

# Assignment3 实验报告

58121124 张博彦

2023 年 6 月 10 号

## 1 实验内容

1. 编译代码 badcnt.c, 使用 `gcc badcnt.c -o badcnt -lpthread` 运行可执行文件 badcnt 并观察其输出结果。
2. 修改程序 badcnt.c, 使程序总是产生预期的输出 (值为  $2*NITER$ )。
3. 完成不完整的代码 producer-consumer.c, 用 Posix 线程和 semaphores 来实现生产者-消费者问题的解决方案。

## 2 实验目的

Learn to use POSIX Semaphores synchronous threads to compile related code and solve related problems learned in class.

## 3 设计思路和流程图

### 3.1 实验内容 1

调用终端命令 `gcc badcnt.c -o badcnt -lpthread` 观察输出结果

### 3.2 实验内容 2

使用互斥锁 `sem_t mutex` 进行进程阻塞, 从而达到保护对 cnt 的访问, 确保每次只有一个线程能够修改它。

### 3.3 实验内容 3

依据生产者，消费者问题的解决方法：一个线程消费（或生产）完，其他线程才能进行竞争 CPU，获得消费（或生产）的机会。对于这一点，可以使用条件变量进行线程间的同步：生产者线程在 product 之前，需要 wait 直至获取自己所需的信号量之后，才会进行 product 的操作；同样，对于消费者线程，在 consume 之前需要 wait 直到没有线程在访问共享区（缓冲区），再进行 consume 的操作，之后再解锁并唤醒其他可用阻塞线程。根据以上原理编写 Consumer 函数。

## 4 主要数据结构及其说明

在 Consumer 函数中，使用 `sem_wait(&shared.full)` 确认是否有资源可以消耗，若没有则等待；使用 `sem_wait(&shared.mutex)` 等待其他进程结束资源使用；之后便进行资源使用的输出打印，表示使用该资源，最后释放 buffer 并将资源空余数加一。

## 5 源程序

### 5.1 goodcnt.c

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

#define NITER 1000000

int cnt = 0;
sem_t mutex;

void * Count(void * a)
{
    int i, tmp;
```

```

    for(i = 0; i < NITER; i++)
    {
        sem_wait(&mutex);
        tmp = cnt;        /* copy the global cnt locally */
        tmp = tmp+1;      /* increment the local copy */
        cnt = tmp;        /* store the local value into the global cnt */
        sem_post(&mutex);
    }
}

int main(int argc, char * argv[])
{
    pthread_t tid1, tid2;
    sem_init(&mutex, 0, 1);
    if(pthread_create(&tid1, NULL, Count, NULL))
    {
        printf("\n ERROR creating thread 1");
        exit(1);
    }

    if(pthread_create(&tid2, NULL, Count, NULL))
    {
        printf("\n ERROR creating thread 2");
        exit(1);
    }

    if(pthread_join(tid1, NULL))          /* wait for the thread 1 to finish */
    {
        printf("\n ERROR joining thread");
        exit(1);
    }

    if(pthread_join(tid2, NULL))          /* wait for the thread 2 to finish */

```

```

    {
        printf("\n ERROR joining thread");
        exit(1);
    }

    if (cnt < 2 * NITER)
        printf("\n BOOM! cnt is [%d], should be %d\n", cnt, 2*NITER);
    else
        printf("\n OK! cnt is [%d]\n", cnt);

    sem_destroy(&mutex);
    pthread_exit(NULL);
}

```

## 5.2 producer-consumer.c

```

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

#define BUFF_SIZE    5           /* total number of slots */
#define NP           3           /* total number of producers */
#define NC           3           /* total number of consumers */
#define NITERS       4           /* number of items produced/consumed */

typedef struct {
    int buf[BUFF_SIZE];          /* shared var */
    int in;                      /* buf[in%BUFF_SIZE] is the first empty slot */
    int out;                     /* buf[out%BUFF_SIZE] is the first full slot */
    sem_t full;                  /* keep track of the number of full spots */
    sem_t empty;                 /* keep track of the number of empty spots */
    sem_t mutex;                 /* enforce mutual exclusion to shared data */
}

```

```

} sbuf_t;

sbuf_t shared;

void *Producer(void *arg)
{
    int i, item, index;

    index = (int) arg;

    pthread_detach(pthread_self());

    for (i=0; i < NITERS; i++) {

        /* Produce item */
        item = i;

        /* Prepare to write item to buf */

        /* If there are no empty slots, wait */
        sem_wait(&shared.empty);
        /* If another thread uses the buffer, wait */
        sem_wait(&shared.mutex);
        shared.buf[shared.in] = item;
        shared.in = (shared.in+1)%BUFF_SIZE;
        printf("[P%d] Producing %d ...\n", index, item); fflush(stdout);
        /* Release the buffer */
        sem_post(&shared.mutex);
        /* Increment the number of full slots */
        sem_post(&shared.full);

        /* Interleave producer and consumer execution */
        if (i % 2 == 1) sleep(1);
    }
}

```

```

    }
    return NULL;
}

void *Consumer(void *arg)
{
    int i, item, index;

    index = (int)arg;

    pthread_detach(pthread_self());

    for (i = 0; i < NITERS; i++) {
        /* Prepare to read item from buf */

        /* If there are no full slots, wait */
        sem_wait(&shared.full);
        /* If another thread uses the buffer, wait */
        sem_wait(&shared.mutex);
        item = shared.buf[shared.out];
        shared.out = (shared.out + 1) % BUFF_SIZE;
        printf("[C%d] Consuming %d ...\n", index, item); fflush(stdout);
        /* Release the buffer */
        sem_post(&shared.mutex);
        /* Increment the number of empty slots */
        sem_post(&shared.empty);

        /* Interleave producer and consumer execution */
        if (i % 2 == 1) sleep(1);
    }
    return NULL;
}

```

```

int main()
{
    pthread_t idP, idC;
    int index;

    sem_init(&shared.full, 0, 0);
    sem_init(&shared.empty, 0, BUFF_SIZE);
    sem_init(&shared.mutex, 0, 1); // Initialize mutex

    for (index = 0; index < NP; index++) {
        pthread_create(&idP, NULL, Producer, (void *)index);
    }

    for (index = 0; index < NC; index++) {
        pthread_create(&idC, NULL, Consumer, (void *)index);
    }

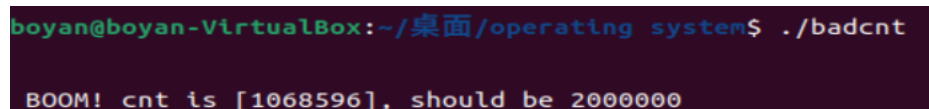
    pthread_exit(NULL);
}

```

## 6 程序运行结果及分析

### 6.1 实验内容 1

运行结果如下



```

boyan@boyan-VirtualBox:~/桌面/operating system$ ./badcnt
BOOM! cnt is [1068596], should be 2000000

```

由于在多线程环境中，对共享变量 `cnt` 的访问是不同步的。由于两个线程同时对 `cnt` 进行递增操作，存在竞争条件，使得多个线程同时访问共享资源，并试图对其进行修改，从而导致最终结果依赖于线程执行的顺序。在这

种情况下，两个线程可能同时读取 cnt 的值，进行递增操作，并将结果写回 cnt，但由于操作的交替执行顺序不确定，最终的结果可能与预期不符。

## 6.2 实验内容 2

运行结果如下

```
boyan@boyan-VirtualBox:~/桌面/operating system$ ./badcnt  
  
OK! cnt is [2000000]
```

经过程序修改，成功使得 cnt 的值为 2\*NITER。

## 6.3 实验内容 3

运行结果如下

```
[P0] Producing 0 ...  
[P0] Producing 1 ...  
[C0] Consuming 0 ...  
[C0] Consuming 1 ...  
[P1] Producing 0 ...  
[C1] Consuming 0 ...  
[P1] Producing 1 ...  
[C1] Consuming 1 ...  
[P2] Producing 0 ...  
[P2] Producing 1 ...  
[C2] Consuming 0 ...  
[C2] Consuming 1 ...  
[P2] Producing 2 ...  
[P2] Producing 3 ...  
[P0] Producing 2 ...  
[P0] Producing 3 ...  
[C0] Consuming 2 ...  
[C0] Consuming 3 ...  
[C1] Consuming 2 ...  
[C1] Consuming 3 ...  
[P1] Producing 2 ...  
[P1] Producing 3 ...  
[C2] Consuming 2 ...  
[C2] Consuming 3 ...
```

完善程序后，根据输出结果可知生产者消费者的问题成功解决。



## 7 实验体会

在本次实验中，了解到了进程访问共享资源区的问题，学会了使用互斥锁达到每个进程互斥访问资源，从而得到正确的运行结果。同时将课上学到的生产者消费者问题使用 C 语言进行了编程解决。