Review：管道 信号

- 管道
  - 以文件的形式使用
  - 类似“水管”，单向
  - 匿名、有名
  - 实现进程之间的数据传输
- 信号
  - 是一个整数，是一种机制
  - 内核发进程，进程发进程，进程发自己
  - 可设置其处理函数，实现特定功能
  - 可重入函数、定时

第7章 进程间通信

主要内容

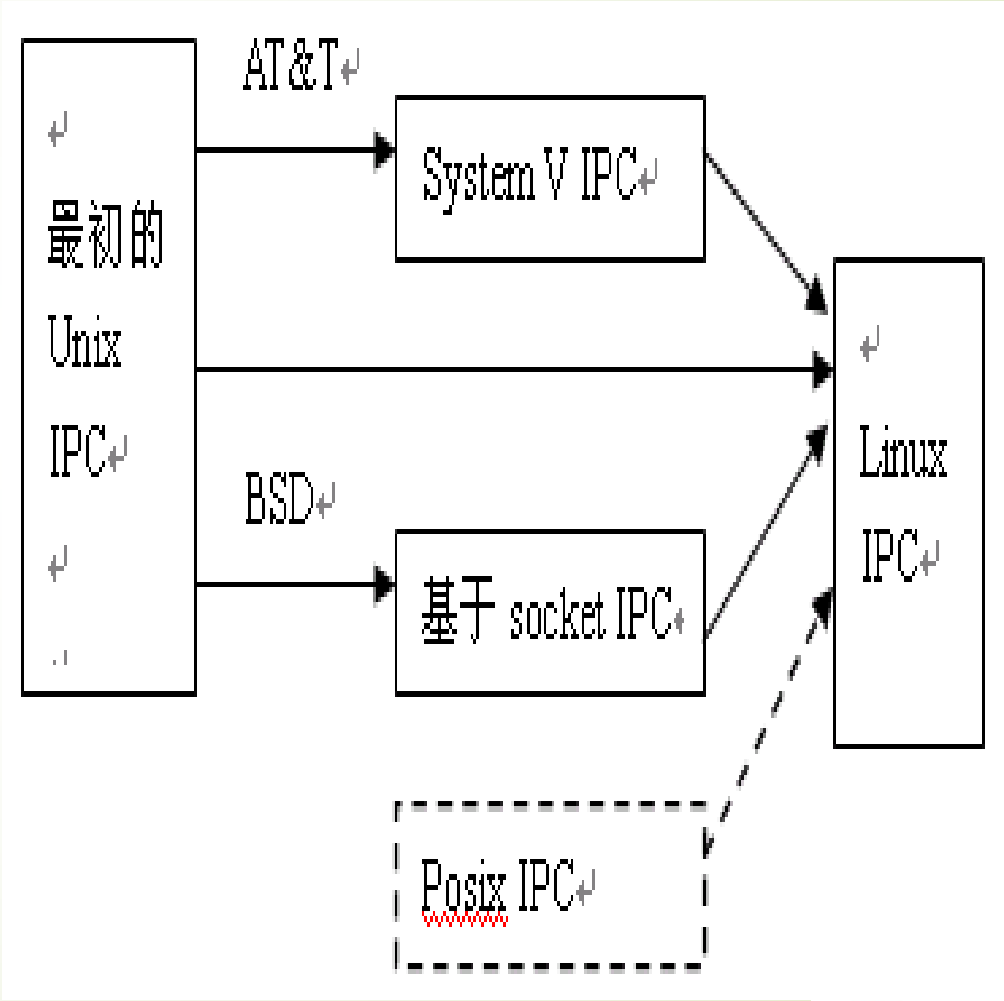
- 7.1 进程间通信基本概念
- 7.3 System V信号量
- 7.4 POSIX信号量
- 7.5 共享内存
- 7.6 消息队列

7.1 基本概念

- 进程间通信的主要作用
  - 数据传输
  - 通知事件
  - 资源共享
  - 进程控制
    - 有些进程希望完全控制另一个进程的执行（如gdb进程控制被调试的程序），此时控制进程能够拦截另一个进程的所有系统调用和异常，并能够及时知道它状态的改变。

进程通信的实现方式

- 单机内的进程通信
- 跨主机的基于**socket**的进程间通信
- 标准化的进程间通信方式



进程间通信的分类

- 进程之间传递信息量大小的不同，可将进程间通信划分为两大类型：
  - 传递控制信息的低级通信
    - 低级通信主要用于进程之间的同步、互斥、终止、挂起等控制信息的传递
  - 大批数据信息的高级通信
    - 高级通信主要用于大量数据的传递。

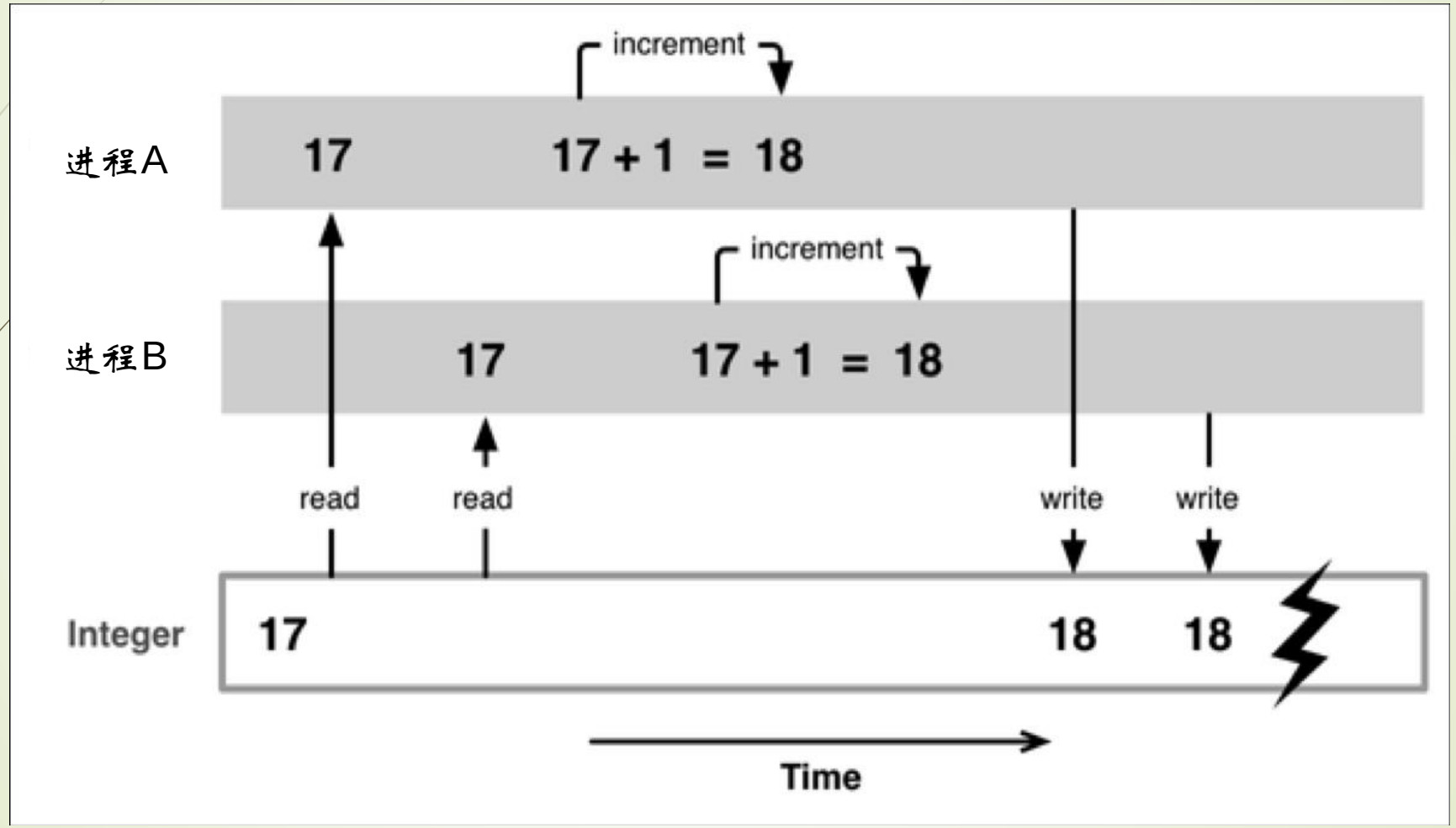
主要的几种方式

- 管道：有名与无名管道
- 信号（Signal）
- 消息队列：消息队列是消息所构成的链接表，包括POSIX消息队列和System V消息队列。
- 共享内存：使得多个进程可以访问同一块内存空间，是最快的可用IPC形式。
- 信号量：主要作为进程间以及同一进程不同线程之间的同步手段。
- 套接字（Socket）：更为一般的进程间通信机制，可用于不同机器之间的进程间通信。

同步和互斥的概念

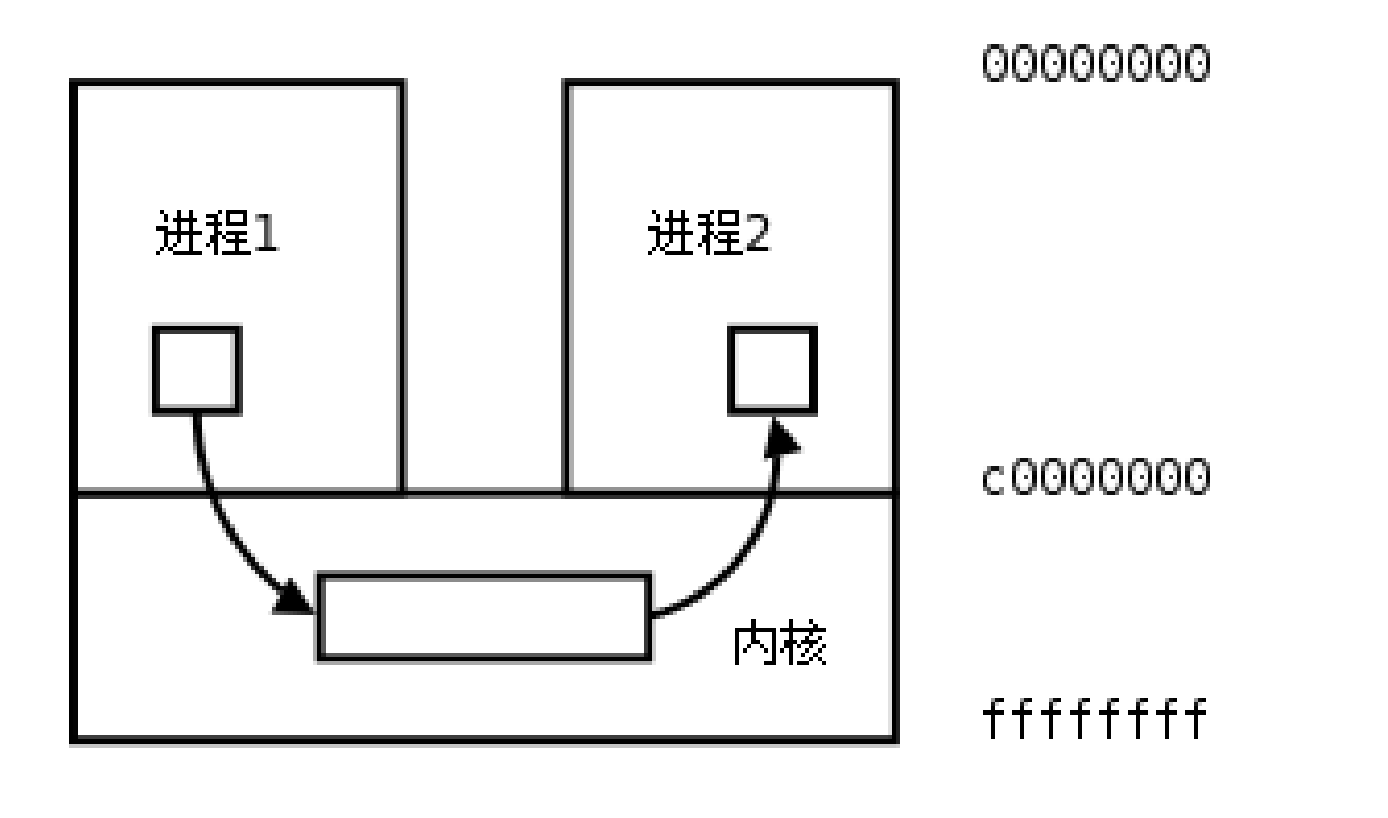
- 临界资源
  - 许多资源在某一时刻只能允许一个进程使用，这种资源称为临界资源
    - 例如打印机等数据结构，如果有多个进程同时去使用这类资源就会造成混乱
- 互斥
  - 进程间相互排斥的使用临界资源的现象，就叫互斥
- 同步
  - 进程之间的关系是相互依赖关系。比如前一个进程的输出作为后一个进程的输入，当第一个进程没有输出时第二个进程必须等待
- 临界区
  - 几个进程若共享同一临界资源，它们必须以互相排斥的方式使用这个临界资源。把访问临界资源的代码段称为临界区

多进程引发的“竞态”



可用于空间理解的示意图

全局变量的作用范围是?



主要内容

- 7.1 进程间通信基本概念
- 7.3 System V信号量
- 7.4 POSIX信号量
- 7.5 共享内存
- 7.6 消息队列

7.3 信号量

- SYSTEM V 的来历
- 信号量的作用：实现对共享资源的保护，防止多进程或多线程的并发带来的不一致问题
- 同步和竞争
  - 临界资源在同一时刻只允许有限个（通常只有一个）进程可以访问（读）或修改（写）的资源
  - 通常包括硬件资源（处理器、内存、存储器及其它外围设备等）和软件资源（共享代码段、共享结构和变量等）
  - 访问临界资源的代码叫做临界区，临界区本身也会称为临界资源。

信号量相关数据结构

- 一个结构体，一对原子操作
  - 包括一个称为信号量的变量和在该信号量下等待资源进程等待队列
  - 以及对信号量进行的两个原子操作（**P/V操作**）
- 信号量对应于某一种资源，取一个非负的整形值
  - 信号量值指的是当前可用的该资源的数量
  - 若等于0则意味着目前没有可用的资源

信号量的组成

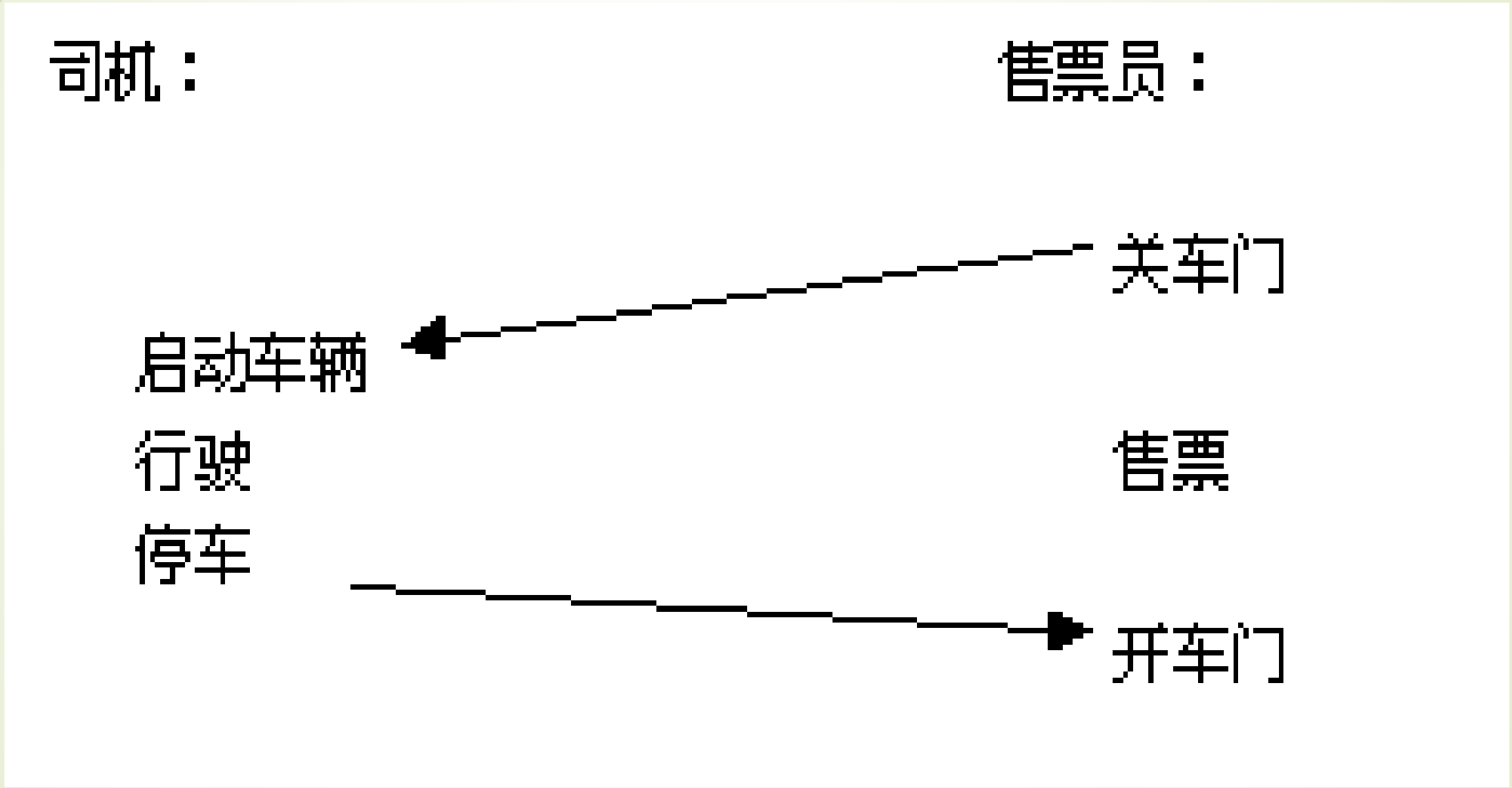
```
struct semaphore {
        int count;
        queueType queue;
};
void semWait(semaphore s)
{
        s.count--;
        if (s.count < 0) {
          /* 把当前进程插入队列*/;
          /* 阻塞当前进程*/;
        }
}
void semSignal(semaphore s)
{
        s.count++;
        if (s.count <= 0) {
          /* 把进程 p 从队列中移除*/;
          /* 把进程 p 插入就绪队列*/;
        }
}
```

数据结构

P 操作

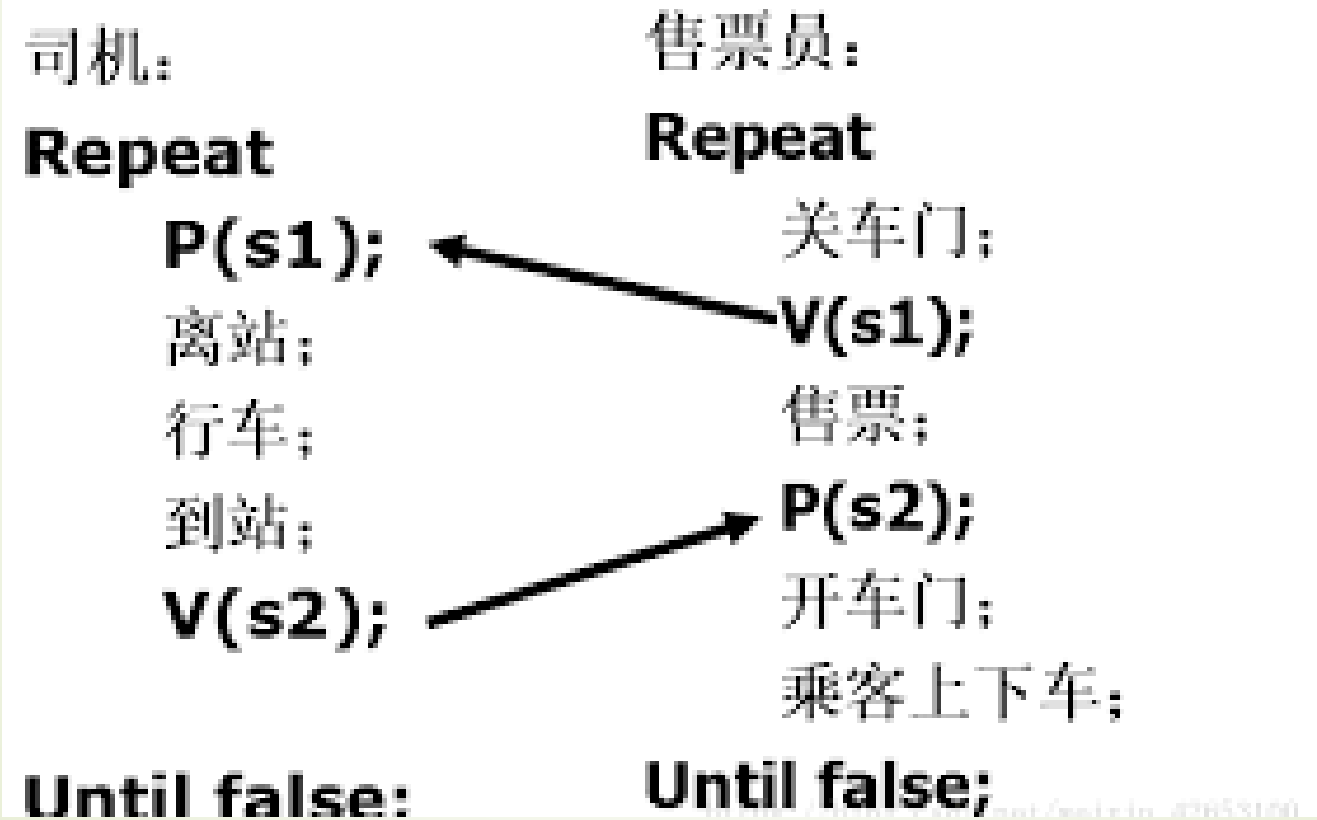
V 操作

同步问题

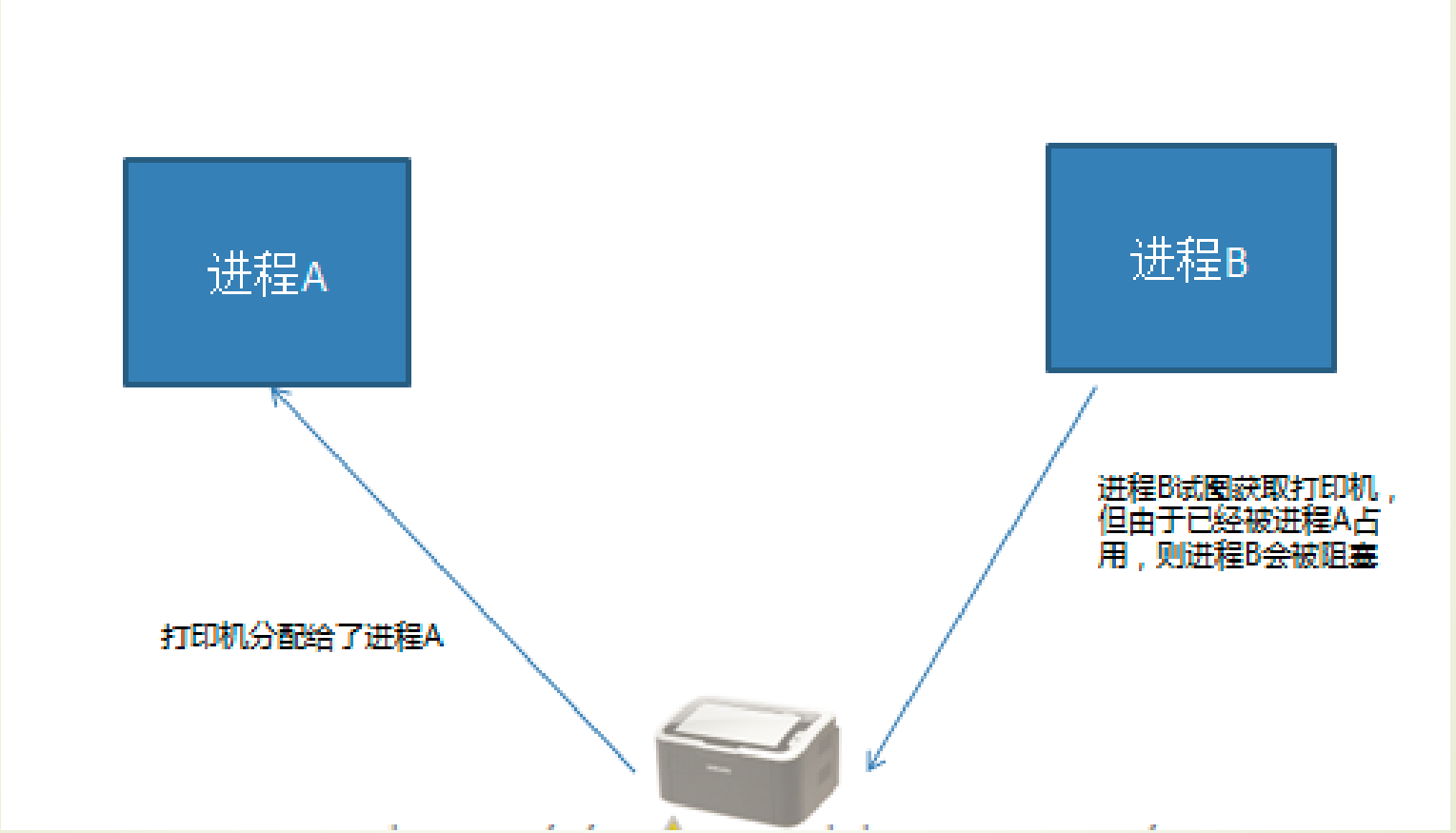


信号量实现同步

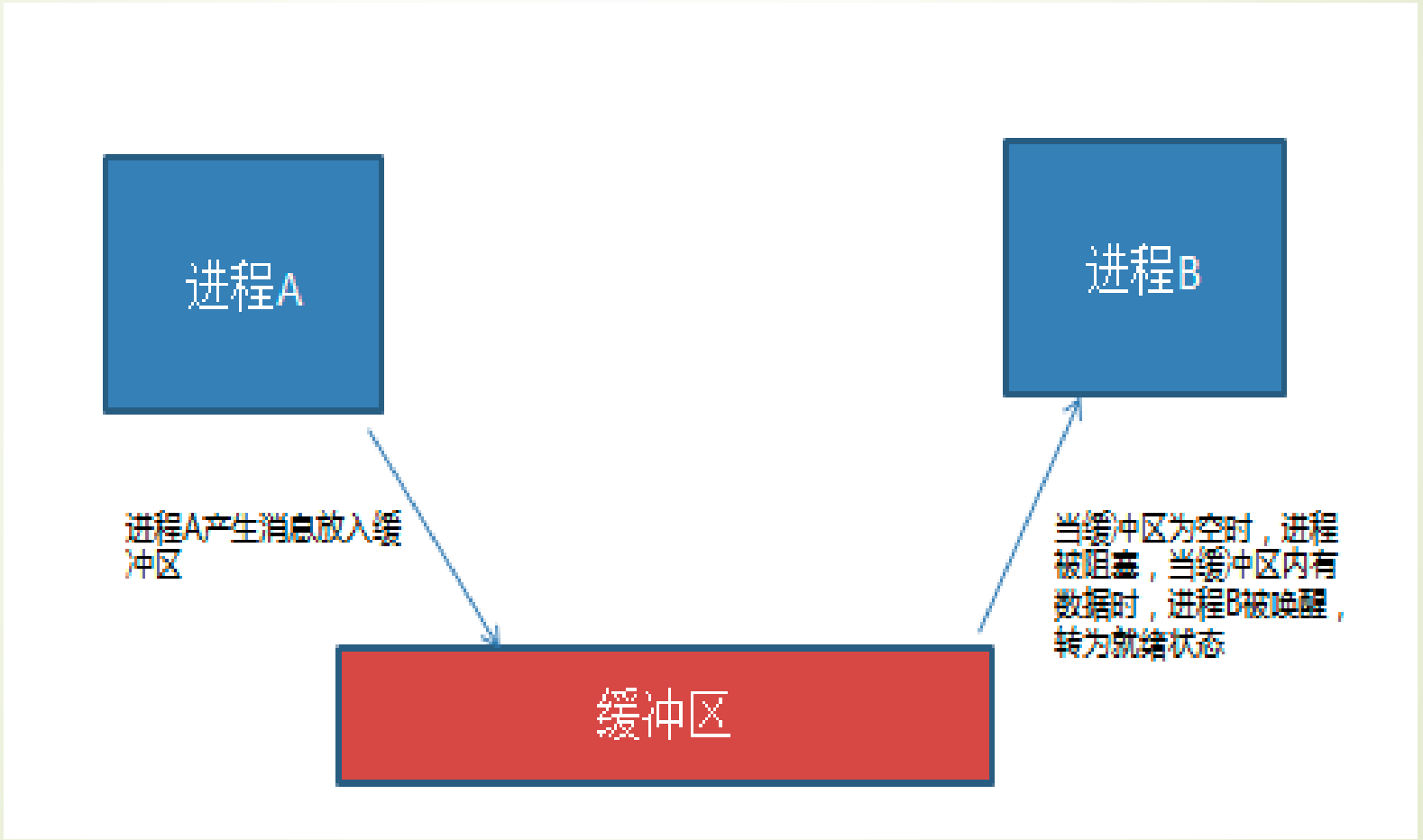
- 同步　初始值为0



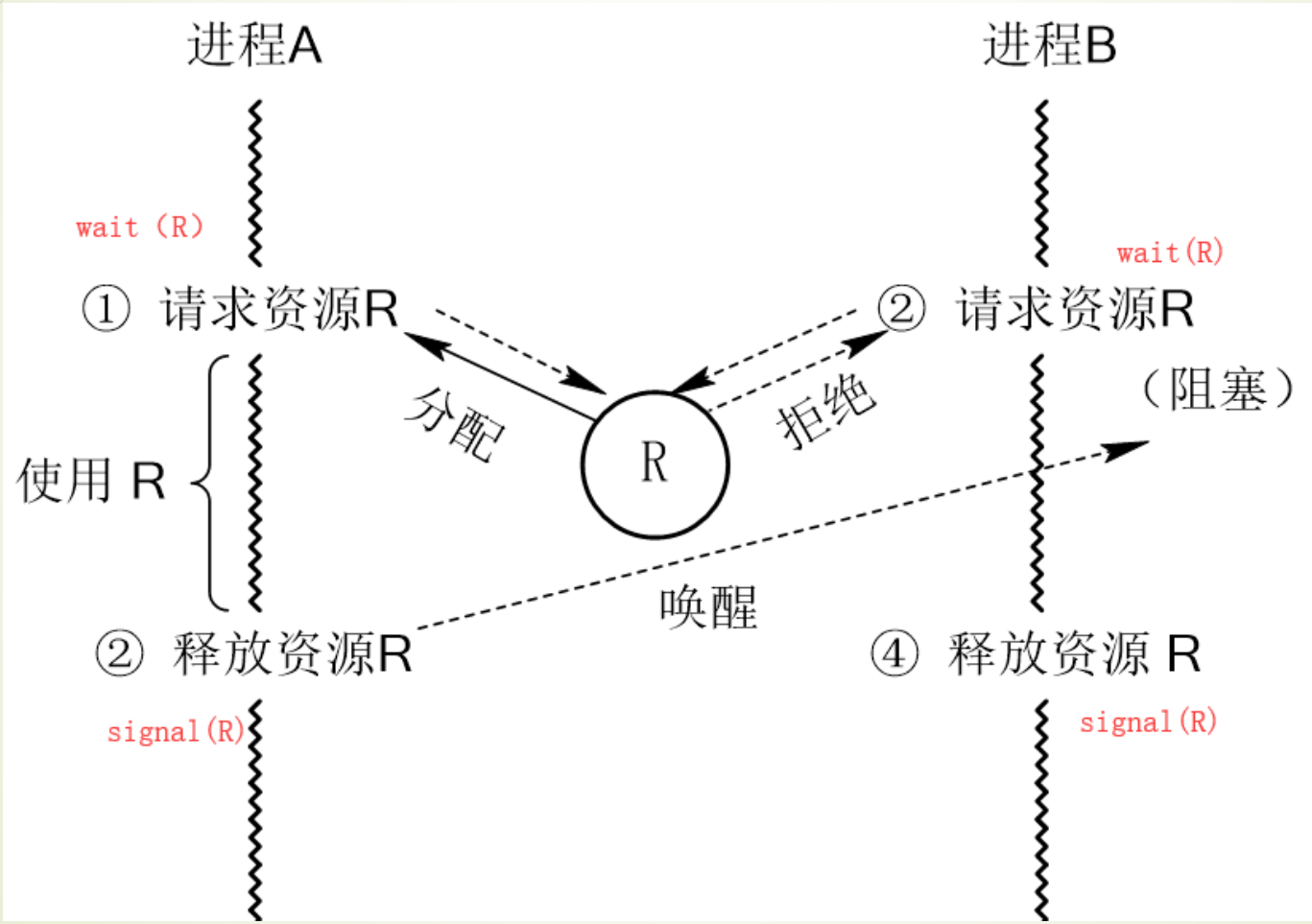
互斥问题



互斥问题



信号量实现互斥


进程A
进程B
wait (R)
wait(R)
① 请求资源R
② 请求资源R
分配
拒绝
R
（阻塞）
使用 R
唤醒
② 释放资源R
④ 释放资源 R
signal(R)
signal(R)

总结：利用信号量的实现同步和互斥

- 设信号量为S
- 实现互斥　初始化 S = 1

进程1	进程2
P(S)	P(S)
临界区	临界区
V(S)	V(S)

- 实现同步　初始化 S = 0

进程1	进程2
……	……
P(S)	V(S)
……	……

使用System V信号量以下4个步骤

- ① 创建信号量或获得在系统中其它进程已经创建的已存信号量，此时需要调用 **semget()** 函数。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 初始化信号量，此时使用 **semctl()** 函数的 **SETVAL操作**。当使用二维信号量时，通常将信号量初始化为1。
- ③ 信号量的P和V操作，此时，调用 **semop()**函数。这一步是实现进程间的同步和互斥的核心工作部分。
- ④ 当不需要信号量时，从系统中删除它，此时使用 **semctl()函数**的 **IPC_RMID操作**。

SYSTEM V信号量函数汇总

- 头文件： <sys/sem.h>
- IPC对象创建或打开函数： semget
- IPC控制函数： semctl
- IPC操作函数： semop

semget()

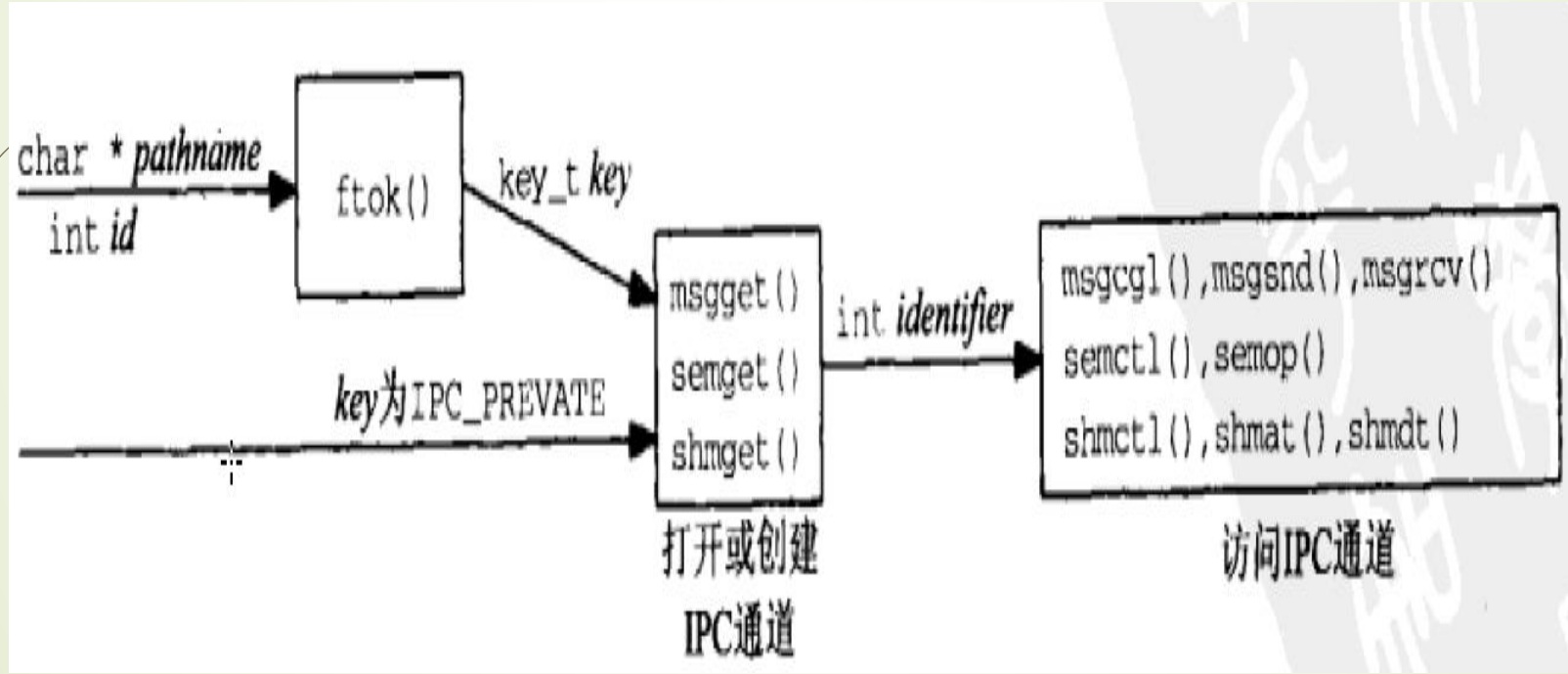
- semget()函数的功能为创建一个新的信号量集，或获取一个系统中已经存在的信号量集，该函数存在于函数库sys/sem.h中，其函数声明如下：
- int semget(key_t key, int nsems, int semflg);
- 若该函数调用成功则返回信号量的标识符，否则返回-1，并设置errno。
  - 常见errno的值与其含义如下：
    - EACCES。表示进程无访问权限；
    - ENOENT。表示传入的键值不存在；
    - EINVAL。表示nsens小于0，或信号量数已达上限；
    - EEXIST。当semflg设置指定了ICP_CREAT和IPC_EXCL时，表示该信号量已经存在。

semget()----续

- 参数key表示信号量的键值，通常为一个整数（详见后面ftok函数说明）；
- 参数nsems表示创建的信号量数目；
- 参数semflg为标志位，与open()、msgget()函数中的标志位功能相似，都用来设置权限，权限位可与**IPC_CREAT**以及**IPC_EXCL**发生位或
  - 另外若该标志位设置为IPC_PRIVATE，表示该信号量为当前进程的私有信号量。

由IPC键生成IPC标识符

- 一定要注意pathname是已存在路径,比如当前目录./



semctl()

- semctl()函数可以对信号量或信号量集进行多种控制，该函数存在于函数库sys/sem.h中，其函数声明如下：
- int semctl(int semid, int semnum, int cmd, ...);
  - 参数semid表示信号量标识符，通常为semget()的返回值；
  - 参数semnum表示信号量在信号量集中的编号，该参数在使用信号量集时才会使用，通常设置为0，表示取第一个信号；
  - 参数cmd表示对信号量进行的操作。

semctl ()----续1

- 最后一个参数是一个可选参数，依赖于参数cmd，使用该参数时，用户必须在程序中自定义一个如右所示的共用体：

```
union semun{
    int val;                    //cmd为SETVAL时，用于指定信号量值
    struct semid_ds *buf;       //cmd为IPC_STAT时或IPC_SET时生效
    unsigned short *array;      //cmd为GETALL或SETALL时生效
    struct seminfo *_buf;       //cmd为IPC_INFO时生效
};
```

semctl ()----续2

- semun共用体中的struct semid_ds是一个由内核维护，记录信号量属性信息的结构体
- 该结构体的类型定义如下：

```
struct semid_ds {
    struct ipc_perm sem_perm;  //所有者和标识权限
    time_t          sem_otime;     //最后操作时间
    time_t          sem_ctime;     //最后更改时间
    unsigned short  sem_nsems; //信号集中的信号数量
};
```

semctl ()----续3

- cmd常用的设置为SETVAL和IPC_RMID
- 其含义分别如下：
  - SETVAL。表示semctl()的功能为初始化信号量的值，信号量值通过可选参数传入，在使用信号量前应先对信号量值进行设置
  - IPC_RMID。表示semctl()的功能为从系统中删除指定信号量。信号量的删除应由其所有者或创建者进行，没有被删除的信号量将会一直存在于系统中

semop()

- semop()函数的功能为改变信号量的值，该函数存在于函数库sys/sem.h中，函数声明如下：
- int semop(int semid, struct sembuf *sops, unsigned nsops);
- 若该函数调用成功返回0，否则返回-1，并设置errno。
  - 参数semid同样为semget()返回的信号量标识符；
  - 参数nsops表示参数sops所指数组中元素的个数。
  - 参数sops为一个struct sembuf类型的数组指针，该数组中的每个元素设置了要对信号量集中的哪个信号做哪种操作。

semop ()----续1

struct sembuf结构体定义如下：

```
struct sembuf{
    short sem_num;//信号量在信号量集中的编号
    short sem_op;   //设置为-1时表示P操作
                    //设置为+1时表示V操作
    short sem_flag; //标志位
};
```

- 结构体成员sem_flg通常设置为SEM_UNDO，若进程退出前没有删除信号量，信号量将会由系统自动释放。

key_t键和ftok函数

- System V IPC使用key_t值作为IPC对象名字。
  - 头文件<sys/types.h>把key_t这个数据类型定义为一个整数，它通常是一个至少32位的整数，这些整数值通常是由ftok函数赋予的。
  - 函数ftok把一个已存在的路径名和一个整数标识符转换成一个key_t值。
- #include <sys/ipc.h>
- // 成功则返回IPC键，出错则返回-1
- key_t ftok(const char *pathname, int id);

ftok()函数

- 当在进程中使用System V IPC系列的接口进行通信时，必须指定一个key值，这是一个key_t类型的变量
  - 通过Linux系统中的一个函数——ftok()来获取
  - ftok()函数位于函数库sys/types中，其定义如下：
- key_t ftok(const char *pathname, int proj_id);
  - 参数pathname表示已存在路径名，一般会设置为当前目录“.”；
  - 参数proj_id由用户指定，为一个整型数据，一般设置为0。
  - 当ftok()函数被调用时，该函数首先会获取目录的inode，其次将十进制的inode及参数proj_id否转换为十六进制，最后将这两个十六进制数链接，生成一个key_t类型的返回值。

例7-8 P157 /* semlib.h */

```
#define DELAY_TIME  3
union semun{        //定义信号量编号联合体
    int val;
    struct semid_ds *buf;
    unsigned short *array;
};

int init_sem(int sem_id,int init_value); //信号量初始化函数

int del_sem(int sem_id); //信号量删除函数

int sem_p(int sem_id); //信号量的P操作函数

int sem_v(int sem_id); //信号量的V操作函数
```

```
/*semlib.c*/          #include "semlib.h"
int init_sem(int sem_id,int init_value)
{
        union semun sem_union;
                //设置信号量的初始值
        sem_union.val=init_value;
                //SETVAL参数表示设置信号量的初始值
        if(semctl(sem_id,0,SETVAL,sem_union) == -1)      {
                perror("initializing semaphore");
                return -1;
        }       return 0;
}
int del_sem(int sem_id) /*从系统中删除信号量*/
{
        union semun sem_union;
                // IPC_RMID 参数表示删除sem_id信号量
        if(semctl(sem_id,0,IPC_RMID,sem_union)==-1)    {
                perror("Delete semaphore failed");
                return -1;
        }
}
}
```

```
int sem_p(int sem_id) /*信号量P操作函数 */
{
        struct sembuf sem_b;
        sem_b.sem_num=0;  //信号量编号，单个信号量时，设置为0
        sem_b.sem_op= -1;  //设置为-1表示进行P操作
        sem_b.sem_flg=SEM_UNDO;
                //当程序退出时未释放该信号量时，由操作系统负责释放
                // 对编号为sem_id的信号量执行P操作
        if(semop(sem_id,&sem_b,1)==-1)    {
                perror("P operation failed");     return -1;
        }    return 0;
}
int sem_v(int sem_id){
        struct sembuf sem_b;
        sem_b.sem_num=0;
        sem_b.sem_op=1; //1表示V操作
        sem_b.sem_flg=SEM_UNDO;
        if(semop(sem_id,&sem_b,1)==-1)          {
                perror("V operation failed");
                return -1;
        } return 0;
}
```

```
/*semexample.c 主程序，实现父子进程之间执行顺序的控制*/  #include "semlib.h"
int main(void){       pid_t result;      int sem_id;
   /*获取信号量的标识符，在下列中，不同独立进程可获取相同的信号量标识符，该操作通过ftok函数获得信号量的键值。ftok函数通过获取第一个参数所指定的文件的i节点号，在其之前加上子序号作为键值返回*/
       sem_id=semget(ftok(".",'a'),1,0666|IPC_CREAT);
       init_sem(sem_id,0);/*设置信号量的初值为0*/
       result=fork();
       if(result==-1)
              perror("Fork failed\n");
       else if(result==0)/*子进程*/{
              sleep(DELAY_TIME);
              printf("The child progress output\n");     printf("BBBBBBBBBBB\n");
              sem_v(sem_id);       //V操作
       }
       else {   /*父进程*/
              sem_p(sem_id);    //P操作
              printf("The father process output \n");   printf("AAAAAAAAAA\n");
              del_sem(sem_id);
       }
       exit(0);
}
```

实例

- 教材配套源代码\chapter7 进程间通信
  - \semlib.h
- 教材配套源代码\chapter7 进程间通信
  - \semlib.c
- 教材配套源代码\chapter7 进程间通信
  - \semexample.c

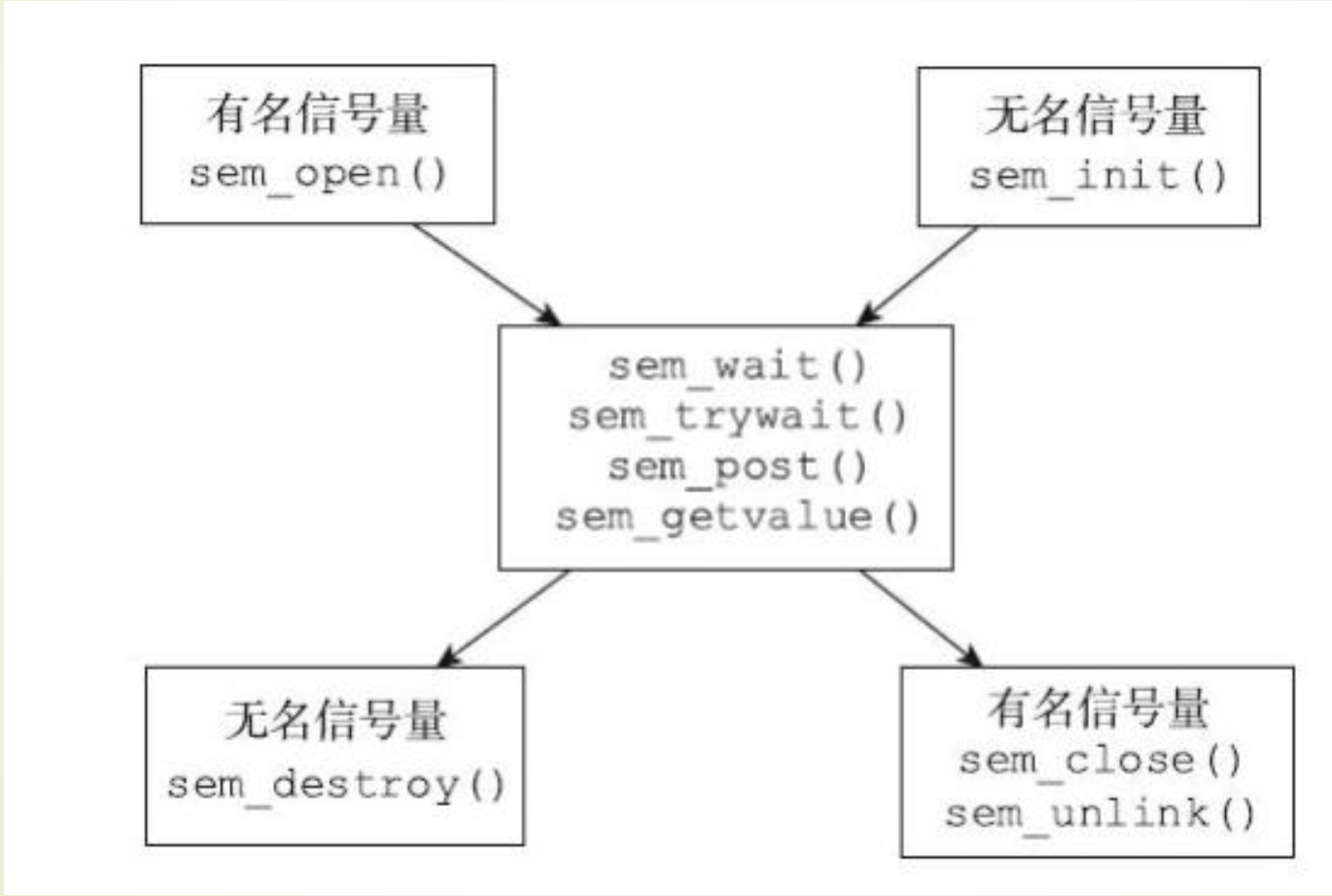
主要内容

- 7.1 进程间通信基本概念
- 7.3 System V信号量
- 7.4 POSIX信号量
- 7.5 共享内存
- 7.6 消息队列

7.4 POSIX有名信号量

- POSIX的来历
- 任何关系的进程之间的同步
- 有名信号量把信号量的值保存在文件中。这决定了它的用途非常广：
  - 可用于线程，也可用于相关进程间，甚至是不相关进程

POSIX的两类信号量



POSIX有名信号量

- 打开一个已存在的有名信号量，或创建并初始化一个有名信号量。该调用完成信号量的创建、初始化和权限的设置。
  - sem_t* sem_open(const char * name, int oflag, mode_t mode , int value);
    - name：文件的路径名； 一般有名信号量文件都是在/dev/shm目录下的。
    - oflag：O_CREAT或O_CREAT|O_EXCL两个取值；
    - mode_t ： 控制新的有名信号量的访问权限；
    - value ： 指定信号量的初始化值。

- 信号量的关闭和销毁
- 可通过sem_close(sem_t)进行关闭，然后使用sem_unlink(const char *name)将有名信号量删除

- P操作，请求资源：
  - int sem_wait(sem_t *sem);
  - 阻塞的函数， 测试所指定信号量的值。它的操作是原子的。
    - sem>0，那么它减1并立即返回
    - sem==0，则在此调用处阻塞直到sem>0时为止，此时立即减1，然后返回

- V操作，释放资源：
  - int sem_post(sem_t *sem);把指定的信号量sem的值加1，唤醒正在等待该信号量的任意进程
  - 在编译使用sempahore.h的函数时，需要使用-lrt或者-lpthread链接库文件
- 教材配套源代码\chapter7 进程间通信
  - \outdef.c
- 教材配套源代码\chapter7 进程间通信
  - \outabc.c

```
char SEM_NAME[]= "process1";   char SEM_NAME2[]= "process2";     #define SHMSZ 27
int main(int argc,char **argv)   {
    char ch;       int shmid;       key_t key;       int fd;       sem_t *mutex,*mutex2;
    //create & initialize existing semaphore
    mutex = sem_open(SEM_NAME,O_CREAT,0777,0);
    mutex2 = sem_open(SEM_NAME2,O_CREAT,0777,1);
    if(mutex == SEM_FAILED | |mutex2==SEM_FAILED)    {
       perror("unable to execute semaphore");   sem_close(mutex);   sem_close(mutex2);   exit(-1);
    }
    fd=open("a.txt",O_CREAT|O_WRONLY|O_TRUNC|O_APPEND);
    if(fd == -1)    {  perror("open failed");      sem_close(mutex);       exit(-1);    }
    while (1) {
        sem_wait(mutex);  //P操作          write(fd,"ABC\n",sizeof("ABC\n")-1);       close(fd);
        fd=open("a.txt",O_WRONLY|O_APPEND);          sleep(1);
        sem_post(mutex2);       //V操作
     }
    sem_close(mutex);
    sem_unlink(SEM_NAME);
    close(fd);       exit(0);
} 例7-9 P161
```

```
char SEM_NAME[]= "process1";      char SEM_NAME2[]= "process2";     #define SHMSZ 27
int main(int argc,char **argv)   {
    char ch;       int shmid;       key_t key;       int fd;       sem_t *mutex,*mutex2;
     key = 19753; //name the shared memory segment
     mutex = sem_open(SEM_NAME,O_CREAT,0777,0); //create & initialize existing semaphore
    mutex2 = sem_open(SEM_NAME2,O_CREAT,0777,1);
    if(mutex == SEM_FAILED  | | mutex2 == SEM_FAILED)    {
        perror("unable to execute semaphore");       sem_close(mutex);       sem_close(mutex2);       exit(-1);
    }
    fd=open("a.txt",O_CREAT | O_WRONLY | O_TRUNC | O_APPEND);
    if(fd == -1)    {
        perror("open failed");       sem_close(mutex);       sem_close(mutex2);       exit(-1);
    }
    while (1) {
         sem_wait(mutex2);  //P操作      write(fd,"DEF\n",sizeof("DEF\n")-1);       close(fd);
         fd=open("a.txt",O_WRONLY | O_APPEND);          sleep(1);
         sem_post(mutex);      //V操作
        }
    sem_close(mutex);
    sem_unlink(SEM_NAME);
    close(fd);    exit(0);
}
```

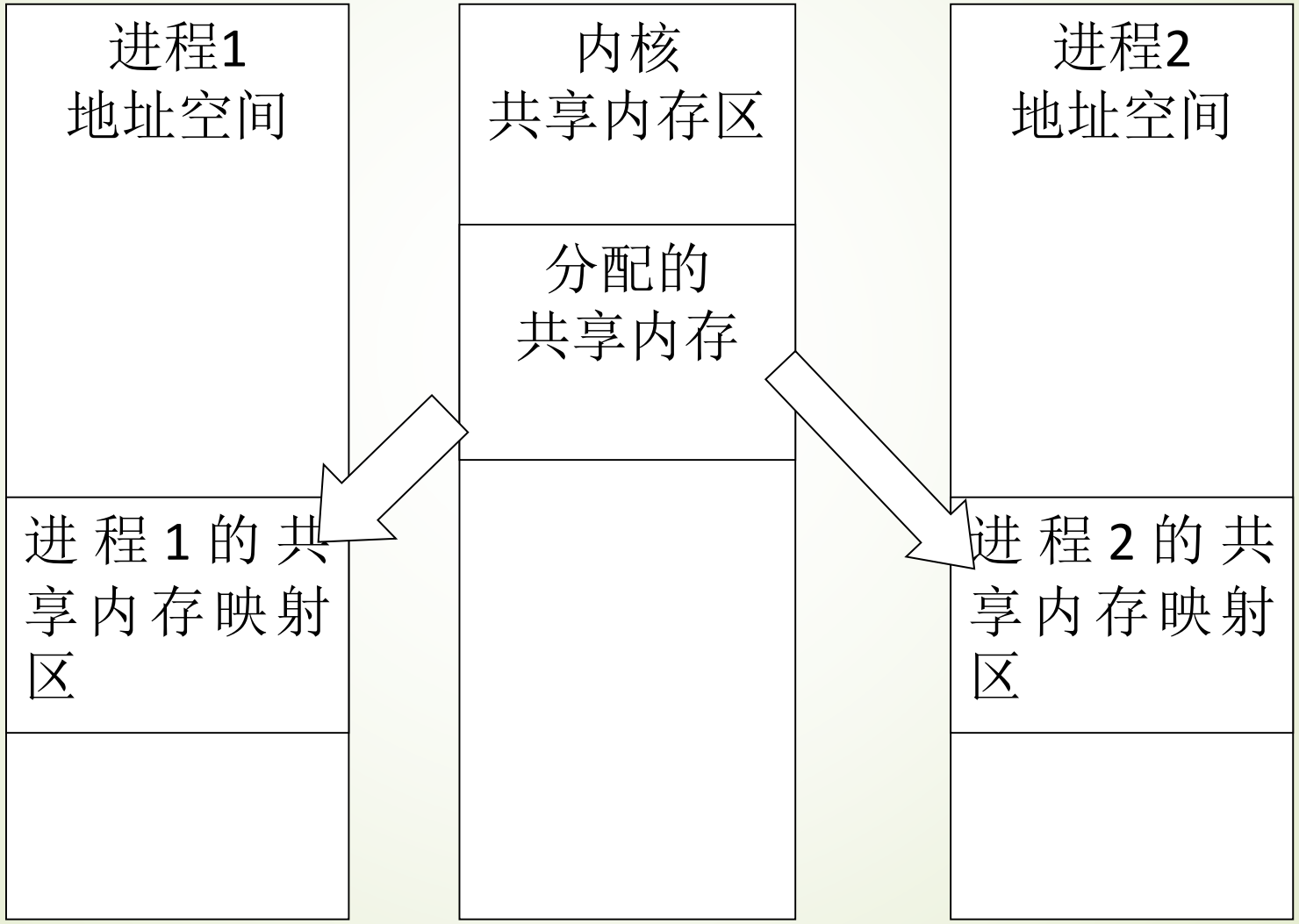
主要内容

- 7.1 进程间通信基本概念
- 7.3 System V信号量
- 7.4 POSIX信号量
- 7.5 共享内存
- 7.6 消息队列

共享内存

- 与消息队列和管道通信机制相比，一个进程要向队列/管道中写入数据时，引起数据从用户地址空间向内核地址空间的一次复制
- 进行消息读取时也要进行一次复制
- 共享内存的优点是完全省去了这些操作
- 共享内存是GNU/Linux现在可用的最快速的进程间通信机制

共享内存



共享内存的使用

- 创建共享内存。从内存中获得一段共享内存区域，这里用到的函数是shmget()；
- 映射共享内存。创建的共享内存映射到具体的进程空间中，使用函数shmat()。
- 撤销映射。使用完共享内存就需要撤销，用到的函数是shmdt()。

SYSTEM V 共享变量函数汇总

- 头文件： <sys/shm.h>
- IPC对象创建或打开函数： shmget
- IPC控制函数： shmctl
- IPC操作函数： shmat　　shmdt

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>
函数原型	int shmget(key_t key, int size, int shmflg)
参数	key：共享内存的键值，多个进程可以通过它访问同一个共享内存，其中有个特殊值IPC_PRIVATE，用于创建当前进程的私有共享内存。 如果要想在key标识的共享内存不存在时，创建它的话，可以与IPC_CREAT做或操作。 共享内存的权限标志与文件的读写权限一样
	size：共享内存区大小
	shmflg：同open()函数的权限位，也可以用八进制表示法
函数返回值	成功：共享内存段标识符
	出错：-1

函数原型	char *shmat(int shmid, const void *shmaddr, int shmflg)	
参数	shmid：要映射的共享内存区标识符	
	shmaddr：将共享内存映射到指定地址（若为0则表示系统自动分配地址并把该段共享内存映射到调用进程的地址空间）	
	shmflg	SHM_RDONLY：共享内存只读
		默认0：共享内存可读写
返回值	成功：被映射的段地址	
	出错：-1	

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>
函数原型	int shmdt(const void *shmaddr)
参数	shmaddr：被映射的共享内存段地址
返回值	成功：0
	出错：-1

所需头文件	#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h>
函数原型	int shmctl(int shmid, int cmd, struct shmid_ds *buf)
参数	shmid共享内存标识符 cmd表示对共享内存的属性执行的相关命令，主要有： • IPC_STAT表示得到共享内存的状态，把共享内存的shmid_ds结构复制到buf中； • IPC_SET：改变共享内存的状态，把buf所指的shmid_ds结构中的uid、gid、mode复制到共享内存的shmid_ds结构内 • IPC_RMID：删除该共享内存 buf：共享内存管理结构体
返回值	成功：0 出错：-1

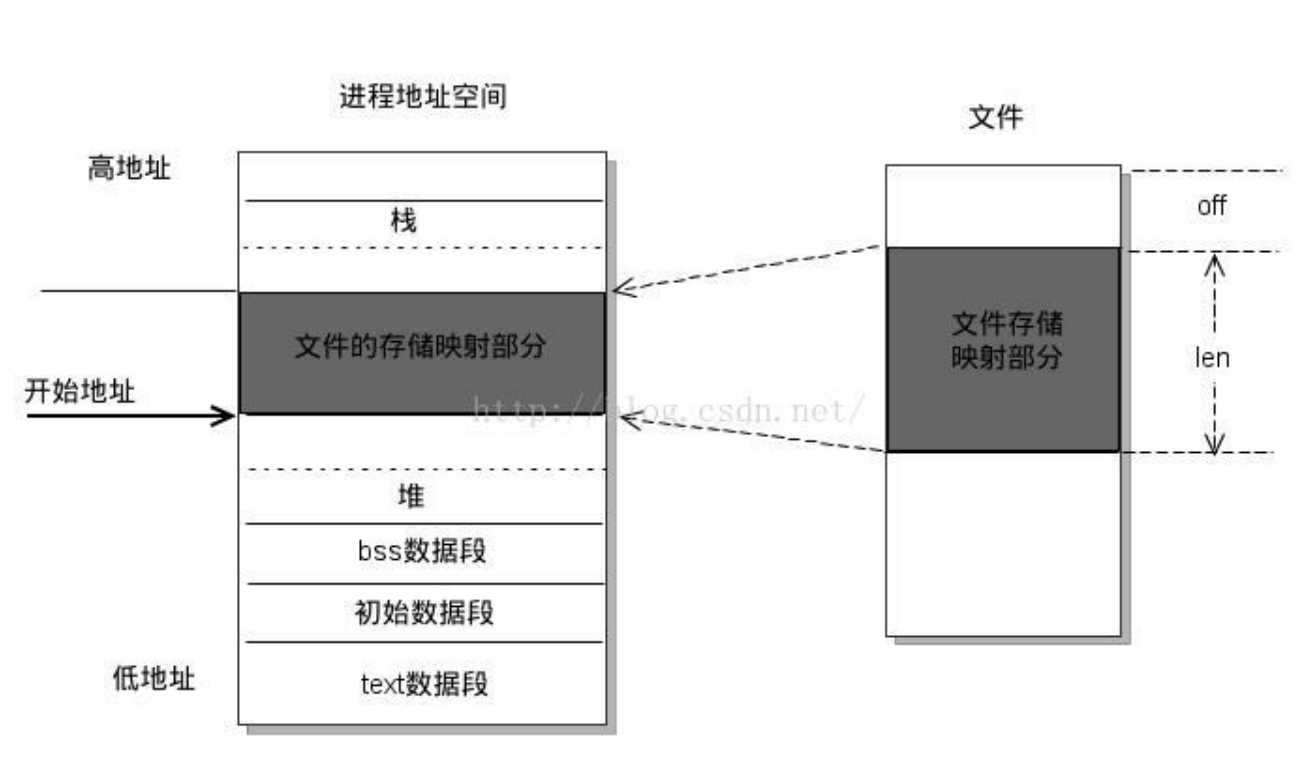
```
// shmemPar.c        #define SIZE 1024
int main()  {
   int shmid ; //共享内存段标识符        char *shmaddr ;       char buff[30];
   int shmstatus; //获取共享内存属性信息        int pid ; //进程ID号
   shmid = shmget(IPC_PRIVATE, SIZE, IPC_CREAT | 0600 ) ;
   if ( shmid < 0 )   {              perror("get shm  ipc_id error") ;              return -1 ;       }
   pid = fork() ;
   if ( pid == 0 )    { //子进程向共享内存中写入数据
      shmaddr = (char *)shmat( shmid, NULL, 0 ) ; //映射共享内存，可读写
      if ( (int)shmaddr == -1 )       {              perror("shmat addr error") ;              return -1 ;          }
      strcpy( shmaddr, "Hello World!\n") ;
      shmdt( shmaddr ) ;
      return  0;
   } else if ( pid > 0) {        sleep(10);
      shmaddr = (char *) shmat(shmid, NULL, 0 ) ;
      if ( (int)shmaddr == -1 )       {              perror("shmat addr error") ;              return -1 ;          }
      strcpy(buff,shmaddr);
      printf("I have  got from shared memory :%s\n", buff) ;
      shmdt( shmaddr ) ;
      shmctl(shmid, IPC_RMID, NULL) ;
   }else{
      perror("fork error") ;
      shmctl(shmid, IPC_RMID, NULL) ;
   }
   return 0 ;
} 例7-10  P165
```

```
#define BUFFER_SIZE  2048  //例7-11  P167
int main(){
        pid_t pid; int sem_id; //信号量ID      int shmid;//共享内存段ID
        char *shm_addr=NULL; //共享内存首地址char buff[40]; //字符串
        int i=0;
        sem_id=semget(ftok(".",'a'),1,0666|IPC_CREAT);/*信号量*/
        init_sem(sem_id,0);//信号量初始化
        if((shmid=shmget(IPC_PRIVATE,BUFFER_SIZE,0666))<0){   perror("shmget");  return -1;     }
        pid=fork();
        if(pid==-1){  perror("fork");  return -1; }
        else if(pid==0) {//子进程
                if ((shm_addr=shmat(shmid,0,0))==(char *)-1) {  perror("shmat");   return -1;  }
                while ( i<3 ) {
                    printf("Child process is waiting for data:\n");
                    sem_p(sem_id); //P操作
                        strcpy(buff,shm_addr);  //读取数据
                        printf("Child get data from shared-memory:%s\n",buff);
                    sem_v(sem_id); //V操作
                    i++;
                }
```

```
            del_sem(sem_id); //
            if((shmdt(shm_addr))<0) {  perror("shmdt");   return -1; }
            if(shmctl(shmid,IPC_RMID,NULL)==-1){  perror("child process delete shared memory "); return -1;}
        }
        else        { //父进程
            if((shm_addr=shmat(shmid,0,0))==(char *)-1){   perror("Parent shmat failed");   return -1;}
            while(i<3)  {
                printf("Please input some string:\n");
                fgets(buff,BUFFER_SIZE,stdin);
                strncpy(shm_addr,buff,strlen(buff));
                sem_v(sem_id);//V操作
                    i++;
                sem_p(sem_id); //P操作
            }
            if((shmdt(shm_addr))<0)         {
                perror("Parent:shmdt");    exit(1);
            }
            waitpid(pid,NULL,0);
        }
        return 0;
}
```

7.5.4 系统调用mmap创建内存映射文件

- Linux提供了内存映射函数mmap,它把普通文件内容映射到一段虚拟内存上,通过对这段内存的读取和修改,实现对文件的读取和修改。
  - 普通文件被映射到进程地址空间后，进程可以向访问普通内存一样对文件进行访问，不必再调用read()，write（）等操作。



mmap系统调用

所需头文件	#include<sys/mman.h>
函数原型	void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
函数作用	将磁盘文件映射到进程的某段内存空间中
参数	addr: 指定映射的起始地址, 通常设为NULL, 由系统指定. length: 将文件的多大长度映射到内存. prot: 映射区的保护方式, 可以是: • PROT_EXEC: 映射区可被执行. • PROT_READ: 映射区可被读取. • PROT_WRITE: 映射区可被写入. • PROT_NONE: 映射区不能存取. flags: 映射区的特性, 可以是: • MAP_SHARED: 对映射区域的写入数据会复制回文件, 且允许其他映射该文件的进程共享. • MAP_PRIVATE: 对映射区域的写入操作会产生一个映射的复制(copy-on-write), 对此区域所做的修改不会写回原文件. fd: 由open返回的文件描述符, 代表要映射的文件. offset: 以文件开始处的偏移量, 必须是分页大小的整数倍, 通常为0, 表示从文件头开始映射.
返回值	成功：文件映射到进程空间的地址 出错：-1

munmap系统调用

所需头文件	#include<sys/mman.h>
函数原型	void *munmap(void *addr, size_t len);
函数作用	解除进程地址空间中的映射关系
参数	addr: 是调用mmap时返回的地址， len: 是映射区的大小
返回值	成功：0
	出错：-1

msync系统调用

所需头文件	#include<sys/mman.h>
函数原型	void *msync (void *addr, size_t len, int flags);
函数作用	实现磁盘上文件内容与共享内存区的内容一致
参数	addr:是调用mmap时返回的地址， len: 是映射区的大小 . flags: 映射区的特性, 可以是: • MS_SYNC要求回写完成后才返回 • MS_ASYNC发出回写请求后立即返回 • MS_INVALIDATE使用回写的内容更新该文件的其它映射
返回值	成功：0
	出错：-1

```
/*first.c          例7-12   P169    */
typedef struct{
    char name[4];
     int  age;
 }student;
main(int argc, char** argv) {
     int fd,i;      student *p_map;    char temp;
     fd=open("stu.txt",O_CREAT|O_RDWR|O_TRUNC,00777);
     lseek(fd,sizeof(student)*5-1,SEEK_SET);
     write(fd,"",1);
     p_map = (student*) mmap( NULL,sizeof(student)*10,PROT_READ|PROT_WRITE,  MAP_SHARED,fd,0 );
     if (p_map==(void *)-1){         perror("mmap failed\n");         exit(0);
     }
     close( fd );
     temp = 'a';
      for(i=0; i<10; i++)  {
          temp += 1;  //学生姓名
          memcpy( ( *(p_map+i) ).name, &temp,2 );    ( *(p_map+i) ).name[1]='\0';
          ( *(p_map+i) ).age = 20+i;  //学生年龄
     }
     sleep(10);
     munmap( p_map, sizeof(student)*10 );
}
```

```
/*------------second.c----------*/
typedef struct{
    char name[4];
    int  age;
}student;
main(int argc, char** argv)
{
    int fd,i;
    student *p_map;
    fd=open( "stu.txt",O_CREAT|O_RDWR,00777 );
    p_map = (student*)mmap(NULL,sizeof(student)*10,PROT_READ|PROT_WRITE,
                                        MAP_SHARED,fd,0);
    for(i = 0;i<10;i++)  {
         printf( "name: %s age %d;\n",(*(p_map+i)).name, (*(p_map+i)).age );
     }
     munmap( p_map,sizeof(student)*10 );
}
```

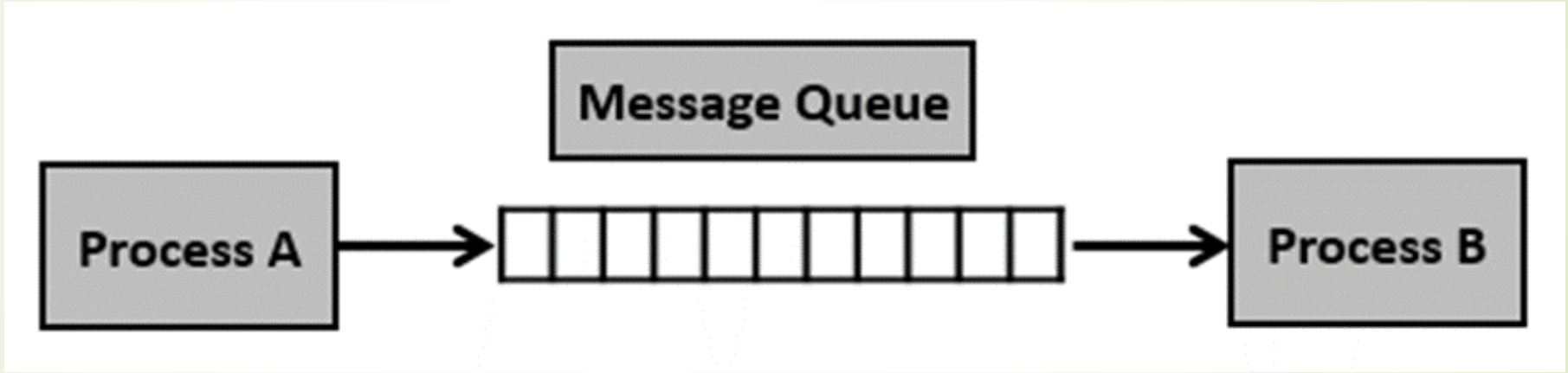
主要内容

- 7.1 进程间通信基本概念
- 7.3 System V信号量
- 7.4 POSIX信号量
- 7.5 共享内存
- 7.6 消息队列

消息队列

- 消息队列就是一个消息的链表即将消息看作一个记录，并且这个记录具有特定的格式以及特定的优先级
- 对消息队列有写权限的进程可以按照一定的规则添加新消息到队列的末尾
- 对消息队列有读权限的进程则可以从消息队列中读取消息

消息队列实现原理



消息队列的特点

- 发送方不必等待接收方去接收该消息就可不断发送数据，而接收方若未收到消息也无需等待。
- 这种方式实现了发送方和接收方之间的松耦合，发送方和接收方只需负责自己的发送和接收功能，无需等待另外一方，从而为应用程序提供了灵活性。

消息队列的使用过程

(1)创建消息队列

(2)发送数据到消息队列

(3)从消息队列中读取数据

(4)删除队列

消息队列的优点

- 消息队列与管道以及命名管道相比，具有更大的灵活性
  - 首先，它提供有格式字节流，有利于减少开发人员的工作量
  - 其次，消息具有类型，在实际应用中，可为不同类型的消息分配不同的优先级

SYSTEM V消息队列函数汇总

- 头文件： <sys/msg.h>
- IPC对象创建或打开函数： msgget
- IPC控制函数： msgctl
- IPC操作函数： msgsnd/msgrcv

消息队列的使用

所需头文件	#include<sys/msg.h>
函数原型	int msgget (key_t key, int msgflg);
函数作用	创建一个消息队列或取得一个已经存在的消息队列
参数	key：消息队列的键值 msgflg：创建消息队列的创建方式或权限。创建方式有： •IPC_CREAT，如果内存中不存在指定消息队列，则创建一个消息队列，否则取得该消息队列； •IPC_EXCL，消息队列不存在时，新的消息队列才会被创建，否则会产生错误
返回值	成功：返回消息队列的标示符
	出错：-1

函数原型	**int msgsnd(int msgid,struct msgbuf *msgp ,int msgsz, int msgflg)**
函数作用	往队列中发送一个消息
参数	**msgid：** 消息标识id，也就是msgget（）函数的返回值 **msgp：** 指向消息缓冲区的指针，该结构体为 **struct mymesg {** **long mtype;** /*消息类型*/ **char mtext[512];** /*消息文本*/ **}** **msgsz：** 消息文本的大小，不包含消息类型 **msgflg：** 可以设置为0，或者IPC_NOWAIT。如为IPC_NOWAIT，则当消息队列已满，则此消息将不写入消息队列，将返回到调用进程，如果没有指明（即为0），调用进程会被挂起，直到消息写入消息队列为止。
返回值	成功：0
	出错：-1

所需头文件	#include<sys/msg.h>
函数原型	int msgrcv(int msgid , struct msgbuf *msgp,int msgsz , long mtype,int msgflg)
函数作用	从消息队列中读走消息（即读完之后消息就从队列中消失）
参数	msgid：消息队列的id号； msgp：存储所读取消息的结构体指针； msgsz：消息的长度 mtype：从消息队列中读取的消息的类型。 如果为0，则会读取驻留消息队列时间最长的那一条消息，不论它是什么类型； msgflg：为0时表示该进程将一直阻塞，直到有消息可读； 还可设为IPC_NOWAIT，表示如果没有消息可读时立刻返回-1，否则进程被挂起。
返回值	成功：0 出错：-1

所需头文件	#include<sys/msg.h>
函数原型	int msgctl(int msgid, int cmd ,struct msgqid _ds *buf);
函数作用	消息队列控制系统调用
参数	msgid：消息队列的ID，即函数msgget()的返回值； cmd：消息队列的处理命令，主要包括如下几种类型： •IPC_RMID：从系统内核中删除消息队列，相当于命令“ipcrm -q id”； •IPC_STAT：获取消息队列的详细消息，包含权限、各种时间id等； •IPC_SET：设置消息队列的信息； •buf：存放消息队列状态的结构体指针。
返回值	成功：0
	出错：-1

```
/*msgexamp.h*/  例7-13 P173
#ifndef MSGQUE_EXAMP
#define MSGQUE_EXAMP
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/msg.h>
#include <sys/stat.h>
#define MAX_TEXT 512
#define MSG_KEY 335
struct my_msg_st   //消息结构体
{
      long my_msg_type;  //消息类型，其值分别为1，2，3代表不同交易消息
      char  text[MAX_TEXT];  //存放具体消息内容
};
#endif
```

```
/*msgsnd.c文件随机发送交易消息程序*/  #include "msgexamp.h“  #include<stdlib.h>
int main(){
        int index = 1;    struct my_msg_st some_data;  int msgid;  char buffer[BUFSIZ];
        msgid = msgget((key_t)MSG_KEY, IPC_CREAT|S_IRUSR|S_IWUSR);
        if(msgid == -1){
                perror("create message queue failed ");
         return -1;
        }
        srand((int)time(0));
        while(index<5)   {
                printf("[%d]Enter some text: less than %d\n",msgid,MAX_TEXT);
                fgets(buffer, BUFSIZ, stdin);
                some_data.my_msg_type = rand()%3+1;  //随机值
                printf("my msg_type=%ld\n",some_data.my_msg_type);
                strcpy(some_data.text, buffer);  //随机数字字符串
                 if (msgsnd(msgid, (void *)&some_data, sizeof(some_data), 0) == -1)
                {
                        fprintf(stderr, "msgsnd failed\n");
                        exit(-1);
                 }
   index++;
} exit(0);  }
```

```
/*msgrcv.c 接收指定类型消息的进程，头文件与上个程序一样 */    #include "msgexamp.h“
int main(int argc,char **argv) {
        int msgid;   int type;
        struct my_msg_st *my_data=(struct my_msg_st *)malloc(sizeof(struct my_msg_st));
        if (argc<2) {
                printf("USAGE msgexample msgtype\n");     return -1 ;
         }
        type=atoi(argv[1]);
        if ( type<0  | |  type>3 ) {
                printf("msgtype should be one of 1,2,3");     return -1;          }
        msgid = msgget((key_t)MSG_KEY, IPC_CREAT|S_IRUSR|S_IWUSR);
        if(msgid == -1){
                 perror("get message queue failed ");     return -1;   }
        while (1) {
                 if (msgrcv(msgid,(void *)my_data, sizeof(struct my_msg_st),(long)type,IPC_NOWAIT) !=-1) {
                        printf("The message type is:%ld\n",my_data->my_msg_type);
                        printf("The message content is:%s\n",my_data->text);
                }
                else if (ENOMSG==errno)       {
                        printf("there is no any message which is matched to message type \n");
                        break;                  }
        }
}
```

第7章 结束