第5章
进程控制与进程间通信

进程控制

进程与程序

- **程序**
  - 指令和数据的集合
  - 存放在磁盘上的一个普通文件里
  - 文件的i节点中标为可执行，内容符合系统要求
- **进程**
  - 包括指令段、用户数据段和系统数据段的执行环境
- **进程和程序的关系**
  - 程序用于初始化进程的指令段和用户数据段，初始化后，进程和初始化它的程序之间无联系
  - 进程运行时磁盘上的程序文件不可修改/删除
  - 同时运行的多个进程可由同一程序初始化得到，进程之间没什么联系。内核通过安排它们共享指令段以节省内存，但这种安排对用户来说是透明的

进程的组成部分(1)

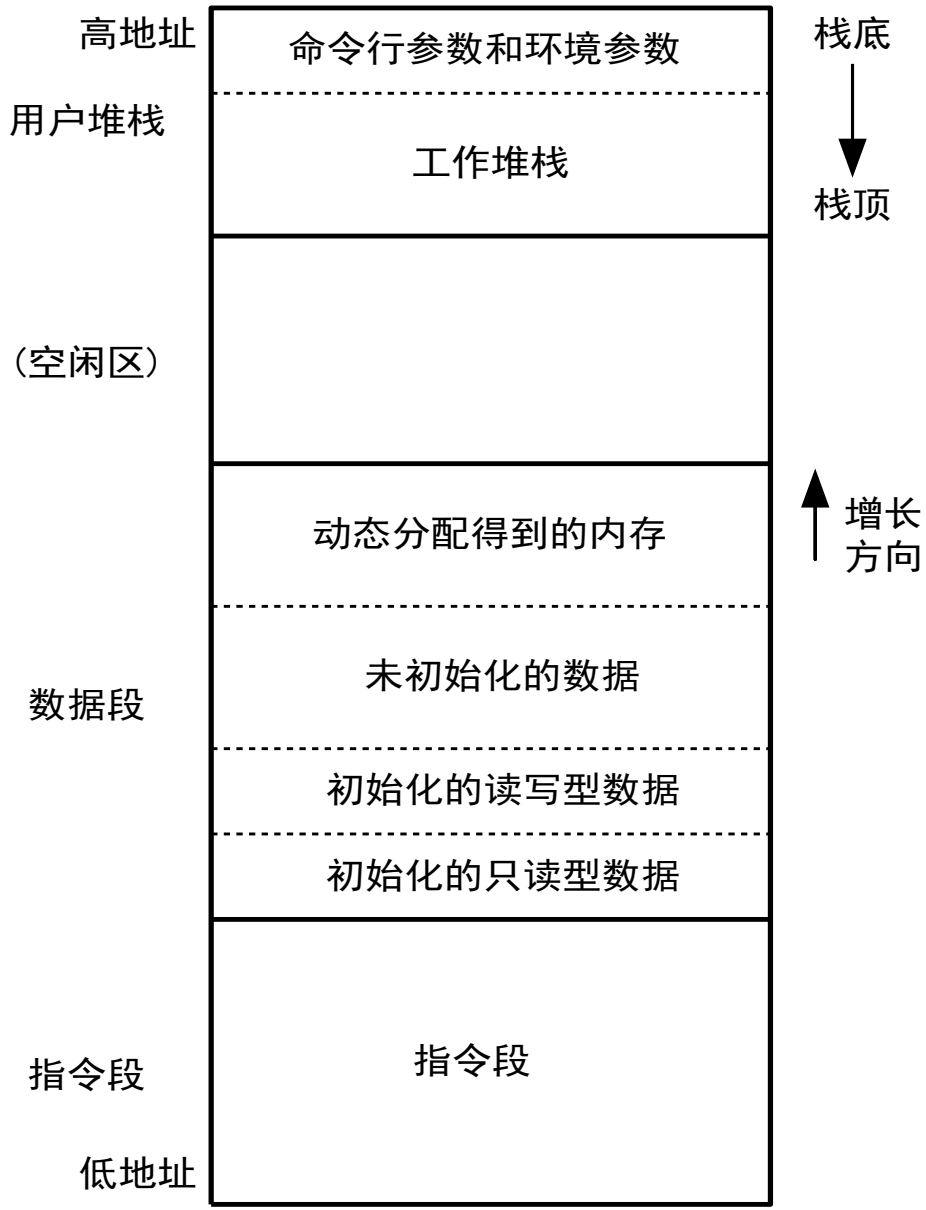
四部分：指令段，数据段，堆栈段和系统数据

- **指令段(Text)**
  - 程序的CPU指令代码,包括：主程序和子程序编译后的CPU指令代码，以及调用的库函数代码
  - 指令段的大小固定不变，只读
- **用户数据段**
  - 全局变量，静态(static)变量，字符串常数
  - 允许数据段增长和缩小，实现内存的动态分配
    - 系统调用sbrk()允许编程调整数据段的大小
    - 内存管理库函数，如：malloc()，free()

进程的组成部分(2)

- **用户堆栈段**
  - 程序执行所需要的堆栈空间，实现函数的调用
    - 用于保存子程序返回地址
    - 在函数和被调函数之间传递参数
    - 函数体内部定义的变量(静态变量除外)
  - main函数得到的命令行参数以及环境参数
    - 存放在堆栈的最底部
    - main函数运行之前，这些部分就已经被系统初始化
  - 堆栈段的动态增长与增长限制
- **系统数据段**
  - 上述三部分在进程私有的独立的逻辑地址空间内
  - 系统数据段是内核内的数据，每个进程对应一套
    - 包括页表和进程控制块PCB

进程逻辑地址空间的布局



进程的系统数据

在UNIX内核中，含有进程的属性，包括:

- 页表
- 进程状态，优先级信息
- 核心堆栈
- 当前目录(记录了当前目录的i-节点)，根目录
- 打开的文件描述符表
- umask值
- 进程PID，PPID
- 进程主的实际UID/GID，有效UID/GID
- 进程组组号

UNIX的user+proc结构

- **进程PCB被分为user结构和proc结构两部分**
  - user结构(约5000字节)，<sys/user.h>
    - 进程运行时才需要的数据在user结构
    - 核心态堆栈占用了较多空间
  - proc结构(约300字节)，<sys/proc.h>
    - 进程不运行时也需要的管理信息存于proc结构
  - 用户程序不能直接存取和修改进程的系统数据
  - 系统调用可用来访问或修改这些属性
    - chdir， umask，open， close
    - setpgrp， getpid， getppid

进程的基本状态

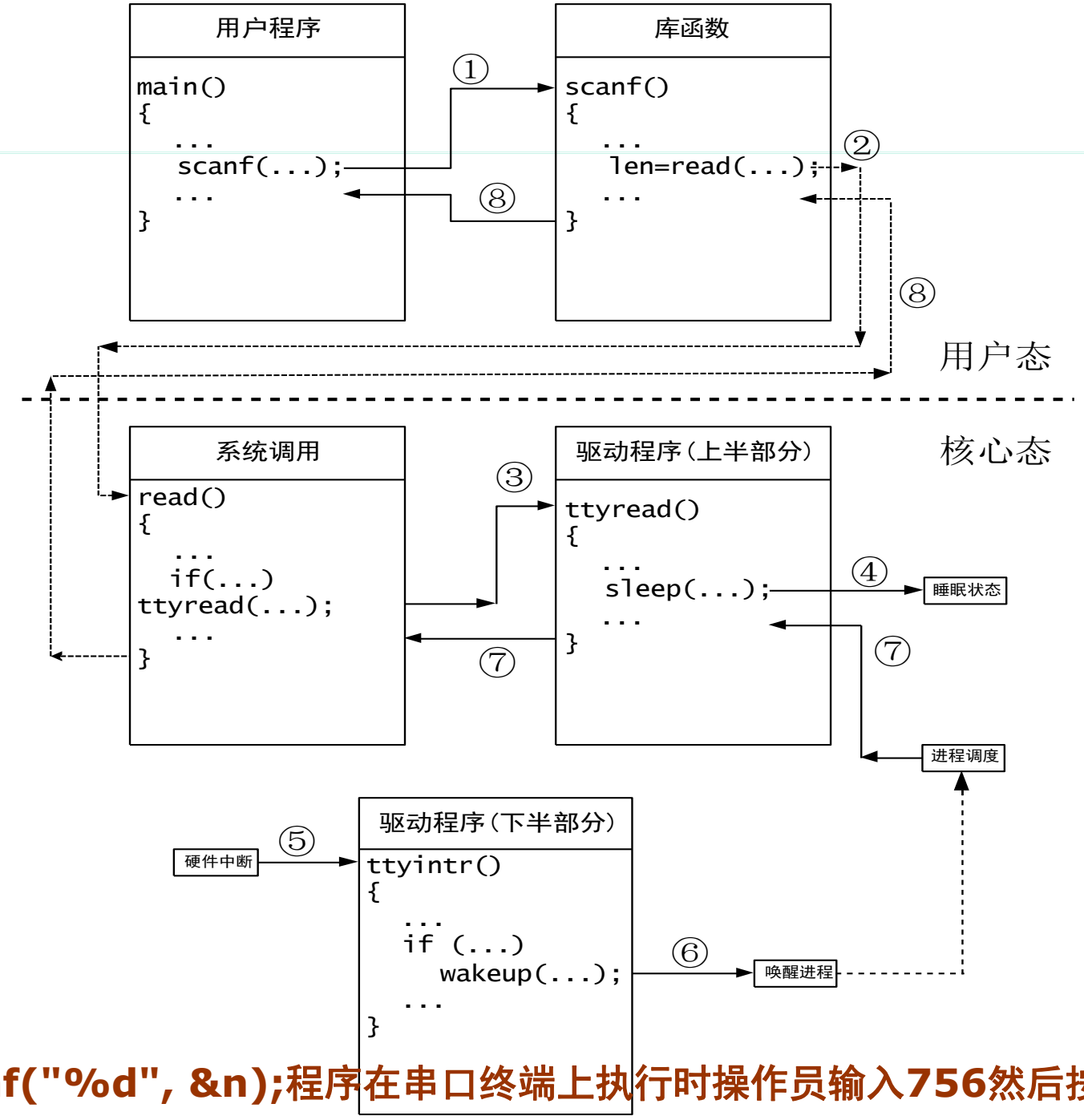
基本状态

- 进程创建之后，主要有运行状态和睡眠状态(也叫阻塞状态，等待状态，挂起状态，等等)
- 内核总是在分时处理运行状态的进程，而不顾那些处于睡眠状态的进程
- 睡眠状态的进程，在条件满足后转化为运行状态
- 进程在睡眠时，不占用CPU时间
  - 注意：在编程时，尽量不要让程序处于忙等待状态

进程的调度

调度优先级

- 内核将可运行进程按优先级调度，高优先级进程优先
- 进程的优先级总在不停地发生变化
- 处于睡眠状态的进程一旦被叫醒后，被赋以高优先级，以保证人机会话操作和其它外设的响应速度
- 用户程序用nice()系统调用有限地调整进程的优先级



scanf("%d", &n);程序在串口终端上执行时操作员输入756然后按下回车

time:进程执行的时间

- **进程执行时间包括**
  - 睡眠时间，CPU时间(用户时间和系统时间)
  - 外部命令/usr/bin/time
  - B-shell和C-shell都有个内部命令time

/usr/bin/time find /usr -name '*.c' -print

Real 6.06
User 0.36
System 2.13

- C-shell: time find /usr -name '*.h' -print

0.4u 6.2s 0:10 61% 4+28k 0+0io 0pf+0w

- **与CPU时间有关的命令vmstat**

```
$ vmstat 10
   procs                  memory      swap        io     system      cpu
 r  b  w   swpd   free   buff  cache   si   so    bi  bo   in   cs us sy id
 0  0  0      0  55916   6128  38156    0    0   439 118  146  180  8 15 76
 0  0  0      0  55252   6160  38160    0    0     0  32  112   54 26  1 73
```

系统调用times()

系统调用times()

当前进程CPU时间，已结束子进程占用过的CPU时间

```
clock_t times(struct tms *buf);
struct tms {
      clock_t tms_utime;  /* user  CPU time */
      clock_t tms_stime;  /* system CPU time */
      clock_t tms_cutime; /* user CPU time, terminated children */
      clock_t tms_cstime; /* system CPU time, terminated children */
};
```

clock()

- 返回times()的四个CPU时间的总和。单位是1/CLOCKS_PER_SEC秒

getrusage()函数，times()函数的升级版本

- 返回CPU时间，还返回表示资源使用状况的另外14个值，包括内存使用情况，I/O次数，进程切换次数

与时间有关的函数

- **标准函数库中time()：获得当前时间坐标**
  - 坐标0为1970年1月1日零点，单位：秒
  - t=time(0);和time(&t);都会使t值为当前时间坐标
- **函数gettimeofday()**
  - 获得当前时间坐标，坐标的0是1970年1月1日零点
  - 可以精确到微秒μs(10^{-6}秒)
- **localtime()**
  - 将坐标值转换为本地时区的年月日时分秒
- **mktime**
  - 将年月日时分秒转换为坐标值
- **ctime()和asctime()**
  - 坐标值和年月日时分秒转换为可读字符串

忙等待

- **多任务系统中“忙等待”的程序是不可取的**

```
 1 #include <fcntl.h>
 2 main(int argc, char **argv)
 3 {
 4     unsigned char buf[1024];
 5     int len, fd;
 6     if ( (fd = open(argv[1], O_RDONLY)) < 0) {
 7         perror("Open file");
 8         exit(1);
 9     }
10     lseek(fd, 0, SEEK_END);
11     for(;;) {
12         while ( (len = read(fd, buf, sizeof buf)) > 0)
13             write(1, buf, len);
14     }
15 }
```

- 程序14行前增加sleep(1)：1秒定时轮询
- select()调用可将睡眠时间设为10毫秒级精度

fork：创建新进程

- **功能**
  - fork系统调用是创建新进程的唯一方式
  - 原先的进程称做“父进程”，新创建进程被称作“子进程”
  - 完全复制：新进程的指令，用户数据段，堆栈段
  - 部分复制：系统数据段
- **fork返回值：父子进程都收到返回值，但不相同**
  - 返回值很关键，它用于区分父进程(返回值>0，是子进程的PID)和子进程(返回值=0)，失败时返回-1
- **内核实现**
  - 创建新的proc结构，复制父进程环境(包括user结构和内存资源)给子进程
  - 父子进程可以共享程序和数据(例如：copy-on-write技术)，但是系统核心的这些安排，对程序员透明

fork举例(1)

```
int a;
int main(int argc, char **argv)
{
    int b;
    printf("[1] %s: BEGIN\n", argv[0]);
    a = 10;
    b = 20;
    printf("[2] a+b=%d\n", a + b);
    fork();
    a += 100;
    b += 100;
    printf("[3] a+b=%d\n", a + b);
    printf("[4] %s: END\n", argv[0]);
}
```

执行结果

```
[1] ./fork1: BEGIN
[2] a+b=30
[3] a+b=230
[4] ./fork1: END
[3] a+b=230
[4] ./fork1: END
```

fork举例(2)

```
main()
{
    int a, ret;
    printf("Hello\n");
    a = 3;
    ret = fork();
    a += 3;
    if (ret > 0) {
        printf("PID=%d child=%d, a=%d\n",
            getpid(), ret, a);
    } else if (ret == 0) {
        printf("PID=%d ppid=%d, a=%d\n",
            getpid(), getppid(), a);
    } else {
        perror("Create new process");
    }
    printf("Bye\n");
}
```

执行结果

```
Hello
PID=18378 ppid=18377, a=6
Bye
PID=18377 child=18378, a=6
Bye
```

命令ps

- **功能**
  - 查阅进程状态(process status)(实际上就是将内核中proc[]和user[]数组的内容有选择地打印出来)
- **选项**
  - 用于控制列表的行数(进程范围)和列数(每进程列出的属性内容)
  - 无选项：只列出在当前终端上启动的进程
    - 列出的项目有：PID，TTY，TIME，COMMAND
  - **e**选项：列出系统中所有的进程(进程范围)
  - **f**选项：以full格式列出每一个进程(控制列的数目)
  - **l**选项：以long格式列出每一个进程(控制列的数目)

命令ps举例

例：ps -ef命令的输出

```
 UID   PID  PPID  C    STIME  TTY TIME COMMAND
root     0     0  0 16:11:14  ?   0:00 sched
root     1     0  0 16:11:14  ?   0:01 /etc/init
root     2     0  0 16:11:14  ?   0:00 vhand
root     3     0  0 16:11:14  ?   0:00 bdflush
root  6547   335  0 16:15:05  01  0:00 server 4 1
root   175     1  0 16:11:14  ?   0:00 inetd
root   173     1  0 16:11:14  ?   0:00 syslogd
root   296     1  0 16:11:21  ?   0:00 /tcb/files/no_luid/sdd
root  6551   291 80 16:17:16  08  0:37 busy_check
root   335     1  0 16:11:31  01  0:00 server_ctl
root   337     1  0 16:11:33  01  0:05 server_ap
root   353     1  0 16:12:01  12  0:00 client_ctl 100.1.1.3
root   356   295  0 16:12:07  12  0:04 client_ap 1403
```

命令ps列出的进程属性

- UID：用户ID(注册名)
- PID：进程ID
- **C**：CPU占用指数：最近一段时间(秒级别)进程占用CPU情况。不同系统算法不同，例如：正占CPU进程10ms加1，所有进程1秒衰减一半
- PPID：父进程的PID
- STIME：启动时间
- **SZ**：进程逻辑内存大小(Size)
- TTY：终端的名字
- COMMAND：命令名
- WCHAN：进程睡眠通道(Wait Channel)，进程阻塞在何处
- **TIME**：累计执行时间(占用CPU的时间)
- PRI：优先级
- **S**：状态，S(Sleep)，R(Run)， Z(Zombie)

命令行参数和环境参数

- **位于进程堆栈底部的初始化数据**
- **访问命令行参数的方法（argc,argv)**
  - 查阅进程状态(process status)(实际上就是将内核中proc[]和user[]数组的内容有选择地打印出来)
- **访问环境参数的三种方法**
  - 通过C库定义的外部变量environ
  - main函数的第三个参数
  - getenv库函数调用

访问环境参数的三种方法

访问环境参数的三种方法

```
main()
{
    extern char  **environ;
    char **p;
    p = environ;
    while (*p) printf("[%s]\n", *p++);
}
main(int argc, char **argv, char **env)
{
    char **p;
    p = env;
    while (*p) printf("[%s]\n", *p++);
}
main()
{
    char *p;
    p=getenv("HOME");
    if (p) printf ("[%s]\n");
}
```

exec系统调用

- **功能**
  - 用一个指定的程序文件，重新初始化一个进程
  - 可指定新的命令行参数和环境参数(初始化堆栈底部)
  - exec不创建新进程，只是将当前进程重新初始化了指令段和用户数据段，堆栈段以及CPU的PC指针
- **6种格式exec系统调用**
  - exec前缀，后跟一至两个字母

    l—list，**v**—vector

    e—env，**p**—path
  - **l**与**v**：指定命令行参数的两种方式，**l**以表的形式，**v**要事先组织成一个指针数组
  - **e**：需要指定envp来初始化进程。
  - **p**：使用环境变量PATH查找可执行文件
  - 六种格式的区别：不同的参数方式初始化堆栈底部

exec系统调用格式

int execl(char *file，char *arg0，char *arg1，...，0);

int execv(char *file, char **argv);

int execle(char *file, char *arg0, char *arg1, ..., 0, char **envp);

int execve(char *file, char **argv, char** envp);

int execlp(char *file, char *arg0, char *arg1, ..., 0);

int execvp(char *file, char **argv);

“僵尸”进程

- **“僵尸”进程(zombie或defunct)**
  - 进程生命期结束时的特殊状态
    - 系统已经释放了进程占用的包括内存在内的系统资源，但仍在内核中保留进程的部分数据结构，记录进程的终止状态，等待父进程来“收尸”
    - 父进程的“收尸”动作完成之后，“僵尸”进程不再存在
  - 僵尸进程占用资源很少，仅占用内核进程表资源
    - 过多的僵尸进程会导致系统有限数目的进程表被用光
  - 孤儿进程

wait系统调用

- **功能**
  - 等待进程的子进程终止
  - 如果已经有子进程终止，则立即返回
- **函数原型**

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

函数返回值为已终止的子进程PID

- 例

```
int status, pid;
pid = wait(&status);
```

直接调用就可以

status中含有子进程终止的原因

TERMSIG(status)为被杀信号

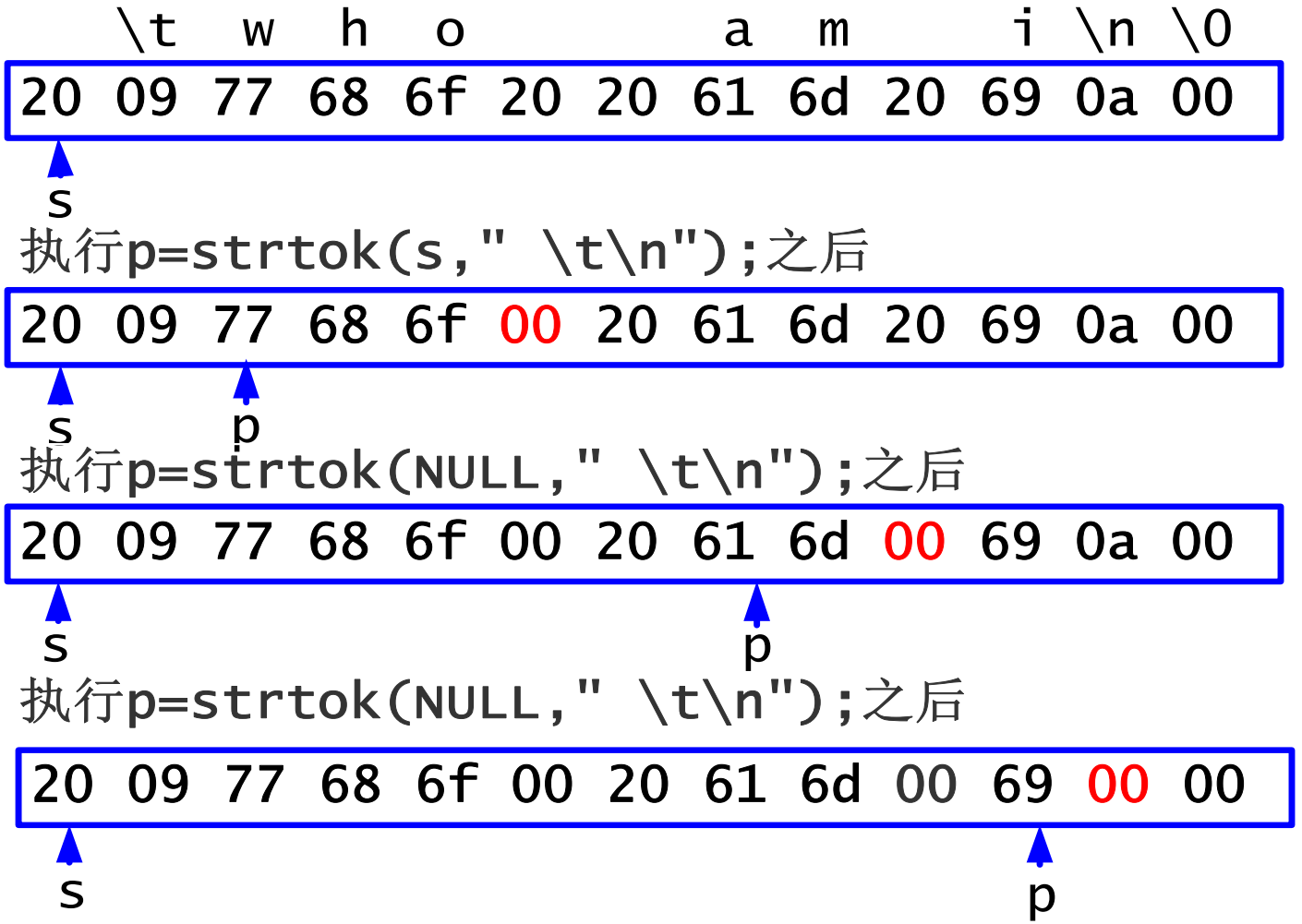
EXITSTATUS(status)为退出码

- **waitpid()和wait3() : wait系统调用的升级版本**

字符串库函数strtok

```
char *strtok(char *str,  char *tokens)
```

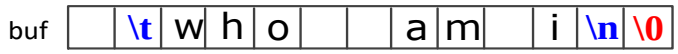
功能：返回第一个单词的首字节指针



最简单的shell: xsh0

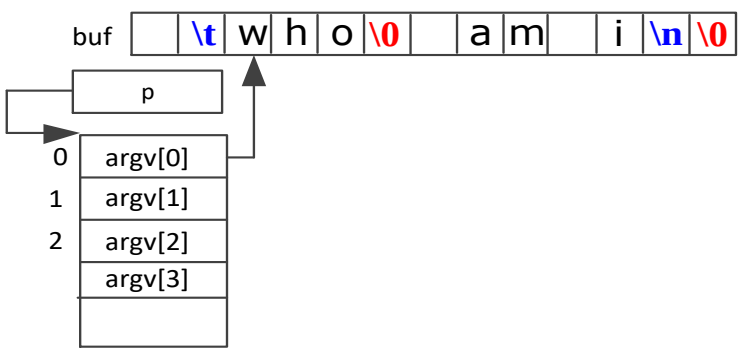
```
void main(void)
{
    char buf[256], *argv[256], **p;
    int sv;
    for (;;) {
        printf("=> ");
        if (fgets(buf, sizeof(buf), stdin) == NULL)
            exit(0);
        for (p = &argv[0], *p = strtok(buf, " \t\n"); *p;
               *++p = strtok(NULL, " \t\n"));
        if (argv[0] == NULL)
            continue;
        if (strcmp(argv[0], "exit") == 0)
            exit(0);
        if (fork() == 0) {
            execvp(argv[0], argv);
            fprintf(stderr, "** Bad command\n");
            exit(1);
        }
        wait(&sv);
    }
}
```

执行fgets后的状态



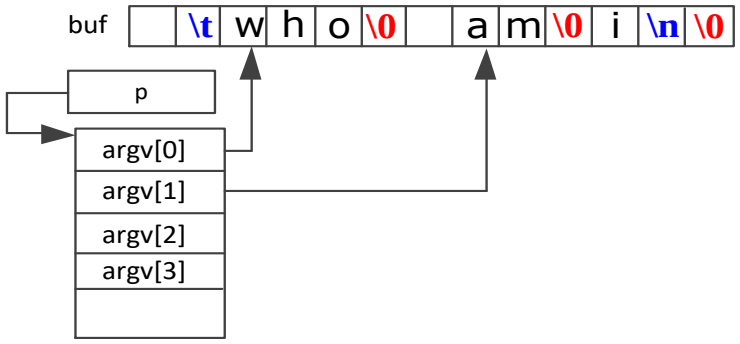
执行for循环初始化操作后的状态

```
p = &argv[0], *p = strtok(buf, " \t\n");
```

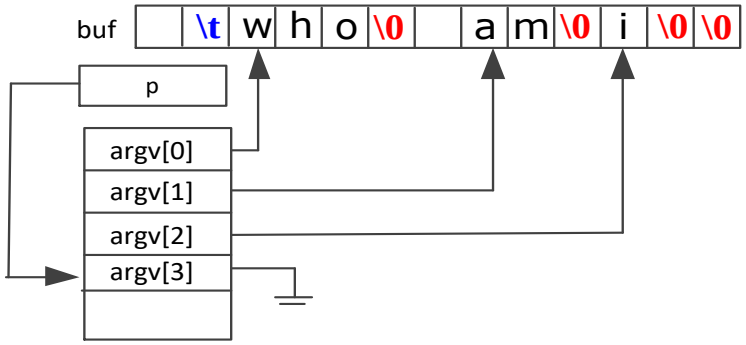


执行for循环第1轮之后的状态

```
*++p = strtok(NULL, " \t\n");
```



执行for循环第3轮之后的状态



执行xsh0

```
=>
=> who am i
root       ttyp0        Nov 29 09:56
=> ps -f
 UID   PID  PPID C    STIME   TTY      TIME CMD
root  1012  1011 0 09:56:00 ttyp0 00:00:00 login -c -p
root  1020  1012 0 09:56:14 ttyp0 00:00:00 -csh
root  1060  1020 2 10:36:37 ttyp0 00:00:00 xsh0
root  1062  1060 4 10:36:46 ttyp0 00:00:00 ps -f
=> ls -l *.c
ls: *.c not found: No such file or directory (error 2)
=> mmx
** Bad command
=> ls -l xsh0
-rwxr--r--   1 root   other  43417 Dec  1  1998 xsh0
=> exit
```

system：运行一个命令

int system(char *_string_);

- 库函数system执行用字符串传递的shell命令，可使用管道符和重定向
- 库函数system()是利用系统调用fork，exec，wait实现的

system应用举例

```
#include <stdio.h>
main()
{
    char fname[256], cmd[256], buf[256];
    FILE *f;
    tmpnam(fname);
    sprintf(cmd, "netstat -rn > %s", fname);
    printf("Execute \"%s\"\n", cmd);
    system(cmd);
    f = fopen(fname, "r");
    while (fgets(buf, sizeof buf, f))
        printf("%s", buf);
    fclose(f);
    printf("Remove file \"%s\"\n", fname);
    unlink(fname);
}
```

进程与文件描述符

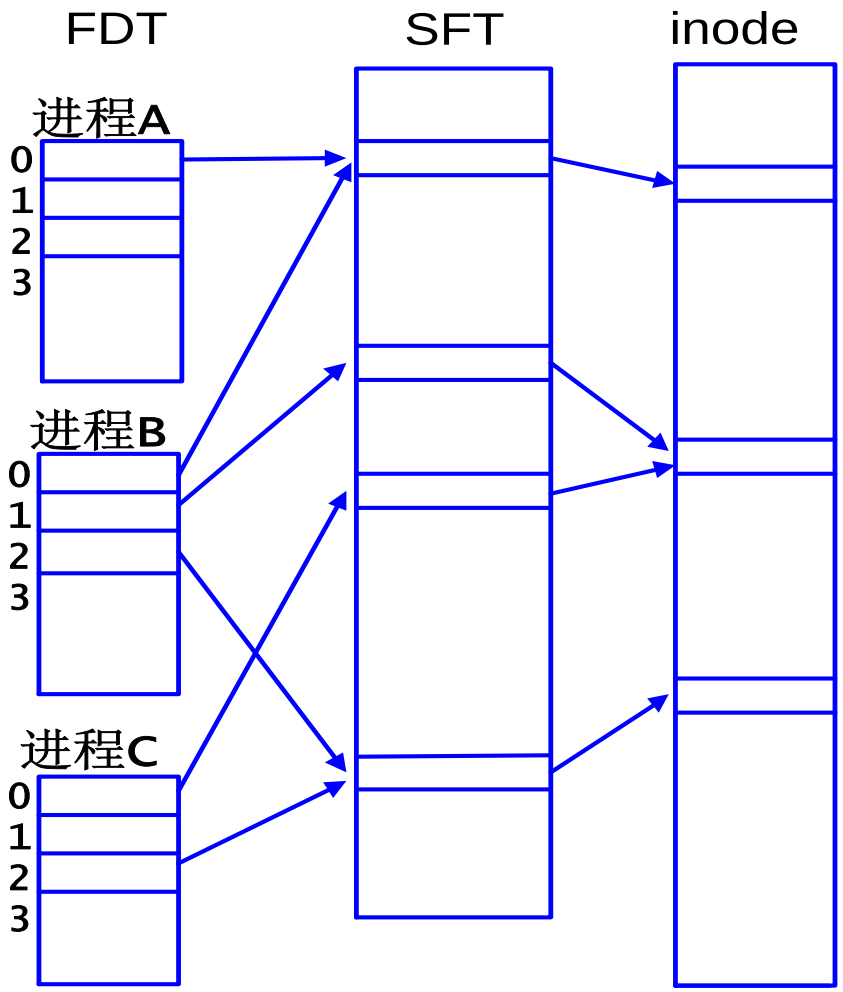
活动文件目录AFD

- **磁盘文件目录(分两级)**
  - 文件名，i节点
- **活动文件目录(分三级)**
  - **文件描述符表FDT**：每进程一张，PCB的user结构中
    - user结构中整型数组u_ofile记录进程打开的文件
    - 文件描述符fd是u_ofile数组的下标
  - **系统文件表SFT**：整个核心一张，file结构

```
struct file {
    char f_flag;    /* 读、写操作要求 */
    char f_count;   /* 引用计数 */
    long f_offset; /* 文件读写位置指针 */
    int  f_inode;   /* 内核中inode数组的下标 */
};
```

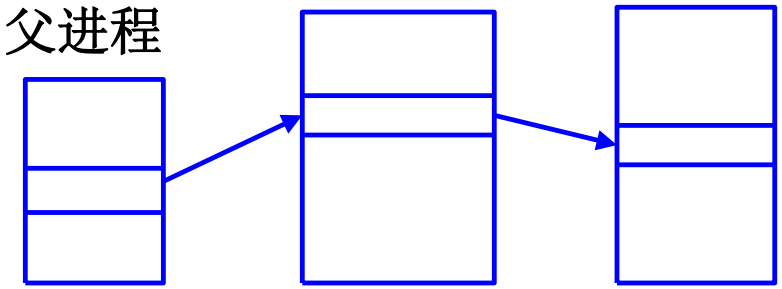
  - **活动i节点表**：整个核心一张，inode结构
    - 内存中inode表是外存中inode的缓冲
    - 内存inode表里也有个专用的引用计数

活动文件目录AFD(图)

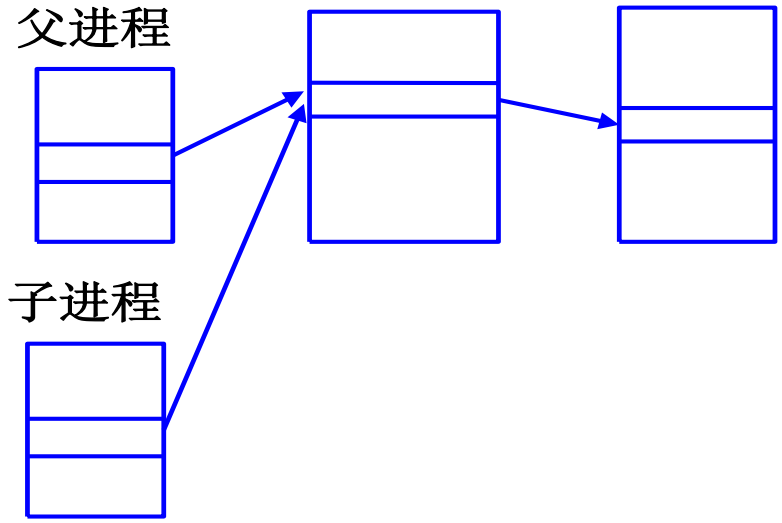


文件描述符的继承与关闭

- fork创建的子进程继承父进程的文件描述符表
- 父进程在fork前打开的文件，父子进程有相同的文件偏移



fork前



fork后

例:文件描述符的继承与关闭(1)

```
void main()
{
    int fd;
    fd = open("xxf1f2.txt", O_CREAT | O_WRONLY, 0666);
    if (fork() > 0) {
        char *str = "Message from process F1\n";
        int i;
        for (i = 0; i < 200; i++) {
            write(fd, str, strlen(str));
            sleep(1);
        }
        close(fd);
    } else {
        char fdstr[16];
        sprintf(fdstr, "%d", fd);
        execlp("./f2", "f2", fdstr, 0);
        printf("failed to start 'f2': %m\n");
    }
}
```

例:文件描述符的继承与关闭(2)

- 文件f2. c

```
int main(int argc, char **argv)
{
    int fd, i;
    static char *str = "Message from process F2\n";
    fd = strtol(argv[1], 0, 0);
    for (i = 0; i < 200; i++) {
        if (write(fd, str, strlen(str)) < 0)
            printf("Write error: %m\n");
        sleep(1);
    }
    close(fd);
}
```

close-on-exec标志

文件close-on-exec标志

- 文件描述符fd设置了close-on-exec标志，执行exec系统调用时，系统会自动关闭这些文件

函数

#include <fcntl.h>
int fcntl (*fd*, *cmd*, *arg*);
cmd: F_GETFD 获取文件*fd*的控制字
控制字的比特0(FD_CLOEXEC): close-on-exec标志位
flag = fcntl(fd, F_GETFD, 0);
cmd: F_SETFD 设置文件*fd*的控制字
fcntl(fd, F_SETFD, flag);

例

```
flags = fcntl(fd, F_GETFD, 0);
flags |= FD_CLOEXEC;
fcntl(fd, F_SETFD, flags);
```

文件描述符的复制

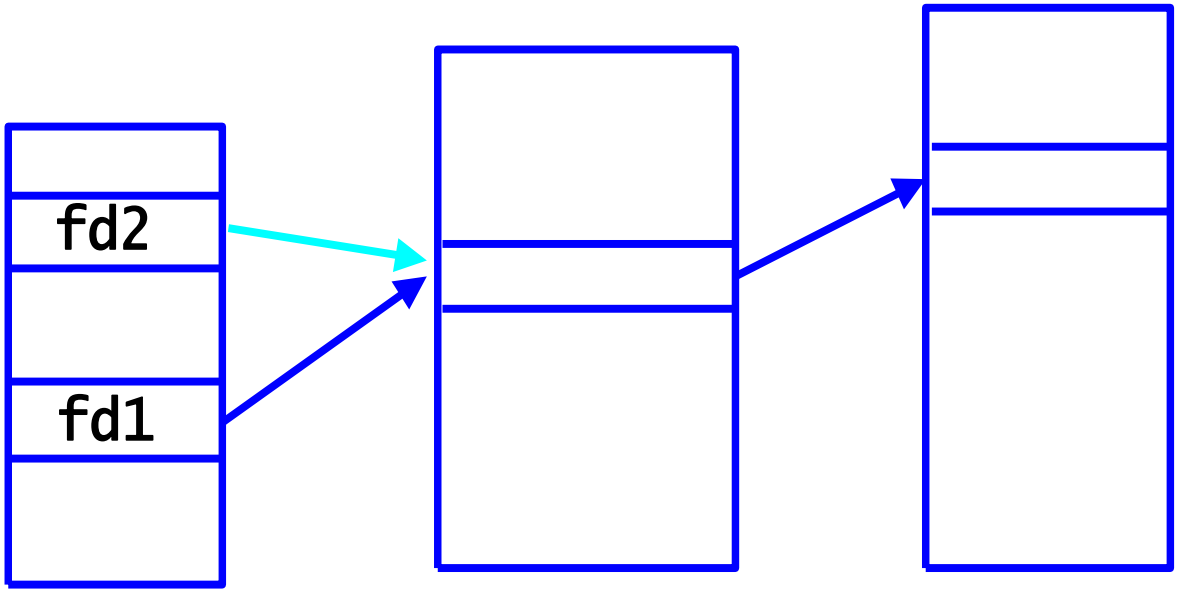
- **系统调用**

  int dup2(int *fd1*, int *fd2*);

- **功能**

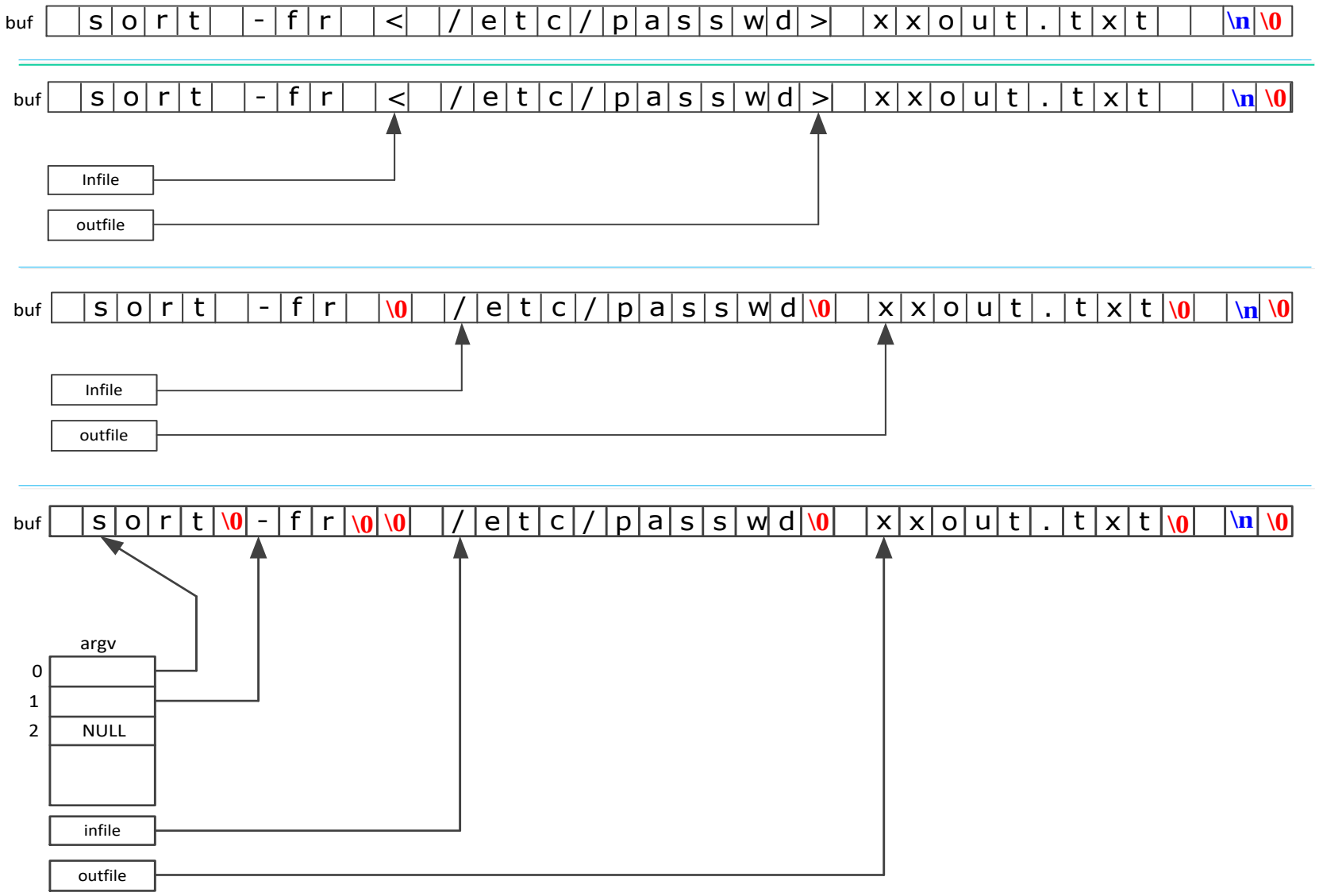
  复制文件描述符*fd1*到*fd2*

  - *fd2*可以是空闲的文件描述符
  - 如果*fd2*是已打开文件，则关闭已打开文件



xsh1:输入输出重定向(1)

```
void main(void)
{
     char buf[256], *argv[256], **p, *in, *out;
     int sv;
     for (;;) {
          printf("=> ");
          if (fgets(buf, sizeof(buf), stdin) == NULL) exit(0);
          in = strstr(buf, "<");
          out = strstr(buf, ">");
          if (in != NULL) {
               *in++ = '\0';
               in = strtok(in, " \t\n");
          }
          if (out != NULL) {
               *out++ = '\0';
               out = strtok(out, " \t\n");
          }
          for (p = &argv[0], *p = strtok(buf, " \t\n"); *p;
             *++p = strtok(NULL, " \t\n"));
          if (argv[0] == NULL)
               continue;
          if (strcmp(argv[0], "exit") == 0)
               exit(0);
```


buf
s o r t - f r < / e t c / p a s s w d > x x o u t . t x t \n \0
buf
s o r t - f r < / e t c / p a s s w d > x x o u t . t x t \n \0
Infile
outfile
buf
s o r t - f r \0 / e t c / p a s s w d \0 x x o u t . t x t \0 \n \0
Infile
outfile
buf
s o r t \0 - f r \0 \0 / e t c / p a s s w d \0 x x o u t . t x t \0 \n \0
argv
0
1
2
NULL
infile
outfile

xsh1:输入输出重定向(2)

```
            if (fork() == 0) {
                int fd0 = -1, fd1 = -1;
                if (in != NULL)
                    fd0 = open(in, O_RDONLY);
                if (fd0 != -1) {
                    dup2(fd0, 0);
                    close(fd0);
                }
                if (out != NULL)
                    fd1 = open(out, O_CREAT | O_WRONLY, 0666);
                if (fd1 != -1) {
                    dup2(fd1, 1);
                    close(fd1);
                }
                execvp(argv[0], argv);
                fprintf(stderr, "** Bad command\n");
                exit(1);
            }
            wait(&sv);
        }
}
```

管道操作(1)

- **创建管道**

  int pipe(int *pfd*[2]);

  int pipe(int **pfd*);

  int pipe(int *pfd*[]);

  - 创建一个管道，*pfd*[0]和*pfd*[1]分别为管道两端的文件描述字，*pfd*[0]用于读，*pfd*[1]用于写

- **管道写**

  ret = write(pfd[1], buf, n)

  - 若管道已满，则被阻塞，直到管道另一端read将已进入管道的数据取走为止
  - 管道容量：某一有限值，如8192字节，与操作系统的实现相关

管道操作(2)

- **管道读**

  ret = read(pfd[0], buf, n)

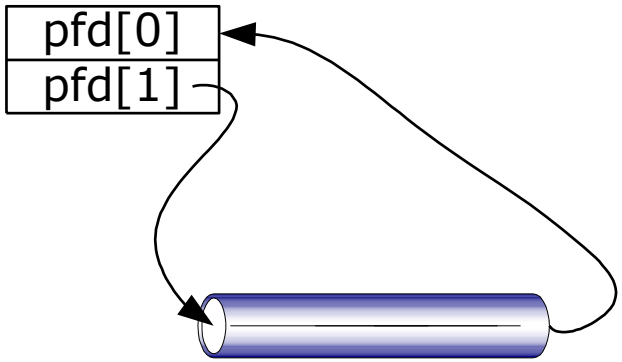
  - 若管道写端已关闭，则返回0
  - 若管道为空，且写端文件描述字未关闭，则被阻塞
  - 若管道不为空(设管道中实际有m个字节)
    - n≥m，则读m个；
    - 如果n＜m则读取n个
  - 实际读取的数目作为read的返回值。
  - 注意：管道是无记录边界的字节流通信

- **关闭管道close**

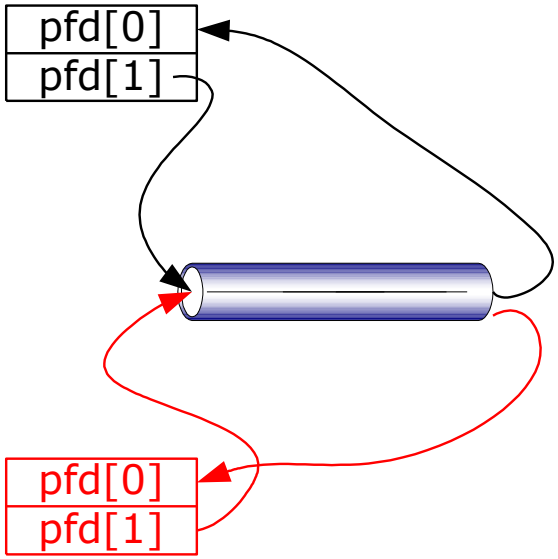
  - 关闭写端则读端read调用返回0。
  - 关闭读端则写端write导致进程收到SIGPIPE信号(默认处理是终止进程，该信号可以被捕捉)
    - 写端write调用返回-1，errno被设为EPIPE

进程间使用管道通信

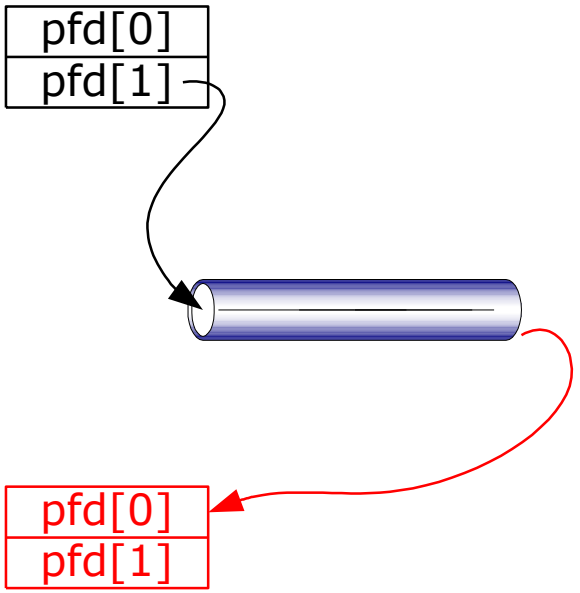
- 父进程



- fork后



- 父进程关闭读端，子进程关闭写端



管道通信：写端

```
main()
{
    int pfd[2], i;
    char fdstr[10], *message = "Send/Receive data using pipe ";
    pipe(pfd);
    if (fork() == 0) { /* child */
        close(pfd[1]);
        sprintf(fdstr, "%d", pfd[0]);
        execlp("./pread", "pread", fdstr, 0);
        printf("Execute pread file error: %m\n");
    }
    else { /* parent */
        close(pfd[0]);
        for (i = 0; i < 3; i++) {
            if (write(pfd[1], message, strlen(message)) == -1)
                printf("Write pipe error %m\n");
            // sleep(1);
        }
        close(pfd[1]);
    }
}
```

管道通信：读端

```
main(int argc, char **argv)
{
    int fd, nbyte;
    char buf[1024];
    fd = strtol(argv[1], 0, 0);
    do {
        nbyte = read(fd, buf, sizeof(buf));
        switch (nbyte) {
        case -1:
            printf("\nRead pipe error %m\n");
            break;
        case 0:
            printf("\nPipe closed\n");
            break;
        default:
            printf("\nReceive %d bytes\n", nbyte);
            write(1, buf, nbyte);
            break;
        }
    } while (nbyte > 0);
}
```

管道通信：应注意的问题

- **管道传输是一个无记录边界的字节流**
  - 写端一次write所发数据读端可能需多次read才能读取
  - 写端多次write所发数据读端可能一次read就全部读出
- **父子进程需要双向通信时，应采用两个管道**
  - 用一个管道，进程可能会收到自己刚写到管道去的数据
  - 增加其他同步方式太复杂
- **父子进程使用两个管道传递数据，有可能死锁**
  - 父进程因输出管道满而写，导致被阻塞
  - 子进程因要向父进程写回足够多的数据而导致写也被阻塞，这时死锁发生
  - 多进程通信问题必须仔细分析流量控制和死锁问题
- **管道的缺点：没有记录边界**

命名管道

- **pipe创建的管道的缺点**

  只限于同祖先进程间通信

- **命名管道**

  允许不相干的进程(没有共同的祖先)访问FIFO管道

- **命名管道的创建**
  - 用命令 mknod pipe0 p
  - 创建一个文件，名字为pipe0
  - 用ls -1列出时，文件类型为p

- **发送者**

```
fd = open(”pipe0”, O_WRONLY);
write(fd, buf, len);
```

- **接收者**

```
fd = open(”pipe0”, O_RDONLY);
read(fd, buf, sizeof(buf));
```

xsh2:管道(1)

```
void main(void)
{
     char buf[256], *argv1[256], **p, *cmd2, *argv2[256];
     int sv, fd[2];

     for (;;) {
          printf("=> ");
          if (fgets(buf, sizeof(buf), stdin) == NULL)
               exit(0);
          if ((cmd2 = strstr(buf, "|")) == NULL)
               exit(0);
          *cmd2++ = '\0';

          for (p = &argv1[0], *p = strtok(buf, " \t\n"); *p;
                    *++p = strtok(NULL, " \t\n"));
          for (p = &argv2[0], *p = strtok(cmd2, " \t\n"); *p;
                    *++p = strtok(NULL, " \t\n"));

          if (argv1[0] == NULL || argv2[0] == NULL)
               exit(0);
```

xsh2:管道(2)

```
            pipe(fd);
            if (fork() == 0) {
                dup2(fd[1], 1);
                close(fd[1]);
                close(fd[0]);
                execvp(argv1[0], argv1);
                fprintf(stderr, "** bad command 1: %m\n");
                exit(1);
            } else if (fork() == 0) {
                dup2(fd[0], 0);
                close(fd[0]);
                close(fd[1]);
                execvp(argv2[0], argv2);
                fprintf(stderr, "** bad command 2: %m\n");
                exit(1);
            }
            close(fd[0]);
            close(fd[1]);
            wait(&sv);
            wait(&sv);
        }
}
```

作业4

使用fork(), exec(), dup2(), pipe()， open()系统调用完成与下列shell命令等价的功能。

（提示：为简化编程，不需要用strtok断词，直接实现能达到下述shell命令相同功能的程序即可）

grep -v usr</etc/passwd|wc -l>result.txt

信号

命令kill

- **用法与功能**

  kill –*signal PID-list*

  kill命令用于向进程发送一个信号

- **举例**
  - kill 1275
    - 向进程1275的进程发送信号，默认信号为15(SIGTERM)，一般会导致进程死亡
  - kill -9 1326
    - 向进程1326发送信号9(SIGKILL)，导致进程死亡

进程组

进程组

- 进程在其proc结构中有p_pgrp域
- p_pgrp都相同的进程构成一个“进程组”
- 如果p_pgrp=p_pid则该进程是组长
- setsid()系统调用将proc中的p_pgrp改为进程自己的PID，从而脱离原进程组，成为新进程组的组长
- fork创建的进程继承父进程p_pgrp，与父进程同组

举例

- kill命令的PID为0时，向与本进程同组的所有进程发送信号

信号机制

- **功能**
  - 信号是送到进程的“软件中断”，通知进程出现了非正常事件
- **信号的产生**
  - 进程自己或者其他进程发出的
    - 使用kill()或者alarm()调用
  - 核心产生信号
    - 段违例信号SIGSEGV：当进程试图存取它的地址空间以外的存贮单元时，内核向进程发送段违例信号
    - 浮点溢出信号SIGFPE：零做除数时，内核向进程发送浮点溢出信号
    - 信号SIGPIPE：关闭管道读端则写端write导致进程收到信号SIGPIPE

信号类型

- **在<sys/signal.h>文件中定义宏**

SIGTERM　软件终止信号。用kill命令时产生
SIGHUP　挂断。当从注册shell中logout时，
　　同一进程组的所有进程都收到SIGHUP
SIGINT　中断。用户按Del键或Ctrl-C键时产生
SIGQUIT 退出。按Ctrl-\时产生，产生core文件
SIGALRM 闹钟信号。计时器时间到，与alarm()有关
SIGCLD　进程的一个子进程终止。
SIGKILL 无条件终止，该信号不能被捕获或忽略。
SIGUSR1，SIGUSR2　用户定义的信号
SIGFPE　浮点溢出
SIGILL　非法指令
SIGSEGV 段违例

进程对信号的处理

- **进程对到达的信号可以在下列处理中选取一种**
  - 信号被忽略
  - 设置为缺省处理方式
  - 信号被捕捉
    - 用户在自己的程序中事先定义好一个函数，当信号发生后就去执行这一函数
- **信号被设为缺省处理方式**

  signal(SIGINT,SIG_DFL);
- **信号被忽略**

  signal(SIGINT,SIG_IGN);
  - 在执行了这个调用后，进程就不再收到SIGINT信号
  - 注意：被忽略了的信号作为进程的一种属性会被它的子进程所继承

信号被忽略：举例

```
/* File name : abc.c */
#include <sys/signal.h>
main()
{
    signal(SIGTERM, SIG_IGN);
    if (fork()) {
        for(;;) sleep(100);
    } else
        execlp(”xyz”, ”xyz”, 0);
}
/* File name : xyz.c */
main()
{
    int i = 0;
    for (;;) {
        printf(”%d\n”, i++);
        sleep(1);
    }
}
```

- 单独运行xyz时，使用kill -15命令可杀死该进程
- 由abc进程来启动的xyz进程就不能用kill -15命令杀死

信号的捕捉

- 信号被捕捉并由一个用户函数来处理
- 信号到达时，这个函数将被调用来处理那个信号

```
#include <sys/signal.h>
sig_handle(int sig)
{
  printf(”HELLO! Signal %d catched.\n”,sig);
}

main()
{
    signal(SIGINT, sig_handle);
    signal(SIGQUIT, sig_handle);
    for(;;) sleep(500);
}
```

- 注意：在信号的用户函数被调用之前，自动被重置到它的缺省行为。

僵尸进程

- **僵尸子进程**
  - 子进程终止，僵尸进程(defunct或zombie)出现，父进程使用wait系统调用收尸后消除僵尸
- **僵尸进程对资源的占用**
  - 僵尸进程不占用内存资源等但占用内核proc表项，僵尸进程太多会导致proc表耗尽而无法再创建新进程
- **子进程中止后的通知机制**
  - 子进程中止后，系统会向父进程发送信号SIGCLD
- **不导致僵尸子进程出现的方法**
  - 忽略对SIGCLD信号的处理

    signal(SIGCLD,SIG_IGN);
  - 捕获SIGCLD信号，执行wait系统调用

捕获信号SIGCLD

```
int subprocess_die(int sig)
{
   int pid, status;
   pid = wait(&status);
          .
          .
   signal(SIGCLD, subprocess_die);
}

int main()
{
    signal(SIGCLD, subprocess_die);
        .
        .
}
```

发送信号

- **系统调用kill**

  int kill(int *pid*, int *sig*)

  返回值： 0--成功　　-1--失败

- **kill调用分几种情况**
  - 当pid>0时，向指定的进程发信号
  - 当pid=0时，向与本进程同组的所有进程发信号
  - 当pid<0时，向以-pid为组长的所有进程发信号
  - 当sig=0时，则信号根本就没有发送，但可据此判断一个已知PID的进程是否仍然运行
    - kill(pid, 0)；如果函数返回值为-1就可根据errno判断：errno=ESRCH说明不存在pid进程

系统调用与信号

进程睡眠

- 系统调用执行时会导致进程处于睡眠状态
  - 如：scanf()，sleep()，msgrcv()，操作外设的read()，write()，等等。

睡眠进程收到信号后处理

- 进程正在睡眠时收到信号，进程就会从睡眠中被惊醒，系统调用立即被半途终止，返回值-1， erron一般被置为EINTR
- 注意：有的系统调用特殊情况下睡眠很深，信号到达不能将它惊醒(有的进程用kill -9也杀不死)

pause与alarm系统调用

- **pause()**
  - 等待信号，进程收到信号前一直处于睡眠状态
- **设置进程报警时钟(闹钟)**

  int alarm(int *secs*)
  - 进程报警时钟存贮在它内核系统数据中，报警时钟到时，进程收到SIGALRM信号
    - 子进程继承父进程的报警时钟值。报警时钟在exec执行后保持这一设置
    - 进程收到SIGALRM后的默认处理是终止进程
  - alarm参数为*secs*
    - 当*secs*>0时，将时钟设置成***secs***指定的秒数
    - 当*secs*=0时，关闭报警时钟。

alarm系统调用举例

```
#include <sys/signal.h>
char cmd[100];
default_cmd(int sig)
{
    strcpy(cmd, "CMD_A");
}

main()
{
    signal(SIGALRM, default_cmd);
    alarm(5);
    printf("Input command : ");
    scanf("%s", cmd);
    alarm(0);
    printf("cmd=[%s]\n", cmd);
        .
        .
}
```

进程间通信（IPC机制）

信号灯

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

信号灯的创建

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

创建一个新的或获取一个已存在的信号灯组

nsems：该信号灯组中包含有多少个信号灯

函数返回一个整数，信号灯组的ID号；如果返回-1，表明调用失败

flags：创建或者获取

信号灯的删除

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

对信号灯的控制操作，如：删除，查询状态

snum: 信号灯在信号灯组中的编号

cmd: 控制命令

arg: 执行这一控制命令所需要的参数存放区

返回值为-1，标志操作失败；否则，表示执行成功，

对返回值的解释依赖于*cmd*

删除系统中信号灯组的调用

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

信号灯操作

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

对信号灯操作，可能会导致调用进程在此睡眠

ops：一个sembuf结构体数组，每个元素描述对某一信号灯操作

nops：指出上述的*ops*数组中有几个有效元素

返回值为-1，标志操作失败；否则，表示执行成功

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

当sem_op<0时，P操作

当sem_op>0时，V操作

当sem_op=0时，不修改信号灯的值，等待直到变为非负数

原子性：一次semop()调用可指定对多个信号灯的操作，UNIX内核要么把多个操作一下全部做完，要么什么都不做

共享内存

- **特点**
  - 多个进程共同使用同一段物理内存空间
  - 使用共享内存在多进程之间传送数据，速度快，但多个进程之间必须自行解决对共享内存访问的互斥和同步问题（例如：使用信号灯通过P/V操作）
- **应用举例**
  - 数据交换：进程使用共享内存向动态产生的数量不固定的多个查询进程发布数据
  - 运行监视：协议处理程序把有限状态机状态和统计信息放入共享内存中
    - 协议处理程序运行过程中可随时启动监视程序，从共享内存中读取数据以窥视当前的状态，了解通信状况
    - 监视程序的启动与终止不影响通信进程，而且这种机制不影响协议处理程序的效率

共享内存操作

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

创建一个新的或获取一个已存在的共享内存段

函数返回的整数是共享内存段的ID号；返回-1，表明调用失败

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

获取指向共享内存段的指针(进程逻辑地址)，返回-1：操作失败

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

对共享内存段的控制操作，如：删除，查询状态

cmd: 控制命令

arg: 执行这一控制命令所需要的参数存放区

“生产者-消费者”问题

- **问题**
  - 生产者和消费者使用N个缓冲区构成的环形队列来交换数据
- **程序设计：使用共享内存和信号量**
  - ctl create 创建共享内存段和信号灯
  - ctl remove 删除所创建的共享内存段和信号灯
  - producer *n* 启动多个生产者进程
  - consumer *n* 启动多个消费者进程
- **源程序文件有四个**
  - ctl.h　　公用的头文件
  - ctl.c　　控制程序，创建/删除所需要的IPC机制
  - producer.c　生产者程序
  - consumer.c　消费者程序

内存映射文件

内存映射文件I/O

- **传统的访问磁盘文件的模式**
  - 打开一个文件，然后通过read和write访问文件
- **“内存映射”(Memory Map)方式读写文件**
  - 现代的操作系统，包括UNIX和Windows都提供了一种“内存映射”(Memory Map)方式读写文件的方法。
  - 将文件中的一部分连续的区域，映射成一段进程逻辑地址空间中的内存
    - 进程获取这段映射内存的指针后，就把这个指针当作普通的数据指针一样引用。修改其中的数据，实际修改了文件，引用其中的数据值，就是读取了文件
    - 访问文件跟内存中的数据访问一样
  - MMAP方式访问文件利用虚拟内存功能
    - 系统不会为数据文件的内存映射区域分配相同大小的物理内存，而是由页面调度算法自动进行物理内存分配
    - 根据虚拟内存的页面调度算法，按需调入数据文件中的内容，必要时淘汰（可能需要写入)内存页面

内存映射文件I/O的优点

- **比使用read，write方式速度更快**

  这两个系统调用的典型用法：

  len = read(fd, buf, nbyte);

  len = write(fd, buf, nbyte);

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

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

内存映射文件相关系统调用(1)

- **系统调用mmap**

  通知系统把哪个文件的哪个区域以何种方式映射

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

  执行成功，返回一个指针；否则返回-1，errno记录了失败原因

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

内存映射文件相关系统调用(2)

- **举例**

  char *p;

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

  - p是一个指针，是文件fd从0开始的65536个字节
  - 程序像访问普通数组元素那样访问p[0]~p[65535]。操作这段内存最终操作磁盘文件。系统会在合适时机将修改内容写回磁盘文件或者读取文件

- **系统调用munmap**

  程序调用函数munmap，或者，进程终止时，文件的内存映射区被删除

  int munmap(void *addr, size_t len);

文件和记录的锁定

一个文件访问的问题程序

```
int fd, cnt;
fd = open("sanya",O_RDWR);
for (;;) {
     printf("按回车键售出一张票，按Ctrl-D退出 . . .");
     if (getchar() == EOF) break;
     lseek(fd, SEEK_SET, 0);
     if (read(fd, &cnt, sizeof(int)) < sizeof(int)) {
         printf("Read file error");
         exit(1);
     }
     if (cnt > 0) {
         printf("飞往三亚机票，票号:%d\n", cnt);
         cnt--;
         lseek(fd, SEEK_SET, 0);
         if (write(fd, &cnt, sizeof(int)) < sizeof(int)) {
             printf("Write file error");
             exit(1);
         }
     } else {
         printf("无票\n");
     }
 }
 close(fd);
```

文件和记录锁定机制

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

共享锁和互斥锁

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

文件锁操作：系统调用fcntl

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

结构体flock定义如下：

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

cmd在对记录上锁或解锁的应用中，应当取F_SETLKW

结构体flock

- **l_type的取值**

  F_RDLCK　对记录上锁，锁的类型为读锁

  F_WRLCK　对记录上锁，锁的类型为写锁

  F_UNLCK　对记录解锁

- **l_whence和l_start**
  - 描述了该记录从文件的何处开始
  - 描述方法与系统调用lseek()的描述方法相同(SEEK_SET/SEEK_CUR/SEEK_END)
- **l_len**
  - 描述记录大小，该记录含有多少字节
  - 当l_len取值为0时，锁的范围总是被置成从指定位置开始超过文件尾

举例(写锁)

售票程序

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

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

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

举例（读锁）

查询程序

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

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

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

复习(1)

- 什么叫进程
- 进程与程序的关系
- 进程的4个组成部分以及每部分的作用（与实例C源代码中变量和程序的对应关系）
- 进程逻辑地址空间的布局
- 进程的系统数据以及获值方法
- Unix的user/proc结构
- 进程的基本状态
- 进程调度
- 通过C实例代码解释进程状态切换
- time、vmstat命令
- 系统调用 times/clock/getrusage
- 与时间有关的几个库函数
- 忙等待
- fork
- ps命令
- 进程的命令行参数和环境变量：访问方法和在逻辑地址空间的位置
- 环境变量的三种访问方法
- exec系统调用
- exec的六种格式以及为何设置6种格式
- 僵尸进程
- 孤儿进程
- wait：销毁僵尸子进程
- C语言库函数strtok

复习(2)

- fork+exec+wait:xsh0.c
- system函数
- fork/exec与文件描述符关系
- 三级活动文件目录
- 设置三级活动文件目录的原因
- 文件描述符的继承与关闭
- close-on-exec标志
- 文件描述符的复制
- 输入输出重定向：xsh1.c
- 管道
- 管道与进程状态
- 管道的读写模型
- 命名管道
- 管道操作：xsh2.c
- kill命令
- 进程组，分组的目的
- 信号机制
- 常用的信号
- 忽略信号
- 捕捉信号
- 信号的发送：kill的三种用法
- 系统调用和进程状态与信号关系
- sleep,pause,alarm
- 全局跳转的必要性
- 全局跳转的两个主要调用
- IPC，消息队列，死锁问题
- 信号灯
- 共享内存机制
- 信号灯+共享内存:生产者消费者问题

复习(3)

- 内存映射方式访问文件
- mmap与read/write效率
- 文件和记录的锁定的必要性
- 读锁与写锁
- 咨询式锁的使用：注意安全性保障以及进程状态的变化
- 强制性锁定