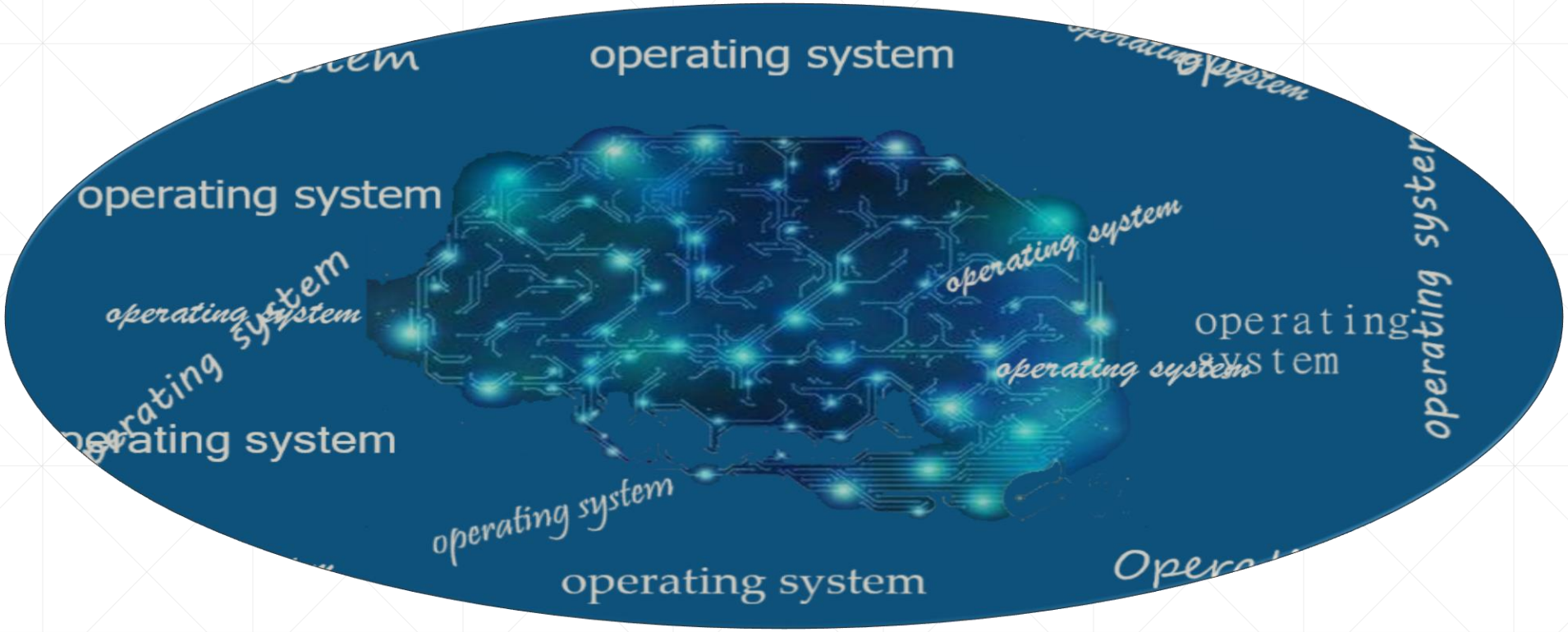


第十二章 操作系统接口与应用开发

授课教师：任立勇

电子邮箱：17524677@qq.com

12.1系统调用的实现机制

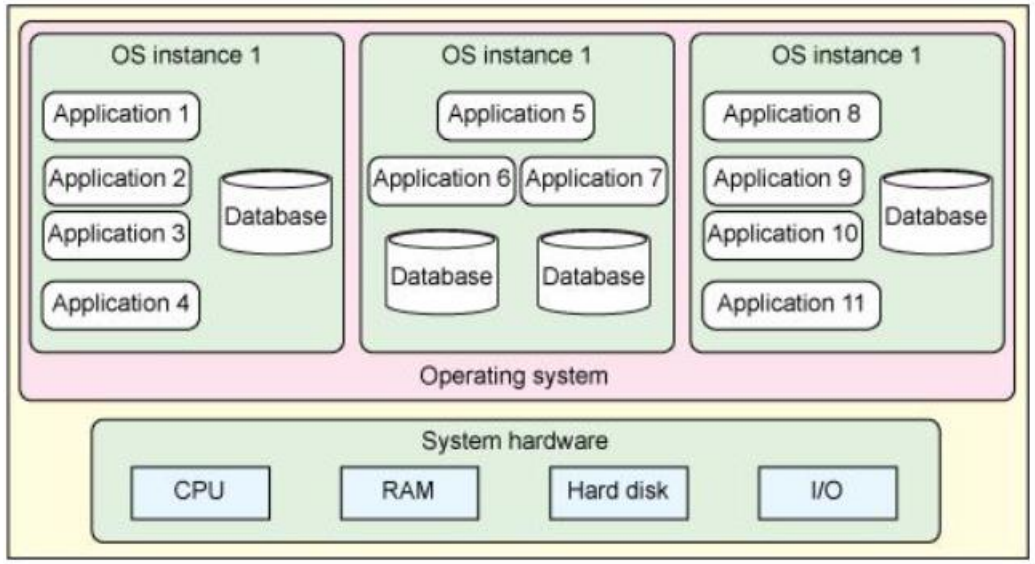
操作系统与用户接口

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（POSIX）、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操作系统之间添加了一个中间层，该层的主要作用有两个:**
- 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；
- 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）**

系统调用

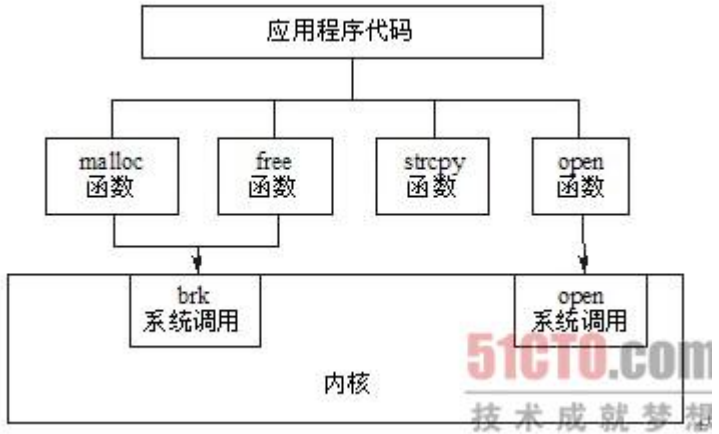
- 4.4BSD版本提供了大约110个系统调用，UNIX System V Release 4提供了大约120个系统调用，Linux根据版本的不同有240到260个系统调用。

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。

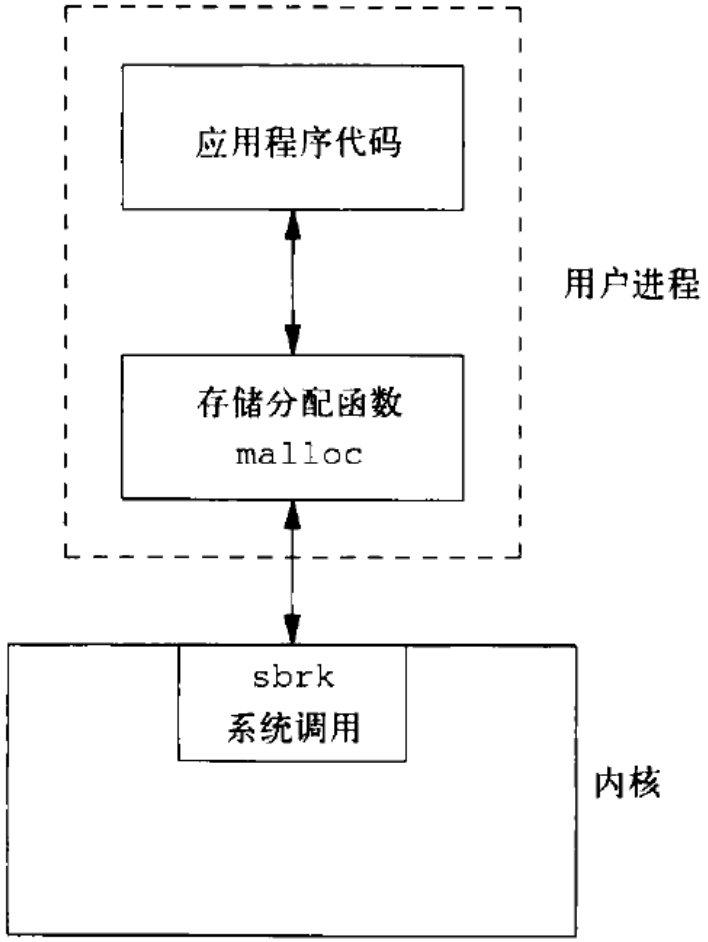
系统调动与库函数之间的关系

- 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。



系统调动与库函数之间的关系

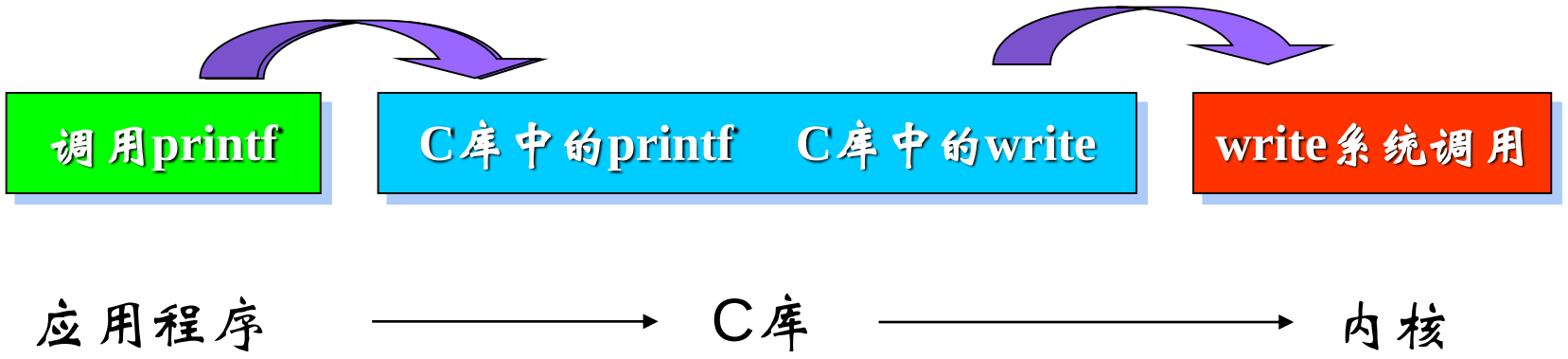
- 系统调用通常提供一种内核功能的最小接口，而库函数通常提供比较复杂的功能
- 并非所有的库函数都会调用系统调用，例如，printf函数会调用write系统调用以输出一个字符串，但strcpy和atoi函数才不使用任何系统调用



API（POSIX）、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



12.2 文件

UNIX/Linux文件的I/O操作

UNIX/Linux文件属性的获取

Unix/Linux文件的I/O操作

Open( )函数

头文件：fcntl.h

int open(const char *pathname, int oflag, ...);

- **该函数打开或创建一个文件。其中第二个参数oflag说明打开文件的选项，第三个参数是变参，仅当创建新文件时才使用。**
- **O_RDONLY:：只读打开；**
- **O_WRONLY：只写打开；**
- **O_RDWR：读、写打开；**
- **O_APPEND：每次写都加到文件尾；**

Unix/Linux文件的I/O操作

- O_CREAT：若此文件不存在则创建它，此时需要第三个参数mode，该参数约定了所创建文件的权限，计算方法为mode&~umask
- O_EXCL：如同时指定了O_CREAT，此指令会检查文件是否存在，若不存在则建立此文件；若文件存在，此时将出错。
- O_TRUNC：如果此文件存在，并以读写或只写打开，则文件长度0
- 由open返回的文件描述符一定是最小的未用描述符数值。

Unix/Linux文件的I/O操作

Open函数在内核完成的工作：

```
算法  open
输入：文件名
      打开类型
      文件许可权方式（对以创建方式打开而言）
输出：文件描述符
{
    将文件名转换为索引节点（算法 namei）；
    if  （文件不存在或不允许存取）
        return（错）；
    为索引节点分配文件表项，设置引用数和偏移量；
    分配用户文件描述符表项，将指针指向文件表项；
    if  （打开的类型规定清文件）
        释放所有文件块（算法 free）；
    解锁（索引节点）；  /*  在上面的 namei 算法中上锁  */
    return（用户文件描述符）；
}
```

Unix/Linux文件的I/O操作

Create( )函数

头文件：fcntl.h

int creat(const char *pathname, mode_t mode);

- **该函数用于创建一个新文件，其等效于open函数的如下调用：**

```
open( pathname, O_WROLY | O_CREATE | O_TRUNC,
  mode);
```

- **creat函数的一个不足之处是它以只写方式打开所创建的文件**

Unix/Linux文件的I/O操作

- 早期UNIX版本中open的第二个参数只能是0、1或2，没有办法打开一个尚未存在的文件。如果要创建一个临时文件，并要先写该文件，然后又读该文件。则必须先调用creat，close，然后再open。

Unix/Linux文件的I/O操作

```
//come from /usr/include/bit/fcntl.h
/* open/fcntl - O_SYNC is only implemented on blocks devices and on files located on an ext2 file system */
#define O_ACCMODE       0003        //主要访问权限位为低两位，用来测试权限用
#define O_RDONLY        00          //只读
#define O_WRONLY        01          //只写
```

```
#define O_RDWR          02          //读写方式
#define O_CREAT         0100        //如果没有，创建
#define O_EXCL          0200        //如果存在，返回错误
#define O_NOCTTY        0400        //终端控制信息
#define O_TRUNC         01000       //截短
#define O_APPEND        02000       //追加
```

Unix/Linux文件的I/O操作

Close( )函数

头文件unistd.h

```
int  close( int  filedes );
```

- 该函数关闭以前打开的一个文件。关闭文件的同时也释放该进程加在该文件上的所有记录锁。当一个进程终止时，它所有的打开文件将由内核自动关闭。
- 内核对文件描述符、对应的文件表项和索引节点表项进行相应的处理，来完成关闭文件的操作。
- 进程关闭文件后，就不能通过该文件描述符操作该文件；当一个进程正常退出时，内核将关闭所有打开的文件描述符。

Unix/Linux文件的I/O操作

Read( )函数

- 头文件unistd.h
- ssize_t read(int filedes, void *buf, size_t nbytes);
- **read函数从打开的文件中读数据。如成功，则返回实际读到的字节数，如已到达文件的末尾或无数据可读，则返回0；有多种情况可使实际读到的字节数少于要求读的字节数：**

Unix/Linux文件的I/O操作

- 读普通文件，在读到要求字节数之前就到达文件尾；
- 当从终端设备读，通常一次最多读一行；
- 当从网络读时，网络中的缓冲机构可能造成返回值小于所要求读的字节数；
- 某些面向记录的设备，如磁带，一次最多返回一个记录。
- **读操作完成后，文件偏移量将从读之前的偏移量加上实际读的字节数。**

Unix/Linux文件的I/O操作

- Write( )函数
- 头文件unistd.h
- ssize_t write(int filedes, const void *buf, size_t nbytes);
- 该函数返回实际写的字节数，通常与参数nbytes的值相同，否则表示出错。
- 如果出错，则返回 - 1。write出错的原因可能是磁盘满、没有访问权限、或写超过文件长度限制等等。
- 对于普通文件，写操作从文件的当前偏移量开始写，除非打开文件时指定了O_APPEND选项。完成后，文件偏移量加上实际写的字节数。

Unix/Linux文件的I/O操作

- 如果文件中还没有要写的字节偏移量所对应的块，内核要用算法alloc分配一个新块或者多个块。写操作结束，如果文件大小变大，则需要修改索引节点。
- 内核在write的每次写循环期间每次向磁盘写一块。内核决定是写整块还是写块中的一部分，如果是后者，则内核需要先将该块从磁盘读到高速缓冲区中，再修改需要的部分；如果是前者，内核就不必读了，而是用新块直接覆盖
- 内核一般采用延迟写块，先将数据放到高速缓存；

Unix/Linux文件的I/O操作

例子程序

```
int main(int argc,char **argv)
{
        int n,buffsize;
        char *buff;
        if(argc != 2)       {
                printf("Usage:%s #n\n",argv[0]);
                exit(1);
        }
        buffsize = atoi(argv[1]);
        if((buff = malloc(buffsize)) == NULL)
                err_sys("malloc memory error.");
        while((n = read(STDIN_FILENO,buff,buffsize)) > 0)
```

Unix/Linux文件的I/O操作

```
if(write(STDOUT_FILENO,buff,n) != n)
                    err_sys("write error.");
        if(n < 0)
                err_sys("read error.");
        exit(0);
}
```

I/O的效率

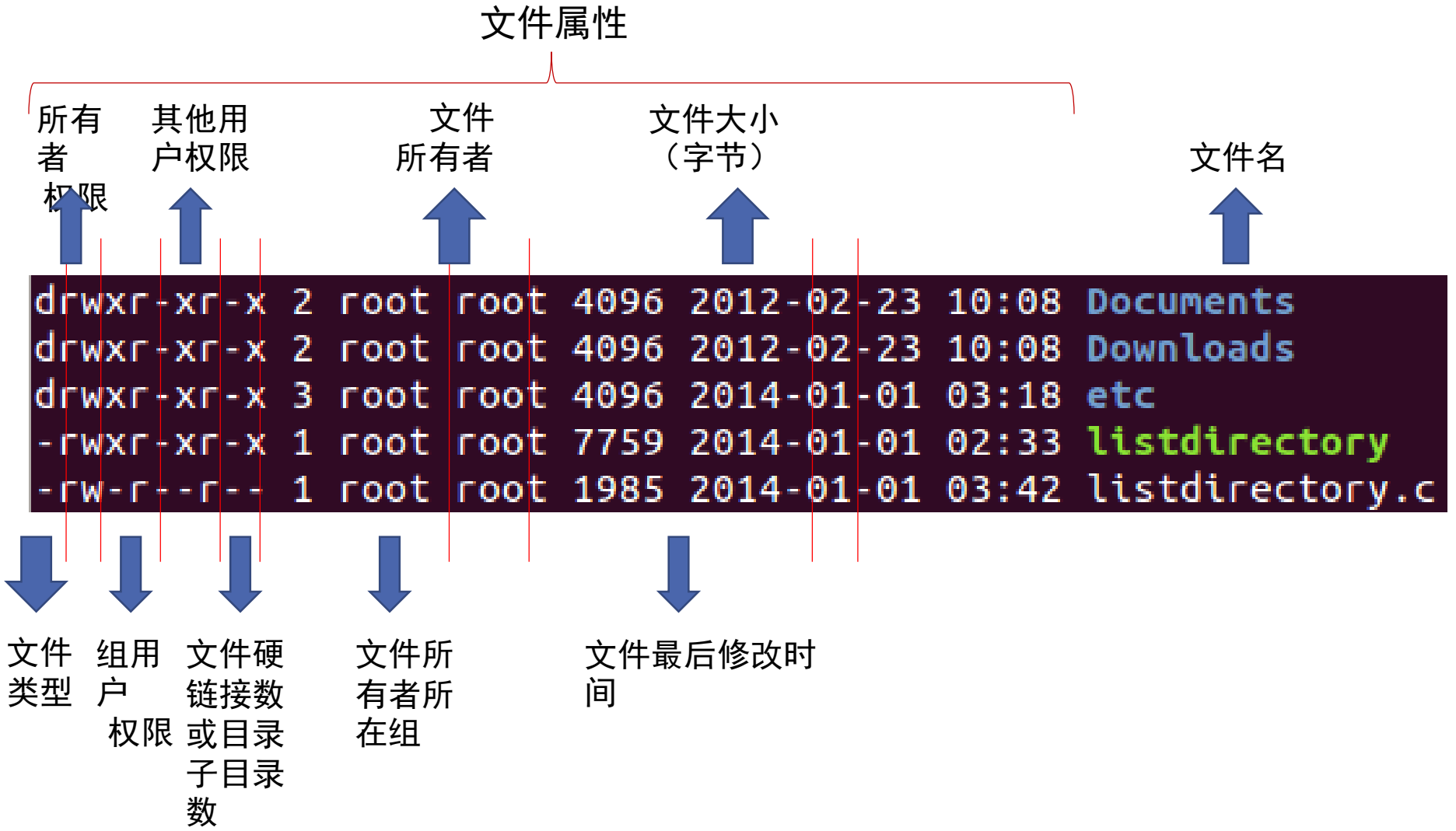
由于文件系统中的磁盘块大小是固定，并且由于高速缓冲区的存在，应用程序执行读写操作时，其每次请求的读写大小在很大程度上影响着应用程序的I/O效率。

文件属性的获取

```
root@xrxy-virtual-machine: ~
root@xrxy-virtual-machine:~# cd /root/
root@xrxy-virtual-machine:~# ls -l
total 64
-rwxr-xr-x 1 root root 7255 2013-12-31 22:43 a.out
drwxr-xr-x 2 root root 4096 2012-02-23 10:08 Desktop
drwxr-xr-x 2 root root 4096 2012-02-23 10:08 Documents
drwxr-xr-x 2 root root 4096 2012-02-23 10:08 Downloads
drwxr-xr-x 3 root root 4096 2014-01-01 03:18 etc
-rwxr-xr-x 1 root root 7759 2014-01-01 02:33 listdirectory
-rw-r--r-- 1 root root 1985 2014-01-01 03:42 listdirectory.c
-rw-r--r-- 1 root root 1985 2014-01-01 02:41 listdirectory.c~
drwxr-xr-x 2 root root 4096 2012-02-23 10:08 Music
drwxr-xr-x 2 root root 4096 2012-02-23 10:08 Pictures
drwxr-xr-x 2 root root 4096 2012-02-23 10:08 Public
drwxr-xr-x 2 root root 4096 2012-02-23 10:08 Templates
drwxr-xr-x 6 root root 4096 2014-01-01 03:18 usr
drwxr-xr-x 2 root root 4096 2012-02-23 10:08 Videos
root@xrxy-virtual-machine:~#
```

列出当前目录的所有文件（包括普通文件、目录文件、字符特殊文件、套接字等），并显示文件类型、访问权限、文件大小等重要属性

文件属性的获取


文件属性
所有者权限
其他用户权限
文件所有者
文件大小（字节）
文件名
drwxr-xr-x 2 root root 4096 2012-02-23 10:08 Documents
drwxr-xr-x 2 root root 4096 2012-02-23 10:08 Downloads
drwxr-xr-x 3 root root 4096 2014-01-01 03:18 etc
-rwxr-xr-x 1 root root 7759 2014-01-01 02:33 listdirectory
-rw-r--r-- 1 root root 1985 2014-01-01 03:42 listdirectory.c
文件类型
组用户权限
文件硬链接数或目录子目录数
文件所有者所在组
文件最后修改时间

文件属性的获取

读取文件属性

- 常用函数：stat，lstat，fstat
- 头文件： sys/stat.h
- 函数定义：
  - int stat(const char *path, struct stat *buf);
  - int lstat(const char *path, struct stat *buf);
    - 两个函数参数相同，功能类似
    - 读取path参数所指定文件的文件属性并将其填充到buf参数所指向的结构体中

文件属性的获取

- 对于符号链接文件，lstat返回符号链接的文件属性，stat返回符号链接引用文件的文件属性

- int fstat(int filedes, struct stat *buf);
  - 与前两个函数功能类似，指定文件的方式改为通过文件描述符

文件属性的获取

文件属性解析

重要数据结构

```
struct stat {
  mode_t    st_mode;      文件类型与访问权限
  ino_t     st_ino;       i节点号
  dev_t     st_dev;       文件使用的设备号
  dev_t     st_rdev;      设备文件的设备号
  nlink_t   st_nlink;     文件的硬链接数
  uid_t     st_uid;       文件所有者用户ID
  gid_t     st_gid;       文件所有者组ID
  off_t     st_size;      文件大小（以字节为单位）
  time_t    st_atime;     最后一次访问该文件的时间
  time_t    st_mtime;     最后一次修改该文件的时间
  time_t    st_ctime;     最后一次改变该文件状态的时间
  blksize_t st_blksize;   包含该文件的磁盘块的大小
  blkcnt_t  st_blocks;    该文件所占的磁盘块 数
};
```

12.3 进程控制

运行环境和进程的标识

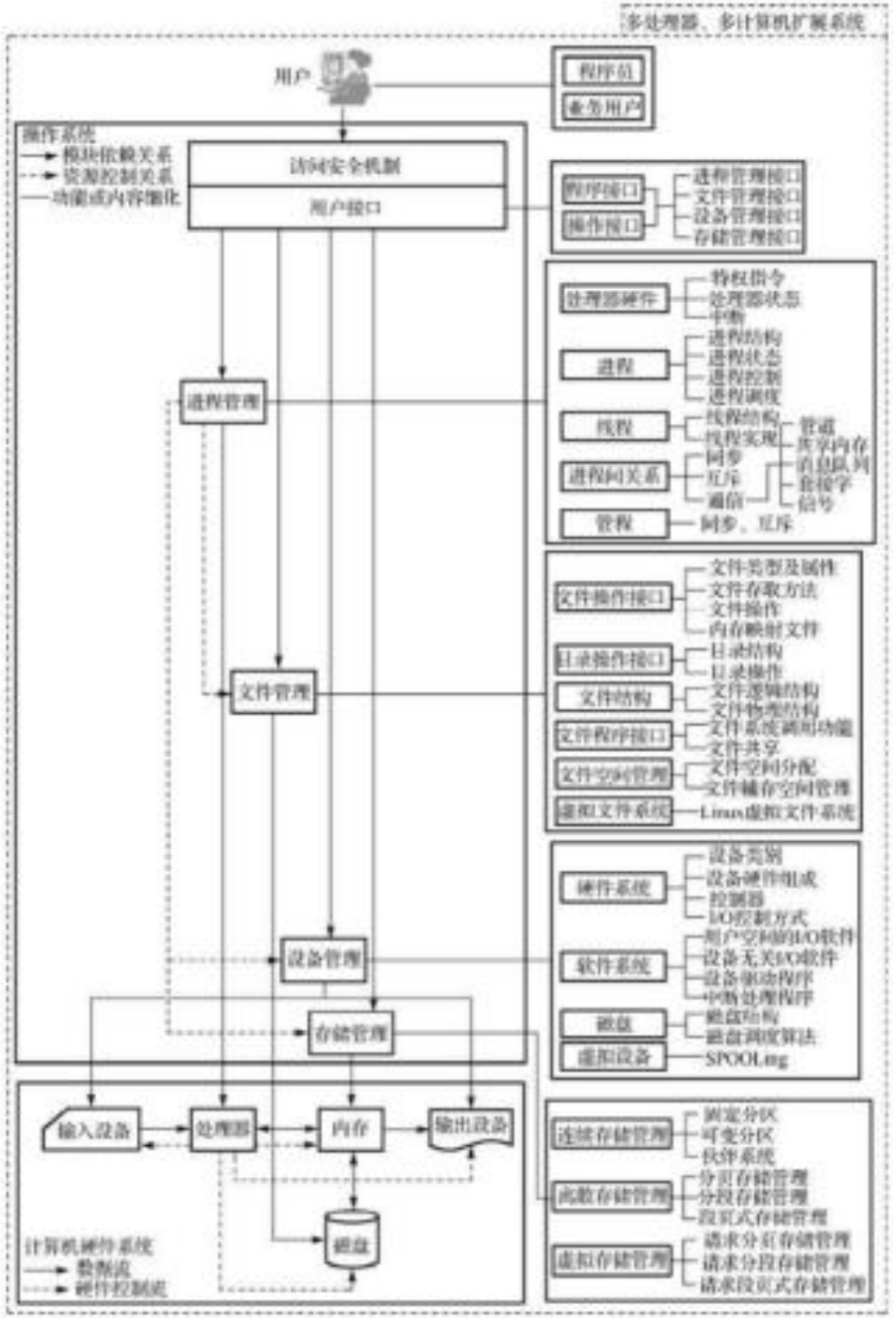
进程的建立和退出

进程的等待与睡眠

进程的执行

进程的跟踪

进程的属性修改



？问题

- 进程在什么环境运行?
- 有哪些描述进程的标识符?
- 如何创建进程?
- 进程在什么情况下进行等待与睡眠?
- 如何修改进程的属性?
- 何时需要对进程进行调度?
- 进程调度有哪些常用算法?

进程描述符

- Linux内核利用一个数据结构（task_struct）代表一个进程，在该结构中保存进程的属性和其他信息，可以从这个结构体中找到与进程相关的所有内核信息。
- 代表进程的数据结构指针形成了一个task数组，这种指针数组有时也称为指针向量。这个指针的大小由NR_TASK（默认为512）保存，表明Linux系统中最多能同时运行的进程数目。当建立新进程时，Linux为新进程分配一个task_struct结构，然后将指针保存在task数组中。调度程序一直维护一个current指针，它指向当前正在运行的进程。所以可以通过current查找到当前正在运行的进程的进程描述符。

进程描述符

- 内核通过一个**唯一的进程标识符**，即**PID号**来标识每个进程，每个进程都有一个非负的唯一进程ID(PID)。虽然是唯一的，但是PID可以重用，当一个进程终止后，其他进程就可以使用它的PID了。
- PID为0的进程为调度进程，该进程是内核的一部分，也称为系统进程；
- PID为1的进程为init进程，它是一个普通的用户进程，但是以超级用户特权运行；

进程描述符

- PID为2的进程是守护进程，负责支持虚拟存储系统的分页操作。除了PID，每个进程还有一些其他的标识符。
- 用来标识进程的结构体包含了很多信息，进程的task_struct结构包含了以下几个字段：
- 状态，调度信息，标识符
- 内部进程通讯
- 链接
- 时间和计数器
- 文件系统

进程描述符

- 虚拟内存
- 处理器的内容
- （1）状态
- 进程在运行时总是在不停地改变它的状态。在Linux系统中，有以下几个状态：
- 运行态：此时进程或者正在运行，或者准备运行。
- 等待态：此时进程在等待一个事件的发生或某种系统资源。
- 睡眠态：分为可中断的和不可中断的。可中断的进程可以被某一信号中断，而不可中断的进程将一直等待硬件状态的改变。

进程描述符

- 停止态：此时进程已经被中止。
- 死亡态：这是一个停止的进程，但还在进程向量数组中占有一个task_struct结构。
- **（2）调度信息**
- 调度算法需要此信息来决定系统中的那一个进程需要执行。
- **（3）标识符**
- 系统中的每一个进程都有一个进程标识符。进程标识符并不是指向进程向量的索引。每个进程同时还包括用户标识符和工作组标识符。

进程描述符

- （4）内部进程通讯
- Linux系统支持信号、管道、信号量等内部进程通讯机制。
- （5）链接
- 在Linux系统中，每个进程都和其他的进程有所联系，除了初始化进程，其他的进程都有父进程。一个新的进程一般都是由其他的进程复制而来的。task_struct结构中包括指向父进程，兄弟进程和子进程的指针。

- （6）时间和计时器
- 内核需要记录进程的创建时间和进程运行所占用的CPU的时间。Linux系统支持进程特殊间隔计时器。进程可以使用系统调用设置计时器，并当计时器失效时给进程一个信号。计时器可以是一次性的或周期性的。

进程描述符

- （7）文件系统

- 进程在运行时可以打开和关闭文件。task_struct结构中包括指向每个打开文件的文件描述符的指针，并且包括两个指向VFS索引节点的指针。VFS的索引节点用来在文件系统内唯一地描述一个文件或目录，并且提供文件系统操作的统一的接口。第一个索引节点是进程的根目录，第二个节点是当前的工作目录。两个VFS索引节点都有一个计数字段用来表明指向节点的进程数。

进程描述符

- （8）虚拟内存
- 大多数的进程都需要虚拟内存，Linux系统必须了解如何将虚拟内存映射到系统的物理内存。

- （9）处理器的内容
- 一个进程可以说是系统当前状态的总和。每当一个进程正在运行时，它都要使用处理器的寄存器及堆栈等资源；当一个进程挂起时，所有有关处理器的内容都要保存到进程的task_struct中；当进程恢复运行时，所有保存的内容再装入到处理器中。

进程描述符

- 在一个进程的task_struct中，有4对进程和组标识符：
- uid,gid：正在运行的进程用户标识符和组标识符。
- 有效uid和gid：有些程序可能将正在运行的进程的uid和gid改为自己所有，这些程序一般被称为setuid程序，它们十分有用，因为这是一种限制服务存取权限的方法。有效uid和gid是那些setuid程序的uid和gid，并且它们保持不变。每当内核检查权限时，都要检查有效uid和gid。
- 文件系统uid和gid：文件系统uid和gid一般和有效uid和gid相同，它们用于检查文件系统的存取权限。
- 保留uid和gid：它们用来和POSIX标准兼容，在程序通过系统调用改变进程的uid和gid时，它们可以保存真正的uid和gid。

进程的建立与退出

- 进程的创建
- Linux下的进程与线程相同点是都有进程控制块（PCB，具体的类是task_struct），区别在于一个有独立的进程资源，一个是共享的进程资源。内核线程完全没有用户空间，进程资源包括进程的PCB、线程的系统堆栈、进程的用户空间、进程打开的设备（文件描述符集）等。
- Linux用户进程不能直接被创建出来，因为不存在这样的API，它只能从某个进程中复制出来，有的需要通过exec这样的API来切换到实际想要运行的程序文件。

进程的建立与退出

- 复制API包括三种：fork、clone、vfork。
- 在linux源码中这三个调用的执行过程是执行fork(),vfork(),clone()时，通过一个系统调用表映射到sys_fork(),sys_vfork(),sys_clone(),再在这三个函数中去调用do_fork()去做具体的创建进程工作。**这三个API的内部实际都是调用一个内核内部函数do_fork，只是填写的参数不同而已。**

小测试

- 以下各项步骤中，哪个不是创建进程所必须的步骤?
- A.建立一个进程控制块PCB
- B.由CPU调度程序，为进程调度CPU
- C.为进程分配内存等必要的资源
- D.将CPU链入进程就绪队列

(答案：B)

进程的撤销

- Linux下进程的退出分为正常退出和异常退出两种:
- (1)正常退出
- a.在main()函数中执行return。
- b.调用exit()函数。
- c.调用_exit()函数。
- (2)异常退出
- a.调用abort函数
- b.进程收到某个信号，而该信号使程序终止。
- 不管是哪种退出方式，系统最终都会执行内核中的同一代码。这段代码用来关闭进程所用已打开的文件描述符，释放它所占用的内存和其他资源。

总结（关于父子进程）

- 父子进程之间的关系

- （1）父进程先于子进程终止，此种情况就是我们前面所用的孤儿进程。当父进程先退出时，系统会让init进程接管子进程。

- （2）子进程先于父进程终止，而父进程又没有调用wait函数，此种情况子进程进入僵死状态，并且会一直保持下去直到系统重启。子进程处于僵死状态时，内核只保存进程的一些必要信息以备父进程所需。此时子进程始终占有着资源，同时也减少了系统可以创建的最大进程数。

小测试

- 1.父进程一般可以撤销子进程，对吗?
- 2.进程的撤销是指，当一个进程完成指定任务后，操作系统收回该进程所占的（），取消该进程的（）

小测试

- 1.进程的等待状态是指等待占用CPU时的进程状态，对吗?
- 2.当进程需要的资源已经满足，或等待的事情已经发生，此时进程由等待状态变成什么状态?
- 3.当进程完成I/O后，进程由等待状态变成什么状态?

12.4 Linux中基本的进程通讯工具

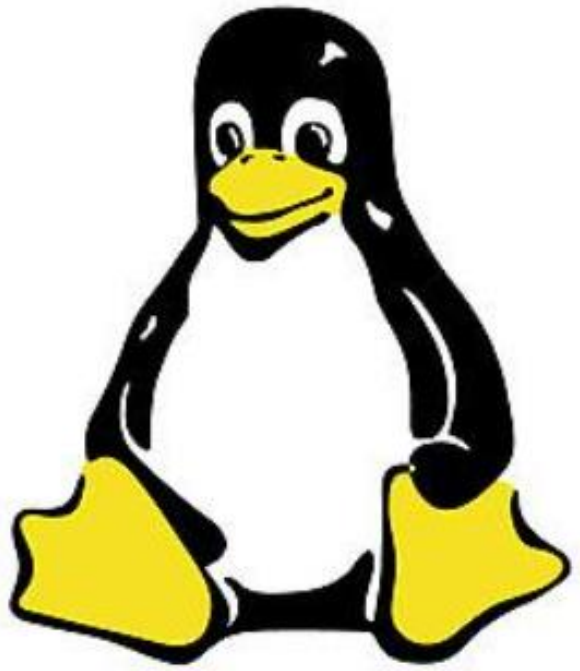
Linux/Unix系统中进程间通信的基本概念介绍

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。
- 本章将主要介绍在Linux中常用的多种通信方式以及它们进行协同通信的具体方法，详细说明进程间通信的多种形式：管道、命名管道(FIFO)、通常称为XSIIPC的3种形式的IPC(消息队列、信号量和共享存储），以及POSIX提供的替代信号量机制，通过相关概念及代码学习到在多进程环境中如何实现进程之间的相互通讯。

管道

- 管道是：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。
- 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

- 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。
- 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

- 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。

小测试

- 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

12.5 共享存储

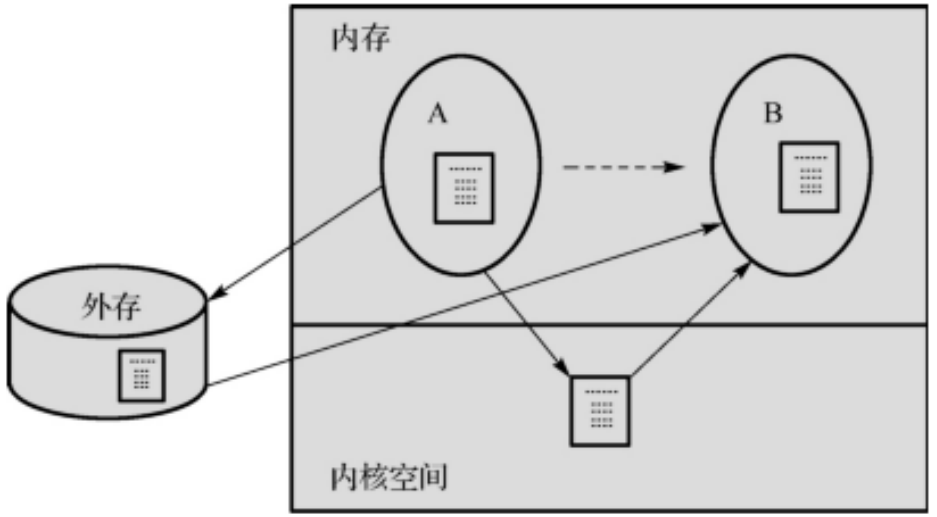
共享存储的介绍

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。

共享存储

- **Linux共享内存的实现**
- **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。

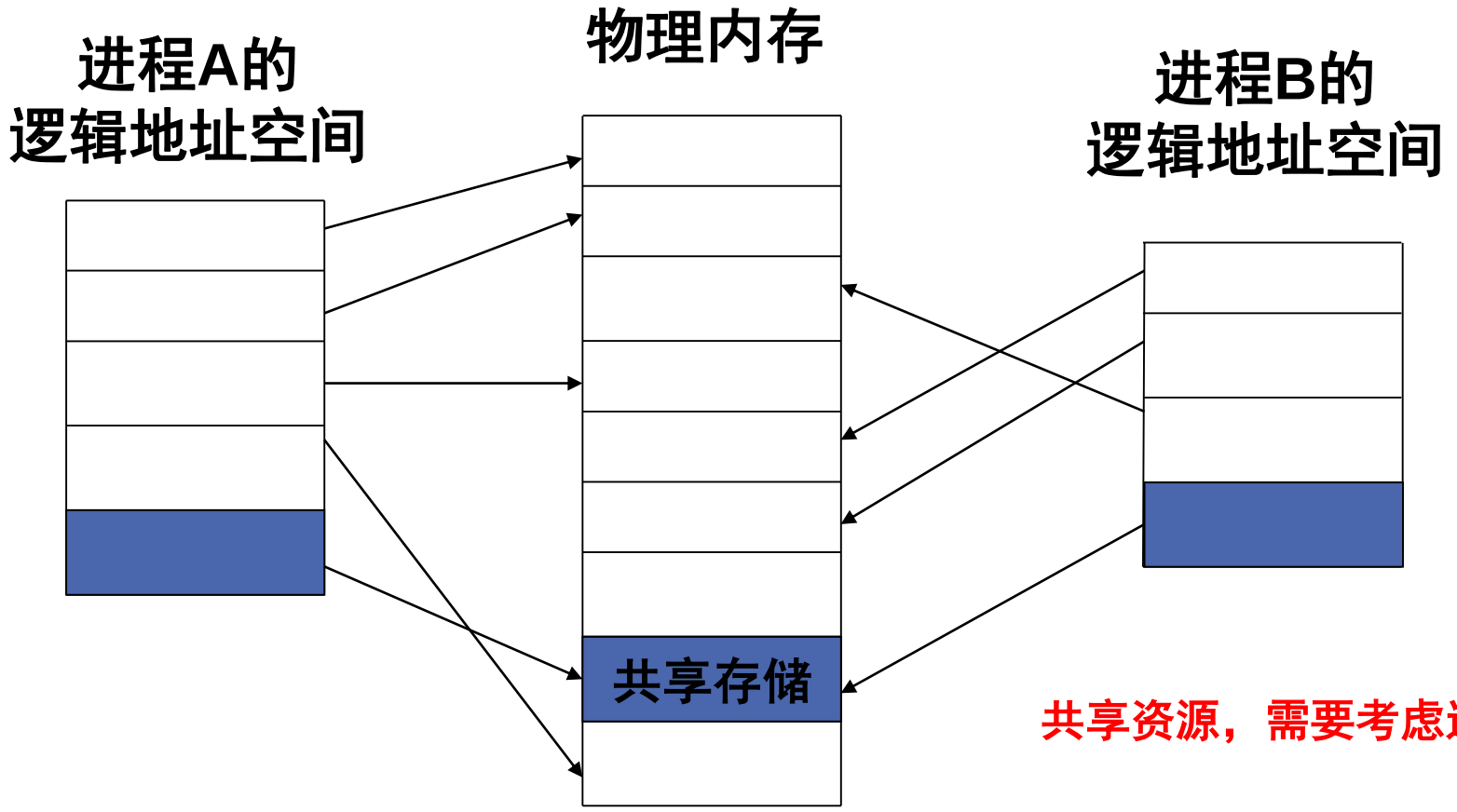
（2）共享内存绑定（映射共享内存区到调用进程地址空间）：需要通信的进程将先前创建的共享内存映射到自己的虚拟地址空间，使共享内存成为进程地址空间的一部分，随后可以像访问本地空间一样访问共享内存。

（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_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

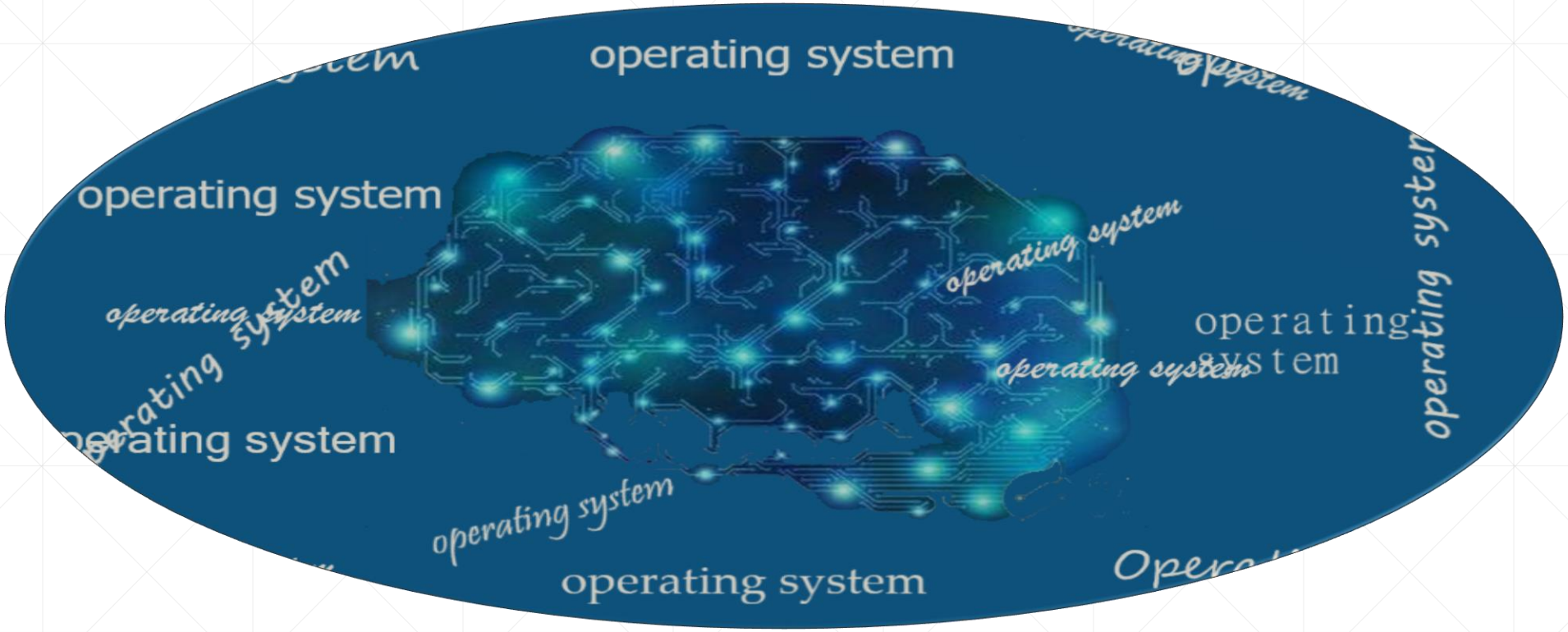
```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int semget(key_t key, size_t size, int shmflag);
```

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。

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



感谢观看！

授课教师：任立勇