进程控制

进程的生命周期：从生到死

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PCB，复制父进程环境(包括PCB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，COW)，但是系统核心的这些安排，对程序员透明

fork举例(1)

```c
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)

```
int main(void)
{
    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)
- 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(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);
```

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

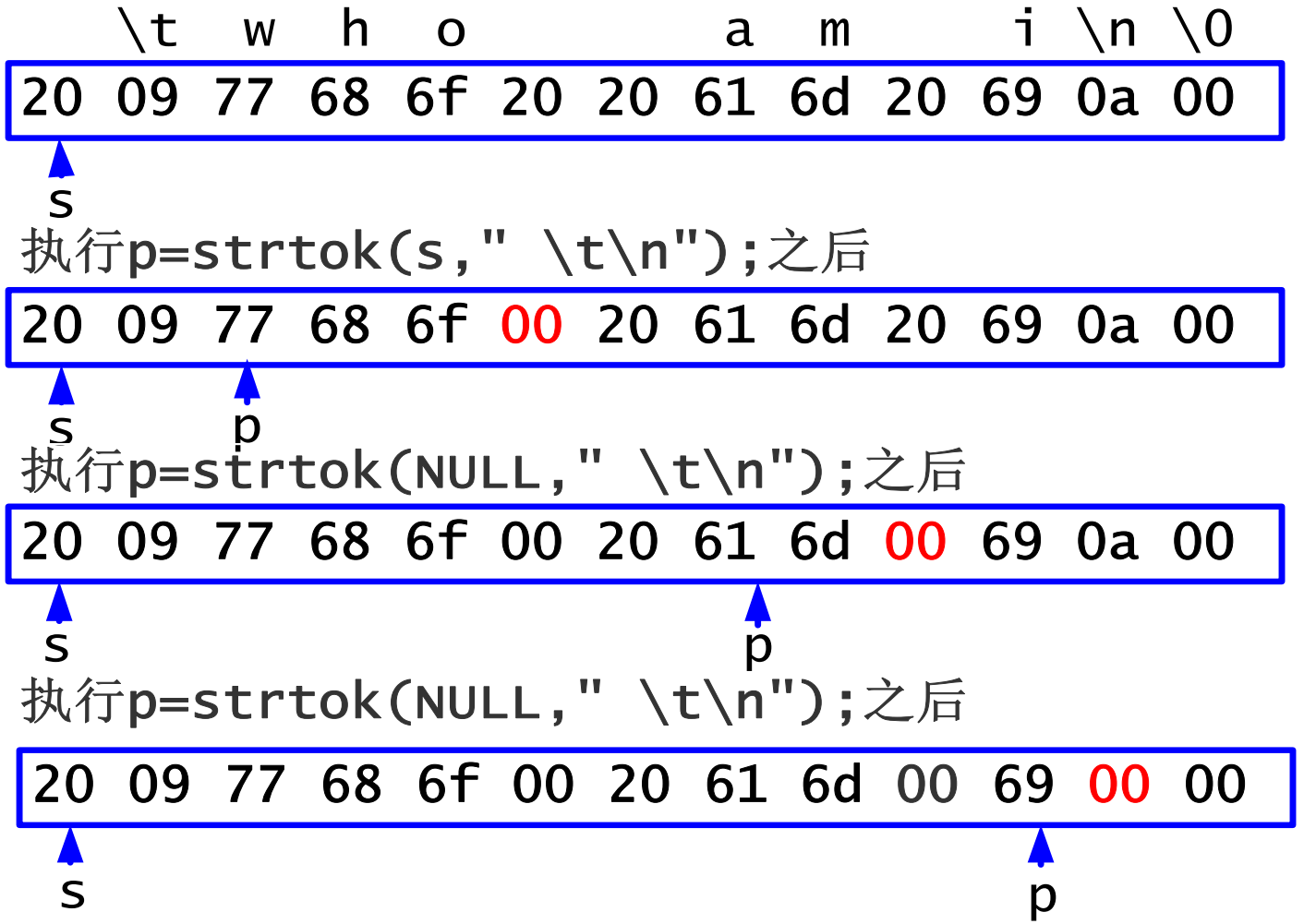
- **waitpid()和wait3() : wait系统调用的升级版本**

自编shell: xsh0

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



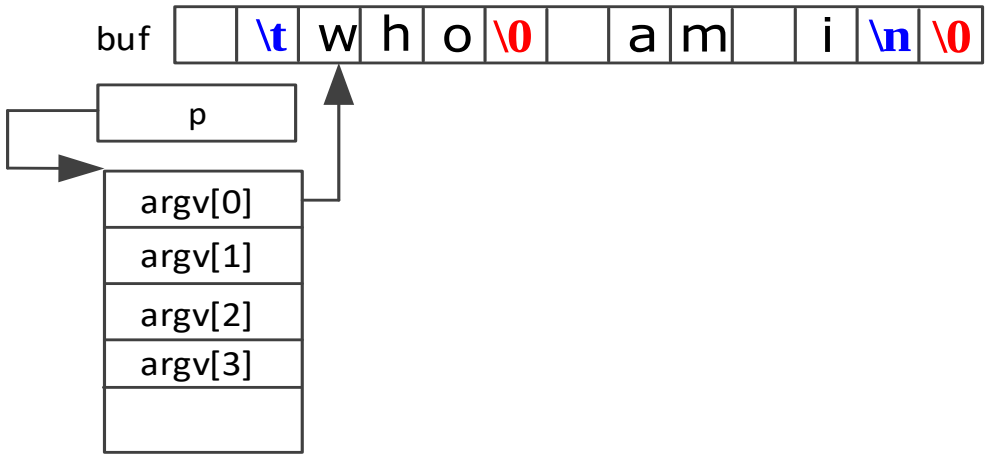
最简单的shell: xsh0

```
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

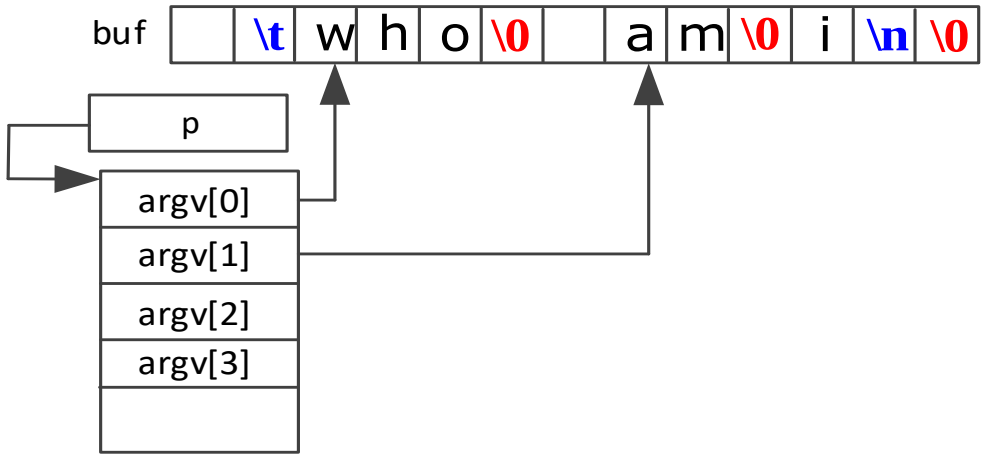
执行for循环初始化操作后的状态

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

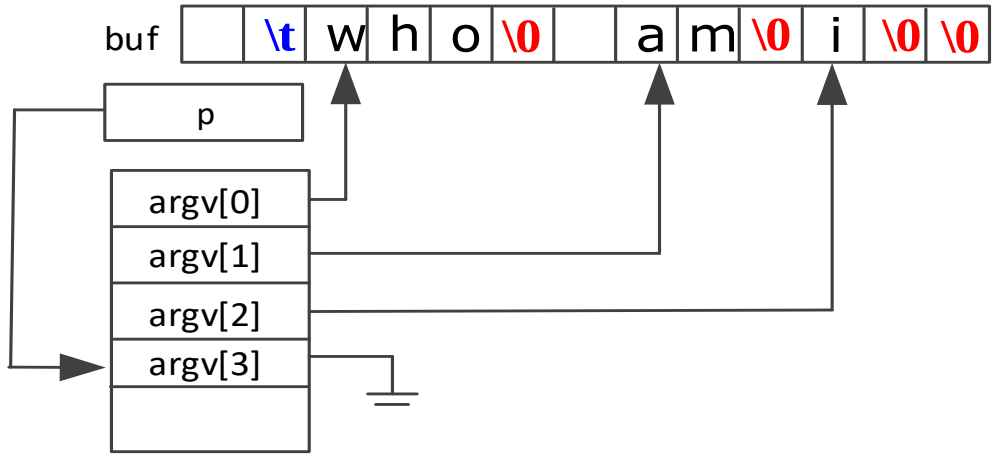


执行for循环第1轮之后的状态

```
*++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
```

库函数system：运行一个命令

int system(char *_string_);

- 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

库函数system应用举例

```
int main(void)
{
    char fname[256], cmd[256], buf[256];
    FILE *f;

    sprintf(fname, "/tmp/eth-status-%d.txt", getpid());
    sprintf(cmd, "ifconfig -a > %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);
}
```

进程控制：小结

- 深入理解系统调用fork()
- 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：等待子进程终止并销毁僵尸子进程获取其退出状态
- C语言库函数strtok
- fork+exec+wait:xsh0.c
- 库函数system的作用

重定向与管道

进程与文件描述符

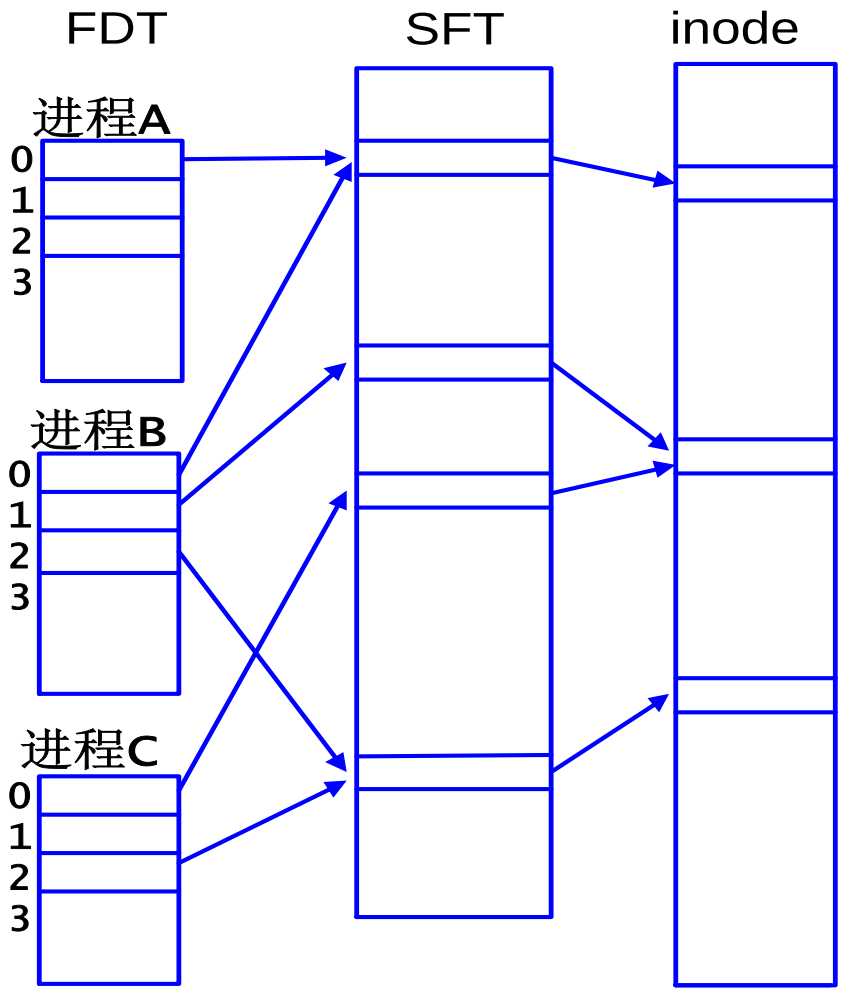
活动文件目录AFD

- **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 */
};
```

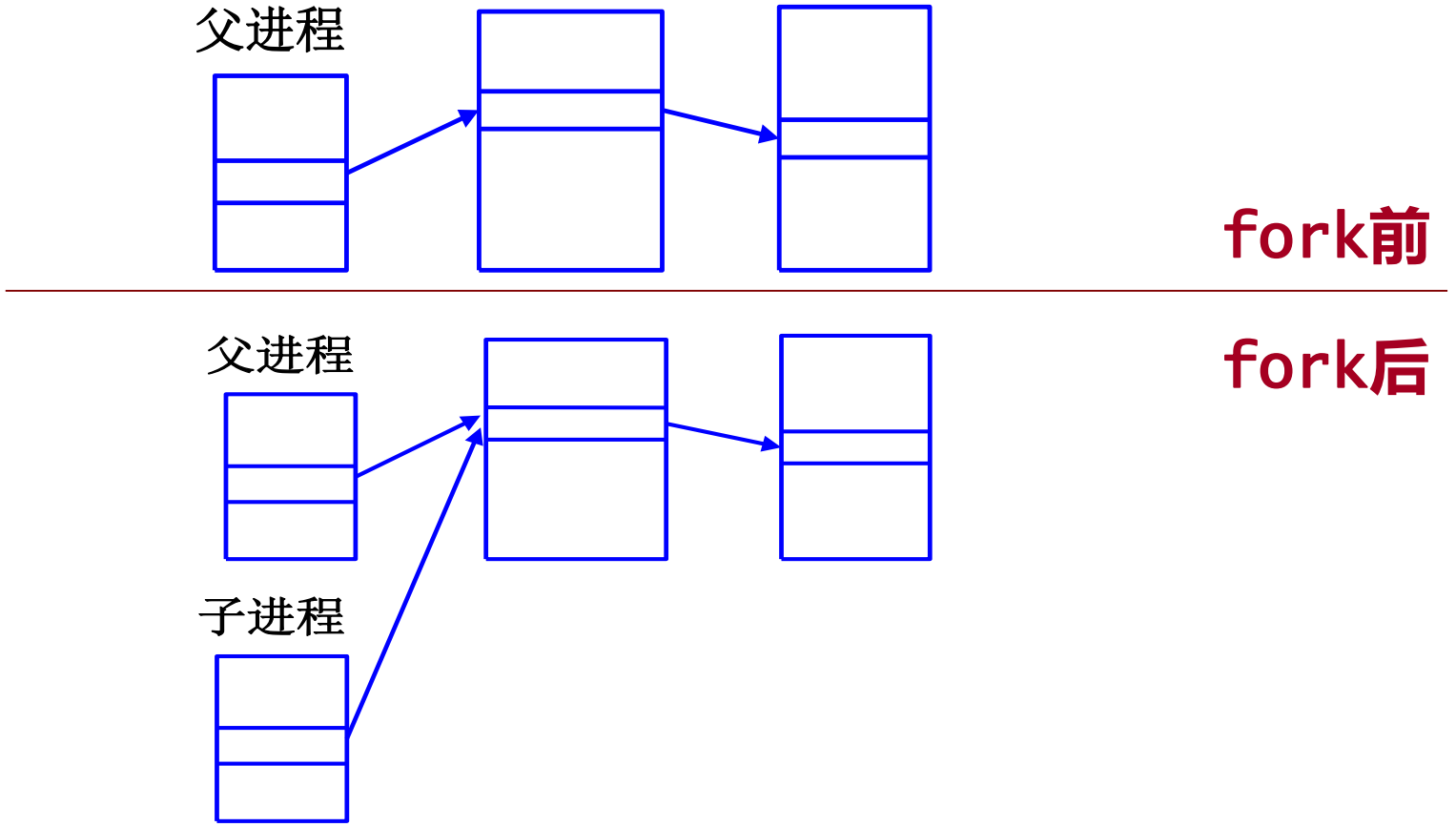
  - **活动i节点表**：整个核心一张，inode结构
    - 内存中inode表是外存中inode的缓冲
    - 内存inode表里也有个专用的引用计数

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



例:文件描述符的继承与关闭(1)

文件f1. c

```
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

- **文件设置了close-on-exec标志，执行exec()系统会自动关闭这些文件**
  - 在open()调用的第三个参数里可以加O_CLOEXEC属性
  - 通过系统调用fcntl()设置
- **函数**

  #include <fcntl.h>

  int fcntl (*fd*, *cmd*, *arg*);

  cmd: F_GETFD 获取文件*fd*的控制字flag，控制字的比特0为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);
```

重定向

文件描述符的复制

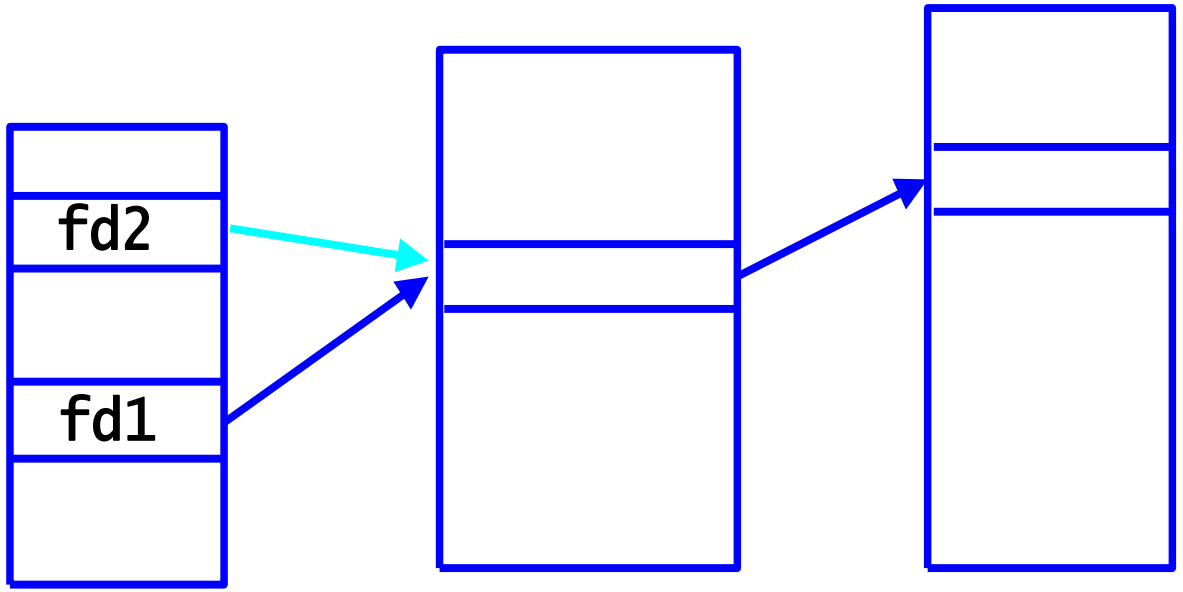
- **系统调用**

  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
sort -fr < /etc/passwd > xxout.txt \n \0
Infile
outfile
\0
argv
0
1
2
NULL
infile

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);
        }
    }
```