目录

一、 进程

1. 基本概念

程序：静态文件

进程：运行着的实体

进程是一个实体。每一个进程都有它自己的地址空间，一般情况下，包括文本区域（text region）、数据区域（data region）和堆栈（stack region）。文本区域存储处理器执行的代码；数据区域存储变量和进程执行期间使用的动态分配的内存；堆栈区域存储着活动过程调用的指令和本地变量。

进程是一个“执行中的程序”。程序是一个没有生命的实体，只有处理器赋予程序生命时（操作系统执行之），它才能成为一个活动的实体，我们称其为进程。

2. 结构特征

进程由程序、数据和进程控制块（PID）三部分组成。

3. 查看进程信息

进程关系：pstree（进程之间存在“父子关系”“兄弟关系”这样的亲缘关系）

进程的身份证 PID：ps -ef

二、 创建进程

1. 进程复制

函数原型：pid_t fork(void);

fork() 这个函数很特殊，成功创建子进程后居然有两个返回值，给父进程返回子进程 pid，给子进程返回 0，如果创建失败那么就返回 -1。

fork 函数执行完后会复制一份原来的进程（创建进程）。

fork 函数后面的代码会执行两遍。fork 之后父进程先执行还是子进程先执行不确定，取决于内核所使用的调度算法。

2. 子进程“偷梁换柱”

当用 fork 函数创建新的子进程后，子进程往往要调用一种 exec 函数以执行另一个程序。当程序调用一种 exec 函数时，该进程执行的程序完全替换为新程序，而新程序则从其 main 函数开始执行。因为调用 exec 并不创建新进程，所以前后的进程 ID 并未改变。exec 只是用磁盘上的一个新程序替换了当前进程的正文段、数据段、堆段和栈段。

有 7 种不同的 exec 函数可供使用，它们常常被统称为 exec 函数。

```
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 execvpe(const char *file, char *const argv[],char *const
envp[]);
```

*path：可执行文件的路径名。

*arg：可执行程序所带的参数，第一个参数为可执行文件名字，没有带路经且 arg 必须以 NULL 结束。

*file：如果参数 file 中包含/，则就将其视为路径名，否则就按 PATH 环境变量，在它所指定的各目录中搜寻可执行文件。

函数名中的字符可以帮助理解和分辨：

l（list）表示以参数列表的形式调用（需要指定绝对路径执行）。

v（vector）表示以参数数组的方式调用

e（environment）表示用户提供自定义环境变量，当调用 execve 函数时，操作系统首先根据 filename 指定的路径和名称找到对应的可执行文件。然后，操作系统创建一个新的进程，并将该可执行文件加载到新进程的内存空间中。接下来，操作系统将新进程的参数和环境变量设置为 argv 和 envp 指定的内容。最后，操作系统启动新进程的执行，从新程序的入口点开始执行代码。

p（path）表示 PATH 中搜索执行的文件，如果给出的不是绝对路径就会去 PATH 搜索相应名字的文件，如 PATH 没有设置，则会默认在/bin,/usr/bin 下搜索。

- 返回值：

如果执行成功则函数不会返回，执行失败则直接返回-1，失败原因存于 errno 中。

- 注意事项

l、v 后缀必须任选其一使用，p、e 任选其一。

有可能执行失败：文件路径错误；没有加 NULL 结尾；新程序没有执行权限

三、 进程退出

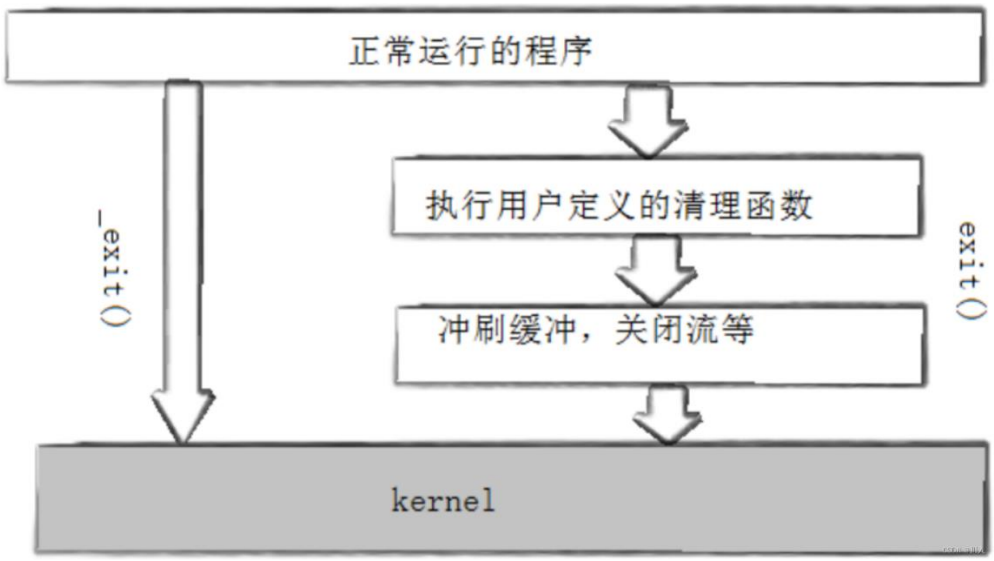
1. 函数

exit()函数用于正常退出进程。当调用 exit()函数时，它会执行一些清理操作（如关闭打开的文件描述符）并终止进程。它还通过返回一个退出状态码将控制权返回给操作系统。exit()函数可用于任意函数中，通过调用它可以使整个进程退出。

_exit()系统调用是直接终止进程。当调用_exit()函数时，进程立即终止，而不执行任何清理操作。与 exit()不同，_exit()函数不返回任何状态给操作系统，而是直接终止进程。通常，_exit()函数用于异常情况或在需要立即终止进程而不进行清理操作的情况下。

return 语句用于退出函数或方法，并将控制权返回到调用该函数或方法的位置。return 语句只能用于函数或方法的内部，而不能使整个进程退出。当函数或方法中的所有代码都执行完毕或遇到 return 语句时，该函数或方法的执行将结束，控制权将返回给调用者。

2. exit()和_exit()区别



四、 等待子进程退出

1. wait()函数

pid_t wait(int *status);

功能：等待任一子进程终止，如果子进程终止了，此函数会回收子进程的资源。

调用 wait 函数的进程会被挂起(阻塞)，直到它的一个子进程退出或收到一个不能被忽视的信号时才被唤醒。若调用进程没有子进程或它的子进程已经结束，该函数立即返回。

参数：函数返回时，参数 status 中包含子进程退出时的状态信息。

子进程的退出信息在一个 int 中包含了多个字段，用宏定义可以取出其中的每个字段。

返回值：如果执行成功则返回子进程的进程号。出错返回-1，失败原因存在 errno 中。

```
int main()
{
    pid_t result;
    int status;
    result = fork();

    if(result == -1){
        printf("error!!!\r\n");
    }
    if(result == 0){
        printf("son!!!\r\n");
        exit(0);
    }
    else{
        wait(&status);
        if(WIFEXITED(status) == 1){
            printf("exit value:%d\r\n",WEXITSTATUS(status));
            return 0;
        }
    }

}
```

WIFEXITED(status) 这个宏用来指出子进程是否为正常退出的，如果是，它会返回一个非零值（请注意，虽然名字一样，这里的参数 status 并不同于 wait 唯一的参数---指向整数的指针 status，而是那个指针所指向的整数，切记不要搞混了）

WEXITSTATUS(status) 当 WIFEXITED 返回非零值时，我们可以用这个宏来提取子进程的返回值，如果子进程调用 exit(5)退出，WEXITSTATUS(status) 就会返回 5；如果子进程调用 exit(7)，WEXITSTATUS(status)就会返回 7。请注意，如果进程不是正常退出的，也就是说， WIFEXITED 返回 0，这个值就毫无意义。

2. waitpid 函数

pid_t waitpid(pid_t pid, int *status, int options);

功能：等待指定子进程终止，如果子进程终止了，此函数会回收子进程的资源。

返回值：如果执行成功则返回子进程 ID。出错返回-1，失败原因存于 errno 中。

---- 从本质上讲，系统调用 waitpid 和 wait 的作用是完全相同的，但 waitpid 多出了两个可由用户控制的参数 pid 和 options，从而为我们编程提供了另一种更灵活的方式。

pid：当参数 pid 取不同的值时，有不同的意义：

1> pid>0 时，只等待进程 ID 等于 pid 的子进程，不管其它已经有多少子进程运行结束退出了，只要指定的子进程还没有结束，waitpid 就会一直等下去。

2> pid=-1 时，等待任何一个子进程退出，没有任何限制，此时 waitpid 和 wait 的作用一模一样。

3> pid=0 时，等待和该进程在同一个进程组中的任何子进程，如果某个子进程已经加入了别的进程组，waitpid 不会对它做任何理睬。

options：目前在 Linux 中只支持 WNOHANG 和 WUNTRACED 两个选项，可以用"|"运算符把它们连接起来使用，如：

ret = waitpid(-1,NULL,WNOHANG|WUNTRACED);

WNOHANG，表示即使没有子进程退出，它也会立即返回，不会像 wait 那样永远等下去。

WUNTRACED，与跟踪调试有关，极少用到。

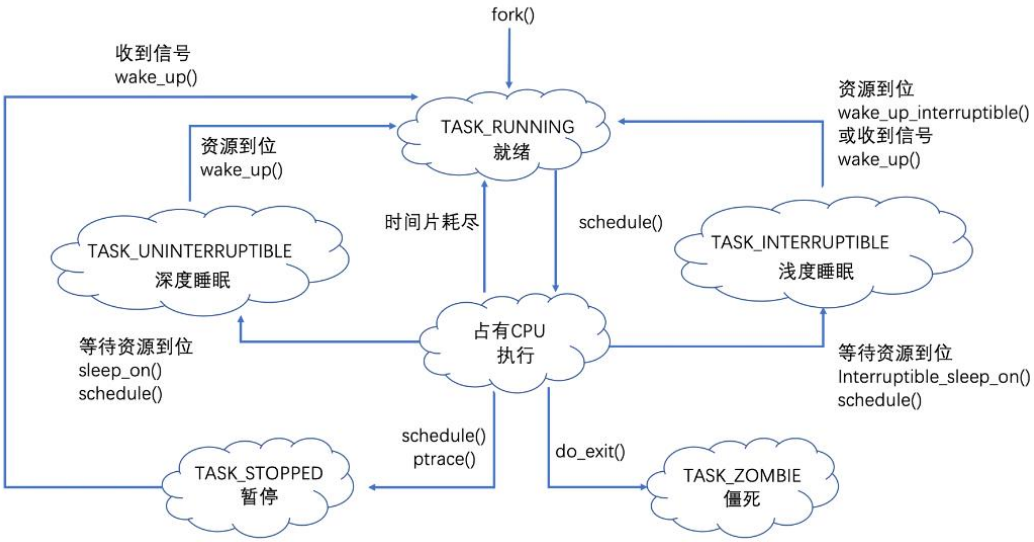
waitpid 的返回值比 wait 稍微复杂一些，一共有 3 种情况：

>> 正常返回的时候，waitpid 返回收集到的子进程的进程 ID；

>> 如果设置了选项 WNOHANG，而调用中 waitpid 发现没有已退出的子进程可收集，则返回 0；

>> 调用中出错，则返回-1，这时 errno 会被设置成相应的值以指示错误所在。例如：当 pid 所指示的子进程不存在，或此进程存在，但不是调用进程的子进程，waitpid 就会出错返回，这时 errno 被设置为 ECHILD。

五、 进程的生命周期



一个进程被 fork 出来后，进入**就绪态**；当被调度到获得 CPU 执行时，进入**执行态**；如果时间片用完或被强占时，进入**就绪态**；资源得不到满足时，进入**睡眠态(深度睡眠或浅度睡眠)**，比如一个网络程序，在等对方发包，此时不能占着 CPU，进入睡眠态，当包发过来时，进程被唤醒，进入就绪态；如果被暂停，进入停止态；执行完成后，资源释放，此时父进程 wait4 还未收到它的信号，进入**僵死态**。

1. 就绪态和执行态

就绪态：进程准备就绪，等待被 CPU 执行时的状态。即进程已经具备运行条件，但是 CPU 还没有分配过来，需等待被 CPU 调度到，进入执行态。

执行态： 占用 CPU，在 CPU 上执行。

2. 僵死态

僵死态：进程结束时，其他资源都释放，只留下了 task_struct，等待父进程 wait4 函数处理时的状态。

一个进程如果进入僵死态时，它占用的系统资源都已释放了，只是保留了 task_struct 等待父进程处理。

如果一个进程一直是僵死态，通过 kill 是无法杀掉的，除非将它的父进程杀掉，它才会消失。

系统中有僵尸态的进程对系统资源来说没影响，可能是你写的程序有问题，未正常退出，使得父进程无法处理。

3. 停止态

人为地暂停进程时的状态。

在 linux 中，按 ctrl+z，当前终端下运行的进程就会进入停止态。按 fg 或 bg 恢复该进程的运行。fg 与 bg 的区别是：fg 是前台运行，bg 是后台运行。

有个工具叫 cpulimit，它限制进程的原理就是不断地停止进程，恢复进程，最终达到限制进程资源的效果

4. 睡眠态

睡眠态分浅度睡眠和深度睡眠，区别在于：

- 浅度睡眠：可以被资源和信号唤醒
- 深度睡眠：只能被资源唤醒，比如你挂载一个 NFS，当 NFS 服务器挂了时，你对这个挂载做不了任何操作，比如用 kill 命令发送任何信号都无效，处理的方法是：1.等待 NFS 服务器恢复；2.重启你的服务器。

5. 描述宏

TASK_RUNNING:就绪/运行状态　　TASK_INTERRUPTIBLE:可中断睡眠状态

TASK_UNINTERRUPTIBLE:不可中断睡眠状态

TASK_TRACED:调试态　　TASK_STOPPED:暂停状态

EXIT_ZOMBIE:僵死状态　　EXIT_DEAD:死亡态

六、 进程组、会话、终端

1. 进程组

- 作用：

  对相同类型的进程进行管理。

- 进程组的诞生

  在 shell 里面直接执行一个应用程序，对于大部分进程来说，自己就是进程组的首进程，进程组只有一个进程；

  如果进程调用了 fork 函数。那么当子进程同属一个进程组，父进程为首进程；

  在 shell 中通过管道执行连接起来的应用程序，两个程序同属一个进程组。第一个程序为进程组的首进程；

- 进程组 id

  pgid，由首进程 pid 决定。

2. 会话

- 作用

  管理进程组，多个进程组组成一个会话。

- 会话的诞生

  调用 setsid 函数，新建一个会话，应用程序作为会话的第一个进程，称为会话首进程

  用户在终端正确登录之后，启动 shell 时 linux 系统会创建一个新的会话，shell 进程作为会话首进程

- 会话 id

  SID，会话首进程 id

- 前台进程组

  前台进程是与用户直接交互的进程。在任何时刻，只有一个进程组可以在前台运行。这个进程组可以从终端接收输入，向终端发送输出。如果一个前台进程正在运行，终端会被阻塞，即用户无法在终端进行其他操作，直到这个前台进程完成。

- 后台进程组

  后台进程是在后台运行，不占用用户终端的进程。它们不会阻塞用户终端，用户可以在同一终端启动新的前台或后台进程。后台进程可以向终端发送输出，但通常不能从终端接收输入。

  jobs：查看有哪些后台进程组。

- 前台进程与后台之间如何切换

  使用 ctrl+z 可以将一个正在前台执行的进程挂起，并把它转到后台。

  使用命令 bg 可以将一个被挂起的进程转到后台并继续执行。

  使用命令 fg+job，job 由 jobs 命令查看可以将一个在后台运行或被挂起的进程转到前台。

  当启动一个进程时，在命令行的最后加上 & 符号，便可以使进程在后台运行。

3. 终端

终端就是处理计算机主机输入输出的一套设备，它用来显示主机运算的输出，并且接受主机要求的输入。

一个会话可以有一个控制终端，当控制终端有输入和输出时都会传递给前台进程组。

- 物理终端（依赖物理设备）

  直接连接在主机上的显示器、键盘鼠标统称。

  串口终端

  LCD 终端

- 伪终端（不依赖物理设备）

  ssh 远程终端

  桌面系统启动的终端

- 虚拟终端

  附加在物理终端之上，用软件方式虚拟实现。

  Linux 自带的。

七、 守护进程

Linux 中的一些系统服务进程，没有控制终端、不能和用户交互、不受用户登录注销的影响，一直运行着，是守护进程。

1. 写一个守护进程

- 创建一个子进程，父进程直接退出。通过 fork()函数创建
- 创建一个新的会话，摆脱终端的影响。通过 setsid()函数。
- 改变守护进程的当前工作目录，改为"/"。 为了防止占用可卸载的文件系统。使用 fork ()创建的子进程是继承了父进程的当前工作目录，由于在进程运行中，当前目录所在的文件系统是不能卸载的，这对以后使用会造成很多的麻烦。因此通常的做法是让"/"作为守护进程的当前目录，当然也可以指定其他的别的目录未作为守护进程的工作目录。通过 chdir()函数。
- 重设文件权限掩码，进程从创建它的父进程那里继承了文件创建掩模。它可能修改守护进程所创建的文件的存取权限。为防止这一点，将文件创建掩模清除。通过 umask()函数
- 关闭不需要的文件描述符

  0~2 这三个文件描述符随着进程创建而打开，分别指向标准输入，标准输出，标准错误输出，由于该会话需要脱离终端，因此不需要这三个文件描述符。实际过程中不是直接 close，因为在使用文件描述符的时候从最小可用的单元找，如果关闭 0 号，则 0 号被认为可用，但是 0 号一般不被其他程序使用，因此一般会把这三个文件描述符重定向到 /dev/null 黑洞中。通过 close()函数。

2. 普通进程伪装成守护进程

nohub 命令:

command & : 后台运行，关掉终端之后程序会停止。

nohub command & : 后台运行，关掉终端程序不会停止，还会继续运行。

3. **程序示例**

```
#include <stdio.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>

#define MAXFILE 3
int main()
{
    pid_t pid;
    int fd,len,i,num;
    char *buf = "the demo is running!!!\n";
    len = strlen(buf)+1;

    //创建子进程，销毁父进程
    pid = fork();
    if(pid<0){
        printf("fork error!!!\n");
        exit(1);
    }
    if(pid>0){
        exit(0);
    }
    setsid();               //创建新的会话，摆脱终端的影响
    chdir("/");             //更改目录
    umask(0);               //重设文件权限掩码
    //关闭默认的文件描述符
    for(i = 0;i<MAXFILE;i++){
        close(i);
    }
    //实现守护进程的功能
    while(1){

fd=open("/var/log/demo.log",O_CREAT|O_WRONLY|O_APPEND ,0666);
        write(fd,buf,len);
        close(fd);
        sleep(5);
    }
}
```

八、 ps 命令

ps (Process Status) 命令主要用来显示 Linux 进程信息，进程信息主要包括进程用户、pid、内存、cpu、启动时间、路径、终端等。ps 命令列出的是当前进程的快照，就是执行 ps 命令的那个时刻的那些进程，如果想要动态的显示进程信息，就可以使用 top 命令-性能分析常用命令。

1. 常用的两个选项：aux、axjif

a:显示一个终端所有的进程

u:显示进程的归属用户及内存使用情况 x:显示没有关联控制终端的进程

j:显示进程归属的进程组 id、会话 id、父进程 id。 f:l 以 ascii 形式显示出进程的层次关系

2. ps aux

user:进程是哪个用户产生的

pid:进程的身份证号码

%cpu:表示进程占用了 cpu 计算能力的百分比

%mem:表示进程占用了系统内存的百分比

vsz:进程使用的虚拟内存大小

rss:进程使用的物理内存大小

tty:表示进程关联的终端

stat:表示进程当前状态，主要进程状态有:

R 运行 runnable (on run queue) ，正在运行或在运行队列中等待)。

S 中断 sleeping，休眠中， 受阻， 在等待某个条件的形成或接受到信号。

D 不可中断 uninterruptible sleep (usually IO)，收到信号不唤醒和不可运行，进程必须等待直到有中断发生)

Z 僵死 a defunct ("zombie") process，进程已终止，但进程描述符存在，直到父进程调用 wait4()系统调用后释放。

T 停止 traced or stopped ，进程收到 SIGSTOP，SIGSTP，SIGTIN，SIGTOU 信号后停止运行。

X：死掉的进程。

N:低优先级。

s:进程是会话首进程。

l:多线程(小写 L)。

-+：位于后台。

start:表示进程的启动时间。

time:记录进程的运行时间。

command:表示、进程技行的具体程序。

3. ps axjif

ppid:表示进程的父进程 id

pid:进程的身份证号码

pgid:进程所在进程组的

idsid:进程所在会话的

idtty:表示进程关联的终端

tpgid:值为-1，表示进程为守护进程

stat:表示进程当前状态

uid:启动进程的用户 id

time:记录进程的运行时间

command:表示进程的层次关系

九、 僵尸进程和托孤进程

1. 进程的正常退出步骤：

子进程调用 exit()函数退出

父进程调用 wait()函数为子进程处理其他事情

2. 僵尸进程

子进程退出后，父进程没有调用 wait()函数处理身后事，子进程变成僵尸进程

3. 托孤进程

父进程比子进程先退出，子进程变为孤儿进程，Linux 系统会把子进程托孤给 1 号进程(init 进程)

十、 进程通信

1. 意义

● 数据传输 ● 资源共享 ● 事件通知 ● 进程控制

2. 通信方式

管道(有名管道、无名管道)、消息队列、共享内存、信号量、Socket、FIFO

3. 无名管道

(1) 特点

- 半双工，数据在同一时刻只能在一个方向上流动。
- 数据只能从管道的一端写入，从另一端读出。
- 写入管道中的数据遵循先入先出的规则。
- 管道所传送的数据是无格式的，这要求管道的读出方与写入方必须事先约定好数据的格式，如多少字节算一个消息等。
- 管道不是普通的文件，不属于某个文件系统，其只存在于内存(内核内存中，而不是进程内存中)中。
- 管道在内存中对应一个缓冲区。不同的系统其大小不一定相同。
- 从管道读数据是一次性操作，数据一旦被读走，它就从管道中被抛弃，释放空间以便写更多的数据。
- 管道没有名字，只能在具有血缘关系(公共祖先)的进程之间使用。
- write 和 read 操作可能会阻塞进程
- 所有文件描述符被关闭之后，无名管道被销毁

(2) 使用步骤

- 父进程 pipe 无名管道
- fork 进程
- close 无用端口
- write/read 读写端口
- close 读写端口

(3) pipe 函数

```
/* @param fd,经参数 fd 返回的两个文件描述符
 * fd[0]为读而打开，fd[1]为写而打开
 * fd[1]的输出是 fd[0]的输入
 * @return 若成功，返回 0；若出错，返回-1 并设置 errno
 */
int pipe(int fd[2]);
```

通过 pipe 函数创建的这两个文件描述符 fd[0]和 fd[1]分别构成管道的两端，往 fd[1]写入的数据可以从 fd[0]读出，并且 fd[1]一端只能进行写操作，fd[0]一端只能进行读操作，不能反过来使用。要实现双向数据传输，可以使用两个管道。

默认情况下，这一对文件描述符都是阻塞的。此时，如果我们用 read 系统调用来读取一个空的管道，则 read 将被阻塞，直到管道内有数据可读；如果我们用 write 系统调用往一个满的管道中写数据，则 write 也将被阻塞，直到管道内有足够的空闲空间可用(read 读取数据后管道中将清除读走的数据)。当然，用户可以将 fd[0]和 fd[1]设置为非阻塞的。

单个进程中的管道几乎没有任何用处。对于一个从父进程到子进程的管道，父进程关闭管道的读端 fd[0]，子进程关闭管道的写端 fd[1]

(4) 实例

```c
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_DATA_LEN 256

int main()
{
        pid_t pid;
        int pipe_fd[2];

        int status;
```

```
        char buf[MAX_DATA_LEN];
        const char data[] = "pipe test program!!!";

        int real_write,real_read;
        //初始化
        memset((void*)buf,0,sizeof(buf));
        //创建管道
        if(pipe(pipe_fd) < 0){
                printf("pipe greate error!!!\n");
                exit(1);
        }
        //创建子进程
        if( (pid = fork() ) == 0){
                close(pipe_fd[1]);      //关闭子进程写描述
                if( (real_read =
read(pipe_fd[0],buf,MAX_DATA_LEN) ) > 0){
                        printf("%d bytes read from the pipe
is:'%s'\n",real_read,buf);
                }
                close(pipe_fd[0]);      //关闭子进程读描述符
                exit(0);
        }
        else if(pid > 0){
                close(pipe_fd[0]);      //关闭父进程读描述
                if( (real_write =
write(pipe_fd[1],data,strlen(data)) ) != -1){
                        printf("parent write %d
byte:'%s'\n",real_write,data);
                }
                close(pipe_fd[1]);      //关闭子进程写描述符
                wait(&status);          //搜集子进程退出状态
                exit(0);
        }
}
```

4. 有名管道

(1) 特点

- 有文件名，可以使用 open 函数打开
- 任意进程间数据传输
- write 和 read 操作可能会阻塞进程.
- write 具有"原子性"，原子性简单来说可以理解为，当命名管道剩余的空间不足以将数据全部写入时，就不执行。

(2) 使用步骤

- 第一个进程 mkfifo 有名管道
- open 有名管道,write/read 数据.
- close 有名管道
- 第二个进程 open 有名管道，read/write 数据.
- close 有名管道

(3) mkfifo 函数

```
// 返回值：成功返回 0，出错返回-1
int mkfifo(const char *pathname, mode_t mode);
```

mkfifo 函数需要两个参数，第一个参数（pathname）是将要在文件系统中创建的一个专用文件。第二个参数（mode）用来规定 FIFO 的读写权限。

其中的 mode 参数与 open 函数中的 mode 相同。一旦创建了一个 FIFO，就可以用一般的文件 I/O 函数操作它。

当 open 一个 FIFO 时，是否设置非阻塞标志（O_NONBLOCK）的区别：

若没有指定 O_NONBLOCK（默认），只读 open 要阻塞到某个其他进程为写而打开此 FIFO。类似的，只写 open 要阻塞到某个其他进程为读而打开它。

若指定了 O_NONBLOCK，则只读 open 立即返回。而只写 open 将出错返回 -1 如果没有进程已经为读而打开该 FIFO，其 errno 置 ENXIO。

(4) **程序实例**

write

```
   #include <sys/stat.h>
#include <fcntl.h>
#include <limits.h>
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>

#define MYFIFO "/tmp/myfifo"       //有名管道文件名
#define MAX_BUFFER_SIZE PIPE_BUF   //4096,定义于 limits.h 中

int main(int argc,char * argv[])
{
    int fd;
    char buff[MAX_BUFFER_SIZE];
    int nwrite;
    //判断命令行参数个数
    if(argc <= 1){
        printf("Usage:./fifo_write string\n");
        exit(1);
    }
    //填充命令行第一个参数到 buff
    sscanf(argv[1],"%s",buff);
    //以只写阻塞方式打开 FIFO 管道
    fd = open(MYFIFO,O_WRONLY);
    if(fd == -1){
        printf("open fifo file error!!!\n");
        exit(1);
    }
    if((nwrite = write(fd,buff,MAX_BUFFER_SIZE)) >0){
        printf("write '%s' to FIFO\n",buff);
    }
    close(fd);
    exit(0);
}
```

read

```c
#include <sys/stat.h>
#include <fcntl.h>
#include <limits.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MYFIFO "/tmp/myfifo"       //有名管道文件名
#define MAX_BUFFER_SIZE PIPE_BUF  //4096,定义于 limits.h 中

int main(int argc,char * argv[])
{
    char buff[MAX_BUFFER_SIZE];
    int fd;
    int nread;
    //判断有名管道是否存在，若尚未创建，则以相同的权限创建
    if(access(MYFIFO,F_OK) == -1){
        if((mkfifo(MYFIFO,0666) <0) && (errno != EEXIST)){
            printf("cannot create fifo file!!!\n");
            exit(1);
        }
    }
    //以只读阻塞方式打开有名管道
    fd = open(MYFIFO,O_RDONLY);
    if(fd == -1){
        printf("open fifo file error!!!\n");
        exit(1);
    }
    //循环读取有名管道数据
    while(1){
        memset(buff,0,sizeof(buff));
        if((nread = read(fd,buff,MAX_BUFFER_SIZE)) > 0){
            printf("read '%s' from FIFO\n",buff);
        }
    }
    close(fd);
    exit(0);
}
```

5. 信号

(1) 信号的概念

- 什么是信号

信号是进程之间事件异步通知的一种方式，属于软中断。

信号就是一个消息，告诉进程一个事件，进程受到信号之后会知道怎么处理这个信号。

- 信号的产生

硬件：

执行非法指令；访问非法内存；驱动程序……

软件

控制台:ctrl+c:中断信号;ctrl+\:退出信号;ctrl+z:停止信号。

kill 命令

程序调用 kil()函数

- 处理信号有三种方法：

使用默认方法;忽略此信号;自定义捕捉

默认方法和忽略信号，就是在 handle 数组对应信号数组中填入 SIG_DEL 和 SIG_IGN

- 查看操作系统拥有的信号：kill -l 命令
- 常用信号分析

信号名	信号编号	产生原因	默认处理方式
SIGHUP	1	关闭终端	终止
SIGINT	2	ctrl+c	终止
SIGQUIT	3	ctrl+\	终止+转储
SIGABRT	6	abort()	终止+转储
SIGPE	8	算术错误	终止
SIGKILL	9	kill -9 pid	终止，不可捕获/忽略
SIGUSR1	10	自定义	忽略
SIGSEGV	11	段错误	终止+转储
SIGUSR2	12	自定义	忽略
SIGALRM	14	alarm()	终止
SIGTERM	15	kill pid	终止
SIGCHLD	17	(子)状态变化	忽略
SIGTOP	19	ctrl+z	暂停，不可捕获/忽略

(2) signal、kill、raise 函数

- signal 函数

  函数原型：

  ```
  typedef void (*sighandler_t)(int);
  sighandler_t signal(int signum, sighandler_t handler);
  ```

  参数：

  signum:要设置的信号

  handler: SIG_IGN (忽略); SIG_DFL (默认);
  void (*sighandler_t)(int):自定义

  返回值：

  成功:上一次设置的 handler

  失败:SIG_ERR

- 代码示例

```c
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
//自定义信号处理函数
void signal_handler(int sig){
    printf("\nthis signal number is %d\n",sig);
    if(sig == SIGINT){
        printf("i have get signal\n\n");
        signal(SIGINT,SIG_DFL);        //恢复为默认处理方式
    }
}
int main(void)
{
    printf("this is an alarm test function\n\n");
    signal(SIGINT,signal_handler);

    while(1){
        printf("waiting for SIGINT signal,please enter
\"ctrl+c\"...\n");
        sleep(1);
    }
    exit(0);
}
```

(3) kill 和 raise 函数

- 函数原型
- kill 函数用于向任何进程组或进程发送信号。

```
int kill(pid_t pid,int sig);
//成功执行返回 0  失败返回-1
//失败是 errno 会被设置为以下值:
//EINVAL        指定的信号编号无效
//EPERM         权限不够无法传送信号给指定进程
//ESRCH         参数 pid 指定的进程或进程组不存在
```

- raise 函数常被进程用来向自身发送信号。

```
int raise(int signo);
//成功返回 0, 失败返回-1
```

- 代码示例

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
int main (void)
{
    pid_t pid;
    int ret;
    //创建一个子进程
    if((pid = fork()) < 0){
        printf("fork error!!!\n");
        exit(1);
    }
    if(pid == 0){
        printf("child is waiting for SIGSTOP signal\n\n");
        raise(SIGSTOP);
        printf("child won't run here forever\n");
        exit(0);
    }
    else{
        sleep(3);
        //发送 SIGKILL 信号杀死子进程
        if((ret = kill(pid,SIGKILL)) == 0){
            printf("parent kill %d\n\n",pid);
```

```
        }
        wait(NULL);     //阻塞直到子进程退出
        printf("parent exit\n");
        exit(0);
    }
}
```

(4) 信号集处理函数

- 概念

  多个信号组成的一个集合称为信号集，其系统数据类型为 sigset_t 。

  信号如果被屏蔽,则记录在来处理信号集中

  非实时信号(1~31)，不排队，只留一个。

  实时信号(34~64)，排队，保留全部。

- 自定义信号集相关函数

  ```
  int sigemptyset(sigset_t *set);
  ```

  - 功能：将信号集中的所有的标志位置为 0
  - 参数：set,传出参数，需要操作的信号集
  - 返回值：成功返回 0， 失败返回-1

  ```
  int sigfillset(sigset_t *set);
  ```

  - 功能：将信号集中的所有的标志位置为 1
  - 参数：set,传出参数，需要操作的信号集
  - 返回值：成功返回 0， 失败返回-1

  ```
  int sigaddset(sigset_t *set, int signum);
  ```

  - 功能：设置信号集中的某一个信号对应的标志位为 1，表示阻塞这个信号
  - 参数：
    - set：传出参数，需要操作的信号集
    - signum：需要设置阻塞的那个信号
    - 返回值：成功返回 0， 失败返回-1

int sigdelset(sigset_t *set, int signum);

- 功能：设置信号集中的某一个信号对应的标志位为 0，表示不阻塞这个信号
- 参数：
  - set：传出参数，需要操作的信号集
  - signum：需要设置不阻塞的那个信号
- 返回值：成功返回 0， 失败返回-1

int sigismember(const sigset_t *set, int signum);

- 功能：判断某个信号是否阻塞
- 参数：
  - set：需要操作的信号集
  - signum：需要判断的那个信号
- 返回值：

  1 ： signum 被阻塞

  0 ： signum 不阻塞

  -1 ： 失败

int sigpromask(int how,const sigset_t *set,sigset_t *oset);

作用：用设置好的信号集区修改信号屏蔽集

如果 set 为空，则 how 没有意义，但此时调用该函数，如果 oset 不为空，则把当前信号屏蔽字保存到 oset 中。

how 的不同取值及操作如下所示：

SIG_BLOCK 把参数 set 中的信号添加到信号屏蔽字中，也就是将两者并集作为新的信号屏蔽字

SIG_UNBLOCK 从信号屏蔽字中删除参数 set 中的信号，也就是将两者差集作为新的信号屏蔽字

SIG_SETMASK 把信号屏蔽字设置为参数 set，并将原信号屏蔽字保存到 oset 中（如果非空）

如果 sigpromask 成功完成返回 0，如果 how 取值无效返回-1，并设置 errno 为 EINVAL。

- 代码示例

```c
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

void myFunc(int signo){
    sigset_t set;
    sigemptyset(&set);                  //所有标志置 0
    sigaddset(&set,SIGINT);             //将 SIGINT 置 1
    sigprocmask(SIG_UNBLOCK,&set,NULL); //解除屏蔽
    printf("hello\n");
    sleep(5);
    printf("world\n");
}
int main (void)
{
    signal(SIGINT,myFunc);
    while(1);
    return 0;
}
```

6. 消息队列

(1) 特点

- 消息队列的本质其实是一个内核提供的链表，内核基于这个链表，实现了一个数据结构；
- 向消息队列中写数据，实际上是向这个数据结构中插入一个新结点；从消息队列汇总读数据，实际上是从这个数据结构中删除一个结点；
- 消息队列提供从一个进程向另外一个进程发送一块数据的方法；
- 消息队列也有管道一样的不足，就是每个数据块的最大长度是有上限的，系统上全体队列的最大总长度也有一个上限。
- 独立于进程
- 没有文件名和文件描述符
- IPC 对象具有 ID 和唯一的 key

(2) 用法

- 定义一个唯一key (ftok)
- 构造消息对象(msgget)
- 发送特定类型消息(msgsnd)
- 接受特定类型消息(msgrcv)
- 删除消息队列 (msgctl)

(3) key 值作用

进程间通信 (IPC),

有两个东西可以标识一个 IPC 结构：标识符(ID)和键(key)。

键值 (ID)

ID 是 IPC 结构的内部名，用来确保使用同一个通讯通道(比如说这个通讯通道就是消息队列)。内部即在进程内部使用，这样的标识方法是不能支持进程间通信的。

标识符(key)

key 就是 IPC 结构的外部名。当多个进程，针对同一个 key 调用 get 函数(msgget 等)，这些进程得到的 ID 其实是标识了同一个 IPC 结构。多个进程间就可以通过这个 IPC 结构通信。

(4) 函数用法

- ftok () 函数

  key_t ftok(const char *pathname, int proj_id);

  //作用：生成唯一的 key 值。

  //参数：

  pathname：代表一个存在的文件路径名

  proj_id:用于区分不同的共享内存、消息队列或信号量的一个整数，通常设置为非 0 值

  //返回值：key 值。

- msgget()函数

  int msgget(key_t key, int msgflg);

  //作用：构造消息对象

  //参数：

  key：表示待创建的消息队列在内存中的唯一标识符

  msgflg：表示创建消息队列的创建方式

  IPC_CREAT:如果消息队列不存在，则创建。

  mode:访问权限

  //返回值：创建成功返回消息队列的标识号，失败返回 -1

- msgsnd()函数

  int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);

  //作用：发送消息到消息队列

  //参数：

  msqid：指定发送消息队列的 ID

  msgp:指向存储待发送消息内容的内存地址，用户可设计自己的消息结构

  msgsz：指定长度，仅记载数据的长度

  msgflg:控制消息发送的方式,IPC_NOWAIT:非阻塞发送，0:阻塞发送（消息队列没有足够空间是不发送）。

  //返回值：成功 0，失败-1。

- msgrcv()函数

  ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);

  //作用:接收消息队列

  //参数:

  msqid: 消息队列的 ID

  msgp: 指向接收消息的缓冲区的指针

  msgsz: 接收消息缓冲区的字节数

  msgtyp: 指定要接收的消息类型

  msgflg: 控制消息接收的方式:

  IPC_NOWAIT:非阻塞读取 (没有消息则返回进程)

  MSG_NOERROR:截断消息(接收长度小于指定长度不报错)

  0：阻塞读取 (没有消息则阻塞进程等待)

  //返回值:

  函数的返回值为实际接收到的消息数据的长度,-1 表示失败。

- msgctl()函数

  int msgctl(int msqid, int cmd, struct msqid_ds *buf);

  //作用: 设置或获取消息队列的相关属性，也可以删除消息队列。

  //参数:

  msqid: 消息队列的 ID

  cmd: 控制命令，用于指定要执行的操作

  IPC_STAT:获取消息队列的属性信息

  IPC_SET:设置消息队列的属性

  IPC_RMID:删除消息队列

  buf: 指向 msqid_ds 结构体的指针，用于存储消息队列的状态信息。

(5) 代码实例

- send:

```
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>

#define BUFFER_SIZE 512

struct message{
    long msg_type;
    char msg_text[BUFFER_SIZE];
};

int main(void)
{
    int qid;
    key_t key;
    struct message msg;
    //产生 key
    if((key == ftok("/tmp",11)) == -1){
        printf("ftok");
        exit(1);
    }
    //创建消息队列
    if((qid == msgget(key,IPC_CREAT|0666)) == -1){
        printf("msgget");
        exit(1);
    }
    printf("open queue %d\n",qid);
    while(1){
        printf("enter some message to the queue:");
        //填充数据
        if((fgets(msg.msg_text,BUFFER_SIZE,stdin)) == NULL){
            puts("no message");
            exit(1);
        }
        msg.msg_type = getpid();    //当前进程的 PID 号作为消息类型
```

```
        //添加数据到消息队列
        if((msgsnd(qid,&msg,strlen(msg.msg_text),0)) < 0){
            printf("message posted");
            exit(1);
        }
        //如果接受消息是 quit（退出），则退出 while 循环
        if(strncmp(msg.msg_text,"quit",4) == 0){
            break;
        }
    }
    exit(0);
}
```

- receive

```c
#include <unistd.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#define BUFFER_SIZE 512
struct message{
    long msg_type;
    char msg_text[BUFFER_SIZE];
};
int main(void)
{
    int qid;
    key_t key;
    struct message msg;
    //产生key
    if((key == ftok("/tmp",11)) == -1){
        printf("ftok");
        exit(1);
    }
    //创建消息队列
    if((qid == msgget(key,IPC_CREAT|0666)) == -1){
        printf("msgget");
        exit(1);
    }
    printf("open queue %d\n",qid);
    do{
        memset(msg.msg_text,0,BUFFER_SIZE);     //清空
        if(msgrcv(qid,(void*)&msg,BUFFER_SIZE,0,0) < 0){
            printf("msgrcv");
            exit(1);
        }
        printf("from process %ld:%s",msg.msg_type,msg.msg_text);
    }while(strncmp(msg.msg_text,"quit",4));
    //删除队列
    if((msgctl(qid,IPC_RMID,NULL))<0){
        printf("msgctl");
        exit(1);
    }
    exit(0);
}
```

7. 信号量

(1) 本质

计数器

(2) 作用

保护共享资源：

互斥：是指散步在不同任务之间的若干程序片断，当某个任务运行其中一个程序片段时，其它任务就不能运行它们之中的任一程序片段，只能等到该任务运行完这个程序片段后才可以运行，最基本的场景就是对资源的同时写，为了保持资源的一致性，往往需要进行互斥访问。

同步：是指散步在不同任务之间的若干程序片断，它们的运行必须严格按照规定的某种先后次序来运行，这种先后次序依赖于要完成的特定的任务，最基本的场景就是任务之间的依赖，比如 A 任务的运行依赖于 B 任务产生的数据。

同步是一种更为复杂的互斥，而互斥是一种特殊的同步。也就是说互斥是两个任务之间不可以同时运行，他们会相互排斥，必须等待一个线程运行完毕，另一个才能运行，而同步也是不能同时运行，但他是必须要安照某种次序来运行相应的线程（也是一种互斥）！因此互斥具有唯一性和排它性，但互斥并不限制任务的运行顺序，即任务是无序的。而同步的任务之间则有顺序关系。

(3) 用法

定义一个唯一key (ftok)

构造一个信号量(semget)

初始化信号量(semctl SETVA)

对信号量进行 P/V 操作(semop)

删除信号量 (semctl RMID)

(4) 相关函数的用法

- semget()

  int semget(key_t key, int nsems, int semflag);

  //作用：创建一个新信号量或获取一个已有信号量的 ID。

  //参数：

  key:信号星键值

  nsems:信号数量

  semflg:

  IPC_CREATE:信号量不存在则创建

  made:信号量的权限

  //返回值：成功：信号量 ID；失败：-1。

- semctl()

  int semctl(int semid, int semnum, int cmd, [union arg]);

  //作用：获取或设置信号量相关属性

  //参数：

  semid:信号量 ID

  semnum:信号量编号

  cmd:

  IPC STAT:获取信号星的属性信息

  IPC_SET:设置信号量的属性

  IPC_RMID:删除信号量

  IPC_SETVAL:设置信号量的值

  arg: ：可选参数，是一个结构，它至少包含以下几个成员：

  union semun{

  int val;

  struct semid_ds *buf;

  unsigned short *array;

  }

  //返回值：成功：由 cmd 类型决定。失败：-1。

- semop()

  int semop(int semid, struct sembuf *sops, size_t nsops);

  //作用：改变信号量的值

  //参数：

  semid:信号量 ID

  sops:信号量操作结构体数组

  struct sembuf

  {

  short sem_num;　　//信号量编号

  short sem_op;　　//信专量 P/V 操作

  short sem flg;　　//信号量行为，通常为 SEM_UNDO,使操作系统跟踪信号，并在进程没有释放该信号量而终止时，操作系统释放信号量。

  }

  nsops:信号量数量

  //返回值：成功：0；失败：-1。

(5) 代码示例

- sem.c

```c
#include <sys/ipc.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/types.h>
#include "sem.h"

union semun{
    int val;
    struct semid_ds *buf;
};

//初始化信号量
int init_sem(int sem_id,int init_value){
    union semun sem_union;
    sem_union.val = init_value;
    if(semctl(sem_id,0,SETVAL,sem_union) == -1){
        printf("Initialize semaphore");
        return -1;
    }
    return 0;
}
//删除信号量
int del_sem(int sem_id){
    union semun sem_union;
    if(semctl(sem_id,0,IPC_RMID,sem_union) == -1){
        perror("Delete semaphore");
        return -1;
    }
    return 0;
}
```

```c
//P 操作(减操作)
int sem_p(int sem_id){
    struct sembuf sops;
    sops.sem_num = 0;     //单个信号的编号应该是 0
    sops.sem_op = -1;     //表示 P 操作
    sops.sem_flg = SEM_UNDO;//系统自动释放会在系统中残留的信号量
    if(semop(sem_id,&sops,1) == -1){
        perror("P operation");
        return -1;
    }
    return 0;
}
//V 操作
int sem_v(int sem_id){
    struct sembuf sops;
    sops.sem_num = 0;         //单个信号量的编号应该为 0
    sops.sem_op = 1;          //表示 V 操作
    sops.sem_flg = SEM_UNDO;//系统自动释放会在系统中残留的信号量
        if(semop(sem_id,&sops,1) == -1){
                perror("V operation");
                return -1;
        }
        return 0;
}
```

- test.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <unistd.h>
#include "sem.h"

#define DELAY_TIME 3

int main(void)
{
    pid_t result;
    int sem_id;
    //初始化
    sem_id = semget((key_t)6666, 1, 0666 | IPC_CREAT);
    init_sem(sem_id,0);

    result = fork();
    if(result == -1){
        perror("fork\n");
    }
    else if(result == 0){
        printf("child process will wait for some seconds...\n");
        sleep(DELAY_TIME);
        printf("the child process is running...\r\n");
        sem_v(sem_id);
    }
    else{
        sem_p(sem_id);//初始为0,会阻塞运行，直到子进程加一后运行
        printf("the father process is running...\r\n");
        sem_v(sem_id);
        del_sem(sem_id);
    }
    exit(0);

}
```

8. 共享内存

(1) 作用

高效率传输大量数据。

(2) 用法

定义一个唯一 key (ftok)

构造一个共享内存对象(shmget)

共享内存映射(shmat)

解除共享内存映射(shmdt)

删除共享内存(shmctl RMID)

(3) 相关函数

- shmget()

  int shmget(key_t key, int size, int shmflg)

  //作用：获取共享内存对象的 ID

  //参数

  key:共享对象键值

  size:共享对象内存大小

  shmflg:IPC_CREATE:共享内存不存在则创建；

  mode:共享内存的权限。

  //返回值：成功：共享内存 ID，失败：-1。

- shmat()

  int shmat(int shmid, const void *shmaddr, int shmflg)

  //作用：映射共享内存

  //参数：

  shmid:共享内存 ID

  shmaddr:映射地址，NULL 为自动分配。

  shmflg:SHM_RAONLY:只读方式映射；0：可读可写。

  //返回值：成功：共享内存首地址；失败：-1

- shmdt()

  int shmdt(const void *shmaddr)

  //作用：解除共享内存映射。

  //参数：shmaddr:映射地址

  //返回值：成功：0；失败：-1。

- shmctl()

  int shmctl(int shmid, int cmd, struct shmid_ds *buf)

  //作用：获取或设置共享内存的相关属性。

  //参数：

  shmid:共享内存 ID。

  cmd:

  IPC_STAT:获取共享内存的属性信息。

  IPC_SET:设置共享内存的属性。

  IPC_RMID:删除共享内存

  buf:属性缓冲区。

  //返回值：成功：由 cmd 类型决定；失败：-1。

(4) 代码示例

- test.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <unistd.h>
#include "sem.h"

#define DELAY_TIME 3

int main(void)
{
    pid_t result;
    int sem_id;
    int shm_id;
    char* addr;
    //创建信号量
    sem_id = semget((key_t)6666, 1, 0666|IPC_CREAT);
    //创建共享内存对象
    shm_id = shmget((key_t)7777, 1024, 0666|IPC_CREAT);
    init_sem(sem_id,0);    //初始化信号量

    result = fork();
    if(result == -1){
        perror("fork\n");
    }
    else if(result == 0){
        printf("child process will wait for some seconds...\n");
        sleep(DELAY_TIME);
        addr = shmat(shm_id, NULL, 0);    //映射共享内存地址
```

```
        if(addr == (void*)-1){
            printf("shmat error!");
            exit(-1);
        }
        memcpy(addr,"helloworld",11);    //设置共享内存内容
        printf("the child process is running...\r\n");
        sem_v(sem_id);
    }
    else{
        sem_p(sem_id);    //初始为0,会阻塞运行，子进程加一后运行
        printf("the father process is running...\r\n");
        addr = shmat(shm_id, NULL, 0);    //映射共享内存地址
        if(addr == (void*)-1){
                        printf("shmat error!");
                        exit(-1);
                }
        printf("shared memory string:%s\r\n",addr);
        shmdt(addr);    //解除共享内存映射
        shmctl(shm_id,IPC_RMID,NULL);    //删除共享内存
        sem_v(sem_id);
        del_sem(sem_id);
    }
    exit(0);
}
```