资料整理
火龙果软件
荐

# 第7章
# 进程控制与进程间通信

资料整理
火龙果软件
荐

# 7.1 进程控制

资料整理
火龙果软件
荐

# 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

资料整理
火龙果软件
荐

# 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()

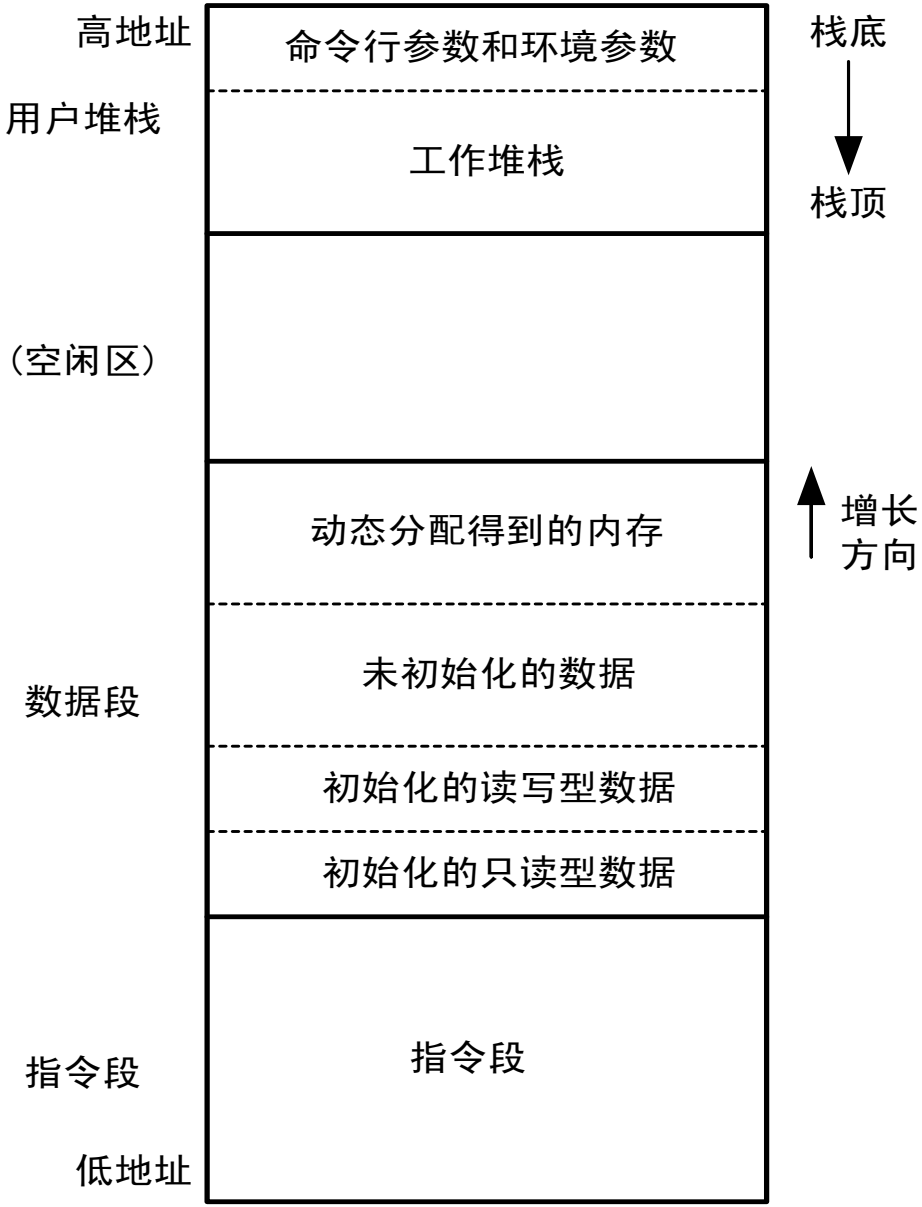
资料整理
火龙果软件
荐

# 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

资料整理
火龙果软件
荐

# 进程逻辑地址空间的布局



资料整理
火龙果软件
荐

# 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

资料整理
火龙果软件
荐

# user结构和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

资料整理
火龙果软件
荐

# 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，程序不要处于忙等待状态

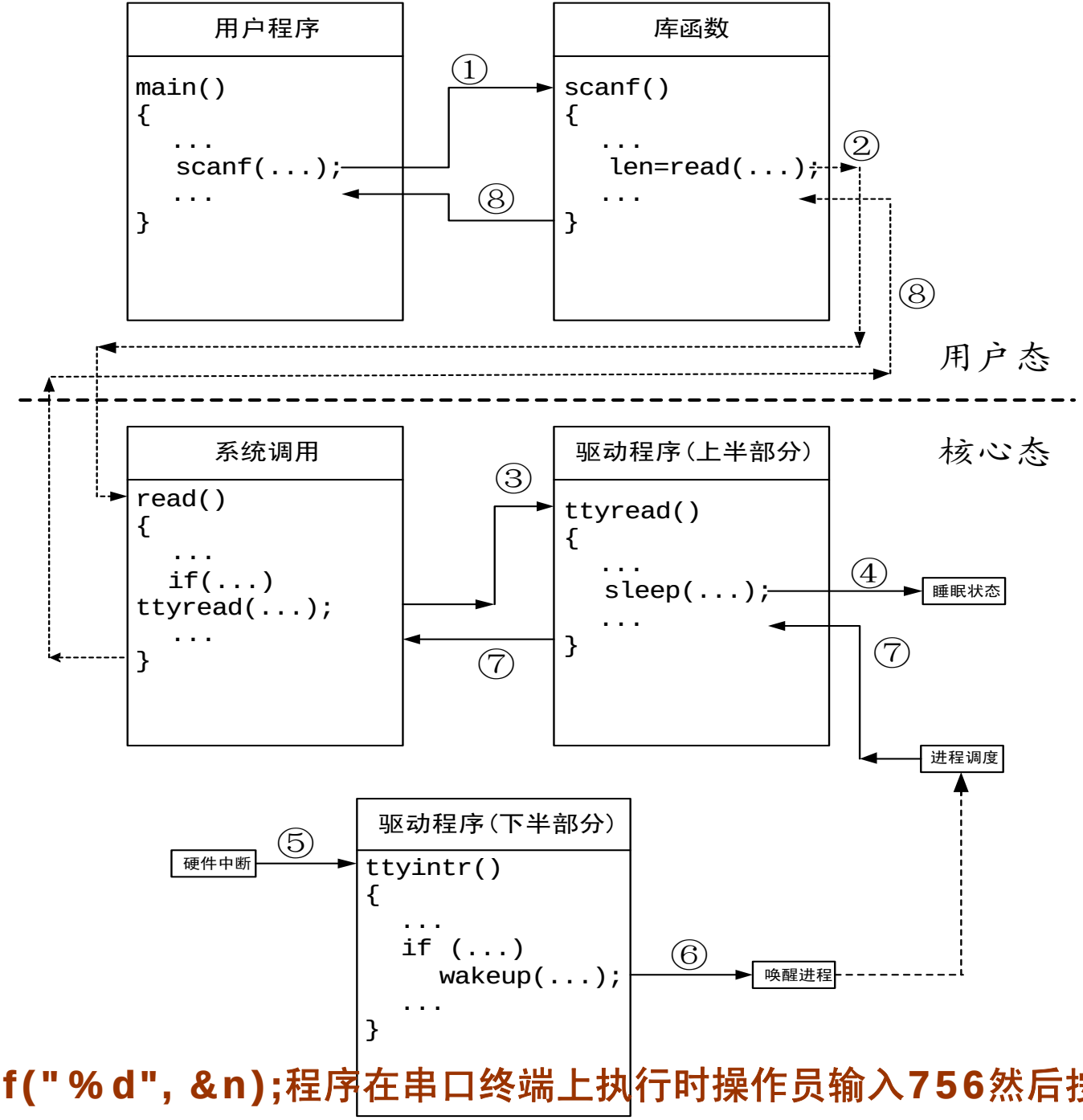
资料整理
火龙果软件
荐

# 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

资料整理
火龙果软件
荐




**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**

资料整理
火龙果软件
荐

# 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
```

资料整理
火龙果软件
荐

# 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

资料整理
火龙果软件
荐

# 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

资料整理
火龙果软件
荐

# 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

资料整理
火龙果软件
荐

# 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)

资料整理
火龙果软件
荐

```
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)

资料整理
火龙果软件
荐

```
main()
{
    int a, pid;
    printf("Hello\n");
    a = 3;
    pid = fork();
    a += 3;
    if (pid > 0) {
        printf("Pid=%d child=%d, a=%d\n",
            getpid(), pid, a);
    } else if (pid == 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=815 ppid=814,a=6
Bye
Pid=814 child=815,a=6
Bye
```

资料整理
火龙果软件
荐

# 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)
- **t**选项：列出指定终端上的所有进程(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

资料整理
火龙果软件
荐

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
```

资料整理
火龙果软件
荐

# 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)
- 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)

# 访问环境参数的三种方法

资料整理
火龙果软件
荐

```
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");
}
```

资料整理
火龙果软件
荐

# 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

资料整理
火龙果软件
荐

# 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);

资料整理
火龙果软件
荐

# “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

资料整理
火龙果软件
荐

# 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);
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**

资料整理
火龙果软件
荐

# 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

```
    \t  w  h  o        a   m    i  \n \0
20 09 77 68 6f 20 20 61 6d 20 69 0a 00
 s
```

执行p=strtok(s," \t\n");之后

```
20 09 77 68 6f 00 20 61 6d 20 69 0a 00
 s      p
```

执行p=strtok(NULL," \t\n");之后

```
20 09 77 68 6f 00 20 61 6d 00 69 0a 00
 s                         p
```

执行p=strtok(NULL," \t\n");之后

```
20 09 77 68 6f 00 20 61 6d 00 69 00 00
 s                                p
```

# 最简单的shell: xsh0

资料整理
火龙果软件
荐

```
main()
{
    char buf[256], *cmd, *argv[100];
    int n, sv;
    for(;;) {
        printf("=> ");
        if (fgets(buf, sizeof(buf), stdin) == NULL) exit(0);
        cmd = strtok(buf," \t\n");
        if (cmd != NULL) continue;
        if (strcmp(cmd, "exit") == 0) exit(0);
        n = 0;
        argv[n++] = cmd;
        while (argv[n++] = strtok(NULL, " \t\n"));
        if (fork() == 0) {
            execvp(cmd, argv);
            fprintf(stderr, "** Bad command\n");
            exit(1);
        }
        wait(&sv);
    }
}
```

资料整理
火龙果软件
荐

# 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
```

资料整理
火龙果软件
荐

# 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

资料整理
火龙果软件
荐

# 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);
}
```

资料整理
火龙果软件
荐

# 7.2 进程与文件描述符

资料整理
火龙果软件
荐

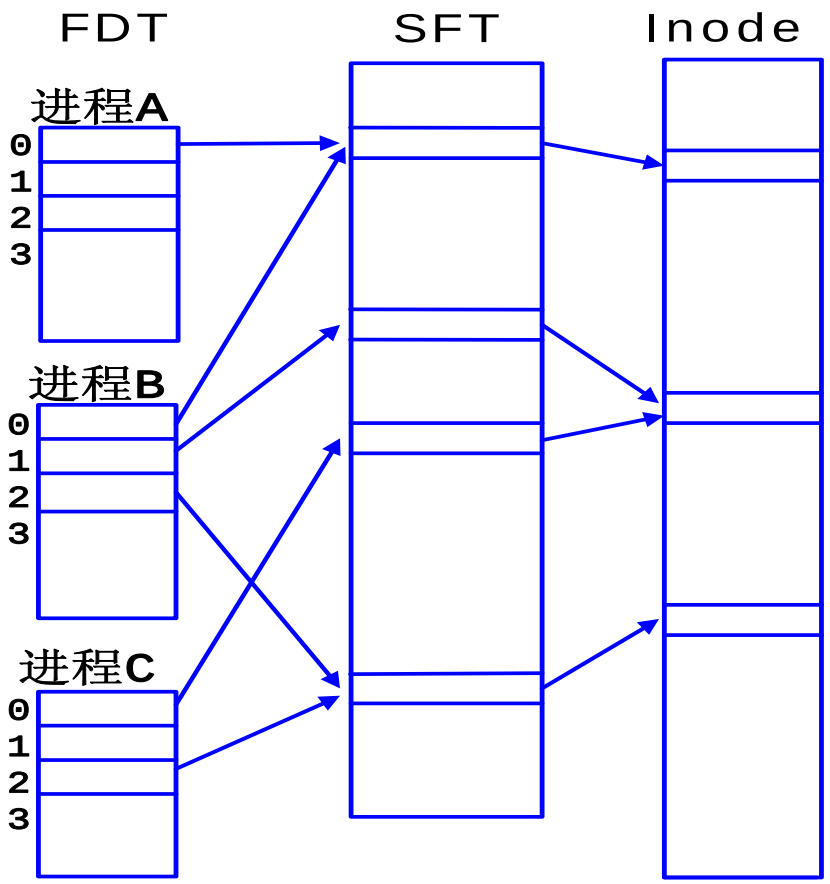
# 活动文件目录AFD

- 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，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 */
    int    f_inode;        /* 内核中inode数组的下标 */
    long f_offset[2]; /* 文件读/写位置指针 */
};
```

  - 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

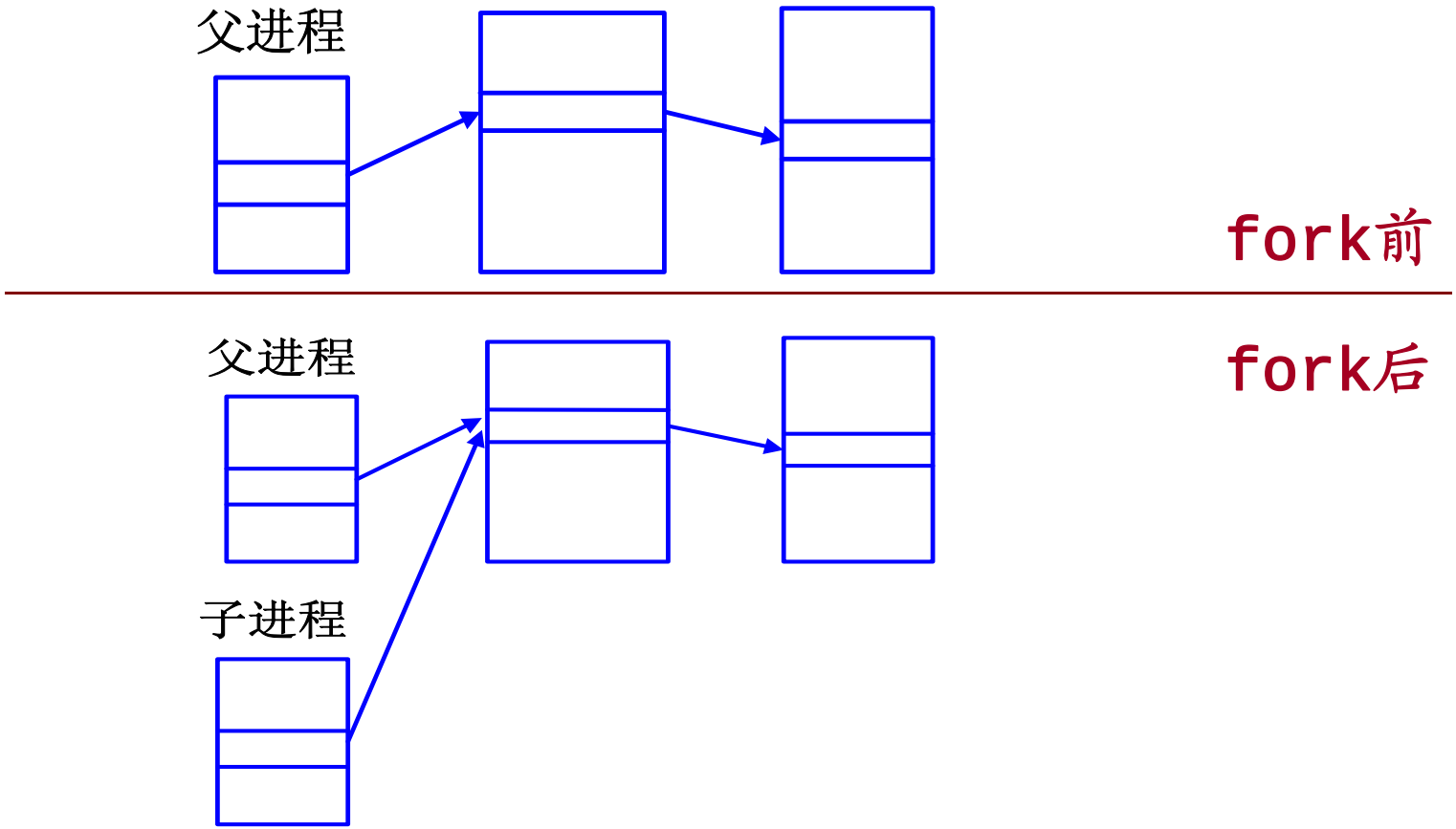
# 活动文件目录AFD(图)



资料整理
火龙果软件
荐

# 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



# 例:文件描述符的继承与关闭(1)

资料整理
火龙果软件
荐

```
#include <fcntl.h>
void main()
{
    int fd;
    static char *str = "F1:ABCD\n";
    fd = open("abc", O_CREAT | O_WRONLY, 0666);
    write(fd, str, strlen(str));
    if (fork() > 0) {
        write(fd, str, strlen(str));
        close(fd);
    } else {
        char fdstr[16];
        sprintf(fdstr, "%d", fd);
        execlp("f2", "f2", fdstr, 0);
        printf("failed to start 'f2'\n");
        exit(0);
    }
}
```

# 例:文件描述符的继承与关闭(2)

资料整理
火龙果软件
荐

## ■文件f2. c

```
int main(int argc, char **argv)
{
    int fd;
    static char *str="F2:WXYZ\n";
    fd = strtol(argv[1], 0, 0);
    write(fd, str, strlen(str));
    close(fd);
}
```

资料整理
火龙果软件
荐

# 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);
```

资料整理
火龙果软件
荐

# 文件描述符的复制

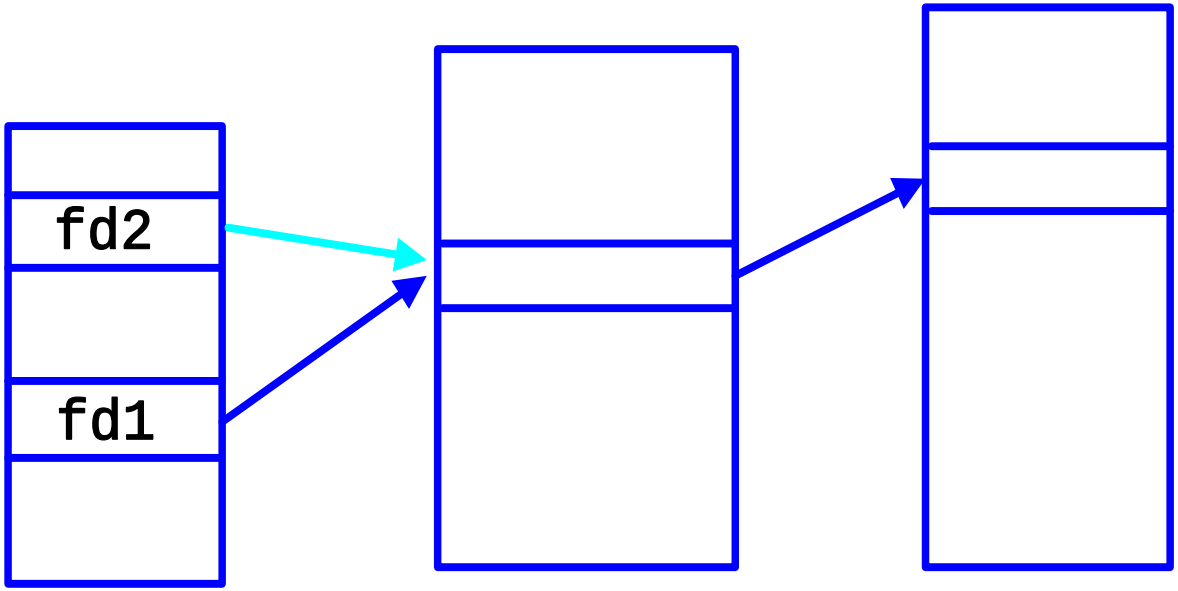
- **系统调用**

  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



资料整理
火龙果软件
荐

# xsh1:输入输出重定向(1)

```
char buf[256], *cmd, *argv[100], *infile, *outfile;
 int n, sv;
 for (;;) {
     printf("=> ");
     if (fgets(buf, sizeof(buf), stdin) == NULL) exit(0);
     infile = strstr(buf, "<");
     outfile = strstr(buf, ">");
     if (infile) {
         *infile = '\0';
         infile = strtok(infile + 1, " \t\n");
     }
     if (outfile) {
         *outfile = '\0';
         outfile = strtok(outfile + 1, " \t\n");
     }
     cmd = strtok(buf, " \t\n");
     if (cmd == NULL) continue;
     if(strcmp(cmd,"exit")==0) exit(0);
     n=0;
     argv[n++]=cmd;
     while(argv[n++]=strtok(NULL," \t\n"));
```

资料整理
火龙果软件
荐

# xsh1:输入输出重定向(2)

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

资料整理
火龙果软件
荐

# 管道操作(1)

## 创建管道

int pipe(int *pfd*[2])

- 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

资料整理
火龙果软件
荐

# 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

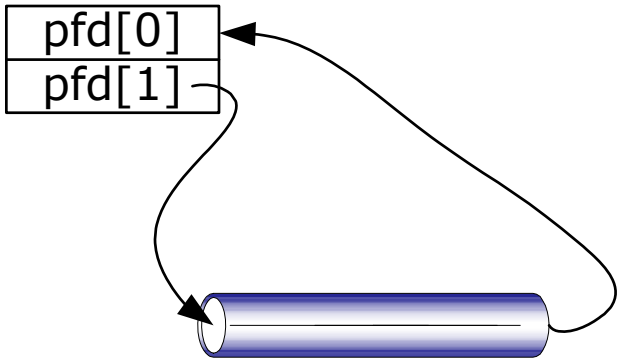
## 关闭管道close

- 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

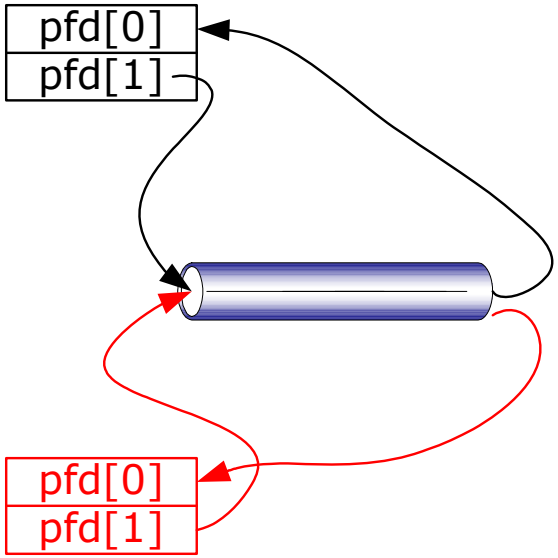
资料整理
火龙果软件
荐

# 父子进程管道通信

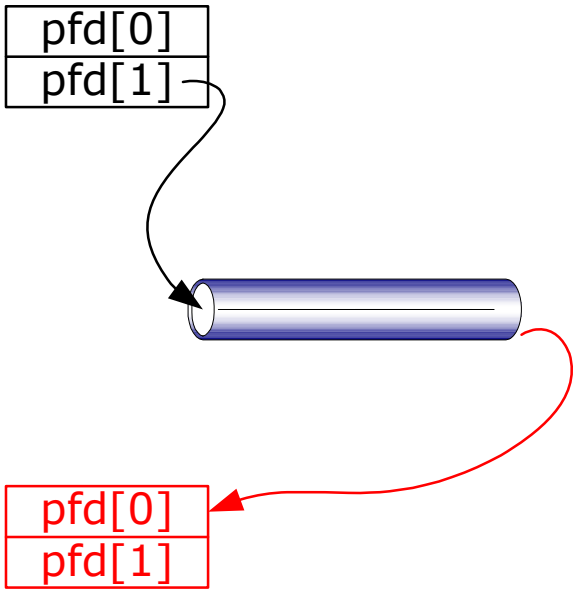
- 父进程




- fork后




- 父进程关闭读端，子进程关闭写端



资料整理
火龙果软件
荐

# 管道通信：写端

```
/* File name : pwrite.c  */
#define syserr(str) { perror(str); exit(1); }
main()
{
    int pfd[2];
    char fdstr[10], *message = "Send/Receive data using pipe";
    if (pipe(pfd) == -1) syserr("Create pipe");
    switch(fork()) {
        case -1:
            syserr("fork");
        case 0: /* child */
            close(pfd[1]);
            sprintf(fdstr, "%d", pfd[0]);
            execlp("./pread", "pread", fdstr, 0);
            syserr("Execute pread file");
            break;
        default: /* parent */
            close(pfd[0]);
            if(write(pfd[1], message, strlen(message) + 1) == -1)
                perror("Write pipe");
            break;
    }
}
```

# 管道通信：读端

资料整理
火龙果软件
荐

```
/* File name : pread.c */
#define syserr(str) { perror(str); exit(1); }
main(int argc, char **argv)
{
    int fd, nbyte, i;
    char buf[1024];
    fd = strtol(argv[1], 0, 0);
    nbyte = read(fd, buf, sizeof(buf));
    switch (nbyte) {
        case -1:
            syserr("Read pipe");
            break;
        case 0:
            printf("Pipe closed\n");
            break;
        default:
            printf("%d bytes [%s]\n", nbyte, buf);
            break;
    }
    exit(0);
}
```

资料整理
火龙果软件
荐

# 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**

资料整理
火龙果软件
荐

# 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));
```

资料整理
火龙果软件
荐

# xsh2:管道(1)

```
main()
{
    char buf[256], *buf2, *cmd, *cmd2, *argv[64], *argv2[64];
    int n, sv, fd[2];
    for (;;) {
        printf("=> ");
        if (fgets(buf, sizeof(buf), stdin) == NULL) exit(0);
        buf2 = strstr(buf, "|");
        if (!buf2) exit(0);
        *buf2++ = '\0';
        cmd = strtok(buf, " \t\n");
        if (!cmd) exit(1);
        n = 0;
        argv[n++] = cmd;
        while (argv[n++] = strtok(NULL, " \t\n"));
        cmd2 = strtok(buf2, " \t\n");
        if (!cmd2) exit(1);
        n = 0;
        argv2[n++] = cmd2;
        while (argv2[n++] = strtok(NULL, " \t\n"));
```

资料整理
火龙果软件
荐

# xsh2:管道(2)

```
        pipe(fd);
        if (fork() == 0) {
            dup2(fd[0], 0);
            close(fd[0]);  close(fd[1]);
            execvp(cmd2, argv2);
            fprintf(stderr, "** bad command\n");
            exit(1);
        } else if (fork() == 0) {
            dup2(fd[1], 1);
            close(fd[0]);
            close(fd[1]);
            execvp(cmd, argv);
            fprintf(stderr, "** bad command\n");
            exit(1);
        }
        close(fd[0]);
        close(fd[1]);
        wait(&sv);
        wait(&sv);
    }
}
```

资料整理
火龙果软件
荐

# 第二次上机作业

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

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

资料整理
火龙果软件
荐

# 7.3 信号

资料整理
火龙果软件
荐

# 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

资料整理
火龙果软件
荐

# 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

资料整理
火龙果软件
荐

# 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

资料整理
火龙果软件
荐

# 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

资料整理
火龙果软件
荐

# 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

资料整理
火龙果软件
荐

# 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

资料整理
火龙果软件
荐

# 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。

资料整理
火龙果软件
荐

# 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

资料整理
火龙果软件
荐

# 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);
        ⋮
}
```

资料整理
火龙果软件
荐

# 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

资料整理
火龙果软件
荐

# 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)

资料整理
火龙果软件
荐

# 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。

资料整理
火龙果软件
荐

# alarm系统调用举例

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

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

资料整理
火龙果软件
荐

# 全局跳转(1)

```
main()
{
    int c;
    for (;;) {
        printf("Input a command:");
        c = getchar();
        switch(c) {
            case 'Q': return(0);  break;
            case 'A': func_a();   break;
            case 'B': func_b();   break;
                .
                .
                .
        }
    }
}
```

- **问题**
  - 用户在输入了命令A后反悔，希望中止func_a()对命令A的处理，而要重新选择另一条命令

资料整理
火龙果软件
荐

# 全局跳转(2)

```
#include <sys/signal.h>
void main_control(int sig)
{
    int c;
    signal(SIGINT, main_control);
    for (;;) {
        printf("Input your choice:");
        scanf("%d", &c);
        switch (c) {
            case 0: return; break;
            case 1: func_1();  break;
            case 2: func_2();  break;
                .
                .
        }
    }
}
main()
{
    main_control();
}
```

资料整理
火龙果软件
荐

# 全局跳转(3)

## 上述程序存在的问题

- 每次按下中断键，程序停留在信号捕捉函数中，堆栈没有清理，嵌套越来越深浪费越来越大
- main_control()一旦返回，进程的执行将弹回到刚才被SIGINT中断的地方恢复刚才的执行
  - 会让用户感到迷惑不解，刚才的动作已打断而且又已经开始了新的工作，可过一段时间后又开始运行
- C语言中的goto语句只限于一个函数体内使用，不能解决上述的问题

资料整理
火龙果软件
荐

# 全局跳转(4)

- **解决方法**
  - 把栈恢复为第一次调用main_control()时的状态，再调用main_control()去重新执行
- **有两个标准的子例程用于这个目的**

```
#include <setjmp.h>
int setjmp(jmp_buf jmpenv);
    /* 返回值为零，或者是longjmp提供的val */
void longjmp(jmp_buf jmpenv, int val)
    /* val是提供给setjmp作返回值*/
```

资料整理
火龙果软件
荐

# 全局跳转(5)

```
#include  <sys/signal.h>
#include <setjmp.h>
static jmp_buf jmpbuf;
void intr_proc(int sig)
{
    printf("\n...INTERRUPTED by signal %d\n", sig)
    longjmp(jmpbuf, 1);
}
main()
{
    int c;
    setjmp(jmpbuf);
    signal(SIGINT, intr_proc);
    for(;;) {
        printf("Input a command:");
        scanf("%d", &c);
        switch(c) {
            case 0: return; break;
            case 1: func_1();  break;
            case 2: func_2();  break;
        }
    }
}
```

资料整理
火龙果软件
荐

# 7.4 消息队列

资料整理
火龙果软件
荐

# 消息队列的创建

- **系统调用msgget**

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

  - 在内核中创建新消息队列或获取已存在消息队列的ID，发送/接收消息的系统调用使用qid(qid是内核中msqid_ds数组的下标)
  - 创建消息队列

```
qid = msgget(462101, IPC_CREAT | 0666);
```

  - 取得一个已存在的消息队列

```
qid = msgget(462101, 0);
```

资料整理
火龙果软件
荐

# 消息队列的系统数据结构

- 内核中每个消息队列对应一个msqid_ds结构

```
struct msqid_ds { /* msg ID data structure */
    struct ipc_perm   msg_perm; /* operation permission struct */
    struct msg        *msg_first;   /* ptr to first message on q*/
    struct msg        *msg_last;    /* ptr to last message on q*/
    ushort            msg_cbytes;   /* current # bytes on q    */
    unsigned short    msg_qnum;     /* # of messages on q  */
    unsigned short    msg_qbytes;   /* max # of bytes on q */
    ushort            msg_lspid;    /* pid of last msgsnd      */
    ushort            msg_lrpid;    /* pid of last msgrcv      */
    time_t            msg_stime;    /* last msgsnd time        */
    time_t            msg_rtime;    /* last msgrcv time        */
    time_t            msg_ctime;    /* last change time        */
};
```

资料整理
火龙果软件
荐

# 消息队列的状态和删除

- **系统调用msgctl**

  int msgctl(int *qid*, int *cmd*, struct msqid_ds **sbuf*);

- **将消息队列从内核中删除**

  msgctl(qid, IPC_RMID, 0);

- **查看消息队列当前状态**

  struct msqid_ds st;

  msgctl(qid, IPC_STAT, &st);

  - 获取内核消息队列的系统数据结构，包括消息队列的总容量，当前队列有多少个消息，消息共多少个字节，等等

资料整理
火龙果软件
荐

# 发送消息msgsnd

int msgsnd(int *qid*, struct msgbuf **buf*, int *nbytes*, int *flags*);

- 返回值为0(成功)，或者为-1(失败)
- *buf*是一个指针，指定一个缓冲区

```
struct msgbuf {
    long mtype;  /* mtype必须大于零 */
    char mtext[1];
};
```

前4字节存放mtype，从第5字节开始*nbytes*字节的数据作为消息的有效内容(*nbytes*没有将mtype的4字节计在内)

- *flags*
  - flags=0时，阻塞方式：如果队列满则等待直到能够把消息发送出去为止。除非有意外发生(如：消息队列被删掉，有信号到达)
  - flags=IPC_NOWAIT时，非阻塞方式：如果队列满，则msgsnd马上返回-1，不等待，errno被设为EAGAIN

资料整理
火龙果软件
荐

# 接收消息msgrcv

int msgrcv(int *qid*, struct msgbuf **buf*, int *nbytes*,
long *mtype*, int *flags*);

- 返回值为-1或者实际接收到的消息中有效数据字节数
- *buf*：用于存放实际收到的消息缓冲区(类型和有效内容)
- *nbytes*：期望收到的消息的最大长度
  - 如果实际消息的长度>*nbytes*会导致msgrcv出错
- *mtype*分三种情况
  - =0：返回队列上的第一个消息
  - >0：返回类型为*mtype*的第一个消息
  - <0：返回第一个类型值≤-*mtype*的消息
- *flags*
  - *flags*=0，如果消息队列空，则等待
  - *flags*=IPC_NOWAIT，若队列空则不等待，立即返回-1，errno被置为ENOMSG

资料整理
火龙果软件
荐

# 命令ipcs和ipcrm

- **命令ipcs：打印IPC的状态**
  - 选项-q：消息队列有关的信息
  - 选项-o：outstanding(未完成的)，滞留在消息队列内的数据的信息，如：CBYTES，QNUM等
  - 选项-p：显示进程号，如LSPID，LRPID
  - 选项-a：显示所有(all)的信息
- **ipcrm用于删除一个已存在的队列**
  - 如：ipcrm -q 700

    删除ID为700的消息队列

资料整理
火龙果软件
荐

# 命令ipcs: 举例

**ipcs -opq**

```
IPC status from /dev/kmem as of Tue Jul 30 13:09:16 2006
T  ID     KEY        MODE    OWNER  GROUP CBYTES QNUM LSPID LRPID
Message Queues:
q 700 0x00001111 -Rrw-rw-rw- root  other      0     0   1009  1017
q    1 0x00123456  --rw-rw-rw- root  other   3450    23  1035  1041
```

**ipcs -aq**

```
IPC status from /dev/kmem as of Tue Jul 30 13:09:25 2006
T  ID     KEY     MODE    OWNER GROUP CREATOR CGROUP CBYTES QNUM
 QBYTES LSPID LRPID    STIME     RTIME     CTIME
Message Queues:
q 700 0x00001111 -Rrw-rw-rw-  root other root     other     0     0
   4096  1009  1017 13:07:59 13:07:59 13:02:40
q   1 0x00123456 --rw-rw-rw-  root other root     other 3450    23
   4096  1035  1041 13:08:28 13:08:43 13:08:28
```

资料整理
火龙果软件
荐

# 程序举例

- 程序c.c作为前台客户程序，等待用户输入数据，收到数据后，将数据发给服务器进程s
- 进程s在对收到的数据作了加工处理后，将结果发回进程c
- 程序s.c作为服务器进程，运行时以后台进程的形式出现
- 系统中允许同时有多个进程c存在，但仅有一个s进程
- 为区分多个进程c，用客户进程PID作为消息类型

资料整理
火龙果软件
荐

# 源程序文件sc.h

```
/* File name : sc.h */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>

#define SC_KEY 0x1111
#define M_IN    1

struct Msg {
    long mtype;
    long src_pid;
    char string[1000];
};

#define MAX_MSG_LEN (sizeof(struct Msg) – sizeof(long))
```

资料整理
火龙果软件
荐

# 后台服务进程s.c

```
main()
{
    struct Msg rmsg, smsg;
    int qid, rlen, slen, i;
    qid = msgget(SC_KEY, IPC_CREAT | 0666);
    for (;;) {
        rlen = msgrcv(qid, &rmsg, MAX_MSG_LEN, M_IN, 0);
        slen = rlen;
        smsg.mtype = rmsg.src_pid;
        smsg.src_pid = 0; /* useless */
        for (i = 0; i < slen - sizeof(long); i++)
            smsg.string[i] = toupper(rmsg.string[i]);
        msgsnd(qid, &smsg, slen, 0);
    }
}
```

资料整理
火龙果软件
荐

# 前台客户进程c.c

```
main()
{
    int qid = -1, pid, ret, slen, rlen;
    struct Msg smsg, rmsg;
    qid = msgget(SC_KEY, 0);
    pid = getpid();
    for (;;) {
        printf("Input a string or ^D for exit: ");
        if (gets(smsg.string) == NULL) exit(0);
        smsg.mtype = M_IN;
        smsg.src_pid = pid;
        slen = sizeof(long) + strlen(smsg.string) + 1;
        ret = msgsnd(qid, &smsg, slen, 0);
        rlen = msgrcv(qid, &rmsg, MAX_MSG_LEN, pid, 0);
        printf("Result : [%s]\n", rmsg.string);
    }
}
```

资料整理
火龙果软件
荐

# 程序缺陷

- 上述客户和服务端程序缺陷
  - 以PID作为消息类型可能导致消息队列中产生垃圾
- 改进方法
  - 例如：以终端号作为消息类型，或者操作员编号
- 需要注意的问题
  - 垃圾和失步

资料整理
火龙果软件
荐

# 消息队列的特点

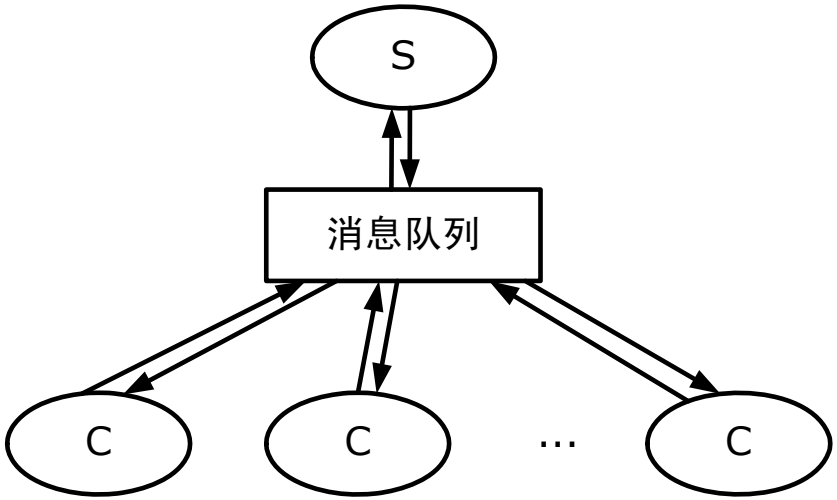
## 主要特点

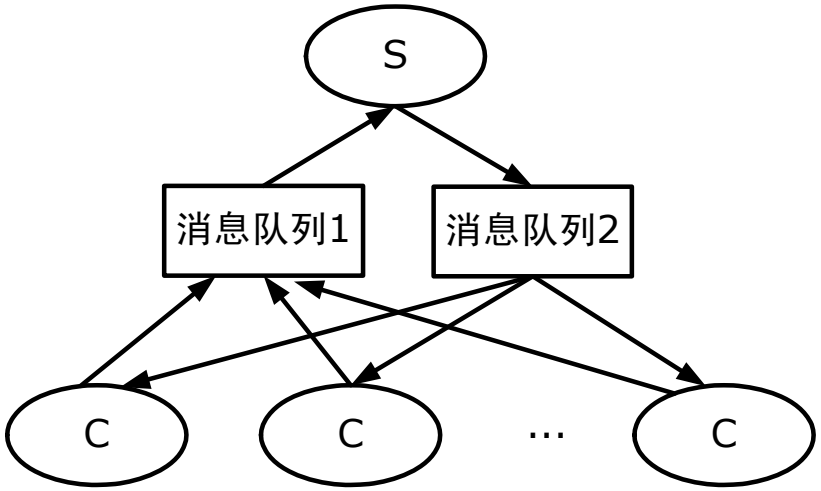
- 保持了记录边界
- 独立启动的进程可以共享消息队列
- 可以多路复用，实现了信箱的功能，便于多个进程共享消息队列
- 同共享内存相比，消息队列需要两次复制数据
- 创建消息队列的进程死亡后，不能自动把消息队列从系统(内核)中删掉

资料整理
火龙果软件
荐

# 死锁

- 设计进程/线程通信时，死锁分析必不可少：使用管道，消息队列, 信号灯都可能产生死锁



资料整理
火龙果软件
荐

# 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

资料整理
火龙果软件
荐

# 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

资料整理
火龙果软件
荐

# 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);
```

资料整理
火龙果软件
荐

# 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

资料整理
火龙果软件
荐

# 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

资料整理
火龙果软件
荐

# 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

资料整理
火龙果软件
荐

# “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