第五章 Linux内核简介

主要内容

- 进程管理
- 文件系统
- 进程通信

进程管理

进程的概念
进程的结构
进程的操作

进程的概念

- 进程（process）
  - 处于执行期的程序及其所包含资源的总称

    程序：可执行程序代码

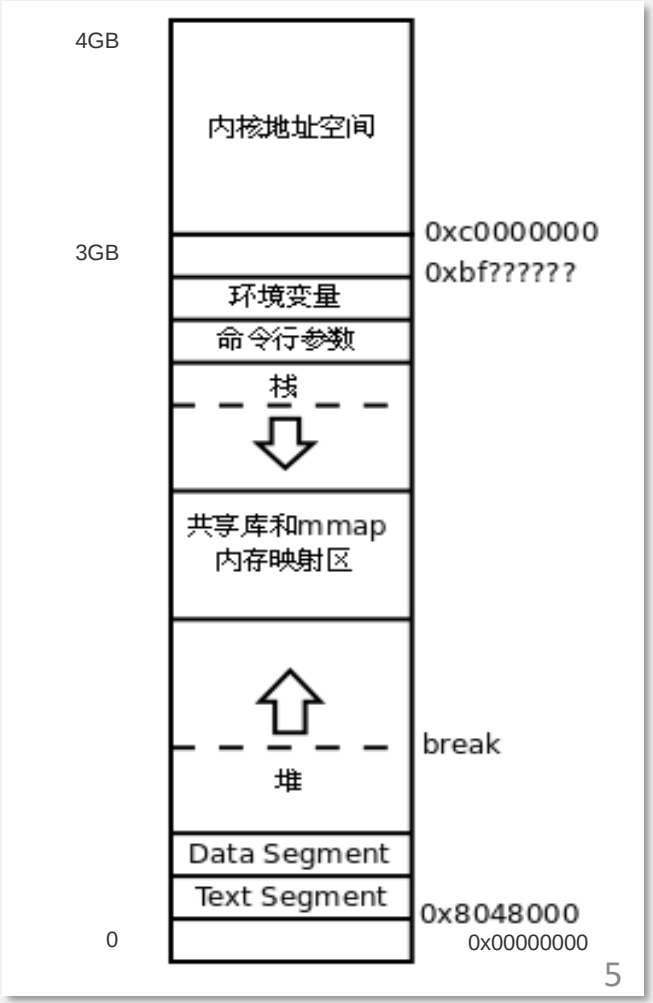
    资源：打开文件、信号、地址空间、数据段等

进程虚拟地址结构

- 以Linux系统为例（共4G空间）
  - 用户空间

    0x 0000 0000 ~ 0x bfff ffff
  - 内核空间

    0x c000 0000以上



进程的结构

- 进程描述符(进程控制块)
  - 数据结构

    task_struct
  - 定义位置

    /usr/src/kernels/3.10.0-693.el7.x86_64/include/linux/sched.h

进程的结构

- task_struct结构的主要成员
  - 身份信息(pid, uid, gid等)
  - 进程状态信息(state)
  - 调度信息(static_prio, normal_proi, run_list, array, policy)
  - 内存管理(mm, active_mm)
  - 家族信息(parent, children, sibling)
  - 进程时间信息(realtime, utime, stime, starttime)
  - 文件系统信息(link_count, fs, files)
  - IPC信息(signal, sighand, blocked, sigmask)

进程的结构

- 进程标识
  - 进程标识符pid

    一个无符号整型，取值范围：0 ~ 32767

    新建进程的pid通常是父进程的pid加1
- 用户相关的进程标识信息
  - 用户标识符uid：进程所有者的id
  - 组标识符gid：进程用户组的id

进程的结构

- 获取进程标识符的系统调用
  - 函数原型

```
#include <unistd.h>
#include <sys/types.h>
pid _t getpid(void);          //返回当前进程的PID
pid_t getppid(void);          //返回父进程的PID
uid_t getuid(void);           //返回当前进程用户的ID
gid_t getgid(void);           //返回当前进程用户组的ID
```

进程的结构

- 获取进程标识符的系统调用
  - 例1

```
#include <stdio.h>
#include <unistd.h>
main()
{
        printf("process id = %d\n", getpid());
        printf("parent process id = %d\n", getppid());
        printf("process's user id = %d\n", getuid());
        printf("process's group id = %d\n", getgid());
}
```

进程的结构

- 进程状态（ state ）
  - 可运行态(TASK_RUNNING)

    正在运行或已处于就绪状态的进程
  - 睡眠状态

    浅度睡眠状态(TASK_INTERRUPTIBLE)：等待资源被满足时被唤醒，可被信号或中断唤醒

    深度睡眠状态(TASK_UNINTERRUPTIBLE)：只能等待资源有效时唤醒，不可被信号唤醒

进程的结构

- 进程状态（state）
  - 暂停状态(TASK_STOPPED)：进程由于收到一个信号致使进程停止
  - 僵死状态(TASK_ZOMBIE)：进程执行结束但尚未消亡的状态，等待父进程回收

进程的结构

- 进程状态（state）
  - 状态定义

    /usr/src/kernels/3.10.0-693.el7.x86_64/include/linux/sched.h

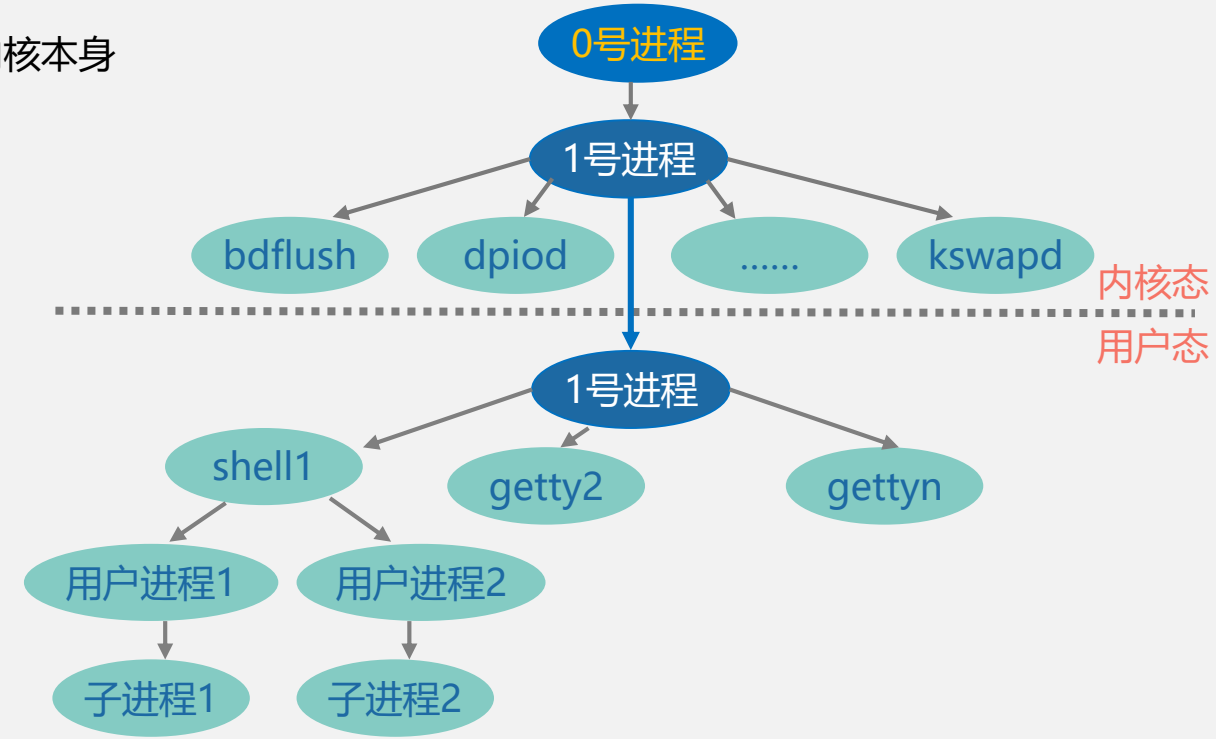
```
#define TASK_RUNNING            0
#define TASK_INTERRUPTIBLE      1
#define TASK_UNINTERRUPTIBLE    2
#define TASK_STOPPED            4
#define TASK_ZOMBIE             8
```

进程的结构

- 进程的家族关系
  - 所有进程都是init进程(pid=1)的后代
  - 每个进程必须有一个父进程
  - Linux系统中各个进程构成了树形的进程族系

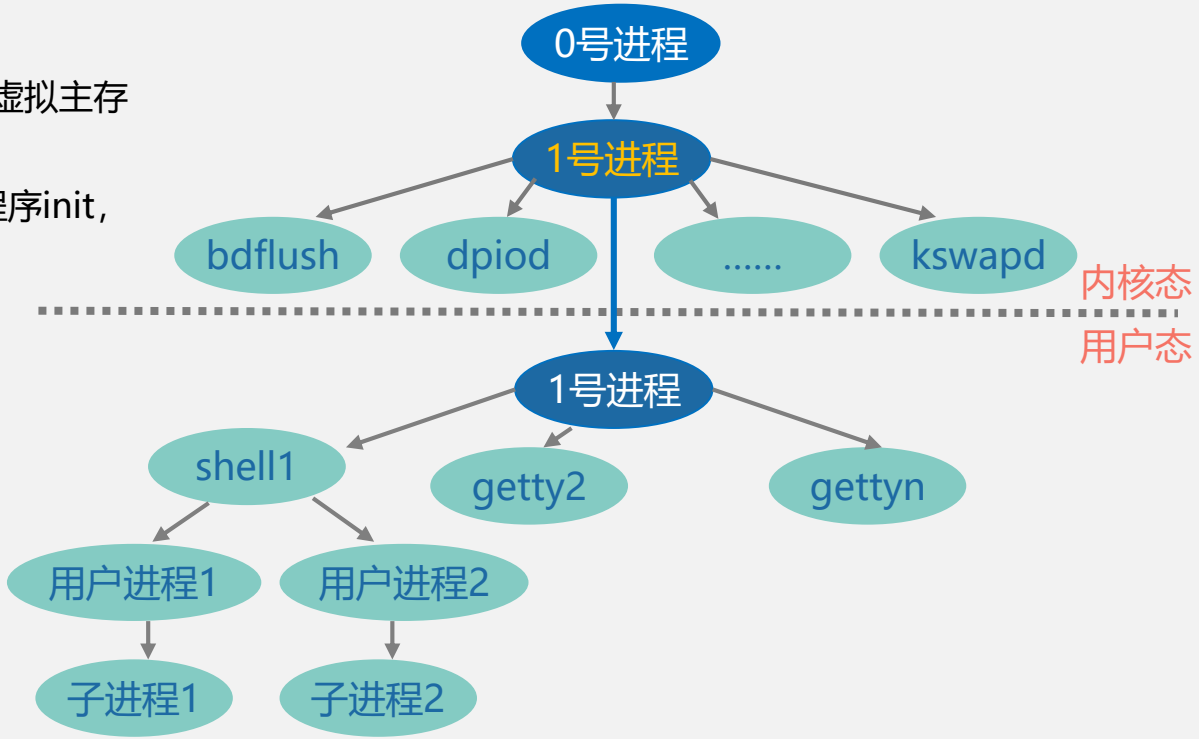
Linux系统启动过程

- 0号进程
  - 系统启动过程，实质为内核本身



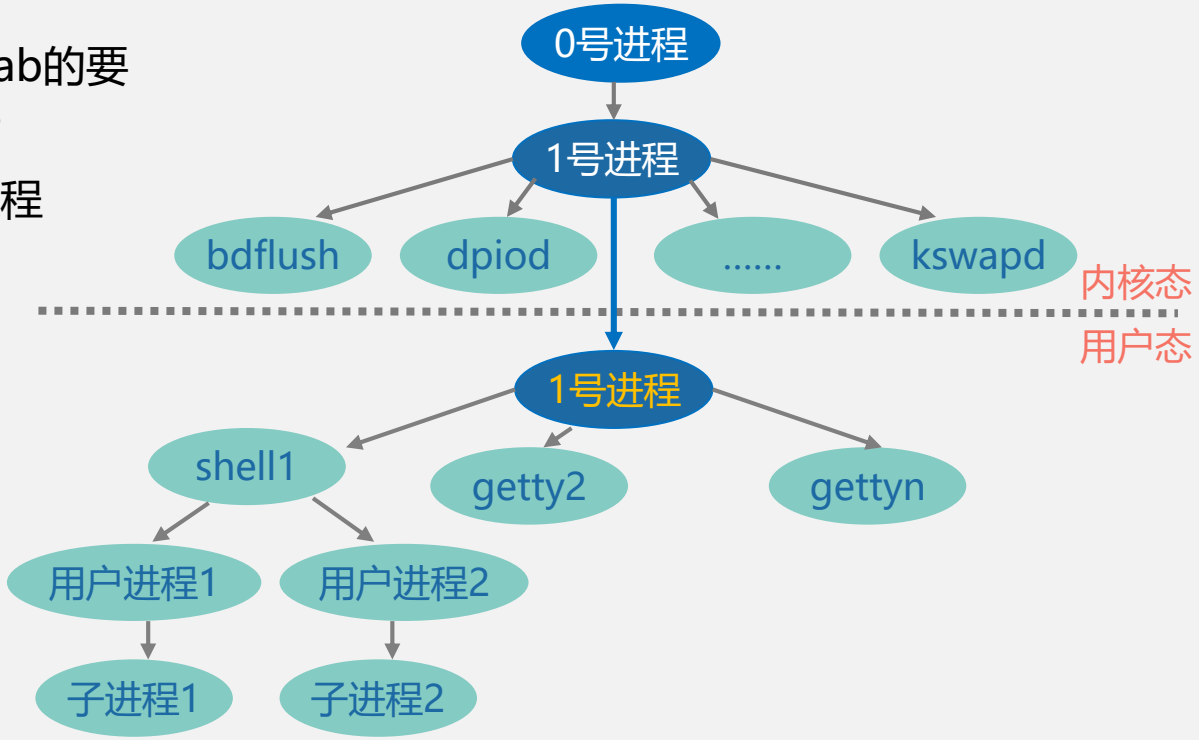
Linux系统启动过程

- 1号内核态进程
  - 初始化内核并进行系统配置
  - 创建若干个用于高速缓存和虚拟主存管理的内核线程
  - 调用execve( ) 装入可执行程序init，演变为1号用户态进程



Linux系统启动过程

- 首个用户进程
  - 按照配置文件/etc/initab的要求，完成系统启动工作
  - 创建若干个终端注册进程getty



进程的操作

- 进程的创建
- 进程映像的更换
- 进程的等待
- 进程的终止

进程的创建

- 函数原型

```
#include <unistd.h>
pid_t fork(void);
pid_t vfork(void);
```

fork()

- 函数原型

```
#include <unistd.h>
pid_t fork(void);
```

- 功能
  - 子进程完全复制父进程的资源
  - 子进程的执行独立于父进程

fork()

- 函数原型

```
#include <unistd.h>
pid_t fork(void);
```

- 返回值
  - 调用成功

    对于父进程，返回值为新创建子进程的PID

    对于子进程，返回值为0
  - 调用失败

    返回值为-1

fork()

- 例2：

```
//forktest1.c
#include <sys/types.h>
#include <unistd.h>
int main()
{
  pid_t val;
  printf ("PID before fork():%d\n", getpid ());
  val = fork ();
  if (val == 0)
   printf ("I am the child process,PID is %d\n", getpid ());
  else if (val > 0)
    printf ("I am the parent process,PID is %d\n", getpid ());
  else
    printf ("error in fork.");
    exit(0);
}
```

fork()

```
//父进程
int main()
{
  pid_t val;
  printf ("PID before fork():%d\n", getpid ());
  val = fork ();
  if (val == 0)
   printf ("I am the child process,PID is %d\n", getpid ());
  else if (val > 0)
    printf ("I am the parent process,PID is %d\n", getpid ());
  else
    printf ("error in fork.");
    exit(0);
}
```

```
//子进程
int main()
{
  pid_t val;
  printf ("PID before fork():%d\n", getpid ());
  val = fork ();
  if (val == 0)
   printf ("I am the child process,PID is %d\n", getpid ());
  else if (val > 0)
    printf ("I am the parent process,PID is %d\n", getpid ());
  else
    printf ("error in fork.");
    exit(0);
}
```

fork()

- 说明
  - fork()后会有两个并发进程执行
  - 子进程复制了父进程的数据段，包括全局变量

fork()

- 例3

```
//forktest2.c
#include <stdio.h>
int myvar = 0;
int main ()
{
  int pid;
  pid = fork ();
  if (pid < 0){
      printf ("fork failed.");
      exit (-1);
    }
  else if (pid == 0){
      printf ("child process executing.\n");
      myvar = 1;
    }
  else{
      wait ();
      printf ("child complete.\n");
      myvar++;
      printf ("father,myvar=%d\n", myvar);
      exit (0);
    }
}
```

vfork()

- 说明
  - vfork()创建的子进程与父进程共享地址空间
  - 调用vfork创建子进程后，父进程被挂起，直到子进程结束。因此，总是子进程先返回

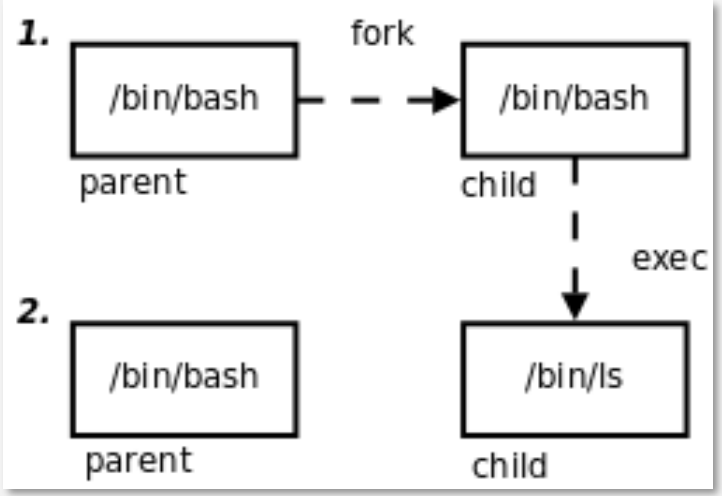
fork()

- 例3

```
//forktest2.c
#include <stdio.h>
#include <stdlib.h>
int myvar = 0;
int main ()
{
  int pid;
  pid = vfork ();
  if (pid < 0){
      printf ("fork failed.");
      exit (-1);
    }
  else if (pid == 0){
      printf ("child process executing.\n");
      myvar = 1;
    }
  else{
      wait ();
      printf ("child complete.\n");
      myvar++;
      printf ("father,myvar=%d\n", myvar);
      exit (0);
    }
}
```

进程映像的更换

- 功能
  - 用新的程序代码覆盖原先的父进程的代码
- Linux产生新进程的模式



exec()

- 6种形式

```
#include <unistd.h>
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char *const envp[ ]);
```

- execl系列：将每个命令行参数作为函数的参数传递。最后要以NULL为结束标志

exec()

- 6种形式

```
#include <unistd.h>
int execv(const char *path, char *const argv[ ]);
int execvp(const char *file, char *const argv[ ]);
int execve(const char *path, char *const argv[ ],char *const envp[ ]);
```

- execv系列：将所有参数包装到一个数组中传递

exec()

- 6种形式

```
#include <unistd.h>
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char *const envp[ ]);
int execv(const char *path, char *const argv[ ]);
int execvp(const char *file, char *const argv[ ]);
int execve(const char *path, char *const argv[ ],char *const envp[ ]);
```

- p：函数的第一个参数是程序文件名
- e：用envp数组的内容初始化环境参数

exec()

- 6种形式

```
#include <unistd.h>
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char *const envp[ ]);
int execv(const char *path, char *const argv[ ]);
int execvp(const char *file, char *const argv[ ]);
int execve(const char *path, char *const argv[ ],char *const envp[ ]);
```

- 返回值

  调用成功，无返回值

  调用失败，则返回-1

exec()

- 例4

```
//exectest.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
        pid_t pid;
        if( (pid=fork()) < 0 ){
                perror("fork");
                exit(0);
        }
        if (pid == 0){
                if (execl("/bin/ls", "ls", "-l",NULL) == -1){
                        perror("execl");
                        exit(0);
                }
        exit(0);
        }
}
```

进程的等待

- 函数原型

```
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int option) ;
```

- 功能
  - 父进程等待当前进程的子进程终止，获取子进程终止的状态，并回收子进程占用的系统资源

wait()

- 函数原型

```
#include <sys/wait.h>
pid_t wait(int *status);
```

- 参数
  - status：指向int类型的指针，保存子进程退出时的状态
    如果不关心子进程的退出状态，则设置status的值为NULL
- 返回值
  - 如果调用成功，返回值是已终止的子进程的PID
  - 如果调用失败，返回值为-1

wait()

- 说明
  - 调用wait()不仅可以获得子进程的终止信息，还可以使父进程阻塞等待子进程终止，从而起到进程间同步的作用

wait()

例5：

```
//waittest.c
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
main()
{
        pid_t pc,pr;
        pc=fork();
        if (pc<0)
                printf("error ocurred!\n");
        else if (pc==0){
                printf("this is child process with pid of %d\n", getpid());
                sleep(10);
        }
        else{
                pr=wait(NULL);
                printf("I catched a child process with pid of %d\n", pr);
        }
        exit(0);
}
```

进程的终止

- 函数原型

```
#include <stdlib.h>
void exit(int status);
```

- 功能
  - 终止正在运行的程序
  - 该函数将回收与进程相关的各种内核数据结构，把进程的状态置为TASK_ZOMBIE。等待父进程调用wait()对其资源进行回收

进程的终止

- 函数原型

```
#include <stdlib.h>
void exit(int status);
```

- 参数说明
  - status：用来传递进程结束时的状态

    一般来说，0表示没有意外的正常结束

    其他数值表示进程出现了错误，非正常结束

文件系统

文件操作的系统调用

ext2文件系统

虚拟文件系统

文件描述符

- 概念
  - 当进程创建一个新文件或者打开现有的文件时，系统内核向进程返回一个文件描述符
  - 当对文件进行I/O操作时，大多数函数都用文件描述符作为参数，代表要操作的文件

文件描述符

- 三个特殊的文件描述符
  - 程序启动时会自动打开三个文件

    标准输入文件

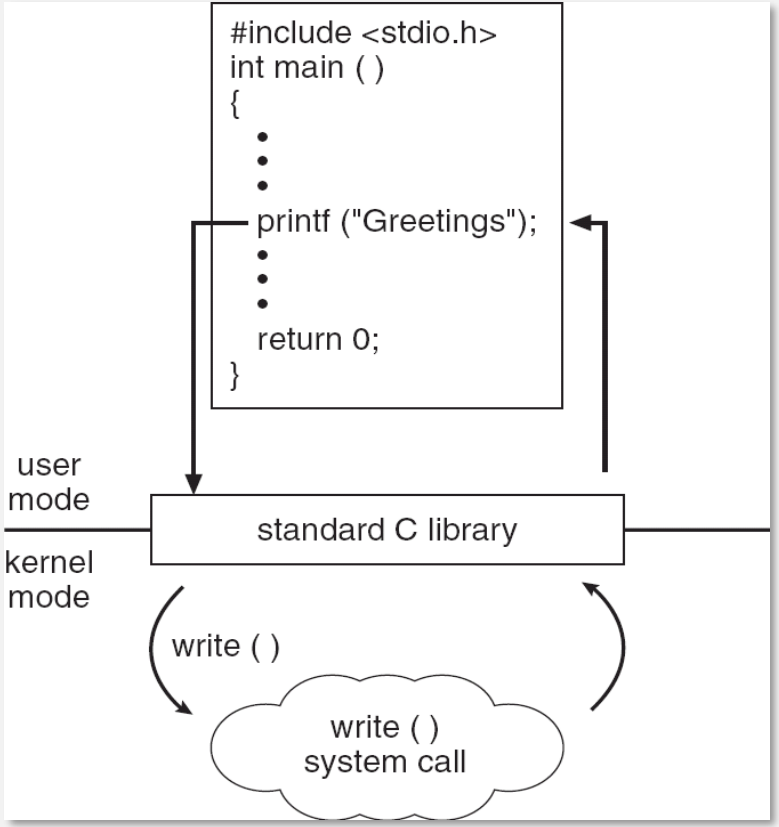
    标准输出文件

    标准错误输出文件
  - 对应的文件描述符如下

```
#define STDIN_FILENO     0     //标准输入文件
#define STDOUT_FILENO    1     //标准输出文件
#define STDERR_FILENO    2     //标准错误输出文件
```

文件操作的系统调用

- 库函数和系统调用的关系



文件操作的系统调用

- 库函数和系统调用的区别
  - 在所有支持C语言的平台上都可以用C标准库函数，而只有在UNIX/Linux平台上才能使用read等系统调用
  - C标准I/O库函数在头文件stdio.h中声明，而read、write等函数在头文件unistd.h中声明

文件操作的系统调用

- creat()
- open()
- read()
- write()
- close()
- lseek()
- chown()
- chmod()

creat()

- 函数原型

```
#include<sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int creat(const char *pathname, mode_t mode);
```

- 功能
  - 创建一个普通文件

creat()

- 函数原型

```
#include<sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int creat(const char *pathname, mode_t mode);
```

- 参数
  - pathname：要创建文件的路径(绝对路径/相对路径)
  - mode：创建文件的访问权限(八进制数/宏定义)
- 返回值
  - 调用成功，则返回文件描述符
  - 调用失败，则返回-1

creat()

- 参数mode的说明
  - 定义在头文件 include/linux/stat.h中

```
#define S_IRWXU 00700          //文件主读、写、执行
#define S_IRUSR 00400          //文件主读
#define S_IWUSR 00200          //文件主写
#define S_IXUSR 00100          //文件主执行

#define S_IRWXG 00070          //同组用户读、写、执行
#define S_IRGRP 00040          //同组用户读
#define S_IWGRP 00020          //同组用户写
#define S_IXGRP 00010          //同组用户执行

#define S_IRWXO 00007          //其他用户读、写、执行
#define S_IROTH 00004          //其他用户读
#define S_IWOTH 00002          //其他用户写
#define S_IXOTH 00001          //其他用户执行
```

creat函数

- 参数mode的说明
  - 例子

    creat("hello.txt",S_IRUSR|S_IWUSR);

    creat("hello.txt", 0600);

open()

- 函数原型

```
#include<sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

- 功能
  - 打开或者创建一个文件

open()

- 函数原型

```
#include<sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

- 参数
  - pathname：要打开文件的路径（绝对路径/相对路径）
  - flags：打开文件的方式
  - mode：可选参数，只在创建新文件时有效。用于定义新建文件的访问权限

open()

- 函数原型

```
#include<sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

- 返回值
  - 调用成功，则返回文件描述符
  - 调用失败，则返回-1

open()

参数flags说明

flags	含义	可选
O_RDONLY	以只读方式打开文件	三选一
O_WRONLY	以只写方式打开文件	
O_RDWR	以读写方式打开文件	
O_APPEND	以追加方式打开	可选
O_CREAT	若打开文件不存在，则创建此文件	
O_TRUNC	如果文件存在，且以只读或只写方式打开，则将文件长度截为0	

open()

- 参数flags说明
  - 例子

```
open ("hello.txt", O_RDONLY);
open ("test.txt", O_CREAT | O_WRONLY | O_TRUNC, 0644);
creat ("test.txt", 0644);
```

close()

- 函数原型

```
#include <unistd.h>
int close(int fd);
```

- 功能
  - 关闭一个已打开的文件

close()

- 函数原型

```
#include <unistd.h>
int close(int fd);
```

- 参数
  - fd：要关闭文件的描述符
- 返回值
  - 调用成功，返回0
  - 调用失败，返回-1

read()

- 函数原型

```
#include <unistd.h>
ssize_t   read ( int fd, const void *buf, size_t count );
```

- 功能
  - 从一个打开的文件中读取数据

read()

- 函数原型

```
#include <unistd.h>
ssize_t   read ( int fd, const void *buf, size_t count );
```

- 参数
  - fd：文件描述符
  - buf：缓冲区指针，用于缓存从文件中读取的数据
  - count：请求读取的字节数

read()

- 函数原型

```
#include <unistd.h>
ssize_t   read ( int fd, const void *buf, size_t count );
```

- 返回值
  - 调用成功，返回本次实际读取的字节数（有些情况下，实际读到的字节数会小于请求读的字节数）
  - 调用失败，返回-1

write()

- 函数原型

```
#include <unistd.h>
ssize_t write ( int fd, const void *buf, size_t count );
```

- 功能
  - 向一个打开的文件写入数据

write()

- 函数原型

```
#include <unistd.h>
ssize_t write ( int fd, const void *buf, size_t count );
```

- 参数
  - fd：文件描述符
  - buf：缓冲区指针，准备写入文件的数据
  - count：要写入文件的字节数
- 返回值
  - 调用成功，返回实际写入的字节数
  - 调用失败，返回-1

文件操作的例子

- 例1

```
//test1.c
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>

int main()
{
        int fd;
        char buf[256];
        fd = creat("hello.txt",0644);
        write(fd, "hello world!\n",13);
        close(fd);
}
```

文件操作的例子

- 说明
  - 对于普通文件，写操作从当前的文件读写指针的位置开始写
  - 如果打开文件时，指定了O_APPEND选项，则从文件末尾开始写

文件操作的例子

- 例2

```
//test2.c
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>

int main()
{
        int fd;
        char buf[256];
        fd = open("hello.txt", O_WRONLY |O_APPEND );
        write(fd, "this is second line!\n", 21);
        close(fd);
}
```

文件操作的例子

- 说明
  - 用于普通文件的处理方法同样适用于设备和其他特殊文件

文件操作的例子

- 例3

```
//test3.c
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>

int main()
{
        int fd, n;
        char buf[256];
        fd = open("hello.txt", O_RDONLY);
        while( (n=read(fd, buf, 256)) > 0 )
        {
                write(STDOUT_FILENO, buf, n);
        }
        close(fd);
}
```

文件操作的例子

- 例4

```
int main(int argc, char *argv[])
{
        int fd1, fd2, n;
        char buf[512],ch='\n';
        if (argc <= 2)
        {
                printf("you forgot the enter the filename\n");
                exit(1);
        }
        fd1= open(argv[1], O_RDONLY, 0644);
        fd2= creat(argv[2], 0644);
        while((n=read(fd1, buf, 512)) > 0)
                write(fd2, buf, n);
        close(fd1);
        close(fd2);
}
```

lseek()

- 函数原型

```
#include <unistd.h>
#include <sys/types.h>
off_t lseek(int fd, off_t offset, int whence);
```

- 功能
  - 对文件的读写指针进行移动，实现文件的随机读写

lseek()

- 函数原型

```
#include <unistd.h>
#include <sys/types.h>
off_t lseek(int fd, off_t offset, int whence);
```

- 参数
  - fd：文件描述符
  - offset：文件偏移量。offset是个相对偏移量，偏移的基准是由whence参数决定的
  - whence：偏移的基准位置

lseek()

- 参数whence的含义

whence	含义
SEEK_SET	从文件的开头计算偏移量
SEEK_CUR	从当前的位置计算偏移量
SEEK_END	从文件的末尾开始加offset后的偏移量

- 例子

```
lseek(fd, 10, SEEK_SET );          //从文件开头偏移10个字节
```

lseek()

- 函数原型

```
#include <unistd.h>
#include <sys/types.h>
off_t lseek(int fd, off_t offset, int whence);
```

- 返回值
  - 调用成功，返回相对于文件开头的实际偏移量
  - 调用失败，则返回-1

lseek()

- 例5

```
int main()
{
        int fd;
        char buf1[]="abcdefghij";
        char buf2[]="ABCDEFGHIJ";
        if ((fd=creat("file.hole",0644 )) < 0 )
                perror("creat error");
        if (write(fd, buf1, 10) != 10 )
                perror("buf1 write error");
        if (lseek(fd, 20, SEEK_SET) == -1)
                perror("lseek error");
        if (write(fd, buf2, 10) != 10)
                perror("buf2 write error");
        exit(0);
}
```

chown()

- 函数原型

```
#include <sys/types.h>
#include <unistd.h>
int chown(const char *path, uid_t owner, gid_t group);
```

- 功能
  - 改变文件的所有者

chown()

- 函数原型

```
#include <sys/types.h>
#include <unistd.h>
int chown(const char *path, uid_t owner, gid_t group);
```

- 参数
  - path：文件路径名
  - owner：变更的uid
  - group：变更的gid

chown()

- 函数原型

```
#include <sys/types.h>
#include <unistd.h>
int chown(const char *path, uid_t owner, gid_t group);
```

- 返回值
  - 成功，返回0
  - 失败，返回-1

chown()

- 例6

```
main()
{
        if (chown("hello.txt",0,0) == -1)
        {
                perror(errno);
        }
        else
        {
                write(STDERR_FILENO, "Call chown success.\n",20);
        }
}
```

chmod()

- 函数原型

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char *path, mode_t mode);
```

- 功能
  - 改变文件的访问权限

chmod()

- 函数原型

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char *path, mode_t mode);
```

- 参数
  - path：文件路径名
  - mode：权限组合
- 返回值
  - 成功，返回0
  - 失败，返回-1

chmod()

- 例7

```
main()
{
        if (chmod("hello.txt",0600) == -1)
        {
                perror(errno);
        }
        else
        {
                write(STDOUT_FILENO,"call chmod success.\n",20);
        }
}
```

进程通信（IPC）

- 传统进程通信
  - 信号（signal）
  - 管道（pipe）
- System V IPC进程通信
  - 消息队列（message）
  - 共享主存（shared memory）
  - 信号量（ semaphore ）
- 远程通信
  - 套接字（socket）

信号

- 概念
  - 信号机制是在软件层次上对中断机制的一种模拟
  - 信号的捕获与处理也称为系统的“软中断”机制
- 信号事件来源
  - 硬件来源

    硬件操作，如键盘操作：Ctrl+C
  - 软件来源

    使用信号发送函数，如：kill()

    非法运算，如：浮点运算溢出或内存访问错误等

常用信号

- 说明
  - 每个信号都有一个编号和一个宏定义名称，名称都以SIG开头
  - 宏定义在signal.h中

```
[wuhua@localhost ~]$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

常用信号

信号值	信号名	功能
1	SIGHUP	从终端上发出的结束信号
2	SIGINT	来自键盘的中断信号（Ctrl-c）
3	SIGQUIT	来自键盘的退出信号（Ctrl-\）
8	SIGFPE	浮点异常信号（例如浮点运算溢出）
9	SIGKILL	终止接收信号的进程
14	SIGALRM	进程的定时器到期时，发送该信号
15	SIGTERM	kill命令发出的信号
17	SIGCHLD	标识子进程停止或结束的信号
19	SIGSTOP	来自键盘(Ctrl-z)或调试程序的停止执行信号

信号的处理

- 缺省操作
  - 内核为信号提供默认处理程序处理
- 忽略信号
  - 进程忽略接收到的信号，不做任何处理
  - SIGKILL和SIGSTOP不能忽略
- 捕获信号
  - 提供一个信号处理函数，要求内核在处理该信号时切换到用户态执行这个处理函数

信号处理函数

- 注册信号处理函数
  - signal()
  - sigaction()
- 信号发送函数
  - kill()
  - alarm()
  - pause()
  - sigqueue()

signal()

- 函数原型

```
#include <signal.h>
void ( *signal( int signum, void( (*func)(int ) ) ) (int) ;
```

- 功能
  - 定义进程收到信号后的处理方法

signal()

- 函数原型

```
#include <signal.h>
void ( *signal( int signum, void( (*func)(int ) ) ) (int) ;
```

- 参数
  - signum：信号名称，整型值
  - func：指向信号处理函数的指针。该参数还可以是
    SIG_IGN：忽略该信号的操作函数
    SIG_DFL：缺省信号操作函数
  - 返回值
    调用成功，返回值为原来的信号处理函数指针
    调用失败，则返回SIG_ERR

signal()

- 例1：signaltest1.c

```
void HandleSigint(int signo)
{
        printf("receive signal %d.\n",signo);
}

main()
{
        if ( signal(SIGINT, HandleSigint) == SIG_ERR )
        {
                perror("signal");
                exit(0);
        }
        pause();
}
```

signal函数

- 例2：signaltest2.c

```c
void Sigroutine(int signo)
{
        switch(signo){
        case 2: printf("Get a signal SIGINT\n");break;
        case 3: printf("Get a signal SIGQUIT\n");break;
        case 15: printf("Get a signal SIGTERM\n");break;
        }
}

int main()
{
        printf("process id is %d.\n",getpid());
        signal(SIGTERM, Sigroutine);
        signal(SIGINT, Sigroutine);
        signal(SIGQUIT, Sigroutine);
        for(;;);
}
```

kill()

- 函数原型

```
#include <signal.h>
int kill(pid_t pid, int sig);
```

- 功能
  - 向进程或进程组发送一个信号

kill()

- 函数原型

```
#include <signal.h>
int kill(pid_t pid, int sig);
```

- 参数
  - pid：接收信号的进程ID

    pid>0：发送给进程号为pid的进程

    pid=0：发送给同组的所有进程
  - sig：信号名称

kill()

- 函数原型

```
#include <signal.h>
int kill(pid_t pid, int sig);
```

- 返回值
  - 信号发送成功，返回0
  - 信号发送失败，返回-1

kill函数

- 例3：killtest.c

```c
int main()
{
        pid_t pid;
        int num;
        if ((pid=fork()) < 0)
                perror("fork");
        else if (pid == 0)
                sleep(10);
        else
        {
                printf("killing process %d\n",pid);
                if ((kill(pid, SIGTERM)) < 0)
                        perror("kill:SIGTERM");
                waitpid(pid, NULL, 0);
                printf("catch a child process with pid of %d\n", pid);
        }
        exit(0);
}
```

管道

- 概念
  - 连接两个进程的连接器（特殊文件）
  - 管道是单向的
- 分类
  - 普通管道（pipe）

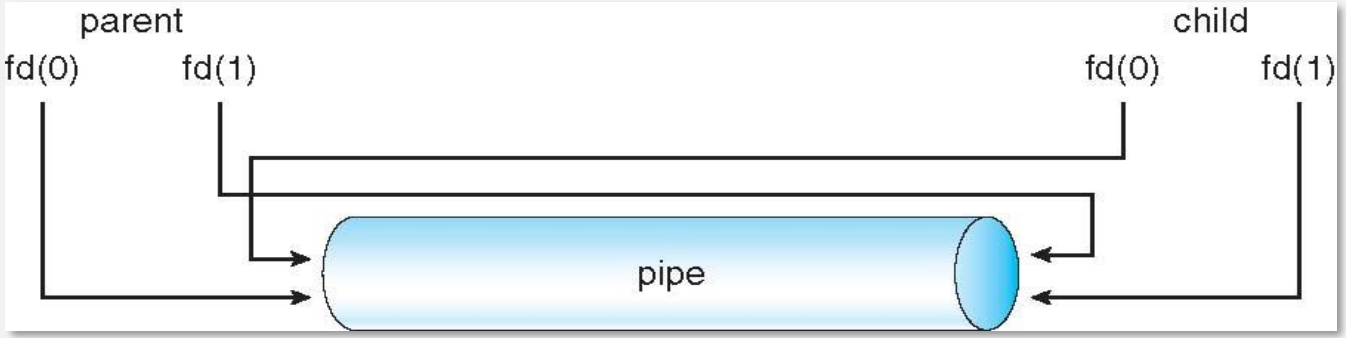
    只能用于有亲缘关系的进程之间的通信
  - 命名管道（named pipe）

    实现无亲缘关系进程间的通信

普通管道

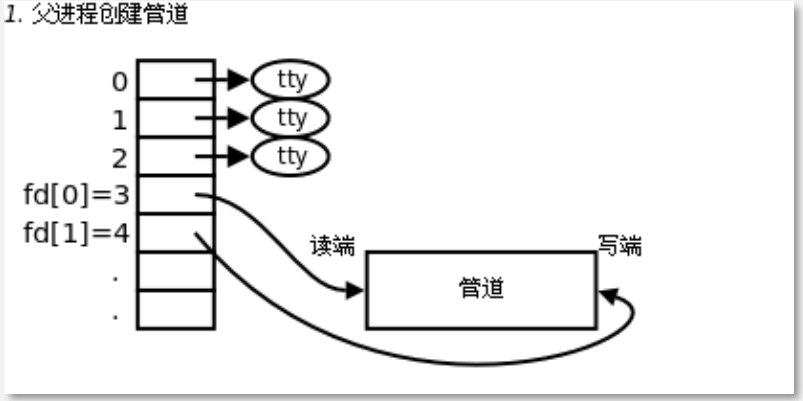
说明

- 当一个管道建立后，将获得两个文件描述符，分别用于对管道读取和写入，通常将其称为管道的写端和管道的读端

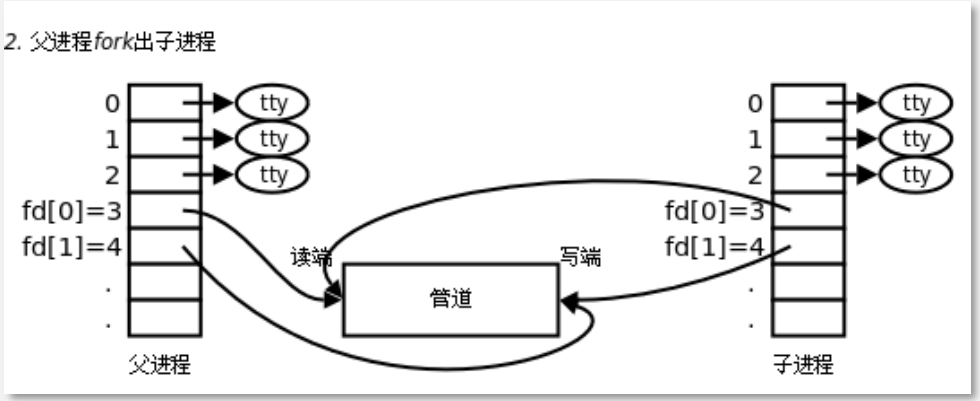


普通管道如何实现两个进程间的通信

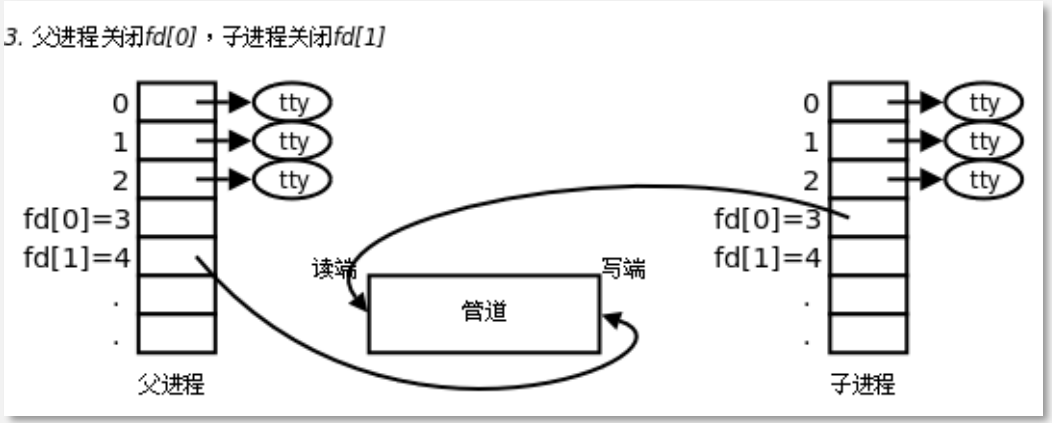
1. 父进程创建管道



2. 父进程fork出子进程



3. 父进程关闭fd[0]，子进程关闭fd[1]



pipe()

- 函数原型

```
#include <unistd.h>
int pipe(int filedes[2]);
```

- 功能
  - 创建一个普通管道

pipe()

- 函数原型

```
#include <unistd.h>
int pipe(int filedes[2]);
```

- 参数
  - filedes数组：存放两个文件描述符
    filedes[0]：存放管道读端文件描述符
    filedes[1]：存放管道写端文件描述符
- 返回值
  - 调用成功，返回0
  - 调用失败，返回-1

管道的读写操作

- 读规则
  - 关闭管道的写端：close (fd[1])
  - 读出：read(fd[0], buf, size); 从管道读端口fd[0]读出size个字符放到buf中
  - 读完关闭管道的读端：close(fd[0])
- 写规则
  - 关闭管道的读端：close(fd[0])
  - 写入：write(fd[1], buf, size); 把buf中的长度为size的字符送到管道写端口fd[1]
  - 写完关闭管道的写端：close (fd[1])

pipe()

- 例4：pipetest.c

```c
#define MAXLINE 80
int main()
{
        int n;
        int fd[2];
        pid_t pid;
        char line[MAXLINE];
        if (pipe(fd) < 0){
                perror("pipe");
                exit(1);
        }
        if ((pid = fork()) < 0){
                perror("fork");
                exit(1);
        }
        if (pid>0){
                close(fd[0]);
                write(fd[1], "hello world\n", 12);
                wait(NULL);
        }
        else{
                close(fd[1]);
                n = read(fd[0], line, MAXLINE);
                write(STDOUT_FILENO, line, n);
        }
        return 0;
}
```

普通管道

- 说明
  - 两个进程通过一个管道只能实现单向通信，如果需要父子进程双向通信，就必须另开一个管道
  - 管道的读写端通过打开的文件描述符来传递，只能用于有亲缘关系的进程之间的通信
  - 在管道满时，写管道操作将被阻塞；在管道空时，读管道操作将被阻塞
  - 管道在使用完毕后，应该调用close关闭

命名管道

- 命名管道与普通管道的区别
  - 普通管道位于内存，只能用于有亲缘关系的进程间通信
  - 命名管道位于文件系统中，可以实现不同进程间的通信
  - 命名管道是一种双向通信管道，可以以读/写模式打开

mkfifo()

- 函数原型

```
#include <sys/types.h>
#include <unistd.h>
int mkfifo(const char * pathname, mode_t mode);
```

- 功能
  - 创建命名管道
  - 在文件系统中将产生一个FIFO文件

mkfifo()

- 函数原型

```
#include <sys/types.h>
#include <unistd.h>
int mkfifo(const char * pathname, mode_t mode);
```

- 参数
  - pathname：创建的FIFO文件名
  - mode：规定FIFO文件的读写权限
- 返回值
  - 成功时，返回0
  - 失败时，返回-1

命名管道

- 例5：创建命名管道。mkfifotest.c

```c
int main()
{
        if (mkfifo("p1", 0644) < 0)
        {
                perror("mkfifo");
                exit(-1);
        }
        return 0;
}
```

命名管道

- 例5：以只写方式打开管道。writepipe.c

```
int main()
{
        int fd;
        if ((fd=open("p1", O_WRONLY)) < 0)
        {
                perror("open");
                exit(-1);
        }
        write(fd, "hello world.\n", 13);
        printf("open fifo p1 for write success.\n");
        close(fd);
}
```

命名管道

- 例5：以只读方式打开管道。readpipe.c

```c
int main()
{
        int fd, n;
        char buf[256];
        if ((fd=open("p1", O_RDONLY)) < 0)
        {
                perror("open");
                exit(-1);
        }
        while((n=read(fd, buf, 512)) >0 )
        {
                write(STDOUT_FILENO, buf, n);
        }
        printf("open fifo p1 for read success.\n");
        close(fd);
}
```

System V进程通信

- IPC (InterProcess Communication)
  - 消息队列（ Message Queues ）

    以异步方式为通信频繁、但数据量少的进程通信提供服务
  - 共享内存（ Shared Memory ）

    为数据量大的进程间通信提供服务
  - 信号量集（ Semaphore Arrays ）

    实现进程同步

IPC

- 查看ipc对象的信息
  - 格式

    ipcs [-asmq]
  - 参数说明

    -a　　//查看全部IPC对象信息

    -m　　//查看共享内存

    -q　　//查看消息队列

    -s　　//查看信号量集

IPC

查看ipc对象的信息

```
[wuhua@localhost ~]$ ipcs

--------- 消息队列 -----------
键         msqid      拥有者   权限     已用字节数 消息

------------ 共享内存段 --------------
键         shmid      拥有者   权限     字节     nattch     状态
0x00000000 491520     wuhua      600        524288     2          目标
0x00000000 2064385    wuhua      600        393216     2          目标
0x00000000 3309570    wuhua      600        4194304    2          目标
0x00000000 1081348    wuhua      600        524288     2          目标

--------- 信号量数组 -----------
键         semid      拥有者   权限     nsems
```

ftok()

- 函数原型

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(char *filename, int id);
```

- 功能
  - 根据文件名（必须是可访问的）和一个整型变量生成一个唯一的键值

ftok()

- 函数原型

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(char *filename, int id);
```

- 参数
  - filename：文件名，可以使用绝对路径或相对路径
  - id：整型变量。指定相同的文件名和不同的整型变量，可生成不同的键值
- 返回值
  - 调用失败，返回-1
  - 调用成功，返回其他

ftok()

- 例6：ftoktest.c

```c
int main()
{
        key_t key1, key2, key3;
        if ( (key1=ftok("/etc/profile",1)) == -1 )
        {
                perror("ftok");
                exit(1);
        }
        if ( (key2=ftok("/etc/profile",2)) == -1 )
        {
                perror("ftok");
                exit(2);
        }
        if ( (key3=ftok("/etc/profile",1)) == -1 )
        {
                perror("ftok");
                exit(3);
        }
        printf("key1=%d\nkey2=%d\nkey3=%d\n",key1, key2, key3);
        return 0;
}
```

消息队列

- 概念
  - 存储消息的线性表
  - 消息队列的数据的输入输出按照FIFO的原则进行

消息队列

- 消息队列资源的限制
  - 消息队列最多个数：1949
  - 每条消息的最大字节数：8KB
  - 每个队列的最大字节数：16KB

消息队列

- 消息结构模板msgbuf
  - include/linux/msg.h

```
struct msgbuf
{
        long msgtype;           //消息的类型，长整型变量
        char mtext[1];          //消息的内容，字符串数组
};
```

  - 说明

    该结构只是一个模版，在实际的编程中，可以根据该模版自行定义消息的长度

消息队列

- 基本操作
  - 创建消息队列 - msgget()
  - 向消息队列发送消息 - msgsnd()
  - 从消息队列读取消息 - msgrcv()
  - 删除消息队列 - msgctl()

msgget()

- 函数原型

```
#include <sys/types.h>
#include <sys/msg.h>
int msgget(key_t key, int flags) ;
```

- 功能
  - 用于创建或打开一个消息队列

msgget()

- 函数原型

```
#include <sys/types.h>
#include <sys/msg.h>
int msgget(key_t key, int flags) ;
```

- 参数
  - key：键值

    可以通过ftok函数产生一个键值

    IPC_PRIVATE：创建当前进程的私有消息队列

msgget()

- 函数原型

```
#include <sys/types.h>
#include <sys/msg.h>
int msgget(key_t key, int flags) ;
```

- 参数
  - flags：标识和权限信息的组合

    IPC_CREAT：如果消息队列不存在，则创建一个新的队列；如果消息队列存在，则引用已存在的消息队列

    0：获取一个已存在的消息队列的标识符

msgget()

- 函数原型

```
#include <sys/types.h>
#include <sys/msg.h>
int msgget(key_t key, int flags) ;
```

- 返回值
  - 调用成功，返回消息队列的标识符
  - 调用失败，返回-1
- 例子

```
key=ftok("/etc/profile",1);
qid=msgget(key, IPC_CREAT|0666);
```

msgsnd()

- 函数原型

```
#include <sys/types.h>
#include <sys/msg.h>
int msgsnd(int msqid, struct msgbuf *msgp, size_t size, int flag);
```

- 功能
  - 向消息队列发送消息

msgsnd()

- 函数原型

```
#include <sys/types.h>
#include <sys/msg.h>
int msgsnd(int msqid, struct msgbuf *msgp, size_t size, int flag);
```

- 参数
  - msqid：消息队列的标识符
  - msgp：指向消息结构指针
  - size：消息内容的长度
  - flag：发送消息可选标志

    0：当消息队列已满时，消息的发送操作阻塞，直到有进程从队列中读出消息

    IPC_NOWAIT：当消息队列满时，进程不阻塞，立即返回-1

msgsnd()

- 函数原型

```
#include <sys/types.h>
#include <sys/msg.h>
int msgsnd(int msqid, struct msgbuf *msgp, size_t size, int flag);
```

- 返回值
  - 调用成功，返回0
  - 调用失败，返回-1

msgsnd()

- 例7：msgsnd.c

```c
struct message
{
        long msg_type;
        char msg_text[256];
};
int main()
{
        key_t key;
        int qid;
        struct message msg;
        if ( (key=ftok("/etc/profile",1)) == -1 ){
                perror("ftok");
        }
        if((qid=msgget(key, IPC_CREAT|0666)) == -1){
                perror("msgget");
        }
        msg.msg_type=2;
        strcpy(msg.msg_text,"this is the input");
        if (msgsnd(qid, &msg, strlen(msg.msg_text), 0) < 0 ){
                perror("msgsnd");
        }
        return 0;
}
```

msgrcv()

- 函数原型

```
#include <sys/types.h>
#include <sys/msg.h>
int msgrcv(int msqid, struct msgbuf *msgp, size_t size, long type, int flag);
```

- 功能
  - 从消息队列中读取一个消息并将其移出消息队列

msgrcv()

- 函数原型

```
#include <sys/types.h>
#include <sys/msg.h>
int msgrcv(int msqid, struct msgbuf *msgp, size_t size, long type, int flag);
```

- 参数
  - msqid：消息队列的标识符
  - msgp：消息结构指针
  - size：要读取消息的长度
  - type：要读取消息的类型
  - flag：接收消息可选标志

msgrcv()

- 函数原型

```
#include <sys/types.h>
#include <sys/msg.h>
int msgrcv(int msqid, struct msgbuf *msgp, size_t size, long type, int flag);
```

- 参数flag的说明
  - 0：当消息队列为空时，进程阻塞
  - IPC_NOWAIT：表明当消息队列空时，进程不阻塞，立即返回-1
  - MSG_NOERROR：允许消息长度大于接收缓冲区长度，截断消息返回；否则，不接受该消息，出错返回

msgrcv()

- 函数原型

```
#include <sys/types.h>
#include <sys/msg.h>
int msgrcv(int msqid, struct msgbuf *msgp, size_t size, long type, int flag);
```

- 返回值
  - 调用成功，返回实际读取到的消息的字节数
  - 调用失败，返回-1

msgrcv()

- 例8：msgrcv.c

```c
struct message
{
        long msg_type;
        char msg_text[256];
};
int main()
{
        int qid, n;
        key_t key;
        struct message msg;
        if ((key = ftok("/etc/profile",1)) == -1){
                perror("ftok");
        }
        if ((qid=msgget(key, 0)) == -1) {
                perror("msgget");
        }
        memset(msg.msg_text, 0, 256);
        if ((n=msgrcv(qid, &msg, sizeof(msg.msg_text), 2, 0)) < 0 ){
                perror("msgrcv");
        }
        else
                printf("msgrcv return length=[%d] text=[%s]\n", n, msg.msg_text);
        return 0;
}
```

msgctl()

- 函数原型

```
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

- 功能
  - 获取和修改消息队列属性信息
  - 查询消息队列描述符
  - 修改消息队列许可权
  - 删除该队列等

msgctl()

- 函数原型

```
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

- 参数
  - msqid：消息队列的标识符
  - buf：指向msqid_ds结构的指针
  - cmd：控制命令

msgctl()

- 函数原型

```
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

- 参数cmd的说明
  - IPC_RMID：删除消息队列
  - IPC_STAT：获取消息队列的msqid_ds结构，保存于buf所指向的缓冲区
  - IPC_SET：设置消息队列的msqid_ds结构，按照buf指向的结构的值

msgctl()

- 函数原型

```
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

- 返回值
  - 调用成功，返回0
  - 调用失败，返回-1

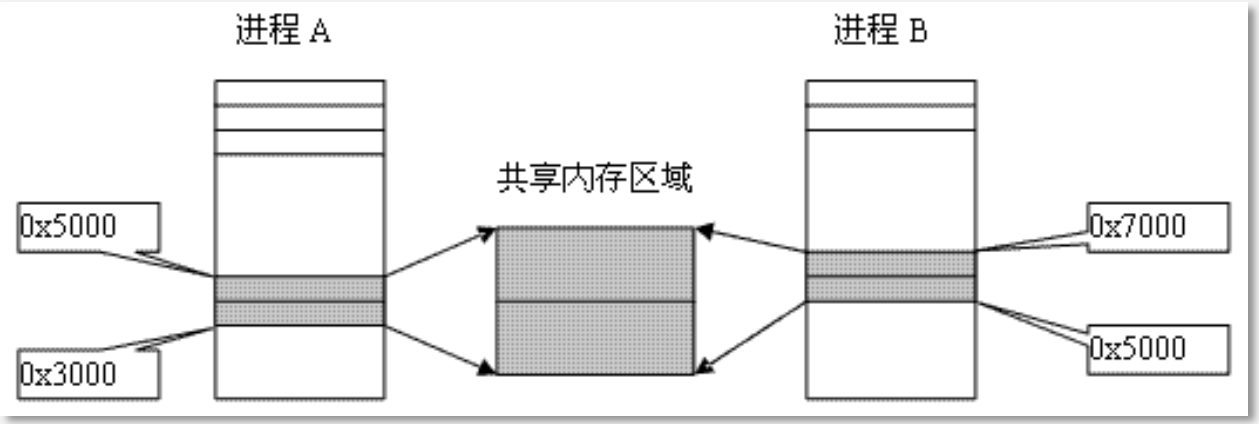
msgctl()

- 例9：msgctltest.c

```c
int main()
{
        key_t key;
        int qid;
        struct msqid_ds qds;
        if ((key=ftok("/etc/profile",1)) == -1) {
                perror("ftok");
        }
        if ((qid= msgget(key,0)) == -1) {
                perror("msgget");
        }
        memset(&qds, 0, sizeof(struct msqid_ds));
        if (msgctl(qid, IPC_STAT, &qds) < 0 ) {
                perror("msgctl IPC_STAT");
        }
        printf("msg_perm.mode=%d\n", qds.msg_perm.mode);
        if (msgctl(qid, IPC_RMID, NULL) < 0 ) {
                perror("msgctl IPC_RMID");
        }
        return 0;
}
```

共享内存

- 基本思想
  - 多个进程共享的一块内存区域
  - 需要访问的进程将这段内存映射到自己的地址空间
  - 进程可以直接读写这段内存，从而大大提高效率



共享内存

- 基本操作
  - 创建一个共享内存 - shmget()
  - 将共享内存映射至进程的地址空间 - shmat()
  - 通过返回的共享内存的读写指针对共享内存进行读写
  - 解除共享内存的映射 - shmdt()
  - 删除共享内存 - shmctl()

shmget()

- 函数原型

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

- 功能
  - 创建/获得一个共享内存

shmget()

- 函数原型

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

- 参数
  - key：键值
  - size：共享内存的大小
  - flags：调用函数的操作类型及设置访问权限

    IPC_CREAT：如果key值已存在，则打开共享内存，否则新建共享内存

shmget()

- 函数原型

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

- 返回值
  - 调用成功，返回共享内存区的标识符
  - 调用失败，则返回-1

shmget()

- 例10：shmgettest.c

```c
int main()
{
        key_t key;
        int shmid;
        if ( (key=ftok("/etc/profile",1)) < 0 )
        {
                perror("ftok");
        }
        if ( (shmid=shmget(key, 256, IPC_CREAT|0666)) == -1 )
        {
                perror("shmget");
        }
        printf("shmid=%d\n",shmid);
        return 0;
}
```

shmat()

- 函数原型

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
void *shmat(int shmid, char *shmaddr, int flag);
```

- 功能
  - 将共享内存映射到本地进程的地址空间

shmat()

- 函数原型

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
void *shmat(int shmid, char *shmaddr, int flag);
```

- 参数
  - shmid：共享内存的标识符
  - shmaddr：内核将该共享内存区域映像到调用进程的地址空间中的位置。一般情况下指定为NULL即可
  - flags：设置标识

    SHM_RDONLY：该内存区域被设置为只读

    0：表示可读可写

shmat()

- 函数原型

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
void *shmat(int shmid, char *shmaddr, int flag);
```

- 返回值
  - 调用成功，返回共享内存区的实际地址
  - 调用失败，则返回-1

shmdt()

- 函数原型

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmdt(char * shmaddr) ;
```

- 功能
  - 解除共享内存与进程地址空间的映射

shmdt()

- 函数原型

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmdt(char * shmaddr) ;
```

- 参数
  - shmaddr：要分离的共享内存区域的指针，即调用shmat函数的返回值
- 返回值
  - 调用成功，返回0
  - 调用失败，返回-1

共享内存实例

- 例11：shmwritetest.c

```c
int main()
{
        int shmid;
        key_t key;
        char *shmaddr;
        if ((key=ftok("/etc/profile", 2)) == -1 )
        {
                perror("ftok");
        }
        if ((shmid=shmget(key, 4096, IPC_CREAT|0666)) == -1)
        {
                perror("shmget");
        }
        shmaddr=shmat(shmid, NULL, 0);
        sprintf(shmaddr,"hello this is a test.");
        if (shmdt(shmaddr) == -1)
        {
                perror("shmdt");
        }
        return 0;
}
```

共享内存实例

- 例12：shmreadtest.c

```c
int main()
{
        int shmid;
        key_t key;
        char *shmaddr;
        if ((key=ftok("/etc/profile",2)) == -1)
        {
                perror("ftok");
        }
        if ((shmid=shmget(key, 4096, IPC_CREAT)) == -1)
        {
                perror("shmget");
        }
        shmaddr=shmat(shmid, NULL, 0);
        printf("get from share memory:%s\n", shmaddr);
        if (shmdt(shmaddr) == -1)
        {
                perror("shmdt");
        }
        return 0;
}
```

shmctl()

- 函数原型

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- 功能
  - 控制共享主存区属性，读取或修改其状态信息

shmctl()

- 函数原型

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- 参数
  - shmid：共享内存的标识符
  - buf：指向shmid_ds结构体的指针
  - cmd：控制命令

shmctl()

- 函数原型

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- 参数cmd的说明
  - IPC_RMID：删除共享存储区
  - IPC_SET：设置该共享存储区的shmid_ds结构
  - IPC_STAT：获取该共享内存的shmid_ds结构
- 例子：删除共享存储区

```
shmctl(shmid, IPC_RMID, NULL);
```

shmctl()

- 函数原型

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- 返回值
  - 调用成功，返回0
  - 调用失败，返回-1

shmctl()

- 例13：shmctltest.c

```c
int main()
{
        int shmid;
        key_t key;
        char *shmaddr;
        if ((key=ftok("/etc/profile",2)) == -1) {
                perror("ftok");
        }
        if ((shmid=shmget(key, 4096, IPC_CREAT)) == -1) {
                perror("shmget");
        }
        shmaddr=shmat(shmid, NULL, 0);
        printf("get from share memory:%s\n", shmaddr);
        if (shmdt(shmaddr) == -1) {
                perror("shmdt");
        }
        if (shmctl(shmid, IPC_RMID, NULL) < 0 ) {
                perror("shmctl");
        }
        return 0;
}
```