中山大學

SUN YAT-SEN UNIVERSITY

实 验 报 告

课程名称：操作系统

姓　　名：方桂安

学　　号：**20354027**

专业班级：**2020** 级智能科学与技术

任课教师：吴贺俊

2022 年 10 月 28 日

实验报告成绩评定表

评定项目	内　　容	满 分	评　分	总　分
实验态度	态度端正、遵守纪律、出勤情况	10		
实验过程	按要求完成算法设计、代码书写、注释清晰、运行结果正确	30		
实验记录	展示讲解清楚、任务解决良好、实验结果准确	20		
报告撰写	报告书写规范、内容条理清楚、表达准确规范、上交及时、无抄袭，抄袭记 0 分，提供报告供抄袭者扣分。	40		
评语： **指导老师签字：**　　年　　月　　日				

实验报告正文

实验四　互斥控制与进程同步

一、　实验目的

1. 加强对进程同步和互斥的理解，学会使用信号量解决资源共享问题。
2. 熟悉 Linux 进程同步原语。
3. 掌握信号量 wait/signal 原语的使用方法，理解信号量的定义、赋初值及 wait/signal 操作。

二、　实验内容

1.　任务描述

i. 用进程或线程实现前面的两个算法，算法 A 和 B，其中：

- 算法 A：算法一到四任选一个；
- 算法 B:Dekker 算法或者 Peterson 算法

观察算法多次运行，检测是否有同时进入临界区的情况，分析原因，撰写报告。

ii. 编写程序，使用 Linux 操作系统中的信号量机制模拟实现生产者-消费者问题。设有一个生产者和一个消费者，缓冲区可以存放产品，生产者不断生成产品放入缓冲区，消费者不断从缓冲区中取出产品，消费产品。

2.　实验方案

i. 算法一到四

```
   /* PROCESS 0 /*                  /* PROCESS 1 */
.                                .
.                                .
while (turn != 0)                while (turn != 1)
   /* do nothing */ ;               /* do nothing */;
/* critical section*/;            /* critical section*/;
turn = 1;                        turn = 0;
.                                .
```

(a) First attempt

```
   /* PROCESS 0 */                  /* PROCESS 1 */
.                                .
.                                .
while (flag[1])                  while (flag[0])
   /* do nothing */;                /* do nothing */;
flag[0] = true;                  flag[1] = true;
/*critical section*/;            /* critical section*/;
flag[0] = false;                 flag[1] = false;
.                                .
```

(b) Second attempt

```
   /* PROCESS 0 */                  /* PROCESS 1 */
.                                .
.                                .
flag[0] = true;                  flag[1] = true;
while (flag[1])                  while (flag[0])
   /* do nothing */;                /* do nothing */;
/* critical section*/;            /* critical section*/;
flag[0] = false;                 flag[1] = false;
.                                .
```

(c) Third attempt

```
   /* PROCESS 0 */                  /* PROCESS 1 */
.                                .
.                                .
flag[0] = true;                  flag[1] = true;
while (flag[1]) {                while (flag[0]) {
   flag[0] = false;                 flag[1] = false;
   /*delay */;                      /*delay */;
   flag[0] = true;                  flag[1] = true;
}                                }
/*critical section*/;            /* critical section*/;
flag[0] = false;                 flag[1] = false;
.
```

(d) Fourth attempt

ii. Dekker 算法

Dekker 互斥算法是由荷兰数学家 Dekker 提出的一种解决并发进程互斥与同步的软件实现方法。

两个全局共享的状态变量 flag[0]和 flag[1]，表示临界区状态及哪个进程想要占用临界区，初始值为 0。

全局共享变量 turn（值为 1 或 0）表示能进入临界区的进程序号，初始值任意，一般为 0。

iii. Peterson 算法

Peterson 算法是一个实现互斥锁的并发程序设计算法，可以控制两个进程访问一个共享的单用户资源而不发生访问冲突。Gary L. Peterson 于 1981 年提出此算法。

Peterson 算法是基于双线程互斥访问的 LockOne 与 LockTwo 算法而来。LockOne 算法使用一个 flag 布尔数组，LockTwo 使用一个 turn 的整型量，都实现了互斥，但是都存在死锁的可能。Peterson 算法把这两种算法结合起来，完美地用软件实现了双线程互斥问题。

算法使用两个控制变量 flag 与 turn. 其中 flag[n]的值为真，表示 ID 号为 n 的进程希望进入该临界区，变量 turn 保存有权访问共享资源的进程的 ID 号。

iv. 信号量机制

1965 年,荷兰学者 Dijkstra 提出的信号量(Semaphores)机制是一种卓有成效的进程同步工具。在长期且广泛的应用

中，信号量机制又得到了很大的发展，它从整型信号量经记录型信号量，进而发展为“信号量集”机制。现在，信号量机制已经被广泛地应用于单处理机和多处理机系统以及计算机网络中。

互斥与同步

信号量实现进程**互斥**：互斥信号量 mutex 初始值为 1，在临界区之前执行 P(mutex)，在临界区之后执行 V(mutex)。

信号量实现进程**同步**：同步信号量 S 初始值为 0，在“前操作”之后执行 V(S)，在“后操作”之前执行 P(S)。

生产者——消费者问题

系统中有一组生产者进程和一组消费者进程，生产者进程每次生产一个产品放入缓冲区，消费者进程每次从缓冲区中取出一个产品并使用。（注：这里的“产品”理解为某种数据）

生产者、消费者共享一个初始为空、大小为 n 的缓冲区。

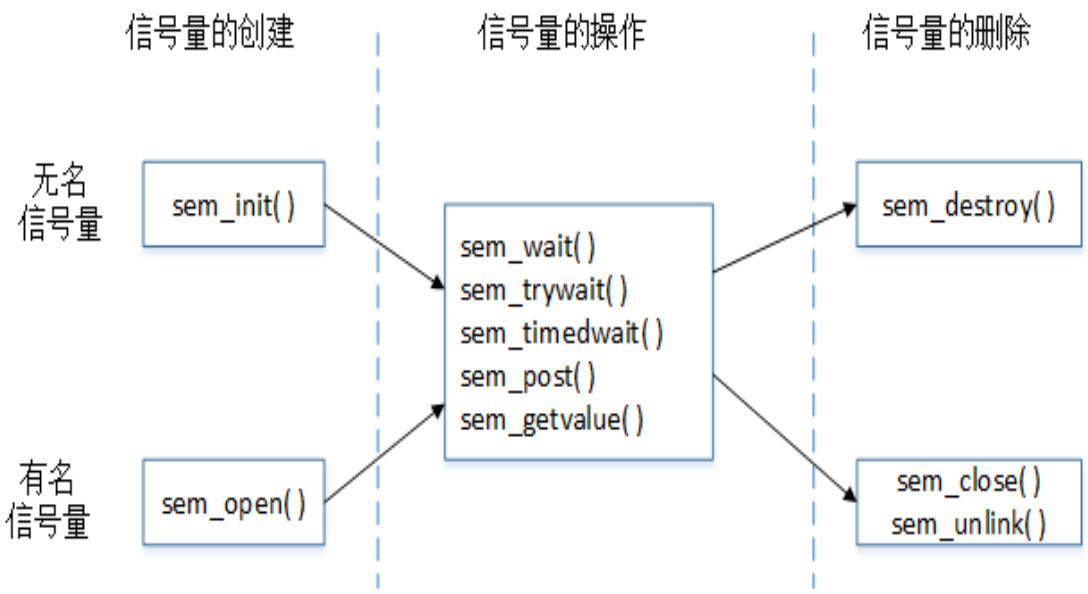
只有缓冲区没满时，生产者才能把产品放入缓冲区，否则必须等待。

只有缓冲区不空时，消费者才能从中取出产品，否则必须等待。

缓冲区是临界资源，各进程必须互斥地访问。

3. 实验说明

- **互斥锁**：保证资源独占
- **自旋锁**：与互斥量类似，但是等待自旋锁时，进程不会释放 CPU，而是一直占用 CPU。
- **条件变量**：等待和通知，一般与互斥锁合用
- **读写锁**：与互斥锁类似，不过读写锁允许更高的并行性
- **记录锁（文件锁）**：在读写锁的基础上进一步细分被锁对象的粒度
- **信号量**：条件变量的升级版
- **POSIX 信号量**
- 有名信号量：
- 基于内存的信号量
- 常用于多线程间的同步，也可用于相关进程间的同步
- 用于进行进程同步时，需要放在进程间的共享内存区中
- 无名信号量：
- 通过 IPC 名字进行进程间的同步
- 特点是把信号量值保存在文件中
- 既可用于线程，也可用于相关进程，甚至是不相关的进程
- **System V 信号量**
- 使用相对复杂
- 在内核中维护



- **使用步骤：**

① 使用 semget()函数创建或获取信号量。不同进程通过使用同一个信号量键值来获得同一个信号量。

② 使用 semctl()函数的 SETVAL 操作初始化信号量。

③ 使用 semop()函数进行信号量的 PV 操作，这是实现进程同步或互斥的核心工作。

④ 如果不需要信号量，则从系统中删除它，此时用 shmctl()函数的 IPC_RMID 操作。

三、 实验记录

1. 实施步骤

编程实现算法一，单标志法：

```
// 算法一：用 turn 序来轮流,严格轮转法
int turn = 0;
void *processA_1(void *arg)
{
    int i;
```

```
    for (i = 0; i < 5; i++)
    {
        while (turn != 0){printf("process1 is waiting\n");};
        printf("process1 is in critical section\n");
        turn = 1; // turn = 1, process2 can enter critical section
    }
}
void *processA_2(void *arg)
{
    int i;
    for (i = 0; i < 5; i++)
    {
        while (turn != 1){printf("process2 is waiting\n");};
        printf("process2 is in critical section\n");
        turn = 0; // turn = 0, process1 can enter critical section
    }
}
```

编程实现算法二，双标志先检查法：

```
// 算法二：用 flag[i]标志进程 i 进入临界区，双标志先检查法：
int flag[2] = {0, 0};
void *processB_1(void *arg)
{
    int i;
    for (i = 0; i < 5; i++)
    {
        while (flag[1] == 1){printf("process1 is waiting\n");};
        flag[0] = 1;
        printf("process1 is in critical section\n");
        flag[0] = 0;
    }
}
void *processB_2(void *arg)
{
    int i;
    for (i = 0; i < 5; i++)
    {
        while (flag[0] == 1){printf("process2 is waiting\n");};
        flag[1] = 1;
        printf("process2 is in critical section\n");
        flag[1] = 0;
    }
}
```

编程实现算法三，双标志后检查法：

```
// 算法三：将 flag[i]标志的设置提前到循环等待之前，双标志后检查法：
void *processC_1(void *arg)
{
    int i;
    for (i = 0; i < 5; i++)
    {
        flag[0] = 1;
        while (flag[1] == 1){printf("process1 is waiting\n");};
        printf("process1 is in critical section\n");
        flag[0] = 0;
    }
}
void *processC_2(void *arg)
{
    int i;
    for (i = 0; i < 5; i++)
    {
        flag[1] = 1;
        while (flag[0] == 1){printf("process2 is waiting\n");};
        printf("process2 is in critical section\n");
        flag[1] = 0;
    }
}
```

编程实现算法四，加入延时：

```
// 算法四：在循环等待中用延时给其他进程进入的机会
void *processD_1(void *arg)
{
    int i;
    for (i = 0; i < 5; i++)
    {
        flag[0] = 1;
        while (flag[1] == 1){flag[0] = 0;printf("process1 is
waiting\n");sleep(1);flag[0] = 1;};
        printf("process1 is in critical section\n");
        flag[0] = 0;
    }
}
void *processD_2(void *arg)
{
    int i;
```

```
    for (i = 0; i < 5; i++)
    {
        flag[1] = 1;
        while (flag[0] == 1){flag[1] = 0;printf("process2 is
waiting\n");sleep(1);flag[1] = 1;};
        printf("process2 is in critical section\n");
        flag[1] = 0;
    }
}
```

编程实现 Dekker 算法：

```
// 算法五：Dekker 算法
void *processE_1(void *arg)
{
    int i;
    for (i = 0; i < 5; i++)
    {
        flag[0] = 1;
        while (flag[1] == 1)
        {
            if (turn == 1)
            {
                flag[0] = 0;
                while (turn == 1){printf("process1 is waiting\n");};
                flag[0] = 1;
            }
        }
        printf("process1 is in critical section\n");
        turn = 1;
        flag[0] = 0;
    }
}
void *processE_2(void *arg)
{
    int i;
    for (i = 0; i < 5; i++)
    {
        flag[1] = 1;
        while (flag[0] == 1)
        {
            if (turn == 0)
            {
                flag[1] = 0;
                while (turn == 0){printf("process2 is waiting\n");};
```

```
                flag[1] = 1;
            }
        }
        printf("process2 is in critical section\n");
        turn = 0;
        flag[1] = 0;
    }
}
```

编程实现 Peterson 算法：

```
// 算法六：Peterson 算法
void *processF_1(void *arg)
{
    int i;
    for (i = 0; i < 5; i++)
    {
        flag[0] = 1;
        turn = 1;
        while (flag[1] == 1 && turn == 1){printf("process1 is
waiting\n");};
        printf("process1 is in critical section\n");
        flag[0] = 0;
    }
}
void *processF_2(void *arg)
{
    int i;
    for (i = 0; i < 5; i++)
    {
        flag[1] = 1;
        turn = 0;
        while (flag[0] == 1 && turn == 0){printf("process2 is
waiting\n");};
        printf("process2 is in critical section\n");
        flag[1] = 0;
    }
}
```

2. 实验记录

1. 算法一

如果两个进程设置的循环次数相同，会严格轮流访问临界区：

```
root@Iecf8ca5de00401097:/hy-tmp/Enderfga# gcc test.c -o test  -l  pthread
root@Iecf8ca5de00401097:/hy-tmp/Enderfga# ./test
process1 is in critical section
process2 is in critical section
process1 is in critical section
process2 is in critical section
process1 is in critical section
process2 is in critical section
process1 is in critical section
process2 is in critical section
process1 is in critical section
process2 is in critical section
root@Iecf8ca5de00401097:/hy-tmp/Enderfga#
```

如果两个进程的循环次数不同，则会出现一直等待的情况。

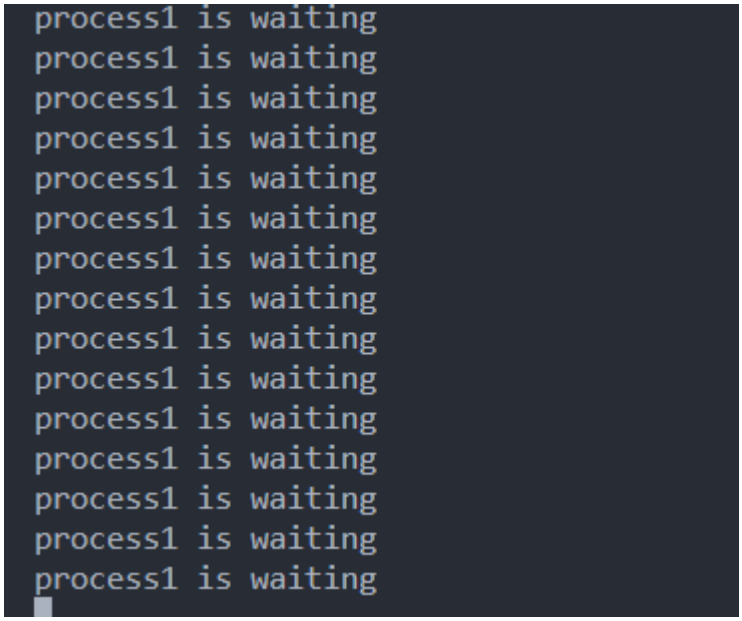
```
process1 is waiting
process1 is waiting
process1 is waiting
process1 is waiting
process1 is waiting
process1 is waiting
process1 is waiting
process1 is waiting
process1 is waiting
process1 is waiting
process1 is waiting
process1 is waiting
process1 is waiting
process1 is waiting
process1 is waiting
^C
root@Iecf8ca5de00401097:/hy-tmp/Enderfga#
```

2. 算法二

```
root@Iecf8ca5de00401097:/hy-tmp/Enderfga# ./test
process1 is in critical section
process1 is in critical section
process1 is in critical section
process2 is waiting
process2 is waiting
process2 is waiting
process2 is waiting
process2 is waiting
process2 is waiting
process2 is waiting
process1 is in critical section
process1 is in critical section
process2 is waiting
process2 is in critical section
process2 is in critical section
process2 is in critical section
process2 is in critical section
process2 is in critical section
root@Iecf8ca5de00401097:/hy-tmp/Enderfga#
```

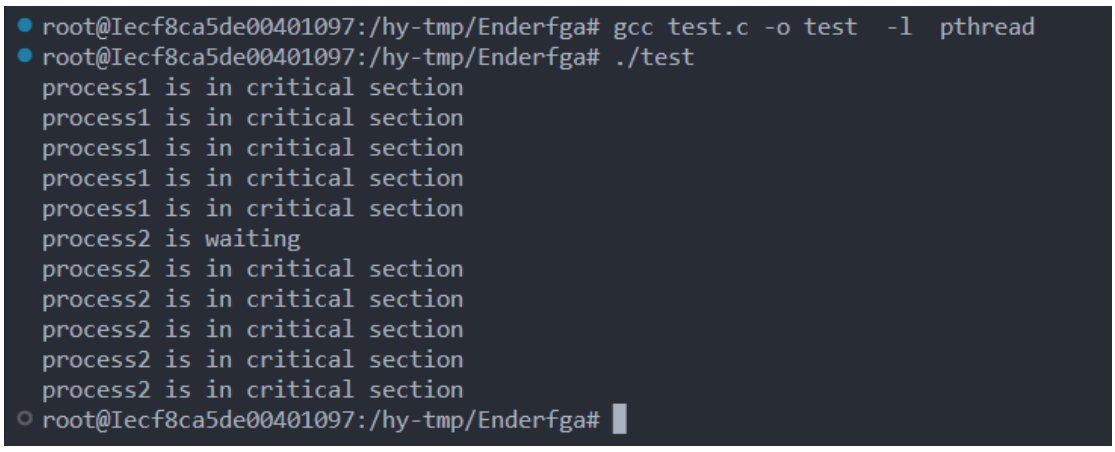
可以看出两个进程轮流完成了任务，不会出现上一种算法的情况。但是进入区的“检查”和“上锁”两个处理不是一气呵成的。“检查”后，“上锁”前可能发生进程切换。

3. **算法三**



经过多次运行，大部分都进入了无限等待的死循环，各进程都长期无法访问临界资源而产生“饥饿”现象。

4. **算法四**



加入延时后算法三导致的问题得到了很好的改善。

5. **Dekker**

```
root@Iecf8ca5de00401097:/hy-tmp/Enderfga# gcc test.c -o test  -l  pthread
root@Iecf8ca5de00401097:/hy-tmp/Enderfga# ./test
process1 is in critical section
process1 is waiting
process1 is waiting
process1 is waiting
process1 is waiting
process1 is waiting
process1 is waiting
process1 is waiting
process1 is waiting
process1 is waiting
process1 is waiting
process1 is waiting
process1 is waiting
process2 is in critical section
process2 is in critical section
process2 is in critical section
process2 is in critical section
process2 is in critical section
process1 is waiting
process1 is in critical section
process1 is in critical section
process1 is in critical section
process1 is in critical section
root@Iecf8ca5de00401097:/hy-tmp/Enderfga#
```

6. Peterson

```
root@Iecf8ca5de00401097:/hy-tmp/Enderfga# gcc test.c -o test  -l  pthread
root@Iecf8ca5de00401097:/hy-tmp/Enderfga# ./test
process1 is in critical section
process1 is waiting
process1 is waiting
process1 is waiting
process1 is waiting
process1 is waiting
process1 is waiting
process1 is waiting
process1 is waiting
process1 is waiting
process1 is waiting
process1 is waiting
process2 is waiting
process2 is in critical section
process2 is waiting
process2 is waiting
process2 is waiting
process2 is waiting
process2 is waiting
process2 is waiting
process2 is waiting
process1 is waiting
process1 is in critical section
process2 is waiting
process2 is in critical section
process2 is waiting
process2 is waiting
process2 is waiting
process2 is waiting
process1 is waiting
process1 is in critical section
process2 is waiting
process2 is in critical section
process2 is waiting
process2 is waiting
process2 is waiting
process1 is waiting
process1 is in critical section
process2 is waiting
process2 is in critical section
process1 is waiting
process1 is in critical section
process2 is waiting
process2 is in critical section
root@Iecf8ca5de00401097:/hy-tmp/Enderfga#
```

对比上述两个算法，显然 Peterson 出现了更多的 waiting，会发生“忙等”。

7. 信号量机制模拟实现生产者-消费者问题

```
root@Iecf8ca5de00401097:/hy-tmp/Enderfga# ./test
input something to buffer:The first product.
read product from buffer:The first product.
The End...
root@Iecf8ca5de00401097:/hy-tmp/Enderfga#
```

3. 实验结果

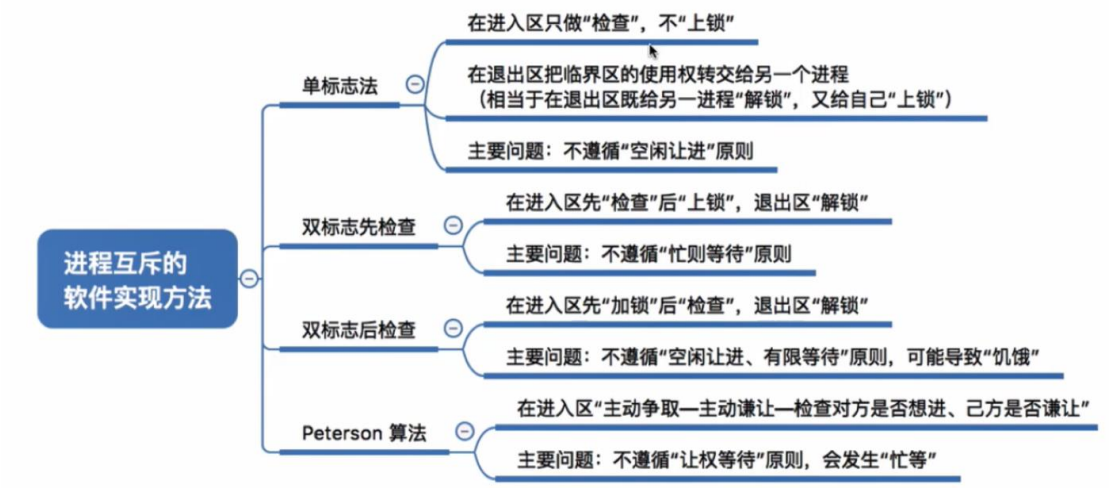
为了实现对临界资源的互斥访问，同时保证系统整体性能遵循以下原则：

1. 空闲让进：临界区空闲时可以允许一个请求进入临界区的进程，立即进入临界区

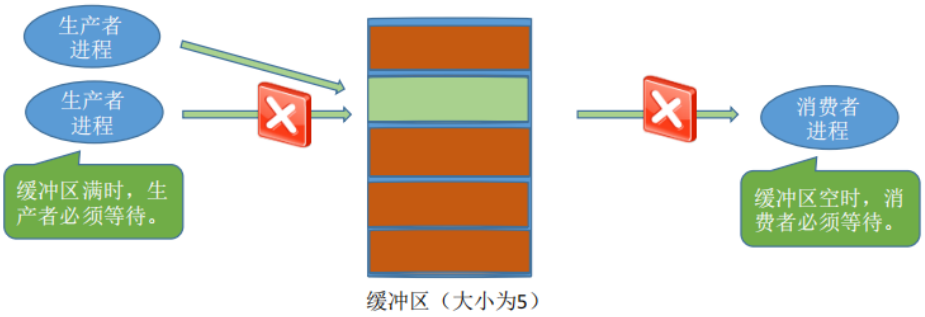
2. 忙则等待：当已有进程，进入临界区时，其他试图进入临界区的进程必须等待

3. 有限等待：对请求访问的进程，应保证能在有限时间内进入临界区，不会饥饿

4. 让权等待：当进程不能进入临界区时，应立即释放处理器资源，防止进程忙等



系统中有一组生产者进程和一组消费者进程，生产者进程每次生产一个产品放入缓冲区，消费者进程每次从缓冲区中取出一个产品并使用。（注：这里的“产品”理解为某种数据）
生产者、消费者共享一个初始为空、大小为n的缓冲区。
只有缓冲区没满时，生产者才能把产品放入缓冲区，否则必须等待。
只有缓冲区不空时，消费者才能从中取出产品，否则必须等待。
缓冲区是临界资源，各进程必须互斥地访问。

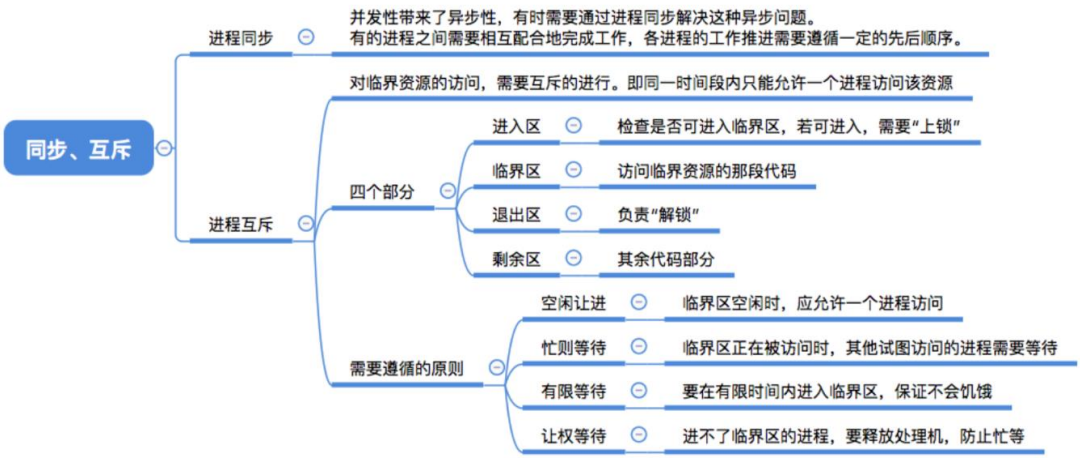


该问题中出现的主要的两种关系：

①生产者—消费者之间的同步关系表现为：一旦缓冲池中所有缓冲区均装满产品时，生产者必须等待消费者提供空缓冲区；一旦缓冲池中所有缓冲区全为空时，消费者必须等待生产者提供满缓冲区。

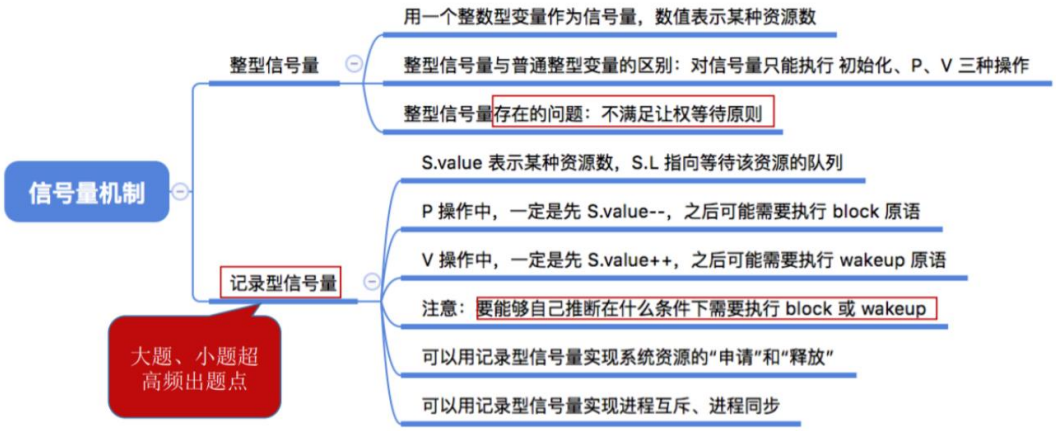
②生产者—消费者之间还有互斥关系：由于缓冲池是临界资源，所以任何进程在对缓冲区进行存取操作时都必须和其他进程互斥进行。

四、　总结与讨论



进程间的协作关系

- **互斥。是指多个进程不允许同时使用同一资源**。当某个进程使用某种资源的时候，其他进程必须等待。所以资源不能被多个进程同时使用。在这种情况下，程序的执行与其他进程无关。
- **同步。是指多个进程中发生的事件存在某种先后顺序**。即某些进程的执行必须先于另一些进程（我们之前画的前驱图）。其任务是使得并发执行的诸多进程有效的共享资源和相互合作，从而使程序的执行具有可再现性。这种情况下，进程间接知道对方。
- **通信。是指多个进程间要传递一定的信息**。这个时候进程直接得知对方。



而信号量机制，是更强大的同步和互斥机制，解决在双标志先检查法，进入区检查和上锁操作无法一气呵成，从而导致两个进程有可能进入临界区的问题；并实现了原先所有算法都无法实现的让权等待。

五、附：程序模块的源代码

```
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#define MAX 256
```

```
char *buffer;
sem_t empty;    //定义同步信号量 empty
sem_t full;    //定义同步信号量 full
sem_t mutex;   //定义互斥信号量 mutex

void * producer(void* args)  //生产者
{
  sem_wait(&empty);    //empty 的 P 操作
  sem_wait(&mutex);    //mutex 的 P 操作
  printf("input something to buffer:");
  buffer=(char *)malloc(MAX); //给缓冲区分配内存空间
  fgets(buffer,MAX,stdin);    //输入产品至缓冲区
  sem_post(&mutex);    //mutex 的 V 操作
  sem_post(&full);     //full 的 V 操作
}
void * consumer(void * args) //消费者
{
  sem_wait(&full);     //full 的 P 操作
  sem_wait(&mutex);    //mutex 的 P 操作
  printf("read product from buffer:%s",buffer); //从缓冲区中取出产品
  memset(buffer,0,MAX);                 //清空缓冲区
  sem_post(&mutex);    //mutex 的 V 操作
  sem_post(&empty);    //empty 的 V 操作
}
int main()
{
  pthread_t id_producer;
  pthread_t id_consumer;
  int ret;


  sem_init(&empty,0,10);       //设置 empty 到初值为 10
  sem_init(&full,0,0);         //设置 full 到初值为 0
  sem_init(&mutex,0,1);                //设置 mutex 到初值为 1
  //创建生产者线程
  ret=pthread_create(&id_producer,NULL,producer,NULL);
  //创建消费者线程
  ret=pthread_create(&id_consumer,NULL,consumer,NULL);
  pthread_join(id_producer,NULL);     //等待生产者线程结束
  pthread_join(id_consumer,NULL);//等待消费者线程结束
  sem_destroy(&empty);         //删除信号量
  sem_destroy(&full);
  sem_destroy(&mutex);
  printf("The End...\n");}
```