

系列课程 —Linux系统编程

## 第五章

# 并发与竞态

讲师：任继梅

QQ：59189174

# 课前提问

1. 如果两台电脑同时向同一台打印机送出打印内容，打印内容是各自期望的吗？如何解决？
2. 客车上只有20个座位，A队有30人从前门上车，B队有30人从后门上车，问座位排满时，A队多少人？B队多少人？
3. 一条生产线上，两个相邻工位，前一个工位工作没完成时，后一个工位处于什么状态？
4. 两个进程打开同一个文件，同时对文件进行操作，会出现什么问题？

# 本章目标

✓ 文件共享



✓ 同步互斥



✓ 临界资源



✓ 死锁饥饿



✓ 同步机制



✓ 典型问题



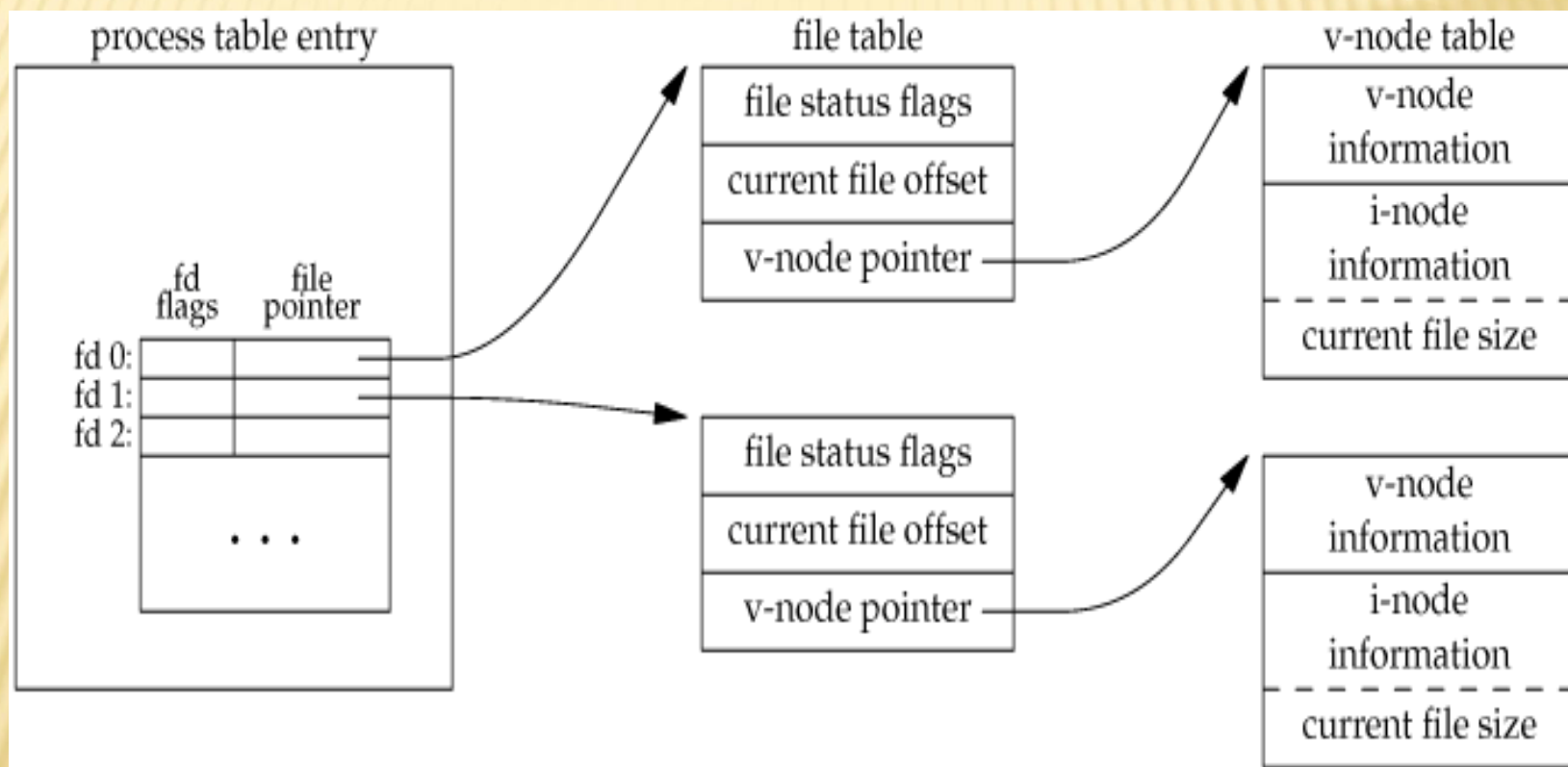


# **第一节**

## **文件共享**

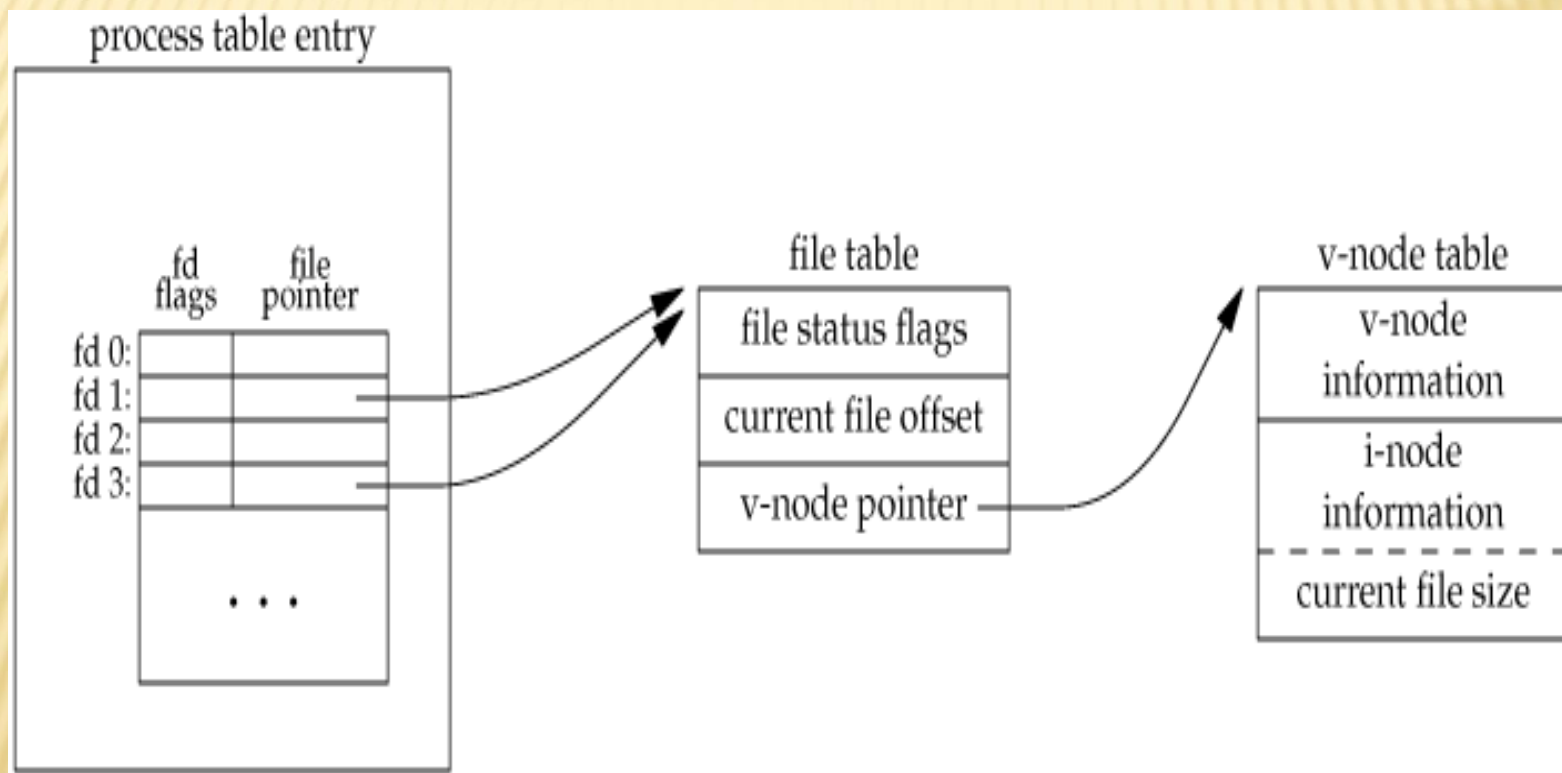
# 文件共享

## ● 文件操作内核实现结构



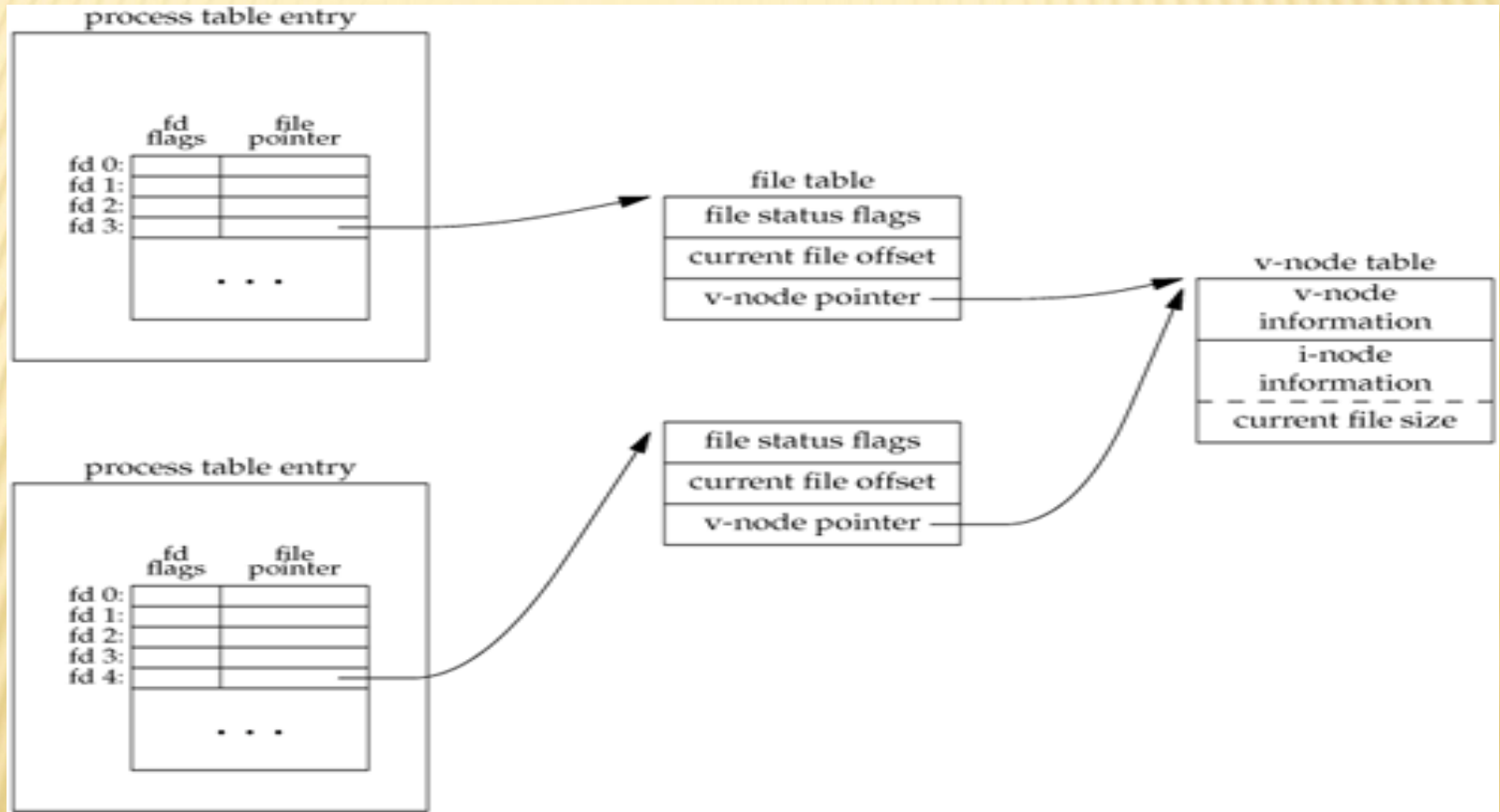
# 文件共享

## ● dup和fork时的文件描述符操作



# 文件共享

● 两个独立进程各自打开同一文件





# 文件共享

## ● 创建锁文件

- ◆ 以“原子操作”创建一个锁文件
- ◆ 原子操作：无法被打断的操作
- ◆ 锁文件充当一个指示器的角色
- ◆ Open文件时选择**O\_CREAT&O\_EXCL**
- ◆ open原子操作完成先判断文件是否存在，如何不存在则创建它
- ◆ **/tmp/LCK.name**
- ◆ **EEXIST**错误号
- ◆ 建议性锁非强制



# 文件共享

## ● 文件区域锁

- ◆ 文件段锁定、文件区域锁定、字节范围锁定
- ◆ `int fcntl(int fildes , int command, ...)`
- ◆ 对文件描述符操作，根据`command`设置不同的任务
- ◆ 用于文件区域锁的三个`command`:
  - ◆ `F_GETLK`: 获取文件描述符的锁信息
  - ◆ `F_SETLK`: 对文件某个区域加锁或解锁
  - ◆ `F_SETLKW`: 作用同`F_SETLK`，但无法获取锁时一直阻塞到能锁
- ◆ 实际原型:
  - ◆ `int fcntl(int fildes , int cmd, struct flock * flock_structure)`
- ◆ 程序对某个文件拥有的所有锁都将在文件描述符被关闭时自动清除
- ◆ 程序在结束时也会清除各种锁
- ◆ 只能使用`read`和`write`系统调用配合使用，不能使用`fread`和`fwrite`

## ● 文件区域锁——struct flock说明

- ◆ 至少包含以下成员：
- ◆ short l\_type：见下表
- ◆ short l\_whence：SEEK\_SET、SEEK\_CUR、SEEK\_END
- ◆ off\_t l\_start：区域第一字节相对于l\_whence的相对位置
- ◆ off\_t l\_len：区域字节数
- ◆ pid\_t l\_pid：持有锁的进程ID，一般配合F\_GETLK使用

l_type的取值	
F_RDLCK	读锁，共享锁，多个进程可以同时持有同一区域的共享锁，只要有读权限。一个区域只要一把共享锁，就无法设置独占锁。
F_WRLCK	写锁，独占锁，同一区域只能有一个进程（具有写权限）设置独占锁
F_UNLCK	解锁，用来清除锁

# 文件共享

## ● 文件区域锁——F\_GETLK操作说明

- ◆ 获取指定文件指定区域相关锁的信息
- ◆ `int fcntl(int fildes , F_GETLK, struct flock * flock_structure)`
- ◆ 成功返回非-1值，失败返回-1，锁信息可以通过flock\_structure参数内容来检查（比如l\_pid）

### ◆ 代码范例：

```
struct flock region_to_test;  
region_to_test.l_type = F_WRLCK; region_to_test.l_whence =  
SEEK_SET;  
region_to_test.l_start = 0; region_to_test.l_len = 10;  
region_to_test.l_pid=-1;  
fcntl(fd, F_GETLK,&region_to_test);  
if(region_to_test.l_pid !=-1 ){  
    printf("This region is locked by %d\n",  
region_to_test.l_pid );  
}
```



## ● 文件区域锁——F\_SETLK、F\_SETLKW操作说明

- ◆ 对指定文件的某个指定区域设置指定锁
- ◆ F\_SETLK设置失败将直接返回-1， F\_SETLKW设置失败将阻塞
- ◆ l\_pid无作用

### ◆ 代码范例：

```
struct flock region_read;
region_read.l_type = F_RDLCK; region_read.l_whence = SEEK_SET;
region_read.l_start = 20; region_read.l_len = 20;
ret = fcntl(fd, F_SETLK,&region_read);
if(-1 == ret )
{
    printf("Lock region failed\n");
}
```

# 文件共享

## ● 文件区域锁示例

◆ lockfileregion.c testlockfileregion.c

◆ wraplock.c wraplock.h

## **第二节**





### **同步互斥**






# 同步互斥

## 并发与竞态

### 并发执行：Concurrent

-  多个单元同时、并行被执行
-  这样单元可能是进程或线程
-  并发执行单元对共享资源的访问很容易导致数据错乱
-  解决这样的错乱问题是所有操作系统核心问题之一

### 竞态条件（竞态情况）：Race Condition

-  并发产生竞态，竞态导致共享数据的错乱
-  一种极端低可能性的事件，因此程序员往往会忽视竞态。但是在计算机世界中，百万分之一的事件可能没几秒就会发生，而其结果是灾难性的。
-  竞态是由于对资源的共享访问而产生的

# 同步互斥

## 解决竞态的原则

- ❑ 只要可能，就应该避免资源的共享。若没有并发访问，就不会有竞态。
- ❑ 必须显式地管理对共享资源的访问。常见管理技术有：锁定、互斥、原子操作，确保一次只有一个任务可操作共享资源。

# 同步互斥

## 同步

- ❖ 任务之间直接的制约关系，是为完成某种功能而建立的两个或多个任务，这个任务需要在某些位置上协调他们的工作次序而等待、传递信息所产生的制约关系。
- ❖ 任务间是合作关系，而不是竞争关系，存在着依赖。
- ❖ 工厂生产线
- ❖ 比如说进程A需要从缓冲区读取进程B产生的信息，当缓冲区为空时，进程A因为读取不到信息而被阻塞。而当进程B产生信息放入缓冲区时，进程A才会被唤醒。



# 同步互斥

## 互斥

- ❑ 任务之间的间接制约关系。当一个任务使用共享资源时，另一个任务必须等待。只有当其余任务不在使用共享资源后，这个任务才会解除阻塞状态。
- ❑ 任务间竞争关系，而不是合作关系，存在着争抢。
- ❑ 汽车座位
- ❑ 比如进程B需要访问打印机，但此时进程A占有了打印机，进程B会被阻塞，直到进程A释放了打印机资源,进程B才可以继续执行。

# 临界资源

## 临界资源

- ❑ 但对于某些资源来说，其在同一时间只能被一个任务所占用。这些一次只能被一个进程所占用的资源就是所谓的临界资源或称为共享资源。
- ❑ 典型的临界资源比如物理上的打印机，或是存在硬盘或内存中被多个任务所共享的一些变量和数据等
- ❑ 临界资源必须显式地加以保护管理，否则很有可能造成数据的错乱问题。
- ❑ 对于临界资源的访问，必须是互斥地进行。也就是当临界资源被占用时，另一个申请临界资源的任务会被阻塞，直到其所申请的临界资源被释放。

# 临界资源

## 人 临界区

- ❑ 任务内访问临界资源的代码被成为临界区。
- ❑ 显式处理临界资源的一般步骤：
  - ❑ 1. 进入区:查看临界区是否可访问，如果可以访问，则转到步骤二，否则任务会被阻塞直到临界区可访问。
  - ❑ 2.临界区：在临界区操作共享资源。
  - ❑ 3.退出区:清除临界区被占用的标志。
  - ❑ 4.剩余区：进程与临界区不相关部分的代码。



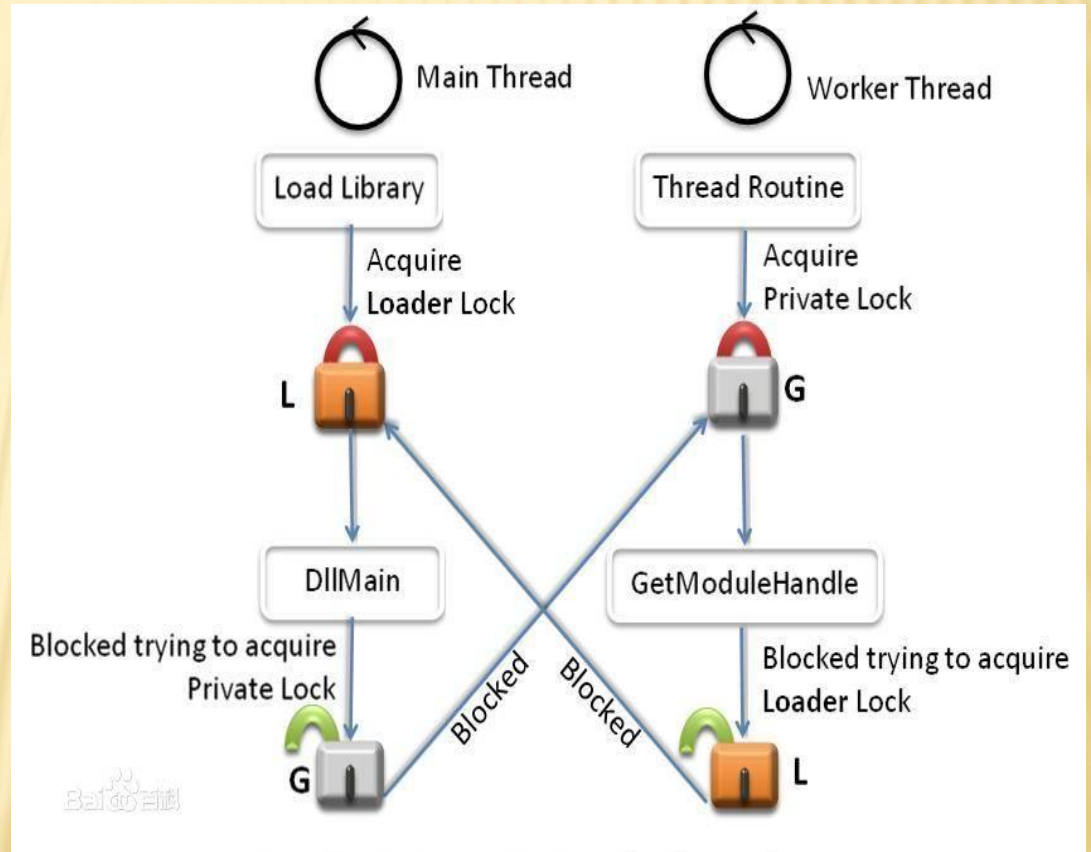
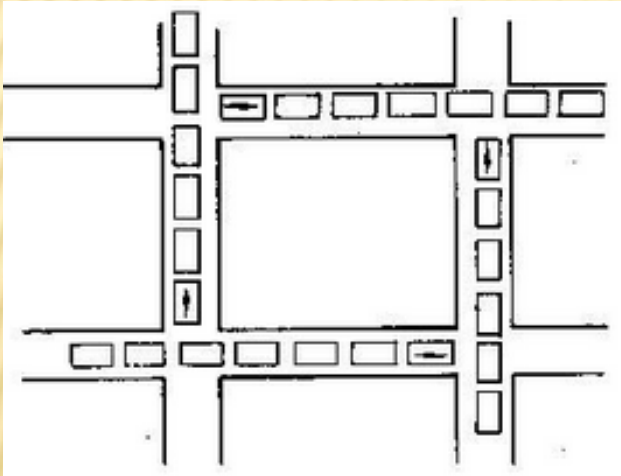
# 死锁饥饿

## 死锁

- ❖ 每一个任务都在等待只能由其他任务才能引发的事件，那么这些是死锁的
- ❖ 对临界资源操作不当，导致任务永远睡眠
- ❖ 有时又称为抱死
- ❖ 现实例子：交通死锁
- ❖ 如果任务A锁住了资源1并等待资源2，而任务B锁住了资源2并等待资源1，这样两个任务就发生了死锁现象
- ❖ 如果某个任务在其临界区代码里试图再次获取同样的共享资源
- ❖ 如果任务A锁住了资源，临界区后没有释放锁，任务B永远没法获得共享资源，则任务B死锁，任务A在其后续代码里又试图获取共享资源，则任务A死锁

# 死锁饥饿

## 死锁



# 死锁饥饿

## 人 活锁

- 对临界资源操作不当，导致任务永远忙等待
- 现象与死锁类似，但是任务不是睡眠，而是以占用时间片的形式忙等待



# 死锁饥饿

## 饥饿

- 指一个可运行的进程尽管能继续执行，但被调度器无限期地忽视，而不能被调度执行的情况。
- 现象描述：如果任务T1封锁了数据R,任务T2又请求封锁R，于是T2等待。T3也请求封锁R，当T1释放了R上的封锁后，系统首先批准了T3的请求，T2仍然等待。然后T4又请求封锁R，当T3释放了R上的封锁之后，系统又批准了T4的请求.....T2可能永远等待
- 有一定几率解开
- 解决方法：先来先服务

# 同步机制

## 人 现实实例

- ❑ 以一个停车场的运作为例。简单起见，假设停车场只有三个车位，一开始三个车位都是空的。
- ❑ 这时如果同时来了五辆车，看门人允许其中三辆直接进入，然后放下车拦，剩下的车则必须在入口等待，此后来的车也都不得不在入口处等待。
- ❑ 这时，有一辆车离开停车场，看门人得知后，打开车拦，放入外面的一辆进去，如果又离开两辆，则又可以放入两辆，如此往复
- ❑ 车位是公共资源（临界资源或共享资源）
- ❑ 每辆车好比一个任务（一个进程或线程或异常处理程序）
- ❑ 看门人起的就是同步机制的作用（锁文件、区域锁、信号量）
- ❑ 解决方法：先来先服务

# 同步机制

## 同步机制原则

- 解决竞态条件问题的方法——同步机制

- 应遵循的原则：

- 空闲让进

- 忙则等待

- 有限等待：避免死等

- 等待方式：

- 让权等待：让出处理器，睡眠

- 忙等待：轮询等待



# 同步机制

## ● 常用同步机制

● 进程：信号量

● 线程：互斥锁

# 同步机制

## 人 信号量

- System V 信号量

- POSIX 信号量

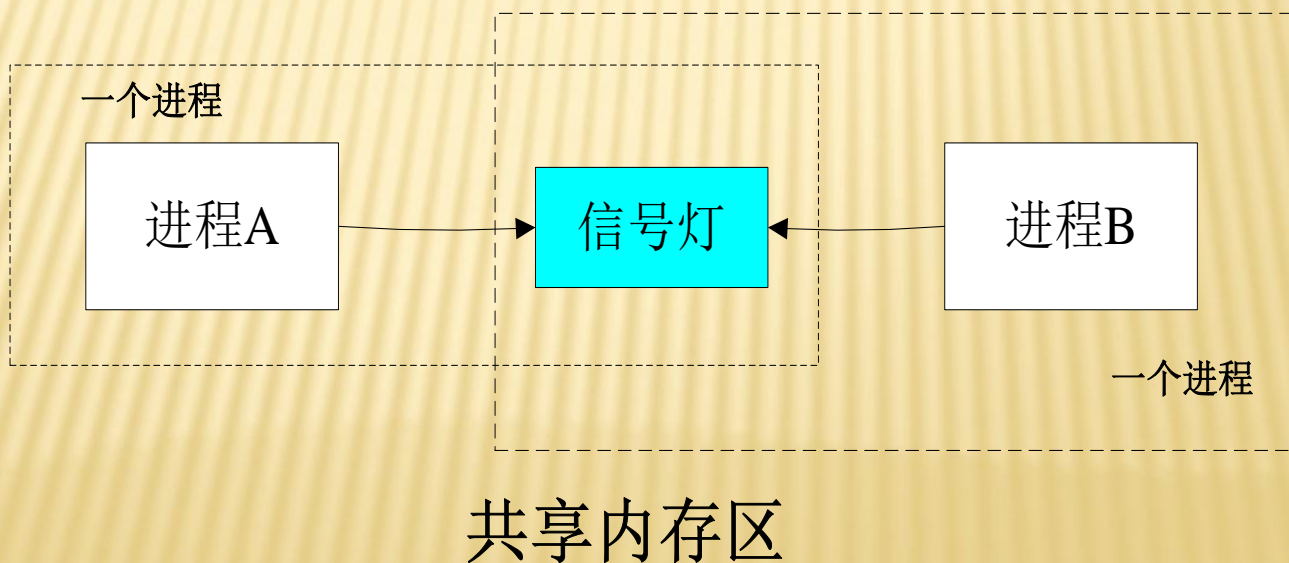
  - 有名信号量

  - 无名信号量

# 同步机制

## System V 信号量

- 常用操作：
- 1. semget() 创建和打开
- 2. semop() 改变信号量的值
- 3. semctl() 控制信号量





# 同步机制

## ftok() 创建关键字

```
#include <sys/types.h>
#include <sys/ipc.h>

key_t ftok(const char *pathname, int proj_id);
```

- @pathname 指定路径名（字符串）
- @proj\_id 指定id（取id低8位）

# 同步机制

## semget() 创建和打开

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

- ❖ @key 信号量标示符
- ❖ @nsems 信号灯集中包含的信号灯数目。
- ❖ @semflg 信号灯集的访问权限，通常为 `IPC_CREAT | 0666`
  - ❖ `IPC_CREAT | IPC_EXCL` 则可以创建一个新的，唯一的信号量，如果信号量已存在，返回一个错误。
- ❖ 成功返回信号集 id，失败返回 -1

# 同步机制

## semop()改变信号量的值

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, unsigned nsops);

int semtimedop(int semid, struct sembuf *sops, unsigned nsops,
               struct timespec *timeout);
```

- ④ @sem\_id是由semget返回的信号量标识符
- ④ @nsops 指信号集中信号量的个数
- ④ @sembuf结构的定义如下:

```
struct sembuf
{
    short sem_num; //除非使用一组信号量，否则它为0
    short sem_op;  //信号量在一次操作中需要改变的数据，通常是：
                  //一个是-1，即P（等待）操作，
                  //一个是+1，即V（发送信号）操作。
    short sem_flg;
    //通常为SEM_UNDO，使操作系统跟踪信号量，并在进程没有释放该信号量而终止时，操作系统释放信号量
};
```



# 同步机制

## semctl()控制信号量

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

```
int semctl(int semid, int semnum, int cmd, ...);
```

- ❑ @sem\_id是由semget返回的信号量标识符
- ❑ @semnum要修改的信号灯编号
- ❑ 如果有第四个参数，它通常是一个union semum结构
- ❑ @command:
  - ❑ SETVAL: 用来把信号量初始化为一个已知的值, 由第四个参数指定
  - ❑ GETVAL: 返回信号量的
  - ❑ IPC\_RMID: 用于删除一个已经无需继续使用的信号量标识符。

# 同步机制

## 📄 System V 信号量 例子:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>

int main()
{
    int semid;
    int semval;

    key_t key = 1234;
    semid = semget(key, 1, IPC_CREAT | 0644);

    //初始化信号量
    semctl(semid, 0, SETVAL, 10);

    //获取信号量的值
    semval = semctl(semid, 0, GETVAL);
    printf("1 : semval = %d\n", semval);
}
```

# 同步机制

## 📄 System V 信号量 例子:

```
//操作信号量
struct sembuf buf;
buf.sem_num = 0;
buf.sem_op = -11;
buf.sem_flg = 0;
semop(semid, &buf, 1); //如果buf.sem_op这个值为负数，则阻塞等待

//获取信号量的值
semval = semctl(semid, 0, GETVAL);
printf("2 : semval = %d\n", semval);

semctl(semid, 0, IPC_RMID);

return 0;
}
```



## **第三节**

### **典型问题分析**

# 典型问题

## 生产者-消费者问题（有限缓冲问题）

- ❑ 有一群生产者进程在生产产品，并将这些产品提供给消费者进程去消费。
- ❑ 为使生产者进程与消费者进程能并发执行，在两者之间设置了一个具有  $n$  个缓冲区的缓冲池
- ❑ 生产者进程将它所生产的产品放入一个缓冲区中；消费者进程可从一个缓冲区中取走产品去消费。
- ❑ 不允许消费者进程到一个空缓冲区去取产品，也不允许生产者进程向一个已装满产品且尚未被取走的缓冲区中投放产品

# 典型问题

## 生产者消费者解决方案

```
semaphore mutex = 1;
semaphore fillCount = 0;
semaphore emptyCount = BUFFER_SIZE;

procedure producer() {
    while (true) {
        item = produceItem();
        down(emptyCount);
        down(mutex);
        putItemIntoBuffer(item);
        up(mutex);
        up(fillCount);
    }
}

procedure consumer() {
    while (true) {
        down(fillCount);
        down(mutex);
        item = removeItemFromBuffer();
        up(mutex);
        up(emptyCount);
        consumeItem(item);
    }
}
```



# 典型问题

## 读者-写者问题

- ❑ 一个数据文件或记录，可被多个进程共享，我们把只要求读该文件的进程称为“Reader进程”，其他进程则称为“Writer 进程”。
- ❑ 允许多个进程同时读一个共享对象，因为读操作不会使数据文件混乱。
- ❑ 但不允许一个 Writer 进程和其他 Reader 进程或 Writer 进程同时访问共享对象，因为这种访问将会引起混乱。

# 典型问题

## 👤 读者-写者问题解决方案

### ➤ 读进程

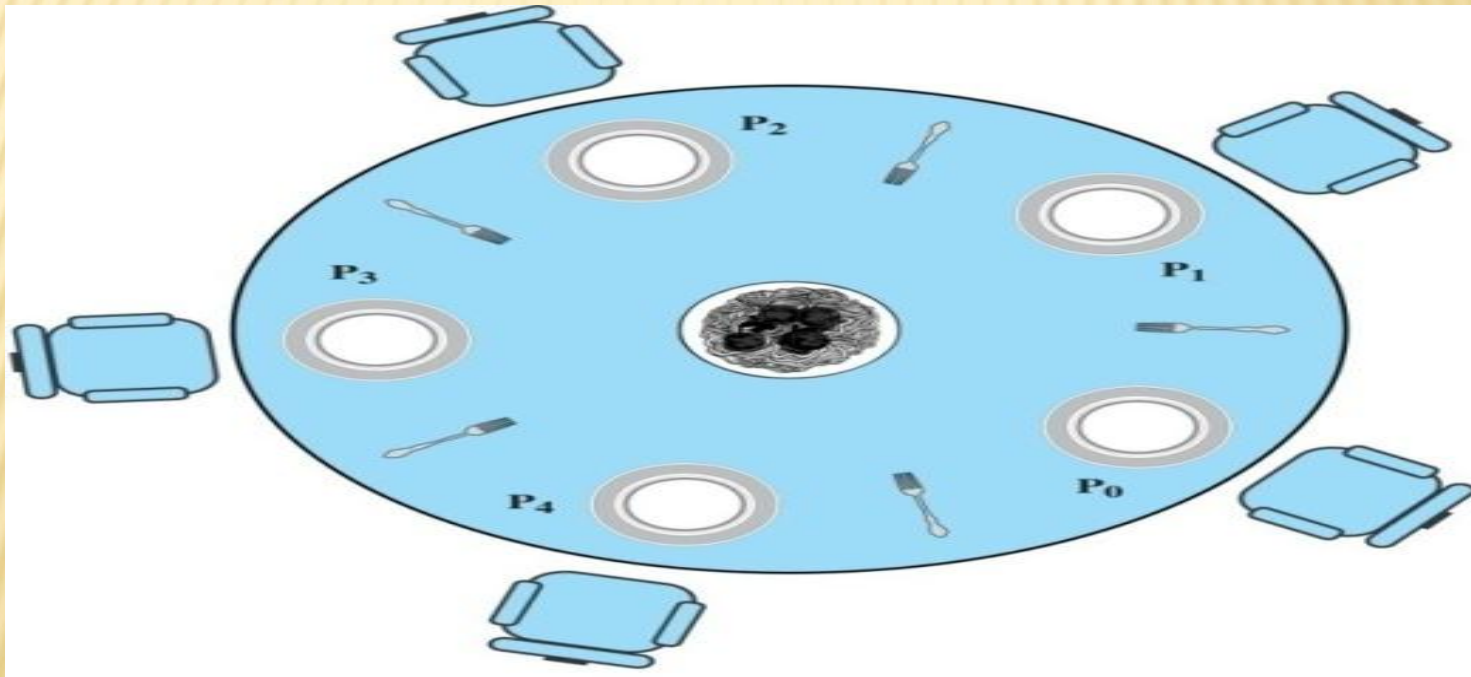
```
wait(rmutex);  
if(0==readcount)  
wait(wmutex);  
readcount++;  
up(rmutex);  
reading;  
wait(rmutex);  
readcount--;  
if(0==readcount) up(wmutex);  
up(rmutex);
```

### ➤ 写进程

```
wait(wmutex);  
writing;  
up(wmutex);
```

# 典型问题

## 哲学家就餐问题





# 典型问题

## 哲学家就餐问题解决方案

◆初步解决方案:

◆mutex chopstick[5] 全部初始化为1则:

```
wait(chopstick[i]);
```

```
wait(chopstick[(i+1)%5]);
```

```
eat();
```

```
signal(chopstick[i]);
```

```
signal(chopstick[(i+1)%5]);
```

```
think();
```

◆死锁可能?

◆解决方法: 至多只允许有四位哲学家同时去拿左边的筷子

# 典型问题

## 哲学家就餐问题解决方案

```
/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int I)
{
    while (true)
    {
        think();
        wait (room);
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal (fork [(i+1) mod 5]);
        signal (fork[i]);
        signal (room);
    }
}

void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
              philosopher (3), philosopher (4));
}
```

**Figure 6.13 A Second Solution to the Dining Philosophers Problem**

# 课程总结

## 本节课程内容

- ▣ 临界同步
- ▣ 互斥资源
- ▣ 死锁饥饿
- ▣ 典型实例

## 下节课程

- ▣ 线程概念
- ▣ 线程创建
- ▣ 线程终止
- ▣ 线程属性
- ▣ 线程同步



# 联系方式

---

QQ: 59189174

E-mail: [yumeifly@sohu.com](mailto:yumeifly@sohu.com)