

目录

第一章：shell 脚本.....	4
1.1shell 概述.....	4
1.2shell 语法.....	7
1.2.1 shell 脚本的定义与执行.....	7
1.2.2 变量.....	8
1.2.3 条件测试语句.....	14
1.2.4 控制语句.....	19
1.2.5 函数.....	26
第二章：系统调用.....	28
2.1 系统编程概述.....	28
2.2 系统调用概述.....	28
2.3 系统调用 I/O 函数.....	30
2.3.1 文件描述符.....	30
2.3.2 open 函数.....	30
2.3.3 close 函数.....	32
2.3.4 write 函数.....	33
2.3.5 read 函数.....	33
2.3.6 remove 库函数.....	34
2.4 系统调用与库函数.....	34
2.4.1 不需要调用系统调用.....	34
2.4.2 需要调用系统调用.....	34
2.4.3 库函数与系统调用的关系：.....	35
2.4.4 练习：实现 cp 命令.....	36
第三章：进程.....	36
3.1 进程概述.....	36
3.1.1 进程的定义.....	36
3.1.2 进程的状态及转换.....	37
3.1.3 进程控制块.....	38
3.2 进程控制.....	38
3.2.1 进程号.....	38
3.2.2 进程的创建 fork 函数.....	40

3.2.3 进程的挂起.....	46
3.2.4 进程的等待.....	46
3.2.5 进程的终止.....	51
3.2.6 进程退出清理.....	53
3.2.7 进程的创建 vfork 函数.....	54
3.2.8 进程的替换.....	57
3.2.9 system 函数.....	62
3.2.10 练习.....	64
第四章：信号.....	65
4.1 进程间通信概述.....	65
4.2 信号.....	66
4.2.1 概述.....	66
4.2.2 信号的基本操作.....	68
4.2.3 可重入函数.....	75
4.2.3 信号集.....	76
4.2.4 信号阻塞集(屏蔽集、掩码).....	80
第五章：管道、命名管道.....	82
5.1 管道概述.....	82
5.2 无名管道的创建 pipe 函数.....	83
5.3 文件描述符概述.....	86
5.4 文件描述符的复制.....	87
5.4.1 dup 函数.....	87
5.4.2 dup2 函数 重定向.....	88
5.5 命名管道(FIFO).....	91
第六章：消息队列.....	102
6.1 消息队列概述.....	103
6.1.1ftok 函数.....	103
6.2 消息队列的操作.....	104
6.2.1 创建消息队列.....	104
6.2.2 发送消息.....	105
6.2.3 接收消息.....	106

6.2.4 消息队列的控制.....	107
第七章：共享内存.....	111
7.1 共享内存概述.....	111
7.2 共享内存操作.....	112
7.2.1 获得一个共享存储标识符.....	112
7.2.2 共享内存映射(attach).....	113
7.2.3 解除共享内存映射(detach).....	114
7.2.4 共享内存控制.....	114
第八章：线程.....	118
8.1 线程概述.....	118
8.1.1 线程的概念.....	118
8.1.2 线程和进程的比较.....	119
8.1.3 多线程的用处.....	120
8.2 线程的基本操作.....	120
8.2.1 线程的创建.....	121
8.2.2 线程等待.....	124
8.2.3 线程分离.....	126
8.2.4 线程退出.....	127
第九章：多任务互斥和同步.....	141
9.1 互斥和同步概述.....	141
9.2 互斥锁.....	141
9.2.1 互斥锁的概念.....	141
9.2.2 初始化互斥锁.....	142
9.2.3 互斥锁上锁.....	142
9.2.4 互斥锁上锁 2.....	143
9.2.5 互斥锁解锁.....	143
9.2.6 销毁互斥锁.....	144
9.3 信号量.....	145
9.3.1 信号量的概念.....	145
9.3.2 信号量的操作.....	147
9.3.3 有名信号量（扩展）.....	152

第一章：shell 脚本

1.1 shell 概述

shell 的两层含义：

既是一种应用程序,又是一种程序设计语言

作为应用程序：

交互式地解释、执行用户输入的命令，将用户的操作翻译成机器可以识别的语言，完成相应功能

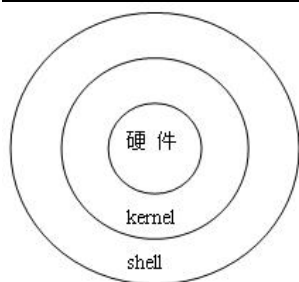
称之为 shell 命令解析器

shell 是用户和 Linux 内核之间的接口程序

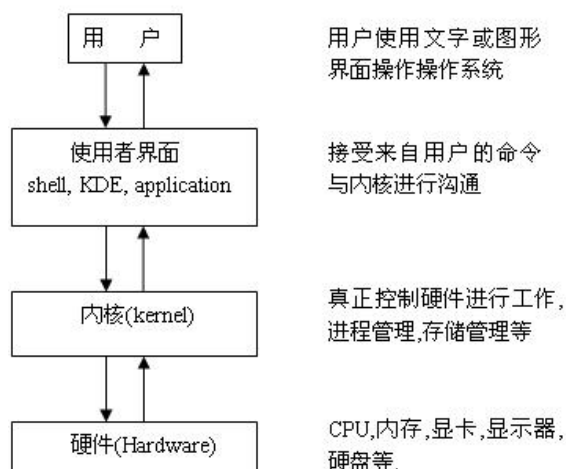
用户在提示符下输入的命令都由 shell 先解释然后传给 Linux 核心

它调用了系统核心的大部分功能来执行程序、并以并行的方式协调各个程序的运行

Linux 系统中提供了好几种不同的 shell 命令解释器，如 sh、ash、bash 等。一般默认使用 bash 作为默认的解释器。我们后面编写的 shell 脚本，都是由上述 shell 命令解释器解释执行的。



shell 是用户跟内核通信几种方式的一种



作为程序设计语言:

它定义了各种变量和参数，并提供了许多在高级语言中才具有的控制结构，包括循环和分支

完成类似于 windows 下批处理操作，简化我们对系统的管理与应用程序的部署**称之为 shell 脚本**

我们学过的 c/c++ 等语言，属于编译性语言（编写完成后需要使用编译器完成编译、汇编、链接等过程变为二进制代码方可执行）

shell 脚本是一种脚本语言，我们只需使用任意文本编辑器，按照语法编写相应程序，增加可执行权限，即可在安装 shell 命令解释器的环境下执行

shell 脚本主要用于：

帮助开发人员或系统管理员将复杂而又反复的操作放在一个文件中，通过简单的一步执行操作完成相应任务，从而解放他们的负担

shell 应用举例：

shell 应用举例：

1、《linux 常用命令_练习.txt》

我们前面完成了这个练习，步骤很多，其实我们只需要将所有操作写入一个文件——cmd.sh(名字跟后缀可任取，为了便于区分我们一般写为*.sh 形式)

然后：

```
chmod +x cmd.sh
```

```
./cmd.sh 直接执行即可
```

2、假设我们要完成以下任务：

判断用户家目录下（~）下面有没有一个叫 test 的文件夹

如果没有，提示按 y 创建并进入此文件夹，按 n 退出

如果有，直接进入，提示请输入一个字符串，并按此字符串创建一个文件，如果此文件已存在，提示重新输入，重复三次自动退出，不存在创建完毕，退出

简单的进行命令堆积无法完成以上任务，这就需要学习相应的 shell 脚本语法规则了

shell 脚本大体可以分为两类：

系统进行调用

这类脚本无需用户调用，系统会在合适的时候调用，如：/etc/profile、~/.bashrc 等

/etc/profile

此文件为系统的每个用户设置环境信息,当用户第一次登录时,该文件被执行,系统的公共环境变量在这里设置

开始自启动的程序，一般也在这里设置

~/.bashrc

用户自己的家目录中的.bashrc

登录时会自动调用，打开任意终端时也会自动调用

这个文件一般设置与个人用户有关的环境变量，如交叉编译器的路径等等

用户编写，需要手动调用的

例如我们上面编写的脚本都属于此类

无论是系统调用的还是需要我们自己调用的，其语法规则都一样

1.2 shell 语法

1.2.1 shell 脚本的定义与执行

1、定义以开头：#!/bin/bash

#!/用来声明脚本由什么 shell 解释，否则使用默认 shell

2、单个"#"号代表注释当前行

3、执行：

chmod + x test.sh ./test.sh 增加可执行权限后执行

bash test.sh 直接指定使用 bash 解释 test.sh

. test.sh(source test.sh) 使用当前 shell 读取解释 test.sh

三种执行脚本的方式不同点：

./和 bash 执行过程基本一致，后者明确指定 bash 解释器去执行脚本，脚本中#!指定的解释器不起作用

前者首先检测#!，使用#!指定的 shell，如果没有使用默认的 shell

用./和 bash 去执行会在后台启动一个新的 shell 去执行脚本

用.去执行脚本不会启动新的 shell,直接由当前的 shell 去解释执行脚本。

例：1.sh

```
#!/bin/bash  
clear  
echo "this is the first shell script"
```

注意：如果是在 windows 通过 notepad++ 编辑脚本程序

需要用 vi 打开脚本，在最后一行模式下执行

:set ff=unix

1.2.2 变量

1.2.2.1 自定义变量

定义变量

变量名=变量值

如：num=10

引用变量

\$变量名

如：i=\$num 把变量 num 的值付给变量 i

显示变量

使用 echo 命令可以显示单个变量取值

```
echo $num
```

清除变量

使用 unset 命令清除变量

```
unset varname
```

变量的其它用法:

```
read string
```

从键盘输入一个字符串付给变量 string

```
readonly var=100
```

定义一个只读变量,只能在定义时初始化,以后不能改变,不能被清除。

```
export var=300
```

使用 export 说明的变量, 会被导出为环境变量, 其它 shell 均可使用

注意: 此时必须使用 source 2_var.sh 才可以生效

注意事项:

1、变量名只能包含英文字母下划线, 不能以数字开头

```
1_num=10 错误
```

```
num_1=20 正确
```

2、等号两边不能直接接空格符, 若变量中本身就包含了空格, 则整个字符串都要用双引号、或单引号括起来; 双引号内的特殊字符可以保有变量特性, 但是单引号内的特殊字符则仅为一般字符。

```
name=aa bb //错误
```

```
name="aa bb" //正确
```

```
echo "$name is me" //输出: aa bb is me
```

```
echo '$name is me'           //输出: $name is me
```

例 2: 2_var.sh

```
#!/bin/bash
echo "this is the var test shell script "
name="edu"
echo "$name is me"
echo '$name is me'

echo "please input a string"
read string
echo "the string of your input is $string"

readonly var=1000
#var=200

export public_var=300
```

1.2.2.2 环境变量

shell 在开始执行时就已经定义了一些和系统的工作环境有关的变量，我们在 shell 中可以直接使用\$name 引用

定义：

一般在~/.bashrc 或/etc/profile 文件中（系统自动调用的脚本）使用 export 设置，允许[用户后来更改](#)

VARNAME=value ; export VARNAME

传统上，所有环境变量均为大写

显示环境变量

使用 env 命令可以查看所有环境变量。

清除环境变量

使用 unset 命令清除环境变量

常见环境变量：

HOME：用于保存注册目录的完全路径名。

PATH：用于保存用冒号分隔的目录路径名，shell 将按 PATH 变量中给出的顺序搜索这些目录，找到的第一个与命令名称一致的可执行文件将被执行。

```
PATH=$HOME/bin:/bin:/usr/bin;export PATH
```

HOSTNAME：主机名

SHELL：默认的 shell 命令解析器

LOGNAME：此变量保存登录名

PWD：当前工作目录的绝对路径名

.....

例：3_export.sh

```
#!/bin/bash
echo "You are welcome to use bash"
echo "Current work dirctory is $PWD"
echo "the host name is $HOSTNAME"
echo "your home dir $HOME"
echo "Your shell is $SHELL"
```

1.2.2.3 预设变量

\$#：传给 shell 脚本参数的数量

\$*: 传给 shell 脚本参数的内容

\$1、\$2、\$3、...、\$9: 运行脚本时传递给其的参数，用空格隔开

\$?: 命令执行后返回的状态

"\$?"用于检查上一个命令执行是否正确(在 Linux 中，命令退出状态为 0 表示该命令正确执行，任何非 0 值表示命令出错)。

\$0: 当前执行的进程名

\$\$: 当前进程的进程号

"\$\$"变量最常见的用途是用作临时文件的名称以保证临时文件不会重复

例：4_\$.sh

```
#!/bin/bash
echo "your shell script name is $0"
echo "the params of your input is $*"
echo "the num of your input params is $#"
```

echo "the params is \$1 \$2 \$3 \$4"


```
ls
echo "the cmd state is $?"
cd /root
echo "the cmd state is $?"
```



```
echo "process id is $$"
```

1.2.2.4 脚本变量的特殊用法："" ``'\0 {}

"" (双引号)：包含的变量会被解释

" (单引号) : 包含的变量会当做字符串解释

` `(数字键 1 左面的反引号): 反引号中的内容作为系统命令, 并执行其内容, 可以替换输出为一个变量

```
$ echo "today is `date` "
```

today is 2012 年 07 月 29 日星期日 12:55:21 CST

\ 转义字符:

同 c 语言 \n \t \r \a 等 echo 命令需加-e 转义

(命令序列):

由子 shell 来完成, 不影响当前 shell 中的变量

{ 命令序列 }:

在当前 shell 中执行, 会影响当前变量

例: 5_var_spe.sh 注意: "{" 、 "}" 前后有一空格

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

```
name=teacher
```

```
string1="good moring $name"
```

```
string2='good moring $name'
```

```
echo $string1
```

```
echo $string2
```

```
echo "today is `date` "
```

```
echo 'today is `date` '
```

```
echo -e "this \n is\ta\ntest"
```

```
( name=student;echo "1 $name" )
```

```
echo 1:$name
```

```
{ name=student; echo "2 $name"; }
```

echo 2:\$name

1.2.3 条件测试语句

在写 shell 脚本时，经常遇到的问题就是判断字符串是否相等，可能还要检查文件状态或进行数字测试，只有这些测试完成才能做下一步动作

test 命令：用于测试字符串、文件状态和数字

test 命令有两种格式：

test condition 或[condition]

使用方括号时，要注意在条件两边加上空格

shell 脚本中的条件测试如下：

文件测试、字符串测试、数字测试、复合测试

测试语句一般与后面讲的条件语句联合使用

1.2.3.1 文件

文件测试：测试文件状态的条件表达式

-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		

例：6_test_file.sh

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

```
test -e /dev/qaz
```

```
echo $?
```

```
test -e /home
```

echo \$?

test -d /home

echo \$?

test -f /home

echo \$?

mkdir test_sh

chmod 500 test_sh

[-r test_sh]

echo \$?

[-w test_sh]

echo \$?

[-x test_sh]

echo \$?

[-s test_sh]

echo \$?

[-c /dev/console]

echo \$?

[-b /dev/sda]

echo \$?

[-L /dev/stdin]

echo \$?

1.2.3.2 字符串

字符串测试

```
test str_operator "str"
```

```
test "str1" str_operator "str2"
```

```
[ str_operator "str" ]
```

```
[ "str1" str_operator "str2" ]
```

其中 str_operator 可以是:

= 两个字符串相等 != 两个字符串不相等

-z 空串 -n 非空串

例: 7_test_string.sh

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

```
test -z $yn
```

```
echo $?
```

```
echo "please input a y/n"
```

```
read yn
```

```
[ -z "$yn" ]
```

```
echo 1:$?
```

```
[ $yn = "y" ]
```

echo 2:\$?

1.2.3.3 数字

测试数值格式如下:

```
test num1 num_operator num2
```

```
[ num1 num_operator num2 ]
```

num_operator 可以是:

-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

-le 数 1 小于等于数 2

-lt 数 1 小于数 2

	英文单词	shell比较符
相等	equal	-eq
不相等	not equal	-ne
大于	greater than	-gt
大于等于	greater equal	-ge
小于等于	less equal	-le
小于	less than	-lt

例: 8_test_num.sh

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

```
echo "please input a num(1-9)"
```

read num

[\$num -eq 5]

echo \$?

[\$num -ne 5]

echo \$?

[\$num -gt 5]

echo \$?

[\$num -ge 5]

echo \$?

[\$num -le 5]

echo \$?

[\$num -lt 5]

echo \$?

1.2.3.4 复合测试

命令执行控制:

&&:

command1 && command2

&&左边命令 (command1) 执行成功(即返回 0) shell 才执行&&右边的命令 (command2)

||

command1 || command2

||左边的命令 (command1) 未执行成功(即返回非 0) shell 才执行||右边的命令 (command2)

例:

test -e /home && test -d /home && echo "true"

test 2 -lt 3 && test 5 -gt 3 && echo "equal"

test "aaa" = "aaa" || echo "not equal" && echo "equal"

多重条件判定

-a	(and)两状况同时成立! <code>test -r file -a -x file</code> file 同时具有 r 与 x 权限时, 才为 true.
-o	(or)两状况任何一个成立! <code>test -r file -o -x file</code> file 具有 r 或 x 权限时, 就传回 true.
!	相反状态 <code>test ! -x file</code> 当 file 不具有 x 时, 回传 true.

1.2.4 控制语句

if case for while until break

1.2.4.1 if 控制语句

格式一:

`if [条件 1]; then`

 执行第一段程序

`else`

 执行第二段程序

`fi`

格式二:

`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

例：9_if_then.sh

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

```
echo "Press y to continue"
```

```
read yn
```

```
if [ $yn = "y" ]; then
```

```
    echo "script is running..."
```

```
else
```

```
    echo "stopped!"
```

```
fi
```

1.2.4.2 case 控制语句

```
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
```

例：10_case1.sh

```
#!/bin/bash

echo "This script will print your choice"

case "$1" in
    "one")
        echo "your choice is one"
        ;;
    "two")
        echo "your choice is two"
        ;;
    "three")
        echo "Your choice is three"
        ;;
    *)
        echo "Error Please try again!"
        exit 1
        ;;
esac
```

例：10_case2.sh

```
#!/bin/bash

echo "Please input your choice:"
```

read choice

```
case "$choice" in
    Y | yes | Yes | YES)
        echo "It's right"
        ;;
    N* | n*)
        echo "It's wrong"
        ;;
    *)
        exit 1
esac
```

1.2.4.3for 控制语句

形式一：

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

初始值：变量在循环中的起始值

限制值：当变量值在这个限制范围内时，就继续进行循环

执行步阶：每作一次循环时，变量的变化量

declare 是 bash 的一个内建命令，可以用来声明 shell 变量、设置变量的属性。
declare 也可以写作 typeset。

declare -i s 代表强制把 s 变量当做 int 型参数运算。

例：11_for1.sh

```
#!/bin/bash

declare -i sum

for (( i=1; i<=100; i=i+1 ))
do
    sum=sum+i
done

echo "The result is $sum"
```

形式二：

```
for var in con1 con2 con3 ...
do
    程序段
done
```

第一次循环时，\$var 的内容为 con1

第二次循环时，\$var 的内容为 con2

第三次循环时，\$var 的内容为 con3

.....

例：11_for2.sh

```
#!/bin/bash

for i in 1 2 3 4 5 6 7 8 9
do
    echo $i
done
```

例：11_for3.sh

```
#!/bin/bash

for name in `ls`
do
    if [ -f $name ];then
        echo "$name is file"
    elif [ -d $name ];then
        echo "$name is directory"
    else
        echo "^_^"
    fi
done
```

1.2.4.4 while 控制语句

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

当 condition 成立的时候进入 while 循环，直到 condition 不成立时才退出循环。

例：12_while.sh

```
#!/bin/bash

declare -i i
declare -i s
while [ "$i" != "101" ]
do
    s+=i;
```

```
i=i+1;  
  
done  
  
echo "The count is $s"
```

1.2.4.5until 控制语句

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

这种方式与 while 恰恰相反，当 condition 成立的时候退出循环，否则继续循环。

例：13_until.sh

```
#!/bin/bash  
  
declare -i i  
declare -i s  
until [ "$i" = "101" ]  
do  
    s+=i;  
    i=i+1;  
done  
  
echo "The count is $s"
```

1.2.4.6break continue

break

break 命令允许跳出循环。

break 通常在进行一些处理后退出循环或 case 语句

continue

continue 命令类似于 break 命令

只有一点重要差别，它不会跳出循环，只是跳过这个循环步骤

1.2.5 函数

有些脚本段间互相重复，如果能只写一次代码块而在任何地方都能引用那就提高了代码的可重用性。

shell 允许将一组命令集或语句形成一个可用块，这些块称为 shell 函数。

定义函数的两种格式：

格式一：

```
函数名 () {  
    命令 ...  
}
```

格式二：

```
function 函数名 () {  
    命令 ...  
}
```

函数可以放在同一个文件中作为一段代码，也可以放在只包含函数的单独文件中

所有函数在使用前必须定义，必须将函数放在脚本开始部分，直至 shell 解释器首次发现它时，才可以使用

调用函数的格式为：

函数名 param1 param2.....

使用参数同在一般脚本中使用特殊变量

\$1, \$2 ...\$9 一样

函数可以使用 **return** 提前结束并带回返回值

return 从函数中返回，用最后状态命令决定返回值。

return 0 无错误返回

return 1 有错误返回

例：14_function.sh

```
#!/bin/bash
is_it_directory()
{
    if [ $# -lt 1 ]; then
        echo "I need an argument"
        return 1
    fi
    if [ ! -d $1 ]; then
        return 2
    else
        return 3
    fi
}
echo -n "enter destination directory:"
read direct
is_it_directory $direct
echo "the result is $?"
```

第二章：系统调用

2.1 系统编程概述

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。

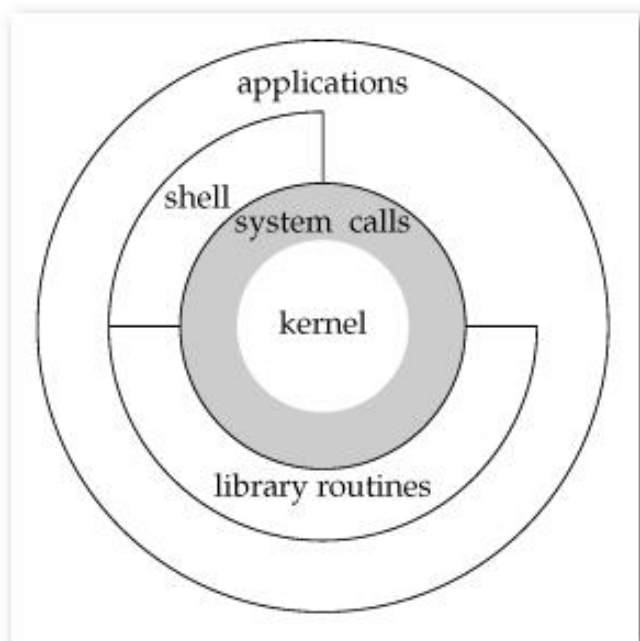
什么是 Linux 系统编程

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，对系统资源进行访问。

学会了 C 语言再知道一些使用系统调用的方法，就可以进行 Linux 系统编程了。

2.2 系统调用概述

类 UNIX 系统的软件层次



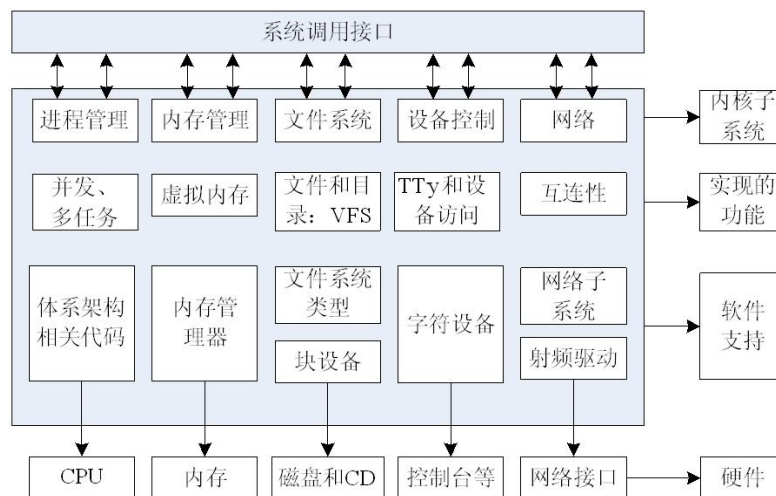
系统调用是操作系统提供给用户程序的一组“特殊”函数接口。

Linux 的不同版本提供了两三百个系统调用。

用户程序可以通过这组接口获得操作系统（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。



系统调用按照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，用一个负的返回值来表明错误，返回一个 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。

2.3 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。

如：open、close、write 、 read、 ioctl

2.3.1 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。

`#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`

程序运行起来后这三个文件描述符是默认打开的。

2.3.2 open 函数

打开一个文件

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

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

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

当文件存在时使用：

```
int open(const char *pathname, int flags);
```

当文件不存在时使用：

```
int open(const char *pathname, int flags, mode_t mode);
```

参数：

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	
取值	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	如果同时指定了 O_CREAT，且文件已经存在，则出错
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		
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

2.3.3 close 函数

关闭一个文件

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

```
int close(int fd);
```

参数:

fd 是调用 open 打开文件返回的文件描述符。

返回值:

成功返回 0。

失败返回-1，可以利用 perror 去查看原因。

2.3.4 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：

fd：文件描述符。

addr：数据首地址。

count：写入数据的字节个数。

返回值：

成功返回实际写入数据的字节个数。

失败返回-1，可以利用 perror 去查看原因。

2.3.5 read 函数

把指定数目的数据读到内存

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

```
ssize_t read(int fd, void *addr, size_t count);
```

参数：

fd：文件描述符。

addr：内存首地址。

count：读取的字节个数。

返回值：

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。

2.3.6 remove 库函数

删除文件

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

```
int remove(const char *pathname);
```

参数:

pathname : 文件的路名+文件名。

返回值:

成功返回 0。

失败返回-1，可以利用 perror 去查看原因。

2.4 系统调用与库函数

库函数由两类函数组成

2.4.1 不需要调用系统调用

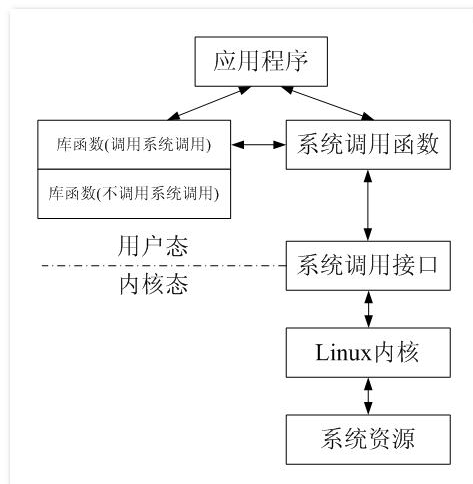
不需要切换到内核空间即可完成函数全部功能，并且将结果反馈给应用程序，如 strcpy、bzero 等字符串操作函数。

2.4.2 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。

2.4.3 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。

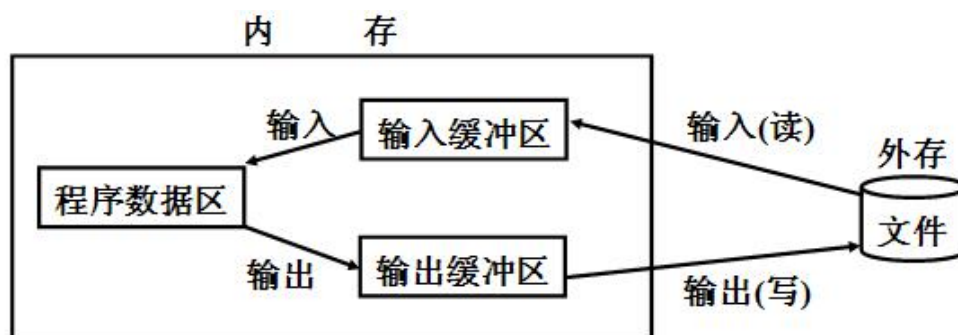


系统调用是需要时间的，程序中频繁的使用系统调用会降低程序的运行效率。

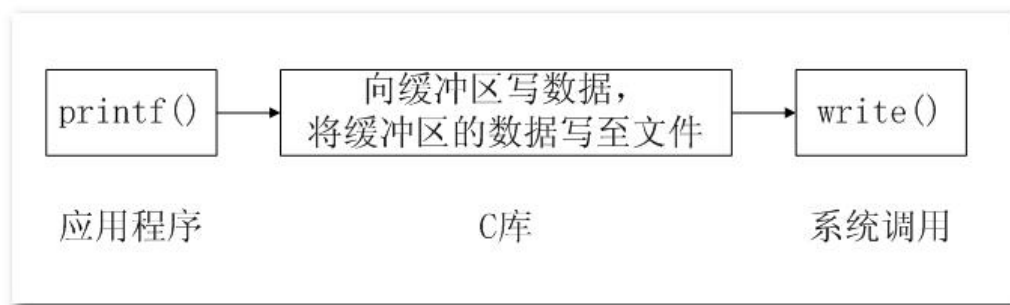
当运行内核代码时，CPU 工作在内核态，在系统调用发生前需要保存用户态的栈和内存环境，然后转入内核态工作。

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



2.4.4 练习：实现 cp 命令

目标：

使用系统调用实现 cp 命令。

原理：

使用系统调用 open 打开文件，使用 read 从文件读数据，使用 write 向文件写数据。

传给可执行程序参数个数存放在 main 函数的 argc 中，参数首地址存放在指针数组 argv 中。

第三章：进程

3.1 进程概述

3.1.1 进程的定义

程序：

程序是存放在存储介质上的一个可执行文件。

进程：

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。

程序是静态的，进程是动态的：

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：

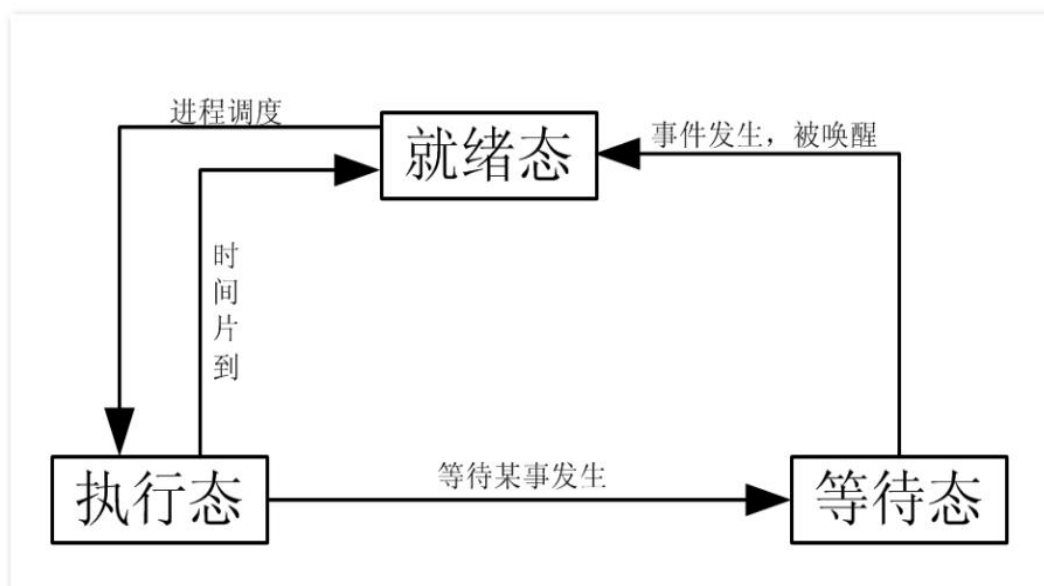
进程已经具备执行的一切条件，正在等待分配 CPU 的处理时间。

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。



进程三种状态的转换关系

3.1.3 进程控制块

OS 是根据 PCB 来对并发执行的进程进行控制和管理。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 & 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。

调度数据

进程的状态、标志、优先级、调度策略等。

时间数据

创建该进程的时间、在用户态的运行时间、在内核态的运行时间等。

文件系统数据

umask 掩码、文件描述符表等。

内存数据、进程上下文、进程标识（进程号）

...

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了。

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()。

需要包含头文件：

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

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

pid_t getpid(void)

功能：获取本进程号(PID)

pid_t getppid(void)

功能：获取调用此函数的进程的父进程号(PPID)

pid_t getpid(pid_t pid)

功能：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

例：01_pid.c

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
    pid_t pid, ppid, pgid;

    pid = getpid();
    printf("pid = %d\n", pid);

    ppid = getppid();
    printf("ppid = %d\n", ppid);

    pgid = getpgid(pid);
    printf("pgid = %d\n", pgid);
    return 0;
}
```

3.2.2 进程的创建 fork 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);
```



```
pid_t vfork(void);
```

创建一个新进程

```
pid_t fork(void)
```

功能:

fork()函数用于从一个已存在的进程中创建一个新进程，新进程称为子进程，原进程称为父进程。

返回值:

成功: 子进程中返回 0，父进程中返回子进程 ID。

失败: 返回-1。

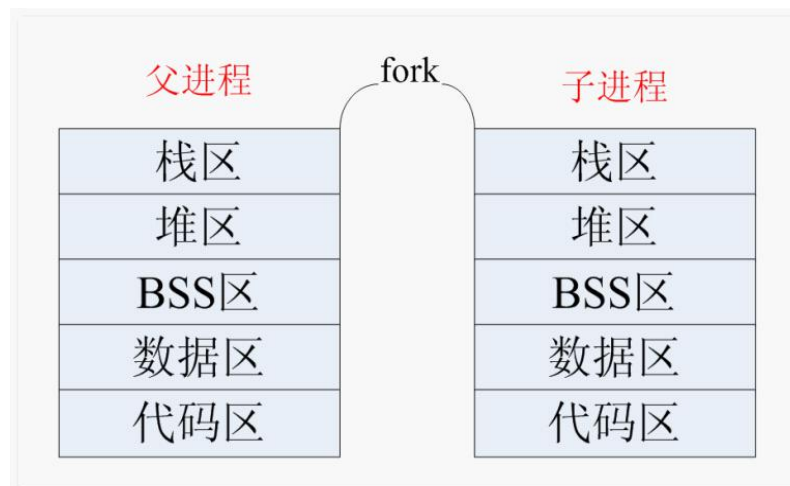
使用 fork 函数得到的子进程是父进程的一个复制品，它从父进程处继承了整个进程的地址空间。

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:



例: 02_fork_1.c **创建一个子进程实现多任务**

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

```
#include <stdlib.h>

#include <unistd.h>

int main(int argc, char *argv[])
{
    pid_t pid;

    pid=fork();
    if(pid<0)
        perror("fork");
    if(pid==0)
    {
        while(1)
        {
            printf("this is son process\n");
            sleep(1);
        }
    }
    else
    {
        while(1)
        {
            printf("this is father process\n");
            sleep(1);
        }
    }
    return 0;
}
```

例：02_fork_2.c 验证父子进程分别有各自独立的地址空间

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int var=10;

int main(int argc, char *argv[])
{
    pid_t pid;
    int num=9;
    pid=fork();
    if(pid<0)
    {
        perror("fork");
    }
    if(pid==0)
    {
        var++;
        num++;
        printf("in son process var=%d,num=%d\n", var, num);
    }
    else
    {
        sleep(1);
        printf("in father process var=%d,num=%d\n", var, num);
    }
    printf("common code area\n");
    return 0;
}
```

从 02_fork_2.c 程序可以看出，子进程对变量所做的改变并不影响父进程中该变量的值，说明父子进程各自拥有自己的地址空间。

一般来说，在 fork 之后是父进程先执行还是子进程先执行是不确定的。这取决于内核所使用的调度算法。

如要求父子进程之间相互同步，则要求某种形式的进程间通信。

例：02_fork_3.c 验证子进程继承父进程的缓冲区

提示：

标准 I/O 提供三种类型的缓冲：

全缓冲：(大小不定)

在填满标准 I/O 缓冲区后，才进行实际的 I/O 操作。术语冲洗缓冲区的意思是进行标准 I/O 写操作。

行缓冲：(大小不定)

在遇到换行符时，标准 I/O 库执行 I/O 操作。这种情况允许我们一次输入一个字符，但只有写了一行后才进行实际的 I/O 操作。

不带缓冲

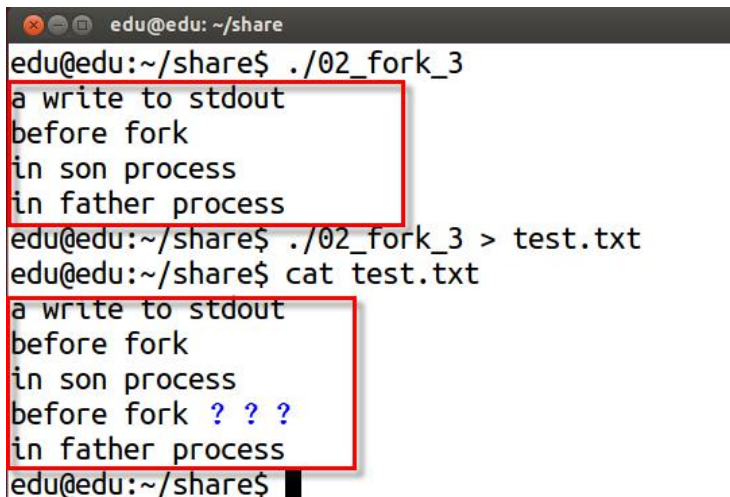
代码：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
int main(int argc, char *argv[])
{
    pid_t pid;
    int length = 0;
    char buf[] = "a write to stdout\n";

    length = write(1, buf, strlen(buf));
    if(length != strlen(buf))
    {
        printf("write error\n");
    }
}
```

```
printf("before fork\n");
pid=fork();
if(pid<0)
{
    perror("fork");
}
else if(pid==0)
{
    printf("in son process\n");
}
else
{
    sleep(1);
    printf("in father process\n");
}
return 0;
}
```

运行方式：



```
edu@edu: ~/share
edu@edu:~/share$ ./02_fork_3
a write to stdout
before fork
in son process
in father process
edu@edu:~/share$ ./02_fork_3 > test.txt
edu@edu:~/share$ cat test.txt
a write to stdout
before fork
in son process
before fork ???
in father process
edu@edu:~/share$
```

提示：

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。

标准 I/O 库是带缓冲的，当以交互方式运行程序时，标准 I/O 库是行缓冲的，否则它是全缓冲的。

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);
```

功能:

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。

返回值:

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。

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()。

需要包含头文件:

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

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

3.2.4.1 wait 函数

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

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。

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。

例: 03_wait.c

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

```
#include <sys/types.h>
#include <sys/wait.h>
int main(int argc, char *argv[])
{
    pid_t pid;

    pid=fork();
    if(pid<0)
        perror("fork");
    if(pid==0)
    {
        int i = 0;
        for(i=0;i<5;i++)
        {
            printf("this is son process\n");
            sleep(1);
        }
        _exit(2);
    }
    else
    {
        int status = 0;

        wait(&status);
        if(WIFEXITED(status)!=0)
        {
            printf("son process return %d\n", WEXITSTATUS(status));
        }
        printf("this is father process\n");
    }
    return 0;
}
```

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：**`pid > 0`：**

等待进程 ID 等于 `pid` 的子进程。

`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。

`pid = -1`：

等待任一子进程，此时 `waitpid` 和 `wait` 作用一样。

`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:

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。

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。

例：03_waitpid.c

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

int main(int argc, char *argv[])
{
    pid_t pid;

    pid=fork();
    if(pid < 0)
        perror("fork");
    if(pid == 0)
    {
        int i = 0;
        for(i=0;i<5;i++)
        {
            printf("this is son process\n");
            sleep(1);
        }
    }
}
```

```
        }
        _exit(2);
    }
    else
    {
        waitpid(pid, NULL, 0);
        printf("this is father process\n");
    }
    return 0;
}
```

特殊进程

僵尸进程 (Zombie Process)

进程已运行结束，但进程的占用的资源未被回收，这样的进程称为僵尸进程。

子进程已运行结束，父进程未调用 wait 或 waitpid 函数回收子进程的资源是子进程变为僵尸进程的原因。

孤儿进程 (Orphan Process)

父进程运行结束，但子进程未运行结束的子进程。

守护进程 (精灵进程) (Daemon process)

守护进程是个特殊的孤儿进程，这种进程脱离终端，在后台运行。

3.2.5 进程的终止

在 linux 下可以通过以下方式结束正在运行的进程：

```
void exit(int value);
```

```
void _exit(int value);
```

3.2.5.1 exit 函数

结束进程执行

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

```
void exit(int value)
```

参数:

status: 返回给父进程的参数(低 8 位有效)。

3.2.5.2 _exit 函数

结束进程执行

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

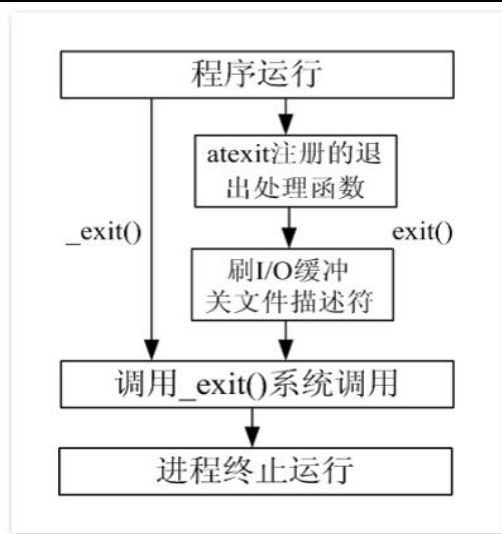
```
void _exit(int value)
```

参数:

status: 返回给父进程的参数(低 8 位有效)。

exit 和 _exit 函数的区别:

exit 为库函数, 而 _exit 为系统调用



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));
```

功能：

注册进程正常结束前调用的函数，进程退出执行注册函数。

参数：

`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。

例：04_atexit.c

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

```
void clear_fun1(void)
{
    printf("perform clear fun1 \n");
}

void clear_fun2(void)
{
    printf("perform clear fun2 \n");
}

void clear_fun3(void)
{
    printf("perform clear fun3 \n");
}

int main(int argc, char *argv[])
{
    atexit(clear_fun1);
    atexit(clear_fun2);
    atexit(clear_fun3);
    printf("process exit 3 sec later!!!\n");
    sleep(3);
    return 0;
}
```

3.2.7 进程的创建 vfork 函数

pid_t vfork(void)

功能:

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。

返回值:

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：

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。

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。

例：05_vfork_1.c 验证 vfork 创建子进程先执行，父进程挂起

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
    pid_t pid;

    pid = vfork();
    if(pid<0)
        perror("vfork");
    if(pid==0)
    {
        int i = 0;
        for(i=0;i<5;i++)
        {
            printf("this is son process\n");
            sleep(1);
        }
        _exit(0);
    }
    else
    {
        while(1)
```

```
        {  
            printf("this is father process\n");  
            sleep(1);  
        }  
    }  
    return 0;  
}
```

例：05_vfork_2.c 验证 **vfork** 创建子进程与父进程共用一个地址空间

```
#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
  
int var = 10;  
int main(int argc, char *argv[])  
{  
    pid_t pid;  
    int num = 9;  
  
    pid = vfork();  
    if(pid < 0)  
    {  
        perror("vfork");  
    }  
    if(pid == 0)  
    {  
        var++;  
        num++;  
        printf("in son process var=%d,num=%d\n", var, num);  
        _exit(0);  
    }  
    else  
    {  
        printf("in father process var=%d,num=%d\n", var, num);  
    }  
    return 0;  
}
```


}



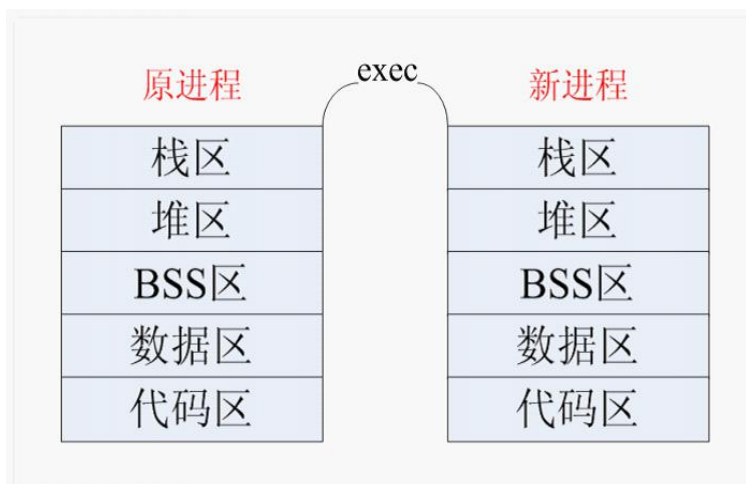
3.2.8 进程的替换

进程的替换

exec 函数族，是由六个 exec 函数组成的。

- 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 函数族

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

```
int execl(const char *pathname,  
          const char *arg0, ...,  
          NULL);
```

```
int execlp(const char *filename,  
          const char *arg0, ...,  
          NULL);
```

```
int execlx(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。

参数地址列表

`char *arg0, char *arg1, ..., char *argn, NULL`

`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。`execle` 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。

例：06_test.c

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

int main(int argc, char *argv[])
```

```
{  
    printf("USER=%s\n",getenv("USER"));  
    printf("GONGSI=%s\n",getenv("GONGSI"));  
    return 0;  
}
```

例：06_exec1.c

```
#include <stdio.h>  
#include <unistd.h>  
  
int main(int argc, char *argv[])  
{  
    execl("/bin/ls", "ls", "-a", "-l", "-h", NULL);  
    perror("execl");  
    return 0;  
}
```

例：06_execlp.c

```
#include <stdio.h>  
#include <unistd.h>  
  
int main(int argc, char *argv[])  
{  
    execlp("ls", "ls", "-a", "-l", "-h", NULL);  
    perror("execlp");  
    return 0;  
}
```

例：06_execl.e.c

```
#include <stdio.h>  
#include <unistd.h>  
  
int main(int argc, char *argv[])  
{  
    char *env[]={"USER=ME", "GONGSI=QF", NULL};
```

```
    execl("./test", "test", NULL, env);
    perror("execl");
    return 0;
}
```

例：06_execv.c

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

int main(int argc, char *argv[])
{
    char *arg[]={"ls", "-a", "-l", "-h", NULL};

    execv("/bin/ls", arg);
    perror("execv");
    return 0;
}
```

例：06_execvp.c

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

int main(int argc, char *argv[])
{
    char *arg[]={"ls", "-a", "-l", "-h", NULL};

    execvp("ls", arg);
    perror("execvp");
    return 0;
}
```

例：06_execve.c

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

int main(int argc, char *argv[])
{
    char *arg[]={"test", NULL};
    char *env[]={"USER=ME", "GONGSI=QF", NULL};

    execve("./test", arg, env);
    perror("execve");
    return 0;
}
```

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

...

3.2.9 system 函数

```
#include <stdlib.h>

int system(const char *command);
```

功能：

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。

参数：

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。

例：07_system.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

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

    status = system("ls -alh");
    if(WIFEXITED(status))
    {
        printf("the exit status is %d \n", status);
    }
    else
    {
        printf("abnormamal exit\n");
    }
    return 0;
}
```

3.2.10 练习

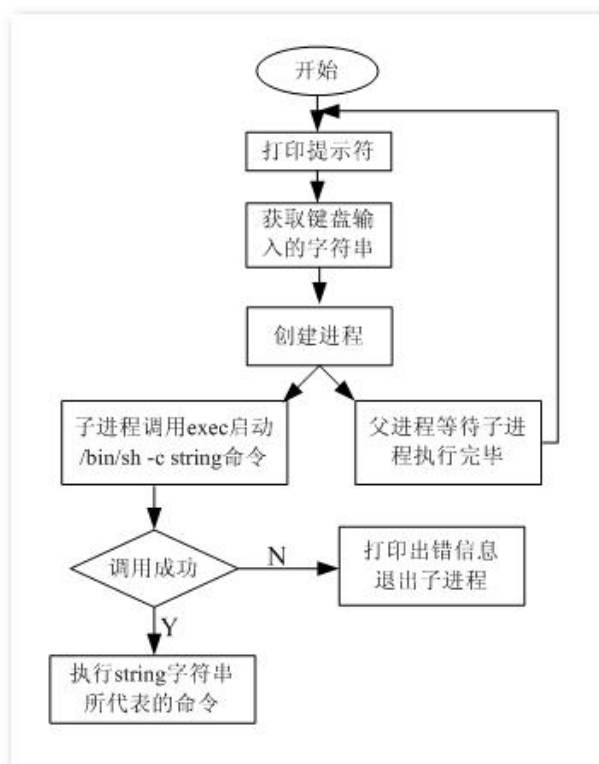
题目：实现 system 函数

提示：

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

父进程等待子进程退出

程序流程图：



第四章：信号

4.1 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 进程间通信 (IPC) 由以下几个部分发展而来

最初的 UNIX 进程间通信

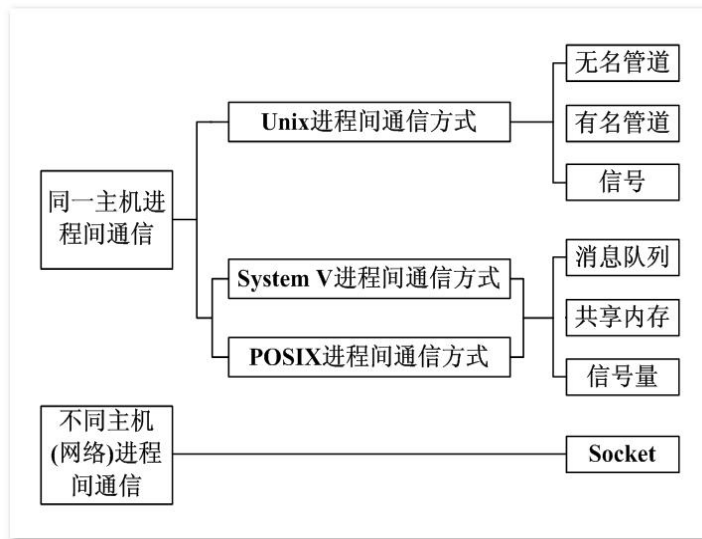
SYSTEM V 进程间通信

POSIX 进程间通信 (POSIX: Portable Operating System interface 可移植操作系统接口)

Socket 进程间通信

Linux 把优势都继承了下来并形成了自己的 IPC

Linux 操作系统支持的主要进程间通信的通信机制



4.2 信号

4.2.1 概述

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。

信号可以导致一个正在运行的进程被另一个正在运行的异步进程中中断，转而处理某一个突发事件。

信号是一种异步通信方式。

进程不必等待信号的到达，进程也不知道信号什么时候到达。

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。

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。

信号名定义路径：

/usr/include/i386-linux-gnu/bits/signum.h

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$
```

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。

例如：

终端上按“Ctrl+c”组合键通常产生中断信号 SIGINT、终端上按“Ctrl+\”键通常产生中断信号 SIGQUIT、终端上按“Ctrl+z”键通常产生中断信号 SIGSTOP。

2、硬件异常将产生信号。

除数为 0，无效的内存访问等。这些情况通常由硬件检测到，并通知内核，然后内核产生适当的信号发送给相应的进程。

3、软件异常将产生信号。

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。

注意：接收信号进程和发送信号进程的所有者必须相同，或发送信号进程的所有者必须是超级用户。

5、运行 kill 命令将发送信号。

此程序实际上是使用 kill 函数来发送信号。也常用此命令终止一个失控的后台进程。

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

对大多数信号来说，系统默认动作是用来终止该进程。

2、忽略此信号

接收到此信号后没有任何动作。

3、执行自定义信号处理函数

用用户定义的信号处理函数处理该信号。

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。

4.2.2 信号的基本操作

4.2.2.1 kill 函数

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

```
#include <signal.h>
```

```
int kill(pid_t pid, int signum);
```

功能：

给指定进程发送信号。

参数:

pid: 详见下页

signum: 信号的编号

返回值:

成功返回 0, 失败返回 -1。

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。

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。

例: 01_kill.c 父进程给子进程发送信号

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
int main(int argc, char *argv[])
{
    pid_t pid;
    pid = fork();
    if(pid<0)
        perror("fork");
    if(pid==0)
    {
        int i = 0;
        for(i=0; i<5; i++)
        {
            printf("in son process\n");
            sleep(1);
        }
    }
}
```

```
        }  
    }  
    else  
    {  
        printf("in father process\n");  
        sleep(2);  
        printf("kill sub process now \n");  
        kill(pid, SIGINT);  
    }  
    return 0;  
}
```

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。

4.2.2.2 alarm 函数

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

```
unsigned int alarm(unsigned int seconds);
```

功能：

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。

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。

例：02_alarm.c

```
#include <stdio.h>  
#include <unistd.h>  
int main(int argc, char *argv[])  
{  
    int seconds = 0;  
    seconds = alarm(5);  
    printf("seconds = %d\n", seconds);  
    sleep(2);  
}
```

```
seconds = alarm(5);  
printf("seconds = %d\n", seconds);  
while(1);  
return 0;  
}
```

4.2.2.3 raise 函数

```
#include <signal.h>
```

```
int raise(int signum);
```

功能:

给调用进程本身送一个信号。

参数:

signum: 信号的编号。

返回值:

成功返回 0, 失败返回 -1。

例: 03_raise.c

```
#include <signal.h>  
#include <stdio.h>  
  
int main(int argc, char *argv[])  
{  
    printf("in raise function\n");  
    sleep(2);  
    raise(SIGALRM);  
    sleep(10);  
    return 0;  
}
```

4.2.2.4 abort 函数

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

```
void abort(void);
```

功能:

向进程发送一个 SIGABRT 信号，默认情况下进程会退出。

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。

4.2.2.5 pause 函数

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

```
int pause(void);
```

功能:

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。

返回值:

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

例：04_pause.c

```
#include <unistd.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    printf("in pause function\n");
    pause();
    return 0;
}
```


4.2.2.6 进程接收到信号后的处理方式

- 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

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);
```

功能：

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。

参数：

signum： 信号编号

handler 的取值：

忽略该信号：SIG_IGN

执行系统默认动作：SIG_DFL

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

例：05_signal_1.c 注册信号处理函数

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

```
#include <signal.h>
```

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

```
void signal_handler(int signo)
```

```
{
    if(signo==SIGINT)
        printf("recv SIGINT\n");
    if(signo==SIGQUIT)
        printf("recv SIGQUIT\n");
}

int main(int argc, char *argv[])
{
    printf("wait for SIGINT OR SIGQUIT\n");
    signal(SIGINT, signal_handler);
    signal(SIGQUIT, signal_handler);

    pause();
    pause();
    return 0;
}
```

例：05_signal_2.c 验证 signal 函数的返回值

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
typedef void (*sighandler_t)(int);

void fun1(int signo)
{
    printf("in fun1\n");
}

void fun2(int signo)
{
    printf("in fun2\n");
}

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

```

```
sighandler_t previous = NULL;

previous = signal(SIGINT,fun1);
if(previous == NULL)
{
    printf("return fun addr is NULL\n");
}
previous = signal(SIGINT,fun2);
if(previous == fun1)
{
    printf("return fun addr is fun1\n");
}
previous = signal(SIGQUIT,fun1);
if(previous == NULL)
{
    printf("return fun addr is NULL\n");
}
return 0;
}
```

4.2.3 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：

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

4.2.3 信号集

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。

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/i386-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

4.2.3.1 sigemptyset 函数

初始化一个空的信号集

```
#include <signal.h>
```

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

功能:

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。

参数:

`set`: 信号集标识的地址，以后操作此信号集，对 `set` 进行操作就可以了。

返回值:

成功返回 0，失败返回 -1。

4.2.3.2 sigfillset 函数

初始化一个满的信号集

```
#include <signal.h>
```

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

功能:

初始化信号集合 `set`，将信号集合设置为所有信号的集合。

参数:

信号集标识的地址，以后操作此信号集，对 `set` 进行操作就可以了。

返回值:

成功返回 0，失败返回 -1。

4.2.3.3 sigismember 函数

判断某个集合中是否有某个信号

```
#include <signal.h>
```

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

功能：

查询 signum 标识的信号是否在信号集合 set 之中。

参数：

set: 信号集标识符号的地址。

signum: 信号的编号。

返回值：

在信号集中返回 1，不在信号集中返回 0

错误，返回 -1

4.2.3.4 sigaddset 函数

向某个集合中添加一个信号

```
#include <signal.h>
```

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

功能：

将信号 signum 加入到信号集合 set 之中。

参数：

set: 信号集标识的地址。

signum: 信号的编号。

返回值：

成功返回 0，失败返回 -1。

4.2.3.5 sigdelset 函数

从某个信号集中删除一个信号

```
#include <signal.h>
```

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

功能:

将 `signum` 所标识的信号从信号集合 `set` 中删除。

参数:

`set`: 信号集标识的地址。

`signum`: 信号的编号。

返回值:

成功: 返回 0

失败: 返回 -1

例: 06_signal_set.c 创建一个空的信号集合, 向集合中添加信号, 判断集合中是否有这个信号

```
#include <signal.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    sigset_t set;
    int ret = 0;

    sigemptyset(&set);
    ret = sigismember(&set, SIGINT);
    if(ret == 0)
        printf("SIGINT is not a member of sigprocmask \nret = %d\n",
ret);

    sigaddset(&set, SIGINT);
    sigaddset(&set, SIGQUIT);
```

```
ret = sigismember(&set, SIGINT);
if(ret == 1)
    printf("SIGINT is a member of sigprocmask \nret = %d\n", ret);

return 0;
}
```

4.2.4 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。

4.2.4.1 sigprocmask 函数

创建一个阻塞集合

```
#include <signal.h>
```

```
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。

set： 要操作的信号集地址。

oldset： 保存原先信号集地址。

how：

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

注：若 set 为 NULL，则不改变信号阻塞集合，函数只把当前信号阻塞集合保存到 oldset 中。

返回值：

成功：返回 0

失败：返回 -1

例：06_sigprocmask.c

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

int main(int argc, char *argv[])
{
    sigset_t set;
    int i=0;

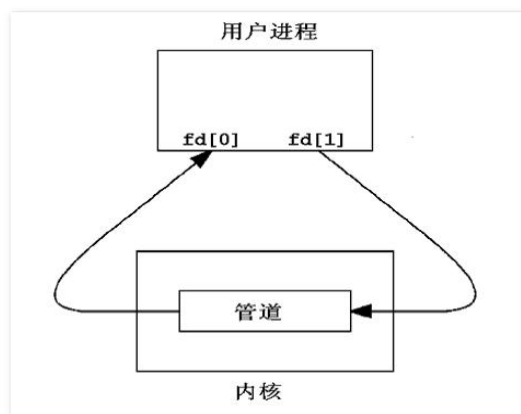
    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    while(1)
    {
        sigprocmask(SIG_BLOCK, &set, NULL);
        for(i=0; i<5; i++)
        {
            printf("SIGINT signal is blocked\n");
            sleep(1);
        }
        sigprocmask(SIG_UNBLOCK, &set, NULL);
        for(i=0; i<5; i++)
        {
            printf("SIGINT signal unblocked\n");
            sleep(1);
        }
    }
    return 0;
}
```

第五章：管道、命名管道

5.1 管道概述

管道(pipe)又称无名管道。

无名管道是一种特殊类型的文件，在应用层体现为两个打开的文件描述符。



管道是最古老的 UNIX IPC 方式，其特点是：

- 1、半双工，数据在同一时刻只能在一个方向上流动。
- 2、数据只能从管道的一端写入，从另一端读出。
- 3、写入管道中的数据遵循先入先出的规则。
- 4、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。
- 5、管道不是普通的文件，不属于某个文件系统，其只存在于内存中。
- 6、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。
- 7、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。

8、管道没有名字，只能在具有公共祖先的进程之间使用

5.2 无名管道的创建 pipe 函数

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

```
int pipe(int filedes[2]);
```

功能：经由参数 `filedes` 返回两个文件描述符

参数：

`filedes` 为 `int` 型数组的首地址，其存放了管道的文件描述符 `fd[0]`、`fd[1]`。

`filedes[0]` 为读而打开，`filedes[1]` 为写而打开管道，`filedes[0]` 的输出是 `filedes[1]` 的输入。

返回值：

成功：返回 0

失败：返回 -1

例：01_pipe_1.c 创建管道，父子进程通过无名管道通信

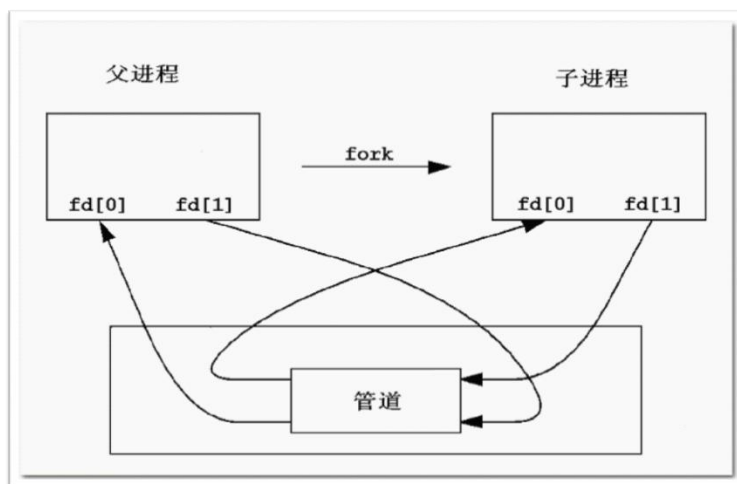
```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    int fd_pipe[2];
    char buf[] = "hello world";
    pid_t pid;

    if (pipe(fd_pipe) < 0)
        perror("pipe");
    pid = fork();
    if (pid < 0)
    {
        perror("fork");
```

```
        exit(-1);
    }
    if (pid == 0)
    {
        write(fd_pipe[1], buf, strlen(buf));
        _exit(0);
    }
    else
    {
        wait(NULL);
        memset(buf, 0, sizeof(buf));
        read(fd_pipe[0], buf, sizeof(buf));
        printf("buf=[%s]\n", buf);
    }
    return 0;
}
```

父子进程通过管道实现数据的传输



注意:

利用无名管道实现进程间的通信，都是父进程创建无名管道，然后再创建子进程，子进程继承父进程的无名管道的文件描述符，然后父子进程通过读写无名管道实现通信

从管道中读数据的特点

- 1、默认用 `read` 函数从管道中读数据是阻塞的。
- 2、调用 `write` 函数向管道里写数据，当缓冲区已满时 `write` 也会阻塞。
- 3、通信过程中，读端口全部关闭后，写进程向管道内写数据时，写进程会（收到 `SIGPIPE` 信号）退出。

编程时可通过 `fcntl` 函数设置文件的阻塞特性。

设置为阻塞：

```
fcntl(fd, F_SETFL, 0);
```

设置为非阻塞：

```
fcntl(fd, F_SETFL, O_NONBLOCK);
```

例：01_pipe_2.c 验证无名管道文件读写的阻塞和非阻塞

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

int main(int argc, char *argv[])
{
    int fd_pipe[2];
    char buf[] = "hello world";
    pid_t pid;

    if (pipe(fd_pipe) < 0)
        perror("pipe");
    pid = fork();
    if (pid < 0)
    {
```

```
        perror("fork");
        exit(-1);
    }
    if (pid == 0)
    {
        sleep(3);
        write(fd_pipe[1], buf, strlen(buf));
        _exit(0);
    }
    else
    {
        //fcntl(fd_pipe[0], F_SETFL, O_NONBLOCK);
        fcntl(fd_pipe[0], F_SETFL, 0);
        while(1)
        {
            memset(buf, 0, sizeof(buf));
            read(fd_pipe[0], buf, sizeof(buf));
            printf("buf=[%s]\n", buf);
            sleep(1);
        }
    }
    return 0;
}
```

5.3 文件描述符概述

文件描述符是非负整数，是文件的标识。

用户使用文件描述符（file descriptor）来访问文件。

利用 open 打开一个文件时，内核会返回一个文件描述符。

每个进程都有一张文件描述符的表，进程刚被创建时，标准输入、标准输出、标准错误输出设备文件被打开，对应的文件描述符 0、1、2 记录在表中。

在进程中打开其他文件时，系统会返回文件描述符表中最小可用的文件描述符，并将此文件描述符记录在表中。

注意：

Linux 中一个进程最多只能打开 NR_OPEN_DEFAULT
(即 1024) 个文件，故当文件不再使用时应及时调用 close 函数关闭文件。

5.4 文件描述符的复制

dup 和 dup2 是两个非常有用的系统调用，都是用来复制一个文件的描述符，使新的文件描述符也标识旧的文件描述符所标识的文件。

```
int dup(int oldfd);
```

```
int dup2(int oldfd, int newfd);
```

dup 和 dup2 经常用来重定向进程的 stdin、stdout 和 stderr。

回顾：ls > test.txt

5.4.1 dup 函数

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

```
int dup(int oldfd);
```

功能：

复制 oldfd 文件描述符，并分配一个新的文件描述符，新的文件描述符是调用进程文件描述符表中最小可用的文件描述符。

参数：

要复制的文件描述符 oldfd。

返回值：

成功：新文件描述符。

失败：返回 -1，错误代码存于 errno 中。

例：02_dup.c 重定向

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

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

int main(void)
{
    int fd1;
    int fd2;

    fd1 = open("test", O_CREAT|O_WRONLY, S_IRWXU);
    if (fd1 < 0)
    {
        perror("open");
        exit(-1);
    }
    close(1);
    fd2 = dup(fd1);
    printf("fd2=%d\n", fd2);
    return 0;
}
```

5.4.2 dup2 函数 重定向

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

```
int dup2(int oldfd, int newfd)
```

功能:

复制一份打开的文件描述符 `oldfd`，并分配新的文件描述符 `newfd`，`newfd` 也标识 `oldfd` 所标识的文件。

注意:

`newfd` 是小于文件描述符最大允许值的非负整数，如果 `newfd` 是一个已经打开的文件描述符，则首先关闭该文件，然后再复制。

参数:

要复制的文件描述符 `oldfd`

分配的新的文件描述符 `newfd`

返回值：

成功：返回 newfd

失败：返回-1，错误代码存于 errno 中

例：03_dup2.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int main(int argc, char *argv[])
{
    int fd1;
    int fd2;

    fd2 = dup(1);//save stdout
    printf("new:fd2=%d\n",fd2);

    fd1 = open("test", O_CREAT|O_RDWR, S_IRWXU);
    close(1);
    dup(fd1);//1---> test
    printf("hello world\n");

    close(1);
    dup(fd2);//reset 1---> stdout
    printf("i love you \n");
    return 0;
}
```

复制文件描述符后新旧文件描述符的特点

使用 dup 或 dup2 复制文件描述符后，新文件描述符和旧文件描述符指向同一个文件，共享文件锁定、读写位置和各项权限。

当关闭新的文件描述符时，通过旧文件描述符仍可操作文件。

当关闭旧的文件描述时，通过新的文件描述符仍可操作文件。

exec 前后文件描述符的特点

close_on_exec 标志决定了文件描述符在执行 exec 后文件描述符是否可用。

文件描述符的 close_on_exec 标志默认是关闭的，即文件描述符在执行 exec 后文件描述符是可用的。

若没有设置 close_on_exec 标志位，进程中打开的文件描述符，及其相关的设置在 exec 后不变，可供新启动的程序使用。

设置 close_on_exec 标志位的方法：

```
int flags;

flags = fcntl(fd, F_GETFD); // 获得标志

flags |= FD_CLOEXEC;        // 打开标志位

flags &= ~FD_CLOEXEC;       // 关闭标志位

fcntl(fd, F_SETFD, flags); // 设置标志
```

练习：

题目：借用外部命令，实现计算器功能

提示：

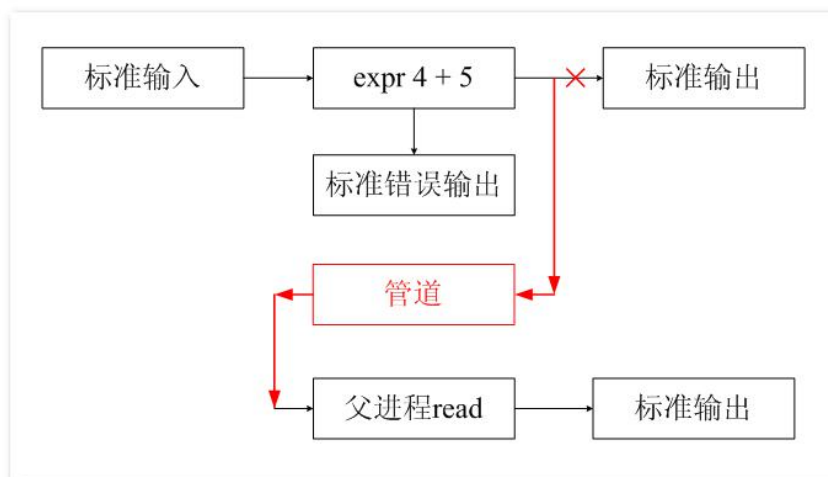
expr 是个外部命令，它向标准输出打印运算结果。

创建一个管道以便让 expr 4 + 5 的输出到管道中

子进程 exec 执行 expr 4 + 5 命令之前重定向“标准输出”到“管道写端”。

父进程从管道读端读取数据，并显示运算结果

外部命令结果信息输出至管道：



5.5 命名管道(FIFO)

命名管道(FIFO)和管道(pipe)基本相同,但也有一些显著的不同, **其特点是:**

- 1、半双工,数据在同一时刻只能在一个方向上流动。
- 2、写入 FIFO 中的数据遵循先入先出的规则。
- 3、FIFO 所传送的数据是无格式的,这要求 FIFO 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。
- 4、FIFO 在文件系统中作为一个特殊的文件而存在,但 FIFO 中的内容却存放在内存中。
- 5、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。
- 6、从 FIFO 读数据是一次性操作,数据一旦被读,它就从 FIFO 中被抛弃,释放空间以便写更多的数据。
- 7、当使用 FIFO 的进程退出后,FIFO 文件将继续保存在文件系统中以便以后使用。
- 8、FIFO 有名字,不相关的进程可以通过打开命名管道进行通信。

操作 FIFO 文件时的特点

系统调用的 I/O 函数都可以作用于 FIFO,如 open、close、read、write 等。

打开 FIFO 时,非阻塞标志(O_NONBLOCK)产生下列影响:

特点一:

不指定 O_NONBLOCK(即 open 没有位或 O_NONBLOCK)

- 1、open 以只读方式打开 FIFO 时，要阻塞到某个进程为写而打开此 FIFO
- 2、open 以只写方式打开 FIFO 时，要阻塞到某个进程为读而打开此 FIFO。

例:04_fifo_read_1.c 验证阻塞方式，open 的阻塞效果

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int main(int argc, char *argv[])
{
    int fd;
    int ret;
    ret = mkfifo("my_fifo", S_IRUSR|S_IWUSR);
    if(ret != 0)
    {
        perror("mkfifo");
    }
    printf("before open\n");
    fd = open("my_fifo", O_RDONLY);
    if(fd < 0)
    {
        perror("open fifo");
    }
    printf("after open\n");
    return 0;
}
```

04_fifo_write_1.c 验证阻塞方式，open 的阻塞效果

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

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

int main(int argc, char *argv[])
{
    int fd;
    int ret;

    ret = mkfifo("my_fifo", S_IRUSR|S_IWUSR);
    if(ret != 0)
    {
        perror("mkfifo");
    }
    printf("before open\n");
    fd = open("my_fifo", O_WRONLY);
    if(fd < 0)
    {
        perror("open fifo");
    }
    printf("after open\n");
    return 0;
}
```

3、open 以只读、只写方式打开 FIFO 时会阻塞，调用 read 函数从 FIFO 里读数据时 read 也会阻塞。

例:04_fifo_read_2.c 以阻塞模式，验证 read 函数也会阻塞

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

int main(int argc, char *argv[])
{
    int fd;
```

```
int ret;
char recv[100];

ret = mkfifo("my_fifo", S_IRUSR|S_IWUSR);
if(ret != 0)
{
    perror("mkfifo");
}
printf("before open\n");
fd = open("my_fifo", O_RDONLY);
if(fd < 0)
{
    perror("open fifo");
}
printf("after open\n");
printf("before read\n");
bzero(recv, sizeof(recv));
read(fd, recv, sizeof(recv));
printf("read from my_fifo buf=[%s]\n",recv);
return 0;
}
```

例:04_fifo_write_2.c 以阻塞模式, 验证 read 函数也会阻塞

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

int main(int argc, char *argv[])
{
    int fd;
    char send[100] = "Hello I love you";

    printf("before open\n");
```

```
fd = open("my_fifo", O_WRONLY);
if(fd < 0)
{
    perror("open fifo");
}
printf("after open\n");
printf("before write\n");
sleep(5);
write(fd, send, strlen(send));
printf("write to my_fifo buf=%s\n",send);
return 0;
}
```

4、通信过程中若写进程先退出了，则调用 read 函数从 FIFO 里读数据时不阻塞；若写进程又重新运行，则调用 read 函数从 FIFO 里读数据时又恢复阻塞。

例:04_fifo_read_3.c 阻塞方式打开命名管道，验证 写进程退出，会导致 read 不阻塞

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

int main(int argc, char *argv[])
{
    int fd;
    int ret;

    ret = mkfifo("my_fifo", S_IRUSR|S_IWUSR);
    if(ret != 0)
    {
        perror("mkfifo");
    }
    fd = open("my_fifo", O_RDONLY);
    if(fd < 0)
    {
```

```
        perror("open fifo");
    }
    while(1)
    {
        char recv[100];

        bzero(recv, sizeof(recv));
        read(fd, recv, sizeof(recv));
        printf("read from my_fifo buf=[%s]\n",recv);
        sleep(1);
    }
    return 0;
}
```

例：04_fifo_write_3.c 阻塞方式打开命名管道，验证 写进程退出，会导致 read 不阻塞

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

int main(int argc, char *argv[])
{
    int fd;
    char send[100] = "Hello I love you";

    fd = open("my_fifo", O_WRONLY);
    if(fd < 0)
    {
        perror("open fifo");
    }
    write(fd, send, strlen(send));
    printf("write to my_fifo buf=%s\n",send);
}
```



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

5、通信过程中，读进程退出后，写进程向命名管道内写数据时，写进程也会（收到 SIGPIPE 信号）退出。

例:04_fifo_read_4.c 阻塞方式打开命名管道，验证 读进程结束后，写进程再向管道写数据写进程会收到信号退出

```
#include <stdio.h>  
#include <string.h>  
#include <unistd.h>  
#include <sys/types.h>  
#include <sys/stat.h>  
#include <fcntl.h>  
  
int main(int argc, char *argv[])  
{  
    int fd;  
    int ret;  
  
    ret = mkfifo("my_fifo", S_IRUSR|S_IWUSR);  
    if(ret != 0)  
    {  
        perror("mkfifo");  
    }  
    fd = open("my_fifo", O_RDONLY);  
    if(fd < 0)  
    {  
        perror("open fifo");  
    }  
    while(1)  
    {  
        char recv[100];  
  
        bzero(recv, sizeof(recv));
```

```
        read(fd, recv, sizeof(recv));
        printf("read from my_fifo buf=[%s]\n",recv);
        sleep(1);
    }
    return 0;
}
```

例：04_fifo_write_4.c 阻塞方式打开命名管道，验证 读进程结束后，写进程再向管道写数据写进程会收到信号退出

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

int main(int argc, char *argv[])
{
    int fd;
    char send[100] = "Hello I love you";

    fd = open("my_fifo", O_WRONLY);
    if(fd < 0)
    {
        perror("open fifo");
    }
    while(1)
    {
        write(fd, send, strlen(send));
        printf("write to my_fifo buf=%s\n",send);
        sleep(1);
    }
    return 0;
}
```

6、调用 write 函数向 FIFO 里写数据，当缓冲区已满时 write 也会阻塞。

特点二：

指定 O_NONBLOCK(即 open 位或 O_NONBLOCK)

- 1、先以只读方式打开：如果没有进程已经为写而打开一个 FIFO，只读 open 成功，并且 open 不阻塞。
- 2、先以只写方式打开：如果没有进程已经为读而打开一个 FIFO，只写 open 将出错返回-1。
- 3、read、write 读写命名管道中读数据时不阻塞。
- 4、通信过程中，读进程退出后，写进程向命名管道内写数据时，写进程也会（收到 SIGPIPE 信号）退出。

例：04_fifo_read_5.c 非阻塞方式打开命名管道，验证 open 和 read 都不阻塞

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

int main(int argc, char *argv[])
{
    int fd;
    int ret;

    ret = mkfifo("my_fifo", S_IRUSR|S_IWUSR);
    if(ret != 0)
```

```
{
    perror("mkfifo");
}
fd = open("my_fifo", O_RDONLY|O_NONBLOCK);
if(fd < 0)
{
    perror("open fifo");
}
while(1)
{
    char recv[100];

    bzero(recv, sizeof(recv));
    read(fd, recv, sizeof(recv));
    printf("read from my_fifo buf=[%s]\n",recv);
    sleep(1);
}
return 0;
}
```

例：04_fifo_write_5.c 非阻塞方式打开命名管道，验证 open 和 read 都不阻塞

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

int main(int argc, char *argv[])
{
    int fd;
    char send[100] = "Hello I love you";

    fd = open("my_fifo", O_WRONLY|O_NONBLOCK);
    if(fd<0)
    {
        perror("open fifo");
    }
}
```

```
write(fd, send, strlen(send));  
printf("write to my_fifo buf=%s\n", send);  
while(1);  
return 0;  
}
```

注意:

open 函数以可读可写方式打开 FIFO 文件时的特点:

- 1、open 不阻塞。
- 2、调用 read 函数从 FIFO 里读数据时 read 会阻塞。
- 3、调用 write 函数向 FIFO 里写数据，当缓冲区已满时 write 也会阻塞。

练习:

题目: 实现单机聊天程序

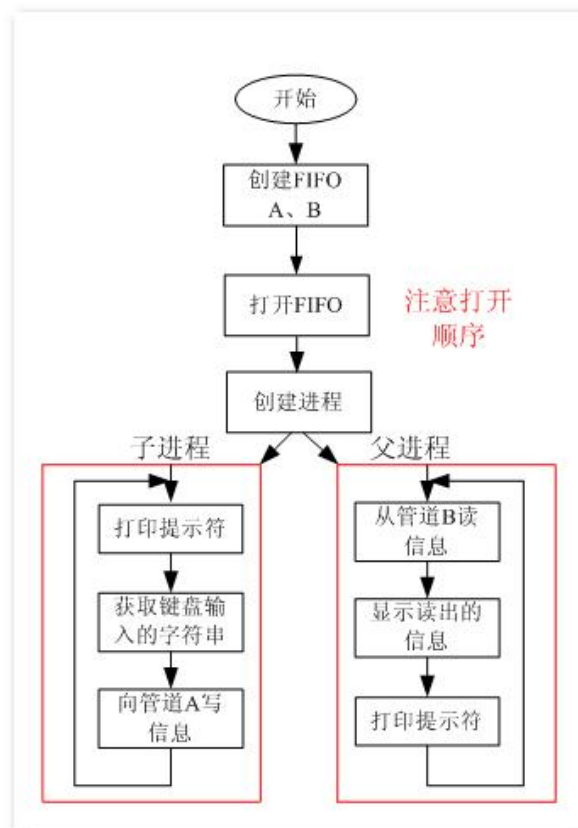
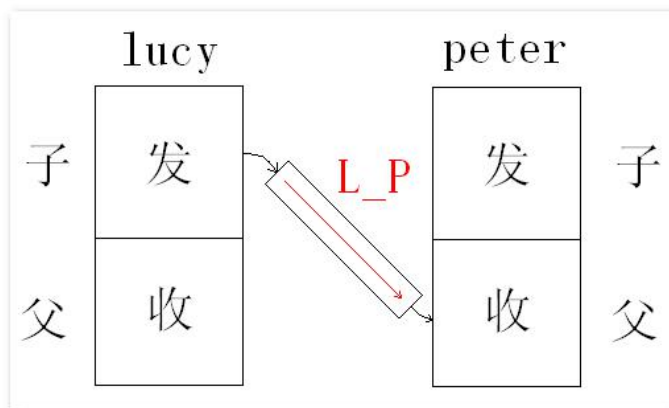
提示:

父进程创建子进程，实现多任务。

父进程负责发信息(向 FIFO 里写数据)，子进程负责接收信息(从 FIFO 里读数据)。

打开命名管道的用阻塞的方法打开。

聊天程序框架及流程图



第六章：消息队列

消息队列是消息的链表，存放在内存中，由内核维护

6.1 消息队列概述

消息队列的特点

- 1、消息队列中的消息是有类型的。
- 2、消息队列中的消息是有格式的。
- 3、消息队列可以实现消息的随机查询。消息不一定要以先进先出的次序读取，编程时可以按消息的类型读取。
- 4、消息队列允许一个或多个进程向它写入或者读取消息。
- 5、与无名管道、命名管道一样，从消息队列中读出消息，消息队列中对应的数据都会被删除。
- 6、每个消息队列都有消息队列标识符，消息队列的标识符在整个系统中是唯一的。
- 7、只有内核重启或人工删除消息队列时，该消息队列才会被删除。若不人工删除消息队列，消息队列会一直存在于系统中。

在 ubuntu 12.04 中消息队列限制值如下：

每个消息内容最多为 8K 字节

每个消息队列容量最多为 16K 字节

系统中消息队列个数最多为 1609 个

系统中消息个数最多为 16384 个

System V 提供的 IPC 通信机制需要一个 key 值，通过 key 值就可在系统内获得一个唯一的消息队列标识符。

key 值可以是人为指定的，也可以通过 ftok 函数获得。

6.1.1 ftok 函数

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

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

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

功能:

获得项目相关的唯一的 IPC 键值。

参数:

pathname: 路径名

proj_id: 项目 ID, 非 0 整数(只有低 8 位有效)

返回值:

成功返回 key 值, 失败返回 -1

6.2 消息队列的操作

6.2.1 创建消息队列

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

```
int msgget(key_t key, int msgflg);
```

功能:

创建一个新的或打开一个已经存在的消息队列。不同的进程调用此函数, 只要用相同的 key 值就能得到同一个消息队列的标识符。

参数:

key: IPC 键值。

msgflg: 标识函数的行为及消息队列的权限。

msgflg 的取值:

IPC_CREAT: 创建消息队列。

IPC_EXCL: 检测消息队列是否存在。

位或权限位: 消息队列位或权限位后可以设置消息队列的访问权限, 格式和 open 函数的 mode_t 一样, 但可执行权限未使用。

返回值:

成功: 消息队列的标识符, 失败: 返回-1。

使用 shell 命令操作消息队列:

查看消息队列

```
ipcs -q
```

删除消息队列

```
ipcrm -q msqid
```

消息队列的消息的格式。

```
typedef struct _msg
{
    long mtype;          /*消息类型*/
    char mtext[100];     /*消息正文*/
    ... /*消息的正文可以有多个成员*/
}MSG;
```

消息类型必须是长整型的, 而且必须是结构体类型的第一个成员, 类型下面是消息正文, 正文可以有多个成员 (正文成员可以是任意数据类型的)。

6.2.2 发送消息

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

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

功能:

将新消息添加到消息队列。

参数:

msqid: 消息队列的标识符。

msgp: 待发送消息结构体的地址。

msgsz: 消息正文的字节数。

msgflg: 函数的控制属性

0: msgsnd 调用阻塞直到条件满足为止。

IPC_NOWAIT: 若消息没有立即发送则调用该函数的进程会立即返回。

返回值:

成功: 0; 失败: 返回-1。

6.2.3 接收消息

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

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

功能:

从标识符为 msqid 的消息队列中接收一个消息。一旦接收消息成功，则消息在消息队列中被删除。

参数:

msqid: 消息队列的标识符，代表要从哪个消息列中获取消息。

msgp: 存放消息结构体的地址。

msgsz: 消息正文的字节数。

msgtyp: 消息的类型、可以有以下几种类型

msgtyp = 0: 返回队列中的第一个消息

msgtyp > 0: 返回队列中消息类型为 msgtyp 的消息

msgtyp < 0: 返回队列中消息类型值小于或等于 msgtyp 绝对值的消息，如果这种消息有若干个，则取类型值最小的消息。

注意:

若消息队列中有多种类型的消息，msgrcv 获取消息的时候按消息类型获取，不是先进先出的。

在获取某类型消息的时候，若队列中有多条此类型的消息，则获取最先添加的消息，即先进先出原则。

msgflg: 函数的控制属性

0: msgrcv 调用阻塞直到接收消息成功为止。

MSG_NOERROR: 若返回的消息字节数比 nbytes 字节数多, 则消息就会截短到 nbytes 字节, 且不通知消息发送进程。

IPC_NOWAIT: 调用进程会立即返回。若没有收到消息则立即返回-1。

返回值:

成功返回读取消息的长度, 失败返回-1。

6.2.4 消息队列的控制

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

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

功能:

对消息队列进行各种控制, 如修改消息队列的属性, 或删除消息消息队列。

参数:

msqid: 消息队列的标识符。

cmd: 函数功能的控制。

buf: msqid_ds 数据类型的地址, 用来存放或更改消息队列的属性。

cmd: 函数功能的控制

IPC_RMID: 删除由 msqid 指示的消息队列, 将它从系统中删除并破坏相关数据结构。

IPC_STAT: 将 msqid 相关的数据结构中各个元素的当前值存入到由 buf 指向的结构中。

IPC_SET: 将 msqid 相关的数据结构中的元素设置为由 buf 指向的结构中的对应值。

返回值:

成功: 返回 0; 失败: 返回 -1

例: 01_message_queue_write.c

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

typedef struct _msg
{
    long mtype;
    char mtext[50];
}MSG;

int main(int argc, char *argv[])
{
    key_t key;
    int msgqid;
    MSG msg;

    key = ftok(".", 2012);
    msgqid = msgget(key, IPC_CREAT|0666);
    if(msgqid == -1)
    {
        perror("msgget");
        exit(-1);
    }
    msg.mtype = 10;
    strcpy(msg.mtext, "hello world");
    msgsnd(msgqid, &msg, sizeof(msg.mtext), 0);
    return 0;
}
```

例：01_message_queue_read.c

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

typedef struct _msg
{
    long mtype;
    char mtext[50];
}MSG;

int main(int argc, char *argv[])
{
    key_t key;
    int msgqid;
    MSG msg;

    key = ftok(".", 2012);
    msgqid = msgget(key, IPC_CREAT|0666);
    if(msgqid == -1)
    {
        perror("msgget");
        exit(-1);
    }
    msgrcv(msgqid, &msg, sizeof(msg.mtext), 10, 0);
    printf("msg.mtext=%s\n", msg.mtext);
    msgctl(msgqid, IPC_RMID, NULL);
    return 0;
}
```

练习

题目：消息队列实现多人聊天程序

提示：

消息结构体类型

```
typedef struct msg
{
    long type;           //接收者类型
    char text[100];      //发送内容
    char name[20];       //发送者姓名
}MSG;
```

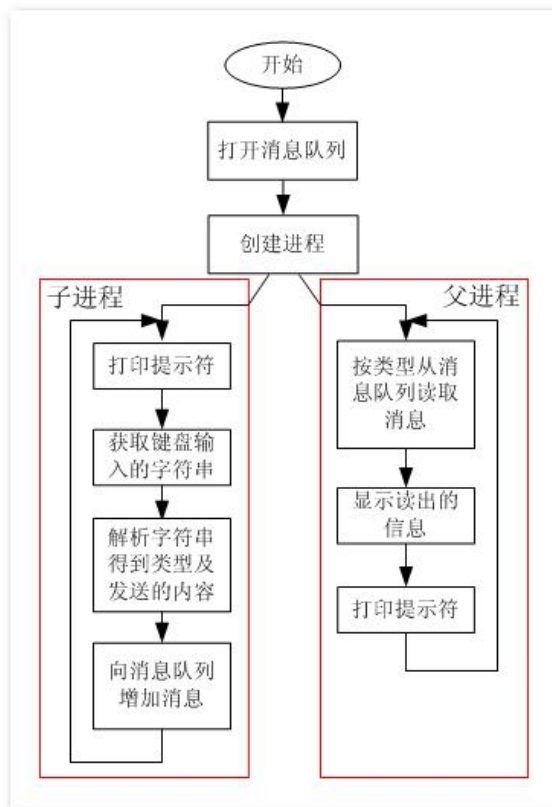
每个程序都有两个任务，一个任务是负责接收消息，一个任务是负责发送消息，通过 fork 创建子进程实现多任务。

一个进程负责接收信息，它只接收某种类型的消息，只要别的进程发送此类型的消息，此进程就能收到。收到后通过消息的 name 成员就可知道是谁发送的消息。

另一个进程负责发信息，可以通过输入来决定发送消息的类型。

设计程序的时候，接收消息的进程接收消息的类型不一样，这样就实现了发送的消息只被接收此类型消息的人收到，其它人收不到。这样就是实现了给特定的人发送消息。

消息队列多人聊天程序流程图



第七章：共享内存

7.1 共享内存概述

共享内存允许两个或者多个进程共享给定的存储区域。

共享内存的特点

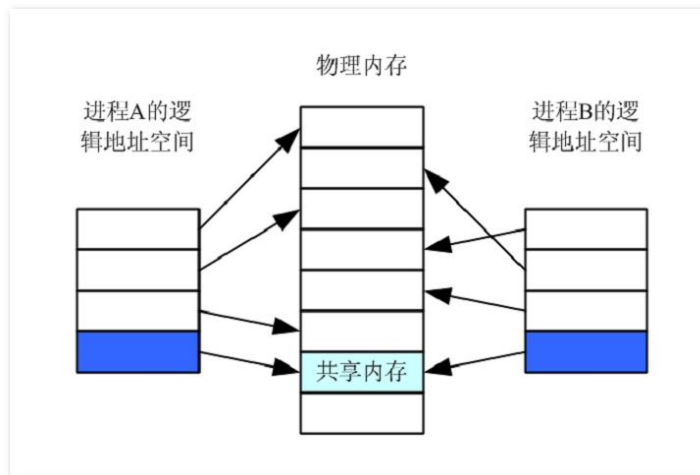
1、 共享内存是进程间共享数据的一种最快的方法。

一个进程向共享的内存区域写入了数据，共享这个内存区域的所有进程就可以立刻看到其中的内容。

2、使用共享内存要注意的是多个进程之间对一个给定存储区访问的互斥。

若一个进程正在向共享内存区写数据，则在它做完这一步操作前，别的进程不应当去读、写这些数据。

共享内存示意图



在 ubuntu 12.04 中共享内存限制值如下

- 1、共享存储区的最小字节数：1
- 2、共享存储区的最大字节数：32M
- 3、共享存储区的最大个数：4096
- 4、每个进程最多能映射的共享存储区的个数：4096

7.2 共享内存操作

7.2.1 获得一个共享存储标识符

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, size_t size,int shmflg);
```

功能：

创建或打开一块共享内存区

参数：

key: IPC 键值

size: 该共享存储段的长度(字节)

shmflg: 标识函数的行为及共享内存的权限。

IPC_CREAT: 如果不存在就创建

IPC_EXCL: 如果已经存在则返回失败

位或权限位: 共享内存位或权限位后可以设置共享内存的访问权限, 格式和 open 函数的 mode_t 一样, 但可执行权限未使用。

返回值:

成功: 返回共享内存标识符。

失败: 返回-1。

使用 shell 命令操作共享内存:

查看共享内存

ipcs -m

删除共享内存

ipcrm -m shmid

7.2.2 共享内存映射(attach)

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

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

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

功能:

将一个共享内存段映射到调用进程的数据段中。

参数:

shmid: 共享内存标识符。

shmaddr: 共享内存映射地址(若为 NULL 则由系统自动指定), 推荐使用 NULL。

shmflg: 共享内存段的访问权限和映射条件

0: 共享内存具有可读可写权限。

SHM_RDONLY: 只读。

SHM_RND: (shmaddr 非空时才有效)

没有指定 SHM_RND 则此段连接到 shmaddr 所指定的地址上 (shmaddr 必需页对齐)。

指定了 SHM_RND 则此段连接到 shmaddr- shmaddr%SHMLBA 所表示的地址上。

返回值:

成功: 返回共享内存段映射地址

失败: 返回 -1

注意:

shmat 函数使用的时候第二个和第三个参数一般设为 NULL 和 0, 即系统自动指定共享内存地址, 并且共享内存可读可写。

7.2.3 解除共享内存映射(detach)

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

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

```
int shmdt(const void *shmaddr);
```

功能:

将共享内存和当前进程分离 (仅仅是断开联系并不删除共享内存)。

参数:

shmaddr: 共享内存映射地址。

返回值:

成功返回 0, 失败返回 -1。

7.2.4 共享内存控制

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

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

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

功能:

共享内存空间的控制。

参数:

shmid: 共享内存标识符。

cmd: 函数功能的控制。

buf: shmid_ds 数据类型的地址, 用来存放或修改共享内存的属性。

cmd: 函数功能的控制

IPC_RMID: 删除。

IPC_SET: 设置 shmid_ds 参数。

IPC_STAT: 保存 shmid_ds 参数。

SHM_LOCK: 锁定共享内存段(超级用户)。

SHM_UNLOCK: 解锁共享内存段。

返回值:

成功返回 0, 失败返回 -1。

例: 02_shared_memory_write.c

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

#define BUFSZ 2048

int main(int argc, char *argv[])
{
    int shmid;
    int ret;
    key_t key;
    char *shmadd;

    key = ftok(".", 2012);
    if(key == -1)
    {
```

```
        perror("ftok");
    }
    system("ipcs -m");
    /*打开共享内存*/
    shmid = shmget(key, BUFSZ, IPC_CREAT|0666);
    if(shmid < 0)
    {
        perror("shmget");
        exit(-1);
    }
    /*映射*/
    shmadd = shmat(shmid, NULL, 0);
    if(shmadd < 0)
    {
        perror("shmat");
        exit(-1);
    }
    /*读共享内存区数据*/
    printf("data = [%s]\n", shmadd);
    /*分离共享内存和当前进程*/
    ret = shmdt(shmadd);
    if(ret < 0)
    {
        perror("shmdt");
        exit(1);
    }
    else
    {
        printf("deleted shared-memory\n");
    }
    /*删除共享内存*/
    shmctl(shmid, IPC_RMID, NULL);
    system("ipcs -m");
    return 0;
}
```

例：02_shared_memory_read.c

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

#define BUFSZ 2048

int main(int argc, char *argv[])
{
    int shmid;
    int ret;
    key_t key;
    char *shmadd;

    key = ftok(".", 2012);
    if(key == -1)
    {
        perror("ftok");
    }
    /*创建共享内存*/
    shmid = shmget(key, BUFSZ, IPC_CREAT|0666);
    if(shmid < 0)
    {
        perror("shmget");
        exit(-1);
    }
    /*映射*/
    shmadd = shmat(shmid, NULL, 0);
    if(shmadd < 0)
    {
        perror("shmat");
        _exit(-1);
    }
    /*拷贝数据至共享内存区*/
```

```
printf("copy data to shared-memory\n");  
bzero(shmadd, BUFSZ);  
strcpy(shmadd, "data in shared memory\n");  
return 0;  
}
```

注意：

SHM_LOCK 用于锁定内存，禁止内存交换。并不代表共享内存被锁定后禁止其它进程访问。

其真正的意义是：被锁定的内存不允许被交换到虚拟内存中。

这样做的优势在于让共享内存一直处于内存中，从而提高程序性能。

第八章：线程

8.1 线程概述

8.1.1 线程的概念

每个进程都拥有自己的数据段、代码段和堆栈段，这就造成进程在进行创建、切换、撤销操作时，需要较大的系统开销。

为了减少系统开销，从进程中演化出了线程。

线程存在于进程中，共享进程的资源。

线程是进程中的独立控制流，由环境（包括寄存器组和程序计数器）和一系列的执行指令组成。

每个进程有一个地址空间和一个控制线程。



8.1.2 线程和进程的比较

调度:

线程是 CPU 调度和分派的基本单位。

拥有资源:

进程是系统中程序执行和资源分配的基本单位。

线程自己一般不拥有资源（除了必不可少的程序计数器，一组寄存器和栈），但它可以去访问其所属进程的资源，如进程代码段，数据段以及系统资源（已打开的文件，I/O 设备等）。

系统开销:

同一个进程中的多个线程可共享同一地址空间，因此它们之间的同步和通信的实现也变得比较容易。

在进程切换时候，涉及到整个当前进程 CPU 环境的保存以及新被调度运行的进程的 CPU 环境的设置；而线程切换只需要保存和设置少量寄存器的内容，并不涉及存储器管理方面的操作，从而能更有效地使用系统资源和提高系统的吞吐量。

并发性:

不仅进程间可以并发执行，而且在一个进程中的多个线程之间也可以并发执行。

8.1.3 多线程的用处

使用多线程的目的主要有以下几点：

多任务程序的设计

一个程序可能要处理不同应用，要处理多种任务，如果开发不同的进程来处理，系统开销很大，数据共享，程序结构都不方便，这时可使用多线程编程方法。

并发程序设计

一个任务可能分成不同的步骤去完成，这些不同的步骤之间可能是松散耦合，可能通过线程的互斥，同步并发完成。这样可以为不同的任务步骤建立线程。

网络程序设计

为提高网络的利用效率，我们可能使用多线程，对每个连接用一个线程去处理。

数据共享

同一个进程中的不同线程共享进程的数据空间，方便不同线程间的数据共享。

在多 CPU 系统中，实现真正的并行。

8.2 线程的基本操作

就像每个进程都有一个进程号一样，每个线程也有一个线程号。

进程号在整个系统中是唯一的，但线程号不同，线程号只在它所属的进程环境中有效。

进程号用 `pid_t` 数据类型表示，是一个非负整数。线程号则用 `pthread_t` 数据类型来表示。

有的系统实现 pthread_t 的时候，用一个结构体来表示，所以在可移植的操作系统实现不能把它做为整数处理。

8.2.1 线程的创建

```
#include <pthread.h>
```

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,  
                  void *(*start_routine) (void *), void *arg);
```

功能：

创建一个线程。

参数：

thread：线程标识符地址。

attr：线程属性结构体地址。

start_routine：线程函数的入口地址。

arg：传给线程函数的参数。

返回值：

成功：返回 0

失败：返回非 0

与 fork 不同的是 pthread_create 创建的线程不与父线程在同一点开始运行，而是从指定的函数开始运行，该函数运行完后，该线程也就退出了。

线程依赖进程存在的，如果创建线程的进程结束了，线程也就结束了。

线程函数的程序在 pthread 库中，故链接时要加上参数 -lpthread。

例：01_pthread_create_1.c 验证多线程实现多任务，及线程间共享全局变量

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

```
int var = 8;

void *thread_1(void *arg)
{
    while(1)
    {
        printf("this is my new thread1 var++\n");
        var++;
        sleep(1);
    }
    return NULL;
}

void *thread_2(void * arg)
{
    while(1)
    {
        printf("this is my new thread2 var = %d\n", var);
        sleep(1);
    }
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t tid1,tid2;

    /*创建两个线程*/
    pthread_create(&tid1, NULL, thread_1, NULL);
    pthread_create(&tid2, NULL, thread_2, NULL);
    while(1);
    return 0;
}
```

例 01_pthread_create_2.c 验证线程函数传参

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

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

/*传参方法 1*/
void *thread_1(void *arg)
{
    int rec = 0;

    sleep(1);
    rec = (int)(arg);
    printf("new thread1 arg = %d\n", rec);
    return NULL;
}

/*传参方法 2*/
void *thread_2(void * arg)
{
    int rec = 0;

    sleep(1);
    rec = *((int *)arg);
    printf("new thread2 arg = %d\n", rec);
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t tid1,tid2;
    int test = 100;
    //double test = 100;

    /*创建两个线程*/
    pthread_create(&tid1, NULL, thread_1, (void *)test);
    pthread_create(&tid2, NULL, thread_2, (void *)&test);
    //test++;
    while(1);
    return 0;
}
```

}

8.2.2 线程等待

```
#include <pthread.h>
```

```
int pthread_join(pthread_t thread, void **retval);
```

功能:

等待子线程结束，并回收子线程资源。

参数:

thread: 被等待的线程号。

retval: 用来存储线程退出状态的指针的地址。

返回值:

成功返回 0，失败返回非 0。

例：02_pthread_join_1.c 验证 pthread_join 的阻塞效果

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

void *thead(void *arg)
{
    static int num = 123;

    printf("after 2 seceonds, thread will return\n");
    sleep(2);
    return &num;
}

int main(int argc, char *argv[])
{
    pthread_t tid1;
    int ret = 0;
```

```
void *value = NULL;

ret = pthread_create(&tid1, NULL, thead, NULL);
if(ret != 0)
    perror("pthread_create");
pthread_join(tid1, &value);
printf("value = %d\n", *((int *)value));
return 0;
}
```

例：02_pthread_join_2.c 验证 pthread_join 接收线程的返回值

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

void *thead(void *arg)
{
    int num = 123;

    printf("after 2 seceonds, thread will return\n");
    sleep(2);
    return (void *)num;
}

int main(int argc, char *argv[])
{
    pthread_t tid1;
    int ret = 0;
    int value = 0;

    ret = pthread_create(&tid1, NULL, thead, NULL);
    if(ret != 0)
        perror("pthread_create");
    pthread_join(tid1, (void *)&value);
    printf("value = %d\n", value);
    return 0;
}
```

```
}
```

8.2.3 线程分离

创建一个线程后应回收其资源，但使用 `pthread_join` 函数会使调用者阻塞，故 Linux 提供了线程分离函数：`pthread_detach`。

```
#include <pthread.h>
```

```
int pthread_detach(pthread_t thread);
```

功能：

使调用线程与当前进程分离，使其成为一个独立的线程，该线程终止时，系统将自动回收它的资源。

参数：

thread：线程号

返回值：

成功：返回 0，**失败**返回非 0。

例：03_pthread_detach.c

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

void *thead(void *arg)
{
    int i;
    for(i=0; i<5; i++)
    {
        printf("I am runing\n");
        sleep(1);
    }
    return NULL;
}
```

```
int main(int argc, char *argv[])
{
    int ret = 0;
    pthread_t tid1;

    ret = pthread_create(&tid1, NULL, thead, NULL);
    if(ret!=0)
        perror("pthread_create");
    pthread_detach(tid1);
    pthread_join(tid1, NULL);
    printf("after join\n");
    sleep(3);
    printf("I am leaving\n");
    return 0;
}
```

8.2.4 线程退出

在进程中我们可以调用 `exit` 函数或 `_exit` 函数来结束进程，在一个线程中我们可以通过以下三种在不终止整个进程的情况下停止它的控制流。

- 1、线程从执行函数中返回。
- 2、线程调用 `pthread_exit` 退出线程。
- 3、线程可以被同一进程中的其它线程取消。

8.2.4.1 线程退出函数

```
#include <pthread.h>
```

```
void pthread_exit(void *retval);
```

功能:

退出调用线程。

参数:

retval: 存储线程退出状态的指针。

注:

一个进程中的多个线程是共享该进程的数据段，因此，通常线程退出后所占用的资源并不会释放。

例：04_pthread_exit.c

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

void *thread(void *arg)
{
    int i = 0;
    while(1)
    {
        printf("I am runing\n");
        sleep(1);
        i++;
        if(i==3)
        {
            pthread_exit((void *)1);
        }
    }
    return NULL;
}

int main(int argc, char *argv[])
{
    int ret = 0;
    pthread_t tid;
    void *value = NULL;

    ret = pthread_create(&tid, NULL, thread, NULL);
    if(ret!=0)
        perror("pthread_create");
    pthread_join(tid, &value);
    printf("value = %p\n", (int *)value);
}
```



```
return 0;  
}
```

8.2.4.2 线程的取消

取消线程是指取消一个正在执行线程的操作。

```
#include <pthread.h>
```

```
int pthread_cancel(pthread_t thread);
```

功能:

取消线程。

参数:

thread: 目标线程 ID。

返回值:

成功返回 0，失败返回出错编号。

pthread_cancel 函数的实质是发信号给目标线程 thread，使目标线程退出。

此函数只是发送终止信号给目标线程，不会等待取消目标线程执行完才返回。

然而发送成功并不意味着目标线程一定就会终止，线程被取消时，线程的取消属性会决定线程能否被取消以及何时被取消。

线程的取消状态

即线程能不能被取消

线程取消点

即线程被取消的地方

线程的取消类型

在线程能被取消的状态下，是立马被取消结束还是执行到取消点的时候被取消结束

线程的取消状态

在 Linux 系统下，线程默认可以被取消。编程时可以通过 `pthread_setcancelstate` 函数设置线程是否可以被取消。

`pthread_setcancelstate(int state,int *old_state);`

state:

PTHREAD_CANCEL_DISABLE: 不可以被取消

PTHREAD_CANCEL_ENABLE: 可以被取消。

old_state:

保存调用线程原来的可取消状态的内存地址。

例：05_pthread_setcancelstate.c 验证设置线程能否被取消，然后看线程能不能被取消

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

void *thread_cancel(void *arg)
{
    //pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
    while(1)
    {
        printf("this is my new thread_cancel\n");
        sleep(1);
    }
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t tid1;
    int ret = 0;
    pthread_create(&tid1, NULL, thread_cancel, NULL);
```

```

    if(ret != 0)
        perror("pthread_create");
    sleep(3);
    pthread_cancel(tid1);
    pthread_join(tid1, NULL);
    return 0;
}

```

线程的取消点

线程被取消后，该线程并不是马上终止，默认情况下线程执行到消点时才能被终止。编程时可以通过 `pthread_testcancel` 函数设置线程的取消点。

`void pthread_testcancel(void);`

当别的线程取消调用此函数的线程时候，被取消的线程执行到此函数时结束。

POSIX.1 保证线程在调用表 1、表 2 中的任何函数时，取消点都会出现。

表 1:

<code>accept</code>	<code>mq_timedsend</code>	<code>putpmsg</code>	<code>sigsuspend</code>
<code>aio_suspend</code>	<code>msgrocv</code>	<code>pwrite</code>	<code>sigtimedwait</code>
<code>clock_nanosleep</code>	<code>msgsnd</code>	<code>read</code>	<code>sigwait</code>
<code>close</code>	<code>msync</code>	<code>readv</code>	<code>sigwaitinfo</code>
<code>connect</code>	<code>nanosleep</code>	<code>recv</code>	<code>sleep</code>
<code>creat</code>	<code>open</code>	<code>recvfrom</code>	<code>system</code>
<code>fcntl2</code>	<code>pause</code>	<code>recvmsg</code>	<code>tcdrain</code>
<code>fsync</code>	<code>poll</code>	<code>select</code>	<code>usleep</code>
<code>getmsg</code>	<code>pread</code>	<code>sem_timedwait</code>	<code>wait</code>
<code>getpmsg</code>	<code>pthread_cond_timedwait</code>	<code>sem_wait</code>	<code>waitid</code>
<code>lockf</code>	<code>pthread_cond_wait</code>	<code>send</code>	<code>waitpid</code>
<code>mq_receive</code>	<code>pthread_join</code>	<code>sendmsg</code>	<code>write</code>
<code>mq_send</code>	<code>pthread_testcancel</code>	<code>sendto</code>	<code>writerv</code>
<code>mq_timedreceive</code>	<code>putmsg</code>	<code>sigpause</code>	

表 2:

catclose	ftell	getwc	printf
catgets	ftello	getwchar	putc
catopen	ftw	getwd	putc_unlocked
closedir	fwprintf	glob	putchar
closelog	fwrite	iconv_close	putchar_unlocked
ctermid	fwscanf	iconv_open	puts
dbm_close	getc	ioctl	pututxline
dbm_delete	getc_unlocked	lseek	putwc
dbm_fetch	getchar	mkstemp	putwchar
dbm_nextkey	getchar_unlocked	nftw	readdir
dbm_open	getcwd	opendir	readdir_r
dbm_store	getdate	openlog	remove
dlclose	getgrnt	polose	rename
dlopen	getgrgid	perror	rewind
endgrent	getgrgid_r	popen	rewinddir
endhostent	getgrnam	posix_fadvise	scanf
endnetent	getgrnam_r	posix_fallocate	seekdir
endprotoent	gethostbyaddr	posix_madvise	semop
endpwent	gethostbyname	posix_spawn	setgrent
endservent	gethostent	posix_spawnnp	sethostent
endutxent	gethostname	posix_trace_clear	setnetent
fclose	getlogin	posix_trace_close	setprotoent
fcntl	getlogin_r	posix_trace_create	setpwent
fflush	getnetbyaddr	posix_trace_create_withlog	setservent
fgetc	getnetbyname	posix_trace_eventtypelist_getnext_id	setutxent
fgetpos	getnetent	posix_trace_eventtypelist_rewind	strerror
fgets	getprotobyname	posix_trace_flush	syslog
fgetwc	getprotobynumber	posix_trace_get_attr	tmpfile
fgetws	getprotoent	posix_trace_get_filter	tmpnam
fopen	getpwent	posix_trace_get_status	ttyname
fprintf	getpwnam	posix_trace_getnext_event	ttyname_r
fputc	getpwnam_r	posix_trace_open	ungetc
fputs	getpwuid	posix_trace_rewind	ungetwc
fputwc	getpwuid_r	posix_trace_set_filter	unlink
fputws	gets	posix_trace_shutdown	vfprintf
fread	getservbyname	posix_trace_timedgetnext_event	vfwprintf
freopen	getservbyport	posix_typed_mem_open	vprintf
fscanf	getservent	pthread_rwlock_rdlock	vwprintf
fseek	getutxent	pthread_rwlock_timedrdlock	wprintf
fseeko	getutxid	pthread_rwlock_timedwrlock	wscanf
fsetpos	getutxline	pthread_rwlock_wrlock	

例：05_pthread_testcancel.c

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

void *thread_cancel(void *arg)
{
    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    while(1)
    {
        //pthread_testcancel();
    }
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t tid1;
    int ret = 0;
    pthread_create(&tid1, NULL, thread_cancel, NULL);
    if(ret!=0)
        perror("pthread_create");
    sleep(3);
    pthread_cancel(tid1);
    pthread_join(tid1, NULL);
    return 0;
}
```

线程的取消类型

线程被取消后，该线程并不是马上终止，默认情况下线程执行到消点时才能被终止。编程时可以通过 `pthread_setcanceltype` 函数设置线程是否可以立即被取消。

```
pthread_setcanceltype(int type, int *oldtype);
```

type:

PTHREAD_CANCEL_ASYNCHRONOUS: 立即取消、

PTHREAD_CANCEL_DEFERRED: 不立即被取消

oldtype:

保存调用线程原来的可取消类型的内存地址。

例：05_pthread_setcanceltype.c

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

void *thread_cancel(void *arg)
{
    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    //pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS,NULL);
    pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED,NULL);
    while(1)
    {
    }
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t tid1;
    int ret = 0;
    pthread_create(&tid1, NULL, thread_cancel, NULL);
    if(ret!=0)
        perror("pthread_create");
    sleep(3);
    pthread_cancel(tid1);
    pthread_join(tid1, NULL);
    return 0;
}
```

8.2.4.3 线程退出清理函数

和进程的退出清理一样，线程也可以注册它退出时要调用的函数，这样的函数称为线程清理处理程序(thread cleanup handler)。

注意：

线程可以建立多个清理处理程序。

处理程序在栈中，故它们的执行顺序与它们注册时的顺序相反。

注册清理函数

```
#include <pthread.h>
```

```
void pthread_cleanup_push(void (* routine)(void *), void *arg);
```

功能：

将清除函数压栈。即注册清理函数。

参数：

routine：线程清理函数的指针。

arg：传给线程清理函数的参数。

弹出清理函数

```
#include <pthread.h>
```

```
void pthread_cleanup_pop(int execute);
```

功能：

将清除函数弹栈，即删除清理函数。

参数：

execute:线程清理函数执行标志位。

非 0，弹出清理函数，执行清理函数。

0，弹出清理函数，不执行清理函数。

当线程执行以下动作时会调用清理函数：

- 1、调用 `pthread_exit` 退出线程。
- 2、响应其它线程的取消请求。
- 3、用非零 `execute` 调用 `pthread_cleanup_pop`。

无论哪种情况 `pthread_cleanup_pop` 都将删除上一次 `pthread_cleanup_push` 调用注册的清理处理函数。

例：06_pthread_cleanup_exit.c 验证线程调用 `pthread_exit` 函数时，系统自动调用线程清理函数

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <string.h>
void cleanup(void *arg)
{
    printf("clean up ptr = %s\n", (char *)arg);
    free((char *)arg);
}

void *thread(void *arg)
{
    char *ptr = NULL;

    /*建立线程清理程序*/
    printf("this is new thread\n");
    ptr = (char*)malloc(100);
    pthread_cleanup_push(cleanup, (void*)(ptr));
    bzero(ptr, 100);
    strcpy(ptr, "memory from malloc");
    printf("before exit\n");
    pthread_exit(NULL);
    sleep(3);

    /*注意 push 与 pop 必须配对使用，即使 pop 执行不到*/
}
```

```
        printf("before pop\n");
        pthread_cleanup_pop(1);
        return NULL;
    }

int main(int argc, char *argv[])
{
    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL); // 创建一个线程
    pthread_join(tid, NULL);
    printf("process is dying\n");
    return 0;
}
```

例：06_pthread_cleanup_cancel.c 验证线程被取消时，系统自动调用线程清理函数

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

void cleanup(void *arg)
{
    printf("clean up ptr = %s\n", (char *)arg);
    free((char *)arg);
}

void *thread(void *arg)
{
    char *ptr = NULL;

    /*建立线程清理程序*/
    printf("this is new thread\n");
    ptr = (char*)malloc(100);
    pthread_cleanup_push(cleanup, (void*)(ptr));
    bzero(ptr, 100);
}
```

```
strcpy(ptr, "memory from malloc");
sleep(3);

/*注意 push 与 pop 必须配对使用，即使 pop 执行不到*/
printf("before pop\n");
pthread_cleanup_pop(1);
return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL); // 创建一个线程
    sleep(1);
    printf("before cancel\n");
    /*子线程响应 pthread_cancel 后，会执行线程处理函数*/
    pthread_cancel(tid);
    pthread_join(tid, NULL);
    printf("process is dying\n");
    return 0;
}
```

例：06_pthread_cleanup_pop.c 验证调用 pthread_cleanup_pop 函数时，系统自动调用线程清理函数

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

void cleanup_func1(void *arg)
{
    printf("in cleanup func1\n");
    printf("clean up ptr = %s\n", (char *)arg);
    free((char *)arg);
}
```

```
void cleanup_func2(void *arg)
{
    printf("in cleanup func2\n");
}

void *thread(void *arg)
{
    char *ptr = NULL;

    /*建立线程清理程序*/
    printf("this is new thread\n");
    ptr = (char*)malloc(100);
    pthread_cleanup_push(cleanup_func1, (void*)(ptr));
    pthread_cleanup_push(cleanup_func2, NULL);
    bzero(ptr, 100);
    strcpy(ptr, "memory from malloc");
    /*注意 push 与 pop 必须配对使用，即使 pop 执行不到*/
    printf("before pop\n");
    pthread_cleanup_pop(1);
    printf("before pop\n");
    pthread_cleanup_pop(1);
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL); // 创建一个线程
    pthread_join(tid, NULL);
    printf("process is dying\n");
    return 0;
}
```

第九章：多任务互斥和同步

9.1 互斥和同步概述

在多任务操作系统中，同时运行的多个任务可能

都需要访问/使用同一种资源

多个任务之间有依赖关系，某个任务的运行依赖于另一个任务

同步和互斥就是用于解决这两个问题的。

互斥：

一个公共资源同一时刻只能被一个进程或线程使用，多个进程或线程不能同时使用公共资源。POSIX 标准中进程和线程同步和互斥的方法,主要有信号量和互斥锁两种方式。

同步：

两个或两个以上的进程或线程在运行过程中协同步调，按预定的先后次序运行。

9.2 互斥锁

9.2.1 互斥锁的概念

mutex 是一种简单的加锁的方法来控制对共享资源的访问，mutex 只有两种状态，即上锁(lock)和解锁(unlock)。

在访问该资源前，首先应申请 mutex，如果 mutex 处于 unlock 状态，则会申请到 mutex 并立即 lock；如果 mutex 处于 lock 状态，则默认阻塞申请者。

unlock 操作应该由 lock 者进行。

9.2.2 初始化互斥锁

mutex 用 pthread_mutex_t 数据类型表示, 在使用互斥锁前, 必须先对它进行初始化。

静态分配的互斥锁:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

动态分配互斥锁:

```
pthread_mutex_t mutex;
```

pthread_mutex_init(&mutex, NULL); 在所有使用过此互斥锁的线程都不再需要使用时, 应调用

销毁互斥锁

pthread_mutex_destroy 销毁互斥锁。

```
#include <pthread.h>
```

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
```

功能:

初始化一个互斥锁。

参数:

mutex: 互斥锁地址。

attr: 互斥锁的属性, NULL 为默认的属性。

返回值:

成功返回 0, 失败返回非 0。

9.2.3 互斥锁上锁

```
#include <pthread.h>
```

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

功能:

对互斥锁上锁, 若已经上锁, 则调用者一直阻塞到互斥锁解锁。

参数:

mutex: 互斥锁地址。

返回值:

成功返回 0，失败返回非 0。

9.2.4 互斥锁上锁 2

```
#include <pthread.h>
```

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

功能:

对互斥锁上锁，若已经上锁，则上锁失败，函数立即返回。

参数:

mutex: 互斥锁地址。

返回值:

成功返回 0，失败返回非 0。

9.2.5 互斥锁解锁

```
#include <pthread.h>
```

```
int pthread_mutex_unlock(pthread_mutex_t * mutex);
```

功能:

对指定的互斥锁解锁。

参数:

mutex: 互斥锁地址。

返回值:

成功返回 0，失败返回非 0。

9.2.6 销毁互斥锁

```
#include <pthread.h>
```

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

功能:

销毁指定的一个互斥锁。

参数:

mutex: 互斥锁地址。

返回值:

成功返回 0，失败返回非 0。

例：01_pthread_mutex.c 互斥锁实现模拟打印机

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

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void printer(char *str)
{
    pthread_mutex_lock(&mutex);
    while(*str!='\0')
    {
        putchar(*str);
        fflush(stdout);
        str++;
        sleep(1);
    }
    printf("\n");
    pthread_mutex_unlock(&mutex);
}

void *thread_fun_1(void *arg)
{
```



```
        char *str = "hello";
        printer(str);
    }

void *thread_fun_2(void *arg)
{
    char *str = "world";
    //pthread_mutex_unlock(&mutex);
    printer(str);
}

int main(void)
{
    pthread_t tid1, tid2;

    pthread_mutex_init(&mutex, NULL);
    pthread_create(&tid1, NULL, thread_fun_1, NULL);
    pthread_create(&tid2, NULL, thread_fun_2, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    return 0;
}
```

9.3 信号量

9.3.1 信号量的概念

信号量广泛用于进程或线程间的同步和互斥，信号量本质上是一个非负的整数计数器，它被用来控制对公共资源的访问。

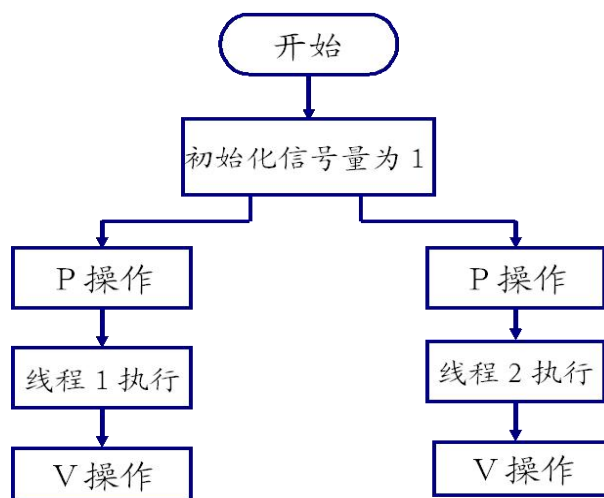
编程时可根据操作信号量值的结果判断是否对公共资源具有访问的权限，当信号量值大于 0 时，则可以访问，否则将阻塞。

P V 原语是对信号量的操作，一次 P 操作使信号量 sem 减 1，一次 V 操作使信号量 sem 加 1。

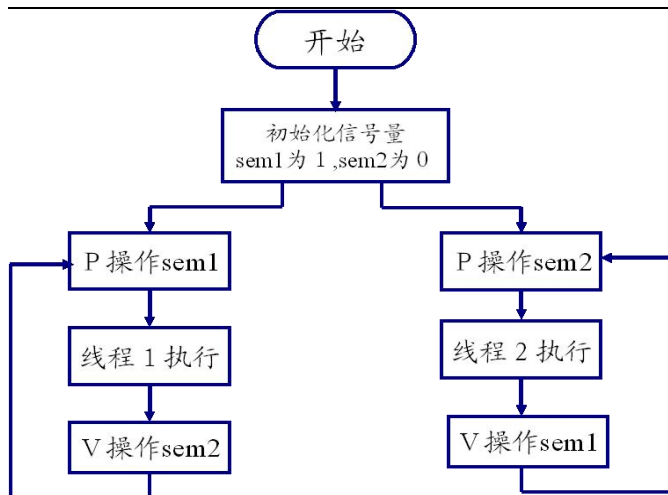
信号量主要用于进程或线程间的同步和互斥这两种典型情况。

- 1、若用于互斥，几个进程（或线程）往往只设置一个信号量。
- 2、若用于同步操作，往往会设置多个信号量，并且安排不同的初始值，来实现它们之间的执行顺序。

信号量用于互斥



信号量用于同步



9.3.2 信号量的操作

9.3.2.1 信号量的初始化

```
#include <semaphore.h>
```

```
int sem_init(sem_t *sem, int pshared,unsigned int value);
```

功能:

创建一个信号量并初始化它的值。

参数:

sem: 信号量的地址。

pshared: 等于 0, 信号量在线程间共享; 不等于 0, 信号量在进程间共享。

value: 信号量的初始值。

返回值:

成功返回 0, 失败返回-1。

9.3.2.2 信号量 P 操作

```
#include <semaphore.h>
```

```
int sem_wait(sem_t *sem);
```

功能:

将信号量的值减 1，若信号量的值小于 0，此函数会引起调用者阻塞。

参数:

sem: 信号量地址。

返回值:

成功返回 0，失败返回-1。

```
#include <semaphore.h>
```

```
int sem_trywait(sem_t *sem);
```

功能:

将信号量的值减 1，若信号量的值小于 0，则对信号量的操作失败，函数立即返回。

参数:

sem: 信号量地址。

返回值:

成功返回 0，失败返回-1。

9.3.2.3 信号量的 V 操作

```
#include <semaphore.h>
```

```
int sem_post(sem_t *sem);
```

功能:

将信号量的值加 1 并发出信号唤醒等待线程。

参数:

sem: 信号量地址。

返回值:

成功返回 0，失败返回-1。

9.3.2.4 获取信号量的计数值

```
#include <semaphore.h>

int sem_getvalue(sem_t *sem, int *sval);
```

功能:

获取 sem 标识的信号量的值，保存在 sval 中。

参数:

sem: 信号量地址。

sval: 保存信号量值的地址。

返回值:

成功返回 0，失败返回-1。

9.3.2.5 信号量的销毁

```
#include <semaphore.h>

int sem_destroy(sem_t *sem);
```

功能:

删除 sem 标识的信号量。

参数:

sem: 信号量地址。

返回值:

成功返回 0，失败返回-1。

例: 02_semaphore_1.c 信号量实现互斥功能，模拟打印机

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

sem_t sem;

void printer(char *str)
{
    sem_wait(&sem);
    while(*str)
    {
        putchar(*str);
        fflush(stdout);
        str++;
        sleep(1);
    }
    sem_post(&sem);
}

void *thread_fun1(void *arg)
{
    char *str1 = "hello";
    printer(str1);
}

void *thread_fun2(void *arg)
{
    char *str2 = "world";
    printer(str2);
}

int main(void)
{
    pthread_t tid1, tid2;

    sem_init(&sem, 0, 1);
```

```
pthread_create(&tid1, NULL, thread_fun1, NULL);
pthread_create(&tid2, NULL, thread_fun2, NULL);

pthread_join(tid1, NULL);
pthread_join(tid2, NULL);
return 0;
}
```

例：02_semaphore_2.c 验证信号量实现同步功能

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

sem_t sem_g, sem_p; //定义两个信号量
char ch = 'a';

void * pthread_g(void *arg) //此线程改变字符 ch 的值
{
    while(1)
    {
        sem_wait(&sem_g);
        ch++;
        sleep(1);
        sem_post(&sem_p);
    }
}

void * pthread_p(void *arg) //此线程打印 ch 的值
{
    while(1)
    {
        sem_wait(&sem_p);
        printf("%c", ch);
    }
}
```

```
        fflush(stdout);
        sem_post(&sem_g);
    }
}

int main(int argc, char *argv[])
{
    pthread_t tid1, tid2;
    sem_init(&sem_g, 0, 0); //初始化信号量
    sem_init(&sem_p, 0, 1);

    pthread_create(&tid1, NULL, pthread_g, NULL);
    pthread_create(&tid2, NULL, pthread_p, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    return 0;
}
```

9.3.3 有名信号量（扩展）

9.3.3.1 有名信号量概念

其实 POSIX 的信号量有两种：

- 1、无名信号量
- 2、有名信号量

前面我们介绍的就是无名信号量，无名信号量一般用于线程间同步或互斥。

而有名信号量一般用于进程间同步或互斥。

9.3.3.2 有名信号量的打开或创建

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

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

```
#include <semaphore.h>
```

当信号量存在时使用：

```
sem_t *sem_open(const char *name, int oflag);
```

当信号量不存在时使用：

```
sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
```

功能：

创建一个信号量。

参数：

name：信号量文件名。

flags：sem_open 函数的行为标志。

mode：文件权限(可读、可写、可执行)的设置。

value：信号量初始值。

返回值：

成功返回信号量的地址，失败返回 SEM_FAILED。

9.3.3.3 信号量的关闭

```
#include <semaphore.h>
```

```
int sem_close(sem_t *sem);
```

功能：

关闭有名信号量。

参数：

sem：指向信号量的指针。

返回值：

成功返回 0，失败返回-1。

9.3.3.4 信号量的删除

```
#include <semaphore.h>
```

```
int sem_unlink(const char *name);
```

功能:

删除信号量的文件。

参数:

name: 信号量文件名。

返回值:

成功返回 0，失败返回-1。

9.3.4 信号量实做

生产者消费者:

有一个仓库，生产者负责生产产品，并放入仓库，消费者会从仓库中拿走产品(消费)。

要求:

- 1、仓库中每次只能入一人(生产者或消费者)。
- 2、仓库中可存放产品的数量最多 10 个,当仓库放满时，生产者不能再放入产品。
- 3、当仓库空时，消费者不能从中取出产品。
- 4、生产、消费速度不同。