Linux及系统编程

一、Bash解析器及快捷键

1. `Tab`键

> 补全命令
>
> 补全路劲
>
> 显示当前目录下的所有目录

2. `清屏`:`clear`

```
1 ubuntu@ubuntu:~$ clear
```

3. 中断进程

> `ctlr+c`的作用时中断终端的操作

4. 遍历历史命令

> `ctlr+p`，`↑`往上遍历
>
> `ctlr+n`，`↓`往下遍历

5. 光标相关操作

> `ctlr+b`，`←`光标左移
>
> `ctlr+f`，`→`光标右移
>
> `ctlr+a`移动到头部
>
> `ctlr+e`移动到尾部

6. 字符删除

> `ctlr+h` 删除光标前边的字符
>
> `ctlr+d` 删除光标后边的字符
>
> **`ctlr+u` 删除光标前边的所有字符**
>
> **`ctlr+k` 删除光标后边的所有字符**

二、Linux命令

1.命令类型

1. 内建命令

> 本身自带的命令
>
> 内建命令帮助文档查看方法
>
> `help+命令`

2. 外部命令

> 安装软件附带的
>
> 外部命令帮助文档查看方法
>
> `ls --help`

3. 查看命令类型

```
ubuntu@ubuntu:~$ type type
type 是 shell 内建
ubuntu@ubuntu:~$ ^C
ubuntu@ubuntu:~$ type cd
cd 是 shell 内建
ubuntu@ubuntu:~$ type ls
ls 是 `ls --color=auto' 的别名
```

4. 绝对路径和相对路径

2.目录相关命令

1. `pwd`打印当前的绝对路径

```
ubuntu@ubuntu:~$ help pwd
pwd: pwd [-LP]
    打印当前工作目录的名字。
    选项：
      -L    打印 $PWD 变量的值，如果它包含了当前的工作目录
      -P    打印当前的物理路径，不带有任何的符号链接
    默认情况下，`pwd' 的行为和带 `-L' 选项一致
    退出状态：
    除非使用了无效选项或者当前目录不可读，否则返回状态为0。
```

2. `cd`跳转目录

```
cd # 切换到当前用户的主目录
cd ~ # 切换到当前用户的主目录
cd . # 切换到当前目录
cd .. # 切换到上级目录
cd - # 可进入上一个进入的目录
```

3. `mkdir`创建一个目录（不存在的目录）

```
mkdir "名称"  # 当前目录先创建文件夹
mkdir 名称{1..100} # 当前文件夹下创建100个目录
mkdir -p a/b  # 循环创建
```

4. `rmdir`和上面相对的（删除一个目录）

3.文件相关命令

1. `ls`查看文件目录

```
ls -a # 显示指定目录下所有子目录与文件，包含隐藏文件
ls -l # 以列表方式显示
ls -h # 显示文件大小
```

2. `touch`创建文件

```
# 1. 如果文件不存在，创建新文件
# 2. 如果文件存在，更新文件时间
touch filename
touch file{1..100}
```

3. `cp`拷贝

```
cp file1 file2 dirs/    # 将文件1,2拷贝到dirs目录下
cp -r dir   # 循环拷贝目录
```

4. `rm`删除

```
rm -i 文件    # 删除时提示
rm -r 文件     # 删除非空目录
rm -f 文件    # 强制删除
```

5. `mv`移动

```
mv file1 file2
```

4.文件内容查看命令

1. `cat`将文件内容一次性输出到终端

```
cat file
cat -n file # 对输出行进行编号
cat -b file # 对输出行进行编号，空行不进行编号
```

2. `less`查看内容
3. `head`查看前面

```
head -n file # n表示显示前面的几行
head -30 a.txt
```

4. `tail`从文件尾开始显示
5. `du` 和 `df`查看某个目录大小

```
du -sh file    # 显示指定文件或目录占用的数据快
df -h # 显示占用空间
```

5.查找相关命令

1. `find`查找

```
# 按文件名查找
find 路劲 -name filename
# 按大小
find 路径 -size +100k
# 大于用+ 小于用- k必须用小写

# 按文件类型查找
find 路径 -type 类型
```

2. `grep`文本搜索工具

```
grep -n "关键字" 路径
# 加-n 会显示行号
```

3. `|`管道

```
输出 | 输入
ubuntu@ubuntu:~$ ifconfig | grep "192"
        inet 192.168.3.217  netmask 255.255.255.0  broadcast
192.168.3.255
```

6.打包及压缩

1. `tar` 打包

```
# bagname.tar 打包文件名
# file{1..10} 归档的文件
tar -cvf bagname.tar file{1..10}
# 解除归档文件
tar -xvf bagname.tar
# 列出归档文件中的内容
tar -tvf bagname.tar
```

2. `gzip` 压缩-----`bzip2`和`gzip`一样

```
gzip -d filename.gz # 解压
gunzip file.gz # 解压
gzip -r filename # 压缩所有子目录
gzip f1 f2 f3 f4 # 压缩指定文件(只能压缩文件)，目录得归档之后才能压缩
```

3. `tar`和`gzip`和`bzip2`结合使用

```
# 目录得先打包再压缩,一步到位
tar -czvf file.tar.gz file{1..10} # gzip
tar -cjvf file.tar.bz2 file{1..10} # bzip2
# 解压
tar -xzvf file.tar.gz # gzip
tar -xjvf file.tar.gz # bzip2
tar -xvf file.tar.gz # 万能解压
# 解压到指定目录下,后面加-C和目录
tar -xzvf file.tar.gz -C ./xxx/

```

4. `zip`和`unzip`打包压缩

```
zip filename file{1..10} # 压缩
unzip filename # 解压
unzip -d bag filename # filename解压到目录bag中
```

7.用户权限管理及软件安装

1. 文件权限
    1. 所有者

    > 只允许用户自己访问

    2. 用户组

    > 允许同组人访问

3. 其他用户

> 允许其他人访问

4. 访问权限说明

> `r`读 `w`写 `x`执行

2. `chmod`修改问价权限

[u/g/o/a]	含义
u	user表示个文件的所有者
g	group表示该文件属于哪一个组
o	other表示其他以外的人
a	all表示这三者皆是

```
# +表示加权限 -表示减权限
chmod u/g/o/a+r/w/x filename
chmod u=rwx,g=rwx,o=rwx filename
chmod 0777 filename
```

3. `chown`修改文件所有者

```
sudo chown :ubuntu file.txt # 修改文件所属组
sudo chown ubuntu file.txt # 修改文件所属者
```

4. 软件安装

在线安装

```
sudo apt install 软件名 # 安装软件
sudo apt remove 软件名 # 卸载
sudo apt updata 软件名 # 更新软件
sudo apt clean 软件名

#案例
sudo apt install sl # 安装软件
# 调用sl
sl
# 卸载
sudo apt remove sl # 卸载
```

离线安装

```
# 安装
sudo dpkg -i xxx.deb
# 卸载
sudo dpkg -r 软件名
```

补：

1. `>` 重定向

```
ll > b.txt # 会覆盖之前内容
ll >> b.txt # 会追加到后面
# 错误脚本会重定向到黑洞/dev/null  后面要加数字2
lll 2> /dev/null
lll 2>> /dev/null
```

2. 链接

软链接：不占用内存

硬链接：占用磁盘空间

```
# ln 源文件 链接文件 硬链接
ln file.txt file
# ln -s 源文件 链接文件 符号链接
ln -s file.txt f_hard
```

3. 关机指令

```
sudo shutdown -h now
```

三、Linux下编程

1.vim

1. vim三种模式

> 命令模式：任何情况下，不管用户处于何种模式，只要按`ESC`就可以进入命令模式。
>
> 编辑模式：打开命令模式，输入`i(I)`、附加命令`a(A)`、打开命令`o(O)`、替换命令`s(S)`都可以进入编辑模式
>
> 末行模式：可以显示行号是`:`

2. `vim`基本操作

```
vim filename.txt   # 文件存在，打开，不存在关闭
```

3. 保存文件

```
# 第一步 进入命令模式 ESC
# 第二步 Shift+zz
```

4. 编辑模式

切换编辑模式

按键	功能
i	光标位置当处插入文字
I	光标所有行首插入文字
o	光标下一行插入文字（新行）
O	光标上一行插入文字（新行）
a	光标位置右边插入新行
A	光标所在行尾插入文字
s	删除光标后边的字符，从当前位置插入
S	删除光标所在当前行，从行首插入

> 以下都是在命令模式下操作

光标移动

按键	功能
gg	到文件第一行
G	到文件最后一行
mG和mgg	到文件指定行，m为行数
0	当前行行首
$	当前行行尾
h/l/k/j	左，右，上，下移光标

复制粘贴

按键	功能
[n]yy	复制从当前行开始的n行
p	把站跳板上的诶人插入到当前行

删除

按键	功能
[n]x	删除光标后n个字符
[n]X	删除光标前n个字符
D	删除光标所在开始当此行尾的字符
[n]dd	删除从当前行开始的n行（也叫剪切p就可以粘贴出来）
dG	删除光标所在开始到文件尾的所有字符
dw	删除光标开始位置的字符，包含光标所在字符
d0	删除光标当前行内容
dgg	删除光标所在开始到文件首行第一个字符开始的所有字符

撤销和恢复

按键	功能
.(点)	执行上一次
u	撤销上一次操作
ctrl+r	反撤销
100+.	执行上一次操作100次

查找和替换

按键	功能
/字符串	当前光标位置向下查找（n向下查找，N向上查找）
?字符串	从当前光标位置向上查找（n向上查找，N向下查找）
r	替换当前字符
R	替换当前行光标后的字符

可视模式

按键	功能
v	按字符移动，选中文本，可配合h/l/k/j选择内容，使用d删除，使用y复制
shift+v	行选
Ctrl+v	列选

> 末行模式

1. **保存退出**

按键	功能
:wq	保存退出
:x	保存退出
:w filename	保存指定文件
:q	退出，不保存，文件修改的话，不让你退出
:!q	强制退出，不保存

2. **替换**

按键	功能
:s/abc/ABC	光标第一次abc换成大写ABC
:s/abc/ABC/g	光标第一次abc换成大写ABC指当前行所有内容
1,10:s/abc/ABC/g	指定行号abc换成ABC

2.gcc

1. 编译生成可执行文件

```
gcc filename.c -o filename
```

2. 查看版本号

```
gcc -v
```

3. 其他操作

```
gcc -g filename.c -o filename # 包含调试信息
gcc -Wall filename.c -o filename # 显示警告
```

3.文件操作

> 在 Linux 的世界里，一切设备皆文件。我们可以系统调用中 I/O 的函数（I：input，输入；O：output，输出），对文件进行相应的操作（ open()、close()、write() 、read() 等）。

1. 常用文件IO函数
    1. `open` 函数

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

```
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
功能：
    打开文件，如果文件不存在则可以选择创建。
参数：
    pathname：文件的路径及文件名
    flags：打开文件的行为标志，必选项 O_RDONLY, O_WRONLY, O_RDWR
    mode：这个参数，只有在文件不存在时有效，指新建文件时指定文件的权限
返回值：
    成功：成功返回打开的文件描述符
    失败：-1
```

2. `close` 函数

```
#include <unistd.h>
int close(int fd);
功能：
    关闭已打开的文件
参数：
    fd : 文件描述符，open()的返回值
返回值：
    成功：0
    失败：  -1, 并设置errno
```

3. `write` 函数

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
功能：
    把指定数目的数据写到文件（fd）
参数：
    fd :  文件描述符
    buf : 数据首地址
    count : 写入数据的长度（字节）
返回值：
    成功：实际写入数据的字节个数
    失败：  - 1
```

4. `read` 函数

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
功能：
    把指定数目的数据读到内存（缓冲区）
参数：
    fd : 文件描述符
    buf : 内存首地址
    count : 读取的字节个数
返回值：
    成功：实际读取到的字节个数
    失败：  - 1
```

5. `lseek`函数

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
功能：
    改变文件的偏移量
参数：
    fd：文件描述符
    offset：根据whence来移动的位移数（偏移量），可以是正数，也可以负数，如果正数，则相对于whence往右移动，如果是负数，则相对于whence往左移动。如果向前移动的字节数超过了文件开头则出错返回，如果向后移动的字节数超过了文件末尾，再次写入时将增大文件尺寸。

    whence：其取值如下：
        SEEK_SET：从文件开头移动offset个字节
        SEEK_CUR：从当前位置移动offset个字节
        SEEK_END：从文件末尾移动offset个字节
返回值：
    若lseek成功执行，则返回新的偏移量
    如果失败，  返回-1
```

四、进程

1.基础

1. 进程和程序

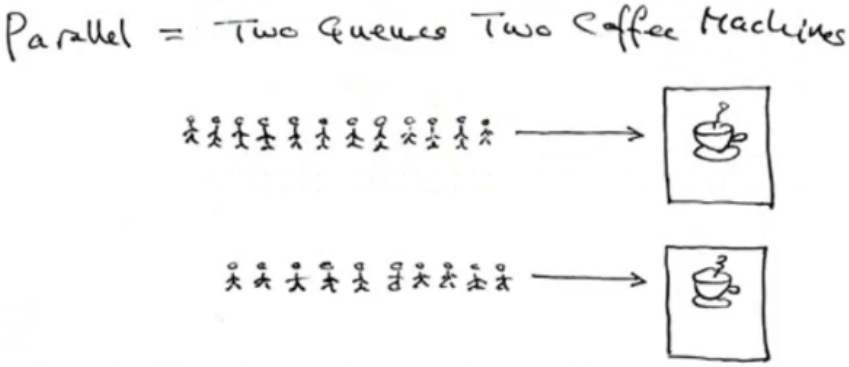
> 程序是存放在存储介质上的可执行文件，**进程是程序执行的过程**，进程的状态是变化的，器包括经常的创建，调度和消亡，进程是动态的。**进程是管理事务的基本单元**

2. 单道，多道程序设计

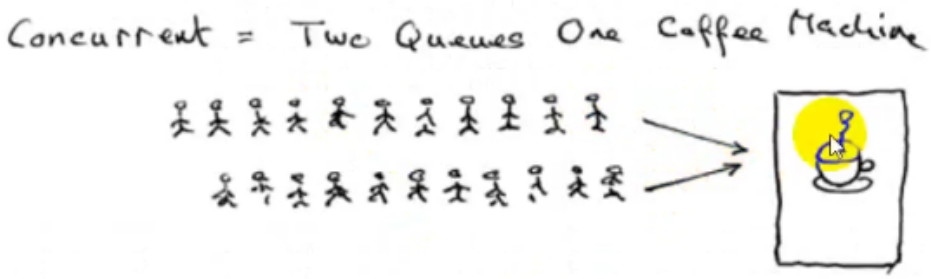
1. 单道程序设计，是所有进程一个一个排队执行，若A阻塞，B只能等待，即CPU处于空闲状态。所以在系统资源利用上不合理。
2. 多道程序设计，在计算机中，时钟中断及为多道程序设计模型的理论基础。（时间片轮转法）

3. 并行和并发

并行：指在同一时刻，有多条指令在多个处理器上同时执行。

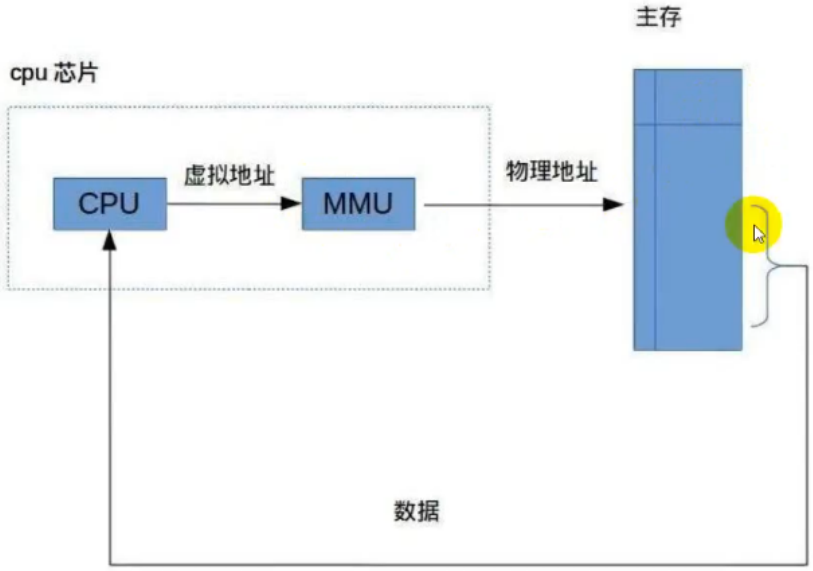


并发：同一时刻只能指向一条指令，但是多个进程指令被快熟的轮换执行，使得在宏观上具有多个基础同时执行的效果。



4. MMU内存管理单元

> 内存管理单元，他是中央处理器用来管理虚拟存储器，物理出出气的控制线路，同时负责虚拟地址映射为物理地址，以及提供硬件机制的内存访问授权，多用户多进程操作系统。



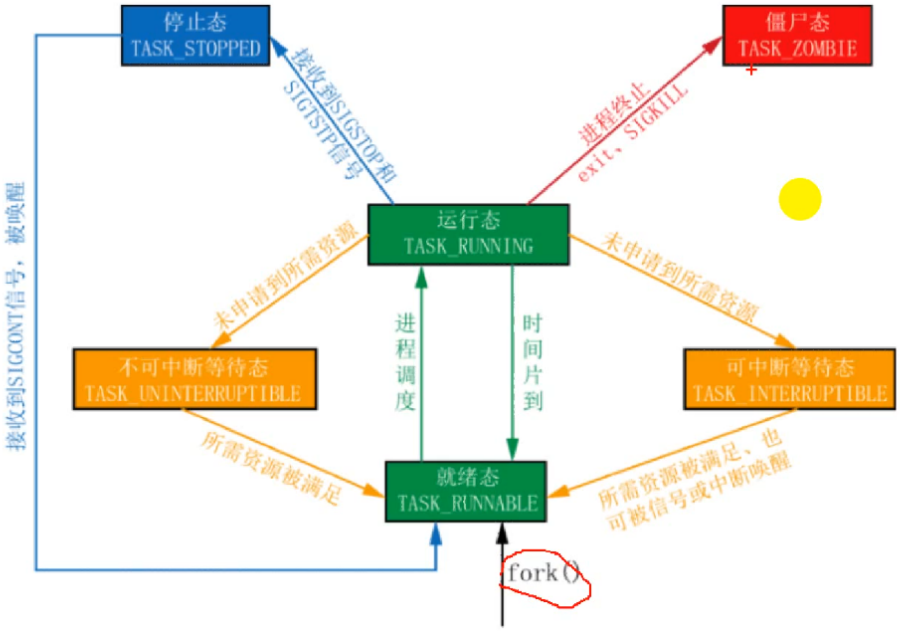
5. 进程控制块PCB

> 其内部成员很多，我们需要掌握以下内容

- 进程id：系统中么一个进程有唯一的id。
- 进程状态：有就绪，运行，挂起，停止等状态。
- 进程切换时需要保存和恢复的一些CPU寄存器，保存线程和恢复线程。
- 描述虚拟地址信息和控制终端的信息
- 当前工作目录
- umask暗码
- 文件描述符表，包含很多指向file结构体指针
- 和信号相关的信息
- 用户id和组id
- 会话和进程组

6. 进程状态

> 三态模型：运行态，就绪态，阻塞态。
>
> 五态模型：新建态，终止态，运行态，就绪态，阻塞态。



7. 查看进程命令

```
ps aux
```

2.进程相关命令

1. `ps -a`

```
ubuntu@ubuntu:~$ ps -a
  PID TTY          TIME CMD
12142 pts/0    00:00:00 ps
# PID   进程号
# TTY   终端
# TIME  时间
# CMD   命令
```

2. `ps -au` 显示进程详细信息

```
ubuntu@ubuntu:~$ ps -au
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME
COMMAND
root      1973  0.0  0.0   5344  1608 ttyS0    Ss+  00:28   0:00
/sbin/agetty -o -p -- \u --keep-baud 1
```

3. `ps -aux` 一般用这个
4. `top` 动态显示运行的进程

```
ubuntu@ubuntu:~$ top
# 按q退出
```

5. `kill PID`

```
ubuntu@ubuntu:~$ sleep 3000 &
[1] 17443
ubuntu@ubuntu:~$ ps -a
  PID TTY          TIME CMD
17443 pts/0    00:00:00 sleep
17444 pts/0    00:00:00 ps
ubuntu@ubuntu:~$ kill 17443
ubuntu@ubuntu:~$ ps -a
  PID TTY          TIME CMD
17445 pts/0    00:00:00 ps
[1]+  已终止               sleep 3000
ubuntu@ubuntu:~$ kill -9 17443 # 强制杀死
```

6. `killall -9 进程名`

```
ubuntu@ubuntu:~$ ps
  PID TTY          TIME CMD
10696 pts/0    00:00:00 bash
17452 pts/0    00:00:00 sleep
17453 pts/0    00:00:00 sleep
17454 pts/0    00:00:00 sleep
17457 pts/0    00:00:00 ps
ubuntu@ubuntu:~$ killall -9 sleep
[1]   已杀死               sleep 3000
[2]-  已杀死               sleep 3000
[3]+  已杀死               sleep 3000
ubuntu@ubuntu:~$ ps -a
  PID TTY          TIME CMD
17467 pts/0    00:00:00 ps
```

3.进程号和相关函数

> 进程号可以被重用，不是唯一的，但是进程号在被使用后，就不能给其他端口用相同进程号

1. 进程号 `PID`

> 标识进程的一个非法=负整数

2. 父进程号 `PPID`

> 任何进程（除init进程）都是有由另一个进程创建，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。如A进程创建了B进程，A的进程号就是B进程的父进程号。

3. 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个中断的各种信号，**关联的进程有一个进程组号**。简单来说就，进程组号就相当于QQ群号，进程组就相当于QQ群，主要方便管理。

4. `gitpid`函数：获取本进程的进程号(PID)

```
#include <sys/types.h>
#includde <unistd.h>
pid_t gitpid(void);
/*
功能：获取本进程的进程号(PID)
参数：无
返回值：本进程号
*/
```

5. `gitppid`函数：获取本进程的父进程号(PPID)

```
#include <sys/types.h>
#includde <unistd.h>
pid_t gitppid(void);
/*
功能：获取本进程的父进程号(PPID)
参数：无
返回值：父进程号
*/
```

6. `gitpgid`函数：获取本进程的进程组号(PGID)

```
#include <sys/types.h>
#includde <unistd.h>
pid_t gitpgid(pid_t pid);
/*
功能：获取本进程的进程组号(PGID)
参数：要获取的进程号的组号
返回值：进程组号
*/
```

案例：

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main() {
    printf("本进程号:%d\n",getpid());
    printf("父进程号:%d\n",getppid());
    printf("进程号组:%d\n",getpgid(getpid()));
    return 0;
```

```
}
/*
ubuntu@ubuntu:~/Learn_Linux$ ./a.out
本进程号:17680
父进程号:10696
进程号组:17680
*/
```

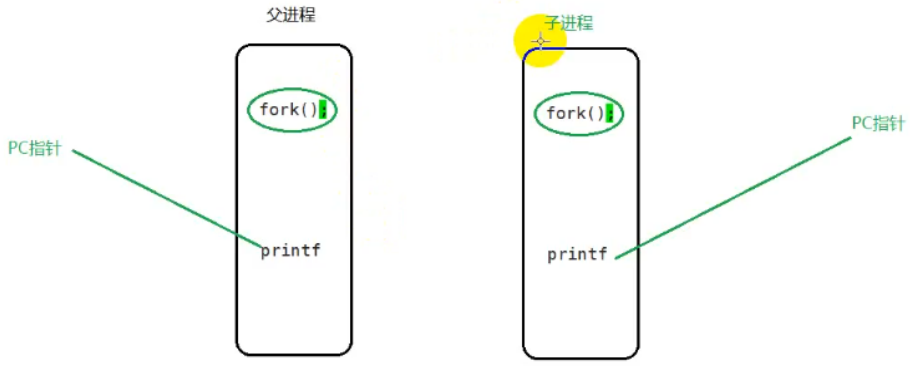
7. `fork`进程创建

```
#include <sys/types.h>
#includde <unistd.h>
pid_t fork(void);
/*
功能：用于从一个已经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
参数：无
返回值：
    成功:子进程中返回0,父进程中返回子进程ID。pid_t为整型
    失败:返回-1
    失败的两个原因:
        进程已经达到上限
        系统内存不足
*/
```

子进程一般都是父进程的克隆体,且执行fork之后的语句，不会从main开始执行

案例：

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main() {
    fork();
    printf("hello world\n");
    return 0;
}
```



8. `exit` 进程退出函数

```c
#include <stdlib.h>
void exit(int status);

#include <unistd.h>
void _exit(int status);
/*
功能：结束调用次函数进程，不会刷新缓冲区
参数：返回给父进程的参数(低8位有效)，根据需求填写
返回值：无
*/
```

9. `wait()或者waitpid()` 得到退出状态同时彻底清除掉这个进程

> wait()和waitpid()功能一样，区别在于，前者函数会阻塞，后者可以设置不阻塞，后者还可以指定等待那个进程结束

```c
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
/*
功能：等待任意一个子进程结束，如果任意一子进程结束了，此函数会回收进程资源
参数：进程退出时的状态
返回值：
    成功：已经结束子进程的进程号
    失败：-1
*/
```

案例：

```c
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int main() {
    int statue = 0;
    pid_t pid = -1;
    // 创建子进程
    pid = fork();
    if(pid == -1){
        printf("子进程创建失败");
        return 1;
    }
    // 子进程
    if(pid == 0) {
```

```
        for(int i = 0; i < 15 ; i++){
            printf("子进程id:%d, %d\n",getpid(),i);
            sleep(1);
        }
        // 子进程终止，如果这里不写终止进程，就会接着运行下面的内容
        exit(10);
    }
    // 父进程执行
    printf("父进程等待子进程退出，回收其资源\n");
    // 阻塞方式
    int ret =  wait(&statue);
    if(ret == -1){
        printf("wait");
        return 1;
    }
    printf("父进程回收字进程资源...\n");
    return 0;
}
```

宏函数可分为三组

- `WIFEXITED(status)`

  > 为非0，进程正常结束

  若宏为真，使用 `WEXITSTATUS(status)` 获取进程退出状态（exit的参数）

- `WIFSIGNALED(status)`

  > 为非0，进程异常终止

  若宏为真，使用 `WTERMSIG(status)` 获取使进程终止的那个信号的编号。

- `WIFSTOPPED(status)`

  > 为非0，进程处于暂停状态

  若宏为真，使用`WSTOPSIG(status)`，取得使进程暂停的那个信号的编号

  若宏为真，使用`WIFCONTINUED(status)`，进程暂停后已经继续运行

```c
if(WIFEXITED(statue)){
    // 进程正常结束会进入
    printf("进程正常结束，返回状态为%d\n",WEXITSTATUS(statue));
}else if(WIFSIGNALED(statue)){
    // 异常终止会进入
    printf("程序异常终止，终止信号%d\n",WTERMSIG(statue));
}else if(WIFSTOPPED(status)){
    // 向指定进程发送暂停信号
    // kill -19 PID 暂停
    // kill -18 PID 启动
    printf("子程序被信号%d暂停\n",WSTOPSIG(status));
}
```

`waitpid()`函数

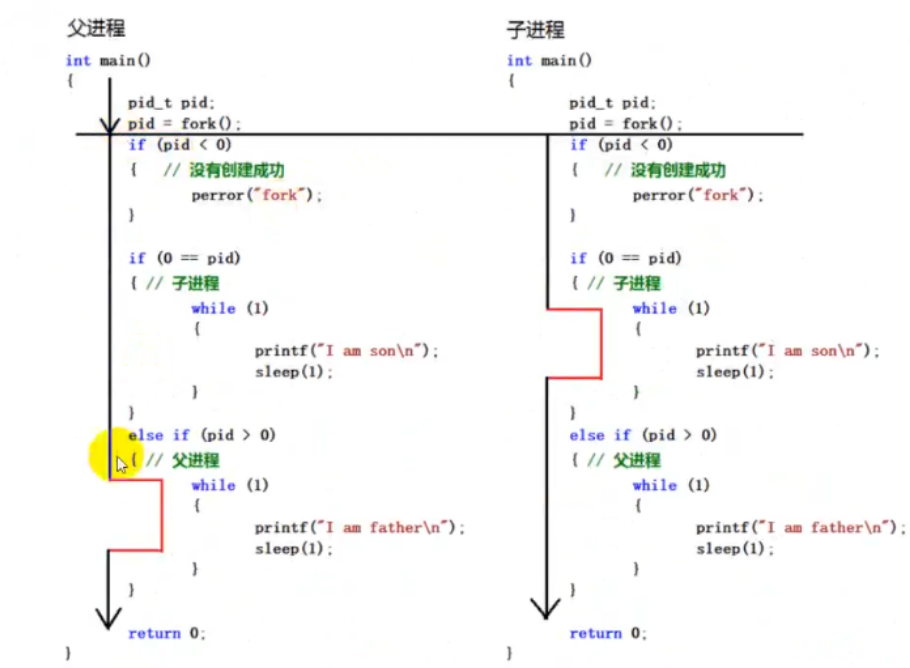
```c
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid,int *status,int options);
/*
功能：等待子进程终止，如果子进程结束了，此函数会回收进程资源
参数：
    pid:参数pid的值有以下几种情况
        pid>0 等待的继承id
        pid=0 等待同一个进程组中的任何子进程
        pid=-1 等待任意子进程，现在和wait一样
        pid<-1 等待指定进程组中的任何子进程，这个进程组的ID等于pid的绝对值
    status:进程退出时的状态信息
    options:提供了一下额外选项来控制
        0：同wait()
        WNOHANG:没有任何已经结束的子进程，则立即返回
        WUNTRACED:进程暂停立马返回。
返回值：
    1.正常返回，返回收集到的进程的进程号
    2.设置WNOHANG，发现没有发一句退出的进程可等待，则返回0
    3.出错返回-1
*/
```

4.父子进程关系

1. 区分父子进程

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

```c
#include <unistd.h>
#include <stdlib.h>
int main() {
    pid_t pid = fork();
    if (pid==0){
        printf("hello zhf\n");
        printf("子进程：%d,父进程：%d\n",getpid(),getppid());
        exit(0);
    }else{
        printf("hello world\n");
        printf("父进程：%d,子进程%d\n",getpid(),pid);
    }
    return 0;
}
```




```
deng@itcast:~/share/5th$ gcc 8fork.c
deng@itcast:~/share/5th$ ./a.out
子进程hello itcast pid:104394 ppid:104393
父进程hello world pid:104393 cpid: 104394
子进程hello itcast pid:104394 ppid:104393
父进程hello world pid:104393 cpid: 104394
父进程hello world pid:104393 cpid: 104394
子进程hello itcast pid:104394 ppid:104393
```

2. 父子进程地址空间

> 父子进程地址空间遵循：读时共享，写时拷贝

5.孤儿进程和僵尸进程

> 孤儿进程：父进程结束了，而它的一个或多个子进程还在运行，那么这些子进程就成为孤儿进程(father died)。子进程的资源由init进程(进程号PID = 1)回收。
>
> 僵尸进程：子进程退出了，但是父进程没有用wait或waitpid去获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中，这种进程称为僵死进程。

解决方案

1)kill杀死元凶父进程(一般不用)

严格的说，僵尸进程并不是问题的根源，罪魁祸首是产生大量僵死进程的父进程。因此，我们可以直接除掉元凶，通过kill发送SIGTERM或者SIGKILL信号。元凶死后，僵尸进程进程变成孤儿进程，由init充当父进程，并回收资源。

或者运行：kill -9 父进程的pid值。

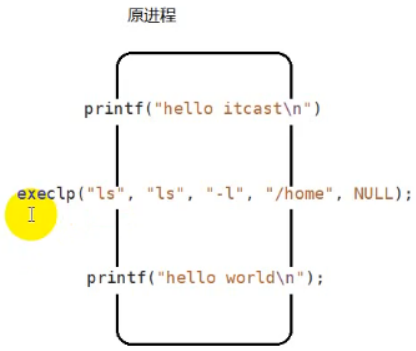
2)父进程用wait或waitpid去回收资源(方案不好)

父进程通过wait或waitpid等函数去等待子进程结束，但是不好，会导致父进程一直等待被挂起，相当于一个进程在干活，没有起到多进程的作用。

3)通过信号机制，在处理函数中调用wait，回收资源

通过信号机制，子进程退出时向父进程发送SIGCHLD信号，父进程调用signal(SIGCHLD,sig_child)去处理SIGCHLD信号，在信号处理函数sig_child()中调用wait进行处理僵尸进程。什么时候得到子进程信号，什么时候进行信号处理，父进程可以继续干其他活，不用去阻塞等待。

6.进程替换



```c
int execl(const char *path, const char *arg, .../* (char  *) NULL */);
int execlp(const char *file, const char *arg, .../* (char  *) NULL */);
int execle(const char *path, const char *arg, .../*(char *) NULL, char * const envp[] */);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],char *const envp[]);
```

五、进程间通信

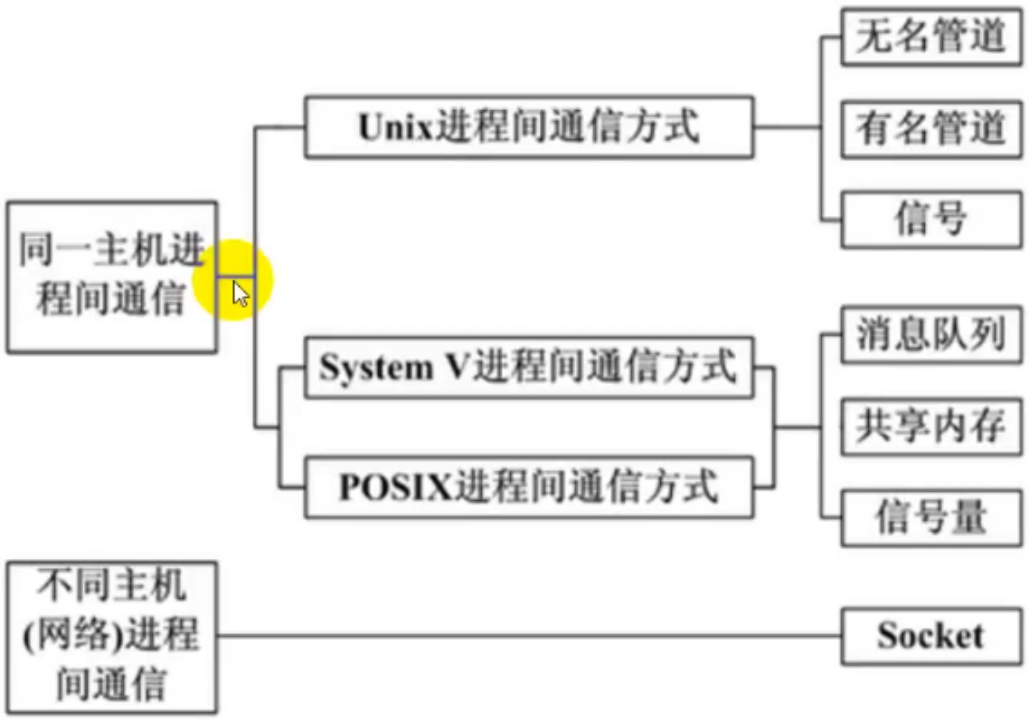
1.概念

> 进程不是孤立的，不同进程需要进程信息的交互和状态的传递等，因此需要进程间通信（IPC）

1. 目的：

> 数据传输、通知事件、资源共享、进程控制

2. 通信机制



2.无名管道

> 是操作系统中最古老的通信方式，所有操作系统都支持的一种通信机制。

1. 管道特点

```
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（父进程与子进程，或者两个兄弟进程，具有亲缘关系）之间使用。
```

2. `pipe` 创建无名管道

```
#include <unistd.h>
int pipe(int pipefd[2]);
功能：创建无名管道。
参数：
    pipefd ： 为 int 型数组的首地址，其存放了管道的文件描述符 pipefd[0]、pipefd[1]。

    当一个管道建立时，它会创建两个文件描述符 fd[0] 和 fd[1]。其中 fd[0] 固定用于读管道，而 fd[1] 固定用于写管道。一般文件 I/O的函数都可以用来操作管道(lseek() 除外)。
返回值：
    成功：0
    失败：-1
```

案例：

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
int main() {
    // 创建无名管道
    int pipe_fb[2] = {0};
    if(pipe(pipe_fb) == -1) {
        printf("无名管道创建失败\n");
    }
    // 创建子进程
```

```
    pid_t pid = fork();
    // 子进程内容执行
    if(pid == 0){
        // 对无名管道写入数据
        char buf[] = "hello zhf";
        write(pipe_fb[1],buf,sizeof(buf));
        // 手动关闭无名管道
        close(pipe_fb[0]);
        close(pipe_fb[1]);
        // 退出子进程，并返回状态码为10
        exit(10);
    } else {
        // 当pid非0时，为父进程
        printf("父进程等待子进程退出，回收其资源\n");
        // 等待子进程结束，并回收子进程，和子进程退出状态
        // wait返回值是子进程的pid
        int* status_son = (int *)malloc(sizeof(int *));
        int val = wait(status_son);
        char buf[20] = {0};
        read(pipe_fb[0],buf,9);
        printf("%s\n",buf);
        // 手动关闭无名管道
        close(pipe_fb[0]);
        close(pipe_fb[1]);
    }
    return 0;
}
```

3. 查看管道缓冲区大小

bash指令查看

```
ulimit -a
```

函数查看

```
#include <unistd.h>
long fpathconf(int fd, int name);
功能：该函数可以通过name参数查看不同的属性值
参数：
    fd：文件描述符
    name：
        _PC_PIPE_BUF，查看管道缓冲区大小
        _PC_NAME_MAX，文件名字字节数的上限
返回值：
    成功：根据name返回的值的意义也不同。
    失败：  -1
```

4. 非阻塞设置

```
// 获取原来的flags
int flags = fcntl(fd[0],F_GETFL);
// 设置新的flags
flags |= O_NONBLOCK;
fcntl(fb[0],F_SETFL,flags);
```

3.有名管道

> 管道，由于没有名字，只能用于亲缘关系的进程间通信。为了克服这个缺点，提出了命名管道（FIFO），也叫有名管道、FIFO文件。
>
> 命名管道（FIFO)和无名管道（pipe）有一些特点是相同的，不一样的地方在于：
>
> 1. 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。
> 2. 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。
> 3. FIFO 有名字，不相关的进程可以通过打开命名管道进行通信。

1. 通过命令创建无名管道

```
mkfifo fifoname  // 创建有名管道
```

2. 通过函数创建

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
功能：
    命名管道的创建。
参数：
    pathname : 普通的路径名，也就是创建后 FIFO 的名字。
    mode : 文件的权限，与打开普通文件的 open() 函数中的 mode 参数相同。(0666)
返回值：
    成功：0   状态码
    失败：如果文件已经存在，则会出错且返回 -1。
```

3. 读写

```
//进行1，写操作
int fd = open("my_fifo", O_WRONLY);

char send[100] = "Hello Mike";
write(fd, send, strlen(send));

//进程2，读操作
int fd = open("my_fifo", O_RDONLY);//等着只写

char recv[100] = { 0 };
//读数据，命名管道没数据时会阻塞，有数据时就取出来
read(fd, recv, sizeof(recv));
printf("read from my_fifo buf=[%s]\n", recv);
```

4.共享内存

1. 概述

> 存储映射I/O (Memory-mapped I/O) 使一个磁盘文件与存储空间中的一个缓冲区相映射。
>
> 于是当从缓冲区中取数据，就相当于读文件中的相应字节。于此类似，将数据存入缓冲区，则相应的字节就自动写入文件。这样，就可在不适用read和write函数的情况下，使用地址（指针）完成I/O操作。
>
> 共享内存可以说是最有用的进程间通信方式，也是最快的IPC形式, 因为进程可以直接读写内存，而不需要任何数据的拷贝。

2. 存储映射函数

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

```
功能：
    一个文件或者其它对象映射进内存
参数：
    addr ：  指定映射的起始地址，通常设为NULL，由系统指定
    length：映射到内存的文件长度
    prot：   映射区的保护方式，最常用的 ：
        a）读：PROT_READ
        b）写：PROT_WRITE
        c）读写：PROT_READ | PROT_WRITE
    flags：   映射区的特性，可以是
        a）MAP_SHARED ：写入映射区的数据会复制回文件，且允许其他映射该文件的进程共享。
        b）MAP_PRIVATE ：对映射区的写入操作会产生一个映射区的复制(copy - on - write)，对此区域所做的修改不会写回原文件。
    fd：由open返回的文件描述符，代表要映射的文件。
    offset：以文件开始处的偏移量，必须是4k的整数倍，通常为0，表示从文件头开始映射
返回值：
    成功：返回创建的映射区首地址
    失败：MAP_FAILED宏
```

案例：

```
int fd = open("xxx.txt", O_RDWR); //读写文件
int len = lseek(fd, 0, SEEK_END);   //获取文件大小

//一个文件映射到内存，ptr指向此内存
void * ptr = mmap(NULL, len, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
if (ptr == MAP_FAILED)
{
    perror("mmap error");
    exit(1);
}

close(fd); //关闭文件

char buf[4096];
printf("buf = %s\n", (char*)ptr); // 从内存中读数据，等价于从文件中读取内容

strcpy((char*)ptr, "this is a test");//写内容

int ret = munmap(ptr, len);
if (ret == -1)
```

```
{
    perror("munmap error");
    exit(1);
}
```

3. 匿名映射

> 通过使用我们发现，使用映射区来完成文件读写操作十分方便，父子进程间通信也较容易。但缺陷是，每次创建映射区一定要依赖一个文件才能实现。
>
> 通常为了建立映射区要open一个temp文件，创建好了再unlink、close掉，比较麻烦。 可以直接使用匿名映射来代替。
>
> 其实Linux系统给我们提供了创建匿名映射区的方法，无需依赖一个文件即可创建映射区。同样需要借助标志位参数flags来指定。
>
> 使用**MAP_ANONYMOUS (或MAP_ANON)**。

```
int *p = mmap(NULL, 4, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANONYMOUS, -1, 0);

// 案例
// 创建匿名内存映射区
int len = 4096;
void *ptr = mmap(NULL, len, PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANON, -1, 0);
if (ptr == MAP_FAILED) {
    perror("mmap error");
    exit(1);
}
// 创建子进程
pid_t pid = fork();
if (pid > 0) { //父进程
    // 写数据
    strcpy((char*)ptr, "hello mike!!");
    // 回收
    wait(NULL);
}
else if (pid == 0) { //子进程
    sleep(1);
    // 读数据
    printf("%s\n", (char*)ptr);
}
// 释放内存映射区
int ret = munmap(ptr, len);
if (ret == -1) {
    perror("munmap error");
```

```
28      exit(1);
29  }
```

六、信号

1.基本概念

> 信号是 Linux 进程间通信的最古老的方式。信号是软件中断，它是在软件层次上对中断机制的一种模拟，是一种异步通信的方式 。信号可以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断，转而处理某一个突发事件。

1. 信号的编号

> 1）信号编号：
>
> Unix早期版本就提供了信号机制，但不可靠，信号可能丢失。Berkeley 和 AT&T 都对信号模型做了更改，增加了可靠信号机制。但彼此不兼容。POSIX.1对可靠信号例程进行了标准化。
>
> Linux 可使用命令：kill -l（"l" 为字母），查看相应的信号。
>
> 不存在编号为0的信号。其中1-31号信号称之为常规信号（也叫普通信号或标准信号），34-64称之为实时信号，驱动编程与硬件相关。名字上区别不大。而前32个名字各不相同。

2)常规信号

编号	信号	对应事件	默认动作
1	SIGHUP	用户退出shell时，由该shell启动的所有进程将收到这个信号	终止进程
2	**SIGINT**	当用户按下了 **<Ctrl+C>** 组合键时，用户终端向正在运行中的由该终端启动的程序发出此信号	终止进程
3	**SIGQUIT**	用户按下 **<ctrl+\>** 组合键时产生该信号，用户终端向正在运行中的由该终端启动的程序发出些信号	终止进程
4	SIGILL	CPU检测到某进程执行了非法指令	终止进程并产生core文件
5	SIGTRAP	该信号由断点指令或其他 trap指令产生	终止进程并产生core文件
6	SIGABRT	调用abort函数时产生该信号	终止进程并产生core文件
7	SIGBUS	非法访问内存地址，包括内存对齐出错	终止进程并产生core文件
8	SIGFPE	在发生致命的运算错误时发出。不仅包括浮点运算错误，还包括溢出及除数为0等所有的算法错误	终止进程并产生core文件

编号	信号	对应事件	默认动作
9	SIGKILL	无条件终止进程。本信号不能被忽略，处理和阻塞	终止进程，可以杀死任何进程
10	SIGUSE1	用户定义的信号。即程序员可以在程序中定义并使用该信号	终止进程
11	**SIGSEGV**	指示进程进行了无效内存访问(段错误)	终止进程并产生core文件
12	SIGUSR2	另外一个用户自定义信号，程序员可以在程序中定义并使用该信号	终止进程
13	**SIGPIPE**	Broken pipe向一个没有读端的管道写数据	终止进程
14	SIGALRM	定时器超时，超时的时间 由系统调用alarm设置	终止进程
15	SIGTERM	程序结束信号，与SIGKILL不同的是，该信号可以被阻塞和终止。通常用来要示程序正常退出。执行shell命令Kill时，缺省产生这个信号	终止进程
16	SIGSTKFLT	Linux早期版本出现的信号，现仍保留向后兼容	终止进程
17	**SIGCHLD**	子进程结束时，父进程会收到这个信号	忽略这个信号
18	SIGCONT	如果进程已停止，则使其继续运行	继续/忽略
19	SIGSTOP	停止进程的执行。信号不能被忽略，处理和阻塞	为终止进程

编号	信号	对应事件	默认动作
20	SIGTSTP	停止终端交互进程的运行。按下<ctrl+z>组合键时发出这个信号	暂停进程
21	SIGTTIN	后台进程读终端控制台	暂停进程
22	SIGTTOU	该信号类似于SIGTTIN，在后台进程要向终端输出数据时发生	暂停进程
23	SIGURG	套接字上有紧急数据时，向当前正在运行的进程发出些信号，报告有紧急数据到达。如网络带外数据到达	忽略该信号
24	SIGXCPU	进程执行时间超过了分配给该进程的CPU时间 ，系统产生该信号并发送给该进程	终止进程
25	SIGXFSZ	超过文件的最大长度设置	终止进程
26	SIGVTALRM	虚拟时钟超时时产生该信号。类似于SIGALRM，但是该信号只计算该进程占用CPU的使用时间	终止进程
27	SGIPROF	类似于SIGVTALRM，它不公包括该进程占用CPU时间还包括执行系统调用时间	终止进程
28	SIGWINCH	窗口变化大小时发出	忽略该信号
29	SIGIO	此信号向进程指示发出了一个异步IO事件	忽略该信号
30	SIGPWR	关机	终止进程
31	SIGSYS	无效的系统调用	终止进程并产生core文件

编号	信号	对应事件	默认动作
34~64	SIGRTMIN ~ SIGRTMAX	LINUX的实时信号，它们没有固定的含义（可以由用户自定义）	终止进程

3. 信号的状态
    1. **产生**
    2. **未决状态：没有被处理**
    3. **递达状态：信号被处理了**
4. 阻塞信号集和未决信号集
    1. 信号的实现手段导致信号有很强的延时性，但对于用户来说，时间非常短，不易察觉。
    2. Linux内核的进程控制块PCB是一个结构体，task_struct, 除了包含进程id，状态，工作目录，用户id，组id，文件描述符表，还包含了信号相关的信息，主要指**阻塞信号集和未决信号集**。
    3. **阻塞信号集(信号屏蔽字)**

        将某些信号加入集合，对他们设置屏蔽，当屏蔽x信号后，再收到该信号，该信号的处理将推后(处理发生在解除屏蔽后)。
    4. **未决信号集**

        信号产生，未决信号集中描述该信号的位立刻翻转为1，表示信号处于未决状态。当信号被处理对应位翻转回为0。这一时刻往往非常短暂。

        信号产生后由于某些原因(主要是阻塞)不能抵达。这类信号的集合称之为未决信号集。在屏蔽解除前，信号一直处于未决状态。

2.信号参数函数

1. kill函数

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
功能：给指定进程发送指定信号(不一定杀死)
参数：
    pid : 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。
```

```
    sig : 信号的编号，这里可以填数字编号，也可以填信号的宏定义，可以通过命令 kill - l("l" 为字母)进行相应查看。不推荐直接使用数字，应使用宏名，因为不同操作系统信号编号可能不同，但名称一致。
返回值：
    成功：0
    失败：-1
```

案例：

```
int main()
{
    pid_t pid = fork();
    if (pid == 0)
    {//子进程
        int i = 0;
        for (i = 0; i<5; i++)
        {
            printf("in son process\n");
            sleep(1);
        }
    }
    else
    {//父进程
        printf("in father process\n");
        sleep(2);
        printf("kill sub process now \n");
        kill(pid, SIGINT);
    }
    return 0;
}
```

2. raise函数

```
#include <signal.h>
int raise(int sig);
功能：给当前进程发送指定信号(自己给自己发)，等价于 kill(getpid(), sig)
参数：
    sig：信号编号
返回值：
    成功：0
    失败：非0值
```

3. abort函数

```
#include <stdlib.h>

void abort(void);
功能：给自己发送异常终止信号 6) SIGABRT，并产生core文件，等价于 kill(getpid(), SIGABRT);

参数：无

返回值：无
```

4. alarm函数（闹钟）

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
功能：
    设置定时器(闹钟)。在指定seconds后，内核会给当前进程发送14）SIGALRM信号。进程收到该信号，默认动作终止。每个进程都有且只有唯一的一个定时器。
    取消定时器alarm(0)，返回旧闹钟余下秒数。
参数：
    seconds：指定的时间，以秒为单位
返回值：
    返回0或剩余的秒数
```

5. setitimer函数(定时器)

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

int setitimer(int which,  const struct itimerval *new_value, struct itimerval *old_value);
功能：
    设置定时器(闹钟)。 可代替alarm函数。精度微秒us，可以实现周期定时。
参数：
    which：指定定时方式
        a) 自然定时：ITIMER_REAL → 14）SIGALRM计算自然时间
        b) 虚拟空间计时(用户空间)：ITIMER_VIRTUAL → 26）SIGVTALRM  只计算进程占用cpu的时间
        c) 运行时计时(用户 + 内核)：ITIMER_PROF → 27）SIGPROF计算占用cpu及执行系统调用的时间
    new_value：struct itimerval，负责设定timeout时间
        struct itimerval {
            struct timerval it_interval; // 闹钟触发周期
            struct timerval it_value;    // 闹钟触发时间
        };
```

```
        struct timeval {
            long tv_sec;            // 秒
            long tv_usec;           // 微秒
        }
        itimerval.it_value:  设定第一次执行function所延迟的秒数
        itimerval.it_interval:   设定以后每几秒执行function
    old_value:  存放旧的timeout值，一般指定为NULL
返回值:
    成功：0
    失败：-1
```

3.信号集

1. 概述

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

2. 自定义信号集

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

```
#include <signal.h>

int sigemptyset(sigset_t *set);        //将set集合置空
int sigfillset(sigset_t *set);            //将所有信号加入set集合
int sigaddset(sigset_t *set, int signo);  //将signo信号加入到set集合
int sigdelset(sigset_t *set, int signo);   //从set集合中移除signo信号
int sigismember(const sigset_t *set, int signo); //判断信号是否存在
//除sigismember外，其余操作函数中的set均为传出参数。sigset_t类型的本质是位图。但不应该直接使用位操作，而应该使用上述函数，保证跨系统操作有效。
```

3. `sigprocmask` 函数

> 信号阻塞集也称信号屏蔽集、信号掩码。每个进程都有一个阻塞集，创建子进程时子进程将继承父进程的阻塞集。信号阻塞集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）。

```
#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，有 3 种情况：
        SIG_BLOCK：向信号阻塞集合中添加 set 信号集，新的信号掩码是set和旧信号掩码的并集。相当于 mask = mask|set。
        SIG_UNBLOCK：从信号阻塞集合中删除 set 信号集，从当前信号掩码中去除 set 中的信号。相当于 mask = mask & ~ set。
        SIG_SETMASK：将信号阻塞集合设为 set 信号集，相当于原来信号阻塞集的内容清空，然后按照 set 中的信号重新设置信号阻塞集。相当于mask = set。
    set : 要操作的信号集地址。
        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。
    oldset : 保存原先信号阻塞集地址
返回值：
    成功：0，
    失败：-1，失败时错误代码只可能是 EINVAL，表示参数 how 不合法。
```

3. 未决信号集

```
#include <signal.h>
int sigpending(sigset_t *set);
功能：读取当前进程的未决信号集
参数：
    set：未决信号集
返回值：
    成功：0
    失败：-1
```

4.信号的捕捉

1. `signal`函数

```
#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。此函数不会阻塞。

参数：
    signum：信号的编号，这里可以填数字编号，也可以填信号的宏定义，可以通过命令 kill - l("l" 为字母)进行相应查看。
    handler : 取值有 3 种情况：
```

```
        SIG_IGN：忽略该信号
        SIG_DFL：执行系统默认动作
        信号处理函数名：自定义信号处理函数，如：func
        回调函数的定义如下：
          void func(int signo)
          {
              // signo 为触发的信号，为 signal() 第一个参数的值
          }
返回值：
    成功：第一次返回 NULL，下一次返回此信号上一次注册的信号处理函数的地址。如果需要使用此返回值，必须在前面先声明此函数指针的类型。
    失败：返回 SIG_ERR
```

2. `sigaction`函数 **一般都用这个**

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
功能：
    检查或修改指定信号的设置（或同时执行这两种操作）。
参数：
    signum：要操作的信号。
    act：   要设置的对信号的新处理方式（传入参数）。
    oldact：原来对信号的处理方式（传出参数）。
    如果 act 指针非空，则要改变指定信号的处理方式（设置），如果 oldact 指针非空，则系统将此前指定信号的处理方式存入 oldact。
返回值：
    成功：0
    失败：-1
/* sigaction的参数 */
struct sigaction {
    void(*sa_handler)(int); //旧的信号处理函数指针
    void(*sa_sigaction)(int, siginfo_t *, void *); //新的信号处理函数指针
    sigset_t   sa_mask;       //信号阻塞集,在信号处理函数执行过程中，临时屏蔽指定的信号。
    int        sa_flags;      //信号处理的方式
    /*sa_flags可以是以下值的“按位或”组合：
        Ø SA_RESTART：使被信号打断的系统调用自动重新发起（已经废弃）
        Ø SA_NOCLDSTOP：使父进程在它的子进程暂停或继续运行时不会收到 SIGCHLD 信号。
        Ø SA_NOCLDWAIT：使父进程在它的子进程退出时不会收到 SIGCHLD 信号，这时子进程如果退出也不会成为僵 尸进程。
        Ø SA_NODEFER：使对信号的屏蔽无效，即在信号处理函数执行期间仍能发出这个信号。
```

```
        Ø SA_RESETHAND：信号处理之后重新设置为默认的处理方式。
        Ø SA_SIGINFO：使用 sa_sigaction 成员而不是 sa_handler 作为信号处理函数。
    */
    void(*sa_restorer)(void); //已弃用
};
```

> sa_sigaction、sa_handler 两者之一赋值，其取值如下：
>
> a) SIG_IGN：忽略该信号
>
> b) SIG_DFL：执行系统默认动作
>
> c) 处理函数名：自定义信号处理函数

七、进程组、会话和守护进程

1.进程组

1. 概述

> 进程组，也称之为作业。BSD于1980年前后向Unix中增加的一个新特性。**代表一个或多个进程的集合。**
>
> 当父进程，创建子进程的时候，默认子进程与父进程属于同一进程组。进程组ID为第一个进程ID(组长进程)。所以，组长进程标识：其进程组ID为其进程ID

2. 相关函数

```
#include <unistd.h>

pid_t getpgrp(void);                 /* POSIX.1 version */
功能：获取当前进程的进程组ID
参数：无
返回值：总是返回调用者的进程组ID

pid_t getpgid(pid_t pid);
功能：获取指定进程的进程组ID
参数：
    pid：进程号，如果pid = 0，那么该函数作用和getpgrp一样
返回值：
    成功：进程组ID
    失败：-1

int setpgid(pid_t pid, pid_t pgid);
功能：
    改变进程默认所属的进程组。通常可用来加入一个现有的进程组或创建一个新进程组。
参数：
```

```
    将参1对应的进程，加入参2对应的进程组中
返回值：
    成功：0
    失败：-1
```

2.会话

> 会话是一个或多个**进程组**的集合。

1. 相关函数

```
#include <unistd.h>
pid_t getsid(pid_t pid);
功能：获取进程所属的会话ID
参数：
    pid：进程号，pid为0表示查看当前进程session ID
返回值：
    成功：返回调用进程的会话ID
    失败：-1
pid_t setsid(void);
功能：
    创建一个会话，并以自己的ID设置进程组ID，同时也是新会话的ID。调用了setsid函数的进程，既是新的会长，也是新的组长。
参数：无
返回值：
    成功：返回调用进程的会话ID
    失败：-1
```

3.守护进程

1. 创建子进程，父进程退出(必须)

- 所有工作在子进程中进行形式上脱离了控制终端

2. 在子进程中创建新会话(必须)

- setsid()函数
- 使子进程完全独立出来，脱离控制

3. 改变当前目录为根目录(不是必须)

- chdir()函数
- 防止占用可卸载的文件系统
- 也可以换成其它路径

4. 重设文件权限掩码(不是必须)

- umask()函数
- 防止继承的文件创建屏蔽字拒绝某些权限
- 增加守护进程灵活性

5. 关闭文件描述符(不是必须)

- 继承的打开文件不会用到，浪费系统资源，无法卸载

6. 开始执行守护进程核心工作(必须)

案例

```
/*
* time_t rawtime;
* time ( &rawtime  ); --- 获取时间，以秒计，从1970年1月一日起算，存于rawtime
* localtime ( &rawtime  ); //转为当地时间，tm 时间结构
* asctime() // 转为标准ASCII时间格式：
*/
void write_time(int num)
{
    time_t rawtime;
    struct tm * timeinfo;
    // 获取时间
    time(&rawtime);
#if 0
    // 转为本地时间
    timeinfo = localtime(&rawtime);
    // 转为标准ASCII时间格式
    char *cur = asctime(timeinfo);
#else
    char* cur = ctime(&rawtime);
#endif
    // 将得到的时间写入文件中
    int fd = open("/home/edu/timelog.txt", O_RDWR | O_CREAT | O_APPEND, 0664);
    if (fd == -1)
    {
        perror("open error");
        exit(1);
    }
    // 写文件
    int ret = write(fd, cur, strlen(cur) + 1);
    if (ret == -1)
    {
        perror("write error");
        exit(1);
```

```
    }
    // 关闭文件
    close(fd);
}
int main(int argc, const char* argv[])
{
    pid_t pid = fork();
    if (pid == -1)
    {
        perror("fork error");
        exit(1);
    }
    if (pid > 0)
    {
        // 父进程退出
        exit(1);
    }
    else if (pid == 0)
    {
        // 子进程
        // 提升为会长，同时也是新进程组的组长
        setsid();
        // 更改进程的执行目录
        chdir("/home/edu");
        // 更改掩码
        umask(0022);
        // 关闭文件描述符
        close(STDIN_FILENO);
        close(STDOUT_FILENO);
        close(STDERR_FILENO);
        // 注册信号捕捉函数
        //先注册，再定时
        struct sigaction sigact;
        sigact.sa_flags = 0;
        sigemptyset(&sigact.sa_mask);
        sigact.sa_handler = write_time;
        sigaction(SIGALRM, &sigact, NULL);
        // 设置定时器
        struct itimerval act;
        // 定时周期
        act.it_interval.tv_sec = 2;
        act.it_interval.tv_usec = 0;
        // 设置第一次触发定时器时间
        act.it_value.tv_sec = 2;
        act.it_value.tv_usec = 0;
```

```
79          // 开始计时
80          setitimer(ITIMER_REAL, &act, NULL);
81          // 防止子进程退出
82          while (1);
83      }
84      return 0;
85  }
```

4.线程

1.概述

> 在许多经典的操作系统教科书中，总是把进程定义为程序的执行实例，它并不执行什么，只是维护应用程序所需的各种资源，而线程则是真正的执行实体。
>
> 所以，线程是轻量级的进程（LWP：light weight process），在Linux环境下线程的本质仍是进程。
>
> 为了让进程完成一定的工作，进程必须至少包含一个线程。
>
> 进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动，进程是系统进行资源分配和调度的一个独立单位。
>
> 线程是进程的一个实体，是 CPU 调度和分派的基本单位，它是比进程更小的能独立运行的基本单位。线程自己基本上不拥有系统资源，只拥有一点在运行中必不可少的资源（如程序计数器，一组寄存器和栈），但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源。

2.特点

> 线程是轻量级进程(light-weight process)，也有PCB，创建线程使用的底层函数和进程一样，都是clone
>
> 从内核里看进程和线程是一样的，都有各自不同的PCB.
>
> 进程可以蜕变成线程
>
> 在linux下，**线程最是小的执行单位；进程是最小的分配资源单位**

3.线程常用操作

1. `pthread_self` 获取线程号

```
#include <pthread.h>
pthread_t pthread_self(void);
功能：
    获取线程号。
参数：
    无
返回值：
    调用线程的线程 ID
```

2. `pthread_equal` 判断线程号 t1 和 t2 是否相等

```
int pthread_equal(pthread_t t1, pthread_t t2);
功能：
    判断线程号 t1 和 t2 是否相等。为了方便移植，尽量使用函数来比较线程 ID。
参数：
    t1，t2：待判断的线程号。
返回值：
    相等：  非 0
    不相等：0
```

3. `pthread_create` 创建一个线程

```
#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id。
    attr：线程属性结构体地址，通常设置为 NULL。
    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：0
    失败：非 0
```

> 在一个线程中调用pthread_create()创建新的线程后，当前线程从pthread_create()返回继续往下执行，而新的线程所执行的代码由我们传给pthread_create的函数指针start_routine决定。
>
> 由于pthread_create的错误码不保存在errno中，因此不能直接用perror()打印错误信息，可以先用strerror()把错误码转换成错误信息再打印。

4. `pthread_join` 回收线程资源

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **retval);
功能：
    等待线程结束（此函数会阻塞），并回收线程资源，类似进程的 wait() 函数。如果线程已经结束，那么该函数会立即返回。
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：0
    失败：非 0
```

> 调用该函数的线程将挂起等待，直到id为thread的线程终止。thread线程以不同的方法终止，通过pthread_join得到的终止状态是不同的，总结如下：
>
> 1. 如果thread线程通过return返回，retval所指向的单元里存放的是thread线程函数的返回值。
> 2. 如果thread线程被别的线程调用pthread_cancel异常终止掉，retval所指向的单元里存放的是常数PTHREAD_CANCELED。
> 3. 如果thread线程是自己调用pthread_exit终止的，retval所指向的单元存放的是传给pthread_exit的参数。

5. 线程分离

> 一般情况下，线程终止后，其终止状态一直保留到其它线程调用pthread_join获取它的状态为止。但是线程也可以被置为detach状态，这样的线程一旦终止就立刻回收它占用的所有资源，而不保留终止状态。
>
> 不能对一个已经处于detach状态的线程调用pthread_join，这样的调用将返回EINVAL错误。也就是说，如果已经对一个线程调用了pthread_detach就不能再调用pthread_join了。

`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，分离后不代表此线程不依赖与当前进程，线程分离的目的是将线程资源的回收工作交由系统自动来完成，也就是说当被分离的线程结束之后，系统会自动回收它的资源。所以，此函数不会阻塞。
参数：
    thread：线程号。
返回值：
    成功：0
    失败：非0
```

6. 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。
>
> - 线程从执行函数中返回。
> - 线程调用pthread_exit退出线程。
> - 线程可以被同一进程中的其它线程取消。

`pthread_exit`

```
#include <pthread.h>
void pthread_exit(void *retval);
功能：
    退出调用线程。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。
参数：
    retval：存储线程退出状态的指针。
返回值：无
```

7. 线程取消

```
#include <pthread.h>
int pthread_cancel(pthread_t thread);
功能：
    杀死(取消)线程
参数：
    thread ： 目标线程ID。
返回值：
    成功：0
    失败：出错编号
```

4.互斥锁

1. 同步和互斥

> **互斥**：是指散步在不同任务之间的若干程序片断，当某个任务运行其中一个程序片段时，其它任务就不能运行它们之中的任一程序片段，只能等到该任务运行完这个程序片段后才可以运行。最基本的场景就是：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。
>
> **同步**：是指散步在不同任务之间的若干程序片断，它们的运行必须严格按照规定的某种先后次序来运行，这种先后次序依赖于要完成的特定的任务。最基本的场景就是：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。比如 A 任务的运行依赖于 B 任务产生的数据。

目的

> 在多任务操作系统中，同时运行的多个任务可能都需要使用同一种资源。这个过程有点类似于，公司部门里，我在使用着打印机打印东西的同时（还没有打印完），别人刚好也在此刻使用打印机打印东西，如果不做任何处理的话，打印出来的东西肯定是错乱的。

2. 函数

    1. `pthread_mutex_init` 初始化互斥锁

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
    const pthread_mutexattr_t *restrict attr);
功能：
    初始化一个互斥锁。
参数：
    mutex：互斥锁地址。类型是 pthread_mutex_t 。
    attr：设置互斥量的属性，通常可采用默认属性，即可将 attr 设为 NULL。

    可以使用宏 PTHREAD_MUTEX_INITIALIZER 静态初始化互斥锁，比如：
    pthread_mutex_t  mutex = PTHREAD_MUTEX_INITIALIZER;
这种方法等价于使用 NULL 指定的 attr 参数调用 pthread_mutex_init() 来完成动态初始化，不同之处在于 PTHREAD_MUTEX_INITIALIZER 宏不进行错误检查。
返回值：
    成功：0 ，成功申请的锁默认是打开的。
    失败：非 0 错误码
```

> restrict，C语言中的一种类型限定符（Type Qualifiers），用于告诉编译器，对象已经被指针所引用，不能通过除该指针外所有其他直接或间接的方式修改该对象的内容。

    2. `pthread_mutex_destroy` 销毁指定的一个互斥锁

```
#include <pthread.h>
int pthread_mutex_destroy(pthread_mutex_t *mutex);
功能：
    销毁指定的一个互斥锁。互斥锁在使用完毕后，必须要对互斥锁进行销毁，以释放资源。
参数：
    mutex：互斥锁地址。
返回值：
    成功：0
    失败：非 0 错误码
```

    3. `pthread_mutex_lock` 互斥锁上锁

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
功能：
    对互斥锁上锁，若互斥锁已经上锁，则调用者阻塞，直到互斥锁解锁后再上锁。
参数：
    mutex：互斥锁地址。
返回值：
    成功：0
    失败：非 0 错误码
int pthread_mutex_trylock(pthread_mutex_t *mutex);
   调用该函数时，若互斥锁未加锁，则上锁，返回 0；
   若互斥锁已加锁，则函数直接返回失败，即 EBUSY。
```

4. `pthread_mutex_unlock` 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：0
    失败：非0错误码
```

案例

```
pthread_mutex_t mutex; //互斥锁
// 打印机
void printer(char *str)
{
    pthread_mutex_lock(&mutex); //上锁
    while (*str != '\0')
    {
        putchar(*str);
        fflush(stdout);
        str++;
        sleep(1);
    }
    printf("\n");
    pthread_mutex_unlock(&mutex); //解锁
}
// 线程一
void *thread_fun_1(void *arg)
{
```

```c
    char *str = "hello";
    printer(str); //打印
}
// 线程二
void *thread_fun_2(void *arg)
{
    char *str = "world";
    printer(str); //打印
}
int main(void)
{
    pthread_t tid1, tid2;
    pthread_mutex_init(&mutex, NULL); //初始化互斥锁
    // 创建 2 个线程
    pthread_create(&tid1, NULL, thread_fun_1, NULL);
    pthread_create(&tid2, NULL, thread_fun_2, NULL);
    // 等待线程结束，回收其资源
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    pthread_mutex_destroy(&mutex); //销毁互斥锁
    return 0;
}
```

5.读写锁

1. 概述

> 当有一个线程已经持有互斥锁时，互斥锁将所有试图进入临界区的线程都阻塞住。但是考虑一种情形，当前持有互斥锁的线程只是要读访问共享资源，而同时有其它几个线程也想读取这个共享资源，但是由于互斥锁的排它性，所有其它线程都无法获取锁，也就无法读访问共享资源了，但是实际上多个线程同时读访问共享资源并不会导致问题。
>
> 在对数据的读写操作中，更多的是读操作，写操作较少，例如对数据库数据的读写应用。为了满足当前能够允许多个读出，但只允许一个写入的需求，线程提供了**读写锁**来实现。

2. 相关函数

    1. `pthread_rwlock_init` 初始化 rwlock 所指向的读写锁

```c
#include <pthread.h>

int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
    const pthread_rwlockattr_t *restrict attr);
功能:
```

```
    用来初始化 rwlock 所指向的读写锁。

参数：
    rwlock：指向要初始化的读写锁指针。
    attr：读写锁的属性指针。如果 attr 为 NULL 则会使用默认的属性初始化读写锁，否则使用指定的 attr 初始化读写锁。

    可以使用宏 PTHREAD_RWLOCK_INITIALIZER 静态初始化读写锁，比如：
    pthread_rwlock_t my_rwlock = PTHREAD_RWLOCK_INITIALIZER;

    这种方法等价于使用 NULL 指定的 attr 参数调用 pthread_rwlock_init() 来完成动态初始化，不同之处在于 PTHREAD_RWLOCK_INITIALIZER 宏不进行错误检查。

返回值：
    成功：0，读写锁的状态将成为已初始化和已解锁。
    失败：非 0 错误码。
```

2. `pthread_rwlock_destroy`销毁一个读写锁

```
#include <pthread.h>
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
功能：
    用于销毁一个读写锁，并释放所有相关联的资源（所谓的所有指的是由 pthread_rwlock_init() 自动申请的资源） 。
参数：
    rwlock：读写锁指针。
返回值：
    成功：0
    失败：非 0 错误码
```

3. `pthread_rwlock_rdlock`读锁定

```
#include <pthread.h>

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
功能：
    以阻塞方式在读写锁上获取读锁（读锁定）。
    如果没有写者持有该锁，并且没有写者阻塞在该锁上，则调用线程会获取读锁。
    如果调用线程未获取读锁，则它将阻塞直到它获取了该锁。一个线程可以在一个读写锁上多次执行读锁定。
    线程可以成功调用 pthread_rwlock_rdlock() 函数 n 次，但是之后该线程必须调用 pthread_rwlock_unlock() 函数 n 次才能解除锁定。
参数：
```

```
    rwlock：读写锁指针。
返回值：
    成功：0
    失败：非 0 错误码

int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
    用于尝试以非阻塞的方式来在读写锁上获取读锁。
    如果有任何的写者持有该锁或有写者阻塞在该读写锁上，则立即失败返回。
```

4. `pthread_rwlock_wrlock` 写锁定

```
#include <pthread.h>
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
功能：
    在读写锁上获取写锁（写锁定）。
    如果没有写者持有该锁，并且没有写者读者持有该锁，则调用线程会获取写锁。
    如果调用线程未获取写锁，则它将阻塞直到它获取了该锁。
参数：
    rwlock：读写锁指针。
返回值：
    成功：0
    失败：非 0 错误码
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
    用于尝试以非阻塞的方式来在读写锁上获取写锁。
    如果有任何的读者或写者持有该锁，则立即失败返回。
```

5. `pthread_rwlock_unlock` 解锁

```
#include <pthread.h>

int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
功能：
    无论是读锁或写锁，都可以通过此函数解锁。
参数：
    rwlock：读写锁指针。
返回值：
    成功：0
    失败：非 0 错误码
```

6. **案例**

```
pthread_rwlock_t rwlock; //读写锁
int num = 1;
//读操作，其他线程允许读操作，却不允许写操作
```

```c
void *fun1(void *arg)
{
    while (1)
    {
        pthread_rwlock_rdlock(&rwlock);
        printf("read num first===%d\n", num);
        pthread_rwlock_unlock(&rwlock);
        sleep(1);
    }
}
//读操作，其他线程允许读操作，却不允许写操作
void *fun2(void *arg)
{
    while (1)
    {
        pthread_rwlock_rdlock(&rwlock);
        printf("read num second===%d\n", num);
        pthread_rwlock_unlock(&rwlock);
        sleep(2);
    }
}
//写操作，其它线程都不允许读或写操作
void *fun3(void *arg)
{
    while (1)
    {
        pthread_rwlock_wrlock(&rwlock);
        num++;
        printf("write thread first\n");
        pthread_rwlock_unlock(&rwlock);
        sleep(2);
    }
}
// 写操作，其它线程都不允许读或写操作
void *fun4(void *arg)
{
    while (1)
    {
        pthread_rwlock_wrlock(&rwlock);
        num++;
        printf("write thread second\n");
        pthread_rwlock_unlock(&rwlock);
        sleep(1);
    }
}
```

```c
int main()
{
    pthread_t ptd1, ptd2, ptd3, ptd4;
    pthread_rwlock_init(&rwlock, NULL);//初始化一个读写锁
    //创建线程
    pthread_create(&ptd1, NULL, fun1, NULL);
    pthread_create(&ptd2, NULL, fun2, NULL);
    pthread_create(&ptd3, NULL, fun3, NULL);
    pthread_create(&ptd4, NULL, fun4, NULL);
    //等待线程结束，回收其资源
    pthread_join(ptd1, NULL);
    pthread_join(ptd2, NULL);
    pthread_join(ptd3, NULL);
    pthread_join(ptd4, NULL);

    pthread_rwlock_destroy(&rwlock);//销毁读写锁
    return 0;
}
```