue(task t)
{   if(queue_current_size < QUEUE_SIZE)
    {
        pthread_mutex_lock(&queue_mutex);
        task_queue[queue_current_size] = t;
        queue_current_size++;
        pthread_mutex_unlock(&queue_mutex);
        return 0;
    }
    else
    {
        printf("Enqueue Failure.\n");
        return 1;
    }
    return 0;
}
```

## 3.4 dequeue()

Similarly, we should use the mutex lock in **`dequeue()`**.

The **critical section** is the same as that in **enqueue()**.

To dequeue an element in a queue using the array, we first get the head of the queue: **task_queue[0]**. Then we need to do **translation** for the rest of the tasks in the queue.

```
task dequeue()
{

    if(queue_current_size!=0)
    {
        pthread_mutex_lock(&queue_mutex);
        worktodo = task_queue[0];
        queue_current_size--;

        for(int i=0; i<queue_current_size; i++)
            task_queue[i] = task_queue[i+1];
        task_queue[queue_current_size].function = NULL;
        task_queue[queue_current_size].data = NULL;

        pthread_mutex_unlock(&queue_mutex);
    }
    else
    {
        worktodo.data = NULL;
        worktodo.function = NULL;
        printf("Dequeue Error: Empty Queue.\n");
    }
    return worktodo;
}
```

## 3.5 worker()

The parameter we pass is the idex of the working thread.

We first dequeue the task, then execute it.

```
    worktodo = dequeue();
    execute(worktodo.function, worktodo.data);
```

After the task is finished by this working thread, we should set the value of **thread_working[thread_idx]** back to 0, call **sem_post** to add to the value of the semaphore, and finally use **pthread_exit** to release the working thread.

```
// the worker thread in the thread pool
void *worker(void *param)
{
    // execute the task
    int *idx = (int*)param;
    int thread_idx = *idx;
    worktodo = dequeue();
    execute(worktodo.function, worktodo.data);

    //finish task
    thread_working[thread_idx]=0;
    sem_post(&thread_sem);
    pthread_exit(0);
}
```

## 3.6 pool_submit()

After finishing some basic modules, we can implement `pool_submit()`.

We first enqueue the task.

```
    worktodo.function = somefunction;
    worktodo.data = p;

    int not_success = enqueue(worktodo);
```

If **dequeue()** successes, we can find a free thread to execute the task.

```
if(!not_success)
    {
        sem_wait(&thread_sem);
        while(1)
        {
            if(thread_working[work_thread_idx]==0)
            {
                thread_working[work_thread_idx]=1;
                break;
            }
            else
            {
                work_thread_idx = (work_thread_idx+1)%(NUMBER_OF_THREADS);
            }
        }
    }
```

## 3.7 pool_shutdown()

To shut down the pool, we join each thread using `pthread_join`, adn destroy the muetx lock and the semaphore.

```
// shutdown the thread pool
void pool_shutdown(void)
{
    for(int i=0; i<NUMBER_OF_THREADS; ++i)
        pthread_join(thread_queue[i], NULL);
    pthread_mutex_destroy(&queue_mutex);
    sem_destroy(&thread_sem);
}
```

# 4 Modification to Show Results

## 4.1 client.c

To make us see how the thread pool works, wo do some modification in **client.c**.

We add some tasks in this program, so that different threads will work.

```
int main(void)
{
    // create some work to do
    struct data work1;
    work1.a = 5;
    work1.b = 10;
    struct data work2;
    work2.a = 100;
    work2.b = 200;
    struct data work3;
    work3.a = 1000;
    work3.b = 2000;
    struct data work4;
    work4.a = 2000;
    work4.b = 5500;

    // initialize the thread pool
    pool_init();

    // submit the work to the queue
    pool_submit(&add,&work1);
    pool_submit(&add, &work2);
    pool_submit(&add, &work3);
    pool_submit(&add, &work4);

    // may be helpful
    //sleep(3);

    pool_shutdown();
```

```
    return 0;
}
```

## 4.2 worker()

What's more, we do some print in the function `worker`, so that we will know which thread is working.

```
// the worker thread in the thread pool
void *worker(void *param)
{
    // execute the task
    int *idx = (int*)param;
    int thread_idx = *idx;
    worktodo = dequeue();
    printf("Execute with thread %u .\n", thread_idx);
    execute(worktodo.function, worktodo.data);

    //finish task
    thread_working[thread_idx]=0;
    sem_post(&thread_sem);
    pthread_exit(0);
}
```

# 5 Program Results

The result of the program show that **the thread pool works**.

```
zyj@ubuntu:~/Documents/osc10e/ch7/project-1/posix$ make
gcc -Wall -c threadpool.c -lpthread
gcc -Wall -o example client.o threadpool.o -lpthread
zyj@ubuntu:~/Documents/osc10e/ch7/project-1/posix$ ./example
Execute with thread 2 .
Execute with thread 0 .
I add two values 1000 and 2000 result = 3000
I add two values 1000 and 2000 result = 3000
Execute with thread 0 .
I add two values 1000 and 2000 result = 3000
Execute with thread 0 .
I add two values 2000 and 5500 result = 7500
```

# Project5-2: The Producer – Consumer Problem

## 1 Introduction

In Section 7.1.1, we presented a semaphore-based solution to the producer–consumer problem using a bounded buffer. In this project, we 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 mutual exclusion) semaphore that protects the actual insertion or removal of items in the buffer. For this project, we 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.

We solve this problem by **Pthread**.

## 1.1 The Buffer

Internally, the buffer will consist of a **fixed-size array** of type buffer item (which will be defined using a typedef). The array of buffer item objects will be manipulated as a circular queue. The definition of buffer item, along with the size of the buffer, can be stored in a header file such as the following:

```
/* buffer.h */
typedef int buffer_item;
#define BUFFER_SIZE 5
```

The buffer will be manipulated with two functions, `insert_item()` and `remove_item()`, which are called by the producer and consumer threads, respectively. A skeleton outlining these functions appears in Figure 7.14.

```
#include "buffer.h"

/* the buffer */
buffer_item buffer[BUFFER_SIZE];

int insert_item(buffer_item item) {
   /* insert item into buffer
    return 0 if successful, otherwise
    return -1 indicating an error condition */
}

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 */
}
```

**Figure 7.14** Outline of buffer operations.

The insert_item() and remove_item() functions will synchronize the producer and consumer using the algorithms outlined in Figure 7.1 and Figure 7.2. The buffer will also require an initialization function that initializes the mutual-exclusion object mutex along with the empty and full semaphores.

```
while (true) {
     . . .
   /* produce an item in next_produced */
     . . .
   wait(empty);
   wait(mutex);
     . . .
   /* add next_produced to the buffer */
     . . .
   signal(mutex);
   signal(full);
}
```

**Figure 7.1** The structure of the producer process.

```
while (true) {
   wait(full);
   wait(mutex);
     . . .
   /* remove an item from buffer to next_consumed */
     . . .
   signal(mutex);
   signal(empty);
     . . .
   /* consume the item in next_consumed */
     . . .
}
```

**Figure 7.2** The structure of the consumer process.

The main() function will initialize the buffer and create the separate producer and consumer threads. Once it has created the producer and consumer threads, **the main() function will sleep for a period of time** and, upon awakening, will terminate the application. The main() function will be passed three parameters on the command line:

1. **How long to sleep before terminating**
2. **The number of producer threads**
3. **The number of consumer threads**

A skeleton for this function appears in Figure 7.15.

```
#include "buffer.h"

int main(int argc, char *argv[]) {
 /* 1. Get command line arguments argv[1],argv[2],argv[3] */
 /* 2. Initialize buffer */
 /* 3. Create producer thread(s) */
 /* 4. Create consumer thread(s) */
 /* 5. Sleep */
 /* 6. Exit */
}
```

**Figure 7.15** Outline of skeleton program.

## 1.2 The Producer and 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**.

An outline of the producer and consumer threads appears in Figure 7.16.

```
#include <stdlib.h> /* required for rand() */
#include "buffer.h"

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

   while (true) {
      /* sleep for a random period of time */
      sleep(...);
      /* generate a random number */
      item = rand();
      if (insert_item(item))
         fprintf("report error condition");
      else
         printf("producer produced %d\n",item);
}

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

   while (true) {
      /* sleep for a random period of time */
      sleep(...);
      if (remove_item(&item))
         fprintf("report error condition");
      else
         printf("consumer consumed %d\n",item);
}
```

**Figure 7.16** An outline of the producer and consumer threads.

# 2 Implementation Details and Methods

## 2.1 buffer.h

In **buffer.h**, we define the type **buffer_item**, which is basicly integer type.

And we define the size of the buffer.

```
#ifndef buffer_h
#define buffer_h

typedef int buffer_item;
#define BUFFER_SIZE 5

#endif
```

## 2.2 Implement Method: Top-Down

To implement the whole program, we implement in a top-down manner, which is to say:

**We first construct the overall code frame, then fulfill the function we need to use.**

Therefore, first we introduce the **main()** part. Second, we introduce each function.

## 2.3 main()

As required, `main()` is divided into six parts:

1. Get command line arguments argv[1],argv[2],argv[3]
2. Initialize buffer
3. Create producer thread(s)
4. Create consumer thread(s)
5. Sleep
6. Exit

We follow this order.

**1. Get command line arguments argv[1],argv[2],argv[3].**

We use `atoi()` to convert the char into the integer.

```
/* 1. Get command line arguments argv[1],argv[2],argv[3] */
    int sleep_time = atoi(argv[1]);
    int num_producer = atoi(argv[2]);
    int num_consumer = atoi(argv[3]);
```

**2. Initialize buffer** .

We initialize the mutex, the semaphore **full&empty**, the array **free_slot** to record whether the buffer slot is free.

```
/* 2. Initialize buffer */
    pthread_mutex_init(&buffer_mutex, NULL);
    sem_init(&full, 0, 0);
    sem_init(&empty, 0, BUFFER_SIZE);
    for(int i=0; i<BUFFER_SIZE; i++)
        free_slot[i]=1; // set all buffers to free
```

**3. Create producer thread(s).**

```
/* 3. Create producer thread(s) */
    pthread_t producer_thread[num_producer];
    for(int i=0; i<num_producer; i++)
    {
        pthread_create(&producer_thread[i], NULL, producer, NULL);
    }
```

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

```
/* 4. Create consumer thread(s) */
    pthread_t consumer_thread[num_consumer];
    for(int i=0; i<num_consumer; i++)
    {
        pthread_create(&consumer_thread[i], NULL, consumer, NULL);
    }
```

**5. Sleep.**

The main program falls into sleeping, and the producers and the consumers will continue executing in this period.

```
/* 5. Sleep */
    printf("*Main* begin to sleep.\n");
    sleep(sleep_time);
    printf("*Main* wake and begin to terminate.\n");
```

**6. Exit**.

All threads are needed to be cancelld. And the mutex and the semaphores are to be destroyed.

```
/* 6. Exit */
    for(int i=0; i<num_producer; i++)
    {
        pthread_cancel(producer_thread[i]);
    }
    for(int i=0; i<num_consumer; i++)
    {
        pthread_cancel(consumer_thread[i]);
    }
    sem_destroy(&full);
    sem_destroy(&empty);
    pthread_mutex_destroy(&buffer_mutex);
    printf("Cancel all. Finish.\n");
```

## 2.3 insert_item() and producer()

`insert_item()` inserts an item into the buffer.

The function loops over the buffer to see whether the slot is free, by **free_slot**.

If there is a free slot, we insert and set **free_slot[i]=0**.

```
/* insert item into buffer
return 0 if successful, otherwise
return -1 indicating an error condition */
int insert_item(buffer_item item) {

    int insert_success=0;
    for(int i=0; i<BUFFER_SIZE; ++i)
    {
        if(free_slot[i]==1)
        {
            insert_success = 1;
            buffer[i] = item;
            free_slot[i] = 0;
            break;
        }
    }

    if(insert_success==1)
        return 0;
    else
        return 1;
}
```

The producer will call **insert_item()**.

But before that, there's need to use **mutex and semaphore**, to avoid the deadlock and the race condition.

---

```
void *producer(void *param) {
    buffer_item item;
    while (1) {
    /* sleep for a random period of time */
    int sleep_a_while = rand()%3;
    sleep(sleep_a_while);

    /* generate a random number */
    sem_wait(&empty);
    pthread_mutex_lock(&buffer_mutex);
    item = rand()%100;
    if (insert_item(item))
        printf("Producer Insert Failure.\n");
    else
        printf("Producer produced %d.\n",item);
    pthread_mutex_unlock(&buffer_mutex);
    sem_post(&full);
    }
}
```

## 2.4 remove_item() and consumer()

`remove_item()` removes the item from the buffer.

The function loops over the buffer to see whether there exists a free slot.

If there is, set **free_slot[i]=1**,which just means the slot is free.

```
/* 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 remove_success=0;
    for(int i=0; i<BUFFER_SIZE; ++i)
```

```
    {
        if(free_slot[i]==0)
        {
            remove_success = 1;
            (*item) = buffer[i];
            free_slot[i] = 1;
            break;
        }
    }

    if(remove_success)
        return 0;
    else
        return 1;
}
```

The consumer will call `remove_item()`, then display the item removed by it.

Note that we need to use **mutex and semaphores** to avoid the deadlock and the race condition.

```
void *consumer(void *param)
{
    buffer_item item;
    while (1) {
    /* sleep for a random period of time */
    int sleep_a_while = rand()%3;
    sleep(sleep_a_while);
    sem_wait(&full);
    pthread_mutex_lock(&buffer_mutex);
    if (remove_item(&item))
    printf("Consumer Remove Failure.\n");
    else
    printf("Consumer consumed %d.\n",item);
    pthread_mutex_unlock(&buffer_mutex);
    sem_post(&empty);
    }
```

```
}
```

## 3 Program Results

Our Makefile is:

```
producer_consumer:
  gcc buffer.h main.c -o producer_consumer -l pthread
clean:
  rm producer_consumer
```

To run the program, enter the command as below:

```
./producer_consumer sleep_time num_producer num_consumer
```

Entering:

```
./producer_consumer 5 3 4
```

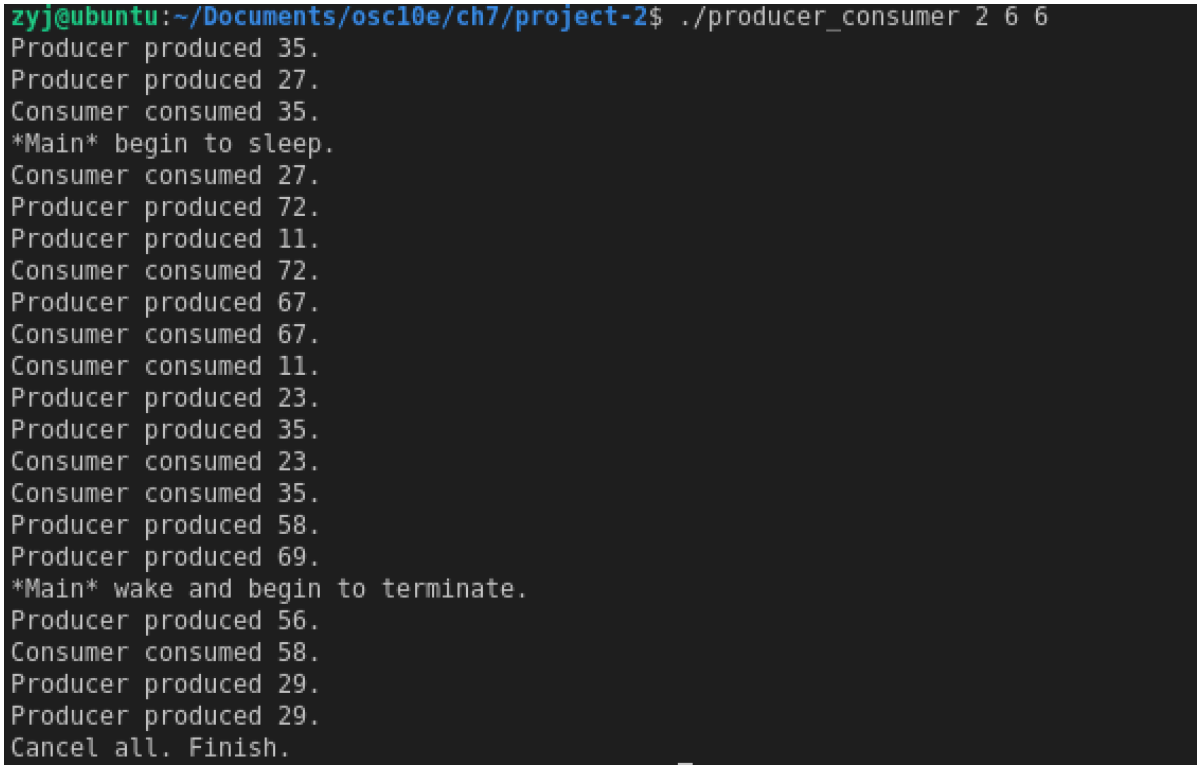
We get:

```
zyj@ubuntu:~/Documents/osc10e/ch7/project-2$ ./producer_consumer 5 3 4
*Main* begin to sleep.
Producer produced 35.
Consumer consumed 35.
Producer produced 62.
Producer produced 27.
Consumer consumed 62.
Consumer consumed 27.
Producer produced 40.
Consumer consumed 40.
Producer produced 36.
Producer produced 68.
Consumer consumed 36.
Producer produced 82.
Consumer consumed 82.
Consumer consumed 68.
Producer produced 67.
Consumer consumed 67.
Producer produced 2.
Consumer consumed 2.
Producer produced 69.
Consumer consumed 69.
Producer produced 56.
Consumer consumed 56.
Producer produced 29.
Producer produced 21.
Consumer consumed 29.
Producer produced 37.
Consumer consumed 37.
Producer produced 15.
Consumer consumed 15.
Producer produced 26.
*Main* wake and begin to terminate.
Consumer consumed 26.
Cancel all. Finish.
```

Entering:

```
./producer_consumer 2 6 6
```

We get:




# Conclusion and Thoughts

In project 5, we dive deep into the usage of the mutex and the semaphore, and get the chance to practise using the two tools to solve the problems.

Concretely, in project5-1, we design a thread pool in **C**, which can function the same as that in Java. We mainly use the mutex to avoid the race condition.

In project5-2, we implement the solution of the producer-consumer problem, which is taught in our OS class. It's a valuable opportunity to solve this problem by hand, since we know the solution theoretically but not do any programming yet.

In conclusion, project 5 further improves our knowledge of the deadlock and how to solve it, and makes us able to construct the program to solve the problem indeed.