

# 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 produce a product C with  $n_1=4$  part As, and  $n_2=3$  part Bs. The worker of machining A and worker of machining B produce  $m_1=2$  part As and  $m_2=1$  part B independently each time. Then the  $m_1$  part A or  $m_2$  part B will be moved to a station, which can hold at most  $N=12$  of part As and part Bs altogether. The produced  $m_1$  part As must be put onto the station simultaneously. The workers must exclusively put a part on the station or get it from the station. In addition, the worker to make C must get all part of As and Bs, i.e.  $n_1$  part As and  $n_2$  part Bs, for one product C once.

Using semaphores to coordinate the three workers who are machining part A, part B and manufacturing the product C without deadlock.

It is required that

- (1) definition and initial value of each semaphore, and
- (2) the algorithm to coordinate 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 一次生产  $m_1=2$  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  $n_1=4$  个 A 和  $n_2=3$  个 B。因此，如果当前工作台空位小于  $m_1$ ，worker A 被阻塞；如果当前工作台没有足够的 A、B，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合理的。

例如，假设当前工作台已经放置了  $N=11$  个零件，只有 1 个空位。worker A 生产出 2 个 A，worker B 生产出 1 个 B。如果 worker A 先通过两次 `wait(empty)` 操作申请两个工作台空位，将被阻塞，生产的零件 A 没有放入工作台。随后 worker B 再执行 `wait(empty)` 申请空位也将被阻塞，而此时工作台有 1 个空位。

因此，正确的解决方案是，

(1) 利用用户空间计数变量 `countA`、`countB`、`Numempty`，配合互斥信号量 `mutexA`、`mutexB`、`mutexNumempty`，统计工作台中零件 A、零件 B、空单元的数目，当有足够多的零件 A、B 和工作台空位时，再一次性申请/获取多个所需的零件 A、零件 B、工作台空位，即将 worker A 所需的  $m_1$  个工作台空位作为一个整体一次性地申请，worker C 所需的工作台空位中的  $n_1$  个 A

和  $n_2$  个 B 作为一个整体一次性地申请；

(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(suspend/block)，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，考虑唤醒由于所需零件不足而处于阻塞态的 worker C；同样地，当 worker C 从工作台取出零件 A、零件 B 后，考虑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  $m_1=2$  个 A 零件，然后申请  $m_1$  个工作台空位，如果空位不足或者B的数量不足，则阻塞自身，等待被唤醒；如果空位足够，则将  $m_1$  个 A 零件放入工作台，并唤醒工人 C。

worker B：

一次生产  $m_2=1$  个 B 零件，然后申请  $m_2$  个工作台空位，如果空位不足或者A的数量不足，则阻塞自身，等待被唤醒；如果空位足够，则将  $m_2$  个 B 零件放入工作台，并唤醒工人 C。

worker C：

一次需要  $n_1=4$  个 A 零件和  $n_2=3$  个 B 零件，然后申请  $n_1$  个 A 零件和  $n_2$  个 B 零件，如果 A 或 B 零件不足，则阻塞自身，等待被唤醒；如果 A 和 B 零件足够，则从工作台取出  $n_1$  个 A 零件和  $n_2$  个 B 零件，并释放  $n_1 + n_2$  个工作台空位，并唤醒工人 A 和工人 B。然后，工人 C 用  $n_1$  个 A 零件和  $n_2$  个 B 零件生产一个 C 产品。

## 程序代码：

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

#define N 12 // 工作台最多容纳的零件数
#define m1 2 // worker A produce 2A onetime
#define m2 1 // worker B produce 1B onetime
#define n1 4 // worker C consum 4A onetime
#define n2 3 // worker C consum 3B onetime

// 定义信号量
```

```

sem_t mutexA, mutexB, mutexNumempty;
sem_t semA, semB, semC;

// 定义计数变量
int countA = 0; // 工作台上的 A 零件数
int countB = 0; // 工作台上的 B 零件数
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)) { // 如果空位不足
            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, countA, countB);
        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)) { // 如果空位不足
            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, countA,
countB);
        sem_post(&mutexB); // 释放互斥信号量

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

// 定义 worker C 的函数
void *workerC(void *arg) {
    while (1) {
        // 申请 n1 个 A 零件和 n2 个 B 零件
        sem_wait(&mutexA); // 互斥访问 countA
        sem_wait(&mutexB); // 互斥访问 countB
        if (countA < n1 || countB < n2) { // 如果 A 或 B 零件不足
            sem_post(&mutexB); // 释放互斥信号量
            sem_post(&mutexA); // 释放互斥信号量
            sem_wait(&semC); // 阻塞自身，等待被唤醒
        } else { // 如果 A 和 B 零件足够
            // 先更新 A、B 零件数，再打印信息
            countA -= n1; // 更新 A 零件数
            countB -= n2; // 更新 B 零件数
            printf("worker C get %d A and %d B\n %d A and %d B on table\n",
n1, n2, countA, countB);
            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;
}
```

**运行结果：**

```
worker C get 4 A and 3 B
6 A and 0 B on table
worker A put 2 A
8 A and 0 B on the table
worker B put 1 B
8 A and 1 B on the table
worker B put 1 B
8 A and 2 B on the table
worker B put 1 B
8 A and 3 B on the table
worker C get 4 A and 3 B
4 A and 0 B on table
worker A put 2 A
6 A and 0 B on the table
worker A put 2 A
8 A and 0 B on the table
worker B put 1 B
8 A and 1 B on the table
worker B put 1 B
8 A and 2 B on the table
worker B put 1 B
8 A and 3 B on the table
worker C get 4 A and 3 B
4 A and 0 B on table
worker A put 2 A
6 A and 0 B on the table
worker B put 1 B
6 A and 1 B on the table
worker B put 1 B
6 A and 2 B on the table
```

```
8 A and 0 B on the table
worker B put 1 B
8 A and 1 B on the table
worker B put 1 B
8 A and 2 B on the table
worker B put 1 B
8 A and 3 B on the table
worker C get 4 A and 3 B
4 A and 0 B on table
worker A put 2 A
6 A and 0 B on the table
worker A put 2 A
8 A and 0 B on the table
worker B put 1 B
8 A and 1 B on the table
worker B put 1 B
8 A and 2 B on the table
worker B put 1 B
8 A and 3 B on the table
worker C get 4 A and 3 B
4 A and 0 B on table
worker A put 2 A
6 A and 0 B on the table
worker A put 2 A
8 A and 0 B on the table
worker B put 1 B
8 A and 1 B on the table
```

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，以及如何通过合理的设计和调整来解决这些问题。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和理解。

## 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的执行效率。具体而言，使用两个线程分别计算 `apple_data->a` 和 `apple_data->b` 的值，以及一个线程用于将这些值写入到 `orange_data` 结构中。这个问题涉及数据分解，通过将大任务分解成多个子任务并分配给多个线程，以提高计算效率。

至于比较加锁与不加锁的效率可以用程序运行结束与开始的时间差计算

## 代码设计

```
//no lock版本
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>
#define ORANGE_MAX_VALUE 1000000
#define APPLE_MAX_VALUE 100000000
struct apple {
    unsigned long long a;
    unsigned long long b;
};
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>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>
#define ORANGE_MAX_VALUE 1000000
#define APPLE_MAX_VALUE 100000000
struct apple {
    unsigned long long a;
    unsigned long long b;
    pthread_mutex_t lock;
};
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++){
        pthread_mutex_lock(&a_data->lock);
        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

```

hby@hby-virtual-machine:~/Desktop/os_ex$ time ./lock
lock:apple.a is 4999999950000000
apple.b is 4999999950000000

real    0m8.042s
user    0m8.025s
sys     0m7.752s

hby@hby-virtual-machine:~/Desktop/os_ex$ time ./unlock
no lock:apple.a is 4999999950000000
apple.b is 4999999950000000

real    0m0.410s
user    0m0.695s
sys     0m0.064s

```

	1	2	3	4	5	6	7	8	9	10	best
lock	0.41	0.377	0.389	0.411	0.396	0.406	0.404	0.405	0.392	0.427	0.377
unlock	8.042	7.966	8.891	7.93	8.174	8.269	8.096	7.855	7.923	8.426	7.855

## 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,这增加了执行的开销。
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\_unlock在每个访问共享数据的循环里都被调用,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程切换和上下文切换,这也产生了额外的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。获取锁的线程释放锁后,需要唤醒其他阻塞的线程,这也是额外的开销。