电 子 科 技 大 学

实　验　报　告

学生姓名：Lolipop　　学号：2018091202000　指导教师：xx

实验地点：校外　　实验时间：2020/06/09

一、实验名称：进程（线程）同步与互斥的经典问题

二、实验学时：4 学时

三、实验目的：

实现哲学家就餐问题，要求不能出现死锁。通过本实验熟悉 Linux 系统的基本环境，了解 Linux 下进程和线程的实现。

实现生产者消费者问题。

1) 掌握进程、线程的概念，熟悉相关的控制语。
2) 掌握进程、线程间的同步原理和方法。
3) 掌握进程、线程间的互斥原理和方法。
4) 掌握使用信号量原语解决进程、线程间互斥和同步方法。

四、实验原理：

哲学家进餐问题：

1) 关系分析。5 名哲学家与左右邻居对其中间筷子的访问是互斥关系。
2) 为了实现哲学家对筷子的获取关系，显然这里有五个进程(线程)。实验的关键是如何让一个哲学家拿到左右两个筷子而不造成死锁或者饥饿现象。
3) 用信号量或者互斥量实现对 5 个筷子的互斥访问。对哲学家按顺序从 0～4 编号，哲学家 i 左边的筷子的编号为 i，哲学家右边的筷子的编号为(i+1)%5。

```
semaphore chopstick[5] = {1,1,1,1,1}; //定义信号量数组
pthread_mutex_t chopstick[5] //或者定义互斥量数组
```

生产者消费者问题：

n 个缓冲区的缓冲池作为一个共享资源，当生产者进程从数据源—文件中读取数据后将会申请一个缓冲区，并将此数据放入缓冲区中。消费者从一个缓冲区中取走数据，并将其中的内容打印输出。当生产者进程正在访问缓冲区时，消费者进程不能同时访问缓冲区，因此缓冲区是个互斥资源。

五、实验内容：

哲学家进餐问题：在 linux 系统下实现教材 2.5.2 节中所描述的哲学家就餐问题。要求显示出每个哲学家的工作状态，如吃饭，思考。连续运行 30 次以上都未出现死锁现象。

生产者消费者问题：

1) 创建 3 个进程（或线程）作为生产者，4 个进程（或线程）作为消费者。创建一个文件作为数据源，文件中事先写入一些内容作为数据。
2) 生产者和消费者进程（或者线程）都具有相同的优先级。
3) 采用信号量方式解决。

六、实验器材（设备、元器件）：

硬件设备：Windows 10 电脑一台；

软件设备：Xshell 以连接 CentOS 服务器。

七、实验步骤：

哲学家进餐问题：

1) 初始化信号量数组和哲学家编号。
2) 设计实现 delay 延时函数，对哲学家们思考、进餐等时间进行设计。
3) 设计实现 eat_think 函数，规定奇数号的哲学家先拿起他左边的筷子,然后再去拿他右边的筷子;而偶数号的哲学家则相反，实现哲学家进餐问题。
4) 初始化进程，创建五个子进程，等待任务执行。
5) 输出执行结果。

生产者消费者问题：

1) 分配具有 n 个缓冲区的缓冲池，作为共享资源。
2) 定义两个资源型信号量 empty 和 full，empty 信号量表示当前空的缓冲区数量，full 表示当前满的缓冲区数量。

3) 定义互斥信号量 mutex，当某个进程访问缓冲区之前先获取此信号量，在对缓冲区的操作完成后再释放此互斥信号量。以此实现多个进程对共享资源的互斥访问。
4) 创建 3 个进程（或者线程）作为生产者，4 个进程（或者线程）作为消费者。创建一个文件作为数据源，文件中事先写入一些内容作为内容（生产者和消费者流程图）。
5) 编写代码实现生产者进程的工作内容，即从文件中读取数据，然后申请一个 empty 信号量，和互斥信号量，然后进入临界区操作将读取的数据放入此缓冲区中。并释放 empty 信号量和互斥信号量。
6) 编写代码实现消费者者进程的工作内容，即先申请一个 full 信号量，和互斥信号量，然后进入临界区操作从缓冲区中读取数据并打印输出。

八、实验结果与分析（含重要数据结果分析或核心代码流程分析）

哲学家进餐问题：

1) 初始化信号量数组和哲学家编号。

代码 8-1 全局变量设置

```
sem_t chopsticks[6];     // 信号量数组，一共有六双筷子
int philosophers[5] = {0, 1, 2, 3, 4};   // 哲学家编号
```

2) 设计实现 delay 延时函数，对哲学家们思考、进餐等时间进行设计。

代码 8-2 delay 延时函数

```
void delay(int maxLength)           // 延时函数，实现根据系统性能的随机延时
{
   int i = rand() % maxLength;
   while (i > 0)
   {
      int x = rand() % maxLength;
      while (x > 0)
      {
```

```
            x--;
        }
        i--;
    }
}
```

3) 设计实现 eat_think 函数，规定奇数号的哲学家先拿起他左边的筷子,然后再去拿他右边的筷子;而偶数号的哲学家则相反，实现哲学家进餐问题。

代码 8-3 eat_think 函数

```
void *eat_think(void *arg)
{
    int i = *(int *)arg;
    int left = i;
    int right = (i + 1) % 6;

    while (1) {
        printf("哲学家 %d 正在思考\n", i);
        delay(50000);
        printf("哲学家 %d 饿了\n", i);

        if(i % 2 == 0) {        // 哲学家为偶数号，先拿起右边的筷子
            sem_wait(&chopsticks[right]);
            printf("哲学家 %d 拿起了 %d 号筷子\n", i, right);
            sem_wait(&chopsticks[left]);
            printf("哲学家 %d 拿起了 %d 号筷子,开始进餐\n", i, left);
            delay(50000);
            sem_post(&chopsticks[left]);
            printf("哲学家 %d 放下了 %d 号筷子\n", i, left);
            sem_post(&chopsticks[right]);
```

```
            printf("哲学家 %d 放下了 %d 号筷子\n", i, right);
        } else {        // 哲学家为奇数号，先拿起左边的筷子
            sem_wait(&chopsticks[left]);
            printf("哲学家 %d 拿起了 %d 号筷子\n", i, left);
            sem_wait(&chopsticks[right]);
            printf("哲学家 %d 拿起了 %d 号筷子,开始进餐\n", i, right);
            delay(50000);
            sem_post(&chopsticks[right]);
            printf("哲学家 %d 放下了 %d 号筷子\n", i, right);
            sem_post(&chopsticks[left]);
            printf("哲学家 %d 放下了 %d 号筷子\n", i, left);
        }
    }
}
```

4) 初始化进程，创建五个子进程，等待任务执行。

代码 8-4 哲学家进餐问题主函数

```
int main(int argc, char **argv)
{
    srand(time(NULL));        // 生成随机种子
    pthread_t PHD[5];         // 创建五个哲学家进程

    int i;

    for (i = 0; i < 6; i++) {
        sem_init(&chopsticks[i], 0, 1);
    }

    for (i = 0; i < 5; i++)
```

```
        pthread_create(&PHD[i], NULL, (void*)eat_think, &philosophers[i]);


    for (i = 0; i < 5; i++)
        pthread_join(PHD[i], NULL);


    return 0;
}
```

5) 输出执行结果。

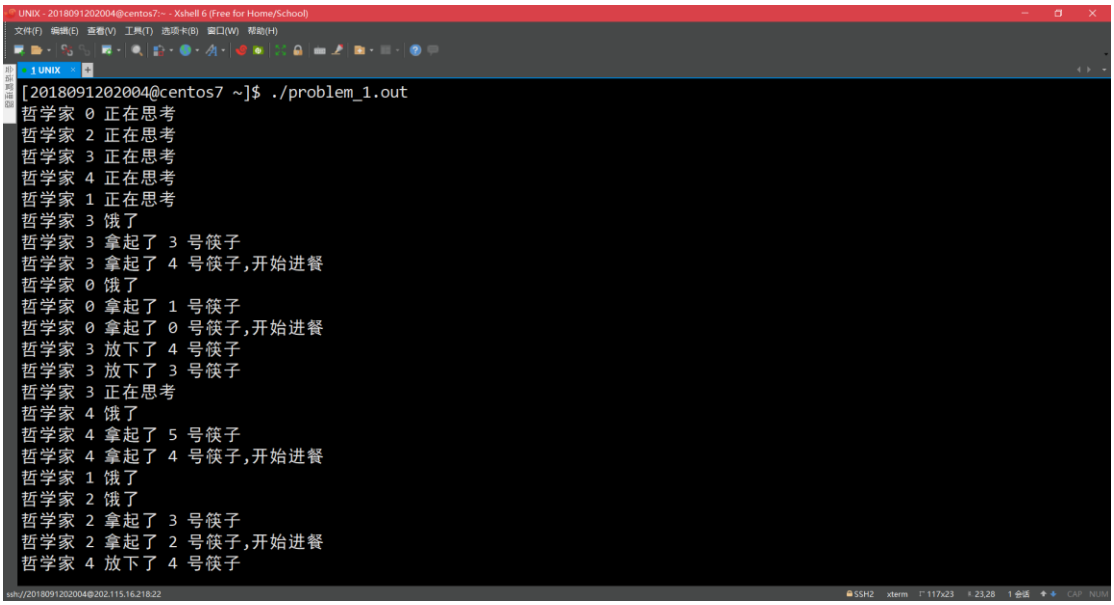

图 8-1 哲学家进餐问题结果

程序执行结果节选如下所示：

```
哲学家 0 正在思考
哲学家 2 正在思考
哲学家 3 正在思考
哲学家 4 正在思考
哲学家 1 正在思考
哲学家 3 饿了
哲学家 3 拿起了 3 号筷子
```

哲学家 3 拿起了 4 号筷子,开始进餐

哲学家 0 饿了

哲学家 0 拿起了 1 号筷子

哲学家 0 拿起了 0 号筷子,开始进餐

哲学家 3 放下了 4 号筷子

哲学家 3 放下了 3 号筷子

哲学家 3 正在思考

哲学家 4 饿了

哲学家 4 拿起了 5 号筷子

哲学家 4 拿起了 4 号筷子,开始进餐

哲学家 1 饿了

哲学家 2 饿了

哲学家 2 拿起了 3 号筷子

哲学家 2 拿起了 2 号筷子,开始进餐

哲学家 4 放下了 4 号筷子

哲学家 4 放下了 5 号筷子

哲学家 4 正在思考

哲学家 2 放下了 2 号筷子

哲学家 2 放下了 3 号筷子

哲学家 2 正在思考

哲学家 0 放下了 0 号筷子

哲学家 0 放下了 1 号筷子

哲学家 0 正在思考

哲学家 1 拿起了 1 号筷子

哲学家 1 拿起了 2 号筷子,开始进餐

哲学家 3 饿了

哲学家 3 拿起了 3 号筷子

哲学家 3 拿起了 4 号筷子,开始进餐

哲学家 0 饿了

哲学家 4 饿了

```
哲学家 4 拿起了 5 号筷子
哲学家 2 饿了
哲学家 1 放下了 2 号筷子
哲学家 1 放下了 1 号筷子
哲学家 1 正在思考
哲学家 0 拿起了 1 号筷子
哲学家 0 拿起了 0 号筷子,开始进餐
哲学家 3 放下了 4 号筷子
哲学家 3 放下了 3 号筷子
哲学家 3 正在思考
哲学家 4 拿起了 4 号筷子,开始进餐
哲学家 2 拿起了 3 号筷子
哲学家 2 拿起了 2 号筷子,开始进餐
哲学家 2 放下了 2 号筷子
哲学家 2 放下了 3 号筷子
哲学家 2 正在思考
哲学家 0 放下了 0 号筷子
哲学家 0 放下了 1 号筷子
哲学家 0 正在思考
哲学家 0 饿了
哲学家 0 拿起了 1 号筷子
哲学家 0 拿起了 0 号筷子,开始进餐
哲学家 2 饿了
哲学家 2 拿起了 3 号筷子
哲学家 2 拿起了 2 号筷子,开始进餐
哲学家 1 饿了……
```

哲学家进餐问题的函数如代码 8-5 所示。

代码 8-5 哲学家进餐问题

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <time.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

sem_t chopsticks[6];     // 信号量数组，一共有六双筷子
int philosophers[5] = {0, 1, 2, 3, 4};   // 哲学家编号

void delay(int maxLength)           // 延时函数，实现根据系统性能的随机延时
{
   int i = rand() % maxLength;
   while (i > 0)
   {
      int x = rand() % maxLength;
      while (x > 0)
      {
         x--;
      }
      i--;
   }
}

void *eat_think(void *arg)
{
   int i = *(int *)arg;
   int left = i;
   int right = (i + 1) % 6;
```

```
    while (1) {
        printf("哲学家 %d 正在思考\n", i);
        delay(50000);
        printf("哲学家 %d 饿了\n", i);

        if(i % 2 == 0) {        // 哲学家为偶数号，先拿起右边的筷子
            sem_wait(&chopsticks[right]);
            printf("哲学家 %d 拿起了 %d 号筷子\n", i, right);
            sem_wait(&chopsticks[left]);
            printf("哲学家 %d 拿起了 %d 号筷子,开始进餐\n", i, left);
            delay(50000);
            sem_post(&chopsticks[left]);
            printf("哲学家 %d 放下了 %d 号筷子\n", i, left);
            sem_post(&chopsticks[right]);
            printf("哲学家 %d 放下了 %d 号筷子\n", i, right);
        } else {        // 哲学家为奇数号，先拿起左边的筷子
            sem_wait(&chopsticks[left]);
            printf("哲学家 %d 拿起了 %d 号筷子\n", i, left);
            sem_wait(&chopsticks[right]);
            printf("哲学家 %d 拿起了 %d 号筷子,开始进餐\n", i, right);
            delay(50000);
            sem_post(&chopsticks[right]);
            printf("哲学家 %d 放下了 %d 号筷子\n", i, right);
            sem_post(&chopsticks[left]);
            printf("哲学家 %d 放下了 %d 号筷子\n", i, left);
        }
    }
}
```

```
int main(int argc, char **argv)
{
    srand(time(NULL));       // 生成随机种子
    pthread_t PHD[5];        // 创建五个哲学家进程

    int i;

    for (i = 0; i < 6; i++) {
        sem_init(&chopsticks[i], 0, 1);
    }

    for (i = 0; i < 5; i++)
        pthread_create(&PHD[i], NULL, (void*)eat_think, &philosophers[i]);

    for (i = 0; i < 5; i++)
        pthread_join(PHD[i], NULL);

    return 0;
}
```

生产者消费者问题：

1) 分配具有 n 个缓冲区的缓冲池，作为共享资源。

代码 8-6 分配缓冲池

```
#define N 10   // 缓存区大小
int buff[N] = {0};
```

2) 定义两个资源型信号量 empty 和 full，empty 信号量表示当前空的缓冲区数量，full 表示当前满的缓冲区数量。

代码 8-7 定义同步信号量

```
sem_t empty_sem; // 同步信号量，记录缓存区中未使用的数
sem_t full_sem;     // 同步信号量，记录缓存区中已使用的数
```

3) 定义互斥信号量 mutex，当某个进程访问缓冲区之前先获取此信号量，在对缓冲区的操作完成后再释放此互斥信号量。以此实现多个进程对共享资源的互斥访问。

代码 8-8 定义互斥信号量

```
pthread_mutex_t mutex;// 互斥信号量，实现互斥访问
```

4) 创建 3 个进程（或者线程）作为生产者，4 个进程（或者线程）作为消费者。创建一个文件作为数据源，文件中事先写入一些内容作为内容（生产者和消费者流程图）。

代码 8-9 创建生产者和消费者进程

```
#define N1 3          // 生产者
#define N2 4   // 消费者
pthread_t id1[N1];
pthread_t id2[N2];
int i;
for(i = 0; i < N1; i++)
{
   pthread_create(&id1[i], NULL, product, (void*)(&i));
}

for(i = 0; i< N2; i++)
{
   pthread_create(&id2[i], NULL, consume, NULL);
}

for(i = 0; i < N1; i++) {
```

```
    pthread_join(id1[i], NULL);
}

for(i = 0; i < N2; i++) {
    pthread_join(id2[i], NULL);
}
```

5) 编写代码实现生产者进程的工作内容，即从文件中读取数据，然后申请一个 empty 信号量，和互斥信号量，然后进入临界区操作将读取的数据放入此缓冲区中。并释放 empty 信号量和互斥信号量。

代码 8-10 生产者进程函数

```
void * product()
{
    int id = ++product_id;
    while(1)
    {
        sleep(1);
        sem_wait(&empty_sem); // 等待缓冲区有空间
        pthread_mutex_lock(&mutex); // 锁定互斥信号量
        if(fscanf(fp, "%d", &data)==EOF)
        {
            fseek(fp, 0, SEEK_SET);
            fscanf(fp, "%d", &data);
        }
        in = in % N;
        buff[in] = data;
        printf("生产者 %d 生产 %d 于 %d\n", id, buff[in], in);
        ++in;
        pthread_mutex_unlock(&mutex);    // 解锁互斥信号量
```

```
        sem_post(&full_sem);     // 向同步信号量发送信号
    }
}
```

6) 编写代码实现消费者者进程的工作内容，即先申请一个 full 信号量，和互斥信号量，然后进入临界区操作从缓冲区中读取数据并打印输出。

代码 8-11 消费者进程函数

```
void *consume()
{
    int id = ++consumer_id;
    while(1)
    {
        sleep(1);
        sem_wait(&full_sem);   // 等待缓冲区存有数据
        pthread_mutex_lock(&mutex);     // 锁定互斥信号量
        out = out % N;
        printf("消费者 %d 消费 %d 于 %d\n", id, buff[out], out);
        buff[out] = 0;
        ++out;
        pthread_mutex_unlock(&mutex);  // 解锁互斥信号量
        sem_post(&empty_sem);       // 向同步信号量发送信号
    }
}
```

7) 代码运行结果如图 8-2 和 8-3 所示。

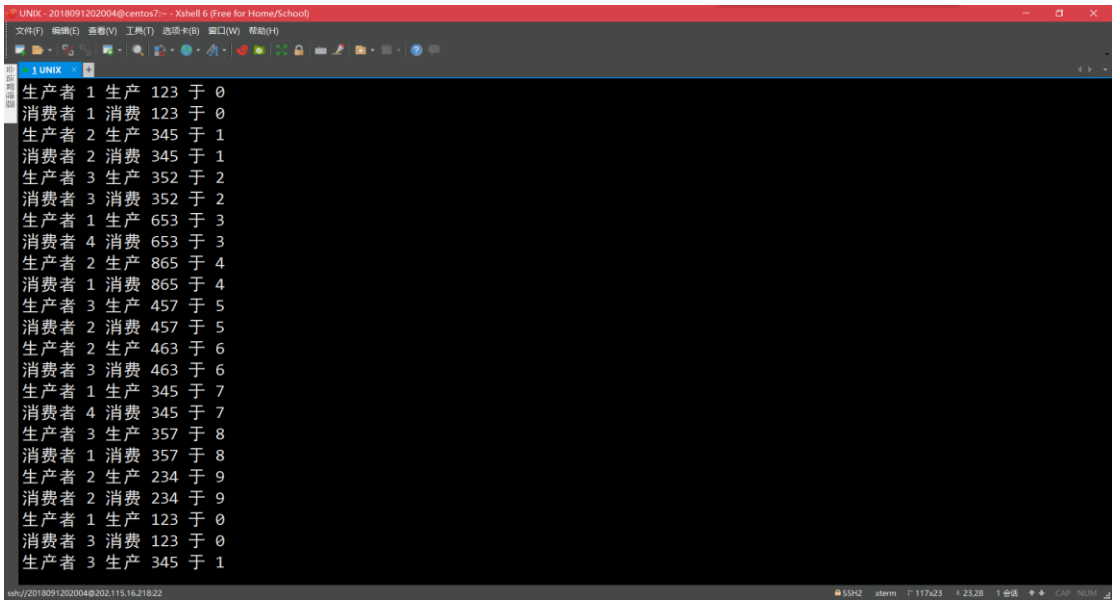

图 8-2 生产者消费者问题结果（上）

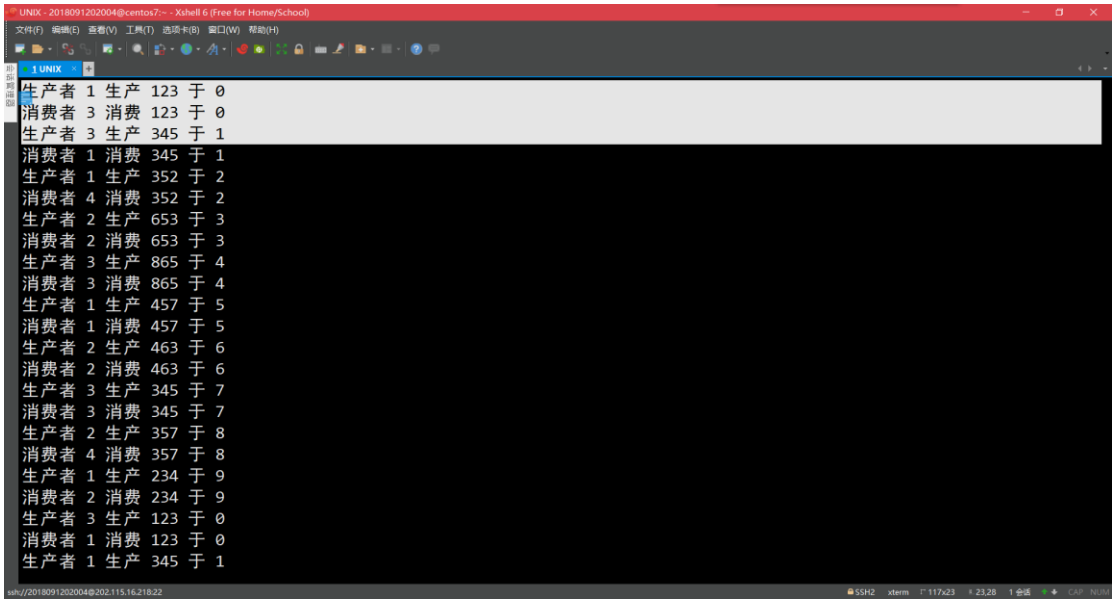

图 8-3 生产者消费者问题结果（下）

8) 生产者消费者问题的代码如代码 8-12 所示。

代码 8-12 生产者消费者问题代码

```
#include <stdio.h>

#include <semaphore.h>

#include <stdlib.h>

#include <pthread.h>
```

```
#include <unistd.h>

#define N1 3   // 生产者
#define N2 4   // 消费者
#define N 10   // 缓存区大小

sem_t empty_sem; // 同步信号量，记录缓存区中未使用的数
sem_t full_sem;     // 同步信号量，记录缓存区中已使用的数
pthread_mutex_t mutex;// 互斥信号量，实现互斥访问

int in = 0;
int out = 0;
int buff[N] = {0};

int product_id = 0;
int consumer_id = 0;

int data;  // 假设数据均为 int 型数字
FILE *fp;

void * product()
{
   int id = ++product_id;
   while(1)
   {
      sleep(1);
      sem_wait(&empty_sem);       // 等待缓冲区有空间
      pthread_mutex_lock(&mutex);     // 锁定互斥信号量
      if(fscanf(fp, "%d", &data)==EOF)
      {
```

```
            fseek(fp, 0, SEEK_SET);
            fscanf(fp, "%d", &data);
        }
        in = in % N;
        buff[in] = data;
        printf("生产者 %d 生产 %d 于 %d\n", id, buff[in], in);
        ++in;
        pthread_mutex_unlock(&mutex);  // 解锁互斥信号量
        sem_post(&full_sem);   // 向同步信号量发送信号
    }
}

void *consume()
{
    int id = ++consumer_id;
    while(1)
    {
        sleep(1);
        sem_wait(&full_sem);   // 等待缓冲区存有数据
        pthread_mutex_lock(&mutex);     // 锁定互斥信号量
        out = out % N;
        printf("消费者 %d 消费 %d 于 %d\n", id, buff[out], out);
        buff[out] = 0;
        ++out;
        pthread_mutex_unlock(&mutex);  // 解锁互斥信号量
        sem_post(&empty_sem);       // 向同步信号量发送信号
    }
}

int main(void)
```

```
{
    pthread_t id1[N1];
    pthread_t id2[N2];

    // 初始化同步量和互斥量
    sem_init(&empty_sem, 0, N);
    sem_init(&full_sem, 0, 0);
    pthread_mutex_init(&mutex, NULL);

    fp = fopen("./problem_2_data", "r"); // 打开存储数据的文件

    int i;
    for(i = 0; i < N1; i++)
    {
        pthread_create(&id1[i], NULL, product, (void*)(&i));
    }

    for(i = 0; i< N2; i++)
    {
        pthread_create(&id2[i], NULL, consume, NULL);
    }

    for(i = 0; i < N1; i++) {
        pthread_join(id1[i], NULL);
    }

    for(i = 0; i < N2; i++) {
        pthread_join(id2[i], NULL);
    }
```

```
    return 0;
}
```

报告评分：

指导教师签字：