

第三天进程与信号

一、回顾知识点

1.1 系统调用I/O函数

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

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

#include <stdio.h>
int remove(const char *pathname); 【库函数】
```

1.2 fcntl函数

```
// 修改标准的输入设备的标识为非阻塞
int flags = fcntl(STDIN_FILENO, F_GETFL);
flags |= O_NONBLOCK;
fcntl(STDIN_FILENO, F_SETFL, flags);
```

取消非阻塞

```
int flags = fcntl(STDIN_FILENO, F_GETFL);
flags ^= O_NONBLOCK;
fcntl(STDIN_FILENO, F_SETFL, flags);
```

获取文件的类型（文件、目录）

```
struct stat info;

stat(char *path, &info);

info.st_mode & S_IFDIR 是否为目录的验证
info.st_mode & S_IFREG 是否为文件的验证
```

打开目录和读取目录：

```
dir *opendir(char *path)
dirent *readdir(dir *)
```

1.3 进程的定义

进程拥有自己独立的处理环境和系统资源（处理器、存储器、I/O 设备、数据、程序）

进程的三种状态：

就绪态： 进程已经具备执行的一切条件，正在等待分配 CPU 的处理时间。

执行态： 该进程正在占用 CPU 运行。

等待态： 进程因不具备某些执行条件而暂时无法继续执行的状态。

进程控制块（PCB）：

OS 是根据 PCB 来对并发执行的进程进行控制和管理的。

系统在创建一个进程的时候会开辟一段内存空间存放与此进程相关的 PCB 数据结构。

PCB 是操作系统中最重要的记录型数据结构，包含进程号、返回状态、运行时间等。

获取进程号：

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void) 获取本进程号(PID)
pid_t getppid(void) 获取父进程号(PPID)
pid_t getpgid(pid_t pid) 获取进程组号(PGID)，参数为0时返回当前PGID，否则返回参数指定的进程的PGID
```

1.4 进程控制函数

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void); 创建独立空间的子进程

#include <unistd.h>
unsigned int sleep(unsigned int sec); 挂起或休眠sec秒

#include <sys/types.h>
#include <sys/wait.h>

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

获取子进程的返回状态：

`WIFEXITED(status)` 如果子进程是正常终止的，取出的字段值非零
`WEXITSTATUS(status)` 返回子进程的退出状态，退出状态保存在 `status` 变量的 8~16 位（低位的第2个字节）。在用此宏前应先用宏 `WIFEXITED` 判断子进程是否正常退出，正常退出才可以使用此宏。

二、进程控制

2.1 ps -aux 查看进程状态【扩展】

-a 显示终端上的所有进程，包含其他用户的进程
-u 显示进程的详细状态
-x 显示没有控制终端的进程
-w 显示加宽，以显示更多的信息
-j 显示父进程号、进程组号等信息

stat的含义：

D 不可中断（`Uninterruptible`）
R 正在运行，或在队列中的进程
S 处于休眠状态
T 停止或被追踪
Z 僵尸进程
W 进入内存交换（内核2.6+ 无效）
X 死掉的进程
< 高优先级
N 低优先级
s 包含子进程
+ 位于前台的进程组

存在以下特殊进程的概念：

僵尸进程：

子进程退出，父进程没有回收子进程资源，子进程为僵尸进程。（有危害）
子进程的PID被占用，系统的PID是有数量限制。

孤儿进程：

父进程先结束，子进程为孤儿进程。（无害的）
孤儿进程 被1号进程接管（当孤儿进程结束时，1号进程负责回收其资源）

守护进程(后台进程)： 是脱离终端的孤儿进程。（关闭0/1/2标准的文件描述，设置当前的进程新会话 `setsid()`、改变当前终端工作目录 `chdir()`）

如：

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

```

int main(int argc, char const *argv[])
{
    int pid = fork();
    if (pid == 0)
    {
        printf("子进程  %d立即结束\n", getpid());
        return 10;
    }
    else if (pid > 0)
    {
        printf("父进程  %d 运行中\n", getpid());
        // 没有回收资源 (wait/waitpid)
        while (1)
            ;
    }

    return 0;
}

```

```

disen@qfxa:~/code3/day03$ ./a.out
父进程  33049 运行中
子进程  33050立即结束

```

```

disen@qfxa:~/code3/day03$ ps ja
  PPID    PID    PGID    SID TTY        TPGID  STAT   UID    TIME  COMMAND
    1      959     959     959 tty7        959    Ss+    0       3:00  /usr/
    1     1918    1918    1918 tty1        1918    Ss+    0       0:00  /sbin
15288   15295   15295   15295 pts/4       15295    Ss+   1000    0:00  bash
31746   31781   31781   31781 pts/22      33049    Ss    1000    0:00  /bin/
31746   32763   32763   32763 pts/24      33069    Ss    1000    0:00  /bin/
31781   33049   33049   31781 pts/22      33049    R+    1000    0:23  ./a.o
33049   33050   33049   31781 pts/22      33049    Z+    1000    0:00  [a.ou
32763   33069   33069   32763 pts/24      33069    R+    1000    0:00  ps ja

```

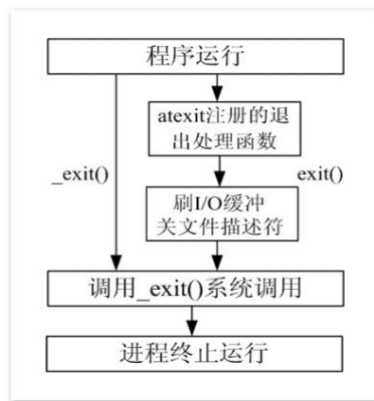
2.2 进程的终止

```

#include <stdlib.h>
void exit(int status);    status: 返回给父进程的参数(低 8 位有效)    【库函数】

#include <unistd.h>
void _exit(int status);    同exit功能, 【系统调用】

```



如1:

```

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

int main(int argc, char const *argv[])
{
    int pid = fork();
    if (pid == 0)
    {
        printf("子进程  %d立即结束\n", getpid());
        sleep(2);
        // _exit(-10); 不要返回负数
        _exit(10);
    }
    else if (pid > 0)
    {
        printf("父进程  %d 运行中\n", getpid());
        // 回收资源 (wait/waitpid)
        int status;
        wait(&status);
        // 0 1111 1111 00000000
        printf("%d 子进程返回的状态值: %d\n", pid, (status & 0xff00) >> 8);
        if (WIFEXITED(status))
        {
            printf("%d 子进程返回的状态值: %d\n", pid, WEXITSTATUS(status));
        }
    }

    return 0;
}

```

```

父进程  33562 运行中
子进程  33563立即结束
33563 子进程返回的状态值: 10
33563 子进程返回的状态值: 10

```

如2: 创建多个子进程, 并等待所有子进程结束, 父进程再退出。

```

#include <sys/types.h>

```

```

#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    // 创建2个子进程
    int i = 0;
    for (i = 0; i < 2; i++)
    {
        pid_t pid = fork();
        if (pid == 0)
        {
            if (i == 0)
                printf("%d ABC\n", getpid());
            else
                printf("%d DEF\n", getpid());
            _exit(0);
        }
    }

    while (1)
    {
        // 等待所有子进程结束
        int pid = waitpid(-1, NULL, WUNTRACED);
        if (pid == -1)
            break;
        printf("子进程 %d over\n", pid);
    }
    printf("主进程--over\n");
    return 0;
}

```

```

33819 DEF
子进程 33819 over
33818 ABC
子进程 33818 over
主进程--over

```

2.3 进程退出清理

进程在退出前可以用 `atexit()` 函数注册退出处理函数。

```

#include <stdlib.h>
int atexit(void (*function)(void));

```

注册进程正常结束前调用的函数，进程退出执行注册函数。

function：进程结束前，调用函数的入口地址。

一个进程中可以多次调用 `atexit` 函数注册清理函数，正常结束前调用函数的顺序和注册时的顺序相反。

如：

```

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

void clearAll1()
{
    printf("%d 清退资源1...\n", getpid());
}
void clearAll2()
{
    printf("%d 清退资源2...\n", getpid());
}

int main(int argc, char const *argv[])
{
    atexit(clearAll1); // 主进程中注册
    atexit(clearAll2); // 主进程中注册
    int pid = fork();
    if (pid == 0)
    {
        sleep(5);
        printf("%d 子进程结束\n", getpid());
        _exit(1);
    }
    else if (pid > 0)
    {
        wait(NULL);
        printf("主进程 %d 结束\n", getpid());
    }
    return 0;
}

```

```

● disen@qfxa:~/code3/day03$ ./a.out
34333 子进程结束
主进程 34332 结束
34332 清退资源2...
34332 清退资源1...

```

2.4 vfork 函数

```
pid_t vfork(void)
```

vfork 函数和 fork 函数一样都是在已有的进程中创建一个新的进程，但它们创建的子进程是有区别的。创建子进程成功，则在子进程中返回 0，父进程中返回子进程 ID。出错则返回-1。

fork 和 vfork 函数的区别：

- 1) `vfork` 保证子进程先运行，在它调用 `exec` 或 `exit` 之后，父进程才可能被调度运行。
- 2) `vfork` 和 `fork` 一样都创建一个子进程，但它并不将父进程的地址空间完全复制到子进程中，因为子进程会立即调用 `exec`(或 `exit`)，于是也就不访问该地址空间。
相反，在子进程中调用 `exec` 或 `exit` 之前，它在父进程的地址空间中运行，在 `exec` 之后子进程会有自己的进程空间。



如:

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

int main(int argc, char const *argv[])
{
    int num = 10;
    int pid = vfork();
    if (pid == 0)
    {
        // 子进程
        num += 10;
        printf("%d子进程 num=%d\n", getpid(), num);
        _exit(0);
    }
    else if (pid > 0)
    {
        // 父进程
        printf("%d父进程 num=%d\n", getpid(), num);
    }
    return 0;
}
```

```
● disen@qfxa:~/code3/day03$ ./a.out
34504子进程 num=20
34503父进程 num=20
```

如2: `vfork()`创建的子进程，只有子进程结束了，父进程才会执行

```
#include <sys/types.h>
```



```

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

int main(int argc, char const *argv[])
{
    int pid = vfork();
    if (pid == 0)
    {
        // 子进程
        for (int i = 0; i < 5; i++)
        {
            printf("%d 子进程 i=%d\n", getpid(), i);
            sleep(1);
        }
        _exit(0); // 子进程结束之后，才会执行父进程
    }
    else if (pid > 0)
    {
        // 父进程
        while (1)
        {
            printf("父进程 %d\n", getpid());
            sleep(1);
        }
    }
    return 0;
}

```

```

34612 子进程 i=0
34612 子进程 i=1
34612 子进程 i=2
34612 子进程 i=3
34612 子进程 i=4
父进程 34611
父进程 34611

```

如3: vfork创建多个子进程，子进程按创建的先后顺序依次执行

```

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

int main(int argc, char const *argv[])
{
    // 创建3个子进程
    for (int i = 0; i < 3; i++)
    {
        int pid = vfork();
    }
}

```

```

    if (pid == 0)
    {
        for (int j = 0; j < 2; j++)
        {
            printf("%d 子进程 j=%d\n", getpid(), j);
            sleep(1);
        }
        _exit(0);
    }
}

// 父进程
printf("父进程 %d\n", getpid());
return 0;
}

```

```

disen@qfxa:~/code3/day03$ gcc core.c -o a.out
disen@qfxa:~/code3/day03$ ./a.out
34914 子进程 j=0
34914 子进程 j=1
34928 子进程 j=0
34928 子进程 j=1
34967 子进程 j=0
34967 子进程 j=1
父进程 34913

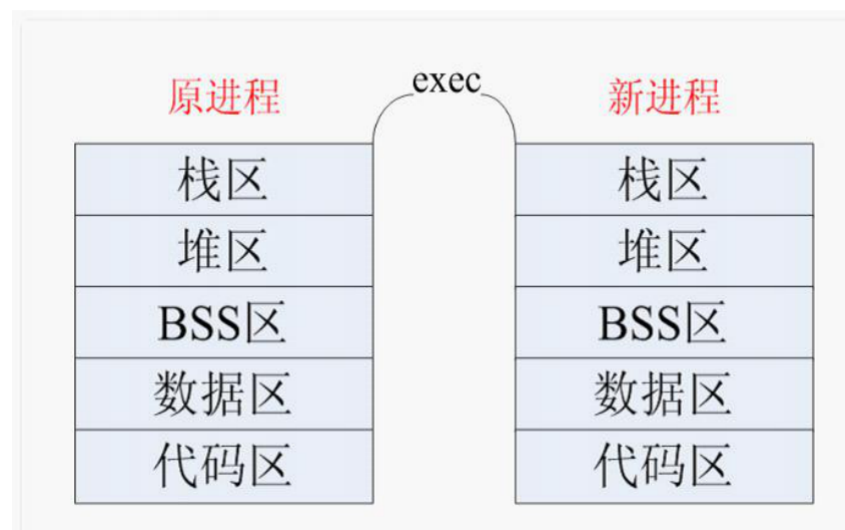
```

2.5 进程的替换

进程替换由exec家族函数完成：

- 1、exec 函数族提供了六种在进程中启动另一个程序的方法。
- 2、exec 函数族可以根据指定的文件名或目录名找到可执行文件。
- 3、调用 exec 函数的进程并不创建新的进程，故调用 exec 前后，进程的进程号并不会改变，其执行的程序完全由新的程序替换，而新程序则从其 main 函数开始执行。

exec 函数族取代调用进程的数据段、代码段和堆栈段。



exec相关函数：

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ..., NULL);
int execlp(const char *filename, const char *arg0, ..., NULL);
int execl_e(const char *pathname, const char *arg0, ..., NULL, char *const envp[]);
int execv(const char *pathname, char *const argv[]);
int execvp(const char *filename, char *const argv[]);

int execve(const char *pathname, char *const argv[], char *const envp[]);
```

六个 exec 函数中只有 execve 是真正意义的系统调用(内核提供的接口)，其它函数都是在此基础上经过封装的库函数。

l(list): 参数地址列表，以空指针结尾。

v(vector): 存有各参数地址的指针数组的地址。使用时先构造一个指针数组，指针数组存各参数的地址，然后将该指针数组地址作为函数的参数。

p(path): 按 PATH 环境变量指定的目录搜索可执行文件。以 p 结尾的 exec 函数取文件名做为参数。当指定 filename 作为参数时，若 filename 中包含 /，则将其视为路径名，并直接到指定的路径中执行程序。

e(environment): 存有环境变量字符串地址的指针数组的地址。execl_e 和 execve 改变的是 exec 启动的程序的环境变量（新的环境变量完全由 environment 指定），其他四个函数启动的程序则使用默认系统环境变量。

【注意】

exec 函数族与一般的函数不同，exec 函数族中的函数执行成功后不会返回。只有调用失败了，它们才会返回-1。失败后从原程序的调用点接着往下执行。

在平时的编程中，如果用到了 exec 函数族，一定要记得加错误判断语句。

如1： 执行ls命令

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char const *argv[])
{
    int ret = execl("/bin/ls", "ls", "-l", "./", NULL);
    if (ret == -1)
    {
        perror("ls");
    }
    return 0;
}
```

如2:

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char const *argv[])
{
    printf("当前进程号: %d\n", getpid());
}
```

```

int ret = execl("./t6", "t6", NULL);
if (ret == -1)
{
    perror("ls");
}
return 0;
}

```

```

● disen@qfxa:~/code3/day03$ ./a.out
当前进程号: 35745
35746 子进程 j=0
35746 子进程 j=1
35767 子进程 j=0
35767 子进程 j=1
35779 子进程 j=0
35779 子进程 j=1
父进程 35745

```

如3: execlp

```

#include <stdio.h>
#include <unistd.h>

int main(int argc, char const *argv[])
{
    printf("当前进程号: %d\n", getpid());
    int ret = execlp("ls", "ls", "-l", "../", NULL);
    if (ret == -1)
    {
        perror("ls");
    }
    return 0;
}

```

```

● disen@qfxa:~/code3/day03$ ./a.out
当前进程号: 35931
total 8
drwxrwxr-x 4 disen disen 4096 8月 15 18:45 day02
drwxrwxr-x 2 disen disen 4096 8月 16 11:27 day03

```

如4:

```

#include <stdio.h>
#include <unistd.h>

int main(int argc, char const *argv[])
{
    printf("当前进程号: %d\n", getpid());
    int ret = execlp("t6", "t6", NULL);
    if (ret == -1)
    {
        perror("t6");
    }
    return 0;
}

```

```
}
```

```
● disen@qfxa:~/code3/day03$ export PATH=./:$PATH
⊗ disen@qfxa:~/code3/day03$ t6
36484 子进程 j=0
36484 子进程 j=1
^C
● disen@qfxa:~/code3/day03$ gcc t7.c
⊗ disen@qfxa:~/code3/day03$ ./a.out
当前进程号: 36552
36553 子进程 j=0
36553 子进程 j=1
^C
```

如: t8.c

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char const *argv[])
{
    // 读取环境变量 PYTHON_HOME, JAVA_HOME
    char *py_home = getenv("PYTHON_HOME");
    char *java_home = getenv("JAVA_HOME");
    printf("py: %s, java: %s\n", py_home, java_home);
    return 0;
}
```

t9.c

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char const *argv[])
{
    // 【注意】 设置环境变量的数组, 最后的位置必须存在一个NULL, 表示环境变量设置结束
    char *env[] = {"JAVA_HOME=/opt/java", "PYTHON_HOME=/opt/py3", NULL};
    int ret = execl("./t8", "t8", NULL, env);
    if (ret == -1)
    {
        perror("./t8");
    }
    return 0;
}
```

```
● disen@qfxa:~/code3/day03$ ./a.out
py: /opt/py3, java: /opt/java
```

如: 使用execve()实现

```

#include <stdio.h>
#include <unistd.h>

int main(int argc, char const *argv[])
{
    // 【注意】 设置环境变量的数组，最后的位置必须存在一个NULL，表示环境变量设置结束
    char *env[] = {"JAVA_HOME=/opt/java", "PYTHON_HOME=/opt/py3", NULL};
    char *args[] = {"t8", NULL};
    int ret = execve("./t8", args, env);
    if (ret == -1)
    {
        perror("./t8");
    }
    return 0;
}

```

效果如上图。

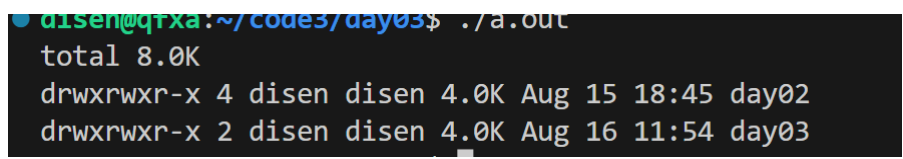
如：

```

#include <stdio.h>
#include <unistd.h>

int main(int argc, char const *argv[])
{
    // 【注意】 设置环境变量的数组，最后的位置必须存在一个NULL，表示环境变量设置结束
    char *env[] = {NULL};
    char *args[] = {"ls", "-lsh", "../", NULL};
    int ret = execve("/bin/ls", args, env);
    if (ret == -1)
    {
        perror("/bin/ls");
    }
    return 0;
}

```



```

disen@qtxa:~/code3/day03$ ./a.out
total 8.0K
drwxrwxr-x 4 disen disen 4.0K Aug 15 18:45 day02
drwxrwxr-x 2 disen disen 4.0K Aug 16 11:54 day03

```

【小结】

一个进程调用 `exec` 后，除了进程 ID，进程还保留了下列特征不变：

- 父进程号
- 进程组号
- 控制终端
- 根目录
- 当前工作目录
- 进程信号屏蔽集
- 未处理信号
- ...

2.6 system 函数

system 会调用 fork 函数产生子进程，子进程调用 exec 启动/bin/sh -c string 来执行参数 string 字符串所代表的命令，此命令执行完后返回原调用进程。

```
#include <stdlib.h>
int system(const char *command);
```

参数command: 要执行的命令的字符串

返回值:

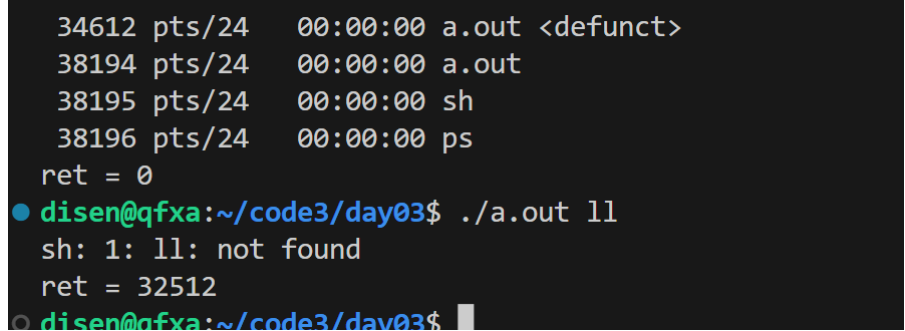
如果 command 为 NULL，则 system() 函数返回非 0，一般为 1。
如果 system() 在调用/bin/sh 时失败则返回 127，其它失败原因返回-1。

【注意】 system 调用成功后会返回执行 shell 命令后的返回值。其返回值可能为 1、127 也可能为-1，故最好应再检查 errno 来确认执行成功

如:

```
#include <stdio.h>
#include <stdlib.h>

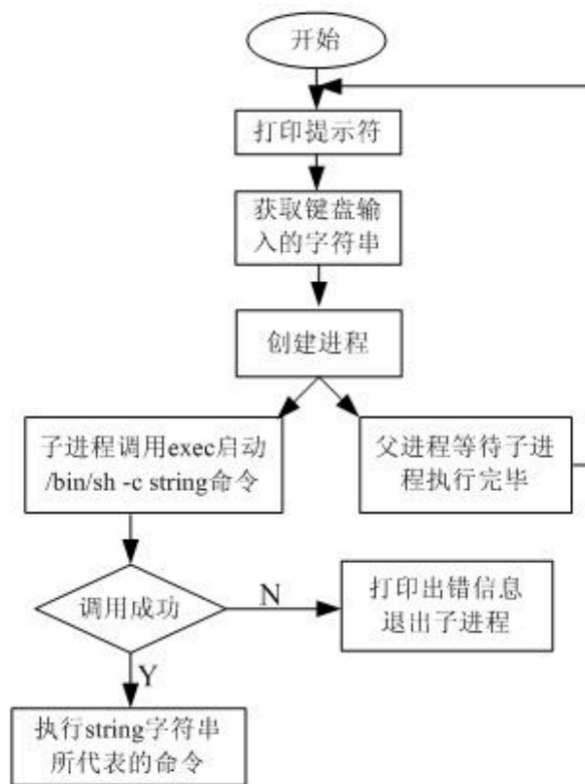
int main(int argc, char const *argv[])
{
    int ret = system(argv[1]);
    printf("ret = %d\n", ret);
    return 0;
}
```



```
34612 pts/24    00:00:00 a.out <defunct>
38194 pts/24    00:00:00 a.out
38195 pts/24    00:00:00 sh
38196 pts/24    00:00:00 ps
ret = 0
● disen@qfxa:~/code3/day03$ ./a.out ll
sh: 1: ll: not found
ret = 32512
○ disen@qfxa:~/code3/day03$
```

2.8 练习

实现 system 函数



三、信号

3.1 进程间通信概述

进程间通信(IPC: Inter Processes Communication)

进程是一个独立的资源分配单元，不同进程（这里所说的进程通常指的是用户进程）之间的资源是独立的，没有关联，不能在一个进程中直接访问另一个进程的资源（例如打开的文件描述符）。

进程不是孤立的，不同的进程需要进行信息的交互和状态的传递等，因此需要进程间通信。

进程间通信功能：

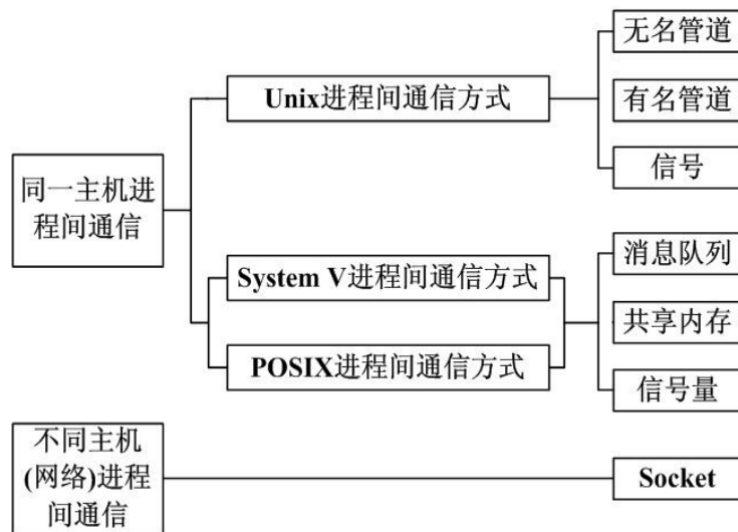
数据传输： 一个进程需要将它的数据发送给另一个进程。

资源共享： 多个进程之间共享同样的资源。

通知事件： 一个进程需要向另一个或一组进程发送消息，通知它们发生了某种事件。

进程控制： 有些进程希望完全控制另一个进程的执行（如 **Debug** 进程），此时控制进程希望能够拦截另一个进程的所有操作，并能够及时知道它的状态改变。

Linux 操作系统支持的主要进程间通信的通信机制：



3.2 信号的概念

信号是 Linux 进程间通信的最古老的方式。

信号是**软件中断**，它是在软件层次上对中断机制的一种模拟，是一种异步通信的方式。

信号可以直接进行用户空间进程和内核空间进程的交互，内核进程可以利用它来通知用户空间进程发生了哪些系统事件。

信号的特点

简单，不能携带大量信息，满足某个特设条件才发出。

每个信号的名字都以字符 SIG 开头。

每个信号和一个数字编码相对应，在头文件 `signal.h` 中，这些信号都被定义为正整数。

在 Linux 下，要想查看这些信号和编码的对应关系，可使用命令：`kill -l`

```

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

信号名称	信号编号	描述
SIGHUP	1	终端挂起或控制进程终止
SIGINT	2	中断信号，通常由终端上的Ctrl+C键触发；终止进程（由终端产生的中断信号）
SIGQUIT	3	退出信号，通常由终端上的Ctrl+\键触发；带有核心转储(core dump)的终止进程
SIGILL	4	非法指令
SIGABRT	6	异常终止
SIGFPE	8	浮点异常
SIGKILL	9	强制终止进程
SIGSEGV	11	无效的内存引用
SIGPIPE	13	消息管道破裂
SIGALRM	14	计时器超时
SIGTERM	15	终止请求
SIGUSR1	30,10,16	用户定义信号1
SIGUSR2	31,12,17	用户定义信号2
SIGCHLD	20,17,18	子进程状态改变
SIGCONT	19,18,25	继续执行（停止的进程）
SIGSTOP	17,19,23	停止进程
SIGTSTP	18,20,24	终端停止信号
SIGTTIN	21,21,26	后台进程尝试读取终端
SIGTTOU	22,22,27	后台进程尝试写入终端

不存在编号为 0 的信号。其中 1-31 号信号称之为常规信号（也叫普通信号或标准信号），34-64 称之为实时信号。

以下条件可以产生一个信号：

- 1、当用户按某些终端键时，将产生信号，如ctrl+c
- 2、硬件异常将产生信号。如 除数为 0，无效的内存访问等
- 3、软件异常将产生信号。如当检测到某种软件条件已发生，并将其通知有关进程时，产生信号
- 4、调用 kill 函数将发送信号
- 5、运行 kill 命令将发送信号。

一个进程收到一个信号的时候，可以用如下方法进行处理：

- 1、执行系统默认动作
- 2、忽略此信号
- 3、执行自定义信号处理函数

【注意】SIGKILL 和 SIGSTOP 不能更改信号的处理方式，因为它们向用户提供了一种使进程终止的可靠方法。

3.2 信号的基本操作

3.2.1 kill函数

给指定进程发送信号。

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int signum);
```

signum: 信号的编号, 一般使用宏名 (SIGKILL、SIGQUIT等)。

pid 的取值有 4 种情况:

```
pid>0: 将信号传送给进程 ID 为 pid 的进程。
pid=0: 将信号传送给当前进程所在进程组中的所有进程。
pid=-1: 将信号传送给系统内所有的进程。【不要轻易使用】
pid<-1: 将信号传给指定进程组的所有进程。这个进程组号等于 pid 的绝对值
```

【注意】使用 kill 函数发送信号，接收信号进程和发送信号进程的所有者必须相同，或者发送信号进程的所有者是超级用户。

如: t11.c

```
int main(int argc, char const *argv[])
{
    fork();
    printf("%d(%d) 努力工作中...\n", getpid(), getpgid(0));

    while (1)
        ;
    return 0;
}
```

```
disen@qfxa:~/code3/day03$ ./t11
2705(2705) 努力工作中...
2706(2705) 努力工作中...
Killed
```

t12.c

```
#include <stdio.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <unistd.h>
#include <fcntl.h>

int main(int argc, char const *argv[])
{
    // printf("干掉 同组的所有进程(自己)\n"); // kill(0, SIGKILL)
    // printf("干掉 系统的所有进程\n"); // kill(-1, SIGKILL);
    printf("干掉 2705组中所有的进程\n");
    int s = kill(-2705, SIGKILL);
    printf("kill result is %d \n", s);
    return 0;
}
```

```
disen@qfxa:~/code3/day03$ ./a.out
干掉 2705组中所有的进程
kill result is 0
```

如2: t13.c

```
#include <stdio.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>

int main(int argc, char const *argv[])
{
    int pid = fork();
    if (pid == 0)
    {
        while (1)
        {
            printf("%d 玩一会游戏\n", getpid());
            sleep(2);
        }
    }
    else if (pid > 0)
    {
        printf("%d 子进程正在学习\n", pid);
        sleep(5);
        kill(pid, SIGKILL);
        printf("主进程 干掉了 子进程 %d\n", pid);
    }
    return 0;
}
```

```
● disen@qfxa:~/code3/day03$ ./a.out
2942 子进程正在学习
2942 玩一会游戏
2942 玩一会游戏
2942 玩一会游戏
主进程 干掉了 子进程 2942
```

3.2.2 alarm函数

在 seconds 秒后，向调用进程发送一个 SIGALRM 信号，SIGALRM 信号的默认动作是终止调用alarm 函数的进程。 【非阻塞的】

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

返回值：

若以前没有设置过定时器，或设置的定时器已超时，返回0；
否则返回定时器剩余的秒数，并重新设定定时器。

如：

```
#include <stdio.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>

int main(int argc, char const *argv[])
{
    int pid = fork();
    if (pid == 0)
    {
        // 子进程
        printf("10秒之后结束\n");
        int s = alarm(10); // 第一次调用， s返回0
        printf("alarm s is %d\n", s);
        sleep(3);
        s = alarm(10);
        printf("alarm s is %d\n", s);

        while (1)
            ;
    }
    return 0;
}
```

```

● disen@qfxx:~/code3/day03$ ./a.out
10秒之后结束
alarm s is 0
○ disen@qfxx:~/code3/day03$ alarm s is 7

```

【扩展】设置定时器（闹钟）

```

#include <stdio.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/time.h>

void alarm_handle(int sig)
{
    printf("接收 SIGALRM(%d)\n", sig);
}

int main(int argc, char const *argv[])
{
    struct itimerval newVal;
    // 首次执行的时间(迟延)
    newVal.it_value.tv_sec = 5;
    newVal.it_value.tv_usec = 0;

    newVal.it_interval.tv_sec = 1;
    newVal.it_interval.tv_usec = 0;
    signal(SIGALRM, alarm_handle); // 注册信号的处理函数
    setitimer(ITIMER_REAL, &newVal, NULL);
    while (1)
    {
        ;
    }
    return 0;
}

```

```

● disen@qfxx:~/code3/day03$ gcc t15.c
⊗ disen@qfxx:~/code3/day03$ ./a.out
接收 SIGALRM(14)
接收 SIGALRM(14)
接收 SIGALRM(14)
接收 SIGALRM(14)
接收 SIGALRM(14)
接收 SIGALRM(14)

```

3.2.3 raise 函数

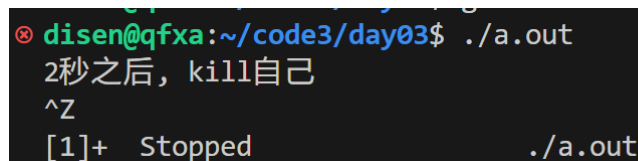
给调用进程本身送一个信号

```
#include <signal.h>
int raise(int signum);
```

返回值：成功返回 0，失败返回 -1。

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

int main(int argc, char const *argv[])
{
    printf("2秒之后，kill自己\n");
    sleep(2);
    raise(SIGALRM); // 发送中断信号（SIGALRM，SIGKILL，SIGQUIT）
    sleep(10);
    printf("---over---\n");
    return 0;
}
```



```
disen@qfxa:~/code3/day03$ ./a.out
2秒之后，kill自己
^Z
[1]+  Stopped                  ./a.out
```

3.2.4 abort 函数

向进程发送一个 SIGABRT 信号，默认情况下进程会退出。

```
#include <stdlib.h>
void abort(void);
```

【注意】即使 SIGABRT 信号被加入阻塞集，一旦进程调用了 abort 函数，进程也还是会被终止，且在终止前会刷新缓冲区，关文件描述符。

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

int main(int argc, char const *argv[])
{
    printf("2秒之后中断自己\n");
    sleep(2);
    printf("---over---");
    // raise(SIGABRT); // 不会刷新缓冲区
    abort(); // 结束进程之前，刷新缓冲区，关闭所有的文件描述符（0/1/2）
}
```

```
    return 0;
}
```

```
⊗ disen@qfxa:~/code3/day03$ ./a.out
2秒之后中断自己
---over---Aborted (core dumped)
○ disen@qfxa:~/code3/day03$
```

3.2.5 pause 函数

将调用进程挂起直至捕捉到信号为止。这个函数通常用于判断信号是否已到

```
#include <unistd.h>
int pause(void);
```

返回值：直到捕获到信号，pause 函数才返回-1，且 errno 被设置成 EINTR。

如：

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

void signal_handle(int sig)
{
    printf("接收到信号: %d\n", sig);
}

int main(int argc, char const *argv[])
{
    printf("挂起当前进程，等待信号\n");
    signal(SIGALRM, signal_handle); // 修改信号的处理方式
    alarm(5);
    int f = pause();
    printf("等到的信号, -1==%d \n", f);
    return 0;
}
```

```
● disen@qfxa:~/code3/day03$ ./a.out
挂起当前进程，等待信号
接收到信号: 14
等到的信号, -1== -1
```


3.2.6 处理信号

程序中可用函数 `signal()` 改变信号的处理方式。

```
#include <signal.h>

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

注册信号处理函数（不可用于 **SIGKILL**、**SIGSTOP** 信号），即确定收到信号后处理函数的入口地址。

handler 的取值：

忽略该信号：	<code>SIG_IGN</code>
执行系统默认动作：	<code>SIG_DFL</code>
自定义信号处理函数：	信号处理函数名

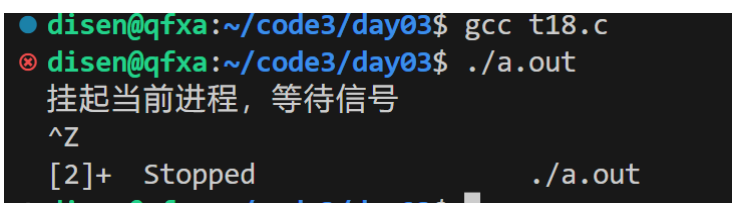
返回值：

成功：返回函数地址，该地址为此信号上一次注册的信号处理函数的地址。
失败：返回 `SIG_ERR`

如：

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

int main(int argc, char const *argv[])
{
    printf("挂起当前进程，等待信号\n");
    // ignore, dfl(default)
    // 忽略SIGALRM信号，信号到达时，自动屏蔽。
    signal(SIGALRM, SIG_IGN); // 修改信号的处理方式
    alarm(5);
    int f = pause();
    printf("等到的信号，-1==%d \n", f);
    return 0;
}
```



```
disen@qfxa:~/code3/day03$ gcc t18.c
disen@qfxa:~/code3/day03$ ./a.out
挂起当前进程，等待信号
^Z
[2]+  Stopped                  ./a.out
```

【扩展】sigaction 函数

检查或修改指定信号的设置（或同时执行这两种操作）

使用sigaction()处理信号时，必须在第一行声明宏：

```
#define _XOPEN_SOURCE 700
```

```
#include <signal.h>
int sigaction(int signum,
               const struct sigaction *act,
               struct sigaction *oldact);
```

参数：

signum：要操作的信号。

act：要设置的对信号的新处理方式（传入参数）。

oldact：原来对信号的处理方式（传出参数）。

如果 **act** 指针非空，则要改变指定信号的处理方式（设置），如果 **oldact** 指针非空，则系统将此前指定信号的处理方式存入 **oldact**，可以为NULL。

返回值：成功：0 失败：-1

```
struct sigaction
{
    void (*sa_handler)(int);           //旧的信号处理函数指针
    void (*sa_sigaction)(int, siginfo_t *, void *); //新的信号处理函数指针
    sigset_t sa_mask;                 //信号阻塞集
    int sa_flags;                     //信号处理的方式
    void (*sa_restorer)(void);        //已弃用
};
```

1) **sa_handler**、**sa_sigaction**：信号处理函数指针，和 **signal()** 里的函数指针用法一样，应根据情况给 **sa_sigaction**、**sa_handler** 两者之一赋值，其取值如下：

- a) **SIG_IGN**：忽略该信号
- b) **SIG_DFL**：执行系统默认动作
- c) 处理函数名：自定义信号处理函数

2) **sa_mask**：信号阻塞集，在信号处理函数执行过程中，临时屏蔽指定的信号。

3) **sa_flags**：用于指定信号处理的行为，通常设置为 0，表使用默认属性。它可以是以下值的“按位或”组合：

SA_RESTART：使被信号打断的系统调用自动重新发起（已经废弃）
SA_NOCLDSTOP：使父进程在它的子进程暂停或继续运行时不会收到 **SIGCHLD** 信号。
SA_NOCLDWAIT：使父进程在它的子进程退出时不会收到 **SIGCHLD** 信号，这时子进程如果退出也不会成为僵尸进程。
SA_NODEFER：使对信号的屏蔽无效，即在信号处理函数执行期间仍能发出这个信号。
SA_RESETHAND：信号处理之后重新设置为默认的处理方式。
SA_SIGINFO：使用 **sa_sigaction** 成员而不是 **sa_handler** 作为信号处理函数

信号处理函数：

```
void(*sa_sigaction)(int signum, siginfo_t *info, void *context);
```

参数说明:

signum: 信号的编号。

info: 记录信号发送进程信息的结构体。

context: 可以赋给指向 `ucontext_t` 类型的一个对象的指针，以引用在传递信号时被中断的接收进程

如:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>
void my_func(int signo)
{
    printf("CTRL+C被按下了\n");
    //_exit(-1);
}
int main(int argc, char const *argv[])
{
    struct sigaction act;
    // act存放回调函数
    act.sa_handler = my_func;

    // act给sa_mask赋值
    //清空阻塞集
    // sigemptyset(&act.sa_mask);
    //将所有信号添加到阻塞集中
    sigfillset(&act.sa_mask);

    //信号的处理方式
    // act.sa_flags = 0;
    act.sa_flags |= SA_RESETHAND;

    //注册CTRL+C信号-->SIGINT
    sigaction(SIGINT, &act, NULL);

    while (1)
        ;
    return 0;
}
```

如t19.c:

```
#define _XOPEN_SOURCE 700
#include <stdio.h>
#include <signal.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

void handler(int sig)
{
    printf("sig is %d, handle ...\n", sig);
    //_exit(1);
}
```

```

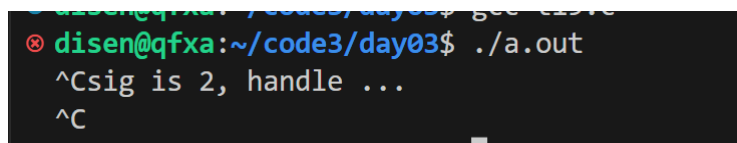
}

int main(int argc, char const *argv[])
{
    struct sigaction action;
    action.sa_handler = handler;

    // 信号只能被处理一次，下次恢复默认行为
    action.sa_flags |= SA_RESETHAND; // 重置默认的信号处理行为

    sigaction(SIGINT, &action, NULL);
    while (1)
        ;
    return 0;
}

```



```

disen@qfxa:~/code3/day03$ ./a.out
^Csig is 2, handle ...
^C

```

3.3 可重入函数

可重入函数是指函数可以由多个任务并发使用，而不必担心数据错误。

编写可重入函数：

- 1、不使用（返回）静态的数据、全局变量（除非用信号量互斥）。
- 2、不调用动态内存分配、释放的函数。
- 3、不调用任何不可重入的函数（如标准 I/O 函数）。

【注意】即使信号处理函数使用的都是可重入函数（常见的可重入函数），也要注意进入处理函数时，首先要保存 `errno` 的值，结束时，再恢复原值。因为，信号处理过程中，`errno` 值随时可能被改变。

常见的可重入函数列表：

accept	fchmod	lseek	sendto	stat
access	fchown	lstat	setgid	symlink
aio_error	fcntl	mkdir	setpgid	sysconf
aio_return	fdatasync	mkfifo	setsid	tcdrain
aio_suspend	fork	open	setsockopt	tcflow
alarm	fpathconf	pathconf	setuid	tcflush
bind	fstat	pause	shutdown	tcgetattr
cfgetispeed	fsync	pipe	sigaction	tcgetpgrp
cfgetospeed	ftruncate	poll	sigaddset	tcsendbreak
cfsetispeed	getegid	posix_trace_event	sigdelset	tcsetattr
cfsetospeed	geteuid	pselect	sigemptyset	tcsetpgrp
chdir	getgid	raise	sigfillset	time
chmod	getgroups	read	sigismember	timer_getoverrun
chown	getpeername	readlink	signal	timer_gettime
clock_gettime	getpgrp	recv	sigpause	timer_settime
close	getpid	recvfrom	sigpending	times
connect	getppid	recvmsg	sigprocmask	umask
creat	getsockname	rename	sigqueue	uname
dup	getsockopt	rmdir	sigset	unlink
dup2	getuid	select	sigsuspend	utime
execle	kill	sem_post	sleep	wait
execve	link	send	socket	waitpid
_Exit & _exit	listen	sendmsg	socketpair	write

如:

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

void show(int n)
{
    for (int i = 0; i < n; i++)
    {
        char buf[32];
        sprintf(buf, "%d 进程 -> %d\n", getpid(), i);
        write(STDOUT_FILENO, buf, strlen(buf));
        sleep(1);
    }
}

int main(int argc, char const *argv[])
{

```

```

for (int i = 0; i < 5; i++)
{
    int pid = fork();
    if (pid == 0)
    {
        show(i + 1);
        _exit(0);
    }
}
while (1)
{
    int status;
    int pid = waitpid(0, &status, WUNTRACED);
    if (pid == -1)
        break;
    printf("%d 子进程结束\n", pid);
}
return 0;
}

```

```

● disen@qfxa:~/code3/day03$ ./a.out
7091 进程 -> 0
7092 进程 -> 0
7093 进程 -> 0
7094 进程 -> 0
7095 进程 -> 0
7091 子进程结束
7092 进程 -> 1
7095 进程 -> 1
7093 进程 -> 1
7094 进程 -> 1
7092 子进程结束
7095 进程 -> 2
7093 进程 -> 2
7094 进程 -> 2
7095 进程 -> 3
7093 子进程结束
7094 进程 -> 3
7094 子进程结束
7095 进程 -> 4
7095 子进程结束

```

3.4 信号集

3.4.1 信号集概述

一个用户进程常常需要对多个信号做出处理。为了方便对多个信号进行处理，在 Linux 系统中引入了信号集。

【扩展】在 PCB 中有两个非常重要的信号集。一个称之为“阻塞信号集”，另一个称之为“未决信号集”。这两个信号集都是内核使用位图机制来实现的。但操作系统不允许我们直接对其进行位操作。而需自定义另外一个集合，借助信号集操作函数来对 PCB 中的这两个信号集进行修改。

信号集是用来表示多个信号的数据类型。

信号集数据类型: `sigset_t`

定义路径: `/usr/include/x86_64-linux-gnu/bits/sigset.h`

信号集相关的操作主要有如下几个函数:

```
sigemptyset sigfillset sigismember sigaddset sigdelset
```

3.4.2 sigemptyset 函数

初始化由 set 指向的信号集, 清除其中所有的信号即初始化一个空信号集

```
#include <signal.h>
int sigemptyset(sigset_t *set)
```

成功返回 0, 失败返回 -1

3.4.3 sigfillset 函数

初始化一个满的信号集

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

3.4.4 sigismember 函数

判断某个集合中是否有某个信号

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

返回值: 在信号集中返回 1, 不在信号集中返回 0

3.4.5 sigaddset 函数

向某个集合中添加一个信号

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

3.4.6 sigdelset 函数

从某个信号集中删除一个信号

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

3.4.7 综合示例

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>
```

```

int main(int argc, char const *argv[])
{
    //定义一个信号集
    sigset_t set;

    //清空set集
    sigemptyset(&set);

    //将SIGINT添加到set集合中
    sigaddset(&set, SIGINT);
    //将SIGTSTP添加到set集合中
    sigaddset(&set, SIGTSTP);

    if (sigismember(&set, SIGINT))
    {
        printf("SIGINT是在set集合中\n");
    }
    else
    {
        printf("SIGINT不在set集合中\n");
    }

    //将SIGINT从集合中删除
    sigdelset(&set, SIGINT);
    if (sigismember(&set, SIGINT))
    {
        printf("SIGINT是在set集合中\n");
    }
    else
    {
        printf("SIGINT不在set集合中\n");
    }

    return 0;
}

```

如t21.c

```

#define _XOPEN_SOURCE 700
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

int main(int argc, char const *argv[])
{
    sigset_t set;
    sigemptyset(&set); // 清空信号集
    // sigfillset(&set); // 全部信号填充信号集
    printf("SIGINT in set ? %d\n", sigismember(&set, SIGINT));

    sigaddset(&set, SIGINT); // CTRL+C 进程中断
    sigaddset(&set, SIGTSTP); // 终端停止
    printf("SIGINT in set ? %d\n", sigismember(&set, SIGINT));

    sigdelset(&set, SIGTSTP);
    printf("SIGTSTP in set ? %d\n", sigismember(&set, SIGTSTP));
}

```



```
    return 0;
}
```

```
● disen@qfxa:~/code3/day03$ ./a.out
SIGINT in set ? 0
SIGINT in set ? 1
SIGTSTP in set ? 0
```

3.5 信号阻塞集(屏蔽集、掩码)

每个进程都有一个阻塞集，它用来描述哪些信号递送到该进程的时候被阻塞(在信号发生时记住它，直到进程准备好时再将信号通知进程)。

所谓阻塞并不是禁止传送信号，而是暂缓信号的传送。若将被阻塞的信号从信号阻塞集中删除，且对应的信号在被阻塞时发生了，进程将会收到相应的信号。

创建一个阻塞集合

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

功能：

检查或修改信号阻塞集，根据 `how` 指定的方法对进程的阻塞集合进行修改，新的信号阻塞集由 `set` 指定，而原先的信号阻塞集合由 `oldset` 保存

参数：`how`：信号阻塞集合的修改方法

`SIG_BLOCK`：向信号阻塞集合中添加 `set` 信号集
`SIG_UNBLOCK`：从信号阻塞集合中删除 `set` 集合
`SIG_SETMASK`：将信号阻塞集合设为 `set` 集合

返回值：0 成功，-1 失败。

如：

```
#define _XOPEN_SOURCE 700
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

int main(int argc, char const *argv[])
{
    sigset_t set;
    sigemptyset(&set); // 清空信号集
    sigaddset(&set, SIGINT);

    printf("---阻塞CTRL+C信号 10秒中---\n");
    sigprocmask(SIG_BLOCK, &set, NULL);
    sleep(10);
    sigprocmask(SIG_UNBLOCK, &set, NULL);
```

```
    return 0;  
}
```

```
⊗ disen@qfxx:~/code3/day03$ ./a.out  
---阻塞CTRL+C信号 10秒中---  
^C  
○ disen@qfxx:~/code3/day03$
```