【Linux 多线程】三个经典同步问题

在了解了《同步与互斥的区别 》之后，我们来看看几个经典的线程同步的例子。通过具体场景可以让我们学会分析和解决这类进程同步的问题，以便以后应用在实际的项目中。

一、生产者-消费者问题

问题描述：

一组生产者进程和一组消费者进程共享一个初始为空、大小为 n 的缓冲区，只有缓冲区没满时，生产者才能把消息放入到缓冲区，否则必须等待；只有缓冲区不空时，消费者才能从中取出消息，否则必须等待。由于缓冲区是临界资源，它只允许一个生产者放入消息，或者一个消费者从中取出消息。

分析：

1. 关系分析：生产者和消费者对缓冲区互斥访问是互斥关系，同时生产者和消费者又是一个相互协作的关系，只有生产者生产之后，消费者才能消费，它们也是同步关系。

2. 整理思路：这里比较简单，只有生产者和消费者两个进程，且这两个进程存在着互斥关系和同步关系。那么需要解决的是互斥和同步的 PV 操作的位置。

3. 信号量设置：信号量 `mutex` 作为互斥信号量，用于控制互斥访问缓冲池，初值为 1 ;信号量 `full` 用于记录当前缓冲池中“满”缓冲区数 ,初值为 0；信号量 `empty` 用于记录当前缓冲池中“空”缓冲区数，初值为 n。

代码示例：（semaphore 类的封装见下文）

```
#include<iostream>
#include<unistd.h>  // sleep
#include<pthread.h>
#include"semaphore.h"
using namespace std;
#define N 5

semaphore mutex("/", 1);           // 临界区互斥信号量
semaphore empty("/home", N);       // 记录空缓冲区数，初值为 N
semaphore full("/home/songlee",0); // 记录满缓冲区数，初值为 0
int buffer[N];                     // 缓冲区，大小为 N
int i=0;
int j=0;

void* producer(void* arg)
{
        empty.P();                 // empty 减 1
        mutex.P();

        buffer[i] = 10 + rand() % 90;
        printf("Producer %d write Buffer[%d]: %d\n",arg,i+1,buffer[i]);
        i = (i+1) % N;

        mutex.V();
        full.V();                  // full 加 1
}

void* consumer(void* arg)
{
        full.P();                  // full 减 1
        mutex.P();

        printf("                              \033[1;31m");
        printf("Consumer %d read Buffer[%d]: %d\n",arg,j+1,buffer[j]);
```

```
35        printf("\033[0m");
36        j = (j+1) % N;
37
38        mutex.V();
39        empty.V();                // empty 加 1
40}
41
42
43int main()
44{
45        pthread_t id[10];
46
47        // 开 10 个生产者线程，10 个消费者线程
48        for(int k=0; k<10; ++k)
49                pthread_create(&id[k], NULL, producer, (void*)(k+1));
50
51        for(int k=0; k<10; ++k)
52                pthread_create(&id[k], NULL, consumer, (void*)(k+1));
53
54        sleep(1);
55        return 0;
56}
```

编译运行输出结果：

```
1Producer 1 write Buffer[1]: 83
2Producer 2 write Buffer[2]: 26
3Producer 3 write Buffer[3]: 37
4Producer 5 write Buffer[4]: 35
5Producer 4 write Buffer[5]: 33
6                            Consumer 1 read Buffer[1]: 83
7Producer 6 write Buffer[1]: 35
8                            Consumer 2 read Buffer[2]: 26
9                            Consumer 3 read Buffer[3]: 37
10                           Consumer 4 read Buffer[4]: 35
11                           Consumer 5 read Buffer[5]: 33
12                           Consumer 6 read Buffer[1]: 35
13Producer 7 write Buffer[2]: 56
14Producer 8 write Buffer[3]: 22
15Producer 10 write Buffer[4]: 79
16                           Consumer 9 read Buffer[2]: 56
17                           Consumer 10 read Buffer[3]: 22
18Producer 9 write Buffer[5]: 11
19                           Consumer 7 read Buffer[4]: 79
```

```
20                              Consumer 8 read Buffer[5]: 11
```

二、读者-写者问题

问题描述：

有读者和写者两组并发线程，共享一个文件，当两个或以上的读线程同时访问共享数据时不会产生副作用，但若某个写线程和其他线程（读线程或写线程）同时访问共享数据时则可能导致数据不一致的错误。因此要求：

- 允许多个读者可以同时对文件执行读操作；
- 只允许一个写者往文件中写信息；
- 任一写者在完成写操作之前不允许其他读者或写者工作；
- 写者执行写操作前，应让已有的读者和写者全部退出。

分析：

1. 关系分析：由题目分析可知，读者和写者是互斥的，写者和写者也是互斥的，而读者和读者不存在互斥问题。

2. 整理思路：写者是比较简单的，它与任何线程互斥，用互斥信号量的 PV 操作即可解决。读者的问题比较复杂，它必须实现与写者的互斥，多个读者还可以同时读。所以，在这里用到了一个计数器，用它来判断当前是否有读者读文件。当有读者的时候写者是无法写文件的，此时读者会一直占

用文件，当没有读者的时候写者才可以写文件。同时，不同的读者对计数器的访问也应该是互斥的。

3. 信号量设置：首先设置一个计数器 `count`，用来记录当前的读者数量，初值为 0；设置互斥信号量 `mutex`，用于保护更新 count 变量时的互斥；设置互斥信号量 `rw` 用于保证读者和写者的互斥访问。

代码示例：

```
#include<iostream>
#include<unistd.h>  // sleep
#include<pthread.h>
#include"semaphore.h"
using namespace std;

int count = 0;           // 记录当前的读者数量
semaphore mutex("/",1);  // 用于保护更新 count 变量时的互斥
semaphore rw("/home",1); // 用于保证读者和写者的互斥

void* writer(void* arg)
{
        rw.P();             // 互斥访问共享文件

        printf("  Writer %d start writing...\n", arg);
        sleep(1);
        printf("  Writer %d finish writing...\n", arg);

        rw.V();             // 释放共享文件
}

void* reader(void* arg)
{
        mutex.P();          // 互斥访问 count 变量
        if(count == 0)       // 当第一个读线程读文件时
                rw.P();          // 阻止写线程写
        ++count;            // 读者计数器加 1
        mutex.V();          // 释放 count 变量

```

```
30        printf("Reader %d start reading...\n", arg);
31        sleep(1);
32        printf("Reader %d finish reading...\n", arg);
33
34        mutex.P();          // 互斥访问 count 变量
35        --count;            // 读者计数器减 1
36        if(count == 0)      // 当最后一个读线程读完文件
37                rw.V();         // 允许写线程写
38        mutex.V();          // 释放 count 变量
39}
40
41
42int main()
43{
44        pthread_t id[8];    // 开 6 个读线程，2 个写线程
45
46        pthread_create(&id[0], NULL, reader, (void*)1);
47        pthread_create(&id[1], NULL, reader, (void*)2);
48
49        pthread_create(&id[2], NULL, writer, (void*)1);
50        pthread_create(&id[3], NULL, writer, (void*)2);
51
52        pthread_create(&id[4], NULL, reader, (void*)3);
53        pthread_create(&id[5], NULL ,reader, (void*)4);
54        sleep(2);
55        pthread_create(&id[6], NULL, reader, (void*)5);
56        pthread_create(&id[7], NULL ,reader, (void*)6);
57
58        sleep(4);
59        return 0;
60}
```

编译运行的结果如下：

```
1Reader 2 start reading...
2Reader 1 start reading...
3Reader 3 start reading...
4Reader 4 start reading...
5Reader 1 finish reading...
6Reader 2 finish reading...
7Reader 3 finish reading...
8Reader 4 finish reading...
9  Writer 1 start writing...
10  Writer 1 finish writing...
```

```
11  Writer 2 start writing...
12  Writer 2 finish writing...
13Reader 5 start reading...
14Reader 6 start reading...
15Reader 5 finish reading...
16Reader 6 finish reading...
```

三、哲学家进餐问题

问题描述：

一张圆桌上坐着 5 名哲学家，桌子上每两个哲学家之间摆了**一根**筷子，桌子的中间是一碗米饭，如图所示：

哲学家们倾注毕生精力用于思考和进餐，哲学家在思考时，并不影响他人。只有当哲学家饥饿的时候，才试图拿起左、右两根筷子（一根一根拿起）。如果筷子已在他人手上，则需等待。饥饿的哲学家只有同时拿到了两根筷子才可以开始进餐，当进餐完毕后，放下筷子继续思考。

分析：

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. 整理思路：显然这里有 5 个线程，那么要如何让一个哲学家拿到左右两个筷子而不造成死锁或饥饿现象？解决方法有两个，一个是让他们同时拿

两个筷子；二是对每个哲学家的动作制定规则，避免饥饿或死锁现象的发生。

3. 信号量设置：定义互斥信号量数组 `chopstick[5] = {1,1,1,1,1}`用于对 5 根筷子的互斥访问。

示例代码：

```
semaphore chopstick[5] = {1,1,1,1,1}  // 信号量数组
Pi()                                  // i 号哲学家的线程
{
        do
        {
                P(chopstick[i]);              // 取左边筷子
                P(chopstick[(i+1)%5]);        // 取右边筷子
                eat;                          // 进餐
                V(chopstick[i]);              // 放回左边筷子
                V(chopstick[(i+1)%5]);        // 放回右边筷子
                think;                        // 思考
        }while(1);
}
```

上面的伪代码存在一个问题：当五个哲学家都想要进餐，分别拿起他们左边筷子的时候（都恰好执行完 `P(chopstick[i])`），筷子已经被拿光了，等到他们再想拿右边的筷子的时候，就全被阻塞了，这就出现了死锁。

为了防止死锁的发生，可以对哲学家线程施加一些限制条件，比如：

- 至多允许四个哲学家同时进餐；
- 仅当一个哲学家左右两边的筷子都可用时才允许他抓起筷子；
- 对哲学家顺序编号，要求奇数号哲学家先抓左边的筷子，然后再抓他右边的筷子，而偶数号哲学家刚好相反。

这里，我们采用第二种方法来改进上面的算法，即当一个哲学家左右两边的筷子都可用时，才允许他抓起筷子。

```
#include<iostream>
#include<vector>
#include<unistd.h>  // sleep
#include<pthread.h>
#include"semaphore.h"
using namespace std;

vector<semaphore*> chopstick;   // 信号量数组
semaphore mutex("/", 1);  // 设置取左右筷子的信号量 <-- 关键

void* P1(void* arg)  // 第 1 个哲学家线程
{
        mutex.P();                // 在取筷子前获得互斥量
        chopstick[0]->P();         // 取左边筷子
        chopstick[1]->P();         // 取右边筷子
        mutex.V();                // 释放取筷子的信号量

        printf("Philosopher 1 eat.\n");

        chopstick[0]->V();          // 放回左边筷子
        chopstick[1]->V();          // 放回右边筷子
}

void* P2(void* arg)  // 第 2 个哲学家线程
{
        mutex.P();                // 在取筷子前获得互斥量
        chopstick[1]->P();         // 取左边筷子
        chopstick[2]->P();         // 取右边筷子
        mutex.V();                // 释放取筷子的信号量

        printf("Philosopher 2 eat.\n");

        chopstick[1]->V();          // 放回左边筷子
        chopstick[2]->V();          // 放回右边筷子
}


void* P3(void* arg)  // 第 3 个哲学家线程
{
```

```
40          mutex.P();                  // 在取筷子前获得互斥量
41          chopstick[2]->P();          // 取左边筷子
42          chopstick[3]->P();          // 取右边筷子
43          mutex.V();                  // 释放取筷子的信号量
44
45          printf("Philosopher 3 eat.\n");
46
47          chopstick[2]->V();          // 放回左边筷子
48          chopstick[3]->V();          // 放回右边筷子
49}
50
51void* P4(void* arg)  // 第 4 个哲学家线程
52{
53          mutex.P();                  // 在取筷子前获得互斥量
54          chopstick[3]->P();          // 取左边筷子
55          chopstick[4]->P();          // 取右边筷子
56          mutex.V();                  // 释放取筷子的信号量
57
58          printf("Philosopher 4 eat.\n");
59
60          chopstick[3]->V();          // 放回左边筷子
61          chopstick[4]->V();          // 放回右边筷子
62}
63
64
65void* P5(void* arg)  // 第 5 个哲学家线程
66{
67          mutex.P();                  // 在取筷子前获得互斥量
68          chopstick[4]->P();          // 取左边筷子
69          chopstick[0]->P();          // 取右边筷子
70          mutex.V();                  // 释放取筷子的信号量
71
72          printf("Philosopher 5 eat.\n");
73
74          chopstick[4]->V();          // 放回左边筷子
75          chopstick[0]->V();          // 放回右边筷子
76}
77
78int main()
79{
80          semaphore *sem1 = new semaphore("/home", 1);
81          semaphore *sem2 = new semaphore("/home/songlee", 1);
82          semaphore *sem3 = new semaphore("/home/songlee/java", 1);
83          semaphore *sem4 = new semaphore("/home/songlee/ADT", 1);
```

```
84          semaphore *sem5 = new semaphore("/home/songlee/Test", 1);
85          chopstick.push_back(sem1);
86          chopstick.push_back(sem2);
87          chopstick.push_back(sem3);
88          chopstick.push_back(sem4);
89          chopstick.push_back(sem5);
90
91          pthread_t id;
92
93          pthread_create(&id, NULL, P1, NULL);
94          pthread_create(&id, NULL, P2, NULL);
95          pthread_create(&id, NULL, P3, NULL);
96          pthread_create(&id, NULL, P4, NULL);
97          pthread_create(&id, NULL, P5, NULL);
98
99          sleep(1);
100         delete sem1;
101         delete sem2;
102         delete sem3;
103         delete sem4;
104         delete sem5;
105         return 0;
106}
```

编译运行的结果如下：

```
1Philosopher 2 eat.
2Philosopher 1 eat.
3Philosopher 3 eat.
4Philosopher 4 eat.
5Philosopher 5 eat.
```

注意：创建信号量时的 路径参数 请改成你的系统中存在的路径！！！

附：semaphore 类的封装

上面的代码中都使用了这个 `semaphore` 类，实现如下：

- **semaphore.h**

```
#pragma once
#include<iostream>
#include<cstdio>
#include<cstdlib>
#include<sys/sem.h>
using namespace std;

// 联合体，用于 semctl 初始化
union semun {
    int              val; /*for SETVAL*/
    struct semid_ds *buf;
    unsigned short  *array;
};


class semaphore {
private:
        int sem_id;
        int init_sem(int);
public:
        semaphore(const char*, int); /*构造函数*/
        ~semaphore();                /*析构函数*/
        void P();                    /*P 操作*/
        void V();                    /*V 操作*/
};
```

- **semaphore.cpp**

```
#include"semaphore.h"

semaphore::semaphore(const char* path, int value)
{
```

```
5         key_t key;
6    /*获取 key 值*/
7    if((key = ftok(path, 'z')) < 0) {
8        perror("ftok error");
9        exit(1);
10   }
11
12   /*创建信号量集，其中只有一个信号量*/
13   if((sem_id = semget(key, 1, IPC_CREAT|0666)) == -1) {
14       perror("semget error");
15       exit(1);
16   }
17
18        init_sem(value);
19}
20
21
22semaphore::~semaphore()
23{
24   union semun tmp;
25   if(semctl(sem_id, 0, IPC_RMID, tmp) == -1) {
26       perror("Delete Semaphore Error");
27                exit(1);
28        }
29}
30
31
32void semaphore::P()
33{
34   struct sembuf sbuf;
35   sbuf.sem_num = 0; /*序号*/
36   sbuf.sem_op = -1; /*P 操作*/
37   sbuf.sem_flg = SEM_UNDO;
38
39   if(semop(sem_id, &sbuf, 1) == -1) {
40       perror("P operation Error");
41   }
42}
43
44
45void semaphore::V()
46{
47   struct sembuf sbuf;
48   sbuf.sem_num = 0; /*序号*/
```

```
49    sbuf.sem_op = 1;  /*V 操作*/
50    sbuf.sem_flg = SEM_UNDO;
51
52    if(semop(sem_id, &sbuf, 1) == -1) {
53        perror("V operation Error");
54    }
55}
56
57
58// 初始化信号量
59int semaphore::init_sem(int value)
60{
61    union semun tmp;
62    tmp.val = value;
63    if(semctl(sem_id, 0, SETVAL, tmp) == -1) {
64        perror("Init Semaphore Error");
65        return -1;
66    }
67    return 0;
68}
```

在这里，要创建不同的信号量，必须传递不同的路径参数（这样获取的 key 值才会不一样）。