进 程 间 协 作

信号灯与共享内存(IPC)

信号灯

UNIX的IPC（Inter-Process Communication，进程间通信）三样：信号灯、共享内存、消息队列 （SEM,SHM,MSG）

- **信号灯(semaphore)**
  - 控制多进程对共享资源的互斥性访问和进程间同步
- **策略和机制分离**
  - UNIX仅提供信号灯机制，访问共享资源的进程自身必须正确使用才能保证正确的互斥和同步
  - 不正确的使用，会导致信息访问的不安全和死锁
- **P操作和V操作**
  - 信号灯机制实现了P操作和V操作，而且比简单PV操作功能更强

信号灯的创建

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(int key, int nsems, int flags);
```

创建一个新的或获取一个已存在的信号灯组

nsems：该信号灯组中包含有多少个信号灯

函数返回一个整数，信号灯组的ID号；如果返回-1，表明调用失败

flags：创建或者获取

信号灯的删除

int semctl(int *sem_id*, int *snum*, int *cmd*, char **arg*);

对信号灯的控制操作，如：删除，查询状态

snum：信号灯在信号灯组中的编号

cmd：控制命令

arg：执行这一控制命令所需要的参数存放区

返回值为-1，标志操作失败；否则，表示执行成功，

对返回值的解释依赖于*cmd*

删除系统中信号灯组的调用

```
semctl(sem_id, 0, IPC_RMID, 0);
```

信号灯操作

int semop(int *sem_id*, struct sembuf **ops*, int *nops*);

信号灯操作（可能会导致调用进程在此睡眠）

ops：有*nops*个元素的sembuf结构体数组，每个元素描述对某一信号灯操作

返回值：-1，标志操作失败；否则，表示执行成功

```
struct sembuf {
    short sem_num; // 信号灯在信号灯组中的编号0,1,2…
    short sem_op;    // 信号灯操作
    short sem_flg;    // 操作选项
};
```

当sem_op<0时，P操作；当sem_op>0时，V操作

当sem_op=0时，不修改信号灯的值，等待直到变为非负数

原子性：一次semop()调用指定的多个信号灯的操作，Linux内核要么把多个操作一下全部做完，要么什么都不做

共享内存

特点

- 多个进程共同使用同一段物理内存空间
- 使用共享内存在多进程间传送数据，速度快，但进程必须自行解决对共享内存访问的互斥和同步问题（例如：使用信号灯通过P/V操作）

应用举例

- 数据交换：多进程使用共享内存交换数据（最快的进程间通信方式）
- 运行监视：协议处理程序把有限状态机状态和统计信息放入共享内存中
  - 协议处理程序运行过程中可随时启动监视程序，从共享内存中读取数据以窥视当前的状态，了解通信状况
  - 监视程序的启动与终止不影响通信进程，而且这种机制不影响协议处理程序的效率

共享内存操作

int shmget(int *key*, int *nbytes*, int *flags*);

创建一个新的或获取一个已存在的共享内存段

函数返回的整数是共享内存段的ID号；返回-1，表明调用失败

void *shmat(int *shm_id*, void **shmaddr*, int *shmflg*);

获取指向共享内存段的指针(进程逻辑地址)，返回-1：操作失败

int shmctl(int *shm_id*, int *cmd*, char **arg*) ;

对共享内存段的控制操作，如：删除，查询状态

cmd：控制命令

arg：执行这一控制命令所需要的参数存放区

“生产者-消费者”问题

- 问题：生产者/消费者用N个缓冲区构成的环形队列交换数据
- 程序设计：使用共享内存和信号量
  - ctl create 创建共享内存段和信号灯
  - ctl remove 删除所创建的共享内存段和信号灯
  - producer 启动1个生产者进程(可以同时启动多个)
  - consumer 启动1个消费者进程(可以同时启动多个)
- 源程序文件有四个
  - ctl.h 公用头文件
  - ctl.c 控制程序，创建/删除所需要的IPC机制
  - producer-consumer.c 生产者和消费者程序（二合一）
- 系统命令ipcs

内存映射文件

内存映射文件I/O

传统的访问磁盘文件的模式

- 打开一个文件，然后通过read和write访问文件

“内存映射”(Memory Map)方式读写文件

- 现代的Linux和Windows都提供了“内存映射”(Memory Map)方式读写文件的方法。
- 将文件中的一部分连续的区域映射成一段进程虚拟地址空间中的内存
  - 进程获取这段映射内存的指针后，就把这个指针当作普通的数据指针一样引用。修改其中的数据，实际修改了文件；引用其中的数据值，就是读取了文件
  - 访问文件跟内存中的数据访问一样
  - 系统不会为数据文件的内存映射区域分配相同大小的物理内存，而是由页面调度算法自动进行物理内存分配
  - 根据虚拟内存的页面调度算法，按需调入数据文件中的内容，必要时淘汰（可能需要写入)内存页面

内存映射文件I/O的优点

- **比使用read，write方式速度更快**

  这两个系统调用的典型用法：

  len = read(fd, buf, nbyte);

  len = write(fd, buf, nbyte);

  - read需要内核将磁盘数据读入到内核缓冲区，再复制到用户进程的缓冲区中，write方法类似
  - 内存映射方式是访问文件速度最快的方法

- **提供了多个独立启动的进程共享内存的一种手段**
  - 多个进程都通过指针映射同一个文件的相同区域，实际访问同一段内存区域，这段内存是同一文件区域的内存映射
  - 某进程修改数据，就会导致另个进程可以访问到的数据发生变化，实现多进程共享内存的另外一种方式
  - 在Windows下就可以通过这种方式实现多进程共享内存

  注意：多进程之间访问时的同步和互斥，必须通过信号量等机制保证

内存映射文件相关系统调用(1)

系统调用mmap

通知系统把哪个文件的哪个区域以何种方式映射

void *mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);

执行成功，返回一个指针；否则返回-1，errno记录失败原因

mmap的参数

- addr指定逻辑地址空间中映射区的起始地址，一般选为0，让系统自动选择
- fd：已打开文件的文件描述符
- 映射的范围是从offset开始的len个字节
- prot对映射区的保护要求：PROT_READ和PROT_WRITE，必须与open打开匹配
- flags选MAP_SHARE，多进程共享方式

内存映射文件相关系统调用(2)

举例

```
char *p;
p = mmap(0, 65536, PROT_READ | PROTO_WRITE, MAP_SHARED, fd, 0);
```

- p是一个指针，是文件fd从0开始的65536个字节
- 程序像访问数组那样访问p[0]~p[65535]。操作这段内存最终读写磁盘文件。系统在合适时机将修改内容写回磁盘文件或者读取文件

系统调用munmap

程序调用函数munmap，或者，进程终止时，文件的内存映射区被删除

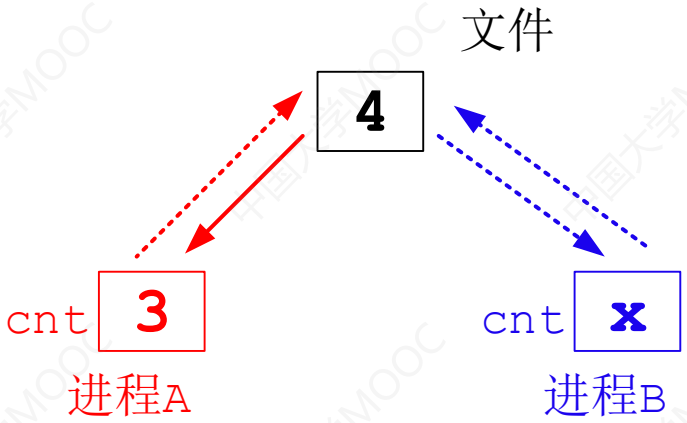
```
int munmap(void *addr, size_t len);
```

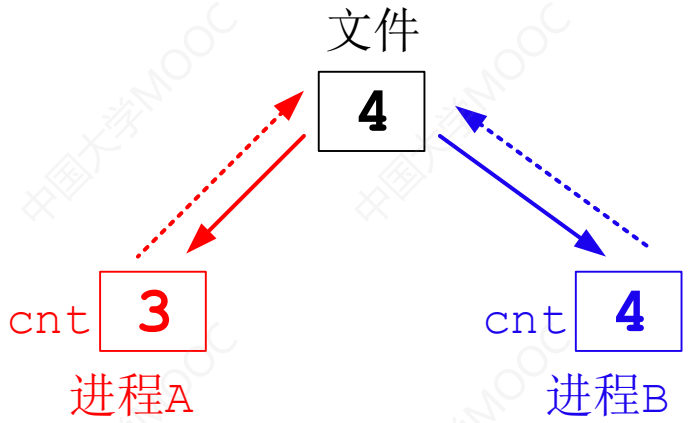
文件和记录的锁定

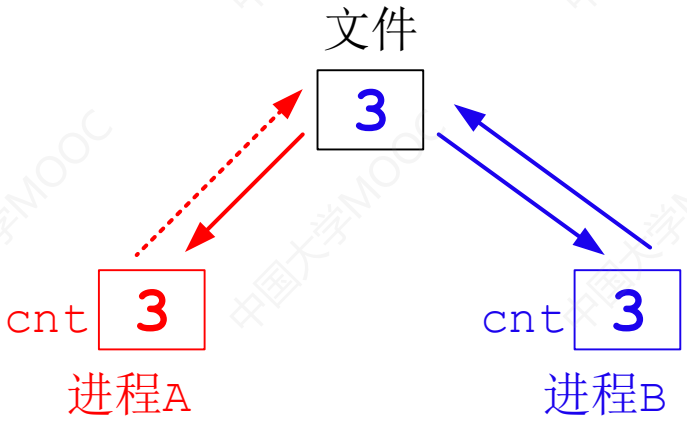
一个文件访问的问题程序

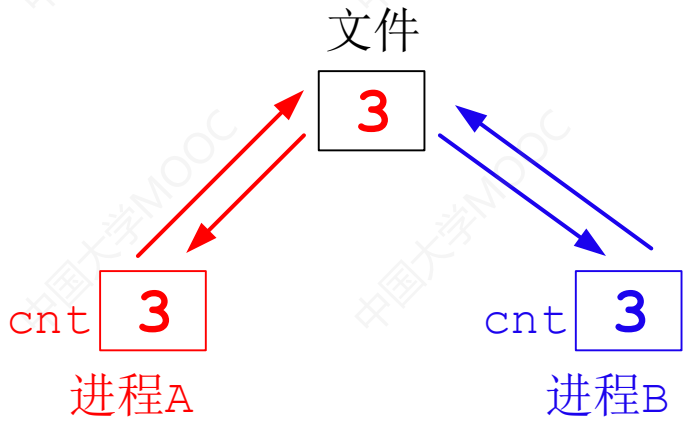
```
/* 程序中省略了open等调用的错误处理 */
int fd, cnt;
fd = open("sanya",O_RDWR);
for (;;) {
     printf("按回车键售出一张票，按Ctrl-D退出 . . .");
     if (getchar() == EOF) break;
     lseek(fd, SEEK_SET, 0);
     read(fd, &cnt, sizeof(int));
     if (cnt > 0) {
          printf("飞往三亚机票，票号:%d\n", cnt);
          cnt--;
          lseek(fd, SEEK_SET, 0);
          write(fd, &cnt, sizeof(int));
     } else
          printf("无票\n");
 }
 close(fd);
```

一个文件访问的问题程序









文件和记录锁定机制

- **文件可以同时被多个进程访问，需要互斥**
  - （操作系统教科书中的“读者写者问题”）
- **使用信号灯机制和共享内存等方法**
  - 非常复杂，Linux提供了对文件和记录的锁定机制，用于多进程间对文件的互斥性访问
- **术语“记录”**
  - 指的是一个文件中从某一位置开始的连续字节流，Linux提供了对记录锁定的机制，用于锁定文件中的某一部分
  - 可以把一个记录定义为从文件首开始直至文件尾，所以，文件锁定实际上是记录锁定的一种特例

共享锁和互斥锁

- **共享锁（或叫读锁）**
  - 多进程读操作可以同时进行，即某一进程读记录时，不排斥其它进程也读该记录，但是排斥任何对该记录的写操作
- **互斥锁（也叫写锁）**
  - 当某进程写记录时，排斥所有其它进程对该记录的读和写

文件锁操作(咨询式锁定)

```
int fcntl(int fd,  int cmd,  struct flock *lock);
```

结构体flock定义如下：

```
struct flock {
    short l_type;
    short l_whence;
    long l_start;
    long l_len;
};
```

type：F_RDLCK读锁；F_WRLCK写锁；F_UNLCK解锁

whence指定坐标0的位置：SEEK_SET文件首；

SEEK_CUR当前读写指针处；SEEK_END文件尾

start指定记录的起点字节：

可为正数或负数，相对于坐标0的偏移量，

len描述记录含有多少字节，值0指从指定位置开始超过文件尾

基本应用：

cmd在对记录上锁或解锁的应用中，一般取F_SETLKW(等待)

其他应用，实现进程单例：

cmd取F_SETLK(不等待)对某数据文件指定字节 上写锁可以实现系统中同一个程序只许启动一次进程的要求

售票程序举例（写锁）

```
lock.l_whence = SEEK_SET;
lock.l_start = 0;
lock.l_len = sizeof(int);
lock.l_type = F_WRLCK;
fcntl(fd, F_SETLKW, &lock);

lseek(fd, SEEK_SET, 0);
read(fd, &cnt, sizeof(int));
if (cnt > 0) {
    printf("飞往三亚机票，票号:%d\n", cnt);
    cnt--;
    lseek(fd, SEEK_SET, 0);
    write(fd, &cnt, sizeof(int));
} else
    printf("无票\n");

lock.l_type = F_UNLCK;
fcntl(fd, F_SETLK, &lock);
```

查询程序举例（读锁）

```
lock.l_whence = SEEK_SET;
lock.l_start = 0;
lock.l_len = sizeof(int);
lock.l_type = F_RDLCK;
fcntl(fd, F_SETLKW, &lock);

lseek(fd, SEEK_SET, 0);
read(fd, &cnt, sizeof(int));
printf("    还剩%d张票\n", cnt);

lock.l_type = F_UNLCK;
fcntl(fd, F_SETLK, &lock);
```

进程间协作：小结

- IPC，消息队列，死锁问题
- 信号灯
- 共享内存机制
- 信号灯+共享内存:生产者消费者问题
- 内存映射方式访问文件
- mmap与read/write比较
- 文件和记录的锁定的必要性
- 读锁与写锁
- 咨询式锁的使用：注意安全性保障以及进程状态的变化