

第二天系统调用与进程

一、回顾知识点

1.1 shell解释器

```
/bin/bash  [默认]
/bin/sh
```

1.2 shell脚本调用

系统自动调用：

```
/etc/profile    系统环境变量， 每次系统启动时调用
~/.bashrc       用户环境变量， 用户登录时调用
```

用户调用：

```
1)  chmod +x xxx.sh
    ./xxx.sh
2)  bash xxx.sh
3)  .  xxx.sh    相当于 source xxx.sh
```

1.3 shell脚本的定义

第一行：

```
#!/bin/bash
```

1.4 shell变量

```
定义：    变量名=值
引用：    $变量名， "$变量名 xxx"
重新赋值： 变量名=新值
算术计算： 变量名=$(( 变量名 算术运算符 值 ))

预定变量：  $#, $*, $?,  $1~9,  $0 进程名, $$ 进程ID
环境变量：  $环境变量名
            export 变量名=值    设置环境变量

特殊的变量写法：  (变量名=值； 其它语句)    不影响外部的同名变量
                  {变量名=值； 其它语句;}    影响外部的同名变量
```

1.5 条件测试语句

语法:

```
test 操作符 变量
test 变量1 操作符 变量2
[ 操作符 变量 ]
[ 变量1 操作符 变量2 ]
```

文件相关:

-e 是否存在	-d 是目录	-f 是文件
-r 可读	-w 可写	-x 可执行
-L 符号连接	-c 是否字符设备	-b 是否块设备
-s 文件非空		

字符串相关:

= 两个字符串相等	!= 两个字符串不相等
-z 空串	-n 非空串

数字相关:

-eq 数值相等	-ne 数值不相等
-gt 数 1 大于数 2	-ge 数 1 大于等于数 2
-lt 数 1 小于数 2	-le 数 1 小于等于数 2

复合测试相关:

command1 && command2	左边命令执行成功(即返回 0) shell 才执行右边的命令
command1 command2	左边的命令未执行成功(即返回非 0) shell 才执行右边的命令

多重条件判定:

-a	多个条件必须都为真(true) 结果才为true
-o	多个条件中只需要一个为真, 结果为true
!	条件取反

1.6 分支语句

if分支:

```
if [ 条件1 ]; then
    执行第一段程序
elif [条件2 ]; then
    执行第二段程序
else
    执行第三段程序
fi
```

case分支:

```
case $变量名称 in
    “第一个变量内容”)
        程序段一
        ;;
    “第二个变量内容”)
        程序段二
        ;;
    *)
        其它程序段
    exit 1
esac
```

for语句：

```
for (( 初始值； 限制值； 执行步阶 ))
do
    程序段
done
```

while语句：

```
while [ condition ]
do
    程序段
done
```

until 语句：

```
until [ condition ]
do
    程序段
done
```

循环语句中可以使用 break和continue。

1.7 函数

定义函数：

```
函数名(){
    命令 ...
}

function 函数名(){
    命令 ...
}
```

调用函数：

```
函数名 param1 param2.....
```

函数体内，可以使用\$1，\$2 ...\$9 读取传递到函数的参数。

vscode应用RemoteSSH插件

1) 免密的ssh远程登录

在Windows的Power Shell终端上，执行命令生成公钥和私钥

```
Windows PowerShell
版权所有 (C) Microsoft Corporation。保留所有权利。

安装最新的 PowerShell，了解新功能和改进！ https://aka.ms/PSWindows

PS C:\Users\Disen> ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (C:\Users\Disen\.ssh\id_rsa):
C:\Users\Disen\.ssh\id_rsa already exists.
Overwrite (y/n)? y
Enter passphrase (empty for no passphrase)
Enter same passphrase again:
Your identification has been saved in C:\Users\Disen\.ssh\id_rsa
Your public key has been saved in C:\Users\Disen\.ssh\id_rsa.pub
The key fingerprint is:
SHA256:YtHvMhRkcjT7yD/oVQy7mZBCgrNHGY82dCCm2Ew64og disen@LAPTOP-5BPRSMG9
The key's randomart image is:
+----[RSA 3072]-----+
|  + + . o *
|.B + * * o
|* = B + + .
|=. = + o * +
|E.. . + S + o
|  . . + = =
|    + 0
|    . + .
|    .
+----[SHA256]-----+
PS C:\Users\Disen>
```

公钥的位置

直接回车

将公钥上传到 远程Linux系统中

```
PS C:\Users\Disen> cd ~/.ssh
PS C:\Users\Disen\.ssh> scp id_rsa.pub disen@192.168.74.129:~/
disen@192.168.74.129's password:
id_rsa.pub                                100% 576 155.4KB/s 00:00
PS C:\Users\Disen\.ssh>
```

进入到Linux系统中，将Window上传的公钥写入到 ~/.ssh/authorized_keys文件中

```
disen@qfxa:~$ ls -l
total 1028
drwxrwxr-x 3 disen disen 4096 8月 14 09:40 cmd_test
drwxrwxr-x 17 disen disen 4096 8月 14 20:22 code
drwxr-xr-x 15 disen disen 4096 8月 4 10:03 code2
drwxr-xr-x 2 disen disen 4096 7月 3 17:04 Desktop
drwxr-xr-x 2 disen disen 4096 7月 3 17:04 Documents
drwxr-xr-x 2 disen disen 4096 7月 3 17:04 Downloads
-rw-r--r-- 1 disen disen 8980 7月 3 16:47 examples.desktop
-rw-rw-r-- 1 disen disen 956786 7月 13 20:36 gcc.txt
-rw-rw-r-- 1 disen disen 576 8月 15 09:20 id_rsa.pub
drwxrwxr-x 2 disen disen 4096 7月 10 10:13 includes
drwxr-xr-x 2 disen disen 4096 7月 3 17:04 Music
drwxrwxr-x 4 disen disen 4096 7月 12 10:20 my
drwxrwxr-x 2 disen disen 4096 7月 12 11:24 my2
drwxr-xr-x 2 disen disen 4096 7月 3 17:04 Pictures
drwxr-xr-x 2 disen disen 4096 7月 3 17:04 Public
drwxrwxr-x 3 disen disen 4096 8月 14 17:47 shells
-rw-r--r-- 1 disen disen 9599 7月 4 11:58 smb.conf
drwxr-xr-x 2 disen disen 4096 7月 3 17:04 Templates
drwxrwxr-x 3 disen disen 4096 8月 14 09:54 test
drwxr-xr-x 2 disen disen 4096 7月 3 17:04 Videos
disen@qfxa:~$ cat id_rsa.pub >> ~/.ssh/authorized_keys
bash: /home/disen/.ssh/authorized_keys: No such file or directory
disen@qfxa:~$ ls .ssh
ls: cannot access '.ssh': No such file or directory
disen@qfxa:~$ mkdir .ssh
disen@qfxa:~$ cat id_rsa.pub >> ~/.ssh/authorized_keys
disen@qfxa:~$
```

Window上传的公钥文件

授权

.ssh不存在，则手动创建

测试ssh远程连接是否需要口令：

```
PS C:\Users\Disen\.ssh> ssh disen@192.168.74.129
Welcome to Ubuntu 16.04 LTS (GNU/Linux 4.4.0-21-generic x86_64)

 * Documentation:  https://help.ubuntu.com/

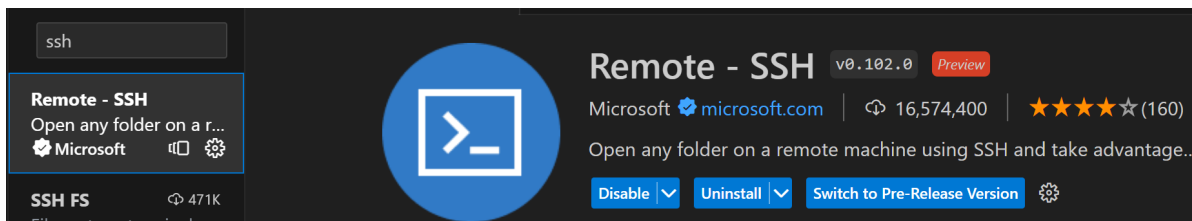
749 packages can be updated.
504 updates are security updates.

New release '18.04.6 LTS' available.
Run 'do-release-upgrade' to upgrade to it.


Last login: Mon Aug 14 15:31:18 2023 from 192.168.74.1
disen@qfxa:~$
```

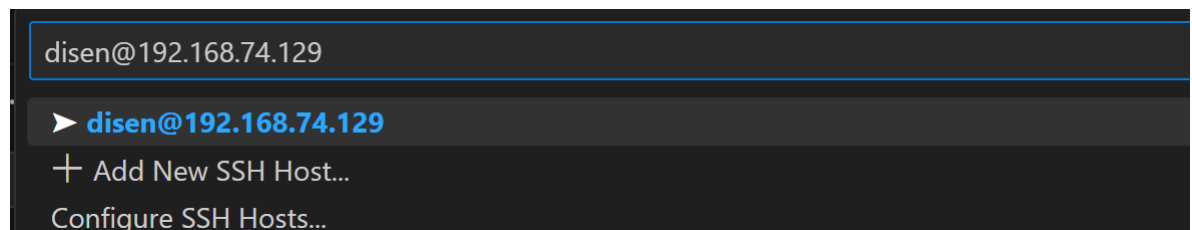
在远程连接时，没有提示输入口令，表示免密SSH登录成功。

2) 打开vscode安装remote ssh插件。

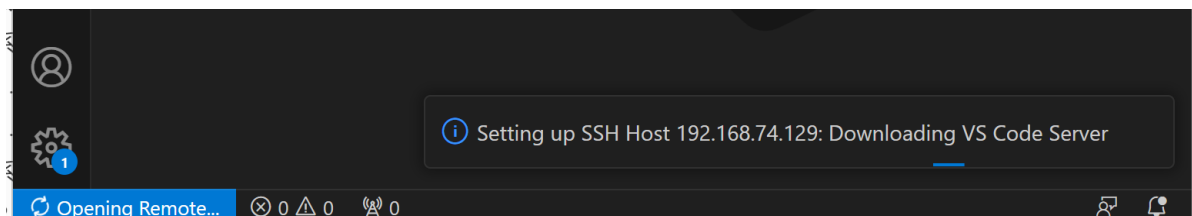


3) 连接远程的Linux

点击vscode左下角的 ，在顶部选项中，选择 **Connect to Host...**，然后在输入框中输入用户名和远程Linux主机的地址，回车后，打开新的连接窗口，并提示远程操作系统的类型（Linux、Windows, macOS）



选择Linux, 则会开始下载vscode server服务，等待2~5分钟。

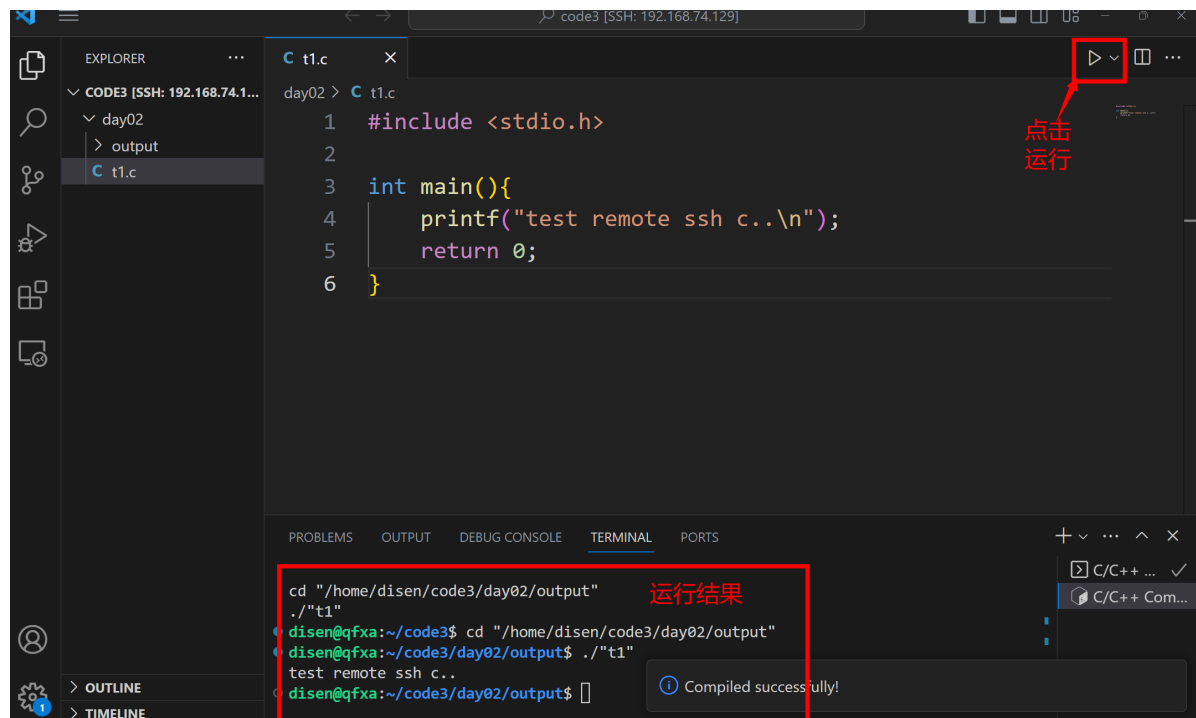


3) 在远程连接的窗口中，安装c/c++插件

4) 打开代码目录

需要在Linux中手动创建一个code3目录

打开code3目录，创建t1.c文件，编写部分代码，并运行。



二、系统调用

2.1 系统调用概述

2.1.1 什么是系统编程

操作系统的职责：

操作系统用来管理所有的资源，并将不同的设备和不同的程序关联起来。

什么是 Linux 系统编程：

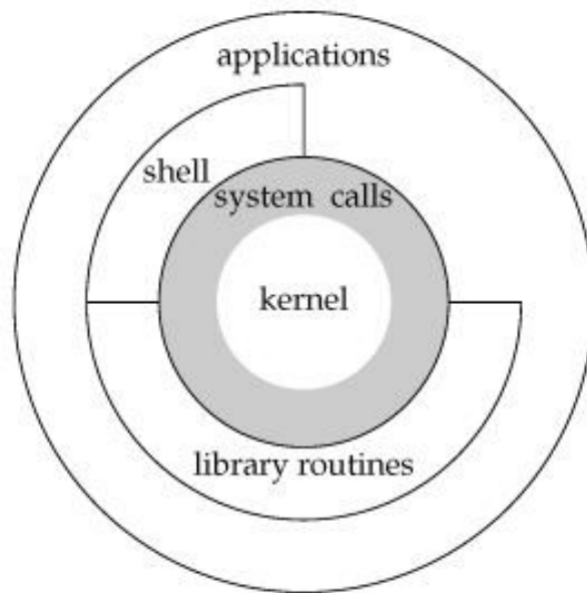
在有操作系统的环境下编程，并使用操作系统提供的系统调用及各种库（c开发），对系统资源进行访问

C 语言+系统调用的函数，就可以进行 Linux 系统编程。

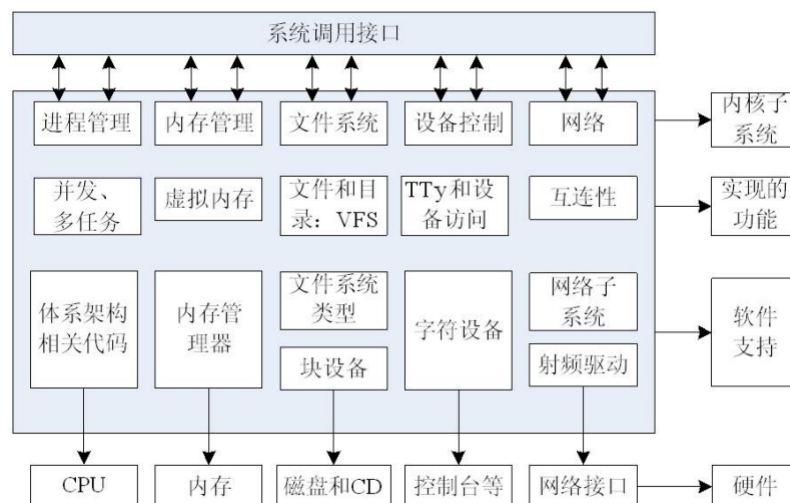
2.1.2 什么是系统调用

系统调用是操作系统提供给用户程序的一组“特殊”函数接口

类 UNIX 系统的软件层次



Linux 的不同版本提供了丰富（200+）的系统调用，用户程序可以通过这些接口获得操作系统（内核）提供的服务。例如，用户可以通过文件系统相关的系统调用，请求系统打开文件、关闭文件或读写文件。



系统调用按功能逻辑大致可分为：

进程控制、进程间通信、文件系统控制、系统控制、内存管理、网络管理、**socket**控制、用户管理。

系统调用的返回值：

通常，用负的返回值来表明错误，返回0值表明成功。
错误信息存放在全局变量**errno**中，用户可用**perror**函数打印出错信息。

系统调用遵循的规范：

在 Linux 中，应用程序编程接口(API)遵循 POSIX 标准

POSIX 标准基于当时现有的 UNIX 实践和经验，描述了操作系统的系统调用编程接口（实际上就是 API），用于保证应用程序可以在源代码一级上在多种操作系统上移植运行。

如：linux 下写的 open、write、read 可以直接移植到 unix 操作系统下。

2.2 系统调用 I/O 函数

系统调用中操作 I/O 的函数，都是针对文件描述符的。

通过文件描述符可以直接对相应的文件进行操作（open、close、write、read、ioctl）。

2.2.1 文件描述符

文件描述符是非负整数。

打开现存文件或新建文件时，系统（内核）会返回一个文件描述符。

文件描述符用来指定已打开的文件。

三个标准描述符：默认打开的

```
#define STDIN_FILENO 0 //标准输入的文件描述符
#define STDOUT_FILENO 1 //标准输出的文件描述符
#define STDERR_FILENO 2 //标准错误的文件描述符
```



fileno 新打开的文件号 1023

【扩展】

`ulimit -a` 查看open files打开的文件最大数。

`ulimit -n 最大数` 设置open files打开的文件最大数。

2.2.2 open 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
```

参数说明：

pathname: 文件的路径及文件名。
flags: open 函数的行为标志。
mode: 文件权限(可读、可写、可执行)的设置

返回值：

成功返回打开的文件描述符。
失败返回-1，可以利用 `perror` 去查看原因

flags 的取值及其含义

O_RDONLY 以只读的方式打开
O_WRONLY 以只写的方式打开
O_RDWR 以可读、可写的方式打开

flags 除了取上述值外，还可与下列值位或

O_CREAT 文件不存在则创建文件，使用此选项时需使用 mode 说明文件的权限
O_EXCL 确保创建操作是原子的；如果同时指定O_CREAT，且文件已经存在，则出错（执行了Create创建时，因为文件已存在则报错）

O_TRUNC 如果文件存在，则清空文件内容
O_APPEND 写文件时，数据添加到文件末尾
O_NONBLOCK 当打开的文件是 FIFO、字符文件、块文件时，此选项为非阻塞标志位

mode 的取值及其含义

mode 的取值及其含义		
取值	八进制数	含义
S_IRWXU	00700	文件所有者的读、写、可执行权限
S_IRUSR	00400	文件所有者的读权限
S_IWUSR	00200	文件所有者的写权限
S_IXUSR	00100	文件所有者的可执行权限
S_IRWXG	00070	文件所有者同组用户的读、写、可执行权限
S_IRGRP	00040	文件所有者同组用户的读权限
S_IWGRP	00020	文件所有者同组用户的写权限
S_IXGRP	00010	文件所有者同组用户的可执行权限
S_IRWXO	00007	其他组用户的读、写、可执行权限
S_IROTH	00004	其他组用户的读权限
S_IWOTH	00002	其他组用户的写权限
S_IXOTH	00001	其他组用户的可执行权限

如：打开与关闭文件

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

#include <unistd.h> // close()

int main()
{
    // 权限： 644
    int fd = open("a.txt", O_WRONLY | O_CREAT | O_TRUNC, 0755);
    // int fd = open("a.txt", O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR |
    S_IRGRP | S_IROTH);
```

```

if (fd < 0)
{
    perror("a.txt文件打开失败");
    return 1;
}

printf("a.txt 文件打开并创建成功\n");
if (close(fd) < 0)
{
    printf("关闭 a.txt文件失败"); // 关闭文件
}
return 0;
}

```

```

● disen@qfxa:~/code3/day02/output$ ./"t2"
a.txt 文件打开并创建成功
● disen@qfxa:~/code3/day02/output$ ls -l
total 88
-rwxr-xr-x 1 disen disen    0 8月  15 11:10 a.txt
-rwxrwxr-x 1 disen disen 29448 8月  15 09:47 t1
-rwxrwxr-x 1 disen disen 56856 8月  15 11:10 t2

```

【查看权限掩码】umask

umask: 查看当前的权限掩码
umask mode: 设置掩码, mode 为八进制 0ddd
umask -S: 查看各组用户的默认操作权限

```

disen@qfxa:~/code3/day02/output$ umask
0002
disen@qfxa:~/code3/day02/output$ umask -S
u=rwx,g=rwx,o=rX
disen@qfxa:~/code3/day02/output$ umask 0133
disen@qfxa:~/code3/day02/output$ umask -S
u=rw,g=r,o=r

```

2.2.3 close 函数

关闭一个文件

```

#include <unistd.h>
int close(int fd);

```

fd 是调用 open 打开文件返回的文件描述符。成功返回 0, 失败返回-1。

2.2.4 write 函数

把指定数目的数据写到文件

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

参数说明:

fd: 文件描述符。
addr: 数据首地址。
count: 写入数据的字节个数

成功返回实际写入数据的字节个数，失败返回-1。

如:

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

int main()
{
    // 以只写模式且清空
    int fd = open("a.txt", O_WRONLY | O_TRUNC);
    if (fd < 0)
    {
        perror("a.txt open fail");
        return 1;
    }
    char content[32] = "hello";
    // ssize_t == long
    ssize_t len = write(fd, content, sizeof(content));
    if (len != -1)
    {
        printf("向文件号: %d, 写入数据成功: %ld bytes \n", fd, len);
    }

    if (close(fd) < 0)
    {
        perror("a.txt close fail");
    }
    return 0;
}
```

```
● disen@qfxa:~/code3/day02/output$ ./"t3"
向文件号: 3, 写入数据成功: 32 bytes
```

2.2.5 read 函数

把指定数目的数据读到内存，默认是**阻塞的**（如果读不到数据，将阻塞不继续执行，直到有数据可读，才继续往下执行）。

非阻塞特性：如果没数据，立即返回，继续执行

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

成功返回实际读取到的字节个数。失败返回-1，可以利用 perror 去查看原因。

如：

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

int main()
{
    int fd = open("a.txt", O_RDONLY);
    if (fd < 0)
    {
        perror("a.txt open fail");
        return 1;
    }

    char buffer[128] = "";
    ssize_t len = read(fd, buffer, 128);
    if (len != -1)
    {
        printf("读取的数据: %s, 大小: %ld bytes\n", buffer, len);
    }

    close(fd);
    return 0;
}
```

```
disen@qfxa:~/code3/day02/output$ ./"t4"
读取的数据: hello, 大小: 32 bytes
```

【扩展】文件阻塞特性

- 1) 尝试从 /dev/tty 终端读取数据（非阻塞）（先printf()打印数据，再读）
- 2) int fcntl(int fd, int cmd, ...) 针对文件描述符进行控制

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd, ... /* arg */);
```

功能：改变已打开的文件性质，fcntl 针对描述符提供控制。

参数：

- fd：操作的文件描述符
- cmd：操作方式
- arg：针对 cmd 的值，fcntl 能够接受第三个参数 int arg。

返回值：

- 成功：返回某个其他值
- 失败：-1

fcntl 函数有 5 种功能：

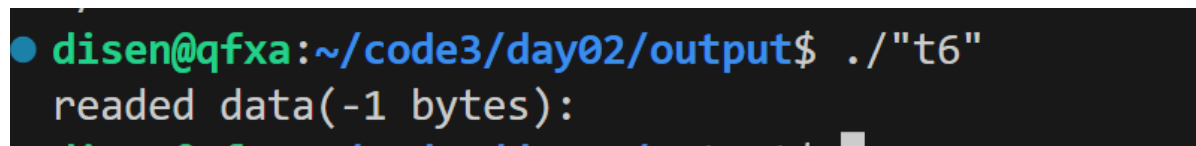
- 1) 复制一个现有的描述符 (cmd=F_DUPFD)
- 2) 获得 / 设置文件描述符标记(cmd=FGETFD 或 FSETFD)
- 3) 获得 / 设置文件状态标记(cmd=FGETFL 或 FSETFL)
- 4) 获得 / 设置异步 I/O 所有权(cmd=FGETOWN 或 FSETOWN)
- 5) 获得 / 设置记录锁(cmd=FGETLK, FSETLK 或 F_SETLKW)

如：

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

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

    // 从标准的输入设备读取32个字节
    char buffer[32] = "";
    // 读取数据时，遇到 回车则结束阻塞(非 O_NONBLOCK )
    // O_NONBLOCK 标记时，立即读取数据，即使终端没有录入数据
    long len = read(STDIN_FILENO, buffer, 32);
    printf("readed data(%ld bytes): %s\n", len, buffer);
    return 0;
}
```



```
disen@qfxa:~/code3/day02/output$ ./"t6"
readed data(-1 bytes):
```

2.2.6 remove 库函数

删除文件或空目录

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

成功返回 0。失败返回-1，可以利用 perror 去查看原因。

```
#include <stdio.h>

int main()
{
    int ret = remove("a.txt");
    if (ret < 0)
    {
        perror("delete a.txt fail");
    }
    return 0;
}
```

2.3 系统调用与库函数

库函数由两类函数组成：不需要调用 系统调用 和 需要调用 系统调用

不需要调用 系统调用：

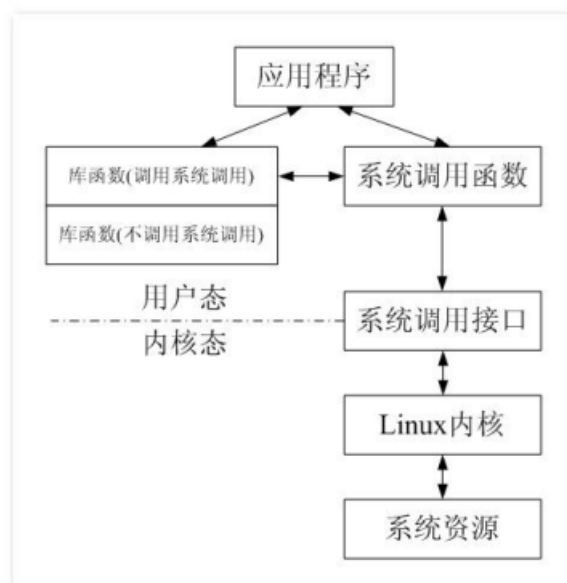
不需要切换到内核空间即可完成函数全部功能，并且将结果反馈给应用程序，如 strcpy、bzero 等字符串操作函数。

需要调用系统调用：

需要切换到内核空间，这类函数通过封装系统调用去实现相应功能，如 printf、fread 等

库函数与系统调用的关系：

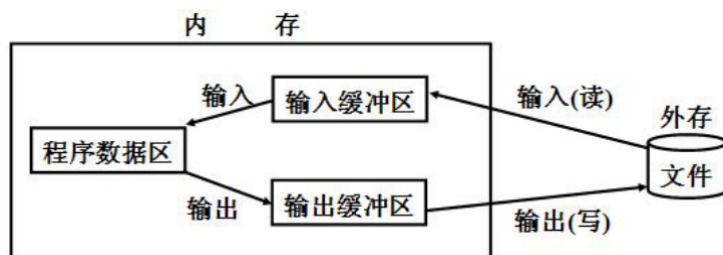
并不是所有的系统调用都被封装成了库函数，系统提供的很多功能都必须通过系统调用才能实现。



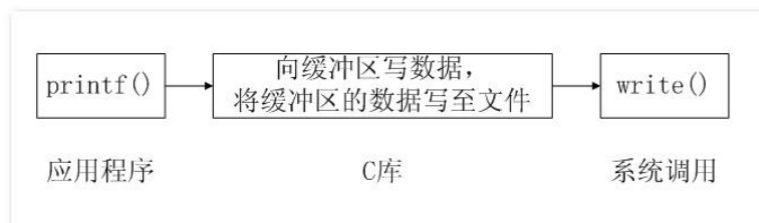
系统调用是需要时间的，程序中频繁的使用系统调用会降低程序的运行效率

当运行内核代码时，CPU 工作在内核态，在系统调用发生前需要保存用户态的栈和内存环境，然后转入内核态工作。系统调用结束后，又要切换回用户态。这种环境的切换会消耗掉许多时间。

库函数访问文件的时候根据需要，设置不同类型的缓冲区，从而减少了直接调用 IO 系统调用的次数，提高了访问效率。



应用程序调用 printf 函数时，函数执行的过程，如下图



2.4 练习：实现 cp 命令

基于系统调用实现： open,close,read, write

命令格式： cp src_file dst_dir

如：

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

int main(int argc, char const *argv[])
{
    printf("args num : %d\n", argc);
    // mycp t1.c abc
    if (argc != 3)
    {
        char error[128] = "命令格式不正确，正确的格式： ./a.out filename dst_dir\n";
        write(STDERR_FILENO, error, 128);
        return 1;
    }

    const char *src_file_path = argv[1];
    const char *dst_dir_path = argv[2];

    char dist_path[128] = "";
    sprintf(dist_path, "%s/%s", dst_dir_path, src_file_path);
    printf("filename: %s cp dst_path: %s\n", src_file_path, dist_path);

    int fd1 = open(src_file_path, O_RDONLY);
    if (fd1 < 0)
    {
        perror("open file");
        return 1;
    }
}
```

```

int fd2 = open(dist_path, O_WRONLY | O_CREAT | O_TRUNC, 0755);
if (fd2 < 0)
{
    perror("open file");
    return 1;
}

while (1)
{
    char buf[1024] = "";
    long len = read(fd1, buf, 1024);
    write(fd2, buf, len);
    if (len < 1024)
        break;
}

close(fd1);
close(fd2);
printf("ok\n");
return 0;
}

```

```

● disen@qfxa:~/code3/day02$ ./a.out t1.c abc
args num : 3
filename: t1.c cp dst_path: abc/t1.c
Ok
● disen@qfxa:~/code3/day02$ ./a.out t2.c abc
args num : 3
filename: t2.c cp dst_path: abc/t2.c
Ok

```

优化任务: 1) filename是一个文件路径 2) dst_dir 验证是否为目录

如: 给定一个文件路径 验证是否为目录或文件

```

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

int main(int argc, char const *argv[])
{
    const char *path = argv[1];

    struct stat fileStat;
    if (stat(path, &fileStat) == 0)
    {
        // ok
        if (fileStat.st_mode & __S_IFDIR)
        {
            printf("%s is dir\n", path);
        }
        else if (fileStat.st_mode & __S_IFREG)
        {
            printf("%s is file\n", path);
        }
    }
}

```



```

    }
}
return 0;
}

```

```

● disen@qfxa:~/code3/day02$ ./a.out t1.c
t1.c is file
● disen@qfxa:~/code3/day02$ ./a.out abc
abc is dir
● disen@qfxa:~/code3/day02$ ./a.out ~/code3/day02
/home/disen/code3/day02 is dir
○ disen@qfxa:~/code3/day02$ █

```

【扩展】目录的遍历

```

#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>
#include <string.h>
int main(int argc, char *argv[])
{
    //1、得到目录句柄
    DIR *dir = opendir("./");
    if(NULL == dir)
    {
        perror("opendir");
        return 0;
    }

    //2、根据目录句柄扫描当前文件
    while(1)
    {
        struct dirent *dirent = readdir(dir);
        if(NULL == dirent)
            break;
        //判断文件类型
        if(dirent->d_type == DT_DIR)
        {
            printf("%s是目录文件\n", dirent->d_name );
        }
        else if(dirent->d_type == DT_REG)
        {
            if(strstr(dirent->d_name, ".c") != NULL)
                printf("%s是普通文件\n", dirent->d_name );
        }
    }

    //关闭目录句柄
    closedir(dir);
}

```

三、进程

3.1 进程概述

3.1.1 进程的定义

程序：是存放在存储介质上的一个可执行文件。 【静态的】

进程：是程序的执行实例，包括程序计数器、寄存器和变量的当前值。 【动态的】

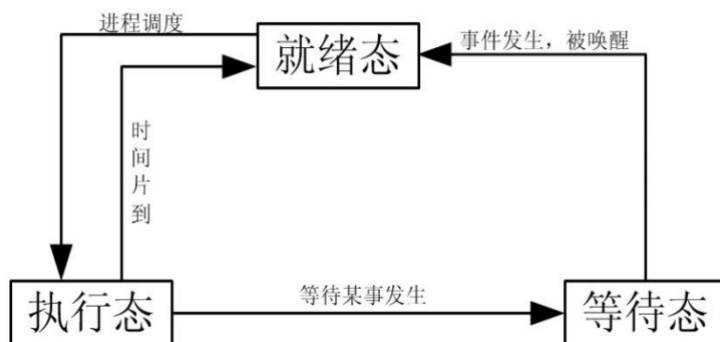
程序是一些指令的有序集合，而进程是程序执行的过程。进程的状态是变化的，其包括进程的创建、调度和消亡。

在 linux 系统中，进程是管理事务的基本单元。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）。

可使用 exec 函数由内核将程序读入内存，使其执行起来成为一个进程。

3.1.2 进程的状态及转换

进程整个生命周期可以简单划分为三种状态：



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。

3.1.3 进程控制块

进程控制块（PCB，Process Control Block）

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。PCB 中记录了用于描述进程进展情况及控制进程运行所需的全部信息。

PCB 是进程存在的唯一标志，在 Linux 中 PCB 存放在 `task_struct` 结构体中。

3.2 进程控制

3.2.1 进程号

每个进程都由一个进程号来标识，其类型为 `pid_t`，进程号的范围：0~32767。

进程号总是唯一的，但进程号可以重用。当一个进程终止后，其进程号就可以再次使用。

在 Linux 系统中进程号由 0 开始：

进程号为 0 及 1 的进程由内核创建
进程号为 0 的进程通常是调度进程，常被称为交换进程(swapper)。
进程号为 1 的进程通常是 `init` 进程
除调度进程外，在 `linux` 下面所有的进程都由进程 `init` 进程直接或者间接创建

进程号(PID)：标识进程的一个非负整型数。

父进程号(PPID)：任何进程(除 `init` 进程)都是由另一个进程创建，该进程称为被创建进程的父进程，对应的进程号称为父进程号(PPID)。

进程组号(PGID)：进程组是一个或多个进程的集合。他们之间相互关联，进程组可以接收同一终端的各种信号，关联的进程有一个进程组号(PGID)。

Linux 操作系统提供了三个获得进程号的函数 `getpid()`、`getppid()`、`getpgid()`

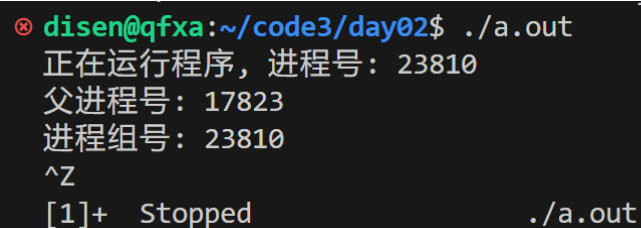
```
#include <sys/types.h>
#include <unistd.h>

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
```

如：

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

int main(int argc, char const *argv[])
{
    printf("正在运行程序，进程号：%d\n", getpid());
    printf("父进程号：%d\n", getppid());
    printf("进程组号：%d\n", getpgid(0));
    while (1)
        ;
    return 0;
}
```



```
disen@qfxa:~/code3/day02$ ./a.out
正在运行程序，进程号：23810
父进程号：17823
进程组号：23810
^Z
[1]+  Stopped                  ./a.out
```

程序运行之后是死循环，无法停止程序的运行，则按ctrl+z或ctrl+c停止。

或者通过 `kill -9 进程号` 杀死进程。

3.2.2 fork 函数

fork函数是创建子进程的函数。

在 Linux 环境下，创建进程的主要方法是调用以下两个函数：

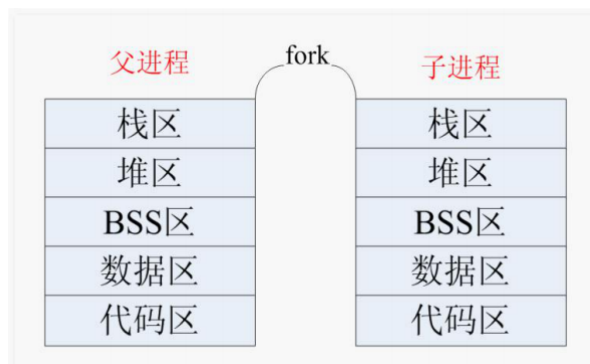
```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);    独立空间
pid_t vfork(void);   共享父进程空间
```

成功：子进程中返回 0，父进程中返回子进程 ID；失败：返回-1。

使用 fork 函数得到的子进程是父进程的一个复制品，它从父进程处继承了整个进程的地址空间。**地址空间**：包括进程上下文、进程堆栈、打开的文件描述符、信号控制设定、进程优先级、进程组号等。**子进程所独有的**只有它的**进程号、计时器等**。因此，使用 fork 函数的代价是很大的。

fork 函数执行结果：



【特别说明】BSS (Block Started by Symbol) 区段是一种用于存储未初始化全局变量和静态变量的内存区域。在程序加载时，BSS区的内存会被系统初始化为零或空值。

如：

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

int main()
{
    int n = 100;
    int total = 0;
    // 父进程完成 给定的整数以内的3的倍数累加和
    // 子进程完成 给定的整数以内的5的倍数累加和
    int pid = fork();
    if (pid < 0)
    {
        perror("fork");
        return 1;
    }
}
```

```

}
else if (pid == 0)
{
    // 子进程
    printf("%d 进程的父进程 %d\n", getpid(), getppid());
    while (n >= 5)
    {
        if (n % 5 == 0)
        {
            total += n;
        }
        n--;
    }
    printf("5的倍数的累加和: %d\n", total);
}
else
{
    // 主进程
    printf("%d 进程的父进程 %d\n", getpid(), getppid());
    while (n >= 3)
    {
        if (n % 3 == 0)
        {
            total += n;
        }
        n--;
    }
    printf("3的倍数的累加和: %d\n", total);
}
printf("%d 进程-----over-----\n", getpid());
return 0;
}

```

```

● disen@qfxa:~/code3/day02$ ./a.out
24226 进程的父进程 17823
3的倍数的累加和: 1683
24226 进程-----over-----
24227 进程的父进程 24226
5的倍数的累加和: 1050
24227 进程-----over-----

```

如：验证子进程继承父进程的缓冲区

```

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

int main(int argc, char const *argv[])
{
    printf("进程ID: %d fork before\n", getpid()); // 向缓冲区写数据
    int pid = fork();
    if (pid == 0)
    {
        printf("in son process\n");
    }
}

```

```

}
else if (pid > 0)
{
    printf("in father process\n");
}
return 0;
}

```

```

● disen@qfxa:~/code3/day02$ ./a.out
进程ID: 24633 fork before
in father process
in son process
● disen@qfxa:~/code3/day02$ ./a.out > a.txt
● disen@qfxa:~/code3/day02$ cat a.txt
进程ID: 24712 fork before
in father process
进程ID: 24712 fork before
in son process

```

调用 fork 函数后，父进程打开的文件描述符都被复制到子进程中。在重定向父进程的标准输出时，子进程的标准输出也被重定向。

write 函数是系统调用，不带缓冲

3.2.3 进程的挂起

进程在一定的时间内没有任何动作，称为进程的挂起。

```

#include <unistd.h>
unsigned int sleep(unsigned int sec);

```

进程挂起指定的秒数，直到指定的时间用完或收到信号才解除挂起。

若进程挂起到 sec 指定的时间则返回 0，若有信号中断则返回剩余秒数。

【注意】进程挂起指定的秒数后程序并不会立即执行，系统只是将此进程切换到就绪态。

如：

```

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

int main()
{
    char buf[32] = "";
    time_t start, end;
    time(&start);
    sprintf(buf, "%d runing... %ld", getpid(), start);
    write(STDOUT_FILENO, buf, 32);
    sleep(10);
    time(&end);
    long delta = end - start;
    printf("\n%d over...delta is %ld \n", getpid(), delta);
}

```

```
    return 0;
}
```

```
● disen@qfxa:~/code3/day02$ ./a.out
25240 runing... 1692091970
25240 over...detla is 10
```

3.2.4 进程的等待

父子进程有时需要简单的进程间同步，如父进程等待子进程的结束。

Linux 下提供了以下两个等待函数 `wait()`、`waitpid()`

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

3.2.4.1 wait函数

```
pid_t wait(int *status);
```

功能说明：

等待子进程终止，如果子进程终止了，此函数会回收子进程的资源。

调用 `wait` 函数的进程会挂起，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，失败原因存于 `errno` 中。

取出子进程的退出信息：

`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。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。

【注意】此 `status` 是个 `wait` 的参数指向的整型变量。

如：

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

```

int main(int argc, char const *argv[])
{
    int pid = fork();
    if (pid < 0)
    {
        perror("fork");
        return 1;
    }
    else if (pid == 0)
    {
        write(4, "123", 4);
        sleep(5);
        printf("%d 子进程结束...", getpid());
        return 10;
    }
    else
    {
        int status;
        int pid = wait(&status); // 等待子进程停止
        printf("\nd父进程中，等到了 %d 子进程结束\n", getpid(), pid);

        if (WIFEXITED(status))
        { // 子进程是正常退出
            printf("status(子进程返回的结果): %d\n", WEXITSTATUS(status));
        }
    }

    return 0;
}

```

```

● disen@qfxa:~/code3/day02$ ./a.out
25547 子进程结束...
25546父进程中，等到了 25547 子进程结束
status: 10

```

```

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

int main(int argc, char const *argv[])
{
    int pid = fork();
    if (pid < 0)
    {
        perror("fork");
        return 1;
    }
    else if (pid == 0)
    {
        // write(1, "123", 4);
        int n = 10 / 0;
    }
}

```



```

        sleep(5);
        printf("%d 子进程结束...", getpid());
        return 10;
    }
    else
    {
        int status;
        int pid = wait(&status); // 等待子进程停止
        printf("\n%d父进程中，等到了 %d 子进程结束\n", getpid(), pid);

        if (WIFEXITED(status))
        { // 子进程是正常退出
            printf("status(子进程返回的结果): %d\n", WEXITSTATUS(status));
        }
        else
        {
            printf("子进程存在异常\n");
        }
    }
}

return 0;
}

```

26037父进程中，等到了 26038 子进程结束
子进程存在异常

3.2.4.2 waitpid 函数

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

功能：等待子进程终止，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

参数 pid 的值有以下几种类型：

- 1) pid>0: 等待进程 ID 等于 pid 的子进程。
- 2) pid=0 等待同一个进程组中的任何子进程，如果子进程已经加入了别的进程组，waitpid 不会等待它。
- 3) pid=-1: 等待任一子进程，此时 waitpid 和 wait 作用一样。
- 4) pid<-1: 等待指定进程组中的任何子进程，这个进程组的 ID 等于 pid 的绝对值

status 参数中包含子进程退出时的状态信息。

options 参数能进一步控制 waitpid 的操作：

- 0: 同 wait，阻塞父进程，等待子进程退出
- WNOHANG: 没有任何已经结束的子进程，则立即返回
- WUNTRACED: 如果子进程暂停了则此函数马上返回，并且不予以理会子进程的结束状态。
【跟踪调试，很少用到】

返回值:

- 1) 成功, 返回状态改变了的子进程的进程号; 如果设置了选项 `WNOHANG` 并且 `pid` 指定的进程存在则返回 0。
- 2) 出错: 返回 -1。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`。

如:

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

int main(int argc, char const *argv[])
{
    int pid = fork();
    if (pid < 0)
    {
        perror("fork");
        return 1;
    }
    else if (pid == 0)
    {
        printf("%d 子进程准备休息5秒----\n", getpid());
        sleep(5);
        printf("%d 子进程结束...\n", getpid());
        return 10;
    }
    else
    {
        int status;
        int pid2 = waitpid(pid, &status, WUNTRACED);
        printf("\n%d父进程中, 等到了 %d 子进程结束\n", getpid(), pid);

        if (WIFEXITED(status))
        { // 子进程是正常退出
            printf("status(子进程返回的结果): %d\n", WEXITSTATUS(status));
        }
        else
        {
            printf("子进程存在异常\n");
        }
    }
}

return 0;
}
```

```

● disen@qfxa:~/code3/day02$ ./a.out
26548 子进程准备休息5秒----
26548 子进程结束...

26547父进程中，等到了 26548 子进程结束
status(子进程返回的结果): 10

```

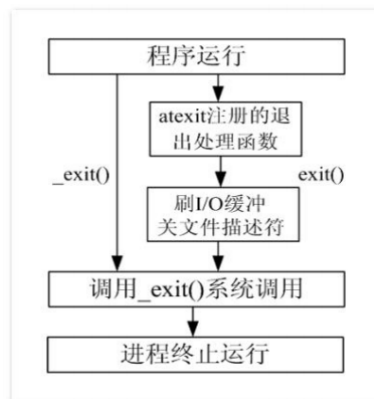
3.2.5 进程的终止

```

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

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

```



3.2.6 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。

3.2.7 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。



3.2.8 进程的替换

3.2.9 system 函数

3.2.10 练习

实现 system 函数