进程相关实验

1.进程/线程同步互斥

分析设置

题目1. 生产流水线

An assembly line is to producea product C with n1=4 part As, and n2=3 part E
machining A and worker ofmachining B produce m1=2 part As and m2=1 par
each time. Then the m1 part Asor m2 part B will be moved to a station, which
N=12 of part As and part Bsaltogether. The produced m1 part As must be put
simultaneously. The workersmust exclusively put a part on the station or get it
In addition, the worker tomake C must get all part of As and Bs, i.e. n1 part As
one product C once.

Using semaphores to coordinate the threeworkers who are machining part A,
manufacturing the product Cwithout deadlock.

It is required that

(1) definition and initialvalue of each semaphore, and

(2) the algorithm tocoordinate the production process for the three workers
should be given.

本题目是生产者-消费者问题的扩展，需要注意题目要求：(a) worker A 一次生
个 A，必须一次性放入工作台，(b)worker C 必须一次性获得所需的 n1=4 个 A
因此，如果当前工作台空位小于 m1，worker A 被阻塞；如果当前工作台没有
worker C 被阻塞。

采用类似于多次 wait(empty)的操作，为 worker A 从工作台获取多个空位是不

（2）当所需工作台空位不满足时，worker A 和 worker B 应主动阻塞自身(susp

防止忙等待。当 worker C 所需的工作台中 A 和 B 不满足时，也应主动阻塞自身

（3）当 worker A、worker B 分别生产并放入新的零件 A、B 后，考虑唤醒由于

足而处于阻塞态的 worker C；同样地，当 worker C 从工作台取出零件 A、零件

唤醒因没有足够空位而处于阻塞态的 worker A、worker B。

程序设计：

定义并初始化信号量

mutexA：用于保护工作台上零件A的互斥信号量，初始值为1。

mutexB：用于保护工作台上零件B的互斥信号量，初始值为1。

mutexNumempty：用于保护工作台上空位的互斥信号量，初始值为1。

emptyCount：表示工作台上空位的信号量，初始值为N。

partACount：表示工作台上零件A的信号量，初始值为0。

partBCount：表示工作台上零件B的信号量，初始值为0。

worker A：

一次生产 m1=2 个 A 零件，然后申请 m1 个工作台空位，如果空位不足或者B的
被唤醒；如果空位足够，则将 m1 个 A 零件放入工作台，并唤醒工人 C。

worker B：

一次生产 m2=1 个 B 零件，然后申请 m2 个工作台空位，如果空位不足或者A的
被唤醒；如果空位足够，则将 m2 个 B 零件放入工作台，并唤醒工人 C。

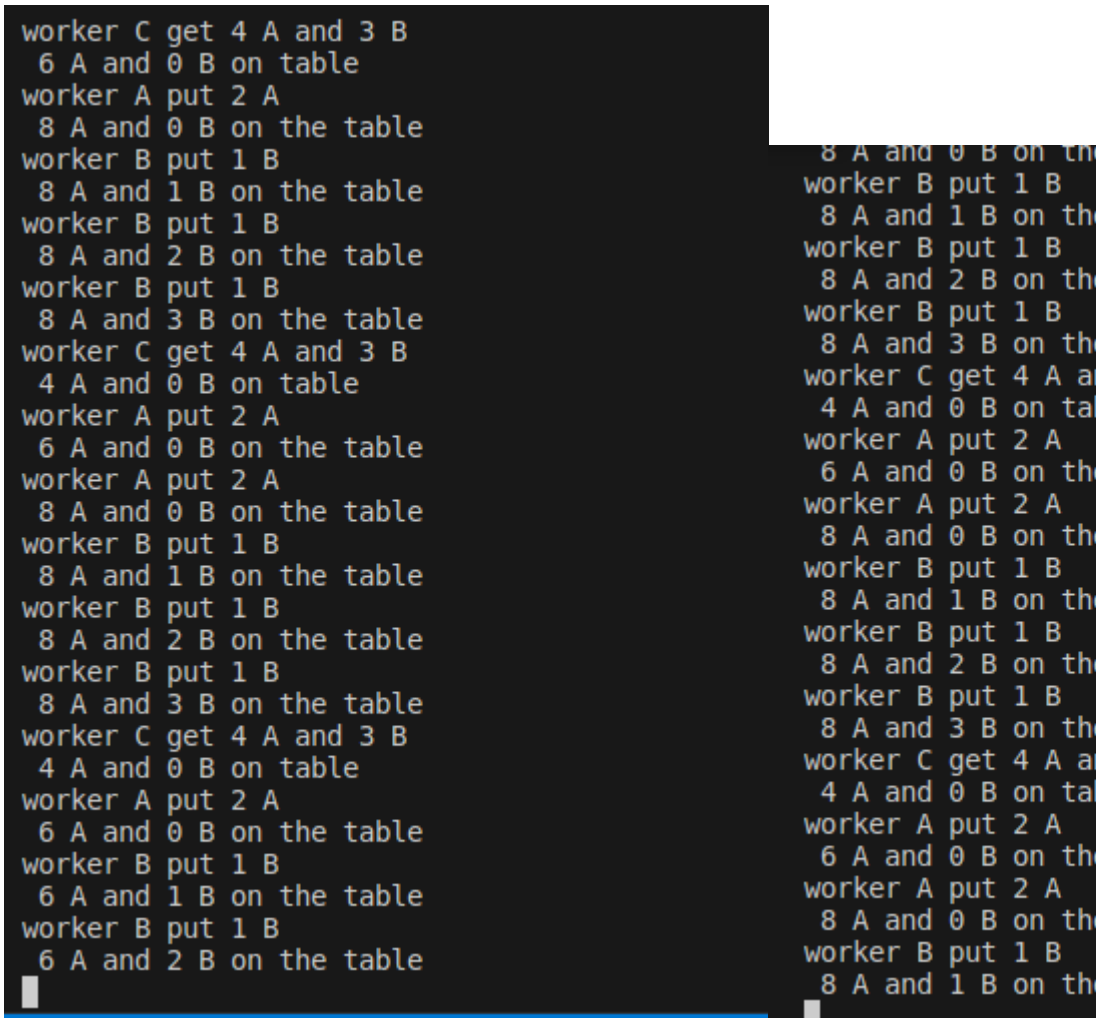
worker C：

一次需要 n1=4 个 A 零件和 n2=3 个 B 零件，然后申请 n1个 A 零件和 n2 个 B
阻塞自身，等待被唤醒；如果 A 和 B 零件足够，则从工作台取出n1 个 A 零件和
放 n1 + n2 个工作台空位，并唤醒工人 A 和工人 B。然后，工人 C 用 n1 个 A
个 C 产品。

程序代码：

```
int Numempty = N; // 工作台上的空位数

// 定义 worker A 的函数
void *workerA(void *arg) {
      while (1) {
         // 生产 m1 个 A 零件

         sleep(1);

         // 申请 m1 个工作台空位
         sem_wait(&mutexNumempty); // 互斥访问 Numempty
         if (Numempty < m1 || (Numempty <= n2-countB+2 && countB <= n2)
               sem_post(&mutexNumempty); // 释放互斥信号量
               sem_wait(&semA); // 阻塞自身，等待被唤醒
         }
         Numempty -= m1; // 更新空位数
         sem_post(&mutexNumempty); // 释放互斥信号量

         // 将 m1 个 A 零件放入工作台
         sem_wait(&mutexA); // 互斥访问 countA
         countA += m1; // 更新 A 零件数
         printf("worker A put %d A \n %d A and %d B on the table\n", m1
         sem_post(&mutexA); // 释放互斥信号量

         // 唤醒 worker C
         sem_post(&semC);
      }
}

// 定义 worker B 的函数
void *workerB(void *arg) {
      while (1) {
         // 生产 m2 个 B 零件
         sleep(1);

         // 申请 m2 个工作台空位
         sem_wait(&mutexNumempty); // 互斥访问 Numempty
         if (Numempty < m2||(countA <= n1&&Numempty <= n1-countA+1)) {
               sem_post(&mutexNumempty); // 释放互斥信号量
               sem_wait(&semB); // 阻塞自身，等待被唤醒
         }
         Numempty -= m2; // 更新空位数
         sem_post(&mutexNumempty); // 释放互斥信号量

         // 将 m2 个 B 零件放入工作台
         sem_wait(&mutexB); // 互斥访问 countB
         countB += m2; // 更新 B 零件数
         printf("worker B put %d B \n %d A and %d B on the table\n", m2
```

```
                countB -= n2; // 更新 B 零件数
                printf("worker C get %d A and %d B\n %d A and %d B on ta
                sem_post(&mutexB); // 释放互斥信号量
                sem_post(&mutexA); // 释放互斥信号量

                // 释放 n1 + n2 个工作台空位
                sem_wait(&mutexNumempty); // 互斥访问 Numempty
                Numempty += n1 + n2; // 更新空位数
                sem_post(&mutexNumempty); // 释放互斥信号量

                // 唤醒 worker A 和 worker B
                sem_post(&semA);
                sem_post(&semB);

                sleep(1);
            }
        }
}

int main() {
        //初始化信号量
        sem_init(&mutexA, 0, 1);
        sem_init(&mutexB, 0, 1);
        sem_init(&mutexNumempty, 0, 1);
        sem_init(&semA, 0, 0);
        sem_init(&semB, 0, 0);
        sem_init(&semC, 0, 0);

        //创建 worker A、worker B、worker C 的线程
        pthread_t tidA, tidB, tidC;
        pthread_create(&tidA, NULL, workerA, NULL);
        pthread_create(&tidB, NULL, workerB, NULL);
        pthread_create(&tidC, NULL, workerC, NULL);

        //等待线程结束
        pthread_join(tidA, NULL);
        pthread_join(tidB, NULL);
        pthread_join(tidC, NULL);

        sem_destroy(&mutexA);
        sem_destroy(&mutexB);
        sem_destroy(&mutexNumempty);
        sem_destroy(&semA);
        sem_destroy(&semB);
        sem_destroy(&semC);
        return 0;
}
```



通过这个实验，我深入理解了多线程环境下的同步和互斥问题，以及如何通过
题。实验不仅仅是一个多线程编程的练习，更是对并发编程中关键概念的应用

2.多核多线程编程及性能分析

实验目的

比较pthread线程加锁和不加锁对程序执行效率的影响。

通过比较加锁与不加锁程序执行之间的差异，加深对于多核多线程编程的理解

题目分析

本实验旨在通过多线程计算和数据分解的方式比较有锁和无锁两种情况下程序

率。具体而言，使用两个线程分别计算 apple_data->a 和 apple_data->b 的值

将这些值写入到 orange_data 结构中。这个问题涉及数据分解，通过将大任务

```c
struct orange {
    int a[ORANGE_MAX_VALUE];
    int b[ORANGE_MAX_VALUE];
};
void *apple_a(void *arg){
    struct apple * a_data = (struct apple_data*)arg;
    int i;
    for(i=0;i<APPLE_MAX_VALUE;i++){
        a_data->a+=i;
    }
    pthread_exit(NULL);
}
void *apple_b(void *arg){
    struct apple * b_data = (struct apple_data*)arg;
    unsigned long long i;
    for(i=0;i<APPLE_MAX_VALUE;i++){
        b_data->b+=i;
    }
    pthread_exit(NULL);
}
void *orange(void*arg){
    struct orange * data = (struct orange_data*)arg;
    unsigned long long sum =0;
    for(unsigned long long index=0;index<ORANGE_MAX_VALUE;index++)
    {
        sum+=data->a[index]+data->b[index];
    }
}
int main() {
    pthread_t tid1, tid2, tid3;
    struct apple apple_data;
    struct orange orange_data;
    printf("no lock:");
    pthread_create(&tid1, NULL, apple_a, &apple_data);
    pthread_create(&tid2, NULL, apple_b, &apple_data);
    pthread_create(&tid3, NULL, orange, &orange_data);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    pthread_join(tid3, NULL);
    printf("apple.a is %llu\n",apple_data.a);
    printf("apple.b is %llu\n",apple_data.b);
    return 0;
}


//lock版本
#include <pthread.h>
```

```
            a_data->a+=i;
            pthread_mutex_unlock(&a_data->lock);
        }
        pthread_exit(NULL);
    }
    void *apple_b(void *arg){
        struct apple * b_data = (struct apple_data*)arg;
        unsigned long long i;
        for(i=0;i<APPLE_MAX_VALUE;i++){
            pthread_mutex_lock(&b_data->lock);
            b_data->b+=i;
            pthread_mutex_unlock(&b_data->lock);
        }
        pthread_exit(NULL);
    }
    void *orange(void*arg){
        struct orange * data = (struct orange_data*)arg;
        unsigned long long sum =0;
        for(unsigned long long index=0;index<ORANGE_MAX_VALUE;index++)
        {
            sum+=data->a[index]+data->b[index];
        }
    }
    int main() {
        pthread_t tid1, tid2, tid3;
        struct apple apple_data;
        struct orange orange_data;
        printf(" lock:");
        if (pthread_mutex_init(&apple_data.lock, NULL) != 0) {
            perror("Mutex initialization failed");
            exit(EXIT_FAILURE);
        }
        pthread_create(&tid1, NULL, apple_a, &apple_data);
        pthread_create(&tid2, NULL, apple_b, &apple_data);
        pthread_create(&tid3, NULL, orange, &orange_data);
        pthread_join(tid1, NULL);
        pthread_join(tid2, NULL);
        pthread_join(tid3, NULL);
        pthread_mutex_destroy(&apple_data.lock);
        printf("apple.a is %llu\n",apple_data.a);
        printf("apple.b is %llu\n",apple_data.b);
        return 0;
    }
```

运行结果

结果分析

1. 加锁版本在每个线程访问共享数据apple时都需要进行加锁和解锁操作,这增

2. 不加锁版本可以同时访问共享数据,没有锁的开销,所以执行更快。

平均执行时间差异巨大，开销增加的非常明显，原因如下：

加锁和解锁的函数调用开销。pthread_mutex_lock和pthread_mutex_unlo的循环里都被调用,这额外增加了函数调用开销。

互斥锁的竞争开销。多个线程在同一锁上进行竞争,获取锁和释放锁需要线产生了额外的CPU开销。

线程阻塞和唤醒开销。一个线程如果获取锁失败,它会被阻塞。获取锁的线阻塞的线程,这也是额外的开销。