

# 第九章 操作系统接口与应用开发

---

授课教师:

电子邮箱:



## 本章主要内容

- 9.1 系统调用的实现机制
- 9.2 Linux中基本的进程通讯工具
- 9.3 管道的建立和应用
- 9.4 信号量的建立和应用
- 9.5 消息队列
- 9.6 共享存储

## 9.1 系统调用的实现机制

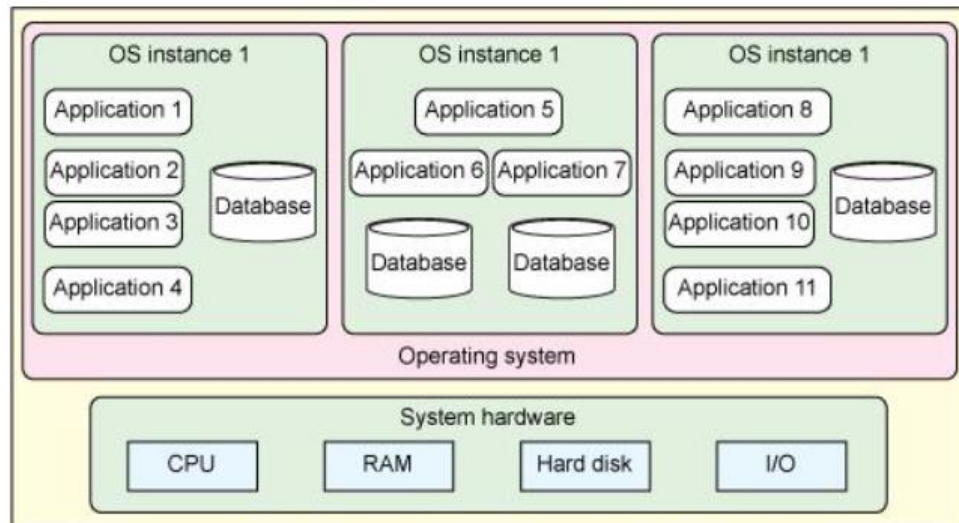
操作系统与用户接口

系统调用

系统调用的类型

系统调用与库函数之间的关系

API、C库、系统调用





# 操作系统与用户接口

1. **命令接口**：方便用户直接或间接控制自己的作业，分为：联机用户接口（交互式方式运行的命令）与脱机用户接口（批处理用户接口）；
2. **程序接口**：为用户程序在执行中访问系统资源而设置，是用户程序取得操作系统服务的唯一途径。它由一组系统调用组成。每一个系统调用都是一个能完成特定功能的子程序。
3. **图形接口**：采用了图形化的操作界面，用非常容易识别的各种图标来将系统的各项功能、各种应用程序和文件，直观、逼真地表示出来。



# 系统调用

- 系统调用在用户空间进程和硬件设备之间添加了一个中间层，该层的主要作用有两个：
  1. 为用户空间提供了一种硬件的抽象界面，例如，当需要读文件时，应用程序可以不管磁盘类型和介质，甚至不用去管文件所在的文件系统到底是哪种类型；
  2. 系统调用保证了系统的稳定和安全。
- 各种版本的UNIX系统都提供了定义明确、数量有限、可直接进入内核的入口点，这些入口点被称为系统调用（system calls）
- Linux根据版本的不同有240到260个系统调用





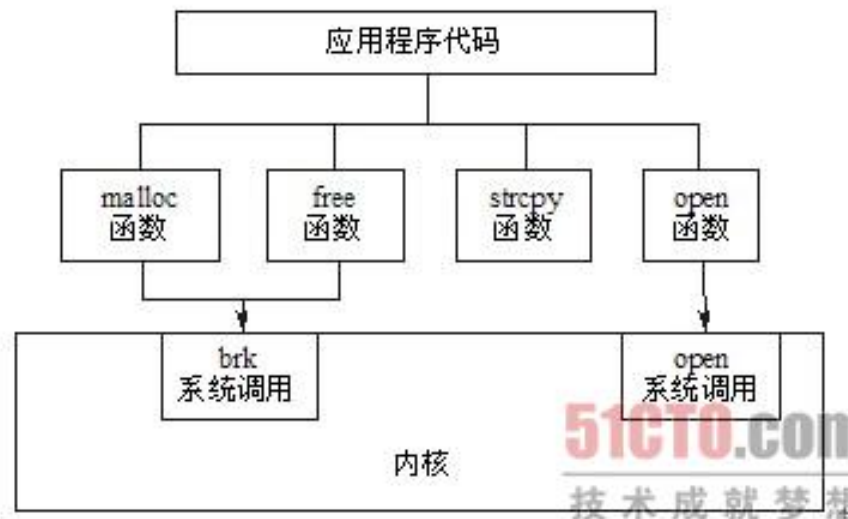
# 系统调用的类型

- 系统调用接口在《UNIX程序员手册》的第2部分中说明
  - 进程控制类系统调用：如创建进程、设置或获取进程属性等；
  - 文件操作类系统调用：如创建、删除、修改文件；
  - 设备管理类系统调用：打开、关闭和操作设备；
  - 通信类系统调用：创建进程间的通信连接，发送、接收消息，或其他的通信方式；
  - 信息维护类系统调用：在用户程序和OS之间传递信息。例如，系统向用户程序传送当前时间、日期、操作系统版本号等。



# 系统调用与库函数之间的关系

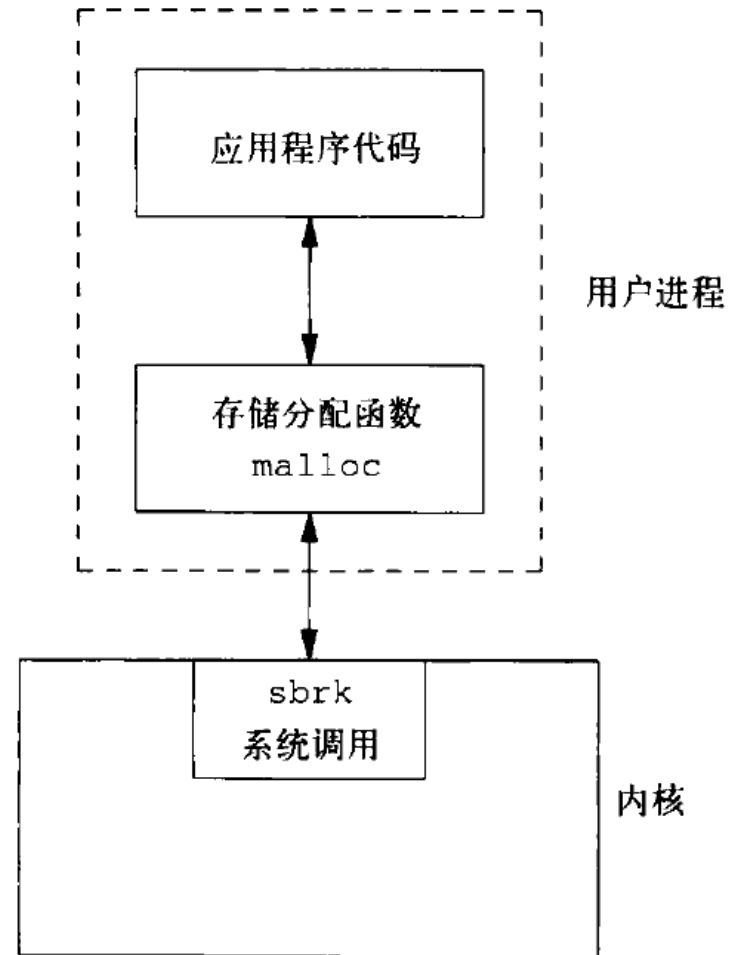
- UNIX/Linux为每个系统调用在标准C库中设置一个具有相同名字的函数；应用程序通过调用C库函数来使用系统调用
- 系统调用和C库函数之间并不是一一对应的关系。可能几个不同的函数会调用到同一个系统调用，比如malloc函数和free函数都是通过sbrk系统调用来扩大或缩小进程的堆栈空间；execl、execlp、execle、execv、execvp和execve函数都是通过execve系统调用来执行一个可执行文件。





# 系统调用与库函数之间的关系

- 系统调用通常提供一种内核功能的最小接口，而库函数通常提供比较复杂的功能
- 并非所有的库函数都会调用系统调用，例如，printf函数会调用write系统调用以输出一个字符串，但strcpy和atoi函数不使用任何系统调用



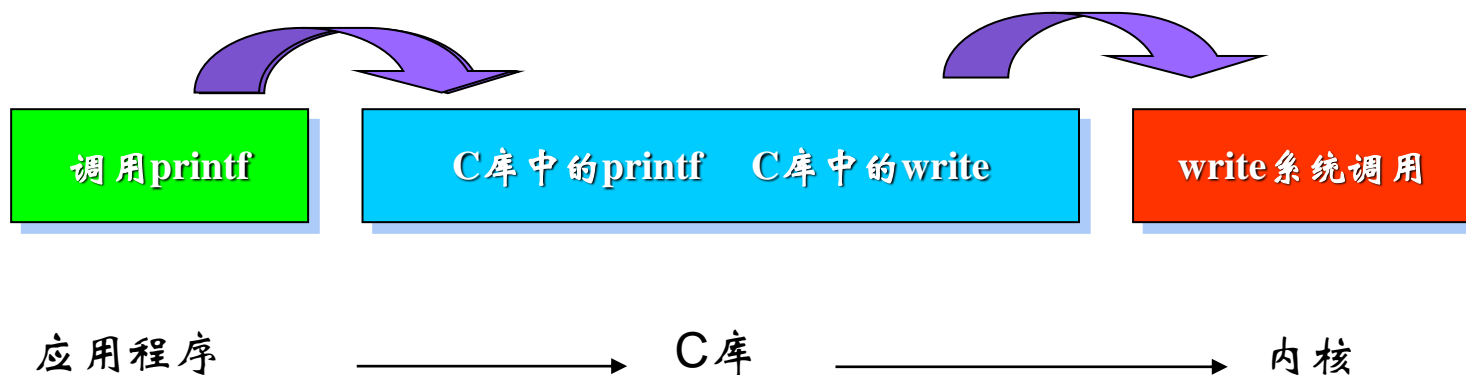




# API、C库、系统调用

- 一般而言，应用程序通过API而不是直接通过系统调用来编程。
- Linux的系统调用像大多数UNIX系统一样，作为C库的一部分提供。C库实现了UNIX系统的主要API，包括标准的C库函数和系统调用。

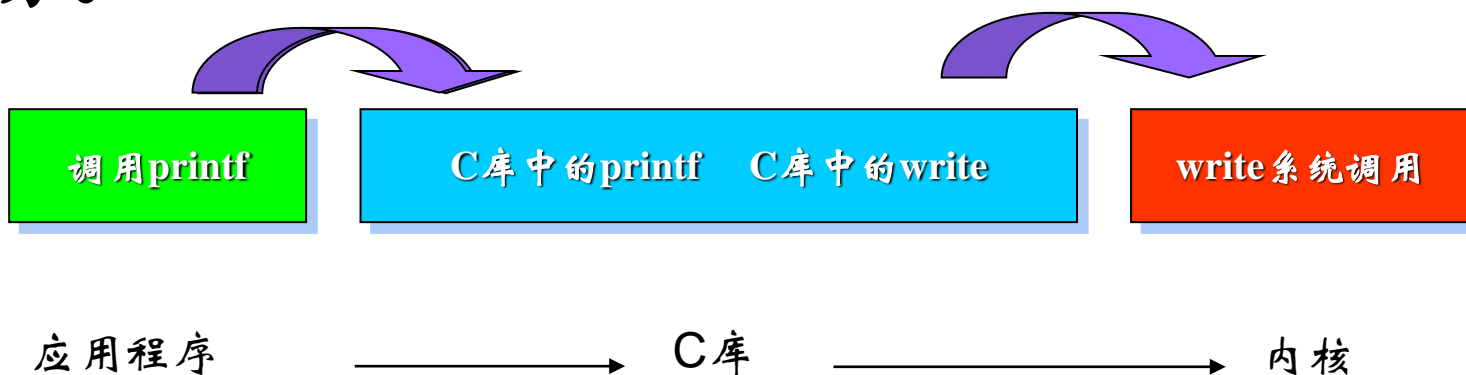
参考： [/usr/include/asm/unistd.h](#)





# 系统调用的实现

- 系统调用的实现与一般过程调用的实现相比，两者间有很大差异。
- 对于系统调用，控制是由原来的用户态转换为系统态，这是借助于中断和陷入机制来完成的，在该机制中包括中断和陷入硬件机构及中断与陷入处理程序两部分。





# 系统调用的实现

用户态转换为系统态，借助中断和陷入机制来完成

◆ 中断是指CPU对系统发生某事件时的这样一种响应：CPU暂停正在执行的程序，在保留现场后自动地转去执行该事件的中断处理程序；执行完后，再返回到原程序的断点处继续执行。

中断分为外中断和内中断

(1) 所谓外中断是指由于外部设备事件所引起的中断。

(2) 内中断则是指由于CPU内部事件所引起的中断。

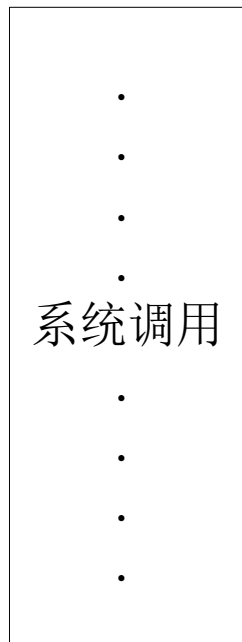
如程序出错（地址越界）、电源故障等。内中断（trap）被译为“捕获”或“陷入”。



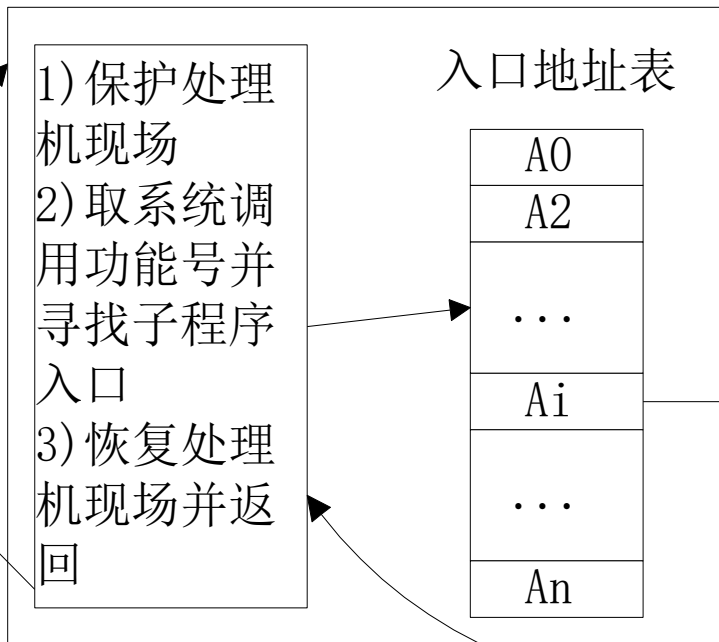
# 系统调用的实现



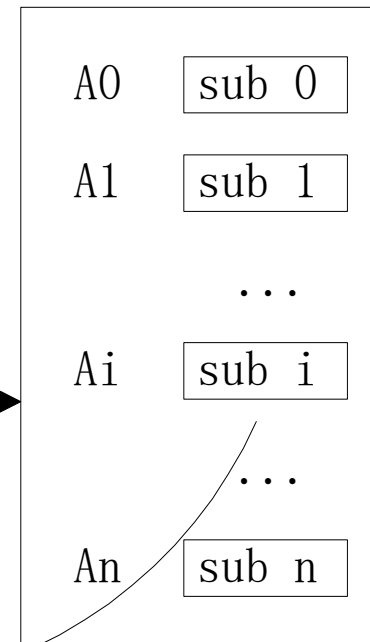
用户程序



陷入处理机构



系统子程序



陷入指令

**int 0x80**



## int 0x80指令

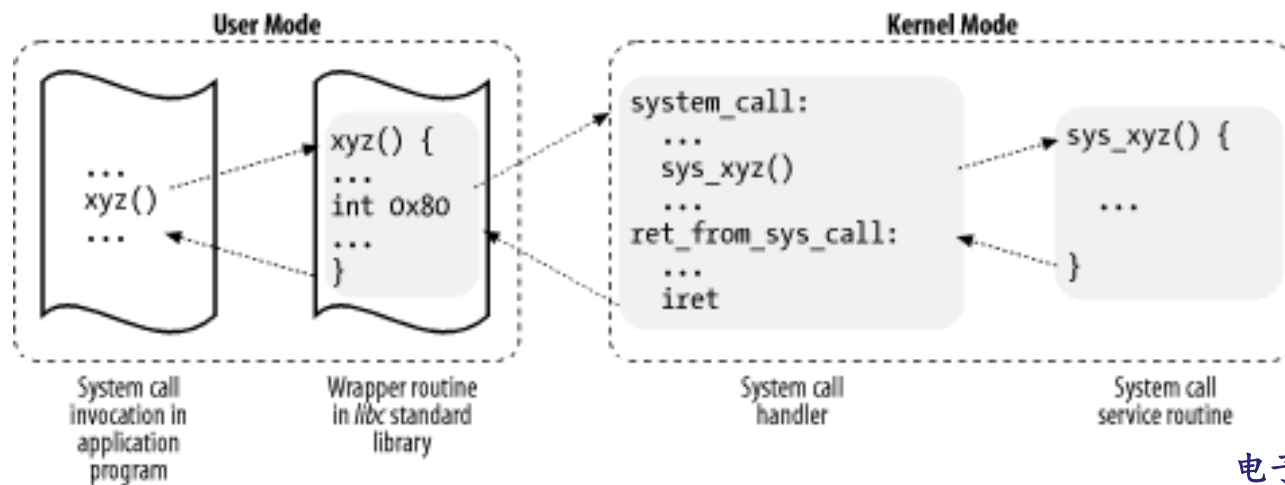
- Linux中实现系统调用利用了i386体系结构中的软件中断。即调用了int \$0x80汇编指令。
- 这条汇编指令将产生向量为128的编程异常，CPU便被切换到内核态执行内核函数，转到了系统调用处理程序的入口：system\_call()。
- int \$0x80指令将用户态的执行模式转变为内核态，并将控制权交给系统调用过程的起点system\_call()处理函数。





## system\_call()函数

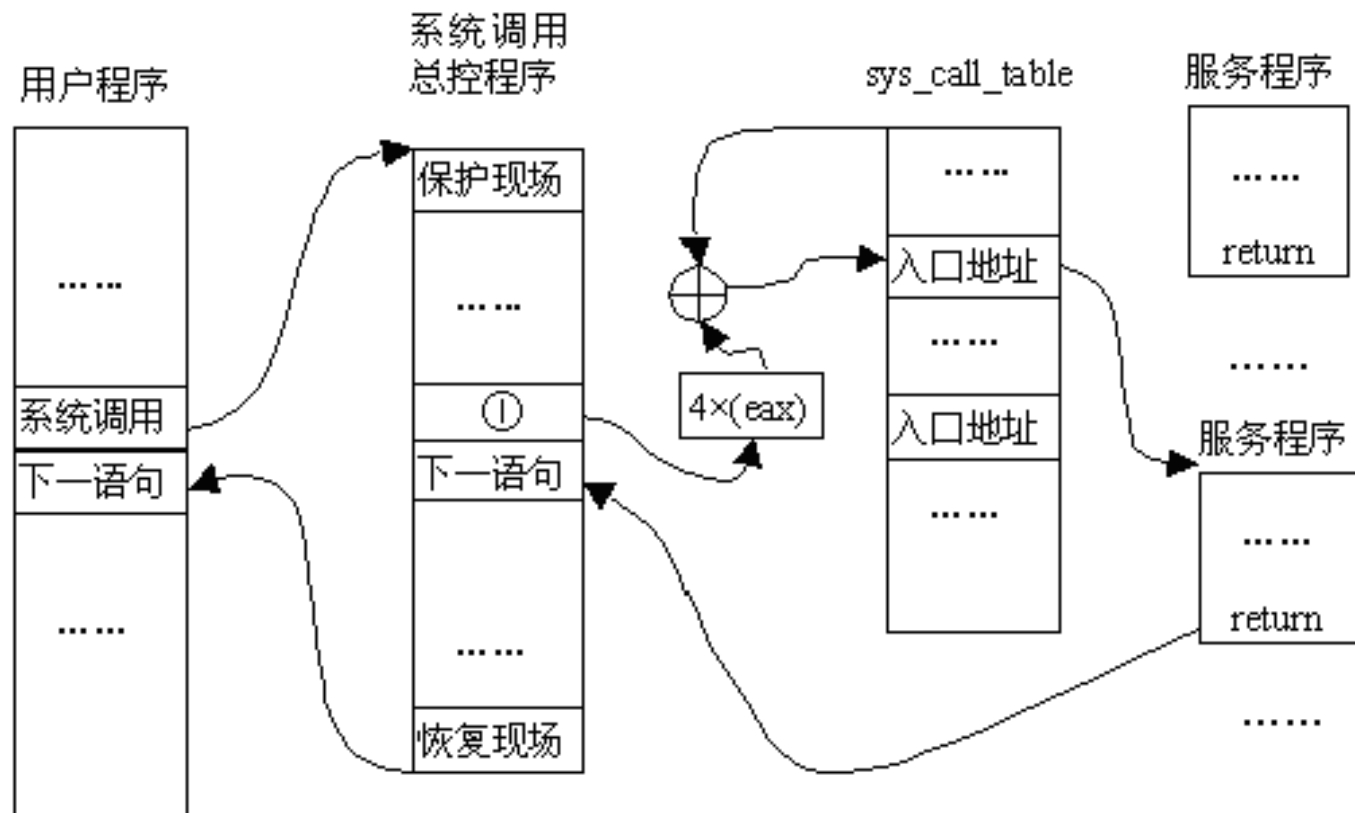
- ◆ system\_call()检查系统调用号，该号码告诉内核进程请求哪种服务。
- ◆ 内核进程查看系统调用表(sys\_call\_table)找到所调用的内核函数入口地址。
- ◆ 接着调用相应的函数，在返回后做一些系统检查，最后返回到进程。





# 系统调用的实现

## linux系统调用过程

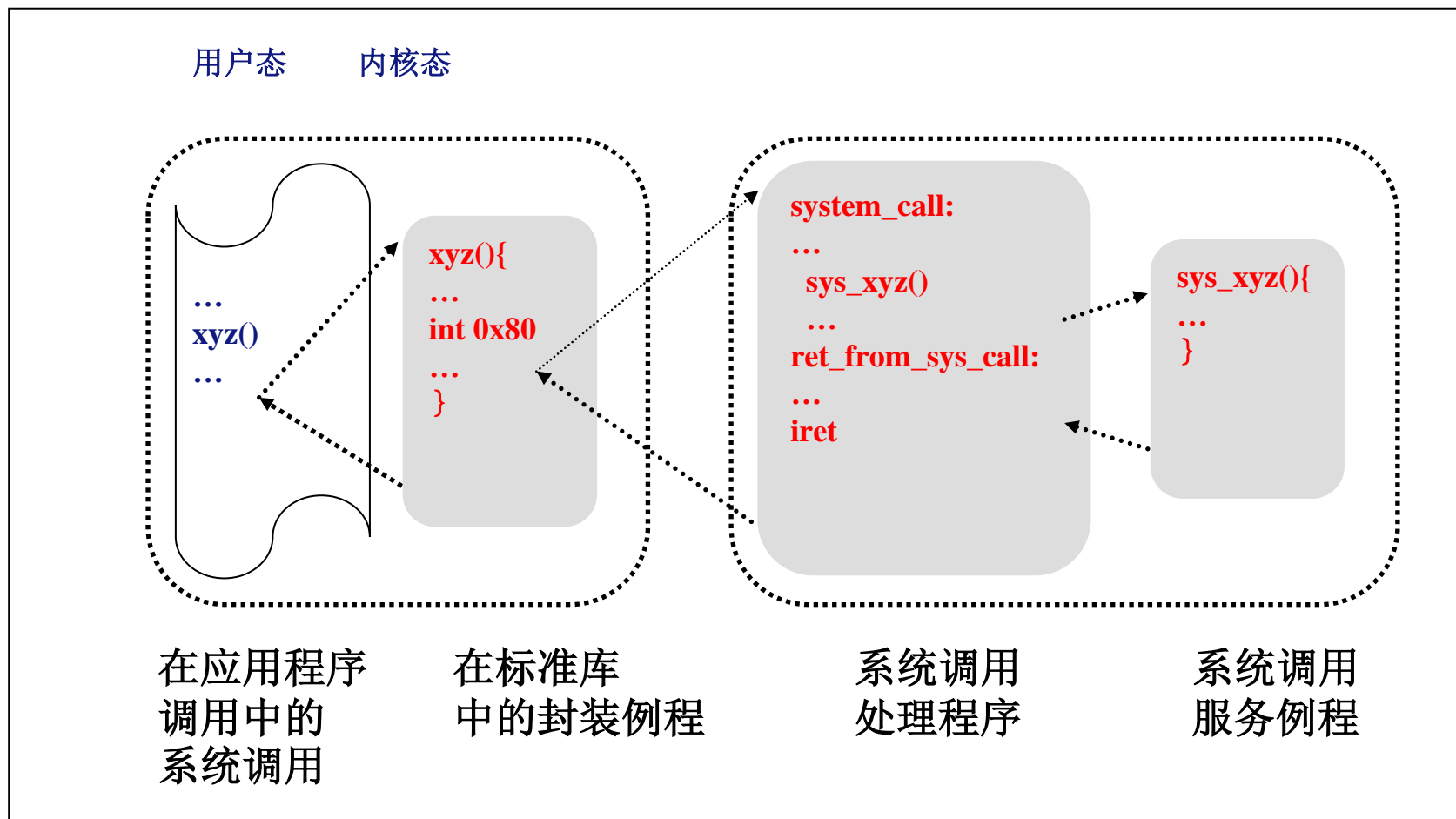


注①: 此处语句为: `call *SYMBOL_NAME(sys_call_table)(,%eax,4);` `eax` 中为系统调用号



# 系统调用的实现

## linux系统调用过程



## 9.2 Linux中基本的 进程通讯工具

---

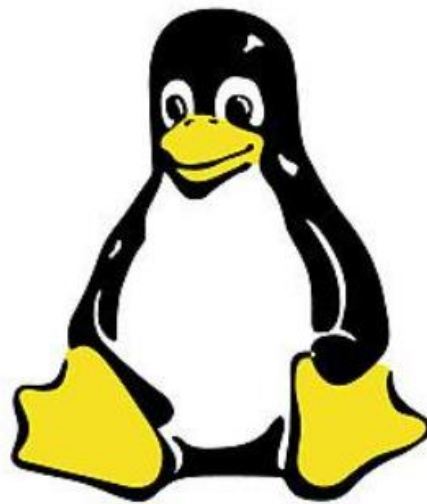
Linux/Unix系统中进程间通信的  
基本概念介绍

管道是什么、管道的特质、局限性

信号灯的作用、类型

消息队列的基本概念

共享存储的基本概念、共享存储  
的例子、结构





# ? 问题

- 什么是管道通信？主要应用于什么场景？
- 信号灯(量) 通信主要是为了解决什么实际问题？
- 什么是消息队列？
- 什么是共享存储？
- 进程之间在什么时间需要通信？进程之间通信的本质是什么？
- 如何归纳实现进程通信的几大方式？





# 进程间的通讯

- 进程用户空间是相互独立的，一般而言是不能相互访问的，但很多情况下进程间需要互相通信来完成系统的某项功能。进程通过与内核及进程之间的互相通信来协调它们的行为。
- Linux中常用的通信方式：
  - 管道、命名管道(FIFO)
  - XSI IPC的3种形式的IPC(消息队列、信号量和共享存储)
  - POSIX提供的替代信号量机制



- 信号量的概念在system v (Unix版本中的一支) 和posix 中都有。
- system v版本的信号量用于实现进程间的通信，而posix版本的信号量主要用于实现线程之间的通信，两者的主要区别在于信号量和共享内存。



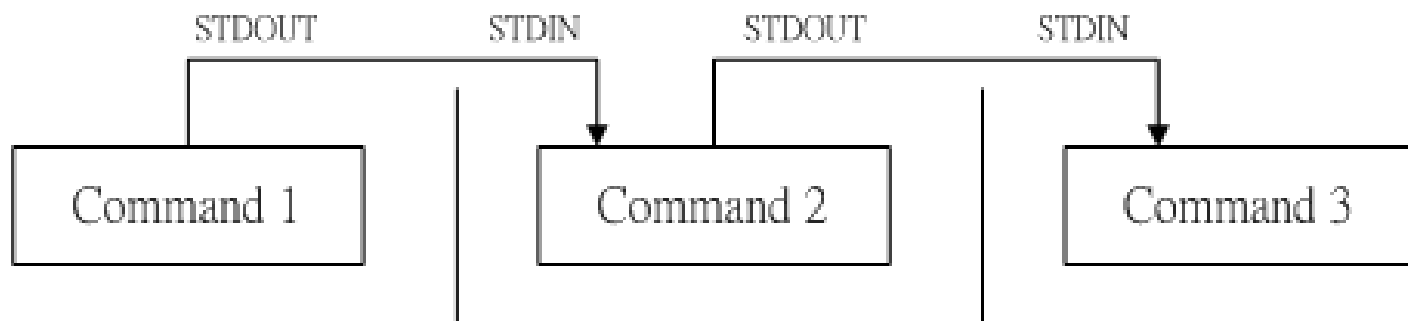
# 进程间的通讯

- system v版本的信号量是跟随于内核存在的，它的接口是：semget, semctl, semop
- posix版本的信号量效率较高，支持无命名的信号量和有命名的信号量。
  - (1) 有命名的信号量一般用于进程同步，使用文件进行关联，该部分的信号量是随着内核而存在的，主要接口有：sem\_open, sem\_close, sem\_unlink.
  - (2) 无命名的信号量一般用于线程的同步，当进程终止时，它也就消亡了，主要接口有：sem\_init, sem\_destory, sem\_wait, sem\_post.



# 管道

- 把一个程序的输出直接连接到另一个程序的输入，即把前一个命令的结果当成后一个命令的输入。
- shell中，管道命令使用“|”作为界定符号，它表现出来的形式将前面每一个进程的输出（stdout）直接作为下一个进程的输入（stdin）。管道命令仅能处理standard output，对于standard error output会予以忽略。





# 管道

- 是由内核管理的一个缓冲区，相当于我们放入内存中的一个纸条。一个缓冲区不需要很大，它被设计成为环形的数据结构，以便管道可以被循环利用。
- 当管道中没有信息的话，从管道中读取的进程会等待，直到另一端的进程放入信息。当管道被放满信息的时候，尝试放入信息的进程会堵塞，直到另一端的进程取出信息。
- 当两个进程都终结的时候，管道也自动消失。
- 管道是一种最基本的IPC机制，作用于有血缘关系的进程之间，完成数据传递。连接在管道的所有进程被称为**进程组**。





# 管道

- 调用pipe系统函数即可创建一个管道，**管道有如下特质：**
  - (1) 其本质是一个文件(实为内核缓冲区)
  - (2) 由两个文件描述符引用，一个表示读出端，一个表示写入端。
  - (3) 规定数据从管道的写入端流入管道，从读出端流出。
- **管道的局限性（重要）：**
  - (1) 数据能自己读不能自己写。
  - (2) 数据一旦被读走，便不在管道中存在，不可反复读取。
  - (3) 由于管道采用**半双工通信方式**。因此，**数据只能在一个方向上流动。**
  - (4) **只能在有公共祖先的进程间使用管道。**



# 管道

在Linux中，管道是一种使用非常频繁的通信机制。从本质上说，管道也是一种文件，但它又和一般的文件有所不同，管道可以克服使用文件进行通信的两个问题，具体表现为：

①限制管道的大小。实际上，管道是一个固定大小的缓冲区。在Linux中，该缓冲区的大小为1页，即4K字节，使得它的大小不像文件那样不加检验地增长。使用单个固定缓冲区也会带来问题，比如在写管道时可能写满，当这种情况发生时，随后对管道的write()调用将默认被阻塞，等待某些数据被读取，以便腾出足够的空间供write()调用。



# 管道

②读取进程也可能工作得比写进程快。当所有当前进程数据已被读取时，管道变空。当这种情况发生时，一个随后的read()调用将默认地被阻塞，等待某些数据被写入，这解决了read()调用返回文件结束的问题。

- 注意：从管道读数据是一次性操作，数据一旦被读，它就从管道中被抛弃，释放空间以便写更多的数据。



# 信号灯

- 信号机制是类UNIX系统中的一种重要的进程间通信手段之一。我们经常使用信号来向一个进程发送一个简短的消息。
- **信号灯 (信号量) 主要提供对进程间共享资源访问控制机制。**信号灯本质上是一个计数器，用于协调多个进程（包括但不限于父子进程）对共享数据对象的读/写。它不以传送数据为目的，主要是用来保护共享资源（信号量、消息队列、socket连接等），保证共享资源在一个时刻只有一个进程独享。
- 信号量是一个特殊的变量，只允许进程对它进行等待信号和发送信号操作。最简单的信号量是取值0和1的二元信号量，这是信号量最常见的形式。



# 信号灯

- 除了用于访问控制外，还可用于进程同步。信号灯有以下两种类型：
  - (1) 二值信号灯：最简单的信号灯形式，信号灯的值只能取0或1，类似于互斥锁。
  - (2) 计数信号灯：信号灯的值可以取任意非负值（当然受内核本身的约束）
- 注：二值信号灯能够实现互斥锁的功能，但两者的关注内容不同。信号灯强调共享资源，只要共享资源可用，其他进程同样可以修改信号灯的值；互斥锁更强调进程，占用资源的进程使用完资源后，必须由进程本身来解锁。





# 信号灯

## ▪ 信号灯的描述

```
struct sem {  
    ushort_t semval; //信号量当前值  
    short     sempid; //最后一次操作该信号量的进程  
    ushort_t semncnt; //有几个线程在等待semval变为大于  
                        当前值（等待的进程/线程被堵塞）  
    ushort_t semzcnt; //有几个线程在等待semval变为0，  
                        （等待的进程/线程被堵塞）  
};
```

```
int semop(int semid, struct sembuf *sops, unsigned nsops); //信号灯操作函数  
int semtimedop(int semid, struct sembuf *sops, unsigned nsops, struct  
timespec *timeout); //信号灯操作函数
```

形参sops指向的数组中的各项操作，会按照数组顺序，并以原子的方式执行



# 消息队列

- Linux提供了一系列消息队列的函数接口来让我们方便地使用它来实现进程间的通信。它的用法与其他两个System V PIC机制，即信号量和共享内存相似。
- 消息队列是消息的链接表，存储在内核中，由消息队列标识符标识。
- 我们通常把消息队列简称为队列，其标识符简称为队列ID。
- `msgget`用于创建一个新队列或打开一个现有队列。`msgsnd`将新消息添加到队列尾端。
- 每个消息包含一个正的长整型类型的字段、一个非负的长度以及实际数据字节数（对应于长度），所有这些都在将消息添加到队列时，传送给`msgsnd`。`msgrcv`用于从队列中取消息。我们并不一定要以先进先出次序取消息，也可以按消息的类型字段取消息。



# 消息队列

- 每个队列都有一个msqid\_ds结构与其相关联:

```
struct msqid_ds{  
    struct ipc__perm  msg__perm;  
    msgqnum_t  msg_qnum; /* of messages onqueue*/  
    msglen_t  msg_qbytes; /* max of bytes onqueue*/  
    pid_t  msg_lspid; /*pid of last msgsnd()*/  
    pid_t  msg_lrpid; /*pid of last msgrcv()*/  
    time_t  msg_stime; /*last msgsnd() time*/  
    time_t  msg_rtime; /*last msgrcv() time*/  
    time_t  msg_ctime;
```



# 消息队列

- 此结构定义了队列的当前状态。结构中所示的各成员是由Single UNIX Specification定义的。具体实现可能包括标准中没有定义的另一一些字段。
- 下图列出了一些影响消息队列的系统限制。

说明	典型值			
	FreeBSD8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
可发送最长消息的字节数	16348	8192	不支持	2048
一个特定队列的最大字节数	2048	16384	不支持	4096
系统中最大消息队列数	40	16	不支持	50
系统中最大消息数	40	导出的	不支持	40



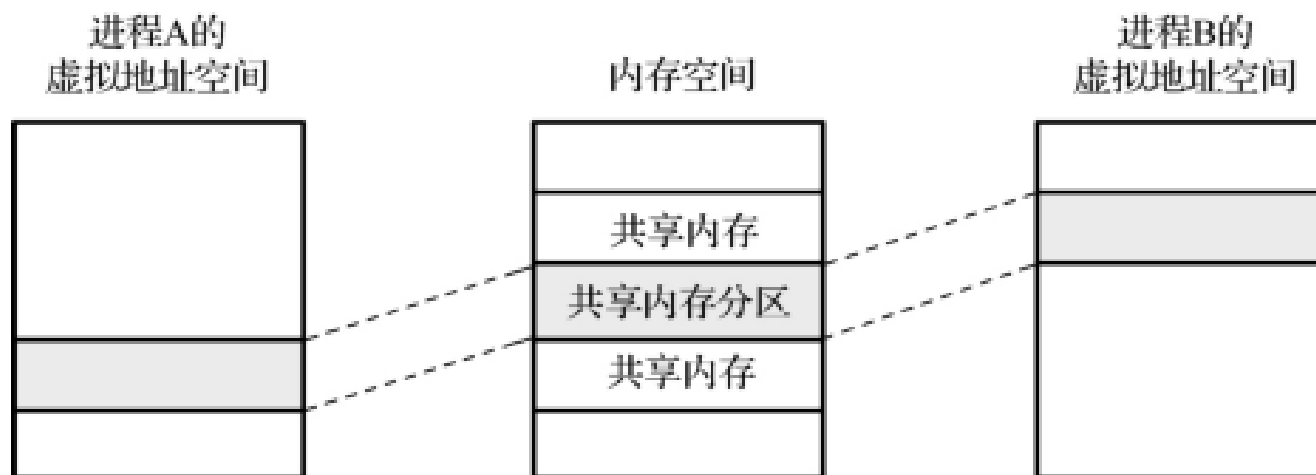
# 共享存储

- 共享存储允许两个或多个进程共享一个给定的存储区。通信时，发送进程将需要交换的信息写入该共享内存分区中，接收进程从该共享内存分区中读取信息，从而实现进程之间的通信。
- 因为数据不需要在客户进程和服务进程之间复制，所以这是最快的一种IPC。
- 每一个进程都有自己的内存空间，一个进程如果访问另一个进程的内存空间就很容易引起错误，但共享内存则可以由多个进程同时进行读／写操作。如果需要读／写同步操作，则可以使用信号量／互斥量来处理。



# 共享存储

- 共享内存通信方式示意如下图所示。



- 若服务器进程正在将数据放入共享存储区，则在它做完这一操作之前，客户进程不应当去取这些数据。
- 通常，信号量用于同步共享存储访问。（不过正如前节最后部分所述，也可以用记录锁或互斥量。）





## 共享存储的例子

- 桌面进程在创建完共享内存后，通过调用 `CreateStockObject` 将字库、预定义的GDI对象复制到共享内存中，客户端进程通过 `Attach` 这块内存，就可以读其中的字库与系统预置的GDI对象。这样，只由桌面进程一次创建就可以多个用户进程共享使用，不必再进程之间传递这些消息，降低了进程间资源的消耗，有利于提高系统效率。



# 共享存储的结构

- 内核为每个共享存储段维护着一个结构，该结构至少要为每个共享存储段包含以下成员：

```
struct shmid_ds{  
    struct ipc_perm shm_perm;  
    size_t  shm_segsz; /*size of segment in bytes*/  
    pid_t  shm_lpid; /*pid of last shmop()*/  
    pid_t  shm_cpid; /*pid of creator*/  
    shmatt_t shm_nattch; /*number of current attaches*/  
    time_t  shm_atime; /*last-attach time*/  
    time_t  shm_dtime; /*last-detach time*/  
    time_t  shm_ctime; /*last-change time */  
}
```



# 共享存储

- （按照支持共享存储段的需要，每种实现会增加其他结构成员。）
- 调用的第一个函数通常是shmget,获得一个共享存储标识符。

```
#include<sys/shm.h>
```

```
int shmget(key_t key, size_t size, int flag);
```

- 返回值：若成功，返回共享存储ID,若出错，返回-1

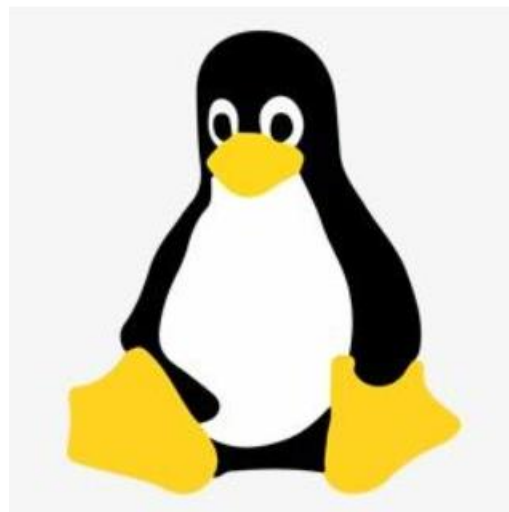
## 9.3 管道的建立和应用

---

管道的使用

管道中数据的读入和撤销

管道中数据的读写入和读取





## ■ 如何使用管道？

管道机制的主体是系统调用`pipe()`,但是由`pipe()`所建立的管道的两端都在同一进程中，所以必须在`fork`的配合下，才能在父子进程之间或者两个子进程之间建立起进程间的通信管道。

## ■ (1) Linux管道的使用

当shell命令对`ls | more`语句进行解释时，实际上要执行以下操作：

- ①调用`pipe()`系统调用。让我们假设`pipe()`返回文件描述符3(管道的读通道)和4(管道的写通道)。
- ②两次调用`fork()`系统调用。
- ③两次调用`close()`系统调用来释放文件描述符3和4。



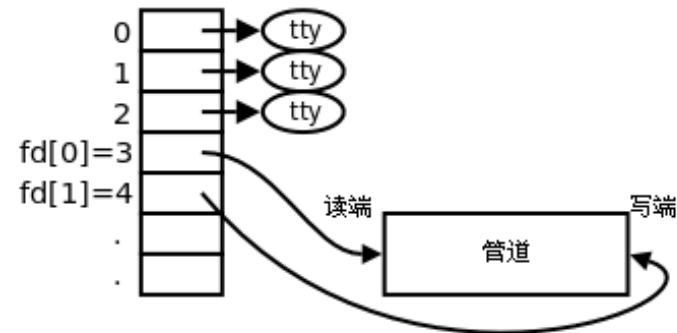
# 管道的使用

①调用pipe()系统调用。让我们假设pipe()返回文件描述符3(管道的读通道)和4(管道的写通道)。

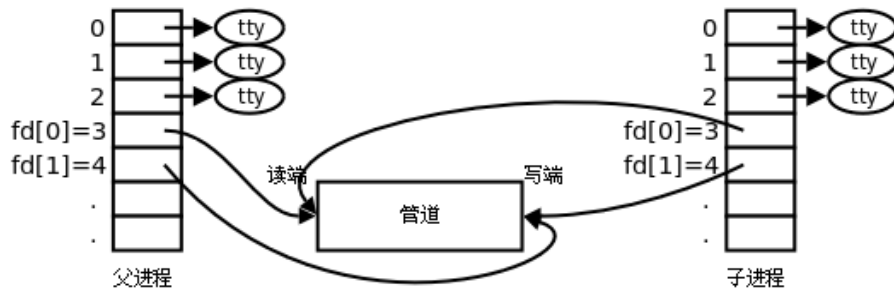
②两次调用fork()系统调用。

③两次调用close()系统调用来释放文件描述符3和4。

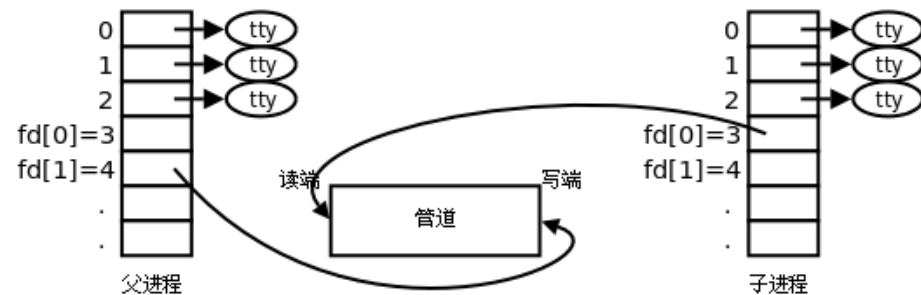
父进程创建管道



父进程fork出子进程



父进程关闭fd[0]，子进程关闭fd[1]







# 管道的使用

## ls | more

- 第一个子进程必须执行ls程序，它执行以下操作：
  - ① 调用`dup2(4,1)`把文件描述符4拷贝到文件描述符1。  
从现在开始，文件描述符1就代表该管道的写通道。
  - ② 两次调用`close()`系统调用来释放文件描述符3和4。
  - ③ 调用`execve()`系统调用来执行ls程序。缺省情况下，这个程序要把自己的输出写到文件描述符为1的那个文件(标准输出)中，也就是说，写入管道中。



## ls | more

■ 第二个子进程必须执行more程序；因此，该进程执行以下操作：

- ① 调用`dup2(3,0)`把文件描述符3拷贝到文件描述符0。从现在开始，文件描述符0就代表管道的读通道。
- ② 两次调用`close()`系统调用来释放文件描述符3和4。
- ③ 调用`execve()`系统调用来执行more程序。缺省情况下，这个程序要从文件描述符为0那个文件(标准输入)中读取输入，也就是说，从管道中读取输入。



# 管道的创建和撤销

## ■ (2) 管道的创建

```
#include <unistd.h>
```

```
int pipe(int fd[2]);
```

- fd为file descriptors的缩写，其为一个二元数组，用于存放pipe函数所创建管道的两个文件描述符
  - fd[0]存放管道读取端的文件描述符
  - fd[1]用于存放管道写入端的文件描述符
- 函数创建的管道的两端处于一个进程中间，在实际应用中并没有太大意义，因此，一个进程在由pipe()创建管道后，一般再fork一个子进程，需要时调用exec函数族使子进程执行所需程序。然后通过管道实现父子进程间的通信。



# 管道的创建和撤销

- (3) 有名管道的创建:

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

```
#include <sys/stat.h>
```

```
int mkfifo(const char* pathname, mode_t mode);
```

- 有名管道的创建: (有名管道)命名管道也被称为FIFO文件, 它是一种特殊类型的文件, **它在文件系统中以文件名的形式存在**, 但是它的行为却和之前所讲的没有名字的管道(匿名管道)类似。
- 命名管道主要用来解决不相关进程间的通信问题。



# 管道的创建和撤销

```
int mkfifo(const char* pathname, mode_t mode);
```

- 函数的第一个参数是一个普通的路径名，也就是创建后FIFO的名字。
- 第二个参数与打开普通文件的open()函数中的mode参数相同。
- 如果mkfifo的第一个参数是一个已经存在的路径名时，会返回EEXIST错误，所以一般典型的调用代码首先会检查是否返回该错误，如果确实返回该错误，那么只要调用打开FIFO的函数就可以了。一般文件的I/O函数都可以用于FIFO，如close、read、write等等。





# 管道的创建和撤销

认清管道和有名管道的读写规则是在程序中应用它们的关键。

- (4) 管道的读写规则：

如果管道的写入端不存在，则认为已经读到了数据的末尾，读函数返回的读出字节数为0，

当管道的写端存在时，如果请求的字节数目大于PIPE\_BUF，则返回管道中现有的数据字节数，如果请求的字节数目不大于PIPE\_BUF，则返回管道中现有数据字节数。

向管道中写入数据时，linux将不保证写入的原子性，管道缓冲区一有空闲区域，写进程就会试图向管道写入数据。如果读进程不读管道缓冲区中的数据，那么写操作将一直阻塞。





# 管道中数据的写入和读取

## ■ 读数据:

从管道中读取数据的进程发出一个`read()`系统调用，为管道的读出端指定一个文件描述符。内核最终调用与这个文件描述符相关的文件操作表中所找到的`read`方法。在管道的情况下，`read`方法在`read_pipe_fops`表中的表项指向`pipe_read()`函数。

这个系统调用可能以两种方式阻塞当前进程:

- \* 当系统调用开始时管道缓冲区为空。
- \* 管道缓冲区没有包含所有请求的字节，写进程在等待缓冲区的空间时曾被置为睡眠。

注意，读操作可以是非阻塞的。在这种情况下，只要所有可用的字节(即使是0个)一被拷贝到用户地址空间中，读操作就完成。还要注意，只有在管道为空而且当前没有进程正在使用与管道的写通道相关的文件对象时，`read()`系统调用才会返回0。



# 管道中数据的写入和读取

## ■ 写数据：

- 希望向管道中写入数据的进程发出一个`write()`系统调用，为管道的写入端指定一个文件描述符。内核通过调用适当文件对象的`write`方法来满足这个请求；`write_pipe_fops`表中相应的项指向`pipe_write()`函数。
- 如果管道没有读进程(也就是说，如果管道的索引节点对象的`readers`字段的值是0)，那么任何对管道执行的写操作都会失败。在这种情况下，内核会向写进程发送一个`SIGPIPE`信号，并停止`write()`系统调用，使其返回一个`-EPIPE`错误码，这个错误码就是我们熟悉的“Brokenpipe(损坏的管道)”消息。



# 管道中数据的写入和读取

- 管道实现的源代码在fs/pipe.c中，在pipe.c中有很多函数，其中有两个函数比较重要
  - 管道读函数pipe\_read()
  - 管道写函数pipe\_wrtie()
- 管道写函数通过将字节复制到VFS索引节点指向的物理内存而写入数据，而管道读函数则通过复制物理内存中的字节而读出数据。当然，内核必须利用一定的机制同步对管道的访问，为此，内核使用了锁、等待队列和信号。



# 管道实例1

```
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
void sys_err(const char *str) {
    perror(str);
    exit(1);
}
int main(void) {
    pid_t pid;
    char buf[1024];
    int fd[2];
    char *p = "test for pipe\n";
    if (pipe(fd) == -1)
        sys_err("pipe");
```

```
    pid = fork();
    if (pid < 0) {
        sys_err("fork err");
    } else if (pid == 0) {
        close(fd[1]);
        int len = read(fd[0], buf, sizeof(buf));
        write(STDOUT_FILENO, buf, len);
        close(fd[0]);
    } else {
        close(fd[0]);
        write(fd[1], p, strlen(p));
        wait(NULL);
        close(fd[1]);
    }
    return 0;
}
```

## 管道实例2

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main(void) {
    pid_t pid;
    int fd[2];
    pipe(fd);
    pid = fork();
    if (pid == 0) { //child
        close(fd[1]); // 子进程从管道中读数据, 关闭写端
        dup2(fd[0], STDIN_FILENO); // 让wc从管道中读取数据
        execlp("wc", "wc", "-l", NULL); //wc 命令默认从标准读入取数据
    } else {
        close(fd[0]); // 父进程向管道中写数据, 关闭读端
        dup2(fd[1], STDOUT_FILENO); // 将ls的结果写入管道中
        execlp("ls", "ls", NULL); //ls 输出结果默认对应屏幕
    }
    return 0;
}
```

dup2都可用来复制一个现存的文件描述符, 使两个文件描述符指向同一个file结构体。



## 课堂练习

管道通信以（）进行写入和写出？

- A. 消息为单位
- B. 自然字符流
- C. 文件
- D. 报文

下列关于管道(pipe)通信叙述中，正确的是？

- A. 一个管道可实现双向数据传输
- B. 管道的容量仅受磁盘容量大小的限制
- C. 进程对管道进行读操作和写操作都可能被阻塞
- D. 一个管道只能有一个读进程或者一个写进程对其操作



## 9.4 信号量的建立和应用

二值信号灯的基本概念

信号灯可以解决的问题





# 信号灯

- 信号灯主要提供对进程间共享资源访问控制机制。相当于内存中的标志，进程可以根据它判定是否能够访问某些共享资源，同时，进程也可以修改该标志。除了用于访问控制外，还可用于进程同步。
- 简单来说：信号灯(semaphore)是一种用于提供不同进程间或一个给定进程的不同线程的同步手段的原语。



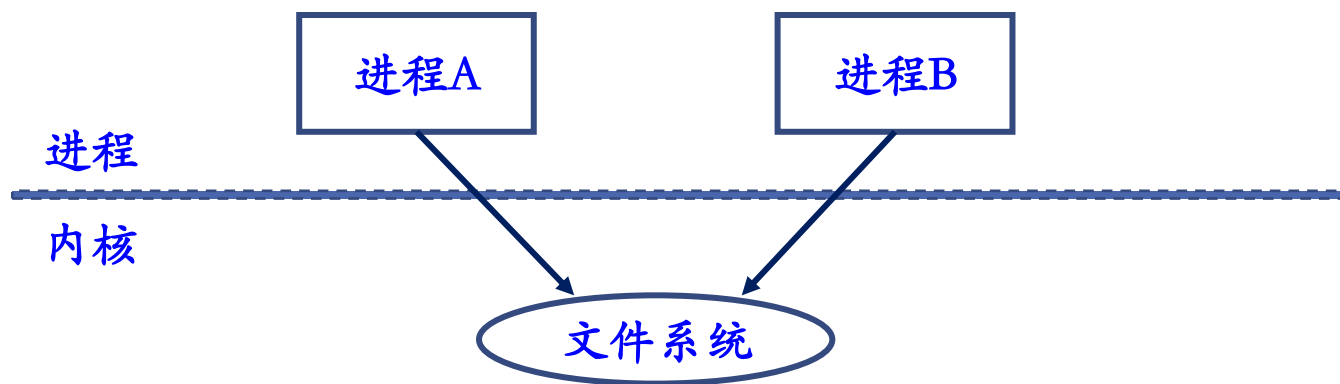
# 信号灯

- 信号灯有以下三种类型（另外一种分类方式）：
  - **Posix有名信号灯**：使用Posix IPC名字标识，可用于进程或进程间的同步
  - **Posix基于内存的信号灯**：存放在共享内存区，可用于进程或线程间的同步
  - **System V信号灯**：在内核中维护，可用于进程或线程间的同步
- 按值来分，信号灯有两种类型：
  - **二值信号灯**：最简单的信号灯形式，信号灯的值只能取0或1，类似于互斥锁。
  - **计算信号灯**：信号灯的值可以取任意非负值（当然受内核本身的约束）



# Posix有名信号灯

有名信号量可以在无关进程间同步，因为有名信号量是将信号量存储在文件中，在不同的进程中打开相同的文件即可，有名信号量的文件都存储在/dev/shm目录下。

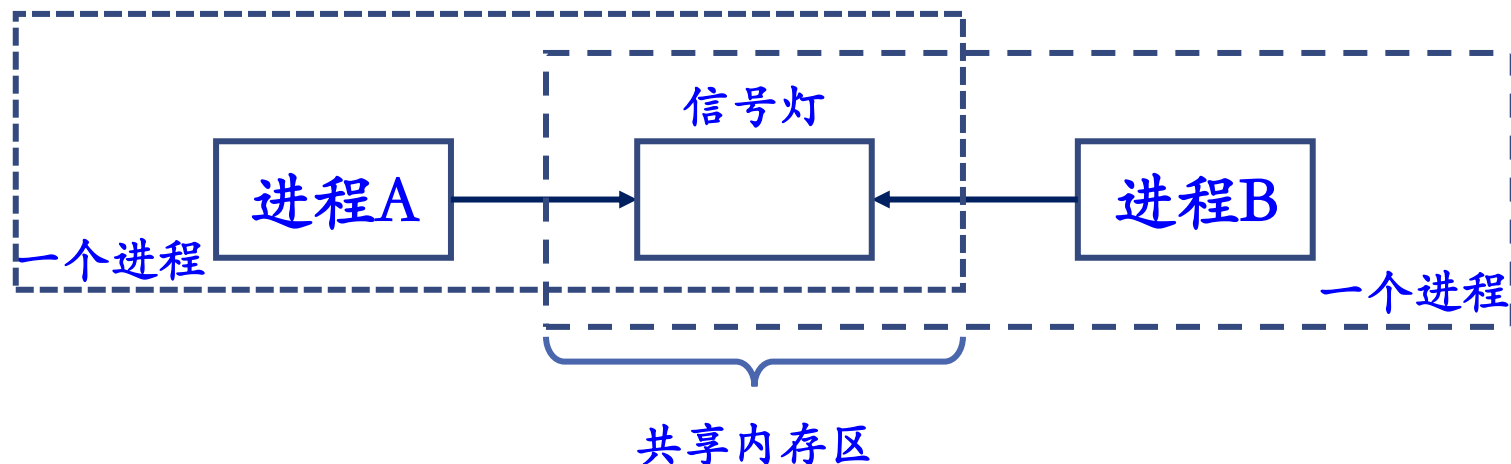




## Posix未命名信号量(基于内存的信号量)

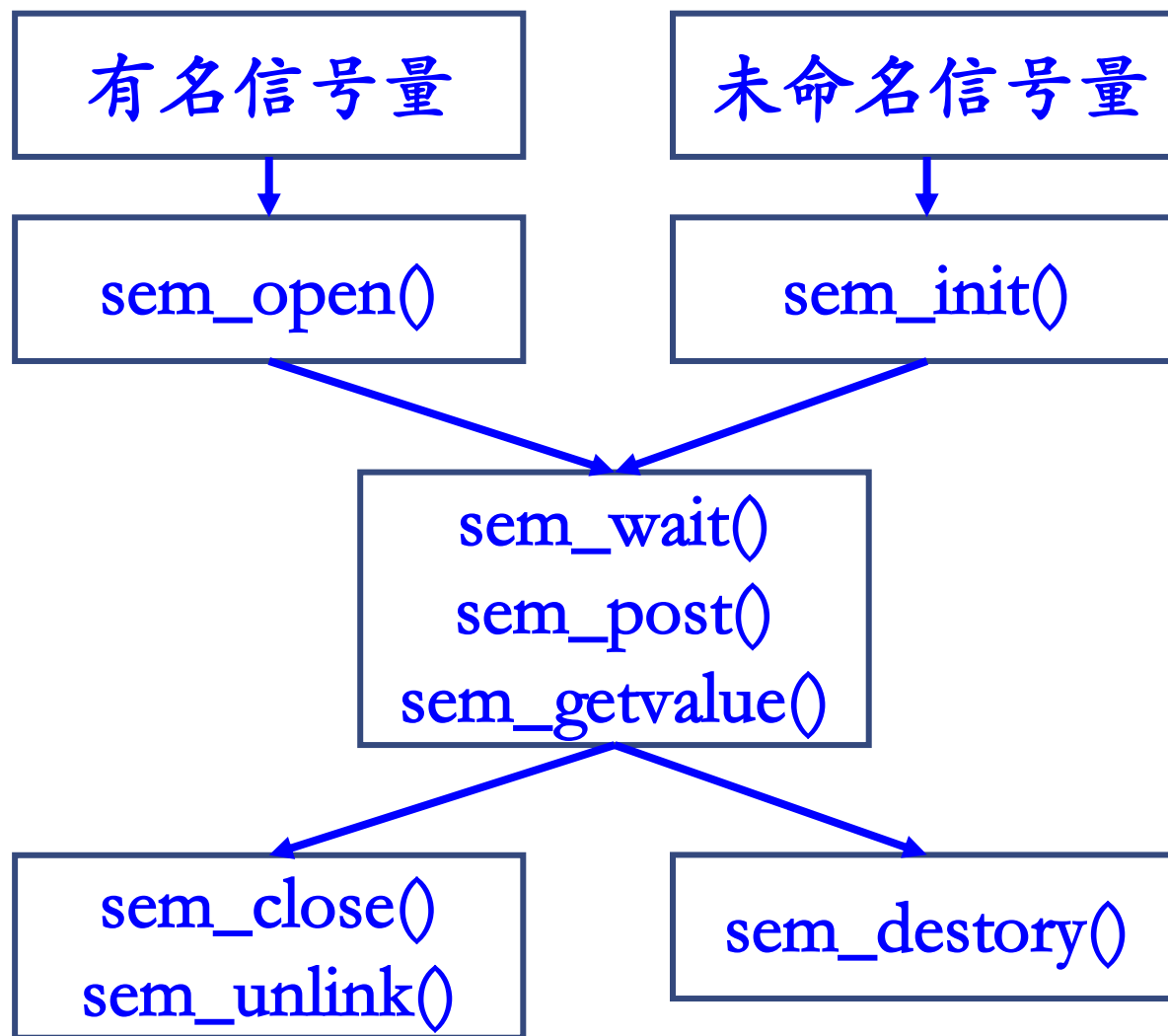
未命名信号量是类型为`sem_t`并存储在应用程序分配的内存的变量。通过将这个信号量放在由几个进程或线程共享的内存区域中就能使这个信号量对这些进程或者线程可用。

基于内存的信号量的通过是指`SHARED`来确定是否除了本进程内部线程共享其他进程也要共享。





# Posix 有名信号量与未命名信号量



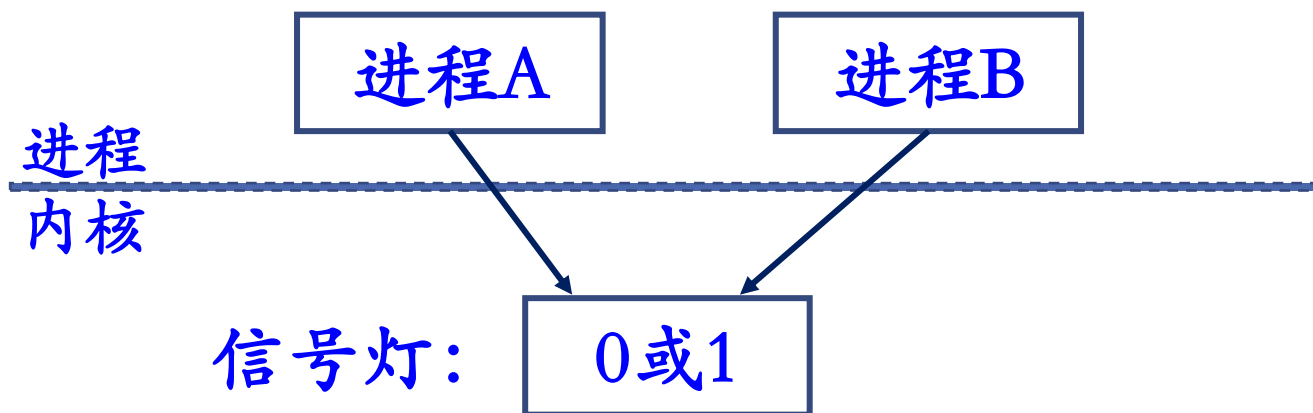




# System V信号灯

System V信号灯是一个或者多个计数信号灯集合，可以同时操作集合中任意多个信号灯。

计数信号灯集(set of counting semaphore): 一个或多个信号灯（构成一个集合），其中每个都是计数信号灯。





## 二值信号灯

- 使用信号灯可以方便的解决临界区问题，来实现进程的互斥。
- 信号灯是一个确定的二元组  $(s, q)$ ， $s$ 是一个具有非负初值的整型变量， $q$ 是一个初始状态为空的队列。整型变量 $s$ 代表资源的实体或并发进程的状态，操作系统利用信号灯的状态对并发进程和共享资源进行管理。一般，当信号灯的值大于或等于零时，表示绿灯，进程可以继续推进；若信号灯的值小于零时，表示红灯，进程被阻。



## 二值信号灯

- 整型变量s的值可以改变，以反映资源或并发进程状态的改变。为了对信号灯的值进行修改，操作系统提供称为P、V操作的一对原语来进行。其可能的取值范围是负整数值、零、正整数值。信号灯是操作系统中实现进程间同步和通信的一种常用工具。
- 一个信号灯的建立必须经过说明，即应该准确说明s的意义和初值（注意：这个初值必须不是一个负值）。每个信号灯都有相应的一个队列，在建立信号灯时，队列为空。



## 二值信号灯

- P、V操作信号灯的数值仅能由P、V操作加以改变。  
对信号灯的P操作记为 $p(s)$ 。
- $p(s)$ 是一个不可分割的原语操作，即取信号灯值减1，若相减结果为负，则调用 $p(s)$ 的进程被阻，并插入到该信号灯的等待队列中，否则可以继续执行。
- P操作的主要动作如下：
  - ①  $s$ 值减1；
  - ② 若相减结果大于或等于0，则进程继续执行；
  - ③ 若相减结果小于零，该进程被封锁，并将它插入到该信号灯的等待队列中，然后转入进程调度程序。



# 二值信号灯



算法 p

输入: 变量 s

输出: 无

```
{  s ← -1;  
  if (s < 0)  
  {  保留调用进程 CPU 现场;  
      将该进程的 pcb 插入 s 的等待队列;  
      置该进程为“等待”状态;  
      转进程调度;  
  }  
}
```



## 二值信号灯

- 对信号灯的V操作记为 $v(s)$ 。
- $v(s)$ 是一个不可分割的原语操作，即取信号灯值加1，若相加结果大于零，进程继续执行，否则，要帮助唤醒在信号灯等待队列上的一个进程。
- V操作的主要动作如下：
  - ①  $s$ 值加1；
  - ② 若相加结果大于零，进程继续执行；
  - ③ 若相加结果小于或等于零，则从该信号灯的等待队列中移出一个进程，解除它的等待状态，然后返回本进程继续执行。





# 二值信号灯



算法 v

输入: 变量 s

输出: 无

```
{  
    s++;  
    if (s ≤ 0)  
    {  
        移出 s 等待队列首元素;  
        将该进程的 pcb 插入就绪队列;  
        置该进程为“就绪”状态;  
    }  
}
```



# 二值信号灯

- 操作信号灯
- 对消息队列的操作无非有下面三种类型：
- (1) 打开或创建信号灯
- (2) 信号灯值操作

linux可以增加或减小信号灯的值，相应于对共享资源的释放和占有。

- (3) 获得或设置信号灯属性：

系统中的每一个信号灯集都对应一个 `struct sem_array` 结构，该结构记录了信号灯集的各种信息，存在于系统空间。为了设置、获得该信号灯集的各种信息及属性，在用户空间有一个重要的联合结构与之对应，即 `union semun`。



# 信号灯可以解决的问题

- 使用信号灯能方便地解决临界区问题。设mutex是用于互斥的信号灯，赋初值为1，表示该临界资源未被占用。只要把进入临界区的操作置于p (mutex) 和v (mutex) 之间，即可实现进程互斥。
- 此时，任何欲进入临界区的进程，必先在互斥信号灯上执行P操作，在完成对临界资源的访问后再执行V操作。
- 由于互斥信号灯的初始值为1，故在第一个进程执行P操作后mutex值变为0，表示临界资源为空闲，可分配给该进程，使之进入临界区。若此时又有第二个进程欲进入临界区，也应先执行P操作，结果使mutex变为负值，这就意味着临界资源已被占用，因此，第二个进程被阻塞。
- 并且，直到第一个进程执行V操作，释放临界资源而恢复mutex值为0后，才唤醒第二个进程，使之进入临界区，待它完成临界资源的访问后，又执行V操作，使mutex恢复到初始值。



# 信号灯可以解决的问题

- 设两个并发进程pa和pb，具有相对于变量n的临界段csa和csb，用信号灯实现它们的互斥描述见如下所示。

```
程序 task2
main( )
{
    int mutex=1;      /* 互斥信号灯 */
    cobegin
        pa( );
        pb( );
    coend
}

pa( )                pb( )
{
    :
    p(mutex);
    csa ;
    v(mutex);
    :
}

    :
    p(mutex);
    csb ;
    v(mutex);
    :
}
```



# 信号灯可以解决的问题

- 对于两个并发进程，互斥信号灯的值仅取1、0和-1三个值。
  - 若 $\text{mutex}=1$ ，表示没有进程进入临界区；
  - 若 $\text{mutex}=0$ ，表示有一个进程进入临界区；
  - 若 $\text{mutex}=-1$ ，表示一个进程进入临界区，另一个进程等待进入。



## Posix 有名信号量 常用函数

```
sem_t *sem_open(const char *name, int oflag);  
sem_t *sem_open(const char *name, int oflag,  
                mode_t mode, unsigned int value);
```

### ◆ 打开信号灯

- ◆ name: 给信号灯起的名字，可以通过 `ll /dev/shm` 即可看到
- ◆ oflag: 打开方式，常用 `O_CREAT | O_EXCL`，IPC 这几个家伙几乎都用这两个
- ◆ mode: 文件权限。常用 `#define FILE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)`，本用户可读可写，同组用户可读，其他用户读。
- ◆ value: 信号量值。二元信号灯值为1，普通的话，就用来表示资源数目就行。





## Posix 有名信号量 常用函数

```
int sem_close(sem_t *sem);
```

- ◆ 关闭信号灯，即便不使用该函数，`exit`或`_exit`的时候也会自动关闭信号灯。但是不能从系统中删除。

```
int sem_unlink(const char* name);
```

- ◆ `name`: 信号灯对应的文件系统名字。该函数作用是将信号灯从文件系统删除。

程序退出前，要确定所有对这个有名信号量的引用都已经通过`sem_close()`函数关闭了，然后只需在退出或是退出处理函数中调用`sem_unlink()`去删除系统中的信号量。

注意因为每个信号灯有一个引用计数器记录当前的打开次数，`sem_unlink`必须等待这个数为0时才能把`name`所指的信号灯从文件系统中删除。也就是要等待最后一个`sem_close`发生。



## Posix 有名信号量 常用函数

int **sem\_wait**(sem\_t \*sem);

int **sem\_trywait**(sem\_t \*sem);

◆让信号灯减1，信号灯值已经是0的话，调用线程将阻塞。

◆sem\_trywait函数则不会阻塞，会返回EAGAIN错误。

int **sem\_post**(sem\_t \*sem);

◆信号灯值加1函数。

int **sem\_getvalue**(sem\_t\* sem, int \*valp);

◆使用int\*获得当前信号灯的值。



# Posix 有名信号量 实例



```
1 /** named_sem1.c
2  * author: Suzkfly
3  * platform: Ubuntu
4  * 配合named_sem2使用, 先运行named_sem1, 此时程序在第二次打印"sem_wait"时卡
5  * 住, 再另开一个终端, 运行named_sem2, 此时named_sem1会继续运行。
6  * 编译时加-lpthread
7  */
8 #include <stdio.h>
9 #include <fcntl.h>
10 #include <sys/stat.h>
11 #include <semaphore.h>
12
13 #define SEM_NAME "mysem" /* 定义信号量的名字 */
14
15 int main(int argc, const char *argv[]) {
16     sem_t *p_sem = NULL;
17     int ret = 0;
18     int value = 0;
19
20     /* 创建信号量 */
21     p_sem = sem_open(SEM_NAME, O_CREAT | O_EXCL, 0666, 1); /* 信号量初值为1 */
22     printf("process 1 sem_open p_sem = %p\n", p_sem);
23     if (p_sem == SEM_FAILED) {
24         printf("process 1 sem_open failed\n");
25         return 0;
26     }
27 }
```

```
27
28 /* 获取信号量的值 */
29 ret = sem_getvalue(p_sem, &value);
30 printf("process 1 sem_getvalue ret = %d\n", ret);
31 printf("value = %d\n", value);
32
33 /* 获取信号量 */
34 printf("process 1 sem_wait\n");
35 ret = sem_wait(p_sem);
36 printf("process 1 sem_wait ret = %d\n", ret);
37
38 /* 获取信号量 */
39 printf("process 1 sem_wait\n");
40 ret = sem_wait(p_sem);
41 printf("process 1 sem_wait ret = %d\n", ret);
42
43 /* 关闭信号量 */
44 ret = sem_close(p_sem);
45 printf("process 1 sem_close ret = %d\n", ret);
46
47 /* 删除信号量文件 */
48 ret = sem_unlink(SEM_NAME);
49 printf("process 1 sem_unlink ret = %d\n", ret);
50
51 return 0;
52 }
```

named\_sem1.c



# Posix 有名信号量 实例



```
1 /** named_sem2.c
2  * author: Suzkfly
3  * platform: Ubuntu
4  * 配合named_sem1使用, 先运行named_sem1, 此时程序在第二次打印"sem_wait"时卡
5  * 住, 再另开一个终端, 运行named_sem2, 此时named_sem1会继续运行。
6  * 编译时加-lpthread
7  */
8 #include <stdio.h>
9 #include <fcntl.h>
10 #include <sys/stat.h>
11 #include <semaphore.h>
12
13 #define SEM_NAME "mysem" /* 定义信号量的名字 */
14
15 int main(int argc, const char *argv[])
16 {
17     int ret = 0;
18     sem_t *p_sem = NULL;
19
20     /* 创建信号量 */
21     p_sem = sem_open(SEM_NAME, O_RDWR);
22     printf("process 2 sem_open p_sem = %p\n", p_sem);
23     if (p_sem == SEM_FAILED) {
24         printf("process 2 sem_open failed\n");
25         return 0;
26     }
27
28     /* 释放信号量 */
29     ret = sem_post(p_sem);
30     printf("process 2 sem_post ret = %d\n", ret);
31
32     /* 关闭信号量 */
33     ret = sem_close(p_sem);
34     printf("process 2 sem_close ret = %d\n", ret);
35
36     return 0;
37 }
```


named\_sem2.c



# Posix 有名信号量 实例

## 最简单的Makefile

```
1 all:named_sem2 named_sem1
2 named_sem1:named_sem1.c
3     gcc -o named_sem1 named_sem1.c -lpthread
4 named_sem2:named_sem2.c
5     gcc -o named_sem2 named_sem2.c -lpthread
```



```
zhaoty@zhaoty-vm-ubuntu20:~/tmp/named_sem$ make
gcc -o named_sem2 named_sem2.c -lpthread
gcc -o named_sem1 named_sem1.c -lpthread
zhaoty@zhaoty-vm-ubuntu20:~/tmp/named_sem$
```



# Posix 有名信号量 实例

```
27
28 /* 获取信号量的值 */
29 ret = sem_getvalue(p_sem, &value);
30 printf("process 1 sem_getvalue ret = %d\n", ret);
31 printf("value = %d\n", value);
32
33 /* 获取信号量 */
34 printf("process 1 sem_wait\n");
35 ret = sem_wait(p_sem);
36 printf("process 1 sem_wait ret = %d\n", ret);
37
38 /* 获取信号量 */
39 printf("process 1 sem_wait\n");
40 ret = sem_wait(p_sem);
41 printf("process 1 sem_wait ret = %d\n", ret);
42
43 /* 关闭信号量 */
44 ret = sem_close(p_sem);
45 printf("process 1 sem_close ret = %d\n", ret);
46
47 /* 删除信号量文件 */
48 ret = sem_unlink(SEM_NAME);
49 printf("process 1 sem_unlink ret = %d\n", ret);
50
51 return 0;
52 }
```

```
zhaoty@zhaoty-vm-ubuntu20: ~/tmp/named_sem
y@zhaoty-vm-ubuntu20:~/tmp/named_sem$ make
o named_sem2 named_sem2.c -lpthread
o named_sem1 named_sem1.c -lpthread
y@zhaoty-vm-ubuntu20:~/tmp/named_sem$ ./named_sem1
ss 1 sem_open p_sem = 0x7faf773da000
ss 1 sem_getvalue ret = 0
    = 1
ss 1 sem_wait
ss 1 sem_wait ret = 0
ss 1 sem_wait
```

阻塞等待

程序阻塞在哪一行?

```
zhaoty@zhaoty-vm-ubuntu20: ~/tmp/named_sem
zhaoty@zhaoty-vm-ubuntu20:~/tmp/named_sem$
```





## Posix 有名信号量 实例

```
27
28  /* 获取信号量的值 */
29  ret = sem_getvalue(p_sem, &value);
30  printf("process 1 sem_getvalue ret = %d\n", ret);
31  printf("value = %d\n", value);
32
33  /* 获取信号量 */
34  printf("process 1 sem_wait\n");
35  ret = sem_wait(p_sem);
36  printf("process 1 sem_wait ret = %d\n", ret);
37
38  /* 获取信号量 */
39  printf("process 1 sem_wait\n");
40  ret = sem_wait(p_sem);
41  printf("process 1 sem_wait ret = %d\n", ret);
42
43  /* 关闭信号量 */
44  ret = sem_close(p_sem);
45  printf("process 1 sem_close ret = %d\n", ret);
46
47  /* 删除信号量文件 */
48  ret = sem_unlink(SEM_NAME);
49  printf("process 1 sem_unlink ret = %d\n", ret);
50
51  return 0;
52 }
```

```
27
28  /* 释放信号量 */
29  ret = sem_post(p_sem);
30  printf("process 2 sem_post ret = %d\n", ret);
31
32  /* 关闭信号量 */
33  ret = sem_close(p_sem);
34  printf("process 2 sem_close ret = %d\n", ret);
35
```

```
zhaoty@zhaoty-vm-ubuntu20:~/tmp/named_sem$ make
gcc -o named_sem2 named_sem2.c -lpthread
gcc -o named_sem1 named_sem1.c -lpthread
zhaoty@zhaoty-vm-ubuntu20:~/tmp/named_sem$ ./named_sem1
process 1 sem_open p_sem = 0x7faf773da000
process 1 sem_getvalue ret = 0
value = 1
process 1 sem_wait
process 1 sem_wait ret = 0
process 1 sem wait
process 1 sem_wait ret = 0
process 1 sem_close ret = 0
process 1 sem_unlink ret = 0
zhaoty@zhaoty-vm-ubuntu20:~/tmp/named_sem$
```

```
zhaoty@zhaoty-vm-ubuntu20:~/tmp/named_sem$ ./named_sem2
process 2 sem_open p_sem = 0x7f2c5fc9d000
process 2 sem_post ret = 0
process 2 sem_close ret = 0
zhaoty@zhaoty-vm-ubuntu20:~/tmp/named_sem$
```



# Posix 有名信号量 实例

ll /dev/shm/

```
zhaoty@zhaoty-vm-ubuntu20: ~/tmp/named_sem
zhaoty@zhaoty-vm-ubuntu20: ~/tmp/named_sem$ ./named_sem1
process 1 sem_open p_sem = 0x7f787706d000
process 1 sem_getvalue ret = 0
value = 1
process 1 sem_wait
process 1 sem_wait ret = 0
process 1 sem_wait

12
13 #define SEM_NAME "mysem" /* 定义信号量的名字 */
14

zhaoty@zhaoty-vm-ubuntu20: ~/tmp/named_sem$ ll /dev/shm/
总用量 0
drwxrwxrwt 2 root root 40 5月 7 07:23 /
drwxr-xr-x 19 root root 4180 5月 6 16:11 ./
zhaoty@zhaoty-vm-ubuntu20: ~/tmp/named_sem$ ll /dev/shm/
总用量 4
drwxrwxrwt 2 root root 60 5月 7 07:24 /
drwxr-xr-x 19 root root 4180 5月 6 16:11 ./
-rw-rw-r-- 1 zhaoty zhaoty 32 5月 7 07:24 sem.mysem
zhaoty@zhaoty-vm-ubuntu20: ~/tmp/named_sem$
```



## Posix 未名信号量 常用函数

`int sem_init (sem_t *sem, int pshared, unsigned int value);`

对sem指定的信号量进行初始化，设置好它的共享选项，并指定一个整数类型的初始值。

pshared参数控制着信号量的类型。如果pshared的值是0，就表示它是当前进程的局部信号量；否则，其它进程就能够共享这个信号量。

`int sem_destroy (sem_t *sem);`

释放信号量sem。



# Posix 未名信号量 实例

```
1 #include <stdio.h>
2 #include <pthread.h>
3 #include <unistd.h>
4 #include <semaphore.h>
5
6 sem_t sem; //信号量
7
8 void printer(char *str) {
9     sem_wait(&sem); //减一
10    while(*str) {
11        putchar(*str);
12        fflush(stdout);
13        str++;
14        sleep(1);
15    }
16    printf("\n");
17    sem_post(&sem); //加一
18 }
19
20 void *thread_fun1(void *arg) {
21     char *str1 = "hello";
22     printer(str1);
23 }
```

```
25 void *thread_fun2(void *arg) {
26     char *str2 = "world";
27     printer(str2);
28 }
29
30 int main(void) {
31     pthread_t tid1, tid2;
32     sem_init(&sem, 0, 1); //初始化信号量, 初始值为 1
33
34     //创建 2 个线程
35     pthread_create(&tid1, NULL, thread_fun1, NULL);
36     pthread_create(&tid2, NULL, thread_fun2, NULL);
37
38     //等待线程结束, 回收其资源
39     pthread_join(tid1, NULL);
40     pthread_join(tid2, NULL);
41
42     sem_destroy(&sem); //销毁信号量
43     return 0;
44 }
```



# System V 信号灯 常用函数

semget函数创建一个信号灯集或访问一个已存在的信号灯集

```
1 #include <sys/sem.h>
2 int semget(key_t key, int nsems, int oflags);
```

返回一个信号灯的标识符。

nsems指定集合中的信号灯数，如果只是打开一个已存在的集合，则该参数指定为0.创建完一个信号灯集，就不能改变其中的信号灯数。

oflags值是SEM\_R和SEM\_A常值的组合。也可以与IPC\_CREAT或IPC\_CREAT|IPC\_EXCL组合。

semctl函数对一个信号灯执行各种控制操作。

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

semid标识待控制其操作的信号灯集，semnum标识该信号灯集中的第semnum个(从0开始)成员，semnum值仅仅用于GETVAL、SETVAL、GETNCNT、GETZCNT和GETPID命令。

**所有System V信号量函数，在它们的名字里面没有下划线。  
例如，应该是semget()而不是sem\_get()。**





# System V 信号灯 常用函数

使用semget打开一个信号灯集合后，对其中的一个或多个信号灯的操作就使用semop函数来执行。

`int semop(int semid, struct sembuf *opsptr, size_t nops);`

semop函数用来对一个或多个信号灯进行操作。

其中struct sembuf结构为：

```
1  struct sembuf
2  {
3      short sem_num;    // semaphores number
4      short sem_op;     // semaphores operations
5      short sem_flg;    // operation flags: 0, IPC_NOWAIT, SEM_UNDO
6  };
```

nops参数指出由opsptr指向的sembuf结构数组中元素的数目。





# System V 信号灯 常用函数

## key\_t键和ftok函数

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，称为IPC键值（也称IPC key键值）。ftok函数原型及说明如下：

`key_t ftok( char * fname, int id )`

ftok（把一个已存在的路径名和一个整数标识符转换成IPC键值）

可以使用计划编号（id）合成IPC key键值，从而避免用户使用key值的冲突。常常简单的设置为0



# System V 信号灯 实例

semcreat

touch info

./semcreat -e info 3

```
1 // semcreat.c
2 #include <sys/types.h>
3 #include <sys/sem.h>
4 #include <stdlib.h>
5 #include <unistd.h>
6
7 #define SVSEM_MODE 0644
8
9 int main(int argc, char **argv) {
10     int c, oflag, semid, nsems;
11
12     oflag = SVSEM_MODE | IPC_CREAT;
13     while((c = getopt(argc, argv, "e")) != -1) {
14         switch(c) {
15             case 'e':
16                 oflag |= IPC_EXCL;
17                 break;
18         }
19     }
20
21     if(optind != argc - 2) {
22         perror("usage: semcreat [-e] <pathname> <nsems>");
23         exit(0);
24     }
25
26     nsems = atoi(argv[optind + 1]);
27     semid = semget(ftok(argv[optind], 0), nsems, oflag);
28
29     return 0;
30 }
```



# System V 信号灯 实例

## semsetvalues

```
1 // semsetvalues.c
2 #include <sys/types.h>
3 #include <sys/ipc.h>
4 #include <stdlib.h>
5 #include <sys/sem.h>
6 #include <stdio.h>
7
8 union semun {
9     int val;
10    struct semid_ds *buf;
11    unsigned short *array;
12 };
13
14 int main(int argc, char **argv) {
15     int semid, nsems, i;
16     struct semid_ds seminfo;
17     unsigned short *ptr;
18     union semun arg;
19
```

./semsetvalues info 1 2 3

```
20     if(argc < 2) {
21         perror("usage: semsetvalues <pathname> [values ...]");
22         exit(0);
23     }
24     semid = semget(ftok(argv[1], 0), 0, 0);
25     arg.buf = &seminfo;
26     semctl(semid, 0, IPC_STAT, arg);
27     nsems = arg.buf->sem_nsems;
28
29     if(argc != nsems + 2) {
30         fprintf(stderr, "%d semaphores in set, %d values specified",
31             nsems, argc - 2);
32         exit(0);
33     }
34
35     ptr = calloc(nsems, sizeof(unsigned short));
36     arg.array = ptr;
37
38     for(i = 0; i < nsems; i++)
39         ptr[i] = atoi(argv[i + 2]);
40     semctl(semid, 0, SETALL, arg);
41     return 0;
42 }
```



# System V 信号灯 实例

## semgetvalues

```
1 // semgetvalues.c
2 #include <sys/ipc.h>
3 #include <sys/sem.h>
4 #include <stdlib.h>
5 #include <stdio.h>
6
7 union semun {
8     int val;
9     struct semid_ds *buf;
10    unsigned short *array;
11 };
12
13 int main(int argc, char **argv) {
14     int semid, nsems, i;
15     struct semid_ds seminfo;
16     unsigned short *ptr;
17     union semun arg;
18
19     if(argc != 2){
20         perror("usage: semgetvalues <pathname>");
21         exit(0);
22     }
23
24     semid = semget(ftok(argv[1], 0), 0, 0);
25     arg.buf = &seminfo;
26     semctl(semid, 0, IPC_STAT, arg);
27     nsems = arg.buf->sem_nsems;
28
29     ptr = calloc(nsems, sizeof(unsigned short));
30     arg.array = ptr;
31
32     semctl(semid, 0, GETALL, arg);
33     for(i = 0; i < nsems; i++)
34         printf("semval[%d] = %d\n", i, ptr[i]);
35     return 0;
36 }
37
```

`./semgetvalues info`

## 9.5 消息队列

---

消息队列的通信

消息队列总结





# 消息队列的通信

消息队列提供了一种从一个进程向另一个进程发送一个数据块的方法。每个数据块都被认为含有一个类型，接收进程可以独立地接收含有不同类型的数据结构。我们可以通过发送消息来避免命名管道的同步和阻塞问题。

双向通信，既可以客户端到服务器端发送消息，也可以用于服务端到客户端发送消息，而管道只能单向通信。服务端通过类型进行区分，将消息发给不同的客户端。给不同的客户端发送的消息是不同类型的消息，客户端接收对应类型的消息。





# 消息队列总结

- Linux消息队列通信机制属于消息传递通信机制。消息队列是内核地址空间中的内部链表，每个消息队列具有唯一的标识符。
- 消息可以顺序地发送到队列中，并以几种不同的方式从队列中获取。
- 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；对消息队列有读权限的进程则可以从消息队列中读出消息。



# 消息队列总结

- 消息的发送方式如下：发送方不必等待接收方检查它所收到的消息即可继续工作，而接收方如果没有收到消息则也无需等待。
- 新的消息总放在队列的末尾，接收的时候并不总是从头来接收的，可以从中间来接收。消息队列随内核存在并和进程相关，即使进程退出它也仍然存在。只有在内核重起或者显示删除一个消息队列时，该消息队列才会真正被删除。因此，系统中记录消息队列的数据结构位于内核中，系统中的所有消息队列都可以在结构msg\_ids中找到访问入口。



# 消息队列总结

- Linux消息队列的主要操作函数如下。
- (1) `msgget`: 创建一个消息队列或者返回已存在消息队列的标识号。
- (2) `msgsnd`: 向一个消息队列发送一个消息。
- (3) `msgrcv`: 从一个消息队列中接收一个消息。
- (4) `msgctl`: 在消息队列上执行指定的操作，如执行检索、删除等操作。



# 消息队列 实例



## msgreceive.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <sys/msg.h>
5 #include <errno.h>
6
7 struct msg_st {
8     long int msg_type;
9     char text[BUFSIZ];
10 };
11
12 int main(int argc, char **argv) {
13     int msgid = -1;
14     struct msg_st data;
15     long int msgtype = 0;    // 注意1
16
17     // 建立消息队列
18     msgid = msgget((key_t)1234, 0666 | IPC_CREAT);
19     if (msgid == -1) {
20         fprintf(stderr, "msgget failed with error: %d\n", errno);
21         exit(EXIT_FAILURE);
22     }
23
24     // 从队列中获取消息, 直到遇到end消息为止
25     while (1) {
26         if (msgrcv(msgid, (void *)&data, BUFSIZ, msgtype, 0) == -1) {
27             fprintf(stderr, "msgrcv failed with error: %d", errno);
28         }
29
30         printf("You wrote: %s\n", data.text);
31
32         // 遇到end结束
33         if (strncmp(data.text, "end", 3) == 0) {
34             break;
35         }
36     }
37
38     // 删除消息队列
39     if (msgctl(msgid, IPC_RMID, 0) == -1) {
40         fprintf(stderr, "msgctl(IPC_RMID) failed\n");
41     }
42
43     exit(EXIT_SUCCESS);
44 }
```



# 消息队列 实例

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4 #include <unistd.h>
5 #include <sys/msg.h>
6 #include <errno.h>
7
8 #define MAX_TEXT 512
9
10 struct msg_st {
11     long int msg_type;
12     char text[MAX_TEXT];
13 };
14
15 int main(int argc, char **argv) {
16     struct msg_st data;
17     char buffer[BUFSIZ];
18     int msgid = -1;
19
20     // 建立消息队列
21     msgid = msgget((key_t)1234, 0666 | IPC_CREAT);
22     if (msgid == -1) {
23         fprintf(stderr, "msgget failed error: %d\n", errno);
24         exit(EXIT_FAILURE);
25     }
26
27     // 向消息队里中写消息, 直到写入end
28     while (1) {
29         printf("Enter some text: \n");
30         fgets(buffer, BUFSIZ, stdin);
31         data.msg_type = 1; // 注意2
32         strcpy(data.text, buffer);
33
34         // 向队列里发送数据
35         if (msgsnd(msgid, (void *)&data, MAX_TEXT, 0) == -1) {
36             fprintf(stderr, "msgsnd failed\n");
37             exit(EXIT_FAILURE);
38         }
39
40         // 输入end结束输入
41         if (strncmp(buffer, "end", 3) == 0) {
42             break;
43         }
44
45         sleep(1);
46     }
47
48     exit(EXIT_SUCCESS);
49 }
```

## msgsend.c



# 消息队列 实例



```
zhaoty@zhaoty-vm-ubuntu20:~/tmp/msg$ ./msgreceive
You wrote: ABC

You wrote: D

zhaoty@zhaoty-vm-ubuntu20:~/tmp/msg$ ./msgsend
Enter some text:
ABC
Enter some text:
D
Enter some text:
```



## 9.6 共享存储

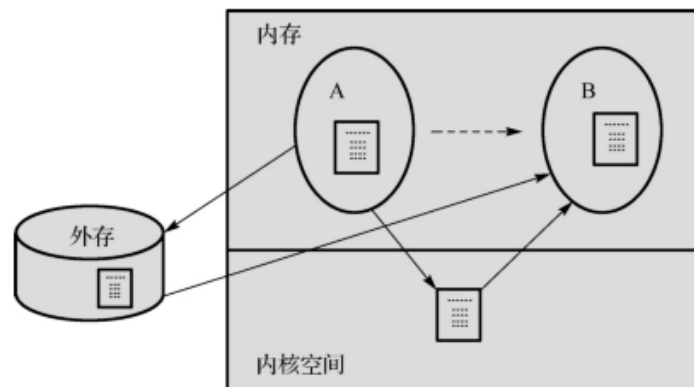
共享存储的介绍

共享存储的例子

共享存储的结构

如何将进程挂接到共享存储

如何查看系统的共享存储资源





# 共享存储

- 共享内存区到进程虚拟地址空间的映射共享内存区映射到进程中未使用的虚地址区，以免与进程映像发生冲突。共享内存的页面在每个共享进程的页表中都有页表项引用，但无需在所有进程的虚地址段都有相同的地址。共享内存区属于临界资源，读写共享内存区的代码属于临界区。
- Linux内核为每个共享内存段维护一个数据结构 `shmid_ds`，其中描述了段的大小、操作权限、与该段有关系的进程标识等。



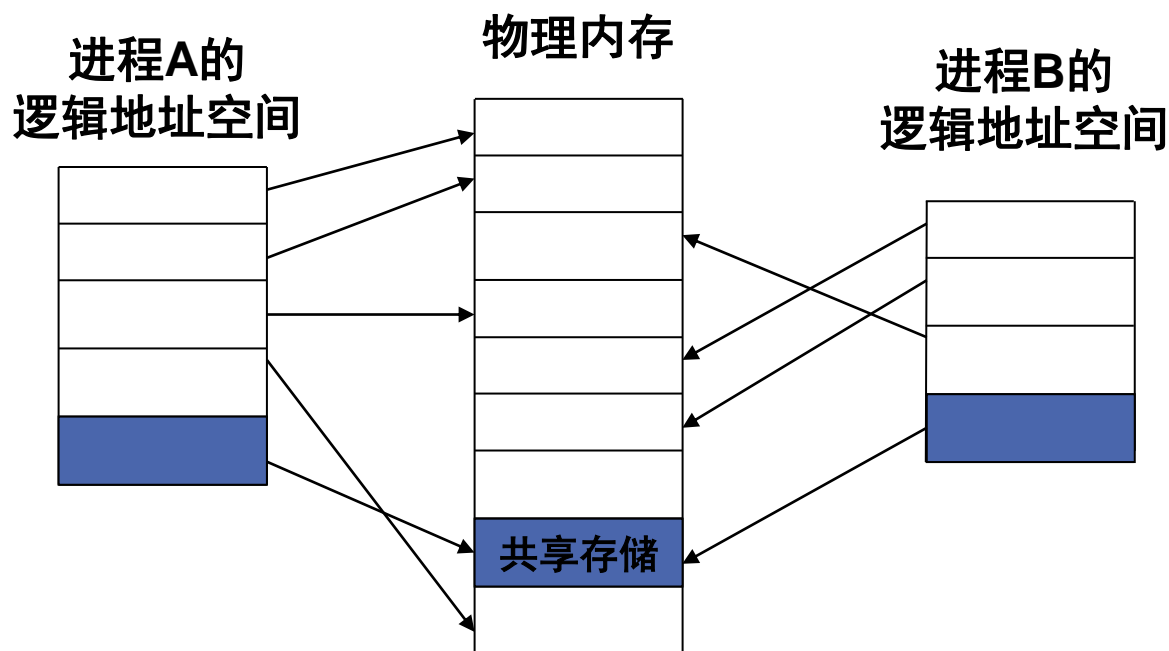
# 共享存储

- 共享内存的主要操作如下。
- (1) 创建共享内存：使用共享内存通信的第一个进程创建共享内存，其他进程则通过创建操作获得共享内存标识符，并据此执行共享内存的读写操作。
- (2) 共享内存绑定（映射共享内存区到调用进程地址空间）：需要通信的进程将先前创建的共享内存映射到自己的虚拟地址空间，使共享内存成为进程地址空间的一部分，随后可以像访问本地空间一样访问共享内存。
- (3) 共享内存解除绑定（断开共享内存连接）：不再需要共享内存的进程可以解除共享内存到该进程虚地址空间的映射。
- (4) 撤销共享内存：当所有进程不再需要共享内存时可删除共享内存。



# 如何将进程挂接到共享存储

- 使用共享存储时要掌握的唯一窍门是多个进程之间对一个给定存储区的同步访问。若服务器进程正在将数据放入共享存储区，则在它做完这一操作之前，客户进程不应当去取这些数据。通常，信号量被用来实现对共享存储访问的同步。



**共享资源，需要考虑进程间的同步！**



# 如何将进程挂接到共享存储

## ■ 共享内存系统调用

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

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

```
#include <sys/shm.h>
```

```
int semget(key_t key, int size, int flag);
```

```
void *shmat(int shmid, void *addr, int flag);
```

```
int shmdt(void *addr);
```

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```



# 如何将进程挂接到共享存储

- `shmat`函数（功能：将共享内存段连接到一个进程的地址空间中。）

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

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

```
#include <sys/shm.h>
```

```
void *shmat(int shm_id, const void *addr, int shmflg);
```

**成功返回共享存储段连接的实际地址，失败返回-1**

- 第一个参数`shm_id`为`shmget`返回的共享内存标识符。
- 第二个参数`addr`指明共享内存段要连接到的地址（进程空间内部地址），通常指定为空指针，表示让系统来选择共享内存出现的地址。
- 第三个参数`shmflg`可以设置为两个标志位（通常设置为0）
  - `SHM_RND`（与`shm_addr`联合使用，控制内存连接的地址）
  - `SHM_RDONLY`，要连接的共享内存段是只读的。





# 如何查看系统的共享存储资源

## ■ shmget函数

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

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

```
#include <sys/shm.h>
```

```
int shmget(key_t key, size_t size, int shmflag);
```

成功返回一个共享内存标识符，失败返回-1

第一个参数key为共享内存段命名（一般由ftok产生）

第二个参数size为需要共享的内存容量。（如果共享内存已存在时，不能不大于该共享内存段的大小）

第三个参数设置访问权限(低9位) 与IPC\_CREAT, IPC\_EXCL的按位或。



# 如何查看系统的共享存储资源

- 查看ipc资源（消息队列、共享内存、信号量）使用ipcs命令其使用的方法如下：
- `ipcs [-a -s -q -m] -i id`
- 默认的情况直接使用-a，会打印出所有的信息，需要查看信号量使用-s，消息队列使用-q，共享内存-m。



# 共享存储 实例

```
1 #include <sys/types.h>
2 #include <string.h>
3 #include <stdlib.h>
4 #include <sys/ipc.h>
5 #include <sys/shm.h>
6 #include <stdio.h>
7 #include <semaphore.h>
8 #include <sys/types.h>
9 #include <sys/stat.h>
10 #include <fcntl.h>
11 #include <unistd.h>
12
13 #define SHMSZ 270
14 char SEM_NAME[] = "aa";
15
16 int main()
17 {
18     char ch;
19     int shmid;
20     key_t key;
21     char *shm,*s;
22     sem_t *mutex;
23     //name the shared memory segment
24     key = 1000;
25     //create & initialize semaphore
26     mutex = sem_open(SEM_NAME,O_CREAT,0644,2);
27     if(mutex == SEM_FAILED)
28     {
29         perror("unable to create semaphore");
30         sem_unlink(SEM_NAME);
31         exit(-1);
32     }
33     //create the shared memory segment with this key
34     shmid = shmget(key,SHMSZ,IPC_CREAT|0666);
35     if(shmid<0) {
36         perror("failure in shmget");
37         exit(-1);
38     }
39     //attach this segment to virtual memory
40     shm = shmat(shmid,NULL,0);
41     //start writing into memory
42     s = shm;
43
44     memset(s,'\0',SHMSZ);
45
46     int n=20;
47     for(;n>-1;n--) {
48         sem_wait(mutex);
49
50         printf("server-step:%d,s:%s\n",n,s);
51
52         if(NULL!=strchr(s,'*'))
53             break;
54
55         sem_post(mutex);
56
57         sleep(1);
58     }
59
60     sem_close(mutex);
61     sem_unlink(SEM_NAME);
62     shmctl(shmid, IPC_RMID, 0);
63     exit(0);
64 }
```



# 共享存储 实例

```
1 #include <sys/types.h>
2 #include <unistd.h>
3 #include <string.h>
4 #include <sys/ipc.h>
5 #include <sys/shm.h>
6 #include <stdio.h>
7 #include <semaphore.h>
8 #include <sys/types.h>
9 #include <sys/stat.h>
10 #include <fcntl.h>
11 #include <stdlib.h>
12
13 #define SHMSZ 270
14 char SEM_NAME[] = "aa";
15 int main()
16 {
17     char ch;
18     int shmid;
19     key_t key;
20     char *shm,*s;
21     sem_t *mutex;
22     //name the shared memory segment
23     key = 1000;
24     //create & initialize existing semaphore
25     mutex = sem_open(SEM_NAME,0,0644,0);
26     if(mutex == SEM_FAILED) {
27         perror("reader:unable to execute semaphore");
28         sem_close(mutex);
29         exit(-1);
30     }
```

```
31     //create the shared memory segment with this key
32     shmid = shmget(key,SHMSZ,0666);
33     if(shmid<0) {
34         perror("reader:failure in shmget");
35         exit(-1);
36     }
37     //attach this segment to virtual memory
38     shm = shmat(shmid,NULL,0);
39     //start reading
40     s = shm;
41
42     int n=10;
43     for(;n>-1;n--) {
44         sem_wait(mutex);
45
46         if(n==0)
47             strcat(s,"*");
48         else
49             strcat(s,"A");
50
51         printf("client-step:%d,s:%s\n",n,s);
52
53         sem_post(mutex);
54
55         sleep(1);
56     }
57
58     //once done signal exiting of reader:This can be
59     // replaced by another semaphore
60     sem_close(mutex);
61     shmctl(shmid, IPC_RMID, 0);
62     exit(0);
63 }
```

client.c



# 共享存储 实例

```
gcc -o client client.c -lpthread
gcc -o server server.c -lpthread
zhaoty@zhaoty-vm-ubuntu20:~/tmp/ipc$ ./server
server-step:20,s:
server-step:19,s:A
server-step:18,s:AA
server-step:17,s:AAA
server-step:16,s:AAAA
server-step:15,s:AAAAA
server-step:14,s:AAAAAA
server-step:13,s:AAAAAAA
server-step:12,s:AAAAAAA
server-step:11,s:AAAAAAA
server-step:10,s:AAAAAAA
server-step:9,s:AAAAAAA*
zhaoty@zhaoty-vm-ubuntu20:~/tmp/ipc$
```

```
zhaoty@zhaoty-vm-ubuntu20:~/tmp/ipc$ ./client
client-step:10,s:A
client-step:9,s:AA
client-step:8,s:AAA
client-step:7,s:AAAA
client-step:6,s:AAAAA
client-step:5,s:AAAAAA
client-step:4,s:AAAAAAA
client-step:3,s:AAAAAAA
client-step:2,s:AAAAAAA
client-step:1,s:AAAAAAA
client-step:0,s:AAAAAAA*
zhaoty@zhaoty-vm-ubuntu20:~/tmp/ipc$
```

## 小测试

- 1.哪种进程间的通信方式不能传递大量的信息?
  - A.共享内存
  - B.消息缓冲
  - C.信箱通信
  - D.信号量及P、V操作
-





## 课后作业

- 1) 编写程序：在父子进程间相互传递消息，用管道实现。
- 2) 编写程序：实现生产者-消费者程序，有多个生产者和多个消费者，它们共用一个共享数组，用信号量实现相应的同步和互斥机制。