目录

UNIX 系统高级编程
主讲人：闵卫

一、GNU 编译器

GNU = GNU Not Unix
1.GCC 的基本特点
1)支持多种硬件架构
x86-64
Alpha
ARM
Motorola 68000
MIPS
PDP-10/11
PowerPC
System/370-390
SPARC
VAX
2)支持多种操作系统
Unix
Linux
BSD
Android
Mac OS X
iOS
Windows
3)支持多种编程语言
C/C++
Objective-C
Java
Fortran
Pascal
Ada
4)查看版本信息
早期：GCC = GNU C Compiler

现代：GCC = GNU Compiler Collection
版本：gcc -v
2.构建过程
从源代码到可执行程序的构建过程：
hello.c -gcc hello.c-> a.out
| |
源代码 可执行程序

预编译(编译预处理)：头文件扩展、宏替换
gcc -E hello.c
gcc -E hello.c -o hello.i

汇编：将高级语言代码翻译成汇编语言代码，得到汇编文件
gcc -S hello.i -> hello.s

编译：将汇编语言代码翻译成机器语言代码，得到目标模块
gcc -c hello.s -> hello.o

链接：将所有的目标模块及其所依赖的库连接成一个整体，得到可执行程序
gcc hello.o <其它目标模块...> -l<其它库...> -o hello

3.文件名后缀
.h - C 语言源代码头文件
.c - 预处理前的 C 语言源代码文件
.i - 预处理后的 C 语言源代码文件
.s - 汇编语言文件
.o - 目标模块文件
.a - 静态库文件
.so - 动态库(共享库)文件
习惯上可执行程序文件不带后缀。
4.编译选项
gcc [选项] [参数] 文件
gcc -E -o hello.i hello.c
| | | |
选项 选项 -o 选项的参数 被处理文件
-o: 指定输出文件的路径
-E: 预编译，缺省输出到屏幕，用-o 选项指定输出到文件
-S: 汇编，将高级语言源文件汇编成汇编语言源文件
-c: 编译，将汇编语言源文件编译成机器语言目标模块
-Wall: 报告所有警告
代码：wall.c
-Werror：将警告视作错误
代码：werror.c
-x: 显式指明源代码所用编程语言 gcc -xc++ -lstdc++ -o cpp.c
-g: 产生调试信息
-O0/O1/O2/O3: 优化策略
O0 - 表示不做优化

O1 - 缺省优化，在空间和时间上选择尽可能折中的优化处理
O2 - 牺牲空间换取时间
O3 - 牺牲时间换取空间

5.头文件

1)头文件里放什么？(两个头文件，三个声明，两个散装)

A.头文件卫士

```
      a.h
    /    \
b.h        c.h  - 钻石包含
    \    /
      d.c
```

在编译 d.c 的过程中会对 a.h 中定义的宏或数据类型等报重定义错。

在 a.h 中加入头文件卫士：

```
#ifndef  __A_H__
#define  __A_H__
只被编译器处理一次的代码...
#endif  //  __A_H__
```

或：

```
#pragma  once
```

B.包含其它头文件

a.c

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

b.c

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

c.c

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

---

common.h

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

a.c

```
#include  "common.h"
```

b.c

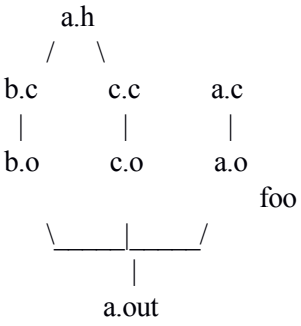
```
#include  "common.h"
```

c.c

```
#include  "common.h"
```

C.宏定义
geom.h
```
    #define PAI 3.14159
    ...
```
D.自定义类型
geom.h
```
    struct Circle {
        double x;
        double y;
        double r;
    };
    ...
```
E.类型别名
types.h
```
    typedef unsigned long long ULL;
    typedef int* (*PFOO)(char*, short const*);
    ...
```
F.声明外部变量
global.c
```
    double e = 2.71828;
    ...
```
global.h
```
    extern double e /* = 2.71828 */;
    ...
```
G.声明函数
geom.h
```
    double circle_area(dobule r);
    ...
```
一个头文件(.h)可能会被多个源文件(.c)包含，写在头文件里的函数定义也会因此被预处理器扩展到多个包含该头文件的源文件中，并在编译阶段被编译到多个不同的目标模块中，这将导致链接错误：multiple definition，多重定义。因此一般情况下，要避免将函数定义写到头文件中。
```
a.h
    int foo(int a, int b) { return a + b; }
     a.h
    /   \
b.c      c.c
 |        |
b.o      c.o
  foo      foo - multiple definition
    \   /
    a.out
```

```
a.c
    int foo(int a, int b) { return a + b; }
a.h
    int foo(int a, int b);
```



2)去哪里找头文件?

头文件位于 / 系统目录：/usr/include 等目录
- 当前目录：执行 gcc 命令的目录
\ 附加目录：通过 gcc 的-I 选项指定的目录

尖括号包含，如：#include <calc.h>
附加目录->系统目录
双引号包含，如：#include "calc.h"
附加目录->当前目录->系统目录
头文件的系统目录：
/usr/include - 标准 C 库
/usr/local/include - 第三方库
/usr/lib/gcc/i686-linux-gnu/<版本号>/include - 编译器相关库
/usr/inlude/c++/<版本号> - 标准 C++库
代码：calc.h、calc.c、math.c

声明 / 定义(实现) | 调用(使用) \

6.预处理指令
#include ：将指定文件的内容插至此指令处
#define ：定义宏
#undef ：删除宏
#if ：如果
#ifdef ：如果宏已定义
#ifndef ：如果宏未定义
#else ：否则，与#if/#ifdef/#ifndef 配合使用
#elif ：否则如果，与#if/#ifdef/#ifndef 配合使用
#endif ：结束判定，与#if/#ifdef/#ifndef 配合使用
#error ：产生错误，提前终止预处理过程
#warning ：产生警告，不会终止预处理过程
代码：error.c
#line ：显式指定下一行的行号
代码：line.c
#pragma ：设置编译器状态或指示编译器操作
#pragma once - 一次性编译 与头文件卫士功能一样
#pragma GCC dependency <被依赖文件> - 指定文件依赖
如果被依赖文件比包含该预处理指令的文件新，则产生警告。
#pragma GCC poison <语言禁忌>
如果在代码中出现语言禁忌，则产生错误。
#pragma pack(1/2/4/8...) - 字节对齐 64 位 8 字节对齐 32 位 4 字节对齐

代码：error.c

```
1 #include<stdio.h>
2 //#define VERSION 3
3 #if(VERSION < 3)
4       #error "版本太低"
5 #endif
6 int main()
7 {
8       printf("%d\n",VERSION);
9
10       return 0;
11 }
```

```
gcc error.c -o error -DVERSION=2

tarena@ubuntu:~/Desktop/uc/day01$ gcc error.c -o error -DVERSION=2
error.c:4:6: error: #error "版本太低"
#error "版本太低"
^
```

warning.c

```
1 #include<stdio.h>
2 //#define VERSION 3
3 #if(VERSION < 3)
4       #error "版本太低"
5 #elif (VERSION > 3)
6       #warning "版本太高"
7 #endif
8 int main()
9 {
10       printf("%d\n",VERSION);
11
12       return 0;
13 }
```

#line 数字 ： 显式指定下一行的行号

代码：line.c

```
  1 #include <stdio.h>
  2 int main(void) {
  3             int a = 123, b = 456;
  4             printf("%d+%d=%d\n", a, b, a + b);
  5             FILE* fp = fopen("file1", "r");
  6             if (!fp) {
  7 #line 100
  8                       fprintf(stderr, "%d> 无法打开文件！\n", __LINE__);
  9                       return -1;
```

```
10              }
11              // ...
12              fclose(fp);
13              if (!(fp = fopen("file2", "r"))) {
14 #line 200
15                      fprintf(stderr, "%d> 无法打开文件！\n", __LINE__);
16                      return -1;
17              }
18              // ...
19              fclose(fp);
20              return 0;
21 }
```

代码：once1.h、once2.h、once3.h、once4.c、pragma.c

Pragma.c

```
 1 #include <stdio.h>
 2 #pragma GCC dependency "dep"
 3 #pragma GCC poison goto float
 4 int main(void) {
 5              int i = 0;
 6              /*
 7 again:
 8              printf("%d ", i++);
 9              if (i < 10)
10                      goto again;
11              */
12              while (i < 10)
13                      printf("%d ", i++);
14              printf("\n");
15              //float f;
16              double f;
17              struct A {
18                      double a; // 8
19                      char   b; // 1
20                      int    c; // 4
21                      short  d; // 2
22              };              // 64: aaaaaaaabxxxxxxxccccddxx - 24
23                              // 32: aaaaaaaabxxxccccddxx - 20
24              printf("%lu\n", sizeof(struct A)); // 24
25 #pragma pack(1)
26              struct B {
27                      double a; // 8
28                      char   b; // 1
29                      int    c; // 4
30                      short  d; // 2
31              };              // 64/32: aaaaaaaabccccdd - 15
32              printf("%lu\n", sizeof(struct B)); // 15
```

```
#pragma pack(2)
        struct C {
                double a; // 8
                char   b; // 1
                int    c; // 4
                short  d; // 2
        };              // 64/32: aaaaaaaabxccccdd - 16
        printf("%lu\n", sizeof(struct C)); // 16
#pragma pack()
        struct D {
                double a; // 8
                char   b; // 1
                int    c; // 4
                short  d; // 2
        };              // 64: aaaaaaaabxxxxxxxccccddxx - 24
                        // 32: aaaaaaaabxxxccccddxx - 20
        printf("%lu\n", sizeof(struct D)); // 24
        return 0;
}
```

7.预定义宏(编译器内置的宏)

__BASE_FILE__ ：正在编译的文件名

__FILE__ ：所在文件的文件名

a.h

```
    __BASE_FILE__ -> b.c
    __FILE__ -> a.h
```

b.c

```
    #include "a.h"
```

```
gcc -c b.c
```

__LINE__ ：行号(受#line 影响)

__FUNCTION__ ：所在函数的函数名

__func__ ：等价于__FUNCTION__

__DATE__ ：编译日期

__TIME__ ：编译时间

__INCLUDE_LEVEL__ ：包含层数，从 0 开始

a.h

```
    __INCLUDE_LEVEL__ -> 2
```

b.h

```
    __INCLUDE_LEVEL__ -> 1
    #include "a.h"
```

c.c

```
    __INCLUDE_LEVEL__ -> 0
    #include "b.h"
```

```
gcc -c c.c
```

__cplusplus ：C++有定义，C 无定义

代码：print.h、predef.h、predef.c

8.环境变量

C_INCLUDE_PATH(C_PATH)：指定 C 语言头文件的附加目录，相当于 gcc 命令的-I 选项
CPLUS_INCLUDE_PATH：指定 C++语言头文件的附加目录，相当于 g++命令的-I 选项
env 查看当前环境变量
env | grep C_INCLUDE_PATH 查看当前目录下的 C_INCLUDE_PATH 环境变量名的内容
LIBRARY_PATH：指定链接时查找库的目录
LD_LIBRARY_PATH：运行时查找动态库的目录
export 环境变量名=环境变量值 ____
export 环境变量名=$环境变量名:追加内容 / |
|
用户主目录/.bashrc
定位头文件的三种方式：
①#include "/home/tarena/include/myheader.h"
一旦头文件路径发生变化，必须修改源程序。
②export C_INCLUDE_PATH=$C_INCLUDE_PATH:/home/tarena/include
export C_PATH=$C_PATH:/home/tarena/include
同时维护多个项目，容易发生版本冲突。
③gcc ... -I/home/tarena/include
gcc ... -I/home/tarena/project/include
既不需要修改源程序，也不会发生版本冲突，推荐使用此方法。

二、静态库和动态库

1.库

```
big.c       -> big.o          -> exe
--------------------------------
text.c      -> text.o      \
image.c     -> image.o     |
video.c     -> video.o     | -> exe
network.c   -> network.o   |
main.c      -> main.o      /
--------------------------------
text.c      -> text.o      \ 实现功能            使用功能
image.c     -> image.o     |      |                  |
video.c     -> video.o     | -> 库 + main.o <- main.c
audio.c     -> audio.o     |      \_____/
network.c   -> network.o   |         |
encrypt.c   -> encrypt.o   /         exe
```

2.静态库

1)静态库的本质就是将多个目标文件(.o)打包成为一个文件。链接静态库就是将库中被调用的代码复制到调用模块中。静态库占用空间比较大，库中的代码一旦修改必须重新链接。使用静态库的代码在运行时无需依赖库，且执行效率高。

2)静态库的形式：libxxx.a

3)构建静态库

```
ar -r libxxx.a yyy.o zzz.o ...
       ^      \_____________/
       |____________|
```

4)链接静态库

```
gcc -c main.o -lxxx -L<库路径> ...
gcc -c main.o -lxxx ... <- LIBRARY_PATH(给链接器用的)
```

5)调用静态库：运行时不需要调用静态库中的代码。
代码：static/
3.动态库(共享库)
1)动态库和静态库最大的不同就是，链接动态库并不需要将库中的被调用代码复制到调用模块中，相反被嵌入到调用模块中的仅仅是被调用代码在动态库中的相对地址。在调用模块实际运行时，再根据动态库的加载地址和被调用代码的相对地址，计算出该代码的绝对地址，读取代码的内容并运行之。如果动态库中的代码同时被多个进程所用，动态库的实例在内存中仅需一份，因此动态库也叫共享库。使用动态库占用的内存空间小，即使修改了动态库中的代码，只要其相对地址不变，无需重新链接。因为在执行过程中，需要计算被调用代码的绝对地址，以及一些附带的额外开销，所以调用动态库会比调用静态库略慢一些。
2)动态库的形式：libxxx.so
ldd 可执行文件:查看依赖
3)构建动态库
编译选项：-fpic，位置无关代码
例如：gcc -c -fpic add.c -o add.o
链接选项：-shared，没有 main 函数的可执行程序
例如：gcc -shared add.o sub.o -o libmath.so
gcc -fpic -shared add.c sub.c -o libmath.so -L.
4)链接动态库
gcc -c main.o -lxxx -L<库路径> ...
gcc -c main.o -lxxx ... <- LIBRARYT_PATH(给链接器用)
5)调用动态库:运行时需要调用动态库中的代码,因此动态库必须位于LD_LIBRARYT_PATH(加载器用)环境变量所包含的某个路径下。
代码：shared/
因为相较于静态库而言，动态库具有明显的优势，因此 gcc 的链接器缺省使用动态库版本。如果一定要使用静态库版本参与链接，需要加上-static 链接选项。
代码：hello.c
4.动态库的动态加载
Linux 操作系统提供的一个用于动态加载动态库的动态库：dl
#include <dlfcn.h>
-ldl:链接时用
例如：gcc load.c -ldl -o load
1)加载动态库
void* dlopen(const char* filename, int flag);
成功返回动态库句柄(handle)，失败返回 NULL。
filename: 动态库路径，也可以只给文件名，这种情况下函数将根据 LD_LIBRARY_PATH 环境变量中的路径搜索该动态库。
flag：加载方式，可取以下值：
RTLD_LAZY - 延迟加载
RTLD_NOW - 立即加载
该函数所成功返回的动态库句柄唯一地标识系统内核为维护的动态库实例，将作为后续函数调用的参数。
2)获取符号(函数名或变量名)地址
void* dlsym(void* handle, const char* symbol);
成功返回库中指定函数或(全局)变量的地址，失败返回 NULL。
handle: 动态库句柄
symbol: 符号名，即函数名或变量名

该函数返回的指针类型为 void*，需要显示转换为实际的指针类型才能访问其具体目标。可见所获取符号的实际类型必须事先知道。

3)卸载动态库

int dlclose(void* handle);

成功返回 0，失败返回非零。

handle: 动态库句柄

所卸载的动态库未必真的会从内存中立即消失，因为其它程序可能还需要使用该库。

dlopen(...) -> handle1
dlopen(...) -> handle2
dlopen(...) -> handle3
dlclose(handle2)
dlclose(handle3)
dlclose(handle1)

动态库实例

引用计数:0 -> 释放内存

该函数一方面会解除参数句柄和动态库实例之间的关联，使该句柄失效，另一方面会将动态库实例中的引用计数减 1，当其被减到 0 时才会真的释放所占内存资源。

4)获取错误信息

char* dlerror(void);

之前有错误发生则返回指向错误信息字符串的指针，否则返回 NULL。

代码：load.c

```
 1 #include <stdio.h>
 2 #include <dlfcn.h>
 3 int main(void) {
 4         // 加载./shared/libmath.so 动态库
 5         void* handle = dlopen("./shared/libmath.so", RTLD_NOW);
 6         if (!handle) {
 7                 fprintf(stderr, "dlopen: %s\n", dlerror());
 8                 return -1;
 9         }
10         // 获取 add 函数的地址
11         int (*add)(int, int) = (int (*)(int, int))dlsym(handle, "add");
12         if (!add) {
13                 fprintf(stderr, "dlsym: %s\n", dlerror());
14                 return -1;
15         }
16         // 获取 sub 函数的地址
17         int (*sub)(int, int) = (int (*)(int, int))dlsym(handle, "sub");
18         if (!sub) {
19                 fprintf(stderr, "dlsym: %s\n", dlerror());
20                 return -1;
21         }
22         // 获取 show 函数的地址
23         void (*show)(int, char, int, int) = (void (*)(int, char, int, int))d      lsym(
24                 handle, "show");
25         if (!show) {
26                 fprintf(stderr, "dlsym: %s\n", dlerror());
```

```
27                      return -1;
28              }
29              // 调用动态库中的函数
30              int a = 123, b = 456;
31              show(a, '+', b, add(a, b));
32              show(a, '-', b, sub(a, b));
33              // 卸载动态库
34              if (dlclose(handle)) {
35                      fprintf(stderr, "dlclose: %s\n", dlerror());
36                      return -1;
37              }
38              return 0;
39 }
```

5.辅助工具

1)查看符号表

nm 目标模块/可执行程序/静态库/动态库

列出目标模块、可执行程序、静态库或者动态库中的符号(函数或全局变量)。

T: 正文段，函数的代码块，地址

U: 正文段，对函数的调用，其代码块位于某个动态库中，无地址

D: 数据段，全局变量

...

2)反汇编

objdump -S 目标模块/可执行程序/静态库/动态库

将目标模块、可执行程序、静态库或者动态库中的二进制形式的机器代码转换为字符形式的汇编代码，打印在屏幕上。

3)消除冗余信息

strip 目标模块/可执行程序/静态库/动态库

去除目标模块、可执行程序、静态库或者动态库中的符号表、调试信息等非运行所必须的冗余信息，以缩减文件的大小。

4)查看动态库依赖

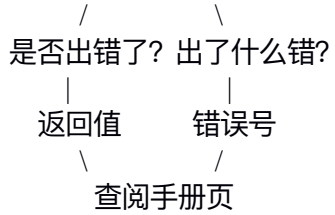
ldd 可执行程序/动态库

显示可执行程序或动态库所依赖的动态库信息。

三、错误处理

```
FILE* fp = fopen(...);
fread(..., fp);
处理数据
fwrite(..., fp);
fclose(fp);
```

当错误发生时：逃跑、提示再逃跑、自动纠错。



1.判断一个函数是否出错

1)返回指针的函数，如：malloc、fopen、dlopen 等，通常用返回 NULL 表示失败。
2)返回值属于特定值域的函数，通常用返回合法值域以外的值表示失败。
3)用返回 0 表示成功，返回-1 或其它非零值表示失败，如果有数据输出，则通过指针型参数向调用者输出数据。
2.标准库预定义了全局变量 errno，表示最近一次函数调用的错误编号。

```
#include <errno.h> // 声明 errno、定义错误号宏
FILE* fp = fopen(...);
if (!fp) {
    if (errno == EINVAL)
        处理无效读写模式错误
    else if (errno == ...)
        处理...错误
    ...
}
```

3.将整数形式的错误号转换为字符串

```
#include <string.h>
char* strerror(int errnum);
```

返回与参数错误号相对应的错误描述字符串指针。

```
#include <stdio.h>
void perror(const char* s);
```

将最近一次函数调用的错误描述字符串打印到标准错误设备上。

```
perror("abcde"); // abcde: 错误描述字符串
```

%m 格式化标志->错误描述字符串
locate xxx.h:查找头文件

代码：errno.c

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <errno.h>
 4 int main(void) {
 5         FILE* fp = fopen("file", "r");
 6         if (!fp) {
 7             printf("fopen: %d\n", errno);
 8             printf("fopen: %s\n", strerror(errno));
 9             perror("fopen");
10             printf("fopen: %m\n");
11             return -1;
12         }
13         // ...
14         fclose(fp);
15         return 0;
16 }
```

注意：函数在出错时会将一个大于零的整数作为错误号存放到全局变量 errno 中，但如果该函数执行成功，通常并不会将 errno 变量赋 0，因此不能用 errno 是否为 0 作为函数成功失败的判断依据，而是根据函数的返回值判断其成功或失败，只有在确定失败的前提下，才能根据 errno 判断具体的失败原因。

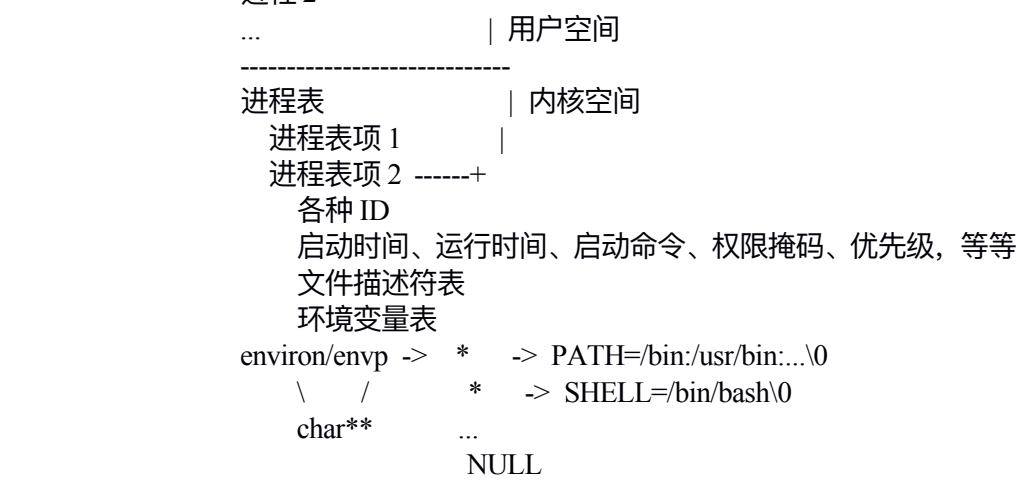
代码：iferr.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <errno.h>
 4 void foo(void) {
 5         FILE* fp = fopen("none", "r");
 6         if (!fp)
 7                 return;
 8         // ...
 9         fclose(fp);
10 }
11 int main(void) {
12         foo();
13         int* p = (int*)malloc(sizeof(int)/* * 0xFFFFFFFF */);
14         //if (errno) {
15         if (!p) {
16                 perror("malloc");
17                 return -1;
18         }
19         *p = 1234;
20         printf("%d\n", *p);
21         free(p);
22         return 0;
23 }
```

四、环境变量

1.环境变量表

每个进程在系统内核中都有一个数据结构来维护与该进程有关的审计信息，称为进程表项，其中就包含了一张专属于该进程的环境变量表。环境变量表其实就是一个以 NULL 指针结尾的字符指针数组。其中每个字符指针类型的数组元素都指向一个以空字符('\0')结尾的字符串。该字符串形如：变量名=变量值，即一个环境变量。



...
...
int main(int argc, char* argv[], char* envp[]) { ... }
environ 需要在主函数声明
2.访问调用进程的环境变量(头文件#include<stdlib.h>)
1)根据一个环境变量的名称获取其值
char* getenv(const char* name);
成功返回与给定环境变量名匹配的环境变量值，失败返回 NULL。
name - 环境变量名
2)修改或新增环境变量
int putenv(char* name_value);
成功返回 0，失败返回非零。
name_value - 形如“变量名=变量值”的字符串，若变量名不存在则新增该环境变量，若变量名已存在则修改其值。
int setenv(const char* name, const char* value,
int overwrite);
成功返回 0，失败返回-1。
name - 环境变量名。若该变量名不存在，则新增环境变量，若已存在则根据 overwrite 的值或修改或保留原值。
value - 环境变量值
overwrite - 当 name 参数所表示的环境变量名已存在时，是否用 value 覆盖原来的变量值，非零则覆盖，零则不覆盖。
3)根据名称删除环境变量
int unsetenv(const char* name);
成功返回 0，失败返回-1。
4)清空环境变量表
int clearenv(void);
成功返回 0，失败返回非零。
一旦环境变量表被清空，全局变量 environ 即成为 NULL 指针。

代码：env.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 void penv(char** envp) {
 4         printf("---- 环境变量表 ----\n");
 5         while(envp && *envp)
 6                 printf("%s\n", *envp++);
 7         printf("--------------------\n");
 8 }
 9 int main(int argc, char* argv[], char* envp[]) {
10         penv(envp);
11         extern char** environ;
12         penv(environ);
13         printf("%s\n", getenv("PATH"));
14         putenv("TEACHER=Youchengwei");
```

```
15              penv(environ);
16              putenv("TEACHER=Minwei");
17              penv(environ);
18              setenv("CITY", "Beijing", 0);
19              penv(environ);
20              setenv("CITY", "Shanghai", 0);
21              penv(environ);
22              setenv("CITY", "Shanghai", 1);
23              penv(environ);
24              unsetenv("TEACHER");
25              penv(environ);
26              clearenv();
27              penv(environ);
28              printf("environ: %p\n", environ);
29              return 0;
30  }
```

五、内存

应用程序：根据业务逻辑选择合适的数据结构

STL：容器、迭代器、内存分配器

C++：new、delete、delete[]　　用户层

C：malloc、calloc、realloc、free

POSIX：sbrk、brk　　系统层

Linux：mmap、munmap

系统内核：kmalloc、vmalloc　　内核层

硬件驱动：get_free_page

硬件：指令集、电路

1.进程映像

程序是由可执行代码和全局数据组成的磁盘文件。运行程序时，需要将磁盘上的可执行程序文件加载到内存中，以使处理器可以执行其中的代码，处理其中的数据，形成进程。进程在内存空间中的布局形成进程映像，从低地址到高地址依次为：

---------- 4G-1　　HIGH

内核空间

---------- 3G-1

命令行参数表和环境变量表的指针

栈区 stack

非静态局部变量

- - - - - - - - - - - ^ - - - - - - - - - - -　栈顶

|

v

预留空间

^

```
                          |
- - - - - - - - - - - - - v - - - - - - - - - - -   堆顶
                        堆区 heap
                      动态内存分配
-----------------------------------------------
                        BSS 区 bss
不带常属性且未初始化的全局变量和静态局部变量
-----------------------------------------------
                       数据区 data
不带常属性且被初始化的全局变量和静态局部变量
-----------------------------------------------
                     代码区 text(只读)
                       可执行指令
                       字面值常量
带有常属性且被初始化的全局变量和静态局部变量
----------------------------------------------- 0     LOW
```

代码：map.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> // IEEE 制定的 POSIX 标准
const int const_global = 10; // 常全局变量
int init_global = 10; // 初始化全局变量
int uninit_global; // 未初始化全局变量
int main(int argc, char* argv[], char* envp[]) {
        const static int const_static = 10; // 常静态变量
        static int init_static = 10; // 初始化静态变量
        static int uninit_static; // 未初始化静态变量
        const int const_local = 10; // 常局部变量
        int prev_local; // 前局部变量
        int next_local; // 后局部变量
        int* prev_heap = malloc(sizeof(int)); // 前堆变量
        int* next_heap = malloc(sizeof(int)); // 后堆变量
        const char* literal = "literal"; // 字面值常量
        printf("------ 命令行参数和环境变量 ------ <高地址>\n");
        printf("          环境变量：%p\n", envp);
        printf("        命令行参数：%p\n", argv);
        printf("-------------- 栈 ----------------\n");
        printf("        后局部变量：%p\n", &next_local);
        printf("        前局部变量：%p\n", &prev_local);
        printf("        常局部变量：%p\n", &const_local);
        printf("-------------- 堆 ----------------\n");
        printf("          后堆变量：%p\n", next_heap);
        printf("          前堆变量：%p\n", prev_heap);
        printf("-------------- BSS ----------------\n");
        printf("  未初始化全局变量：%p\n", &uninit_global);
        printf("  未初始化静态变量：%p\n", &uninit_static);
```

```
30              printf("------------ 数据 ----------------\n");
31              printf("     初始化静态变量：%p\n", &init_static);
32              printf("     初始化全局变量：%p\n", &init_global);
33              printf("------------ 代码 ----------------\n");
34              printf("          常静态变量：%p\n", &const_static);
35              printf("          字面值常量：%p\n", literal);
36              printf("          常全局变量：%p\n", &const_global);
37              printf("                函数：%p\n", main);
38              printf("---------------------------------- <低地址>\n");
39              return 0;
40  }
```

通过 size 命令+可执行程序可以观察特定可执行程序(的进程实例)的代码区(text)、数据区(data)和 BSS 区(bss)的字节数，以及它们十进制(dec)及十六进制(hex)形式的总和。

```
tarena@ubuntu:~/Desktop/uc$ size map
   text    data     bss     dec     hex filename
   2527     296      12    2835     b13 map
```

```
                / 目标模块(.o) \
Linux 系统的 | 静态库(.a)    | ELF (Executable and Linkable
 二进制模块 | 动态库(.so)    | Format, 可执行可链接文件格式)
                \ 可执行程序    /
```

所谓编译和链接就是将用高级语言编写的文本格式的源代码文件翻译成二进制的机器指令，最终形成 ELF 格式的可执行文件的过程。

所谓可执行程序的加载就是将保存在磁盘上的 ELF 格式的可执行文件读入内存形成进程映像的过程。

2.虚拟内存

32 位机器，地址总线和数据总线的宽度 32 位，地址范围即虚拟内存。

```
                                  _____________________________________|
                                  |
                                  v
  _____________0 -- 2^32-1 (4G-1)_____________
/                                               \
+-------------------+      +--------------------+
|半导体内存 (内存条)|<-->| 交换分区(换页文件) | - 物理内存
+-------------------+      +--------------------+
     ^              v
+--------+ +--------+
|地址总线| |数据总线
+--------+ +--------+
     ^     +---+     v
   物理-|CPU|<数据
   地址 +---+
```

在(32 位)系统中运行的每个进程，都拥有各自独立的，4G 字节大小的地址空间，谓之虚拟内存。用户程序中使用的都是地址空间中的虚拟内存，但真正存储代码和数据的却是永远无法被直接访问的物理内存。广义的物理内存不仅包括半导体材质的内存条，还包括磁盘上的交换分区(swap)或换页文件(PageFile)。当半导体内存不够用时，可以把一些长期闲置的代码和数据从半

导体内存缓存到交换分区或换页文件中，这叫页面换出，一旦需要使用那些曾被换出的代码数据，再把它们从交换分区或换页文件恢复到半导体内存中，这叫页面换入。因此，系统中的虚拟内存可以比半导体内存大很多。虚拟内存到物理内存，或者说地址空间到内存空间的映射关系，由操作系统内核通过内存映射表动态维护。虚拟内存技术一方面保护了系统的安全，把用户进程与内核进程，用户进程与用户进程，在对内存的访问上完全隔离开，另一方面又借助交换分区或换页文件，允许应用程序以一种完全透明的方式，使用比实际半导体内存大得多的存储空间。

何为内存分配?

在地址空间中划出一定大小的范围，加以标记，表示占用。

在物理内存中找到相应大小的空间，建立映射，记映射表。

何为内存释放?

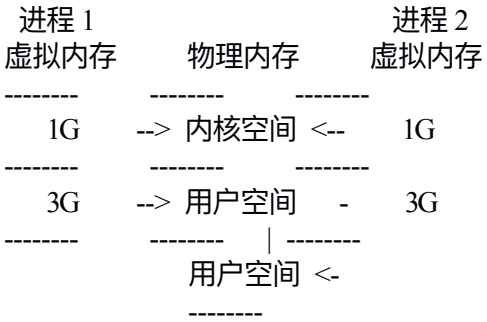
解除物理内存和虚拟内存之间的映射关系，改映射表。

将虚拟内存中的地址范围标为非占用状态，复用地址。

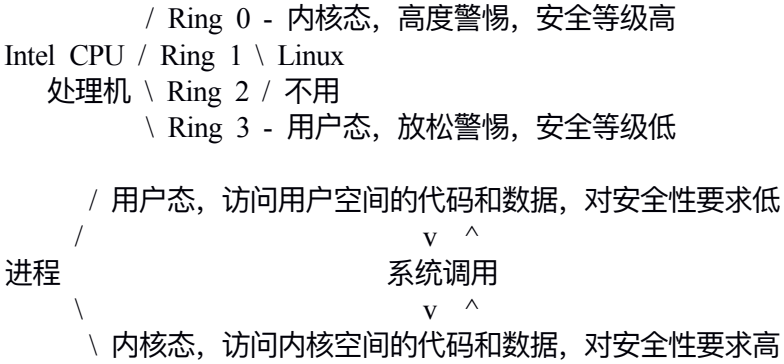
为了避免内存映射建立和解除的频率过高，降低系统运行性能，现代版本的操作系统一般都会通过集约优化的方式减少建立和解除内存映射的频率。

3.用户空间与内核空间

在 32 位系统上的每个进程都拥有“独立”的 4G 字节大小的虚拟内存。其中高地址的 1G 虚拟内存称为内核空间，所映射的物理内存为系统中的所有进程共享，存放系统内核的代码和数据。其余低地址的 3G 虚拟内存称为用户空间，不同的进程映射到不同的物理内存中，存放该进程自己的代码和数据。



用户空间的代码不能直接访问内核空间的代码和数据，可以通过系统调用进入内核态，间接地与系统内核交互。



4.内存壁垒与段错误

1)每个进程的用户空间所对应的物理内存是各自独立的，系统为每个进程的用户空间维护一张专属于该进程的内存映射表，记录虚拟内存到物理内存的映射关系，因此不同进程之间交换虚拟内存地址是毫无意义。

```
------------------------        ------------------------
int a = 10;
printf("%d\n", a); // 10
int* p = &a;                    --> int* q = p;
                                        *q = 20; // 非法，段错误
printf("%d\n", a); // 10
```

2)所有进程的内核空间都对应着相同的物理内存区域，系统为所有进程的内核空间维护同一张内存映射表 init_mm.pgd，记录虚拟内存到物理内存的映射关系，因此不同进程通过系统调用所访问的内核代码和数据都是同一份。

3)用户空间的内存映射表会随着进程的切换而切换，内核空间的内存映射表则无需随着进程的切换而且换。

4)一切对虚拟内存的越权访问都将导致段错误

A.试图访问没有映射到物理内存的虚拟内存

```
int* p;
*p = 10; // 段错误
int* q = (int*)malloc(sizeof(int));
*q = 20; // OK
free(q);
*q = 30; // 段错误
```

B.试图对只读内存做写操作

```
char* p = "Hello, world!";
p[7] = 'W'; // 段错误，字面值是只读的
char s[] = "Hello, world!";
s[7] = 'W'; // OK
```

代码：vm.c

```
 1 #include <stdio.h>
 2 int g_vm = 0;
 3 int main(void) {
 4         printf("&g_vm: %p\n", &g_vm);
 5         printf("输入一个整数：");
 6         scanf("%d%*c", &g_vm);//*表示读出来就完事，不存储
 7         printf("启动另一个进程，输入不同的整数，按<回车>继续...");
 8         getchar();
 9         printf("g_vm: %d\n", g_vm);
10         return 0;
11 }
```

5.虚拟内存的分配与释放

针对进程内存映像中堆内存区域，通过编程的方法，在运行期实现动态内存分配与释放。

堆尾(顶)指针：当前堆内存中最后一个字节的下一个字节的地址。

空堆：

```
L-------------------->H
...|BSS|_
         ^
```

分配 4 字节的堆内存:

```
L-------------------->H
```

```
...|BSS|XXXX_
              ^
```
分配 8 字节的堆内存:
```
L-------------------->H
...|BSS|XXXXYYYYYYYY_
                      ^
```
释放 4 字节的堆内存:
```
L-------------------->H
...|BSS|XXXXYYYY_
                  ^
```
释放 8 字节的堆内存:
```
L-------------------->H
...|BSS|_
          ^
```
1)sbrk

以相对方式分配和释放虚拟内存。

```
#include  <unistd.h>
void*  sbrk(intptr_t  increment);
```
成功返回调用该函数之前的堆尾指针，失败返回-1。

increment: 堆内存增加字节数

>0 - 分配虚拟内存

<0 - 释放虚拟内存

=0 - 获取当前堆尾

```
        void*  p  =  sbrk(4);
                      |
                      v
...|BSS|OOOOXXXX_
                  ^
++++++++++++**********
\__________/\_________/
     4096          4096  (4K=页)
            void*  q  =  sbrk(0);
                         |
                         v
...|BSS|OOOOXXXX_
                  ^
++++++++++++**********
\__________/\_________/
     4096          4096  (4K=页)
            void*  r  =  sbrk(-8);
                         |
                         v
...|BSS|_
          ^
++++++++++++
\__________/
     4096
```

代码：sbrk.c

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <unistd.h>
 4 int main(void) {
 5         setbuf(stdout, NULL);
 6         printf("当前堆尾：%p\n", sbrk(0));
 7         int* p1 = sbrk(sizeof(int));
 8         *p1 = 123;
 9         printf("%d\n", *p1);
10         double* p2 = sbrk(sizeof(double));
11         *p2 = 4.56;
12         printf("%g\n", *p2);
13         char* p3 = sbrk(sizeof(char) * 256);
14         strcpy(p3, "Hello, World!");
15         printf("%s\n", p3);
16         sbrk(-(sizeof(char) * 256 + sizeof(double) + sizeof(int)));
17         printf("当前堆尾：%p\n", sbrk(0));
18         return 0;
19 }
```

2)brk

以绝对方式分配和释放虚拟内存。

```
#include <unistd.h>
int brk(void* end_data_segment);
```

成功返回 0，失败返回-1。

end_data_segment: 新堆尾指针

>当前堆尾指针 - 分配虚拟内存

<当前堆尾指针 - 释放虚拟内存

=当前堆尾指针 - 什么也没有做

原堆尾指针+堆内存增量=新堆尾指针

```
        void* p = sbrk(0); brk(p+4);
                  |
                  v
...|BSS|OOOOXXXX_
                   ^
++++++++++++**********
\__________/\_________/
     4096         4096  (4K=页)
          brk(p)
            |
            v
...|BSS|OOOO_
            ^
++++++++++++
\__________/
```

4096

代码：brk.c

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <unistd.h>
 4 int main(void) {
 5         setbuf(stdout, NULL);
 6         printf("当前堆尾：%p\n", sbrk(0));
 7         int* p1 = sbrk(0);
 8         brk(p1 + 1);
 9         *p1 = 123;
10         printf("%d\n", *p1);
11         double* p2 = (double*)(p1 + 1);
12         brk(p2 + 1);
13         *p2 = 4.56;
14         printf("%g\n", *p2);
15         char* p3 = (char*)(p2 + 1);
16         brk(p3 + 256);
17         strcpy(p3, "Hello, World!");
18         printf("%s\n", p3);
19         brk(p1);
20         printf("当前堆尾：%p\n", sbrk(0));
21         return 0;
22 }
```

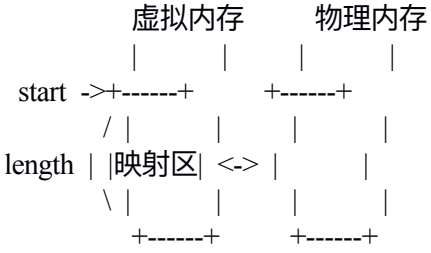
```
   int  double  char[256]

   __   ______   _______
  /  \/       \/        \
  IIIIDDDDDDDDCCCCCC...
  ^    ^         ^         ^
  |    |         |         |
  p1   p2=p1+1  p3=p2+1  p3+256
  ^        |         |        |
  |        v         v        v
sbrk(0)  brk       brk      brk
```

事实上，sbrk 和 brk 不过是移动堆尾指针的两种不同方法，移动过程中还要兼顾虚拟内存和物理内存之间映射关系的建立和解除(以页为单位)。用 sbrk 分配内存比较方便，用多少内存就传多少增量参数，同时返回指向新分配内存区域的指针，但用 sbrk 做一次性内存释放比较麻烦，因为必须将所有的既往增量进行累加。用 brk 释放内存比较方便，只需将堆尾指针设回到一开始的位置即可，但用 brk 分配内存比较麻烦，因为必须根据所需要的内存大小计算出堆尾指针的绝对位置。最好的方法是将这两个函数结合起来用，分配内存用 sbrk，一次性释放内存用 brk。

代码：sb.c

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <unistd.h>
 4 int main(void) {
 5         setbuf(stdout, NULL);
 6         printf("当前堆尾：%p\n", sbrk(0));
 7         int* p1 = sbrk(sizeof(int));
 8         *p1 = 123;
 9         printf("%d\n", *p1);
10         double* p2 = sbrk(sizeof(double));
11         *p2 = 4.56;
12         printf("%g\n", *p2);
13         char* p3 = sbrk(sizeof(char) * 256);
14         strcpy(p3, "Hello, World!");
15         printf("%s\n", p3);
16         brk(p1);
17         printf("当前堆尾：%p\n", sbrk(0));
18         return 0;
19 }
```

3)mmap
建立虚拟内存到物理内存或磁盘文件的映射。
#include <sys/mman.h> linux 自己的
void* mmap(void* start, size_t length, int prot,int flags, int fd, off_t offset);
成功返回映射区(虚拟)内存起始地址，失败返回 MAP_FAILED(-1)。
start: 映射区(虚拟)内存起始地址，NULL 系统自动选定后返回
length: 映射区的字节数，自动按页(4096)圆整
        int((length+4096)/4096)*4096
prot: 访问权限，可取以下值：
    PROT_READ - 映射区可读
    PROT_WRITE - 映射区可写
    PROT_EXEC - 映射区可执行
    PROT_NONE - 映射区不可访问
    可以用位或的方式连接
flags: 映射标志
    MAP_ANONYMOUS|MAP_PRIVATE - 虚拟内存到物理内存的映射
    MAP_SHARED - 虚拟内存到磁盘文件的映射
fd   \ 只用于虚拟内存到磁盘文件的映射，
offset/ 虚拟内存到物理内存的映射，忽略之



| | | |

4)munmap
解除虚拟内存到物理内存或磁盘文件的映射。
int munmap(void* start, size_t length);
成功返回 0，失败返回-1。
start: 映射区(虚拟)内存起始地址，必须是内存页边界
length: 映射区的字节数，自动按页(4096)圆整
int((length+4096)/4096)*4096
代码：mmap.c

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <sys/mman.h>
 4 int main(void) {
 5         char* p = mmap(/*NULL*/sbrk(0), 8192, PROT_READ | PROT_WRITE,
 6                 MAP_ANONYMOUS | MAP_PRIVATE, 0, 0);
 7         if (p == MAP_FAILED) {
 8                 perror("mmap");
 9                 return -1;
10         }
11         char* q = mmap(p + 4096, 8192, PROT_READ | PROT_WRITE,
12                 MAP_ANONYMOUS | MAP_PRIVATE, 0, 0);
13         if (q == MAP_FAILED) {
14                 perror("mmap");
15                 return -1;
16         }
17         sprintf(p, "Hello, World!");
18         printf("%s\n", p);
19         if (munmap(p, 4096) == -1) {
20                 perror("munmap");
21                 return -1;
22         }
23         //printf("%s\n", p);
24         sprintf(p + 4096, "Hello, Linux!");
25         printf("%s\n", p + 4096);
26         if (munmap(p + 4096, 4096) == -1) {
27                 perror("munmap");
28                 return -1;
29         }
30         //printf("%s\n", p + 4096);
31         sprintf(q + 4096, "Hello, Tarena!");
32         printf("%s\n", q + 4096);
33         if (munmap(q + 4096, 4096) == -1) {
34                 perror("munmap");
35                 return -1;
36         }
37         return 0;
```

```
38 }
```

六、系统调用

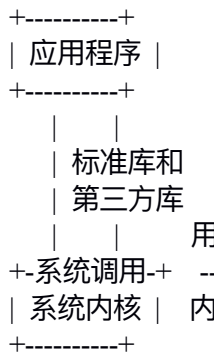
1.Unix 应用的层次结构

1)Unix 系统的大部分功能都是通过系统调用实现的，如 open、close 等。

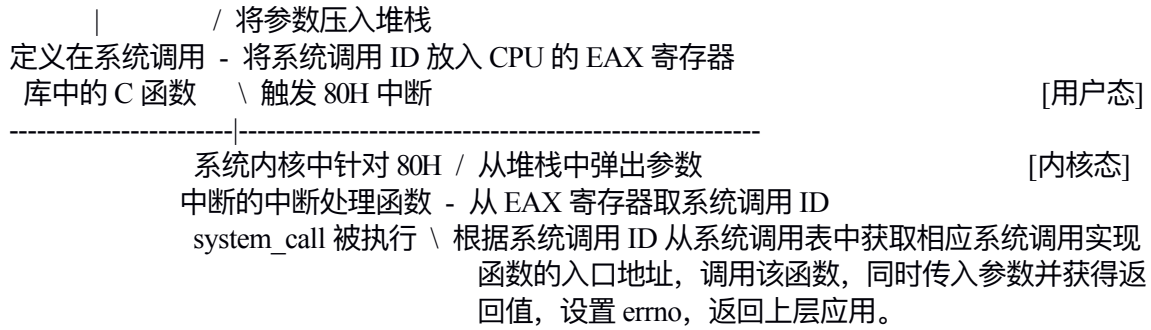
2)Unix 的系统调用已被封装成 C 函数的形式，但它们并不是 C 语言标准库的一部分。

3)标准库函数大部分时间运行在用户态，但部分函数偶尔也会调用系统调用进入内核态，如 malloc、free 等。

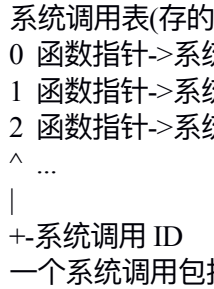
4)程序员自己编写的代码也可以跳过标准库，直接使用系统调用，如 brk、sbrk、mmap、munmap 等，与操作系统内核交互，进入内核态。



2.系统调用的执行过程



系统调用表(存的都是函数指针)



一个系统调用包括两个部分，一个是在用户态执行的，定义在系统调用库中的 C 函数，另一个是在内核态执行的，提供实际功能的系统调用实现函数。二者之间通过 80H 中断相互联系。

3.用户时间和系统时间

一个进程在它的运行过程中，时而处于用户态，访问用户空间的资源，时而又会处于内核态，访问内核空间的资源。进程在这两种状态下的时间，分别称为用户时间和系统时间。

进程

分配内存 - malloc/sbrk/brk/mmap - 内核态(3)
|
存入数据 - *p=10; *q = 20; - 用户态(2)
|
打开文件 - fp=fopen(...); - 内核态(1)
|
计算数据 - r=*p+*q; - 用户态(4)
|
睡眠 - sleep(10); - 挂起态(10)
|
将结果写入文件 - fprintf(fp,"%d", r); - 内核态(5)
|
释放内存 - free/sbrk/brk/munmap - 内核态(6)
|
关闭文件 - fclose(fp); - 内核态(7)

用户时间=2+4=6 \
系统时间=3+1+5+6+7=22 - 6+22+10=38 - 墙钟时间
睡眠时间=10 /
time 可执行程序
real 0m18.705s - 墙钟时间
user 0m0.452s - 用户时间
sys 0m18.173s - 系统时间

七、文件

1.文件系统=数据结构+管理软件

内存：临时性的保存数据。
文件：持久化的保存数据。
物理介质 \
数据结构 - 文件系统
存取访问 /
物理结构：

主要部件：磁头、驱动臂、盘片、主轴、马达、控制电路。

存储原理：利用磁性存储元的磁场方向记录二进制的 0 和 1。

读写原理：通过电生磁写入数据，通过磁生电读取数据。

磁道和扇区：磁盘旋转，磁头固定，每个磁头都会在盘片表面划出一个圆形轨迹。该表磁头的位置，可以形成若干大小不等的同心圆，这些同心圆就叫做磁道(Track)。对于每个磁道，按照每个扇区 512 字节进行划分，得到若干扇区，扇区是文件的基本存储和访问单位。每个磁道上的扇区数并不相等，越靠外圈的磁道所包含的扇区越多。

柱面、柱面组和分区：不同盘片相同半径的磁道所组成的圆柱成为柱面(Cylinder)。整个硬盘的

柱面数与每张盘片的磁道数相等。根据磁头号确定哪个盘面，根据柱面号确定哪个磁道，根据扇区号确定哪个扇区。因此只要明确数据块的磁头号、柱面号和扇区号，就可以定位该数据在磁盘上的位置。将若干个连续的柱面分为一组，即形成柱面组。在若干个连续的柱面组上建立一个文件系统，即形成分区。

1)数据结构

FAT/FAT32/NTFS

UFS/JFS/DFS/EXT2/EXT3

^^^

硬盘:

|分区(逻辑卷)|分区(逻辑卷)|分区(逻辑卷)|

分区:

|引导块|超级块|柱面组|柱面组|柱面组|

柱面组:

|超级块副本|柱面组信息|i 节点映射表|块位图|i 节点表|数据块集|

1843099->99

i 节点表：ls-i

|i 节点|i 节点|i 节点|...|i 节点|...

0 1 2 99

i 节点:

|元数据|数据块索引表|

数据块索引表:

|数据块索引号|数据块索引号|数据块索引号|

100 200 300

数据块集:

...|数据块|...|数据块|...|数据块|...|数据块|...|数据块|...

100 200 300 400 500

|

|400|500|

```
+---------+---------+---------+---------+
|数据块 100|数据块 200|数据块 400|数据块 500|  ->  文件内容
+---------+---------+---------+---------+
```

存放文件内容的数据块叫直接块，如 100、200、400、500。

存放数据块索引号的数据块叫间接块，如 300。

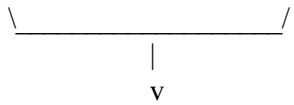
```
                / 磁头号 \
数据块[100]  -  柱面号  |
                \ 扇区号  |
                / 磁头号  |
数据块[200]  -  柱面号  |
                \ 扇区号  /
```

2)管理软件

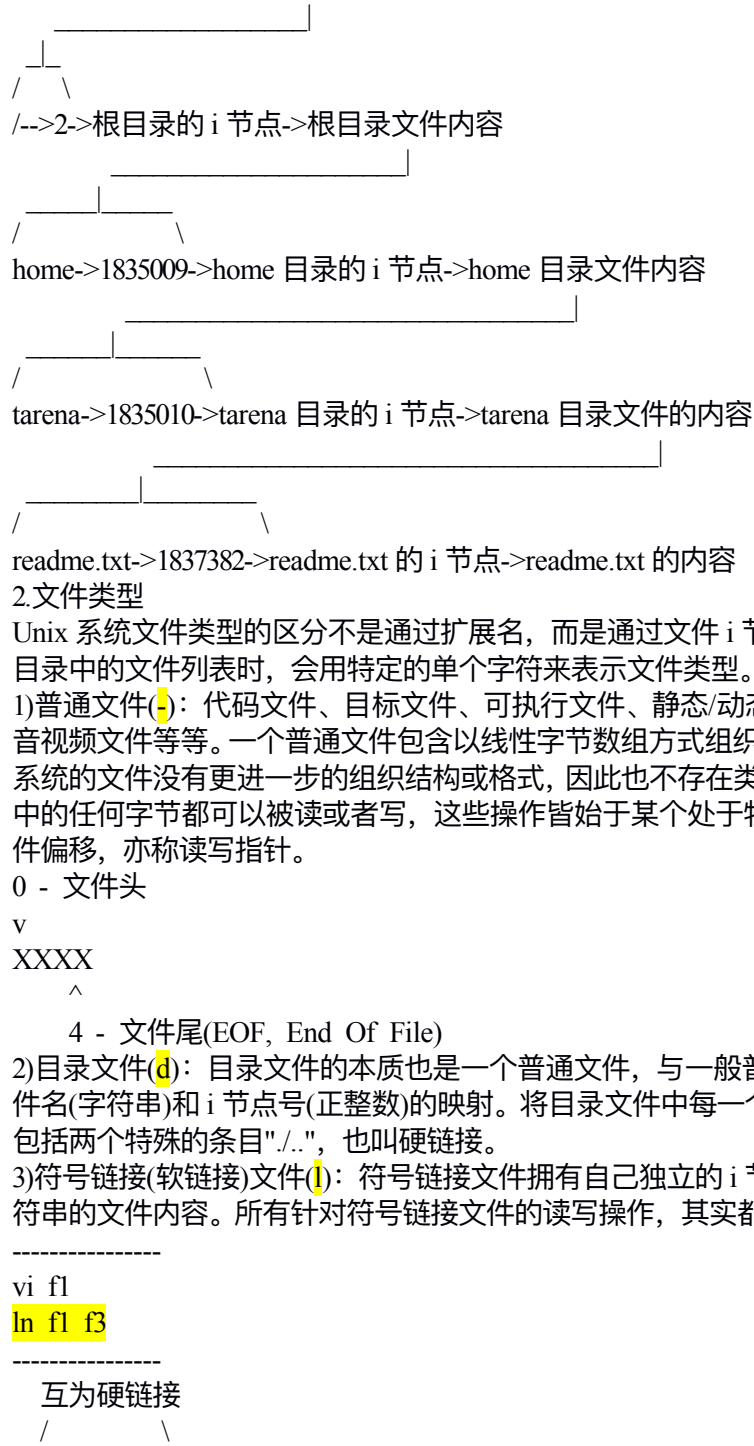
文件路径->目录(文件)->i 节点号->i 节点映射表->i 节点索引

|

```
 +------+------+------+                              v
 |数据块|数据块|数据块|<-数据块索引表<-i 节点<-i 节点表
 +------+------+------+
```



例如：访问/home/tarena/readme.txt 文件



2.文件类型

Unix 系统文件类型的区分不是通过扩展名，而是通过文件 i 节点中的元数据。当用 ls 命令显式目录中的文件列表时，会用特定的单个字符来表示文件类型。

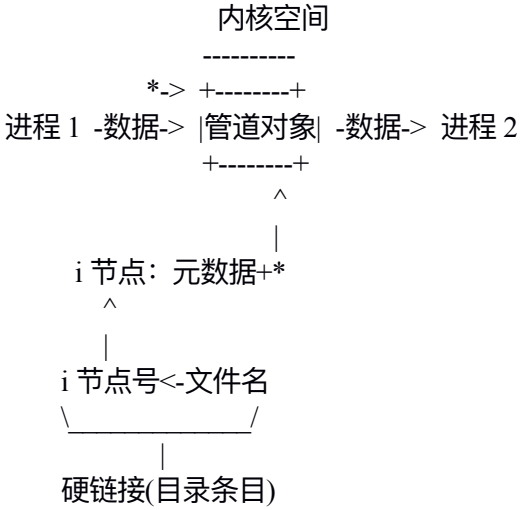
1)普通文件(-)：代码文件、目标文件、可执行文件、静态/动态库文件、文本文件、图像文件、音视频文件等等。一个普通文件包含以线性字节数组方式组织的数据，通常也称为字节流。Unix 系统的文件没有更进一步的组织结构或格式，因此也不存在类似 VMS 系统中记录的概念。文件中的任何字节都可以被读或者写，这些操作皆始于某个处于特定位置的字节，该位置即当前文件偏移，亦称读写指针。

2)目录文件(d)：目录文件的本质也是一个普通文件，与一般普通文件的区别就是它仅仅存储文件名(字符串)和 i 节点号(正整数)的映射。将目录文件中每一个这样的映射称为目录条目，其中包括两个特殊的条目"./..."，也叫硬链接。

3)符号链接(软链接)文件(l)：符号链接文件拥有自己独立的 i 节点和包含一个被链接文件路径字符串的文件内容。所有针对符号链接文件的读写操作，其实都是在读写被其所链接的文件。

```
 /  1837655   \
/     +----+     \
f1->|文件|<-f2
       +----+
----------------
int  file  =  10;
int*  f1  =  &file;  \  指向同一个内存对
int*  f3  =  f1;       /  象的多个一级指针
----------------
vi  f2
ln  -s  f2  f4
----------------
f4 是 f2 的软链接
   /              \
 /  1843094   \
/     +----+     \
f2->|文件|     |
       +----+     |
     1837649     |
       +----+     |
       |"f2"|<-f4
       +----+
----------------
int  file  =  10;
int*  f2  =  &file;  \  最终被解引用到一个
int**  f4  =  &f2;    /  内存对象的多级指针
----------------
```

4)(有名)管道文件(p)：只有 i 节点和目录中的硬链接条目，i 节点中只有元数据而没有数据块索引表。管道文件其实是一个内存中的内核对象，可用于在不同进程之间交换数据。



借助文件系统使不同的进程可以汇合于同一个内核对象。

5)(本地)套接字文件(s)：本质上和管道文件没有区别，也是一种进程间通信的方式，只是因其源自 BSD 关于套接字的 API 集，故与 SVR4 Unix 发生分歧。

6)字符设备文件(c)：设备驱动将一个个字节按顺序写入队列，用户程序从队列中按顺序依次读

出一个个字节，如键盘、串行口。
7)块设备文件(b)：设备驱动将字节数组(块)映射到可寻址设备上，用户程序可以按照任意顺序访问该字节数组中的任意字节，如硬盘。
3.文件的打开和关闭
1)打开/创建文件
#include <fcntl.h>
int open(const char* pathname, int flags, mode_t mode);
成功返回文件描述符，失败返回-1。
pathname: 文件路径
flags: 状态标志，可取以下值：
O_RDONLY - 只读 \
O_WRONLY - 只写 > 只选一个
O_RDWR - 读写 /
O_APPEND - 追加 -- 可以和 O_WRONLY 或 O_RDWR 组合使用
O_CREAT - 创建。不存在则创建，已存在即打开，除非与以下两个标志位组合使用。仅在有此标志位时 mode 参数有效。
O_EXCL - 排斥。不存在则创建，已存在即返回失败。
O_TRUNC - 清空。不存在则创建，已存在即清空其内容，然后再打开。
mode: 权限模式。用三位八进制数表示。
进制前缀 属主(拥有者)用户 (与拥有者)同组(的)用户 其它用户

```
   |          |  |  |              |  |  |          |  |  |
   0          4  2  1              4  2  1          4  2  1
   |          |  |  |              |  |  |          |  |  |
八进制        读 写 行              读 写 行          读 写 行
```

属主用户可读可写可执行，同组用户可读可执行，其它用户只读：
0754 -> mode

```
-------------------- 内存中的用户空间 --------------------
int 文件描述符 = open(...);
read(文件描述符, ...);
----------------------------
FILE* 文件指针 = fopen(...);
              +------------+
文件指针->|      FILE      |
              | ---------- |
              | 文件描述符 |
              +------------+
fread(文件指针, ...);
-------------------- 内存中的内核空间 --------------------
进程表
      进程表项
      进程表项
      文   文件描述符表
      件   0 | 文件描述符标志 | 文件表项指针 |
      描   1 | 文件描述符标志 | 文件表项指针 |
      述   2 | 文件描述符标志 | 文件表项指针 |
```

符 3 | 文件描述符标志 | 文件表项指针 |--+
...
...
+----------+
|
v
文件表项
状态标志
读写位置
v 节点指针--+
+-----+
|
v
v 节点
i 节点
元数据
数据块索引表：|数据块索引|数据块索引|...|
...
---------- 磁盘上的文件系统 ----------
i 节点
元数据
数据块索引表：|数据块索引|数据块索引|...|
2)关闭文件
#include <unistd.h>
int close(int fd);
成功返回 0，失败返回-1。
fd: 文件描述
销毁与被关闭文件有关的内核数据结构。文件描述符表中与该文件描述符相对应的文件表项指针会被置 NULL，该文件描述符表的条目变为空闲，以后再打开文件可以复用该条目。
0
1
2
3 close X 3
4 close X 4
5
6
7

代码：open.c
```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <fcntl.h>
4 int main(void) {
5 int fd1 = open("open.txt", O_RDWR | O_CREAT | O_TRUNC, 0666);
6 if (fd1 == -1) {
```

```
 7                      perror("open");
 8                      return -1;
 9              }
10              printf("fd1: %d\n", fd1);
11              close(fd1);
12              int fd2 = open("open.txt", O_RDWR);
13              if (fd2 == -1) {
14                      perror("open");
15                      return -1;
16              }
17              printf("fd2: %d\n", fd2);
18              close(fd2);
19              //close(fd1);
20              return 0;
21 }
```

3)I/O 重定向

注意：无论是在同一个进程中，还是在不同的进程中，多次通过 open 函数打开同一个文件，系统内核中只会有一个 v 节点，为多个不同的文件描述符所共享。但是，每次打开所创建的文件表项却是独立的，对应不同的文件描述符，共享同一个 v 节点指针。

对于每个进程，系统都会缺省打开三个文件描述符:

STDIN_FILENO (0) - 标准输入(键盘)

STDOUT_FILENO (1) - 标准输出(屏幕，有缓冲区)

STDERR_FILENO (2) - 标准出错(屏幕，无缓冲区)

	UC	标准 C	C++
	文件描述符	文件指针	文件对象
	int	FILE*	iostream
标准输入	0	stdin	cin
标准输出	1	stdout	cout
标准出错	2	stderr	cerr

< 输入文件路径

1> 输出文件路径

2> 出错文件路径

```
close(STDIN_FILENO);
open(输入文件路径); // 0
close(STDOUT_FILENO);
open(输出文件路径); // 1
close(STDERR_FILENO);
open(出错文件路径); // 2
```

代码：redir.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <unistd.h>
 4 #include <fcntl.h>
 5 void doio(void) {
 6              int x, y;
```

```
 7              scanf("%d%d", &x, &y);
 8              printf("%d+%d=%d\n", x, y, x + y);
 9              malloc(0xFFFFFFFFFFFFFFFF);
10              perror("malloc");
11 }
12
13 int main(void) {
14              doio();
15              return 0;
16 }
```

```
tarena@ubuntu:~/Desktop/uc/day04$ vi redir.c
tarena@ubuntu:~/Desktop/uc/day04$ gcc redir.c -o redir
tarena@ubuntu:~/Desktop/uc/day04$ ./redir
123 456
malloc: Cannot allocate memory
123 + 456 = 579
tarena@ubuntu:~/Desktop/uc/day04$ vi i.txt
tarena@ubuntu:~/Desktop/uc/day04$ ./redir <i.txt 1>o.txt 2>e.txt
tarena@ubuntu:~/Desktop/uc/day04$ cat e.txt
```

```
fopen(..., "r");
fopen(..., "w");
fopen(..., "a");
```

代码：redir.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <unistd.h>
 4 #include <fcntl.h>
 5 void redir(void) {
 6              close(STDIN_FILENO);
 7              int fd = open("i.txt", O_RDONLY);                              // 0 -> i.txt
 8              //printf("%d -> i.txt\n", fd);
 9              close(STDOUT_FILENO);
10              fd = open("o.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644); // 1 -> o.txt
11              //printf("%d -> o.txt\n", fd);
12              close(STDERR_FILENO);
13              fd = open("e.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644); // 2 -> e.txt
14              //printf("%d -> e.txt\n", fd);
15 }
16 void resume(void) {
17              close(STDIN_FILENO);
18              stdin = fopen("/dev/tty", "r");   // 0 -> 键盘
19              close(STDOUT_FILENO);
20              stdout = fopen("/dev/tty", "w"); // 1 -> 屏幕
```

```
21              close(STDERR_FILENO);
22              stderr = fopen("/dev/tty", "w"); // 2 -> 屏幕
23              setbuf(stderr, NULL);
24  }
25  void doio(void) {
26              int x, y;
27              scanf("%d%d", &x, &y);                  // 从键盘读取数据
28                                                      // 从标准输入设备读取数据
29                                                      // 从文件描述符为 0 的文件读取数据
30              printf("%d+%d=%d\n", x, y, x + y); // 向屏幕打印数据
31                                                  // 向标准输出设备打印数据
32                                                  // 向文件描述符为 1 的文件打印数据
33              malloc(0xFFFFFFFFFFFFFFFF);
34              perror("malloc");                   // 向屏幕打印数据
35                                                      // 向标准错误设备打印数据
36                                                      // 向文件描述符为 2 的文件打印数据
37  }
38  int main(void) {
39              doio();
40              redir();
41              doio();
42              resume();
43              doio();
44              return 0;
45  }
```

4.文件的读写与随机访问

ssize_t->int,size_t->usigned int

1)写入文件：文件<-内存

```
#include <unistd.h>
ssize_t write(int fd, const void* buf, size_t count);
                    件 ^                  / 内
                    文 \_____________/ 存
```

成功返回实际写入的字节数，失败返回-1。

fd - 文件描述符

buf - 写入缓冲区

count - 期望写入的字节数

2)读取文件：文件->内存

```
#include <unistd.h>
ssize_t read(int fd, void* buf, size_t count);
                  文 \          ^ 存
                    件 \_____/ 内
```

成功返回实际读取的字节数，失败返回-1，如果读写位置已经位于文件尾(文件中最后一个字节的下一个位置，即文件长度)，该函数会直接返回 0。

fd - 文件描述符

buf - 读取缓冲区

count - 期望读取的字节数，一般给 buf 缓冲区的大小

代码：wr.c

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <unistd.h>
 4 #include <fcntl.h>
 5 int main(void) {
 6         int fd = open("wr.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
 7         if (fd == -1) {
 8                 perror("open");
 9                 return -1;
10         }
11         const char* text = "Hello, World!";
12         printf("写入内容：%s\n", text);
13         size_t towrite = strlen(text) * sizeof(text[0]);
14         ssize_t written = write(fd, text, towrite);
15         if (written == -1) {
16                 perror("write");
17                 return -1;
18         }
19         printf("期望写入%lu 字节，实际写入%ld 字节。\n", towrite, written);
20         close(fd);
21         if ((fd = open("wr.txt", O_RDONLY)) == -1) {
22                 perror("open");
23                 return -1;
24         }
25         //char buf[1024] = {};
26         char buf[1024];
27         size_t toread = sizeof(buf) - sizeof(buf[0]);
28         ssize_t readed = read(fd, buf, toread);
29         if (readed == -1) {
30                 perror("read");
31                 return -1;
32         }
33         printf("期望读取%lu 字节，实际读取%ld 字节。\n", toread, readed);
34         buf[readed / sizeof(buf[0])] = '\0';
35         printf("读取内容：%s\n", buf);
36         printf("read 函数返回%ld。\n", read(fd, buf, toread));
37         close(fd);
38         return 0;
39 }
```

hexdump -C wr.txt 以 16 进制查看写入的文件

3)读写二进制文件与读写文本文件

基于 read 和 write 函数读写文件都是面向二进制字节流的 I/O 操作：

内存：10100001 -> HLHLLLLH

文件：NSNSSSSN -> 10100001

因此用这种方式读写二进制文件，无需做任何额外的工作。

如果希望以文本方式保存数据，那么在写入之前需要先将二进制形式的内存数据格式化成字符串，然后再写入，同理在读取到文本文件中的字符串以后，也要从该字符串中解析出二进制形式的数据放到内存中。
内存：10100001 -格式化-> "161"
文件：　　"161" -解析-> 10100001

代码：binary.c

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <fcntl.h>
 4 int main(void) {
 5         int fd = open("binary.dat", O_WRONLY | O_CREAT | O_TRUNC, 0644);
 6         if (fd == -1) {
 7                 perror("open");
 8                 return -1;
 9         }
10         char name[256] = "张飞";
11         if (write(fd, name, sizeof(name)) == -1) {
12                 perror("write");
13                 return -1;
14         }
15         unsigned int age = 38;
16         if (write(fd, &age, sizeof(age)) == -1) {
17                 perror("write");
18                 return -1;
19         }
20         double salary = 20000;
21         if (write(fd, &salary, sizeof(salary)) == -1) {
22                 perror("salary");
23                 return -1;
24         }
25         struct Employee {
26                 char name[256];
27                 unsigned int age;
28                 double salary;
29         }       employee = {"赵云", 25, 8000};
30         if (write(fd, &employee, sizeof(employee)) == -1) {
31                 perror("write");
32                 return -1;
33         }
34         close(fd);
35         if ((fd = open("binary.dat", O_RDONLY)) == -1) {
36                 perror("open");
37                 return -1;
38         }
39         if (read(fd, name, sizeof(name)) == -1) {
40                 perror("read");
```

```
41                      return -1;
42              }
43              printf("姓名：%s\n", name);
44              if (read(fd, &age, sizeof(age)) == -1) {
45                      perror("read");
46                      return -1;
47              }
48              printf("年龄：%u\n", age);
49              if (read(fd, &salary, sizeof(salary)) == -1) {
50                      perror("read");
51                      return -1;
52              }
53              printf("工资：%g\n", salary);
54              if (read(fd, &employee, sizeof(employee)) == -1) {
55                      perror("read");
56                      return -1;
57              }
58              printf("员工：%s %u %g\n", employee.name, employee.age, employee.salary);
59              close(fd);
60              return 0;
61 }
```

text.c

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <unistd.h>
 4 #include <fcntl.h>
 5 int main(void) {
 6              int fd = open("text.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
 7              if (fd == -1) {
 8                      perror("open");
 9                      return -1;
10              }
11              char name[256] = "张飞";
12              unsigned int age = 38;
13              double salary = 20000;
14              char buf[1024];
15              sprintf(buf, "%s %u %g\n", name, age, salary);
16              if (write(fd, buf, strlen(buf) * sizeof(buf[0])) == -1) {
17                      perror("write");
18                      return -1;
19              }
20              struct Employee {
21                      char name[256];
22                      unsigned int age;
23                      double salary;
24              }       employee = {"赵云", 25, 8000};
```

```
25          sprintf(buf, "%s %u %g", employee.name, employee.age, employee.salary);
26          if (write(fd, buf, strlen(buf) * sizeof(buf[0])) == -1) {
27                  perror("write");
28                  return -1;
29          }
30          close(fd);
31          if ((fd = open("text.txt", O_RDONLY)) == -1) {
32                  perror("open");
33                  return -1;
34          }
35          memset(buf, 0, sizeof(buf));
36          if (read(fd, buf, sizeof(buf) - sizeof(buf[0])) == -1) {
37                  perror("read");
38                  return -1;
39          }
40          sscanf(buf, "%s%u%lf%s%u%lf", name, &age, &salary,
41                  employee.name, &employee.age, &employee.salary);
42          printf("姓名：%s\n", name);
43          printf("年龄：%u\n", age);
44          printf("工资：%g\n", salary);
45          printf("员工：%s %u %g\n", employee.name, employee.age, employee.salary);
46          close(fd);
47          return 0;
48  }
```

4)顺序访问与随机访问

打开文件->文件描述符->文件表项指针->文件表项

状态标志

*读写位置

v 节点指针->v 节点

i 节点信息

...

...

A.文件读写位置(文件指针)和顺序访问

每个打开的文件都有一个与其相关的文件读写位置保存在文件表项中，用以记录从文件头开始计算的字节偏移。文件读写位置通常是一个非负的整数，用 off_t 类型表示，在 32 位系统上被定义为 long int，而在 64 位系统上则被定义为 long long int。文件读写位置的数据类型决定了所在操作系统中文件的最大字节数。打开一个文件，除非指定了 O_APPEND 标志，否则文件读写位置一律被初始化为 0，即文件首字节的位置。每一次针对文件的读写操作都从当前的文件读写位置开始，并根据所读写的字节数，同步增加文件读写位置，为下一次读写做好准备。因为文件读写位置是保存在文件表项而非 v 节点中，因此通过多次打开同一个文件得到多个文件描述符，各自拥有各自的文件读写位置。

```
fd1 = open("file", O_WRONLY | O_CREAT | O_TRUNC, ...);

^
0 - 读写位置(fd1)
write(fd1, "0123456789", ...);
```

```
0123456789
          ^
          10--读写位置(fd1)
fd2 = open("file", O_RDWR);
0123456789
^         ^
0         10--读写位置(fd1)
|
+--读写位置(fd2)
read(fd2, buf, 3); -> "012"
0123456789
   ^      ^
   3      10--读写位置(fd1)
   |
   +--读写位置(fd2)
write(fd2, "ABC", 3);
012ABC6789
      ^  ^
      6  10--读写位置(fd1)
      |
      +--读写位置(fd2)
```

B.随机访问
通过人为设置文件读写位置，以非顺序的方式读写文件的内容。
~~~~~~~~~~~~~~~~~~~~~~~~

```
#include <unistd.h>  |
                     v
off_t lseek(int fd, off_t offset, int whence);
```

成功返回调整后的文件读写位置，失败返回-1。

```
fd: 文件描述符
offset: 偏移(调整)量相对某个位置的字节数
whence: 偏移(调整)量的相对位置，可取以下值：
        SEEK_SET - 相对于文件头偏移
        SEEK_CUR - 相对于当前位置偏移
        SEEK_END - 相对于文件尾偏移
012345678_
^  ^  ^  ^
|  |  |  |
|  |  |  +--文件尾
|  |  +--当前位置
|  +--目标位置
+--文件头
lseek(fd,  3, SEEK_SET);
lseek(fd, -6, SEEK_END);
lseek(fd, -3, SEEK_CUR);
lseek(fd,  0, SEEK_CUR); // 返回当前位置
lseek(fd,  0, SEEK_END); // 返回文件长度
```
~~~~~~~~~~~~~~~~~~~~~~~~

```
lseek(fd,  3,  SEEK_END);  //  产生文件空洞
write(fd,  "9",  1);
    用 0 填充文件空洞
             |
             v
            / \
012345678    9
           ^^^^
              |
              +--当前位置
lseek(fd,  -3,  SEEK_SET);  //  返回失败
```

代码：seek.c

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <unistd.h>
 4 #include <fcntl.h>
 5 int main(void) {
 6         int fd = open("seek.txt", O_RDWR | O_CREAT | O_TRUNC, 0644);
 7         if (fd == -1) {
 8                 perror("open");
 9                 return -1;
10         }
11         const char* text = "Hello, World!";
12         if (write(fd, text, strlen(text) * sizeof(text[0])) == -1) {
13                 perror("write");
14                 return -1;
15         }
16         // Hello, World!
17         //                ^
18         //                |
19         if (lseek(fd, -6, SEEK_CUR) == -1) {
20                 perror("seek");
21                 return -1;
22         }
23         // Hello, World!
24         //           ^       ^
25         //           |<- 6-|
26         //           7
27         off_t pos = lseek(fd, 0, SEEK_CUR);
28         if (pos == -1) {
29                 perror("lseek");
30                 return -1;
31         }
32         printf("当前读写位置：%ld\n", pos);
33         text = "Linux";
34         if (write(fd, text, strlen(text) * sizeof(text[0])) == -1) {
```

```
35                      perror("write");
36                      return -1;
37              }
38              // Hello, Linux!
39              //                  ^
40              //                  |
41              if (lseek(fd, 8, SEEK_END) == -1) {
42                      perror("lseek");
43                      return -1;
44              }
45              // Hello, Linux!
46              //                  ^^^^^^^^^
47              //                           |
48              text = "<-This is a hole.";
49              if (write(fd, text, strlen(text) * sizeof(text[0])) == -1) {
50                      perror("write");
51                      return -1;
52              }
53              // Hello, Linux!            <-This is a hole
54              //                  ^^^^^^^^                    ^
55              //                                              |
56              /* 无法将读写位置置于文件头之前
57              if (lseek(fd, -8, SEEK_SET) == -1) {
58                      perror("lseek");
59                      return -1;
60              } */
61              off_t size = lseek(fd, 0, SEEK_END);
62              if (size == -1) {
63                      perror("lseek");
64                      return -1;
65              }
66              printf("文件总字节数：%ld\n", size);
67              close(fd);
68              return 0;
69      }
```

5.系统 I/O 与标准 I/O

系统 I/O：通过 open、read、write 和 close 等系统调用实现文件 I/O。

标准 I/O：通过 fopen、fread、fwrite 和 fclose 等标注库函数实现文件 I/O。

代码：stdio.c

```
 1 #include <stdio.h>
 2 int main(void) {
 3         FILE* fp = fopen("stdio.dat", "wb");
 4         if (!fp) {
 5                 perror("fopen");
 6                 return -1;
 7         }
 8         //setbuf(fp, NULL);
 9         for (int i = 0; i < 1000000; ++i)
10                 fwrite(&i, sizeof(i), 1, fp);
11         fclose(fp);
12         return 0;
13 }
```

sysio.c

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <fcntl.h>
 4 int main(void) {
 5         int fd = open("sysio.dat", O_WRONLY | O_CREAT | O_TRUNC, 0644);
 6         if (fd == -1) {
 7                 perror("open");
 8                 return -1;
 9         }
10         for (int i = 0; i < 1000000; ++i)
11                 write(fd, &i, sizeof(i));
12         close(fd);
13         return 0;
14 }
```

当系统调用函数被执行时，需要在用户态和内核态之间来回切换，因此频繁执行系统调用函数会严重影响性能。标准库做了必要的优化，内部维护一个缓冲区，只在满足特定条件时才将缓冲区与系统内核同步，借此降低执行系统调用的频率，减少进程在用户态和内核态之间来回切换的次数，提高运行性能。

6.复制文件描述符

```
int fd1 = open("file", ...);
int fd2 = fd1; // 这只是变量赋值，并非复制文件描述符
```

文件描述符表

```
0 | 文件描述符标志 | 文件表项指针 | -> 文件表项
1 | 文件描述符标志 | 文件表项指针 | -> 文件表项
2 | 文件描述符标志 | 文件表项指针 | -> 文件表项
3 | 文件描述符标志 | 文件表项指针 | -> 文件表项
  \_____________________________/          ^
                 复 | 制                    |
  _________________v_____________           |
 /                               \          |
```

7 | 文件描述符标志 | 文件表项指针 | -------
通过多次打开同一个文件所得到的可操作同一个文件的不同文件描述符，其各自拥有各自独立的文件表项，因此其中的读写位置也是独立的。但是通过复制文件描述符所得到的可操作同一个文件的不同文件描述符，同享同一个文件表项，因此其中的读写位置也是共用的。

```
#include <unistd.h>
int dup(int oldfd);
```

成功返回新文件描述符，失败返回-1。

oldfd: 旧文件描述符

dup 函数将 oldfd 参数所对应的文件描述符表条目复制到文件描述符表中第一个空闲条目中，同时返回该条目所对应的新文件描述符。

代码：dup.c

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
int main(void) {
        int fd1 = open("dup.txt", O_RDWR | O_CREAT | O_TRUNC, 0644);
        if (fd1 == -1) {
                perror("open");
                return -1;
        }
        printf("fd1: %d\n", fd1);
        //int fd2 = open("dup.txt", O_RDWR);
        int fd2 = dup(fd1);
        if (fd2 == -1) {
                perror("open");
                return -1;
        }
        printf("fd2: %d\n", fd2);
        //int fd3 = open("dup.txt", O_RDWR);
        int fd3 = dup(fd2);
        if (fd3 == -1) {
                perror("open");
                return -1;
        }
        printf("fd3: %d\n", fd3);
        const char* text = "Hello, World!";
        if (write(fd1, text, strlen(text) * sizeof(text[0])) == -1) {
                perror("write");
                return -1;
        }
        if (lseek(fd2, -6, SEEK_END) == -1) {
                perror("lseek");
                return -1;
        }
```

```
35              // open:
36              //   Hello, World!
37              //   ^          ^        ^
38              //   |          |        |
39              // fd3        fd2      fd1
40              // dup:
41              //   Hello, World!
42              //              ^
43              //              |
44              //       fd1/fd2/fd3
45              text = "Linux";
46              if (write(fd3, text, strlen(text) * sizeof(text[0])) == -1) {
47                      perror("write");
48                      return -1;
49              }
50              // open:
51              //   Linux, World!
52              //          ^ ^        ^
53              //          | |        |
54              //        fd3 fd2    fd1
55              // dup:
56              //   Hello, Linux!
57              //                    ^
58              //                    |
59              //              fd1/fd2/fd3
60              close(fd3);
61              close(fd2);
62              close(fd1);
63              return 0;
64      }
```

```
struct Employee { ... };
+---+---+---+---+---+
| 1 | 2 | 3 | 4 | 5 |
+---+---+---+---+---+
+---+---+---+---+
| 1 | 2 | 4 | 5 |
+---+---+---+---+
```

int dup2(int oldfd, int newfd);

成功返回目标文件描述符(newfd)，失败返回-1。

oldfd: 旧文件描述符

newfd: 新文件描述符

dup2 函数的功能与 dup 函数几乎完全一样，唯一的不同就是允许调用者通过 newfd 参数指定目标文件描述符，正常情况下该函数的返回值与 newfd 参数的值相等。如果所指定的新文件描述符 newfd 并非空闲，即与某个处于打开状态的文件相关联，那么 dup2 函数会关闭该文件，令 newfd 文件描述符空闲，然后再行复制。

代码：dup2.c

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <unistd.h>
 4 #include <fcntl.h>
 5 int main(void) {
 6         int fd1 = open("dup2.txt", O_RDWR | O_CREAT | O_TRUNC, 0644);
 7         if (fd1 == -1) {
 8                 perror("open");
 9                 return -1;
10         }
11         printf("fd1: %d\n", fd1);
12         int fd2 = dup(fd1);
13         if (fd2 == -1) {
14                 perror("open");
15                 return -1;
16         }
17         printf("fd2: %d\n", fd2);
18         //int fd3 = dup2(fd2, 100);
19         int fd3 = dup2(fd2, STDOUT_FILENO);
20         if (fd3 == -1) {
21                 perror("open");
22                 return -1;
23         }
24         printf("fd3: %d\n", fd3);
25         const char* text = "Hello, World!";
26         if (write(fd1, text, strlen(text) * sizeof(text[0])) == -1) {
27                 perror("write");
28                 return -1;
29         }
30         if (lseek(fd2, -6, SEEK_END) == -1) {
31                 perror("lseek");
32                 return -1;
33         }
34         // open:
35         //   Hello, World!
36         //   ^          ^        ^
37         //   |          |        |
38         // fd3       fd2      fd1
39         // dup:
40         //   Hello, World!
41         //              ^
42         //              |
43         //       fd1/fd2/fd3
44         text = "Linux";
45         if (write(fd3, text, strlen(text) * sizeof(text[0])) == -1) {
46                 perror("write");
```

```
47                      return -1;
48              }
49              // open:
50              //   Linux, World!
51              //          ^ ^       ^
52              //          | |       |
53              //        fd3 fd2   fd1
54              // dup:
55              //   Hello, Linux!
56              //                ^
57              //                |
58              //             fd1/fd2/fd3
59              close(fd3);
60              close(fd2);
61              close(fd1);
62              return 0;
63      }
```

7.写缓冲与文件同步

1)写缓冲与延迟写

当一个运行在用户态的进程发起 write 系统调用时，系统内核进行几项检查，然后直接将通过参数传入的数据块拷贝到一个被称为"写缓冲"的缓冲区中，随即返回调用进程。稍后，运行在后台系统内核收集所有这样的"脏"缓冲区，将它们按照一定的顺序排入写队列，并逐一写入磁盘，这个过程被称为"回写(writeback)"，而这种将回写操作延迟执行的策略被称为"延迟写"。延迟写一方面使运行在用户态的调用进程不必等待磁盘动作(通常较慢)的实际完成，提高其运行速度，另一方面使系统内核得以将实际的写磁盘工作推迟到相对空闲的时候完成，且以批量方式集中处理，提高内核的工作效率。write 系统调用的成功返回，仅表示通过其参数传入的数据块，已被成功地拷贝到系统内核负责维护的写缓冲中，而此后任何在回写过程中发生的错误，如物理磁盘驱动器故障等，都不会被报告给发出写了请求的进程。如果在脏缓冲区中的数据被实际写入磁盘之前，系统发生了某种不可预期的异常，如软件崩溃、硬件故障或者突然掉电等，写缓冲中数据将永远失去被写入磁盘的机会，造成数据丢失，即失步，这对许多关键业务系统而言无疑是致命的。

2)文件同步：通过一些系统调用或者参数设置，牺牲写延迟所带来的性能优化，保证数据一致且安全。

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

成功返回 0，失败返回-1。

强制将与与 fd 文件有关的脏缓冲区中的数据和元数据回写到磁盘上。

```
int fdatasync(int fd);
```

成功返回 0，失败返回-1。

强制将与 fd 文件有关的脏缓冲区中的数据(但不包括元数据)回写到磁盘上。该函数的执行速度要快于 fsync 函数。

```
void sync(void);
```

永远成功，无返回值。

强制将系统中所有脏缓冲区排入写队列，随即返回，并不等待写磁盘操作完成。通常一次 sync 函数的调用可能需要消耗数分钟的等待时间。

很多硬盘在设计上，出于性能的考虑也会带有缓存，因此那些所谓被回写到磁盘上的关键数据，也未必就能在第一时间被真正存储在硬盘的永久介质上。硬盘缓存中的数据同样存在丢失的可

能。
3)在数据 I/O 的过程中，缓冲区无处不在。
应用程序缓冲区---+
| fwrite |
v |
标准 I/O 库缓冲区 | write
| fflush |
v |
系统内核缓冲区<--+
| fsync/fdatasync/sync
v
硬盘设备缓冲区
4)对于读操作而言，相对于延迟写的预读取优化意义不大。因为应用程序的设计者通常都是仅在需要读取的时候才会读取，而且需要多少数据就读多少数据，因此提前预读比实际需要的更多的数据并不象延迟写那么有价值。多数操作系统的内核也不支持所谓的预读取优化。
8.文件控制
#include <fcntl.h>
int fcntl(int fd, int cmd, ...);
fd - 文件描述符
cmd - 控制命令，不同的控制命令，执行不同的操作，
后续参数因不同操作而异
该函数的返回值同样因不同操作而异，但如果失败肯定会返回-1。
1)复制文件描述符
int fcntl(int oldfd, F_DUPFD, int newfd);
成功返回目标文件描述符(可能是 newfd)，失败返回-1。
oldfd: 旧文件描述符
newfd: 新文件描述符
通过 fcntl 函数复制文件描述符与通过 dup2 函数复制文件描述符几乎完全一样，唯一的不同就是当新文件描述符 newfd 并非空闲，而是处于打开状态时，fcntl 函数并不会将其关闭，而是另找一个空闲的文件描述符作为复制目标，这时函数的返回值将不同于传给它的 newfd 参数。

代码：dup3.c

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <unistd.h>
 4 #include <fcntl.h>
 5 int main(void) {
 6         int fd1 = open("dup3.txt", O_RDWR | O_CREAT | O_TRUNC, 0644);
 7         if (fd1 == -1) {
 8                 perror("open");
 9                 return -1;
10         }
11         printf("fd1: %d\n", fd1);
12         int fd2 = dup(fd1);
13         if (fd2 == -1) {
14                 perror("open");
15                 return -1;
```

```
16          }
17          printf("fd2: %d\n", fd2);
18          //int fd3 = fcntl(fd2, F_DUPFD, 100);
19          int fd3 = fcntl(fd2, F_DUPFD, STDOUT_FILENO);
20          if (fd3 == -1) {
21                  perror("open");
22                  return -1;
23          }
24          printf("fd3: %d\n", fd3);
25          const char* text = "Hello, World!";
26          if (write(fd1, text, strlen(text) * sizeof(text[0])) == -1) {
27                  perror("write");
28                  return -1;
29          }
30          if (lseek(fd2, -6, SEEK_END) == -1) {
31                  perror("lseek");
32                  return -1;
33          }
34          // open:
35          //   Hello, World!
36          //   ^          ^        ^
37          //   |          |        |
38          // fd3       fd2     fd1
39          // dup:
40          //   Hello, World!
41          //              ^
42          //              |
43          //       fd1/fd2/fd3
44          text = "Linux";
45          if (write(fd3, text, strlen(text) * sizeof(text[0])) == -1) {
46                  perror("write");
47                  return -1;
48          }
49          // open:
50          //   Linux, World!
51          //          ^ ^        ^
52          //          | |        |
53          //        fd3 fd2    fd1
54          // dup:
55          //   Hello, Linux!
56          //                    ^
57          //                    |
58          //              fd1/fd2/fd3
59          close(fd3);
60          close(fd2);
61          close(fd1);
62          return 0;
```

```
63 }
```

2)获取/设置文件描述符标志

截止目前唯一有效的文件描述符标志就是 FD_CLOEXEC：

0 - 在原进程中打开文件描述符，到了通过 exec 函数创建的新进程中依然保持打开状态，可以继续使用这些文件描述符访问对应的文件。如果不做特殊设置的话，所有打开的文件默认的文件描述符标志，此位都为 0。

1 - 在原进程中打开文件描述符，到了通过 exec 函数创建的新进程中会被关闭，不能再继续使用这些文件描述符访问对应的文件。如果希望达到此效果，就需要通过 fcntl 函数人为地将此标志位设置为 1。

```
                vfork                    exec
bash 进程---------------->子进程---------------->新进程 \
  文件描述符表                文件描述符表<---------------+
    0 -> 终端键盘               0 -> 终端键盘 \
    1 -> 终端屏幕               1 -> 终端屏幕 |- FD_CLOEXEC=0(默认)
    2 -> 终端屏幕               2 -> 终端屏幕 /
                exec
原进程----------------->新进程
  文件描述符表                文件描述符表
    0 -> 终端键盘               0 -> NULL       \
    1 -> 终端屏幕               1 -> NULL       |- FD_CLOEXEC=1
    2 -> 终端屏幕               2 -> NULL       /
```

fcntl(fd, F_GETFD) -> 当前文件描述符标志

fcntl(fd, F_SETFD, 欲设文件描述符标志);

代码：fd.c

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <fcntl.h>
 4 int main(void) {
 5         int fd = open("fd.txt", O_RDONLY | O_CREAT | O_TRUNC, 0644);
 6         if (fd == -1) {
 7                 perror("open");
 8                 return -1;
 9         }
10         int flags = fcntl(fd, F_GETFD);
11         if (flags == -1) {
12                 perror("fcntl");
13                 return -1;
14         }
15         printf("flags: %08x\n", flags);
16 printf("文件描述符标志%s 包含 FD_CLOEXEC 位。\n",flags & FD_CLOEXEC ? "" : "不");
17         if (fcntl(fd, F_SETFD, flags | FD_CLOEXEC) == -1) {
18                 perror("fcntl");
19                 return -1;
20         }
21         if ((flags = fcntl(fd, F_GETFD)) == -1) {
```

```
22                      perror("fcntl");
23                      return -1;
24              }
25              printf("flags: %08x\n", flags);
26 printf("文件描述符标志%s 包含 FD_CLOEXEC 位。\n",flags & FD_CLOEXEC ? "" : "不");
27              close(fd);
28              return 0;
29 }
```

3)获取/追加文件状态标志

fcntl(fd, F_GETFL) -> 当前文件状态标志

与创建文件有关的三个标志无法获取：O_CREAT、O_EXCL、O_TRUNC。

O_RDONLY 的值为 0，不能用位与检测。

fcntl(fd, F_SETFL, 欲加文件状态标志);

只有 O_APPEND 和 O_NONBLOCK 两个标志可被追加。

代码：fl.c

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <fcntl.h>
 4 void pflags(int flags) {
 5              printf("文件状态标志(%08x)：", flags);
 6              struct {
 7                      int flag;
 8                      const char* desc;
 9              }       flist[] = {
10                      {O_RDONLY,     "O_RDONLY"},
11                      {O_WRONLY,     "O_WRONLY"},
12                      {O_RDWR,       "O_RDWR"},
13                      {O_APPEND,     "O_APPEND"},
14                      {O_CREAT,      "O_CREAT"},
15                      {O_EXCL,       "O_EXCL"},
16                      {O_TRUNC,      "O_TRUNC"},
17                      {O_NOCTTY,     "O_NOCTTY"},
18                      {O_NONBLOCK, "O_NONBLOCK"},
19                      {O_SYNC,       "O_SYNC"},
20                      {O_DSYNC,      "O_DSYNC"},
21                      {O_RSYNC,      "O_RSYNC"},
22                      {O_ASYNC,      "O_ASYNC"}};
23              for (int i = 0; i < sizeof(flist) / sizeof(flist[0]); ++i)
24                      if (flist[i].flag == O_RDONLY) {
25                              if ((flags & O_ACCMODE) == flist[i].flag)
26                                      printf("%s ", flist[i].desc);
27                      }
28                      else if (flags & flist[i].flag)
29                              printf("%s ", flist[i].desc);
30              printf("\n");
```

```
31 }
32 int main(void) {
33             int fd = open("fl.txt", O_RDONLY | O_CREAT | O_TRUNC | O_ASYNC, 0644);
34             if (fd == -1) {
35                         perror("open");
36                         return -1;
37             }
38             int flags = fcntl(fd, F_GETFL);
39             if (flags == -1) {
40                         perror("fcntl");
41                         return -1;
42             }
43             pflags(flags);
44             if (fcntl(fd, F_SETFL, O_RDWR | O_APPEND | O_NONBLOCK) == -1) {
45                         perror("fcntl");
46                         return -1;
47             }
48             if ((flags = fcntl(fd, F_GETFL)) == -1) {
49                         perror("fcntl");
50                         return -1;
51             }
52             pflags(flags);
53             close(fd);
54             return 0;
55 }
```

9.文件锁

1)读写冲突

A.如果两个或两个以上的进程同时向一个文件的某个特定区域写入数据，那么最后写入文件的数据极有可能因为写操作的交错而产生混乱。

进程 1：写"hello" \ hwoelrlldo

进程 2：写"world" /

B.如果一个进程写而其它进程同时在读一个文件的某个特定区域，那么读出的数据极有可能因为读写操作的交错而不完整。

进程 1：写"hello"

进程 2：读"he"

C.多个进程同时读一个文件的某个特定区域，不会有任何问题，它们只是各自把文件中的数据拷贝到各自的缓冲区中，并不会改变文件的内容，相互之间也就不会冲突。

D.由以上 A.B.C.可以得出结论，为了避免在读写同一个文件的同一个区域时发生冲突，进程之间应该遵循以下规则：

	期望访问	读取	写入
文件的某个区域正在被访问	无人访问	允许	允许
	多人在读	允许	禁止
	一人在写	禁止	禁止

读取：共享性操作。
写入：独占性操作。
代码：write.c、read.c
2)读锁和写锁
写锁：独占锁，任何时候都只能有一个进程加写锁成功，即持有该写锁，而其它进程只能在加锁中阻塞，即等待该进程将写锁解除。
读锁：共享锁，对一个文件的特定区域可以同时加多把读锁。

```
        / 为某文件的欲写区域加写锁
写进程 | 向某文件的欲写区域写数据
        \ 解除写数据之前所加的写锁
        / 为某文件的欲读区域加读锁
读进程 | 从某文件的欲读区域读数据
        \ 解除读数据之前所加的读锁
```

假设有进程 1 和进程 2 试图访问同一个文件的同一个区域：
A.进程 1 正在写，进程 2 也想写

进程 1	进程 2
打开文件准备写入	打开文件准备写入
对欲写区域加写锁	
向欲写区域写数据	对欲写区域加写锁，失败阻塞
解除写前所加写锁	从阻塞中恢复运行，加锁成功
	向欲写区域写数据
	解除写前所加写锁
关闭文件	关闭文件

B.进程 1 正在写，进程 2 却想读

进程 1	进程 2
打开文件准备写入	打开文件准备读取
对欲写区域加写锁	
向欲写区域写数据	对欲读区域加读锁，失败阻塞
解除写前所加写锁	从阻塞中恢复运行，加锁成功
	从欲读区域读数据
	解除读前所加读锁
关闭文件	关闭文件

C.进程 1 正在读，进程 2 却想写

进程 1	进程 2
打开文件准备读取	打开文件准备写入
对欲读区域加读锁	
从欲读区域读数据	对欲写区域加写锁，失败阻塞
解除读前所加读锁	从阻塞中恢复运行，加锁成功
	向欲写区域写数据
	解除写前所加写锁
关闭文件	关闭文件

D.进程 1 正在读，进程 2 也想读

进程 1	进程 2

```
打开文件准备读取      打开文件准备读取
对欲读区域加读锁
从欲读区域读数据      对欲读区域加读锁
解除读前所加读锁      从欲读区域读数据
                      解除读前所加读锁
关闭文件              关闭文件
---------+----------+------+-----
         | 期望加锁 | 读锁 | 写锁
---------+----------+------+-----
文件的某 | 无任何锁 | 成功 | 成功
个区域正 | 多把读锁 | 成功 | 失败
在被锁定 | 一把写锁 | 失败 | 失败
---------+----------+------+-----
```

劝谏锁，也叫协议锁，指使用锁的过程必须遵从某种协议，否则没有锁机制的效果。

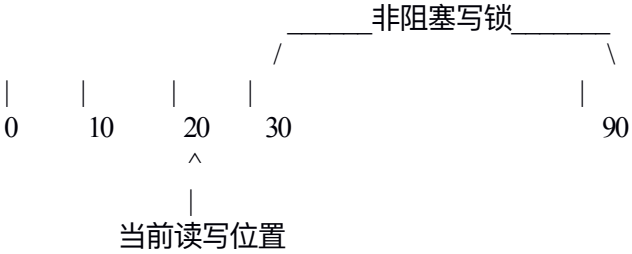
3)fcntl(函数)和 flock(结构)

```
int fcntl(int fd, F_SETLK/F_SETLKW, struct flock* lock);
fd: 文件描述符
cmd: 阻塞加锁    - F_SETLKW，锁不上就不返回
     非阻塞加锁  - F_SETLK，  锁不上返回-1，errno 为 EAGAIN 或者 EACCES
     解锁        - F_SETLK


struct flock {
    short int l_type;    // F_RDLCK   - 加读锁
                         // F_WRLCK   - 加写锁
                         // F_UNLCK   - 解除锁
    short int l_whence; // SEEK_SET  - 相对于文件头
                         // SEEK_CUR  - 相对于当前位置
                         // SEEK_END  - 相对于文件尾
    off_t     l_start;  // 锁区起点
    off_t     l_len;    // 锁区长度
    pid_t     l_pid;    // 加锁进程的 PID，-1 表示自动设置
};
```

例：对相对于文件头 10 字节开始的 20 字节以阻塞模式加读锁。

```
        __阻塞读锁__
       /            \
|      |            |                          |
0      10           30                         90
struct flock lock;
lock.l_type   = F_RDLCK;  // 想干什么?
lock.l_whence = SEEK_SET; // 被锁定区域
lock.l_start  = 10;       // 的起点字节
lock.l_len    = 20;       // 被锁定区域的字节数
lock.l_pid    = -1;       // 谁加的锁?
fcntl(fd, F_SETLKW, &lock);
                 |
        阻塞还是非阻塞?
```

例：对相对于当前位置 10 字节开始到文件尾以非阻塞模式加写锁。




```
struct flock lock;
lock.l_type    = F_WRLCK;
lock.l_whence  = SEEK_CUR;
lock.l_start   = 10;
lock.l_len     = 0; // 一直锁到文件尾
lock.l_pid     = -1;
fcntl(fd, F_SETLK, &lock);
```

例：对整个文件做解锁。

```
struct flock lock;
lock.l_type    = F_UNLCK;
lock.l_whence  = SEEK_SET;
lock.l_start   = 0;
lock.l_len     = 0;
lock.l_pid     = -1;
fcntl(fd, F_SETLK, &lock);
```

代码：write2.c

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <unistd.h>
 4 #include <fcntl.h>
 5 // 加写锁
 6 int wlock(int fd) {
 7         struct flock lock;
 8         lock.l_type    = F_WRLCK;
 9         lock.l_whence  = SEEK_SET;
10         lock.l_start   = 0;
11         lock.l_len     = 0;
12         lock.l_pid     = -1;
13         return fcntl(fd, F_SETLKW, &lock);
14 }
15 // 解除锁
16 int ulock(int fd) {
17         struct flock lock;
18         lock.l_type    = F_UNLCK;
19         lock.l_whence  = SEEK_SET;
20         lock.l_start   = 0;
21         lock.l_len     = 0;
22         lock.l_pid     = -1;
```

```
23              return fcntl(fd, F_SETLK, &lock);
24  }
25  int main(void) {
26              int fd = open("shared.txt", O_WRONLY | O_CREAT | O_APPEND, 0644);
27              if (fd == -1) {
28                      perror("open");
29                      return -1;
30              }
31              if (wlock(fd) == -1) {
32                      perror("wlock");
33                      return -1;
34              }
35              const char* text = "Hello, World!";
36              for (int i = 0; i < strlen(text); ++i) {
37                      if (write(fd, &text[i], sizeof(text[i])) == -1) {
38                              perror("write");
39                              return -1;
40                      }
41                      sleep(1);
42              }
43              if (ulock(fd) == -1) {
44                      perror("ulock");
45                      return -1;
46              }
47              close(fd);
48              return 0;
49  }
```

read2.c

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <unistd.h>
4  #include <fcntl.h>
5  // 加读锁
6  int rlock(int fd) {
7              struct flock lock;
8              lock.l_type    = F_RDLCK;
9              lock.l_whence = SEEK_SET;
10             lock.l_start   = 0;
11             lock.l_len     = 0;
12             lock.l_pid     = -1;
13             return fcntl(fd, F_SETLKW, &lock);
14 }
15 // 解除锁
16 int ulock(int fd) {
17             struct flock lock;
18             lock.l_type    = F_UNLCK;
```

```
19              lock.l_whence = SEEK_SET;
20              lock.l_start  = 0;
21              lock.l_len    = 0;
22              lock.l_pid    = -1;
23              return fcntl(fd, F_SETLK, &lock);
24  }
25  int main(void) {
26              int fd = open("shared.txt", O_RDONLY);
27              if (fd == -1) {
28                      perror("open");
29                      return -1;
30              }
31              if (rlock(fd) == -1) {
32                      perror("rlock");
33                      return -1;
34              }
35              char buf[/*1*/1024];
36              ssize_t readed;
37              while ((readed = read(fd, buf, sizeof(buf))) > 0) {
38                      write(STDOUT_FILENO, buf, readed);
39                      //sleep(1);
40              }
41              printf("\n");
42              if (readed == -1) {
43                      perror("read");
44                      return -1;
45              }
46              if (ulock(fd) == -1) {
47                      perror("ulock");
48                      return -1;
49              }
50              close(fd);
51              return 0;
52  }
```

非阻塞模式就是当因锁区冲突而无法获得欲加之锁时，fcntl 函数立即返回-1，同时置 errno 为 EACCES 或 EAGAIN。

代码：write3.c

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <unistd.h>
4  #include <fcntl.h>
5  #include <errno.h>
6  // 加写锁
7  int wlock(int fd) {
8              struct flock lock;
```

```
 9          lock.l_type    = F_WRLCK;
10          lock.l_whence = SEEK_SET;
11          lock.l_start   = 0;
12          lock.l_len     = 0;
13          lock.l_pid     = -1;
14          return fcntl(fd, F_SETLK, &lock);
15 }
16 // 解除锁
17 int ulock(int fd) {
18          struct flock lock;
19          lock.l_type    = F_UNLCK;
20          lock.l_whence = SEEK_SET;
21          lock.l_start   = 0;
22          lock.l_len     = 0;
23          lock.l_pid     = -1;
24          return fcntl(fd, F_SETLK, &lock);
25 }
26 int main(void) {
27          int fd = open("shared.txt", O_WRONLY | O_CREAT | O_APPEND, 0644);
28          if (fd == -1) {
29                    perror("open");
30                    return -1;
31          }
32          while (wlock(fd) == -1) {
33                    if (errno != EACCES && errno != EAGAIN) {
34                              perror("wlock");
35                              return -1;
36                    }
37                    printf("该文件暂时被锁定，稍后再试...\n");
38          }
39          const char* text = "Hello, World!";
40          for (int i = 0; i < strlen(text); ++i) {
41                    if (write(fd, &text[i], sizeof(text[i])) == -1) {
42                              perror("write");
43                              return -1;
44                    }
45                    sleep(1);
46          }
47          if (ulock(fd) == -1) {
48                    perror("ulock");
49                    return -1;
50          }
51          close(fd);
52          return 0;
53 }
```

read3.c

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
// 加读锁
int rlock(int fd) {
        struct flock lock;
        lock.l_type    = F_RDLCK;
        lock.l_whence = SEEK_SET;
        lock.l_start   = 0;
        lock.l_len     = 0;
        lock.l_pid     = -1;
        return fcntl(fd, F_SETLK, &lock);
}
// 解除锁
int ulock(int fd) {
        struct flock lock;
        lock.l_type    = F_UNLCK;
        lock.l_whence = SEEK_SET;
        lock.l_start   = 0;
        lock.l_len     = 0;
        lock.l_pid     = -1;
        return fcntl(fd, F_SETLK, &lock);
}
int main(void) {
        int fd = open("shared.txt", O_RDONLY);
        if (fd == -1) {
                perror("open");
                return -1;
        }
        while (rlock(fd) == -1) {
                if (errno != EACCES && errno != EAGAIN) {
                        perror("rlock");
                        return -1;
                }
                printf("该文件暂时被锁定，稍后再试...\n");
        }
        char buf[/*1*/1024];
        ssize_t readed;
        while ((readed = read(fd, buf, sizeof(buf))) > 0) {
                write(STDOUT_FILENO, buf, readed);
                //sleep(1);
        }
        printf("\n");
        if (readed == -1) {
                perror("read");
```

```
 48                      return -1;
 49              }
 50              if (ulock(fd) == -1) {
 51                      perror("ulock");
 52                      return -1;
 53              }
 54              close(fd);
 55              return 0;
 56 }
```

4)文件锁的内核结构

```
v 节点
   i 节点
   锁表指针->锁节点                    +->锁节点                    +->
                锁类型                /       锁类型                /
                锁区起点             /        锁区起点             /
                锁区大小            /         锁区大小            /
                加锁进程           /          加锁进程           /
                下一个锁节点-+               下一个锁节点-+
```

10.文件元数据

1)文件元数据存放在该文件的 i 节点中

```
i 节点
   元数据(ls -lh)
   数据块索引表
```

2)从文件的 i 节点中提取元数据

```
#include <sys/stat.h>
int stat (const char* path, struct stat* buf);
int fstat(int fd, struct stat* buf);
int lstat(const char* path, struct stat* buf);不跟踪符号链接(link)
成功返回 0，失败返回-1。
                    目录文件          i 节点映射表
path: 文件路径(-------->i 节点号---------->i 节点->元数据)
                    文件描述符表              v 节点指针
fd: 文件描述符(------------>文件表项--------->v 节点
                                                        |
                                                        v
                                                元数据<---i 节点信息
buf: 输出文件的元数据
a.txt       +------
              v       |
ln -s   a.txt b.txt-------->s:symbol(符号)
          a.txt->[文件]
            ^ b.txt 是 a.txt
            | 的符号链接
          b.txt->[a.txt]
      ____| |____
      |           |
      v           v
```

stat/fstat　lstat
　　|跟踪符　　|不跟踪符
　　v 号链接　　v 号链接
　a.txt 的　　b.txt 的
　元数据　　元数据

```
struct  stat  {
　　dev_t　　st_dev;　　//  设备 ID
　　ino_t　　st_ino;　　//  i 节点号　　　　*
　　mode_t　 st_mode;　 //  文件类型和权限　 *
　　nlink_t  st_nlink;  //  硬链接数　　　　*
　　uid_t　　st_uid;　　//  拥有者用户 ID　　*
　　gid_t　　st_gid;　　//  拥有者组 ID　　 *
　　dev_t　　st_rdev;　 //  特殊设备 ID
　　off_t　　st_size;　 //  总字节数　　　　*
　　time_t　 st_atime;  //  最后访问时间
　　time_t　 st_mtime;  //  最后修改时间　　*　　modification
　　time_t　 st_ctime;  //  最后状态改变时间
　　...
};
```

3)文件元数据中用于表示文件类型和权限的字段 st_mode

```
struct  stat  {
　　...
　　mode_t　 st_mode;　 //  文件类型和权限
　　...
};
```

typedef unsigned int mode_t; // 32 位无符号整型

st_mode 是一个 32 位无符号整数，其中只有低 16 位有效

```
15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0
\_________/ \_____/   \_____/   \_____/   \_____/   \_____/
      |          |         |         |         |         |
 文件类型      设置位     拥有者用  拥有者组  其它用户
              _____|____   户的权限   的权限     的权限
            /      |     \
         设置    设置    粘滞
        用户 ID  组 ID
```

A.文件类型

```
15  14  13  12
 1   0   0   0 -  普通文件(-)
 0   1   0   0 -  目录文件(d)
 1   1   0   0 -  套接字文件(s)
 0   0   1   0 -  字符设备文件(c)
 0   1   1   0 -  块设备文件(b)
 1   0   1   0 -  符号链接文件(l)
 0   0   0   1 -  管道文件(p)
```

B.拥有者用户、拥有者组和其它用户的权限

```
 8   7   6 -  拥有者用户
 5   4   3 -  拥有者组
```

```
 2   1   0 - 其它用户
 |   |   |
读 写 执
权 权 行
限 限 权
       限
```

例：拥有者用户可读可写可执行，拥有者组可读可执行，其它用户可执行。

```
 8   7   6   5   4   3   2   1   0
 1   1   1   1   0   1   0   0   1
 \_____/   \_____/   \_____/
    |         |         |
    7         5         1
    \_____________________/
              |
            0751
```

C.设置位

```
    11          10          9
     |          |          |
  设置        设置        粘滞
 用户 ID      组 ID
(SetUID)  (SetGID)
```

a)带有设置用户 ID 位的可执行文件

系统中每个进程其实都有两个用户 ID，一个叫实际用户 ID，另一个叫有效用户 ID。实际用户 ID 取自登录用户的用户 ID(用 who 命令，id 命令)。一般情况下，有效用户 ID 就取自实际用户 ID。但是，如果一个可执行文件带有设置用户 ID 位，那么运行该可执行文件所得到的进程，其有效用户 ID 不取自实际用户 ID，而取自该可执行文件的拥有者用户 ID。一个进程的权限，即能访问哪些资源，由其有效用户 ID 而非实际用户 ID 决定。

存储用户登录口令的数据文件：

```
ls  -l/etc/passwd
-rw-r--r--  1  root  root  ...  /etc/passwd
```

只用超级用户才能写此文件，而其它任何用户都只能读此文件。

即便是个普通用户，也能修改自己的口令，而所谓修改口令，就是将新口令存入/etc/passwd 文件，以取代旧口令。普通用户为什么也能写只有超级用户才能写的/etc/passwd 文件?

修改口令(passwd)需要运行/usr/bin/passwd 程序：

```
-rwsr-xr-x  1  root  root  ...  /usr/bin/passwd
    |
    +--s=拥有者可执行(x)+设置用户 ID 位
```

这个文件就是一个带有设置用户 ID 位的可执行文件，即使是普通用户运行该程序所得到的进程，其有效用户 ID 也会是超级用户，因此也就可以写只有超级用户才能写的/etc/passwd 文件，将新口令存入其中————有限提权。

b)带有设置组 ID 位的可执行文件

系统中每个进程其实都有两个组 ID，一个叫实际组 ID，另一个叫有效组 ID。实际组 ID 取自登录用户的组 ID。一般情况下，有效组 ID 就取自实际组 ID。但是，如果一个可执行文件带有设置组 ID 位，那么运行该可执行文件所得到的进程，其有效组 ID 不取自实际组 ID，而取自该可执行文件的拥有者组 ID。一个进程的权限，即能访问哪些资源，由其有效组 ID 而非实际组 ID 决定。

c)带有设置用户 ID 位的不可执行文件：没有意义。

d)带有设置组 ID 位的不可执行文件:某些操作系统故意将这种无意义的组合作为强制锁的标志。
e)带有设置用户 ID 位的目录：没有意义。
f)带有设置组 ID 位的目录：在该目录下创建文件及子目录，这些文件或子目录的拥有者组取自其父目录的拥有者组，而非创建它们的用户所隶属的组。
g)带有粘滞位的可执行文件
在其首次运行并结束后，其代码区被连续地保存在磁盘交换区中，而一般磁盘文件的数据块通常是离散存放的。因此，下次运行该程序可以获得较快的载入速度。
h)带有粘滞位的不可执行文件：没有意义。
i)带有粘滞位的目录
如果一个目录带有粘滞位，任何普通用户在该目录下，都只能删除或更名那些属于他自己的文件或子目录，而对于其它用户的文件或子目录，既不能删除也不能更名。超级用户不受限制。
如：/tmp 目录。

```
drwxrwxrwt    l4  root  root         ...  tmp
              |
              +--t=其它用户可执行(x)+粘滞位
-------------+------------+--------------+----------
    设置位      | 可执行文件 | 不可执行文件 |    目录
-------------+------------+--------------+----------
设置用户 ID 位|   有限提权   |    没有意义    | 没有意义
 设置组 ID 位 |   有限提权   |     强制锁     | 继承父组
    粘滞位     |   加速载入   |    没有意义    | 更删自己
-------------+------------+--------------+----------
```

D.文件类型和权限字符串
用 10 个字符表示文件元数据中的 16 个二进制位 st_mode

```
T  UUU  GGG  OOO
|  |||  |||  |||                                    粘滞位
|  |||  |||  ||+--  其它用户是否可执行：x/-  ------>  t/T
|  |||  |||  |+--  其它用户是否可写：w/-
|  |||  |||  +--  其它用户是否可读：r/-
|  |||  |||                                  设置组 ID 位
|  |||  ||+--  拥有者组是否可执行：x/-  ---------->  s/S
|  |||  |+--  拥有者组是否可写：w/-
|  |||  +--  拥有者组是否可读：r/-
|  |||                                  设置用户 ID 位
|  ||+--  拥有者用户是否可执行：x/-  ------------>  s/S
|  |+--  拥有者用户是否可写：w/-
|  +--  拥有者用户是否可读：r/-
|
+--  文件类型：-/d/s/c/b/l/p
```

代码：stat.c

```
  1  #include  <stdio.h>
  2  #include  <string.h>
  3  #include  <sys/stat.h>
  4  #include  <time.h>
  5  const  char*  mtos(mode_t  m)  {
  6              static  char  s[11];
```

```
 7          mode_t type = m & S_IFMT;
 8          if (type == S_IFDIR)
 9                  strcpy(s, "d");
10          else if (type == S_IFLNK)
11                  strcpy(s, "l");
12          else if (type == S_IFBLK)
13                  strcpy(s, "b");
14          else if (type == S_IFCHR)
15                  strcpy(s, "c");
16          else if (type == S_IFSOCK)
17                  strcpy(s, "s");
18          else if (type == S_IFIFO)
19                  strcpy(s, "p");
20          else
21                  strcpy(s, "-");
22          strcat(s, m & S_IRUSR ? "r" : "-");
23          strcat(s, m & S_IWUSR ? "w" : "-");
24          strcat(s, m & S_IXUSR ? "x" : "-");
25          strcat(s, m & S_IRGRP ? "r" : "-");
26          strcat(s, m & S_IWGRP ? "w" : "-");
27          strcat(s, m & S_IXGRP ? "x" : "-");
28          strcat(s, m & S_IROTH ? "r" : "-");
29          strcat(s, m & S_IWOTH ? "w" : "-");
30          strcat(s, m & S_IXOTH ? "x" : "-");
31          if (m & S_ISUID)
32                  s[3] = (s[3] == 'x' ? 's' : 'S');
33          if (m & S_ISGID)
34                  s[6] = (s[6] == 'x' ? 's' : 'S');
35          if (m & S_ISVTX)
36                  s[9] = (s[9] == 'x' ? 't' : 'T');
37          return s;
38  }
39  const char* ttos(time_t t) {
40          static char s[20];
41          struct tm* local = localtime(&t);
42          sprintf(s, "%04d-%02d-%02d %02d:%02d:%02d",
43                  local->tm_year + 1900, local->tm_mon + 1, local->tm_mday,
44                  local->tm_hour, local->tm_min, local->tm_sec);
45          return s;
46  }
47  int main(int argc, char* argv[]) {
48          if (argc < 2)
49                  goto usage;
50          struct stat st;
51          if (argc < 3) {
52                  // 以跟踪符号链接的方式获取文件的元数据
53                  if (stat(argv[1], &st) == -1) {
```

```
54                                  perror("stat");
55                                  return -1;
56                          }
57                  }
58                  else if (!strcmp(argv[2], "-l")) {
59                          // 以不跟踪符号链接的方式获取文件的元数据
60                          if (lstat(argv[1], &st) == -1) {
61                                  perror("lstat");
62                                  return -1;
63                          }
64                  }
65                  else
66                          goto usage;
67                  printf("          设备 ID：%lu\n", st.st_dev);
68                  printf("         i 节点号：%ld\n", st.st_ino);
69                  printf("文件类型及权限：%#o (%s)\n", st.st_mode, mtos(st.st_mode));
70                  printf("        硬链接数：%lu\n", st.st_nlink);
71                  printf("  拥有者用户 ID：%u\n",   st.st_uid);
72                  printf("    拥有者组 ID：%u\n",   st.st_gid);
73                  printf("    特殊设备 ID：%lu\n", st.st_rdev);
74                  printf("       总字节数：%ld\n", st.st_size);
75                  printf("  最后修改时间：%ld (%s)\n", st.st_mtime, ttos(st.st_mtime))      ;
76                  return 0;
77  usage:
78                  fprintf(stderr, "用法：%s <文件> [-l]\n", argv[0]);
79                  return -1;
80  }
```

11.访问测试

```
    运行        有效用户/组 ID        判断
程序---->进程---------->文件---->是否可读/写/执行?
                                  | stat/lstat
                                  v
                                元数据
                                /     \
                          拥有者用   权限
                          户/组 ID
```

```
#include <unistd.h>
int access(const char* pathname, int mode);
```

成功返回 0，失败返回-1。

pathname: 文件路径

mode: 访问模式，可取以下值：

- R_OK - 是否可读?
- W_OK - 是否可写?
- X_OK - 是否可执行?
- F_OK - 文件是否存在?

access 函数的访问测试是基于调用进程的实际用户 ID 和实际组 ID 进行的。因此，如果调用进

程所对应的可执行文件带有设置用户 ID 位或者设置组 ID 位，access 函数返回无权访问，但是实际可能有权访问。(access 只检测有效 id)

代码：access.c

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 int main(int argc, char* argv[]) {
 4         if (argc < 2) {
 5                 fprintf(stderr, "用法：%s <文件>\n", argv[0]);
 6                 return -1;
 7         }
 8         printf("文件\"%s\"", argv[1]);
 9         if (access(argv[1], F_OK) == -1)
10                 printf("不存在(%m)。\n");
11         else {
12                 if (access(argv[1], R_OK) == -1)
13                         printf("不可读(%m)，");
14                 else
15                         printf("可读，");
16                 if (access(argv[1], W_OK) == -1)
17                         printf("不可写(%m)，");
18                 else
19                         printf("可写，");
20                 if (access(argv[1], X_OK) == -1)
21                         printf("不可执行(%m)。\n");
22                 else
23                         printf("可执行。\n");
24         }
25         return 0;
26 }
```

```
tarena@ubuntu:~/Desktop/uc/day07$ ls -l
总用量 24
-rwxrwxr-x 1 tarena tarena 7480 Apr  3 22:18 access
-rw-rw-r-- 1 tarena tarena  673 Apr  3 22:18 access.c
-rwxrwxr-x 1 tarena tarena 7768 Apr  3 20:42 stat
-rw-rw-r-- 1 tarena tarena 1986 Apr  3 22:20 stat.c
lrwxrwxrwx 1 tarena tarena    6 Apr  3 20:11 stat.d -> stat.c
tarena@ubuntu:~/Desktop/uc/day07$ sudo chown root:root access
[sudo] tarena 的密码：
tarena@ubuntu:~/Desktop/uc/day07$ ls -l
总用量 24
-rwxrwxr-x 1 root   root   7480 Apr  3 22:18 access
-rw-rw-r-- 1 tarena tarena  673 Apr  3 22:18 access.c
-rwxrwxr-x 1 tarena tarena 7768 Apr  3 20:42 stat
-rw-rw-r-- 1 tarena tarena 1986 Apr  3 22:20 stat.c
lrwxrwxrwx 1 tarena tarena    6 Apr  3 20:11 stat.d -> stat.c
```

```
tarena@ubuntu:~/Desktop/uc/day07$ sudo chmod u+s access
tarena@ubuntu:~/Desktop/uc/day07$ ls -l access
-rwsrwxr-x 1 root root 7480 Apr  3 22:18 access
tarena@ubuntu:~/Desktop/uc/day07$ who
tarena@ubuntu:~/Desktop/uc/day07$ sudo chmod g+s access
tarena@ubuntu:~/Desktop/uc/day07$ ls -l access
-rwsrwsr-x 1 root root 7480 Apr  3 22:18 access
```

12.权限掩码(umask 命令：查看当前掩码，设置掩码用 umask 022)
每个进程都有自己的权限掩码，用于从进程为创建文件所指定的权限中，屏蔽某些权限。
进程表项(内核中)
  环境变量表
  文件描述符表
  权限掩码
  ...
指定权限：0777
权限掩码：0002 (屏蔽其它用户的写权限)
实际权限：0775

指定权限：0777
权限掩码：0022 (屏蔽同组和其它用户的写权限)
实际权限：0755
通过 Shell 命令 umask 修改 Shall 进程的权限掩码,在这个 Shell 下启动的进程都会继承 Shell(父)进程的权限掩码。用 touch 创建文件，缺省为 0644。
通过系统调用直接改变调用进程自己的权限掩码。

```
#include <sys/stat.h>
mode_t umask(mode_t cmask);
```

永远成功，返回原来的权限掩码。
cmask: 新的权限掩码
实际权限=指定权限&~权限掩码

```
 7   7   7  \
/ \/ \/ \ | -   指定权限 --
111111111 /                  \
 0   2   2  \                   \ &  -
/ \/ \/ \ | -   权限掩码      /    /
000010010 /                  /    /
111101101 ---  ~权限掩码---     /
 7   5   5  \                   /
/ \/ \/ \ | -   实际权限 <---
111101101 /
```

代码：umask.c

```
  1 #include <stdio.h>
  2 #include <unistd.h>
  3 #include <fcntl.h>
```

```
 4 #include <sys/stat.h>
 5 int main(void) {
 6         mode_t old = umask(0);//                umask(0)什么也不屏蔽
 7         int fd = open("new.txt", O_RDWR | O_CREAT | O_TRUNC, 0777);
 8         if (fd == -1) {
 9                 perror("open");
10                 return -1;
11         }
12         close(fd);
13         umask(old);
14         if ((fd = open("old.txt", O_RDWR | O_CREAT | O_TRUNC, 0777)) == -1)     {
15                 perror("open");
16                 return -1;
17         }
18         close(fd);
19         return 0;
```

13.修改权限

```
#include <sys/stat.h>
int chmod(const char* path, mode_t mode);
int fchmod(int fd, mode_t mode);
```

成功返回 0，失败返回-1。

path: 文件路径

mode: 文件权限

fd: 文件描述符

代码：chmod.c

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <sys/stat.h>
 4 int main(void) {
 5         /*   S    U    G    O
 6               7    6    5    4
 7             421  420  401  400
 8                  rw-  r-x  r--
 9                  rwS  r-s  r-T
10         */
11         if (chmod("chmod.txt", 07654) == -1) {
12                 perror("chmod");
13                 return -1;
14         }
15         return 0;
16 }
```

调用进程的有效用户 ID 必须与文件的拥有者用户 ID 一致,或者调用进程的有效用户 ID 为超级用户，才能修改该文件的权限，且受权限掩码的影响。

14.修改文件拥有者用户和(或)组

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

```
int chown(const char* path, uid_t owner, gid_t group);     \
                                                          |-跟踪符号链接
int fchown(int fd, uid_t owner, gid_t group);              _/
int lchown(const char* path, uid_t owner, gid_t group); // 不跟踪符号链接
```
成功返回 0，失败返回-1。
path: 文件路径
owner: 拥有者用户 ID，-1 表示不修改,一般 0 代表 root
group: 拥有者组 ID，-1 表示不修改,一般 0 代表 root
fd: 文件描述符
```
sudo chown root:root access
```
代码：chown.c
```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 int main(void) {
 4 //         if (chown("chown.txt", 0, -1) == -1) {
 5 //         if (chown("chown.txt", -1, 0) == -1) {
 6 //         if (chown("chown.txt", 1000, -1) == -1) {
 7            if (chown("chown.txt", -1, 1000) == -1) {
 8                    perror("chown");
 9                    return -1;
10            }
11            return 0;
12 }
```

如果调用进程的有效用户 ID 为超级用户，则它可以任意修改任何文件的拥有者用户和组，否则它只能把自己名下文件的拥有者组改为自己隶属的某个组。
15.修改文件大小
```
#include <unistd.h>
int truncate(const char* path, off_t length);
int ftruncate(int fd, off_t length);
```
成功返回 0，失败返回-1。
path: 文件路径
length: 目标大小
fd: 文件描述符
无论截短还是增长都是在文件的尾端变化，对于增长而言，新增内容用 0 字节填充。

代码：trunc.c
```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 int main(void) {
 4            if (truncate("trunc.txt", 10) == -1) {
 5                    perror("truncate");
 6                    return -1;
 7            }
 8            return 0;
 9 }
```

16.内存映射文件

在虚拟内存和磁盘文件之间建立映射关系，这样一来就可以按照访问内存的方式操作磁盘文件，而不必使用专门的文件 I/O 接口。同时也可将其视为一种在不同进程之间交换数据的方法。

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

成功返回映射区(虚拟)内存起始地址，失败返回 MAP_FAILED(-1)。

start: 映射区(虚拟)内存起始地址，NULL 系统自动选定后返回

length: 映射区的字节数，自动按页(4096)圆整

```
int((length+4096)/4096)*4096
```

prot: 访问权限，可取以下值：

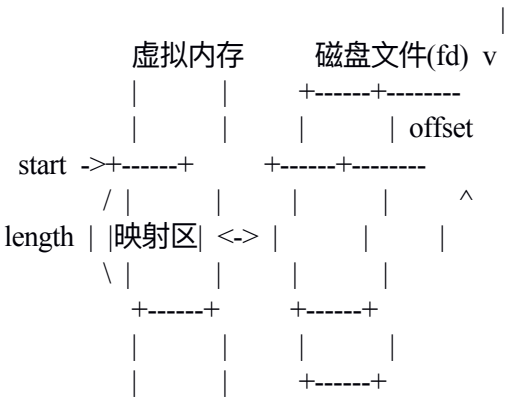
```
PROT_READ  - 映射区可读
PROT_WRITE - 映射区可写
PROT_EXEC  - 映射区可执行
PROT_NONE  - 映射区不可访问
```

flags: 映射标志

```
MAP_ANONYMOUS|MAP_PRIVATE - 虚拟内存到物理内存的映射
MAP_SHARED - 虚拟内存到磁盘文件的映射
```

fd: 被映射文件的文件描述符

offset: 被映射文件中映射区的起始位置，即相对于文件头的偏移字节数



解除虚拟内存到物理内存或磁盘文件的映射。

```
int munmap(void* start, size_t length);
```

成功返回 0，失败返回-1。

start: 映射区(虚拟)内存起始地址，必须是内存页边界

length: 映射区的字节数，自动按页(4096)圆整

```
int((length+4096)/4096)*4096
```

代码：mmap1.c

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <unistd.h>
4 #include <fcntl.h>
5 #include <sys/stat.h>
6 #include <sys/mman.h>
7 int main(void) {
```

```
 8          int fd = open("mmap.txt", O_RDWR | O_CREAT | O_TRUNC, 0644);
 9          if (fd == -1) {
10                  perror("open");
11                  return -1;
12          }
13          const char* text = "Hello, World!";
14          size_t size = strlen(text) * sizeof(text[0]);
15          if (ftruncate(fd, size) == -1) {
16                  perror("ftruncate");
17                  return -1;
18          }
19          void* map = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED,
fd,     0);
20          if (map == MAP_FAILED) {
21                  perror("mmap");
22                  return -1;
23          }
24          memcpy(map, text, size); // 貌似写内存，实则写文件
25          char buf[1024] = {};
26          memcpy(buf, map, size);  // 貌似读内存，实则读文件
27          puts(buf);
28          munmap(map, size);
29          close(fd);
30          return 0;
31  }
```

mmap2.c

```
 1  #include <stdio.h>
 2  #include <string.h>
 3  #include <unistd.h>
 4  #include <fcntl.h>
 5  #include <sys/stat.h>
 6  #include <sys/mman.h>
 7  int main(void) {
 8          int fd = open("mmap.txt", O_RDONLY);
 9          if (fd == -1) {
10                  perror("open");
11                  return -1;
12          }
13          struct stat st;
14          if (fstat(fd, &st) == -1) {
15                  perror("fstat");
16                  return -1;
17          }
18          void* map = mmap(NULL, st.st_size, PROT_READ, MAP_SHARED, fd, 0);
19          if (map == MAP_FAILED) {
20                  perror("mmap");
```

```
21                      return -1;
22              }
23              char buf[1024] = {};
24              memcpy(buf, map, st.st_size); // 貌似读内存，实则读文件
25              printf("%s\n", buf);
26              munmap(map, st.st_size);
27              close(fd);
28              return 0;
29  }
```

```
进程 1 -> 虚拟内存 <-> 用户空间
                       -------- 内存壁垒
                       用户空间 <-> 虚拟内存 -> 进程 2
                    +- 读写磁盘 -+
                    |            |
进程 1 -> 虚拟内存 <-> 磁盘文件 <-> 虚拟内存 -> 进程 2
            \_____________________/
                       |
                形式上的进程间通信
进程 1 -> 虚拟内存 <-> 内核空间 <-> 虚拟内存 -> 进程 2
                       |
              真正意义上的进程间通信
```

17.硬链接

1)硬链接的本质就是目录文件里一个文件名和 i 节点号的对应条目。

2)根据一个已有的硬链接创建一个新的硬链接：

```
          ~~~~~~~~~~~~~          ~~~~~~~~~~~~
                |                      |
             目录 1:                目录 2:
      文件名 1——i 节点号    文件名 2——i 节点号
                |                      ^
                |______________________|
目录 1/文件名 1 \
                 > 同一个文件
目录 2/文件名 2 /
```

#include <unistd.h>

int link(const char* oldpath, const char* newpath);

成功返回 0，失败返回-1。

oldpath: 原路径，即已有的硬链接。

newpath: 新路径，即新的硬链接。

注意：

若 newpath 已存在，则不会创建新硬链接。

oldpath 必须已经存在，且不能为目录。

newpath 中不能包含不存在的目录。

代码：link.c

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

```
 3 int main(int argc, char* argv[]) {
 4         if (argc < 3) {
 5                 fprintf(stderr, "用法：%s <原路径> <新路径>\n", argv[0]);
 6                 return -1;
 7         }
 8         if (link(argv[1], argv[2]) == -1) {
 9                 perror("link");
10                 return -1;
11         }
12         return 0;
13 }
```

3)删除一个已有的硬链接

目录：

```
xxxx   1010
xxxxx  1098 <- 删除
xx     2310
xxxx   1098
...
```

```
#include <unistd.h>
int unlink(const char* pathname);
```

成功返回 0，失败返回-1。

pathname: 硬链接路径

注意：

pathname 只能是文件的路径，而不能是目录的路径，即 unlink 函数只能删除文件的硬链接，不能删除目录的硬链接。

Unix/Linux 系统没有直接删除文件的方法，所谓删除文件其实就是删除针对该文件的硬链接，指向该文件的所有硬链接都被删除了，文件自然也就被删除了，即释放改文件所占用的磁盘空间。

一个文件可以同时拥有多个硬链接，通过 unlink 函数删除其中的一个硬链接并不会导致该文件被删除(释放磁盘上的 i 节点和数据块)，因为必须保证其它引用该文件的硬链接继续有效，但该文件的硬链接数(作为文件的元数据保存在其 i 节点中)会被减 1。如果删除的是该文件的最后一个硬链接，其硬链接数将被减到 0，这表示系统中已经没有任何硬链接引用该文件，直到此刻，该文件才会真正被删除。即便删除的是该文件的最后一个硬链接，但如果此时该文件正被某进程打开，其磁盘上的数据也不会立即被删除，直到所有打开它的进程都显式或隐式地关闭了该文件，其在磁盘上的数据才会被删除。

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

成功返回 0，失败返回-1。

pathname: 硬链接路径

remove 函数和 unlink 函数的功能几乎完全一样，唯一的不同在于 remove 函数不但可以删除引用文件的硬链接，还可以删除引用目录的硬链接，但被删除的目录必须为空，即该目录中不能包含任何文件或子目录。

代码：unlink.c

4)修改硬链接中的路径信息(目录名和文件名)

修改(移动)->目录：

```
        文件名——i 节点号
          ^
```

|
修改(更名)
#include <stdio.h>
int rename(const char* oldpath, const char* newpath);
成功返回 0，失败返回-1。
oldpath - 原路径
newpath - 新路径
注意：
若 newpath 已存在，则被改为引用 oldpath 的目标。
若 oldpath 和 newpath 本来引用就是同一个目标，什么也不做。
若 oldpath 是目录，则 newpath 或者不存在或者为空目录。
代码：rename.c
18.软链接
1)软链接，亦称符号链接，其本质就是一个保存着另一个文件或目录的路径的文件。所有针对软链接文件的访问，最后都会被定向到该软链接所引用的目标文件或目录中————跟踪软链接。
a.txt: Hello, World!
b.txt -> a.txt
c.txt -> b.txt
d.txt -> c.txt
cat d.txt
Hello, World!
2)创建软链接，即创建一个文件，在该文件中保存一个路径字符串
#include <unistd.h>
int symlink(const char* oldpath, const char* newpath);
成功返回 0，失败返回-1。
oldpath: 原路径，即软链接的目标路径。
newpath: 新路径，即软链接文件本身的路径。
注意：
symlink 函数实际上就是创建一个路径为 newpath 的软链接文件，将 oldpath 字符串保存到该文件中。
若 newpath 已存在，则不会创建软链接。
oldpath 可以是目录，甚至可以不存在。
newpath 中不能包含不存在的目录。
3)读取软链接文件的内容，即目标路径
#include <unistd.h>
ssize_t readlink(const char* path, char* buf,
size_t size);
成功返回实际拷贝到 buf 缓冲区中的字节数，失败返回-1。
path: 软链接文件的路径
buf: 输出软链接文件内容的缓冲区
size: 软链接文件内容缓冲区字节数
buf->|/home/tarena/lesson.txtxxxxxxx|
_____23_____/ /
_____30_____/
30->size
返回 23
该函数不负责追加字符串结尾空字符。

不能通过 open 加 read 函数读取软链接文件本身的内容，因为 open 一定会跟踪软链接，所以读到的其实是该软链接的目标文件的内容。

代码：slink.c

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 int main(int argc, char* argv[]) {
 4         if (argc < 3) {
 5                 fprintf(stderr, "用法：%s <原路径> <新路径>\n", argv[0]);
 6                 return -1;
 7         }
 8         if (symlink(argv[1], argv[2]) == -1) {
 9                 perror("symlink");
10                 return -1;
11         }
12         char buf[1024] = {};
13         if (readlink(argv[2], buf, sizeof(buf) - sizeof(buf[0])) == -1) {
14                 perror("readlink");
15                 return -1;
16         }
17         printf("%s -> %s\n", argv[2], buf);
18         return 0;
19 }
```

19.目录(文件)

1)创建一个空目录

```
#include <sys/stat.h>
int mkdir(const char* pathname, mode_t mode);
```

成功返回 0，失败返回-1。

pathname: 目录路径

mode: 访问权限，目录的可执行权限(x)表示可进入

2)删除一个空目录

```
#include <unistd.h>
int rmdir(const char* pathname);
```

成功返回 0，失败返回-1。

pathname: 目录路径

pathname 参数只能是目录不能是文件，且该目录必须空

remove=unlink+rmdir

```
d1/              8
|
+--d2/          4
|   |
|   +--d4/     2
|   |   |
|   |   +--fb  1
|   |
|   +--fc      3
```

```
|
+--d3/          6
|   |
|   +--fd       5
|
+--fa           7
```

rm -r <目录>

递归

代码：dir.c

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <sys/stat.h>
 4 int main(void) {
 5         /*
 6         if (mkdir("d3", 0755) == -1) {
 7                 perror("mkdir");
 8                 return -1;
 9         }
10         */
11         if (rmdir("d3") == -1) {
12                 perror("rmdir");
13                 return -1;
14         }
15         return 0;
16 }
```

3)获取当前工作目录

进程表项

- 环境变量
- 文件描述符表
- 权限掩码
- 当前工作目录

当前工作目录主要影响进程中关于相对路径的处理。

```
#include <unistd.h>
char* getcwd(char* buf, size_t size);
```

成功返回 buf，失败返回 NULL。

buf: 输出当前工作目录的缓冲区

size: 当前工作目录缓冲区字节数

该函数会将当前工作目录字符串拷贝到 buf 所指向的缓冲区中并追加结尾空字符。

4)切换当前工作目录

```
#include <unistd.h>
int chdir(const char* path);
```

成功返回 0，失败返回-1。

path: 目标工作目录

该函数只切换调用进程的当前工作目录，对其父进程和兄弟进程没有任何影响。

代码：cwd.c

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 int main(void) {
 4         char buf[1024];
 5         if (!getcwd(buf, sizeof(buf))) {
 6                 perror("getcwd");
 7                 return -1;
 8         }
 9         printf("当前工作目录：%s\n", buf);
10         if (chdir("..") == -1) {
11                 perror("chdir");
12                 return -1;
13         }
14         if (!getcwd(buf, sizeof(buf))) {
15                 perror("getcwd");
16                 return -1;
17         }
18         printf("当前工作目录：%s\n", buf);
19         return 0;
20 }
```

5)获取目录内容

#include <dirent.h>

A.打开目录流

DIR* opendir(const char* name);

成功返回目录流指针，失败返回 NULL。

name: 目录路径

B.关闭目录流

int closedir(DIR* dirp);

成功返回 0，失败返回-1。

dirp: 目录流指针

C.读取目录流

struct dirent* readdir(DIR* dirp);

成功返回目录流指针，失败或读完返回 NULL。

目录有若干条目组成，每个条目均包含文件名、文件类型、i 节点号等信息。readdir 函数可以连续调用，调用一次返回一个条目，再调用一次返回下一个条目，直到返回 NULL，表示已读完整个目录或者发生错误，为了区分这两种情况，可以通过检查 errno 是否被重置来判断。

```
struct dirent {
    ino_t          d_ino;       // i 节点号
    off_t          d_off;       // 下一条目录的位置
    unsigned short d_reclen;    // 每条记录的长度
    unsigned char  d_type;      // 文件类型
    char           d_name[];    // 文件名
};
```

tree .

代码：list.c

```
#include <stdio.h>
#include <dirent.h>
#include <errno.h>
int main(int argc, char* argv[]) {
        if (argc < 2) {
                fprintf(stderr, "用法：%s <目录>\n", argv[0]);
                return -1;
        }
        DIR* dp = opendir(argv[1]);
        if (!dp) {
                perror("opendir");
                return -1;
        }
        errno = 0;
        struct dirent* de;
        for (de = readdir(dp); de; de = readdir(dp)) {
                switch(de->d_type) {
                        case DT_DIR:
                                printf("      目录：");
                                break;
                        case DT_REG:
                                printf("普通文件：");
                                break;
                        case DT_LNK:
                                printf("   软链接：");
                                break;
                        case DT_BLK:
                                printf("   块设备：");
                                break;
                        case DT_CHR:
                                printf("字符设备：");
                                break;
                        case DT_SOCK:
                                printf("   套接字：");
                                break;
                        case DT_FIFO:
                                printf("      管道：");
                                break;
                        default:
                                printf("      未知：");
                                break;
                }
                printf("%lu, %s\n", de->d_ino, de->d_name);
        }
        if (errno) {
                perror("readdir");
                return -1;
```

```
48              }
49              closedir(dp);
50              return  0;
51  }
```

八、进程

1.进程的基本概念

1)进程与程序

程序是被存储在磁盘上，包括机器指令和数据的文件。进程是程序被装载到内存中，其中的指令可被处理器执行，以处理其中的数据。一个程序可被同时运行为多个进程。系统中的每个进程都有其特定的任务。

程序————文件，以 ELF 格式组织的可执行代码和数据

进程————内存，以进程映像方式组织的可执行代码和数据

线程————过程，处理器执行指令处理数据的流程

2)进程的分类

A.交互式进程：由 Shell 启动，既可在前台运行，也可在后台运行，通过终端接收用户的输入，并为用户提供输出。如：vi、ps 等。

B.批处理进程：与终端没有联系，以进程序列的方式，在无需人工干预的条件下，自动完成一组批量任务。如：各种 Shell 脚本程序。

C.守护进程：又名精灵进程，系统的后台服务。独立于控制终端，周期性地执行某种任务或等待某些事件。在系统引导时自动启动，在系统被关闭时自动终止，生命周期很长。如：crond、lpd 等。

2.进程快照

使用 ps 命令捕获进程运行过程中的一个瞬间状态。实时进程列表查看用：top

1)简单形式：ps

以简略的方式显式当前用户拥有控制终端的进程信息。

PID - 进程标识

TTY - 控制终端次设备号

TIME - 进程运行时间

CMD - 进程启动命令

2)BSD 风格的常用选项

a - 显式所有用户拥有控制终端的进程信息

x - 也包括没有控制终端的进程

u - 以详尽方式显式

w - 以更大列宽显式

3)SVR4 风格的常用选项

-e - 显式所有用户的进程信息 -> BSD:a+x

-f - 按完整格式显式 -> BSD:u

-F - 按更完整格式显式 -> BSD:u

-l - 按长格式显式 -> BSD:w

4)进程信息列表

USER/UID: 进程实际用户 ID

PID: 进程标识

PPID: 父进程标识

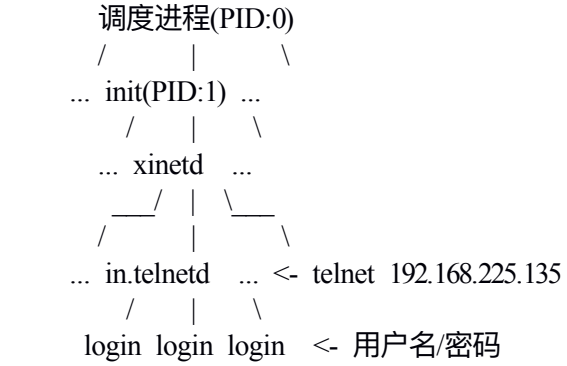
%CPU/C: CPU 使用率

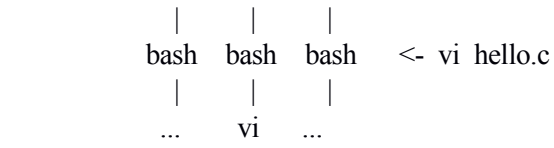
%MEM: 内存使用率

VSZ/SZ: 占用虚拟内存大小(KB)

RSS: 占用半导体内存大小(KB)

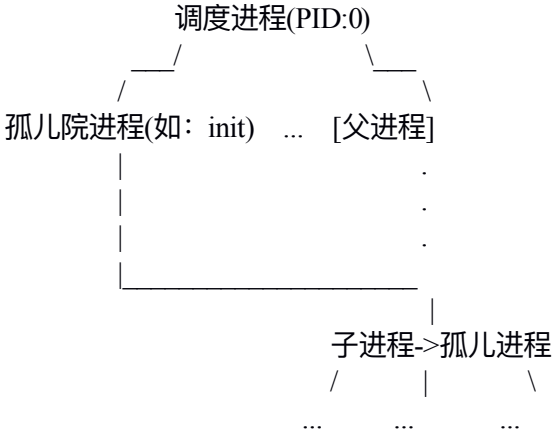
TTY: 终端次设备号
ttyn - 实终端，物理终端；(带有硬件)
pts/n - 尾终端，虚拟终端；
? - 无控制终端，如后台进程。
STAT/S: 进程状态
R - 正在被处理器执行
S - 可唤醒的睡眠
系统中断 \
获得资源 > 唤醒处于睡眠状态的进程
收到信号 /
D - 除了 wake_up 系统调用以外都无法唤醒的进程
T - 收到 SIGSTOP(19)信号，转入暂停状态
收到 SIGCONT(18)信号，唤醒进程继续运行
W - 等待内存分页(2.6 以后的内核已被废弃)
Z - 僵尸进程
< - 高优先级进程
N - 低优先级进程
L - 有被锁到内存中的分页
s - 会话首进程
l - 多线程化
+ - 在前台进程组中
START/STIME: 进程开始时间
TIME：进程运行时间
COMMAND/CMD: 进程启动命令
F: 进程标志
1 - fork 了子进程但是没有 exec 新进程；
4 - 拥有超级用户特权。
NI: 进程 nice 值，介于-20 到 19，越小优先级越高。
PRI: 进程静态优先级=80+nice，介于 60 到 99。负值表示实时进程。
ADDR: 内核进程的内存地址，普通进程显示"-"。
SZ: 占用虚拟内存页数。
WCHAN：进程正在等待的内核函数或事件。
PSR: 进程当前被指派给哪个处理器运行。
3.父进程、子进程、孤儿进程和僵尸进程
1)Unix/Linux 系统中的进程存在父子关系。一个父进程可以创建多个子进程，但每个子进程只能有一个父进程。整个系统中只有一个根进程，即 PID 为 0 的调度进程。系统中的所有进程构成了一棵以调度进程为根的进程树。





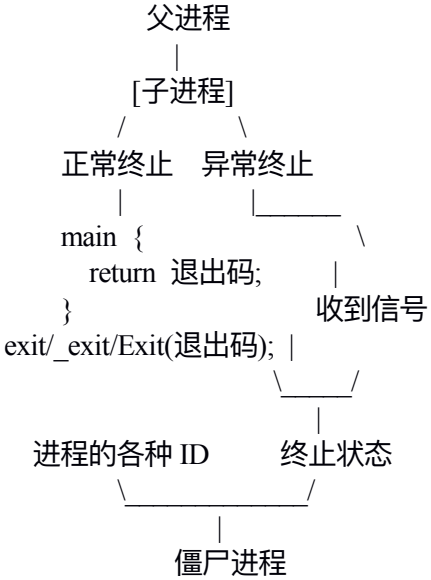
2)孤儿进程

父进程创建子进程以后，子进程在操作系统的调度下与其父进程同时运行。如果父进程先于子进程终止，子进程即成为孤儿进程，同时被系统中的孤儿院进程收养，即成为孤儿院进程的子进程，因此系统中的进程树结构不会因为任何一个进程的终止而被破坏。



3)僵尸进程

如果子进程先于父进程终止，但父进程由于某种原因，没有回收子进程的终止状态，子进程的僵尸就会一直存在于系统之中，形成僵尸进程。



僵尸进程虽然已经不再活动，但其中包含了已终止进程的各种 ID 和终止状态等残余信息，也会占用系统资源，至少进程的 PID 无法被重用，直到被其父进程回收才得以释放。如果父进程直到终止都未回收已成为僵尸进程的子进程，孤儿院进程会立即回收这些处于僵尸状态的子进程。因此一个进程不可能同时既是孤儿进程又是僵尸进程。一个进程成为僵尸进程需要引起注意，如果它的父进程长期运行而不终止，僵尸进程所占用的资源将长期得不到释放，如果数量足够多的话，将严重影响系统的运行性能。

4.进程的各种 ID

进程表项
　　环境变量
　　文件描述符表
　　权限掩码
　　当前工作目录
　　各种 ID
PID: 进程 ID，进程在整个操作系统中的唯一标识符，延迟重用算法
0
1
2
3 - 终止并被回收
4
5 - 终止并被回收
6
7 - 不会重用 3
8
9 - 不会重用 5
.
.
.
99
3
5
PPID: 父进程 ID
UID: 实际用户 ID
EUID: 有效用户 ID
GID: 实际组 ID
EGID: 有效组 ID

```
#include <unistd.h>
pid_t getpid  (void); // 获取进程 ID
pid_t getppid (void); // 获取父进程 ID
uid_t getuid  (void); // 获取实际用户 ID
uid_t geteuid (void); // 获取有效用户 ID
uid_t getgid  (void); // 获取实际组 ID
uid_t getegid (void); // 获取有效组 ID
```

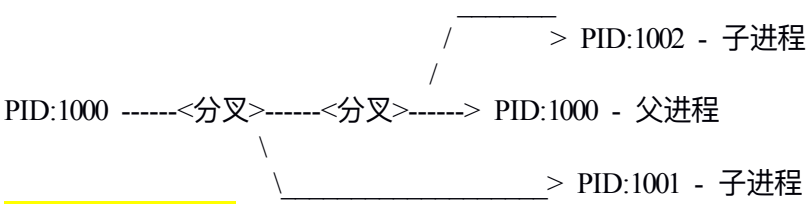
跟在终端输入 id 命令一样

代码：id.c

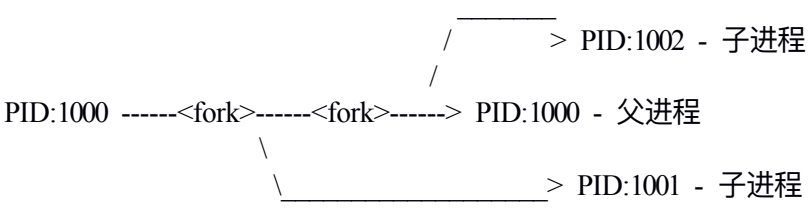
```
1 #include <stdio.h>
2 #include <unistd.h>
3 int main(void) {
4             printf("      进程 ID: %d\n", getpid());
5             printf("  父进程 ID: %d\n", getppid());
6             printf("实际用户 ID: %d\n", getuid());
7             printf("  实际组 ID: %d\n", getgid());
8             printf("有效用户 ID: %d\n", geteuid());
9             printf("  有效组 ID: %d\n", getegid());
```

```
 10              return 0;
 11 }
```

5.子进程
1)创建子进程(进程分叉)




```
#include <unistd.h>
pid_t fork(void);
```
成功分别在父子进程中返回子进程的 PID 和 0，失败返回-1。



一次调用两次返回：在准父进程中调用一次，在父进程和子进程中各返回一次，在父进程中的返回值是一个大于 0 的整数，即子进程的 PID，而在子进程中的返回值是 0。

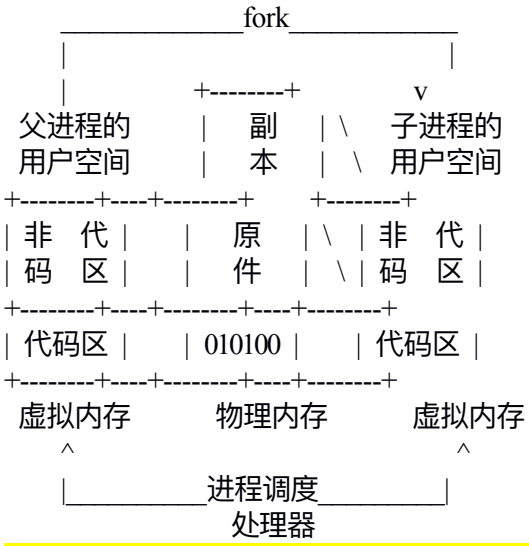
```
pid_t pid = fork(); // 创建子进程
if (pid == -1) {
    perror("fork");
    exit(-1);
}
if (pid == 0) { // 在子进程中 fork 函数返回 0
    // 完成子进程的任务
    exit(0); // 子进程终止
}
// 完成父进程的任务
exit(0); // 父进程终止
```

代码：fork.c

```
  1 #include <stdio.h>
  2 #include <unistd.h>
  3 int main(void) {
  4         printf("%d 进程：我要调用 fork 了...\n", getpid());
  5         pid_t pid = fork();
  6         if (pid == -1) {
  7                 perror("fork");
  8                 return -1;
  9         }
 10         if (pid == 0) {
 11                 printf("%d 进程：我是%d 进程的子进程。\n", getpid(), getppid()      );
 12                 return 0;
```

```
 13              }
 14              printf("%d 进程：我是%d 进程的父进程。\n", getpid(), pid);
 15              sleep(1);
 16              return 0;
 17 }
```

2)子进程是父进程的不完全副本



子进程的数据区、BSS 区、堆栈区(包括 I/O 流缓冲区)、甚至命令行参数和环境变量都从父进程的物理内存空间中复制一份并重建自己的虚拟内存映射，唯有代码区与父进程共享，即映射相同的物理内存。

```
if (pid == 0) {
    ... \
    ... | - CPU2 --+
    ... /          |
}                   \ CPU
else {              / 时间片
    ... \          |
    ... | - CPU1 --+
    ... /
}
```

代码：copy.c

```
  1 #include <stdio.h>
  2 #include <stdlib.h>
  3 #include <unistd.h>
  4 int a = 100; // 数据区
  5 int main(void) {
  6              int b = 200; // 栈区
  7              int* c = malloc(sizeof(int));
  8              *c = 300; // 堆区
  9              printf("父进程：%d %d %d\n", a, b, *c); // 100 200 300
 10              pid_t pid = fork();
```

```
11              if (pid == -1) {
12                      perror("fork");
13                      return -1;
14              }
15              if (pid == 0) {
16                      printf("子进程：%d %d %d\n", ++a, ++b, ++*c); // 101 201 301
17                      free(c);
18                      return 0;
19              }
20              sleep(1);
21              printf("父进程：%d %d %d\n", a, b, *c); // 100 200 300
22              free(c);
23              return 0;
24 }
```

ostream.c

```
1 #include <stdio.h>
  2 #include <unistd.h>
  3 int main(void) {
  4               printf("ABC"); // ABC -> 输出流缓冲区：|ABC|
  5               pid_t pid = fork();
  6               if (pid == -1) {
  7                       perror("fork");
  8                       return -1;
  9               }
 10               if (pid == 0) {
 11                       printf("DEF\n"); // |ABCDEF\n| -> 屏幕
 12                       return 0;
 13               }
 14               sleep(1);
 15               printf("\n"); |ABC\n| -> 屏幕
 16               return 0;
 17 }
```

istream.c

```
  1 #include <stdio.h>
  2 #include <unistd.h>
  3 int main(void) {
  4               printf("父进程：");
  5               int a, b, c;
  6               scanf("%d%d%d", &a, &b, &c); // 1 2 3 4 5 6 | 1->a 2->b 3->c | 4 5 6
  7               pid_t pid = fork();
  8               if (pid == -1) {
  9                       perror("fork");
 10                       return -1;
 11               }
 12               if (pid == 0) {
```

```
13                      a = b = c = 0;
14                      scanf("%d%d%d", &a, &b, &c); // 4 5 6 | 4->a 5->b 6->c | 空
15                      printf("子进程：%d %d %d\n", a, b, c); // 4 5 6
16                      return 0;
17              }
18              sleep(1);
19              printf("父进程：%d %d %d\n", a, b, c); // 1 2 3
20              return 0;
21  }
```

3)自由并发

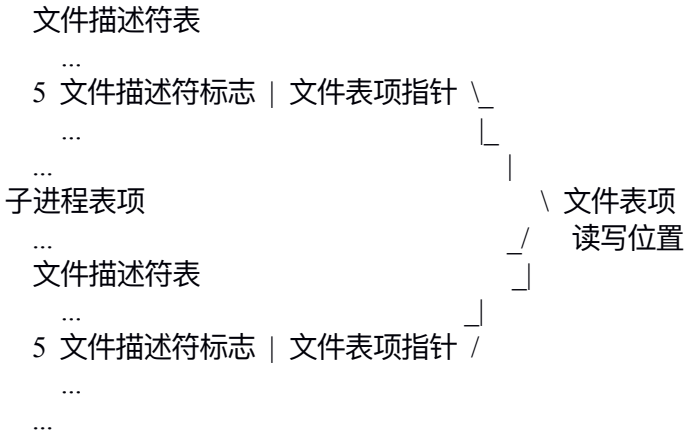
fork 函数成功返回以后，父子进程各自独立地运行，其被调度的先后顺序完全不确定，某些实现可以保证子进程先被调度。

代码：conc.c

```
 1  #include <stdio.h>
 2  #include <unistd.h>
 3  int main(void) {
 4              setbuf(stdout, NULL);
 5              pid_t pid = fork();
 6              if (pid == -1) {
 7                      perror("fork");
 8                      return -1;
 9              }
10              if (pid == 0) {
11                      for (int i = 0; i < 1000; ++i)
12                              printf("+");
13                      return 0;
14              }
15              for (int i = 0; i < 1000; ++i)
16                      printf("-");
17              return 0;
18  }
```

4)内核空间中与进程有关的信息也会在产生子进程的过程中被复制

```
---------------------- 内核空间 ----------------------
...         父进程表项              ...         子进程表项              ...
            环境变量          -->               环境变量
            文件描述符表      -->               文件描述符表
            权限掩码          -->               权限掩码
            当前工作目录      -->               当前工作目录
            PID               -|                PID
            PPID               ->               PPID
            UID/GID           -->               UID/GID
            EUID/EGID         -->               EUID/EGID
父进程表项
  ...
```



fork 函数成功返回以后，系统内核为父进程维护的文件描述符表也被复制到子进程的进程表项中，但文件表项并不复制，而是在父子进程间共享。

代码：ftab.c

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <fcntl.h>
 4 #include <unistd.h>
 5 int main(void) {
 6         int fd = open("ftab.txt", O_RDWR | O_CREAT | O_TRUNC, 0644);
 7         if (fd == -1) {
 8                 perror("open");
 9                 return -1;
10         }
11         const char* text = "Hello, World!";
12         if (write(fd, text, strlen(text) * sizeof(text[0])) == -1) {
13                 perror("write");
14                 return -1;
15         }
16         pid_t pid = fork();
17         if (pid == -1) {
18                 perror("fork");
19                 return -1;
20         }
21         if (pid == 0) {
22                 if (lseek(fd, -6, SEEK_CUR) == -1) {
23                         perror("lseek");
24                         return -1;
25                 }
26                 close(fd);
27                 return 0;
28         }
29         sleep(1);
30         text = "Linux";
31         if (write(fd, text, strlen(text) * sizeof(text[0])) == -1) {
```

```
 32                         perror("write");
 33                         return -1;
 34             }
 35             close(fd);
 36             return 0;
 37 }
```

5)fork 函数在以下两种情况下会返回失败(-1)：
A.系统总线程数达到上限：/proc/sys/kernel/threads-max
B.用户总进程数达到上限：ulimit -u
6)通过 fork 函数创建子进程的场合
任何时候只要希望得到一个与当前进程并发运行且执行同一份代码的操作过程，都可以使用 fork 来创建子进程。

```
for(;;) {
    gets(...); // 从键盘读字符串
    send(...); // 将读到的字符串发给服务器
    recv(...); // 从服务器接收字符串
    puts(...); // 将收到的字符串打印到屏幕上
}
```

```
pid_t pid = fork();
...
if (pid == 0) {
    for (;;) {
        recv(...); // 从服务器接收字符串
        puts(...); // 将收到的字符串打印到屏幕上
    }
}
else {
    for (;;) {
        gets(...); // 从键盘读字符串
        send(...); // 将读到的字符串发给服务器
    }
}
```

7) 孤儿进程和僵尸进程

代码：orphan.c

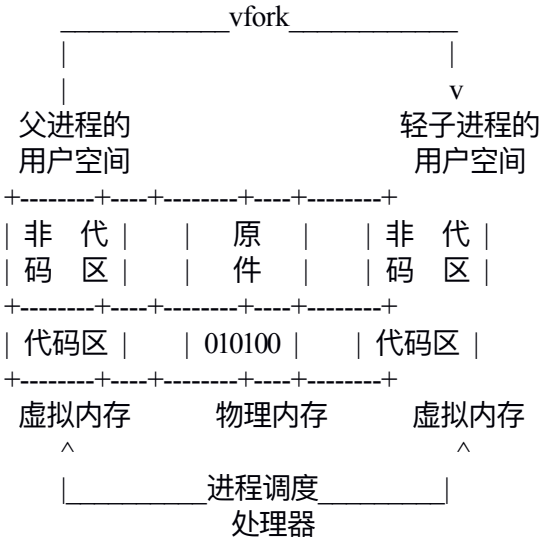
```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 int main(void) {
 4             printf("%d 进程：我是父进程。\n", getpid());
 5             pid_t pid = fork();
 6             if (pid == -1) {
 7                         perror("fork");
 8                         return -1;
 9             }
10             if (pid == 0) {
```

```
11                      printf("%d 进程：我是子进程。\n", getpid());
12                      for (;;);
13                      return 0;
14              }
15              return 0;
16 }
```

zombie.c

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 int main(void) {
 4              printf("%d 进程：我是父进程。\n", getpid());
 5              pid_t pid = fork();
 6              if (pid == -1) {
 7                      perror("fork");
 8                      return -1;
 9              }
10              if (pid == 0) {
11                      printf("%d 进程：我是子进程。\n", getpid());
12                      return 0;
13              }
14              getchar();
15              return 0;
16 }
```

6.轻量级子进程



#include <unistd.h>
pid_t vfork(void);
成功分别在父子进程中返回子进程的 PID 和 0，失败返回-1。
vfork 与 fork 函数的功能基本相同，只有以下两点不同：
1)vfork 函数所创建的子进程不复制父进程用户空间的物理内存，也不拥有自己独立的内存映射表，而是与父进程共享全部的地址空间。

2)vfork 函数所创建的子进程会先于父进程被调度，同时将父进程挂起，直到该子进程终止，或通过 exec 函数启动新进程，父进程才会恢复运行。

注意，通过 fork 函数创建的子进程可以采用三种方法主动终止：
1)在 main 函数中执行 return 语句；
2)调用 C 语言标准库函数 exit；
3)调用 C 语言标准库函数_Exit 或者系统调用函数_exit。
但是，通过 vfork 函数创建的轻子进程主动终止的方法只有一个，那就是调用 C 语言标准库函数_Exit 或者系统调用函数_exit。

代码：vfork.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <unistd.h>
 4 int a = 100; // 数据区
 5 int main(void) {
 6         int b = 200; // 栈区
 7         int* c = malloc(sizeof(int));
 8         *c = 300; // 堆区
 9         printf("父进程：%d %d %d\n", a, b, *c); // 100 200 300
10         pid_t pid = vfork();
11         if (pid == -1) {
12                 perror("fork");
13                 return -1;
14         }
15         if (pid == 0) {
16                 printf("子进程：%d %d %d\n", ++a, ++b, ++*c); // 101 201 301
17                 _exit(0);
18         }
19         printf("父进程：%d %d %d\n", a, b, *c); // 101 201 301
20         free(c);
21         return 0;
22 }
```

```
        / 父子进程共享内存->被线程所取代
vfork <
        \ 快速地创建子进程->被带有写时复制优化特性的 fork 所取代
```

写时复制(CoW，Copy-on-Write)是一种惰性优化的方式，其本质就是在子进程创建伊始，并不复制父进程的物理内存，类似于 vfork 的做法，只复制它的内存映射表即可，父子进程共享同一个地址空间，直到子进程需要写这些数据时，再行内存复制，而且并不会全部复制父进程物理内存，只会复制那些受写操作影响内存页。
目前能见到 vfork 的地方恐怕仅剩和 exec 函数联用的场合。

```
PID:1000 ------<fork>------> PID:1000 - 父进程
                        \
                         \_______> PID:1001 - 子进程
```

原进程 - PID:1000 ------<exec>------> PID:1000 - 新进程
^
|
可执行文件
fork/vfork - 生
exec - 变
PID:1000 ------<vfork>------------------> PID:1000 - 原进程
\
_____<exec>_____> PID:1001 - 新进程
^
|
可执行文件
7.进程终止
1)进程的正常终止
从 main 函数中返回或者调用 exit/_exit/_Exit 函数。
main 函数的返回值和调用 exit/_exit/_Exit 函数时所传入的参数就是进程的退出码，表示进程终止的原因。虽然 main 函数返回值和 exit/_exit/_Exit 函数参数的数据类型都是 int，但作为进程的退出码，只有最低 8 位，即最低字节，可被获取。
关于进程退出码的习惯：
0 - 进程运行正确，任务成功完成，其间没有发生任何错误。
非零 - 进程在运行中发生了某种错误，任务中途取消。一种方式是用不同的正整数表示不同的错误，另一种方式用-1 表示出错，不区分具体何种错误。
main 函数的返回相当于在 main 函数中调用 exit 函数，其结果是令进程终止，而其它函数的返回仅仅表示流程控制回到调用该函数的语句的下一条语句继续执行，不会令进程终止。但在任何函数中调用 exit 函数肯定都会导致进程终止。
#include <stdlib.h>
void exit(int status); \ 标准 C 语言 -- 功能更多
void _Exit(int status); / 的库函数 \
#include <unistd.h> > 功能等价
void _exit(int status); - Unix 系统调用库函数 /
exit 函数的执行过程：
①先调用事先注册的退出处理函数
②冲刷并关闭所有处于打开状态的 I/O 流
③删除所有通过 tmpfile 函数创建临时文件
④调用_exit 函数
关闭所有仍处于打开状态的文件描述符
将所有的子进程托付给孤儿院进程收养
向父进程发送 SIGCHLD(17)信号
终止进程，将退出码交给内核，由内核将其保存在进程僵尸中
注册无参退出处理函数：
#include <stdlib.h>
int atexit(void (*function)(void));
成功返回 0，失败返回非零。
// 无参退出处理函数
void func(void) {
临终善后
}

atexit(func); func -> 被标准 C 库维护的函数指针列表
注册带参退出处理函数：
#include <stdlib.h>
int on_exit(void (*function)(int, void*), void* arg);
成功返回 0，失败返回非零。
// 带参退出处理函数
退出码 vptr
| |
v v
void func(int status, void* arg) {
临终善后
}
on_exit(func, vptr); // func/vptr
// -> 被标准 C 库维护的函数指针列表

代码：exit.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  void doexit1(void) {
5          printf("doexit1 函数被调用。\n");
6  }
7  void doexit2(int status, void* arg) {
8          printf("doexit2 函数被调用，%d，%s。\n", status, (char*)arg);
9  }
10 int foo(void) {
11         printf("foo 函数被调用。\n");
12         /*
13         exit(EXIT_SUCCESS); // #define EXIT_SUCCESS 0
14         exit(EXIT_FAILURE); // #define EXIT_FAILURE 1
15         _exit(5678);
16         */
17         _Exit(5678);
18         return 10;
19 }
20 int main(void) {
21         if (atexit(doexit1)) {
22                 perror("atexit");
23                 return -1;
24         }
25         if (on_exit(doexit2, "快走！")) {
26                 perror("on_exit");
27                 return -1;
28         }
29         printf("foo 函数返回%d。\n", foo());
30         printf("任务圆满完成，返回成功。\n");
31         //return 1234;
```

```
32              exit(1234);
33  }
```

2)进程的异常终止
当进程收到来自系统内核的某些信号时，如果程序中没有对收到这些信号以后的处理做出特别的安排，那么系统就会按照默认的方式来处理这些信号。绝大部分信号的默认处理就是杀死收到信号的进程。这种终止进程的方式就叫做异常终止。
系统内核在什么情况下会给进程发送信号?
A.进程执行某些在操作系统看来是否十分危险的动作
  如：无效内存访问，SIGSEGV(11)，段错误
B.终端用户通过一些键盘触发特殊的信号
  如：Ctrl+C，SIGINT(2)，终端中断符
      Ctrl+\，SIGQUIT(3)，终端退出符
C.执行 kill 命令或者调用 kill 函数向指定进程发送信号
  kill -l // 显式信号列表
  kill -<信号编号> <进程 PID> //没有进程编号默认 15 信号
  int kill(进程 PID, 信号编号);函数
在出现错误的情况下不得不取消进程：
#include <stdlib.h>
void abort(void);
向调用进程发送 SIGABRT(6)信号，该信号的默认处理是杀死进程。如果调用进程事先将 SIGABRT(6)信号的处理方式设置位忽略或捕获，abort 函数会先恢复对 SIGABRT(6)信号的默认处理，然后再补发一次该信号。
8.回收子进程
1)为什么要回收子进程?
A.想知道子进程终止的原因
正常终止还是异常终止？如果是正常终止，任务是成功了还是失败了？如果任务失败了，什么原因导致的失败？如果异常终止，被哪个信号杀了?
B.等待子进程结束。
C.释放僵尸所占用的资源，使 PID 可以被重用，避免僵尸进程过多。
2)等待并回收任意子进程
#include <sys/wait.h>
pid_t wait(int* status);
成功返回所回收子进程的 PID，失败返回-1。
status - 输出所回收子进程的终止状态(正常终止的退出码或异常终止的被杀信号)。置 NULL 表示不输出终止状态。
父进程在创建了若干子进程以后调用 wait 函数：
A.若所有子进程都在运行，则阻塞，直至有子进程终止；
B.若至少有一个子进程已经终止,则立即回收其中一个子进程的僵尸,返回其PID同时通过status参数输出其终止状态；
C.若没有任何活动的或终止的子进程，则返回-1，置 errno 为 ECHILD。
通过宏检查子进程的终止状态：
WIFEXITED(终止状态)
    0-异常终止：WTERMSIG(终止状态)->杀死该子进程的信号编号
    1-正常终止：WEXITSTATUS(终止状态)->子进程的退出码
                                    |
                main 函数的返回值 \ 低 8

调用 exit/_exit/_Exit 函数的参数 / 位
WIFSIGNALED(终止状态) |
0-正常终止：WEXITSTATUS(终止状态)->子进程的退出码
1-异常终止：WTERMSIG(终止状态)->杀死该子进程的信号编号

代码：wait.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <unistd.h>
 4 #include <sys/wait.h>
 5 int main(void) {
 6         pid_t pid = fork();
 7         if (pid == -1) {
 8                 perror("fork");
 9                 return -1;
10         }
11         if (pid == 0) {
12                 int status = 0x12345678; // 低 8 位：0x78
13                 printf("%d 进程：我是子进程。\n", getpid());
14                 //return status;
15                 //exit(status);
16                 //_exit(status);
17                 //_Exit(status);
18                 abort();
19         }
20         printf("%d 进程：我是父进程，我要等待子进程终止...\n", getpid());
21         int status;
22         pid = wait(&status);
23         if (pid == -1) {
24                 perror("wait");
25                 return -1;
26         }
27         printf("%d 进程：回收了%d 进程的僵尸。\n", getpid(), pid);
28         //if (WIFEXITED(status))
29         if (!WIFSIGNALED(status))
30                 printf("%d 进程：该进程正常终止，其退出码是%#x。\n", getpid(),
31                         WEXITSTATUS(status));
32         else
33                 printf("%d 进程：该进程异常终止，杀死它的信号是%d。\n", getpid(),
34                         WTERMSIG(status));
35         return 0;
36 }
```

如果父进程创建了多个子进程，要想将所有的子进程僵尸都回收干净，可以在一个循环中多次调用 wait 函数，直到 wait 函数返回-1 且 errno 为 ECHILD 时退出循环。

代码：loop.c

```
 1 #include <stdio.h>
 2 #include <errno.h>
 3 #include <unistd.h>
 4 #include <sys/wait.h>
 5 int main(void) {
 6         for (int i = 0; i < 3; ++i) {
 7                 pid_t pid = fork();
 8                 if (pid == -1) {
 9                         perror("fork");
10                         return -1;
11                 }
12                 if (pid == 0) {
13                         printf("%d 进程：我是子进程。\n", getpid());
14                         return 0;
15                 }
16         }
17         for (;;) {
18                 printf("%d 进程：我是父进程，我要等待子进程终止...\n", getpid());
19                 pid_t pid = wait(NULL);
20                 if (pid == -1) {
21                         if (errno != ECHILD) {
22                                 perror("wait");
23                                 return -1;
24                         }
25                         printf("%d 进程：子进程的僵尸已被收尽！\n", getpid());
26                         break;
27                 }
28                 printf("%d 进程：回收了%d 进程的僵尸。\n", getpid(), pid);
29         }
30         getchar();
31         return 0;
32 }
```

希望在所有子进程都处于活动状态的情况下，不要阻塞　　___
希望可以等待并回收某个特定子进程，而不是随便收一个 /　　|
3)以阻塞或非阻塞方式等待并回收任意或特定子进程 <-------+

```
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int* status, int options);
                        |                       |
                任意或特定子进程            阻塞或非阻塞
```

成功返回所回收子进程的 PID，失败返回-1。
pid - 正整数表示所要等待并回收的那个特定子进程的 PID，-1 表示等待并回收任意子进程。
status - 输出所回收子进程的终止状态(正常终止的退出码或异常终止的被杀信号)。置 NULL 表示不输出终止状态。
options - 0 表示阻塞模式；WNOHANG 表示非阻塞模式，调用该函数时，如果所有子进程都在运行，则立即返回 0，而不发生阻塞。
waitpid(-1, &status, 0) <==> wait(&status)

代码：waitpid.c

```
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <sys/wait.h>
int main(void) {
        for (int i = 0; i < 3; ++i) {
                pid_t pid = fork();
                if (pid == -1) {
                        perror("fork");
                        return -1;
                }
                if (pid == 0) {
                        printf("%d 进程：我是子进程。\n", getpid());
                        return 0;
                }
        }
        for (;;) {
                printf("%d 进程：我是父进程，我要等待子进程终止...\n", getpid       ());
                pid_t pid = waitpid(-1, NULL, WNOHANG); // 非阻塞版本的 wait
                if (pid == -1) {
                        if (errno != ECHILD) {
                                perror("wait");
                                return -1;
                        }
                        printf("%d 进程：子进程的僵尸已被收尽！\n",
                                getpid());
                        break;
                }
                if (pid)
                        printf("%d 进程：回收了%d 进程的僵尸。\n",
                                getpid(), pid);
                else {
                        printf("%d 进程：暂时没有僵尸需要回收。"
                                "执行空闲处理...\n", getpid());
                        // ...
                }
        }
        getchar();
        return 0;
}
```

9.创建新进程

1)exec 函数族

与 fork/vfork 函数不同，exec 函数不是创建调用进程的子进程，而是创建一个新的进程取代调用进程自身。新进程会用将自己的全部地址空间，覆盖调用进程的地址空间，但进程的 PID 保持不变。

```
    启动------PID:1000------<exec>++++++PID:1000++++++>
     |              |                |                |
可执行文件      原进程          可执行文件        新进程
命令行参数        |_________命令行参数_____^
                                变为

                    $ cmd abc -i 123     Shell 进程的环境变量
                      \_____________/          ____|
                                    |          |
main(int argc, char* argv[], char* envp[]) { ... }
                            / 0 * -> cmd      0 * -> PATH=...
                            | 1 * -> abc      1 * -> USER=...
        字符指针数组 < 2 * -> -i       2 * -> C_PATH=...
                            | 3 * -> 123      3 NULL
                            \ 4 NULL
```

```
int execl(const char* path, const char* arg, ...);
execl("/home/tarena/uc/day10/cmd",
        "cmd", "abc", "-i", "123", NULL);
int execlp(const char* file, const char* arg, ...);
```

如果调用进程的 PATH 环境变量已经包含了可执行文件 cmd 的路径：

```
execlp("cmd", "cmd", "abc", "-i", "123", NULL);
int execle(const char* path, const char* arg, ...,
                const char* envp[]);
                                |
                            环境变量
int execv(const char* path, const char* argv[]);
int execvp(const char* file, const char* argv[]);
int execve(const char* path, const char* argv[],
                const char* envp[]);
```

函数名后缀的意义：

l - 以变长参数列表的形式传递命令行参数

v - 以字符指针数组的形式传递命令行参数

无 p - 需要给出可执行文件的路径

带 p - 只需给出可执行文件的文件名，该函数会在 PATH 环境变量所包含的目录下搜索该文件

无 e - 不指定环境变量，新进程会继承调用进程的环境变量

带 e - 为新进程指定特有环境变量

所有 exec 函数成功不返回，失败返回-1。

代码：cmd.c

```
1 #include <stdio.h>
2 #include <unistd.h>
3 int main(int argc, char* argv[], char* envp[]) {
4         printf("--------------------\n");
5         printf("PID: %d\n", getpid());
6         printf("---- 命令行参数 ----\n");
7         while(argv && *argv)
8                 printf("%s\n", *argv++);
```

```
 9              printf("---- 环境变量表 ----\n");
10              while(envp && *envp)
11                      printf("%s\n", *envp++);
12              printf("--------------------\n");
13              return 13;
14      }
```

exec.c

```
 1      #include <stdio.h>
 2      #include <unistd.h>
 3      int main(void) {
 4              printf("--------------------\n");
 5              printf("PID: %d\n", getpid());
 6              /*
 7              if (execl("./cmd", "cmd", "abc", "-i", "123", NULL) == -1) {
 8                      perror("execl");
 9                      return -1;
10              }
11              */
12              char* argv[] = {"cmd", "abc", "-i", "123", NULL};
13              /*
14              if (execv("./cmd", argv) == -1) {
15                      perror("execl");
16                      return -1;
17              }
18              */
19              char* envp[] = {"NAME=minwei", "COMPANY=tarena", NULL};
20              /*
21              if (execle("./cmd", "cmd", "abc", "-i", "123", NULL, envp) == -1) {
22                      perror("execle");
23                      return -1;
24              }
25              if (execve("./cmd", argv, envp) == -1) {
26                      perror("execve");
27                      return -1;
28              }
29              if (execlp("cmd", "cmd", "abc", "-i", "123", NULL) == -1) {
30                      perror("execlp");
31                      return -1;
32              }
33              */
34              if (execvp("cmd", argv) == -1) {
35                      perror("execvp");
36                      return -1;
37              }
38              printf("原进程终止。\n");
39              return 0;
```

```
40 }
```

2)新进程和原进程之间的差别与继承
通过 exec 函数所创建新进程的一些属性有别于原进程：
A.任何处于阻塞状态的信号都会丢失
B.被设置捕获的信号会还原为默认操作
C.所有关于线程属性的设置会还原为默认值
D.有关进程的统计信息都会复位
E.与进程内存相关的任何数据都会丢失，包括内存映射文件
F.标准库所维护的一切数据结构都会丢失
但也有些属性会被新进程继承下来，比如 PID、PPID、实际用户 ID 和实际组 ID、优先级，以及文件描述符(除非该文件描述符带有 FD_CLOEXEC 标志位)。
3)vfork+exec 模式
调用 exec 函数固然可以创建出新的进程，但是新进程会取代原来的进程。如果既想创建新进程，同时又希望原来的进程继续存在，则可以采用 fork+exec 的模式，即在 fork 产生的子进程里调用 exec 函数，新进程取代了子进程，但父进程依然存在。既然通过 exec 所创建的新进程会将创建者进程(即通过 fork 创建的子进程)的地址空间全部覆盖，那么 fork 为子进程所复制的父进程的用户空间就完全多余了，因此在这种场合使用 vfork 更合适。

代码：ve.c

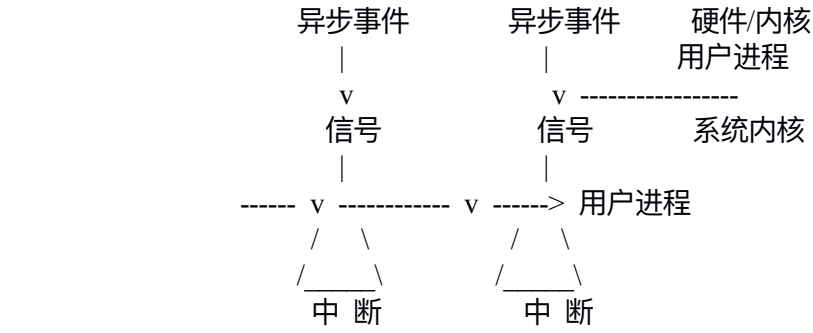
```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <sys/wait.h>
 4 int main(void) {
 5         printf("--------------------\n");
 6         printf("PID: %d\n", getpid());
 7         pid_t pid = vfork();
 8         if (pid == -1) {
 9                 perror("vfork");
10                 return -1;
11         }
12         if (pid == 0) {
13                 char* argv[] = {"cmd", "abc", "-i", "123", NULL};
14                 char* envp[] = {"NAME=minwei", "COMPANY=tarena", NULL};
15                 if (execve("./cmd", argv, envp) == -1) {
16                         perror("execve");
17                         _exit(-1);
18                 }
19         }
20         int status;
21         if ((pid = waitpid(pid, &status, 0)) == -1) {
22                 perror("waitpid");
23                 return -1;
24         }
25         printf("新进程的退出码：%d\n", WEXITSTATUS(status));
26         printf("原进程终止。\n");
27         return 0;
```

```
 28 }
```
4)Shell 工作原理
广义的 Shell，操作系统除了提供实现核心功能的系统内核以外，还提供一套工具环境，以完成系统的日常维护和一些简单的应用。
狭义的 Shell，就是一个程序，不断读取用户输入的命令行，解析命令行字符串。通过 vfork+exec+waitpid 的模式，执行该命令行，并打印出退出码。
Shell 进程
+->显式提示符并等待用户输入 <- 输入命令行字符串：ls -l
| 父进程调用 vfork 创建一个子进程
| 子进程根据用户输入的命令行字符串调用 exec 函数创建出新进程，
| 如：(ls -l)
+--父进程调用 waitpid 等待结束获得子进程的终止状态
5)执行 Shell 命令
```
#include <stdlib.h>
int system(const char* command);
                    ^
                    |
                命令行字符串
```
先调用 vfork 创建子进程
(若出错，返回-1)
  在子进程内部调用 exec 执行命令行 command
  (若失败，_exit(127))
调用 waitpid 等待并回收子进程
(若出错，返回-1，否则返回 waitpid 输出的终止状态)

代码：sys.c
```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <unistd.h>
 4 #include <sys/wait.h>
 5 int main(void) {
 6 //      int status = system("cat sys.c");
 7 //      int status = system("caa sys.c");
 8         int status = system("cat sys.a");
 9         if (status == -1) {
10                 perror("system");
11                 return -1;
12         }
13         printf("退出码：%d\n", WEXITSTATUS(status));
14         return 0;
15 }
```
Shell 终端命令 cal->查看日历

九、信号
1.信号的基本概念
1)信号是提供异步事件处理机制的软件中断。这些异步事件可能来自于硬件设备，如用户同时按了 Ctrl 键和 C 键，也可能来自于系统内核，如试图访问尚未映射到物理内存的虚拟内存，又

或者来自于用户进程，如调用 kill 函数或执行 kill 命令等。



2)用户进程之间也可以用相互发信号的方法实现某种信息交换，因此信号也是一种简单的进程间通信(IPC)手段。

3)信号的异步特性不仅表现为它的产生是异步的，对它的处理同样也是异步的。程序的设计者不可能也不需要精确地预见什么时候触发什么信号，也同样无法预见该信号究竟在什么时候被处理。一切都在系统内核的操控弄下，异步地运行。信号是在软件层面对中断机制的一种模拟。

2.信号处理

信号的生命周期：信号产生于系统内核并在缓存一定时间后被发出，由该信号所针对的进程做出响应————忽略、捕获或默认。

忽略：什么也不做。

SIGKILL(9)和 SIGSTOP(19)信号不能被忽略。

捕获：收到信号的进程先暂停正在执行的代码，跳转到事先注册的信号处理函数，执行之直到从该函数中返回，跳转回捕获信号的地方继续执行。

SIGKILL(9)和 SIGSTOP(19)信号不能被捕获。

默认：未明确被忽略或捕获的信号一律按默认方式处理。

多数信号的默认处理方式就是终止收到信号的进程。

3.信号的名称和编号

系统内核信号是通过正整数形式的编号加以识别的，但人们出于直观性的考虑更倾向于用文本形式的字符串表示信号，这就是信号名。可见信号名与信号编号之间存在严格的一一对应关系。在<signum.h>中已经将每个信号都定义成如下形式的宏：

```
#define  SIGHUP    1
#define  SIGINT    2
#define  SIGQUIT  3
...
```

1~31，31 个不可靠的非实时信号；34~64，31 个可靠的实时信号，共 62 个信号。

4.常用信号

SIGINT(2)　按下终端中断符(Ctrl+C)　终止
SIGQUIT(3)　按下终端退出符(Ctrl+\)　终止+转储
SIGABRT(6)　调用 abort 函数　终止+转储
SIGSEGV(11)　非法内存访问　终止+转储
SIGPIPE(13)　向读端关闭的管道写入　终止
SIGALRM(14)　(真实)定时器到期　终止
SIGCHLD(17)　子进程终止向父进程递送　忽略
SIGCONT(18)　进程由暂停状态恢复运行　忽略
SIGSTOP(19)　暂停进程，不能被忽略和捕获　暂停
SIGTSTP(20)　按下终端停止符(Ctrl+Z)　暂停
SIGIO(29)　异步 I/O 就绪　终止

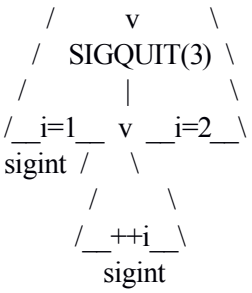
Shell 终端命令;ps a | grep 可执行文件

5.捕获信号

```
void sigxxx(int signum) {
    ...                          <-----------------------------------+
}//自己写的                                                          |
                                                                     |
#include <signal.h>                                                  |
sighandler_t signal (int signum, sighandler_t handler);              |
成功返回原来的信号处理方式，失败返回 SIG_ERR。                          |
signum - 信号编号                                                    |
handler - 当前的处理方式，可取以下值:                                 |
            SIG_IGN - 忽略                                           |
            SIG_DEF - 默认                                           |
            信号处理函数指针 - 捕获函数-------------------------+
signal(SIGINT, sigint);
      Ctrl+C       硬件/内核
        |            用户进程
        v  ------------------
    SIGINT(2)      系统内核
        |
------  v  ------>  用户进程
       /    \
      /_____\
       Sigint
```

6.重入问题

```
      Ctrl+C              Ctrl+\
        |                   |
        v                   v
    SIGINT(2)          SIGQUIT(3)
        |                   |
------  v  ------------  v  ------>  用户进程
       /    \              /    \
      /_____\            /_____\
       sigint             sigint

          Ctrl+C
            |
            v
         SIGINT(2)
            |
--------  v  ---------->  用户进程
         /    \
        /      \
       / Ctrl+\\
      /     |    \
```



系统：在一个信号正在被处理的过程中，相同的信号会被阻塞(即被系统内核延迟递送)，直到前一个信号处理完毕，后一个信号才会被处理。
设计：尽量让一个信号处理函数只处理一种信号，避免出现一个信号处理函数处理多种不同信号的情况。
注意：在信号处理函数中尽量避免使用静态资源，尽可能使用栈内存保存数据。另外类似标准 I/O 等不可重入函数，要谨慎调用。
SIGINT(2) -> sigint -> printf
SIGQUIT(3) -> sigquit -> printf

代码：signal.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <unistd.h>
 4 #include <signal.h>
 5 void sigint(int signum) {
 6         printf("%d 进程：收到%d 信号！\n", getpid(), signum);
 7 }
 8 void sigkill(int signum) {
 9         printf("%d 进程：收到%d 信号！\n", getpid(), signum);
10 }
11 void sigterm(int signum) {
12         printf("%d 进程：收到%d 信号！\n", getpid(), signum);
13         printf("%d 进程：临终前最后工作...\n", getpid());
14         exit(0);
15 }
16 int main(void) {
17         if (signal(SIGINT, sigint/*SIG_IGN*/) == SIG_ERR) {
18                 perror("signal");
19                 return -1;
20         }
21 //      if (signal(SIGKILL, sigkill/*SIG_IGN*/) == SIG_ERR) {
22 //              perror("signal");
23 //              return -1;
24 //      }
25         if (signal(SIGTERM, sigterm) == SIG_ERR) {
26                 perror("signal");
27                 return -1;
28         }
29         for (;;);
```

```
30              return 0;
31 }
```

7.不可靠问题
当一个不可靠信号正在被处理的过程中，相同的多个信号被产生，其中只有第一个被阻塞，其余的则会丢失。

代码：mask.c

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <signal.h>
 4 void handler(int signum) {
 5              printf("%d 进程：收到%d 信号！\n", getpid(), signum);
 6              sleep(5);
 7 }
 8 int main(void) {
 9              if (signal(/*SIGINT*/50, handler) == SIG_ERR) {
10                      perror("signal");
11                      return -1;
12              }
13              for(;;);
14              return 0;
15 }
```

8.一次性问题
在某些 Unix 系统上，通过 signal 函数注册的信号处理函数只能一次性有效。如果希望获得持久的捕获效果，可以在信号处理函数中再注册一次。

```
void sigint(int signum) {
    ...
    signal(SIGINT, sigint);
}
...
signal(SIGINT, sigint);
```

9.太平间信号处理
无论一个进程是正常终止还是异常终止，都会通过系统内核向其父进程发送一个 SIGCHLD(17)信号。父进程完全可以在针对该信号的处理函数中，异步且及时地回收子进程的僵尸，这样信号处理就叫做太平间信号处理。

代码：child.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <errno.h>
 4 #include <unistd.h>
 5 #include <signal.h>
 6 #include <sys/wait.h>
 7 void sigchld(int signum) {
 8              for (;;) {
```

```
 9                      pid_t pid = waitpid(-1, NULL, WNOHANG);
10                      if (pid == -1) {
11                              if (errno != ECHILD) {
12                                      perror("waitpid");
13                                      exit(-1);//因为不是 main 函数，所以用 exit
14                              }
15                              printf("%d 进程：子进程的僵尸已被收尽！\n", getpid());
16                              break;
17                      }
18                      if (pid)
19                              printf("%d 进程：回收了%d 进程的僵尸。\n", getpid(), pid);
20                      else {
21                              printf("%d 进程：暂时没有僵尸需要回收。\n", getpid());
22                              break;
23                      }
24              }
25 }
26 int main(void) {
27              if (signal(SIGCHLD, sigchld) == SIG_ERR) {
28                      perror("signal");
29                      return -1;
30              }
31              for (int i = 0; i < 3; ++i) {
32                      pid_t pid = fork();
33                      if (pid == -1) {
34                              perror("fork");
35                              return -1;
36                      }
37                      if (pid == 0) {
38                              printf("%d 进程：我是子进程。\n", getpid());
39                              return 0;
40                      }
41              }
42              for(;;);
43              return 0;
44 }
```

10.信号处理的继承与恢复

```
父进程(忽略 SIGA，捕获 SIGB，默认 SIGC)
   |                      \_____ 继
 fork/                           \ 承
 vfork--子进程(忽略 SIGA，捕获 SIGB，默认 SIGC)
   |
   v
-------------------------------------------------
原进程(忽略 SIGA，捕获 SIGB，默认 SIGC)
   |                     | 恢
  exec                   | 复
```

```
   |                    v
新进程(忽略 SIGA，默认 SIGB，默认 SIGC)
```

代码：fork.c

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <signal.h>
 4 void sigint(int signum) {
 5         printf("%d 进程：收到%d 信号！\n", getpid(), signum);
 6 }
 7 int main(void) {
 8         if (signal(SIGINT, sigint) == SIG_ERR) {
 9                 perror("signal");
10                 return -1;
11         }
12         if (signal(SIGQUIT, SIG_IGN) == SIG_ERR) {
13                 perror("signal");
14                 return -1;
15         }
16         pid_t pid = fork();
17         if (pid == -1) {
18                 perror("fork");
19                 return -1;
20         }
21         if (pid == 0) {
22                 printf("%d 进程：我是子进程，正在运行中...\n", getpid());
23                 for (;;);
24                 return 0;
25         }
26         sleep(1);
27         printf("%d 进程：我是父进程，即将终止。\n", getpid());
28         return 0;
29 }
```

exec.c

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <signal.h>
 4 void sigint(int signum) {
 5         printf("%d 进程：收到%d 信号！\n", getpid(), signum);
 6 }
 7 int main(void) {
 8         if (signal(SIGINT, sigint) == SIG_ERR) {
 9                 perror("signal");
10                 return -1;
11         }
12         if (signal(SIGQUIT, SIG_IGN) == SIG_ERR) {
```

```
 13                     perror("signal");
 14                     return -1;
 15             }
 16             if (execl("./loop", "loop", NULL) == -1) {
 17                     perror("execl");
 18                     return -1;
 19             }
 20             return 0;
 21 }
```

11.发送信号
1)键盘、错误与命令
Ctrl+C，SIGINT(2)，终端中断符，终止进程
Ctrl+\，SIGQUIT(3)，终端退出符，终止进程并转储核心
Ctrl+Z，SIGTSTP(20)，终端停止符，暂停进程
执行非法指令：SIGILL(4)-一般是机器指令编码的时候
硬件或地址对齐错误：SIGBUS(7)
浮点运算异常：SIGFPE(8)
无效内存访问：SIGSEGV(11)
...

```
kill [-信号(缺省 SIGTERM(15))] <进程 PID>
kill -15 1234                  # 向 1234 进程发 15 信号
kill -SIGTERM 1234 5678 # 向 1234 进程和 5678 进程发 15 信号
kill -1                        # 向所有进程发 15 信号
```

2) 调用函数发送信号

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

向给定进程发送信号：

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

成功返回 0，失败返回-1。
pid: 目标进程的 PID，-1 表示系统中所有进程(只能是一个)
signum: 信号编号，0 用于检查 pid 所表示的进程是否存在,不存在返回-1,置 errno 为 ESRCH,存在返回 0。

代码：kill.c

```
  1 #include <stdio.h>
  2 #include <stdlib.h>
  3 #include <unistd.h>
  4 #include <signal.h>
  5 #include <sys/wait.h>
  6 #include <errno.h>
  7 void sigint(int signum) {
  8         printf("%d 进程：收到%d 信号！\n", getpid(), signum);
  9         exit(0);
 10 }
 11 int isdead(pid_t pid) {
```

```
12          if (kill(pid, 0) == -1) {
13                  if (errno != ESRCH) {
14                          perror("kill");
15                          exit(-1);
16                  }
17                  return 1;
18          }
19          return 0;
20  }
21  int main(void) {
22          pid_t pid = fork();
23          if (pid == -1) {
24                  perror("fork");
25                  return -1;
26          }
27          if (pid == 0) {
28                  printf("%d 进程：我是子进程，正在运行中...\n", getpid());
29                  if (signal(SIGINT, sigint) == SIG_ERR) {
30                          perror("signal");
31                          return -1;
32                  }
33                  for (;;);
34                  return 0;
35          }
36          sleep(1);
37          printf("%d 进程：我是父进程，我要杀死%d 进程...\n", getpid(), pid);
38          if (kill(pid, SIGINT) == -1) {
39                  perror("kill");
40                  return -1;
41          }
42          printf("%d 进程：%d 进程%s。\n", getpid(), pid,
43                  isdead(pid) ? "已死亡" : "还没死");
44          if ((pid = wait(NULL)) == -1) {
45                  perror("wait");
46                  return -1;
47          }
48          printf("%d 进程：%d 进程%s。\n", getpid(), pid,
49                  isdead(pid) ? "已死亡" : "还没死");
50          return 0;
51  }
```

向调用进程自己发送信号：
int raise(int signum);
成功返回 0，失败返回-1。
raise(SIGINT) <==> kill(getpid(), SIGINT)

代码：rasie.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <unistd.h>
 4 #include <signal.h>
 5 #include <sys/wait.h>
 6 #include <errno.h>
 7 void sigint(int signum) {
 8         printf("%d 进程：收到%d 信号！\n", getpid(), signum);
 9         exit(0);
10 }
11 int main(void) {
12         if (signal(SIGINT, sigint) == SIG_ERR) {
13                 perror("signal");
14                 return -1;
15         }
16         printf("%d 进程：我要自杀...\n", getpid());
17 //      if (raise(SIGINT) == -1) {
18         if (kill(getpid(), SIGINT) == -1) {
19                 perror("raise");
20                 return -1;
21         }
22         for (;;)
23                 printf("%d 进程：我还活着！\n", getpid());
24         return 0;
25 }
```

如论是用 kill 还是 raise 给调用进程自己发送被设置为捕获信号，这两个函数的返回都不会早于信号处理函数执行完成。但通过 kill 函数给其它进程发送信号，该函数会在信号发出后立即返回，并不等待信号处理函数的执行。

12.暂停、睡眠和闹钟

1)暂停：使一个进程处于被挂起的状态，即不参与系统内核的时间片分配，不被调度，即不使用处理机。这个状态会永远持续下去，除非用信号将该进程唤醒，使其恢复至被调度状态，继续运行。

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

成功阻塞，失败返回-1。

该函数使调用进程进入无时限的睡眠状态，直到有信号终止了调用进程或被其捕获。如果有信号被调用进程捕获，在信号处理函数返回以后，pause 函数才会返回，且返回值为-1，同时置 errno 为 EINTR，表示阻塞的系统调用被信号中断。pause 函数要么不返回，要么返回失败，永远不会返回成功。

代码：pause.c

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <unistd.h>
 4 #include <signal.h>
 5 #include <errno.h>
```

```
#include <sys/wait.h>
void sigint(int signum) {
        printf("%d 进程：收到%d 信号！\n", getpid(), signum);
}
int main(void) {
        pid_t pid = fork();
        if (pid == -1) {
                perror("fork");
                return -1;
        }
        if (pid == 0) {
                if (signal(SIGINT, sigint) == SIG_ERR) {
                        perror("signal");
                        return -1;
                }
                printf("%d 进程：我是子进程，大睡无期...\n", getpid());
                int ret = pause();
                printf("%d 进程：pause 函数返回%d，errno: %d (%s)\n", getpid(),
                        ret, errno, strerror(errno));
                return 0;
        }
        sleep(5);
        printf("%d 进程：我是父进程，我要向%d 进程发送%d 信号...\n", getpid(),
                pid, SIGINT);
        if (kill(pid, SIGINT) == -1) {
                perror("kill");
                return -1;
        }
        if ((pid = wait(NULL)) == -1) {
                perror("wait");
                return -1;
        }
        printf("%d 进程：%d 进程已终止。\n", getpid(), pid);
        return 0;
}
```

2)睡眠：使一个进程处于被挂起的状态，即不参与系统内核的时间片分配，不被调度，即不使用处理机。这个状态会最多持续一段时间，在这段时间内可以用信号将该进程唤醒，使其恢复至被调度状态，继续运行。超过了这个时间，即使没有信号唤醒该进程，该进程也会自己醒来，恢复运行。

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

返回 0 或剩余秒数。

seconds - 以秒为单位的睡眠时限。

```
                        返回 2      返回 0
sleep(10)               |           |
    |_____________10 秒_____________|
```

```
   |                          ^
   |          8 秒           |
                            信号
```

该函数使调用进程进入最多 seconds 秒的睡眠状态，在此期间如果有信号被调用进程捕获，在信号处理函数返回以后，sleep 函数才会返回，其返回值为剩余秒数，否则该函数将在 seconds 秒期满时返回 0。

代码：sleep.c

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <unistd.h>
 4 #include <signal.h>
 5 #include <errno.h>
 6 #include <sys/wait.h>
 7 void sigint(int signum) {
 8         printf("%d 进程：收到%d 信号！\n", getpid(), signum);
 9 }
10 int main(void) {
11         pid_t pid = fork();
12         if (pid == -1) {
13                 perror("fork");
14                 return -1;
15         }
16         if (pid == 0) {
17                 if (signal(SIGINT, sigint) == SIG_ERR) {
18                         perror("signal");
19                         return -1;
20                 }
21                 printf("%d 进程：我是子进程，小睡片刻...\n", getpid());
22                 int ret = sleep(10);
23                 printf("%d 进程：sleep 函数返回%d。\n", getpid(), ret);
24                 return 0;
25         }
26         sleep(7);
27         printf("%d 进程：我是父进程，我要向%d 进程发送%d 信号...\n", getpid(),
28                 pid, SIGINT);
29         if (kill(pid, SIGINT) == -1) {
30                 perror("kill");
31                 return -1;
32         }
33         if ((pid = wait(NULL)) == -1) {
34                 perror("wait");
35                 return -1;
36         }
37         printf("%d 进程：%d 进程已终止。\n", getpid(), pid);
38         return 0;
39 }
```

#include <unistd.h>
int usleep(useconds_t usec);
时限未到阻塞，时限到了返回 0，时限到之前被信号中断返回-1，同时置 errno 为 EINTR，其它错误返回-1。
usec - 以微秒(1 微秒=0.000001 秒)为单位的睡眠时限。

代码：usleep.c

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>
#include <errno.h>
#include <sys/wait.h>
void sigint(int signum) {
        printf("%d 进程：收到%d 信号！\n", getpid(), signum);
}
int main(void) {
        pid_t pid = fork();
        if (pid == -1) {
                perror("fork");
                return -1;
        }
        if (pid == 0) {
                if (signal(SIGINT, sigint) == SIG_ERR) {
                        perror("signal");
                        return -1;
                }
                printf("%d 进程：我是子进程，小睡片刻...\n", getpid());
                int ret = usleep(100000); // 100 毫秒
                printf("%d 进程：usleep 函数返回%d，errno: %d (%s)\n", getpid(),
                        ret, errno, strerror(errno));
                return 0;
        }
//      usleep(50000); // 50 毫秒
        usleep(200000); // 200 毫秒
        printf("%d 进程：我是父进程，我要向%d 进程发送%d 信号...\n", getpid(),
                pid, SIGINT);
        if (kill(pid, SIGINT) == -1) {
                perror("kill");
                return -1;
        }
        if ((pid = wait(NULL)) == -1) {
                perror("wait");
                return -1;
        }
        printf("%d 进程：%d 进程已终止。\n", getpid(), pid);
```

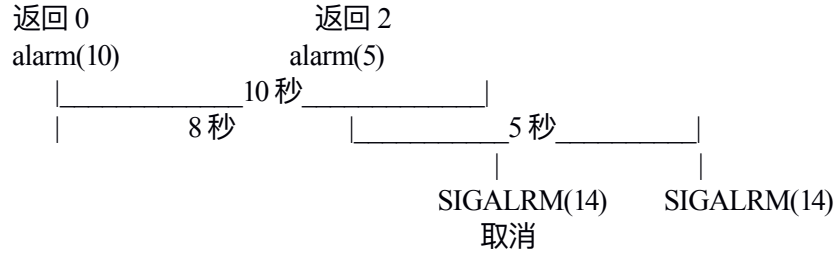
```
40              return 0;
41 }
```

3)闹钟：在一个给定时间之后向本进程发送 SIGALRM(14)信号。

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

返回 0 或先前所设闹钟的剩余秒数。

seconds - 以秒为单位的闹钟时间。



alarm 函数使系统内核在该函数被调用以后 seconds 秒的时候，向调用进程发送 SIGALRM(14)信号。若在调用该函数前已设过闹钟且尚未到期，该函数会重设闹钟，并返回先前所设闹钟的剩余秒数，否则返回 0。若 seconds 参数取 0，则表示仅取消先前设过且尚未到期的闹钟，并不会开启新闹钟。通过 alarm 函数所设置的闹钟只是一次性的，若要获得周期性的效果，可在对 SIGALRM(14)信号的处理函数中再次调用 alarm 函数设定下一个闹钟。

代码：alarm.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <unistd.h>
 4 #include <signal.h>
 5 void sigalrm(int signum) {
 6         printf("%d 进程：收到%d 信号！\n", getpid(), signum);
 7         exit(0);
 8 }
 9 int main(void) {
10         if (signal(SIGALRM, sigalrm) == SIG_ERR) {
11                 perror("signal");
12                 return -1;
13         }
14         printf("%d 进程：设定闹钟，剩余%d 秒。\n", getpid(), alarm(10));
15         sleep(2);
16         printf("%d 进程：重设闹钟，剩余%d 秒。\n", getpid(), alarm(5));
17         sleep(2);
18         printf("%d 进程：取消闹钟，剩余%d 秒。\n", getpid(), alarm(0));
19         for (;;);
20         return 0;
21 }
```

clock.c

```
 1 #include <stdio.h>
 2 #include <time.h>
```

```
 3 #include <unistd.h>
 4 #include <signal.h>
 5 void sigalrm(int signum) {
 6         time_t t = time(NULL);//取当前系统时间
 7         struct tm* lt = localtime(&t);
 8         printf("\r%02d:%02d:%02d", lt->tm_hour, lt->tm_min, lt->tm_sec);
 9         alarm(1);
10 }
11 int main(void) {
12         setbuf(stdout, NULL);
13         if (signal(SIGALRM, sigalrm) == SIG_ERR) {
14                 perror("signal");
15                 return -1;
16         }
17         sigalrm(SIGALRM);
18         for (;;);
19         return 0;
20 }
```

13.信号集

1)位图法

设计一种容器，存放最多 8 个位于[0,7]区间的不同的整数。

```
int a[8] = {5, 1, 4, 7, 3};
foo(a, 5);
int b[8] = {0};
b[5] = b[1] = b[4] = b[7] = b[3] = 1;
bar(b);
char c = 32 | 2 | 16 | 128 | 8;
5 - 00100000 - 32
1 - 00000010 - 2
4 - 00010000 - 16
7 - 10000000 - 128
3 - 00001000 - 8
-------------------
      10111010 - c
      76543210
hum(c);
```

位图法数据集：用位号表示数据，用该位的值 1 或 0 表示有没有该数据。

2)信号集类型

<sigset.h>

```
#define _SIGSET_NWORDS \
      (1024 / (8 * sizeof(unsigned long int)))
      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                                |
                               32

unsigned long int __val[_SIGSET_NWORDS];
```

```
                                |
32 个元素，每个元素 4 个字节，一共 128 字节，1024 个二进制位。
          65-1024   34-64          1-31
           预留     可靠信号       不可靠信号
          ___|___   __|__         ___|___
         /       \/      \       /       \
__val：000 ... 00 ... 0 ... 0 ... 110
         -------------------------------
  位：  111      00      0      0      000   |  读
        000      00      0      0      000   |  顺
        222      66      3      3      000   |  续
        321      43      3      0      210   v
        \_____________________________________/
                                |
                  共 1024 个二进制位
                  每位表示一个信号
SIGINT(2) -> 第 1 位
SIGQUIT(3) -> 第 2 位
void foo(...) {
    unsigned long int __val[_SIGSET_NWORDS] = {};
    SIGINT(2) -> __val
    SIGQUIT(3) -> __val
    内核函数(__val);
    ...
}                                                  用户空间
-----------------------|-----------------------
                        __val(指针)              内核空间

typedef struct {
      unsigned long int __val[_SIGSET_NWORDS];
}     __sigset_t;
typedef __sigset_t sigset_t; // 信号集的数据类型，结构体类型
void foo(...) {
      sigset_t ss;
      SIGINT(2) -> ss.__val
      SIGQUIT(3) -> ss.__val
      内核函数(ss);
      ...
}                                                  用户空间
-----------------------------------------------
                                ss(副本)           内核空间
```

3)信号集运算

```
int sigfillset   (sigset_t* sigset); // 填满信号集
int sigemptyset  (sigset_t* sigset); // 清空信号集
int sigaddset    (sigset_t* sigset, int signum); // 加入信号
int sigdelset    (sigset_t* sigset, int signum); // 删除信号
```

成功返回 0，失败返回-1。
int sigismember(const sigset_t* sigset, int signum);
// 检查信号集中是否包含特定的信号
包含返回 1，不包含返回 0，失败返回-1。

代码：sigset.c

```
 1 #include <stdio.h>
 2 #include <signal.h>
 3 void printb(char byte) {
 4         for (int i = 0; i < 8; ++i)
 5             printf("%c", byte & 1 << 7 - i ? '1' : '0');
 6         printf(" ");
 7 }
 8 void printm(void* buf, int len) {
 9         for (int i = 0; i < len; ++i) {
10             printb(((char*)buf)[len - 1 - i]);
11             if ((i + 1) % 8 == 0 || i == len - 1)
12                 printf("\n");
13         }
14 }
15 int main(void) {
16         sigset_t sigset;
17         printf("满信号集：\n");
18         sigfillset(&sigset);
19         printm(&sigset, sizeof(sigset));
20         printf("空信号集：\n");
21         sigemptyset(&sigset);
22         printm(&sigset, sizeof(sigset));
23         printf("加入 SIGINT(%d)信号：\n", SIGINT);
24         sigaddset(&sigset, SIGINT);
25         printm(&sigset, sizeof(sigset));
26         printf("加入 SIGQUIT(%d)信号：\n", SIGQUIT);
27         sigaddset(&sigset, SIGQUIT);
28         printm(&sigset, sizeof(sigset));
29         printf("删除 SIGQUIT(%d)信号：\n", SIGQUIT);
30         sigdelset(&sigset, SIGQUIT);
31         printm(&sigset, sizeof(sigset));
32         printf("信号集中%sSIGINT(%d)信号。\n",
33             sigismember(&sigset, SIGINT) ? "有" : "无", SIGINT);
34         printf("信号集中%sSIGQUIT(%d)信号。\n",
35             sigismember(&sigset, SIGQUIT) ? "有" : "无", SIGQUIT);
36         return 0;
37 }
```

14.信号屏蔽
1)信号递送、信号未决和信号掩码
信号递送：当信号产生时，系统内核会在其维护的进程表项中，为接收信号的进程设置一个与

该信号相对应的标志位，这个过程就叫递送，保存被递送信号的标志位集就叫做递送集。

进程表项
　　环境变量
　　文件描述符表
　　权限掩码　　+--SIGINT(2)
　　当前工作目录　　|
　　各种 ID　　v
　　信号递送集(sigset_t)：000...010

信号未决：信号从产生到完成递送存在一定的时间间隔，处于这段时间间隔中的信号状态被称为未决状态。用于保存该进程所有处于未决状态的信号的信号集叫做未决集。

信号掩码：每个进程在系统内核中都有一个保存在进程表项中信号掩码，信号掩码本身也是一个信号集，凡是属于该信号集的信号，都会被阻塞于未决状态。

进程表项
　　环境变量
　　文件描述符表　　+--SIGQUIT(3)
　　权限掩码　　|+--SIGINT(2)
　　当前工作目录　　||
　　各种 ID　　vv
　　未决信号集(sigset_t)：000...100
　　掩码信号集(sigset_t)：000...100
　　递送信号集(sigset_t)：000...010

2)设置掩码和检测未决
#include <signal.h>
int sigprocmask(int how, const sigset_t* sigset,sigset_t* oldset);
成功返回 0，失败返回-1。
how: 怎么设置掩码? 可取以下值：
　　SIG_BLOCK　- 将 sigset 中的信号加入当前掩码信号集　(+)
　　SIG_UNBLOCK - 从当前掩码信号集中删除 sigset 中的信号 (-)
　　SIG_SETMASK - 把 sigset 设置成当前掩码信号集　(=)

sigset: 信号集，取 NULL 则忽略此参数。
oldset: 输出调用该函数之前掩码信号集的备份，取 NULL 则忽略此参数。

获取当前未决的信号
int sigpending(sigset_t* sigset);
成功返回 0，失败返回-1。
sigset: 输出未决信号集

注意：对于可靠信号，通过 sigprocmask 函数设置信号掩码以后，每种被屏蔽的信号中的每个信号都会被阻塞，并按先后顺序排队，一旦解除屏蔽，这些信号会被依次递送。而对于不可靠信号，通过 sigprocmask 函数设置信号掩码以后，每种被屏蔽的信号中只有第一个会被阻塞，并在解除屏蔽后被递送，其余的则全部丢失。

代码：sigmask.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <unistd.h>
 4 #include <signal.h>
 5 void sighand(int signum) {
 6         printf("%d 进程：收到%d 信号！\n", getpid(), signum);
 7 }
 8 void updatedb(void) {
 9         for (int i = 0; i < 3; ++i) {
10                 printf("%d 进程：更新第%d 条记录。\n", getpid(), i + 1);
11                 sleep(1);
12         }
13 }
14 int main(void) {
15         if (signal(SIGINT, sighand) == SIG_ERR) {
16                 perror("signal");
17                 return -1;
18         }
19         if (signal(50, sighand) == SIG_ERR) {
20                 perror("signal");
21                 return -1;
22         }
23         printf("%d 进程：屏蔽%d 信号和%d 信号。\n", getpid(), SIGINT, 50);
24         sigset_t newset;
25         sigemptyset(&newset);
26         sigaddset(&newset, SIGINT);
27         sigaddset(&newset, 50);
28         sigset_t oldset;
29         if (sigprocmask(SIG_SETMASK, &newset, &oldset) == -1) {
30                 perror("sigprocmask");
31                 return -1;
32         }
33         pid_t pid = fork();
34         if (pid == -1) {
35                 perror("fork");
36                 return -1;
37         }
38         if (pid == 0) {
39                 pid_t ppid = getppid();
40                 for (int i = 0; i < 3; ++i) {
41         printf("%d 进程：向%d 进程发送%d 信号和%d 信号...\n", getpid(), ppid,
42                                         SIGINT, 50);
43                         kill(ppid, SIGINT);
44                         kill(ppid, 50);
45                 }
46                 return 0;
```

```
47              }
48              updatedb();
49              /*
50              sigset_t pendingset;
51              if (sigpending(&pendingset) == -1) {
52                      perror("sigpending");
53                      return -1;
54              }
55              if (sigismember(&pendingset, SIGINT)) {
56                      printf("%d 进程：发现%d 信号未决。补充处理...\n", getpid(), SIGINT);
57                      // ...
58              }
59              if (sigismember(&pendingset, 50)) {
60                      printf("%d 进程：发现%d 信号未决。补充处理...\n", getpid(), 50);
61                      // ...
62              }
63              */
64              printf("%d 进程：取消对%d 信号和%d 信号的屏蔽。\n", getpid(), SIGINT, 50);
65              if (sigprocmask(SIG_SETMASK, &oldset, NULL) == -1) {
66                      perror("sigprocmask");
67                      return -1;
68              }
69              getchar();
70              return 0;
71      }
```

15.现代风格的信号处理与发送

	经典风格	现代风格
信号处理	signal	sigaction
信号发送	kill	sigqueue

1)信号处理

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

成功返回 0，失败返回-1。

signum: 信号编号。 /忽略

sigact: (针对 signum)信号(所采取的)行动(-默认)。

\捕获及屏蔽等

oldact: 输出原来的信号处理，可置 NULL。

当 signum 信号被递送时，按 sigact 所描述的行为响应之。若 oldact 非 NULL，则通过改参数输出原来的响应行为。

```
struct sigaction {

        void (*sa_handler)(int);// 经典风格的信号处理函数指针
        void (*sa_sigaction)(int, siginfo_t*, void*);// 现代风格的信号处理函数指针
        sigset_t sa_mask;// 信号处理期间的附加(正在被处理的信号以外的)信号掩码
```

```
    int  sa_flags;// 信号处理标志
    void  (*sa_restorer)(void);// 预留项，目前置 NULL
};
```

A.附加屏蔽

缺省情况下，在信号处理函数的执行过程中，会自动屏蔽这个正在被处理的信号，而对于其它信号不会屏蔽。通过 sigaction 结构的 sa_mask 字段可以人为指定，在信号处理函数执行期间，除正在被处理的这个信号以外，还想屏蔽哪些信号，并在信号处理结束后，自动解除对它们的屏蔽。

B.禁止屏蔽

缺省情况下，在信号处理函数的执行过程中，会自动屏蔽这个正在被处理的信号，但是如果在 sigaction 结构的 sa_flags 字段中包含 SA_NOMASK 标志位，那么该信号就不会其处理过程中被屏蔽，而是被抢先递送。

C.风格选择

缺省情况下，sigaction 结构中表示经典风格信号处理函数指针的 sa_handler 字段有效，该指针所指向的信号处理函数只带有一个参数，即信号编号，其功能相对较弱。如果 sigaction 结构的 sa_flags 字段中包含 SA_SIGINFO 标志位，那么该结构中表示现代风格信号处理函数指针的 sa_sigaction 字段有效，该指针所指向的信号处理函数带有三个参数(目前只用前两个参数)，除了第一个表示信号编号的整型参数以外，还有一个 siginfo_t 结构体指针类型的参数，该参数所指向的结构体中包含了有关该信号更多的细节。

```
typedef  struct  siginfo  {
    pid_t       si_pid;     // 信号发送者进程的 PID
    sigval_t  si_value;  // 信号附加数据<--+
    ...                                                |
}   siginfo_t;                                      |
                                                       |
typedef  union  sigval  {                          |
    int     sival_int;     // 整数--+--------+
    void*  sival_ptr;     // 指针_/
}   sigval_t;
```

代码：sigact.c

```
 1 #include  <stdio.h>
 2 #include  <unistd.h>
 3 #include  <signal.h>
 4 void  sigint1(int  signum)  {
 5             printf("%d 进程：收到%d 信号！睡眠...\n",  getpid(),  signum);
 6             sleep(5);
 7             printf("%d 进程：醒了。\n",  getpid());
 8 }
 9 void  sigint2(int  signum,  siginfo_t*  si,  void*  pv)  {
10          printf("%d 进程：收到来自%d 进程的%d 信号！\n",  getpid(),  si->si_pid,  s      ignum);
11 }
12 int  main(void)  {
13             struct  sigaction  act  =  {};
14             /*
```

```
15              printf("只屏蔽 SIGINT(2)信号，不屏蔽 SIGQUIT(3)信号。\n");
16              act.sa_handler = sigint1;
17              *//*
18              printf("既屏蔽 SIGINT(2)信号，也屏蔽 SIGQUIT(3)信号。\n");
19              act.sa_handler = sigint1;
20              sigaddset(&act.sa_mask, SIGQUIT);
21              *//*
22              printf("不屏蔽 SIGINT(2)信号，不屏蔽 SIGQUIT(3)信号。\n");
23              act.sa_handler = sigint1;
24              act.sa_flags = SA_NOMASK;
25              *//*
26              printf("不屏蔽 SIGINT(2)信号，只屏蔽 SIGQUIT(3)信号。\n");
27              act.sa_handler = sigint1;
28              act.sa_flags = SA_NOMASK;
29              sigaddset(&act.sa_mask, SIGQUIT);
30              *//*
31              printf("通过现代风格信号处理函数获取信号的更多细节。\n");
32              act.sa_sigaction = sigint2;
33              act.sa_flags = SA_SIGINFO;
34              */
35              printf("一次性信号处理。\n");
36              act.sa_handler = sigint1;
37              act.sa_flags = SA_ONESHOT;
38              if (sigaction(SIGINT, &act, NULL) == -1) {
39                      perror("sigaction");
40                      return -1;
41              }
42              for(;;) pause();
43              return 0;
44      }
```

D.一次性信号处理

为了像某些特殊的 Unix 系统一样，在第一次执行信号处理函数之前，先将其恢复为默认处理，以后再递送该信号，不调用信号处理函数，而是以默认的方式处理该信号，可在 sigaction 结构的 sa_flags 字段中包含 SA_ONESHOT 标志位。

E.自动重启被中断的系统调用

缺省情况下，当进程正阻塞于某些系统调用时，如果收到信号，系统会在针对该信号的信号处理函数返回以后，中断这个被阻塞的系统调用，不再继续阻塞，而是返回失败，同时置 errno 为 EINTR。为了完成之前尚未完成的任务，需要做一些特殊处理，以继续之前的工作。如果在 sigaction 结构的 sa_flags 字段中包含 SA_RESTART 标志位，该被中断的系统调用将在信号处理函数返回以后被自动重启，即继续阻塞，不会返回失败，之前未完成的任务得以继续执行。

代码：restart.c

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <unistd.h>
```

```
 4 #include <signal.h>
 5 #include <errno.h>
 6 void sigint(int signum) {
 7         printf("%d 进程：收到%d 信号！\n", getpid(), signum);
 8 }
 9 int main(void) {
10         struct sigaction act = {};
11         act.sa_handler = sigint;
12         act.sa_flags = SA_RESTART;
13         if (sigaction(SIGINT, &act, NULL) == -1) {
14                 perror("sigaction");
15                 return -1;
16         }
17         ssize_t len;
18         char buf[256];
19 //again:
20         if ((len = read(STDIN_FILENO, buf, sizeof(buf))) == -1) {
21 //              if (errno == EINTR)
22 //                      goto again;
23                 perror("read");
24                 return -1;
25         }
26         if (write(STDOUT_FILENO, buf, len) == -1) {
27                 perror("write");
28                 return -1;
29         }
30         return 0;
31 }
```

2)信号发送

```
#include <signal.h>
int sigqueue(pid_t pid, int signum,
      const union sigval value);
                              |_____________________________
                                                            |
                                  +-->siginfo_t               v
                                  |         sigval_t si_value;
                                  |         ...
                                  +-----+
                                        |
void sigxxx(int signum, siginfo_t* si, void* pv) { ... }
typedef union sigval {
      int     sival_int; // 整数
      void* sival_ptr; // 指针
}     sigval_t;
```

pid: 接收信号进程的 PID

signum: 信号编号

value: 附加数据
注意：如果伴随信号的附加数据是一个指针，那么一定要保证该指针所指向的内存区域，在接收信号的进程中也要有效且可被访问。

代码：sigqueue.c

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <signal.h>
 4 void sighandler(int signum, siginfo_t* si, void* pv) {
 5         printf("%d 进程：收到来自%d 进程的%d 信号，其附加数据是%d！\n",
 6                 getpid(), si->si_pid, signum, si->si_value.sival_int);
 7 }
 8 int main(void) {
 9         int signum = /*SIGINT*/50;
10         struct sigaction act = {};
11         act.sa_sigaction = sighandler;
12         act.sa_flags = SA_SIGINFO | SA_RESTART;
13         if (sigaction(signum, &act, NULL) == -1) {
14                 perror("sigaction");
15                 return -1;
16         }
17         pid_t pid = fork();
18         if (pid == -1) {
19                 perror("fork");
20                 return -1;
21         }
22         if (pid == 0) {
23                 pid_t ppid = getppid();
24                 for (int i = 0; i < 10; ++i) {
25                         sigval_t sv;
26                         sv.sival_int = i;
27                         if (sigqueue(ppid, signum, sv) == -1) {
28                         //if (kill(ppid, signum) == -1) {
29                                 perror("sigqueue");
30                                 return -1;
31                         }
32                 }
33                 return 0;
34         }
35         getchar();
36         return 0;
37 }
```

利用信号附加数据在不同进程之间建立更复杂的通信机制，这也是一种将信号用于 IPC 的典型场景。

代码：send.c

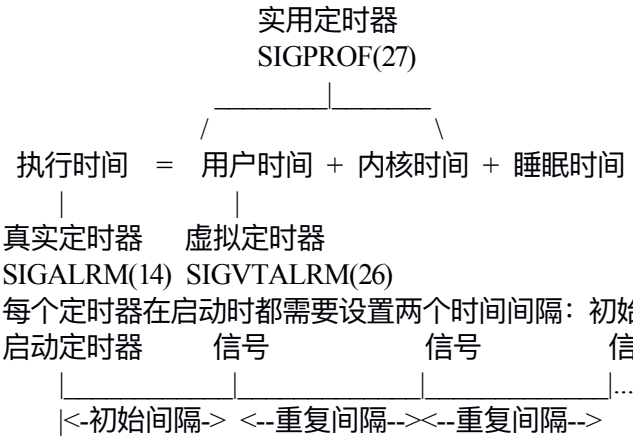
```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <string.h>
 4 #include <unistd.h>
 5 #include <signal.h>
 6 int main(int argc, char* argv[]) {
 7         if (argc < 2) {
 8                 fprintf(stderr, "用法：%s <PID>\n", argv[0]);
 9                 return -1;
10         }
11         char buf[1024];
12         for (;;) {
13                 printf("> ");
14                 fgets(buf, sizeof(buf) / sizeof(buf[0]), stdin);
15                 if (!strcmp(buf, "!\n"))
16                         break;
17                 for (int i = 0; buf[i]; ++i) {
18                         sigval_t sv = {};
19                         sv.sival_int = buf[i];
20                         if (sigqueue(atoi(argv[1]), SIGINT/*50*/, sv) == -1)        {
21                                 perror("sigqueue");
22                                 return -1;
23                         }
24                         usleep(1000);
25                 }
26         }
27         return 0;
28 }
```

recv.c

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <signal.h>
 4 void sighandler(int signum, siginfo_t* si, void* pv) {
 5         printf("%c", si->si_value.sival_int);
 6 }
 7 int main(void) {
 8         printf("PID: %d\n", getpid());
 9         setbuf(stdout, NULL);
10         struct sigaction sa = {};
11         sa.sa_sigaction = sighandler;
12         sa.sa_flags = SA_SIGINFO | SA_RESTART;
13         if (sigaction(SIGINT/*50*/, &sa, NULL) == -1) {
14                 perror("sigaction");
15                 return -1;
16         }
17         getchar();
```

```
18              return 0;
19 }
```

16.周期定时器



每个定时器在启动时都需要设置两个时间间隔：初始间隔和重复间隔

初始间隔=0：关闭定时器，在也收不到定时器的信号了。
重复间隔=0：只发第一个信号，没有周期性效果。

表示定时器属性的数据类型：

```
struct itimerval {
    struct timeval it_interval; // 重复间隔
    struct timeval it_value;    // 初始间隔
};
struct timeval {
    long tv_sec;  // 秒
    long tv_usec; // 微秒
};
```

123.456 秒

```
struct timeval t = {123, 456000};
```

启动、修改、关闭周期定时器：

```
#include <sys/time.h>
int setitimer(int which, const struct itimerval* new_value,
    struct itimerval* old_value);
```

成功返回 0，失败返回-1。

which：哪个定时器，可取以下值：

```
        ITIMER_REAL    - 真实定时器
        ITIMER_VIRTUAL - 虚拟定时器
        ITIMER_PROF    - 实用定时器
```

new_value：新属性。
old_value：输出旧属性，可置 NULL。

获取周期定时器的当前属性：

```
int getitimer(int which, struct itimerval* curr_value);
```

成功返回 0，失败返回-1。

which：哪个定时器，可取以下值：

ITIMER_REAL - 真实定时器
ITIMER_VIRTUAL - 虚拟定时器
ITIMER_PROF - 实用定时器

curr_value: 输出当前属性。

代码：timer.c

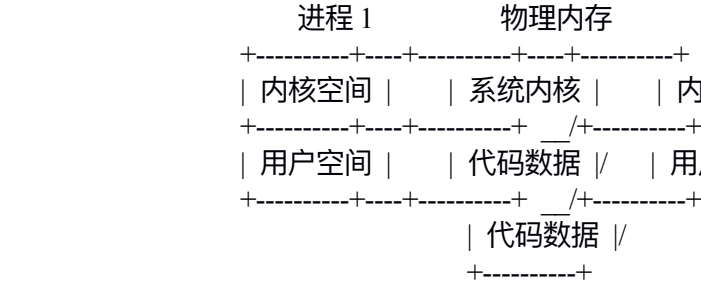
```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <signal.h>
 4 #include <sys/time.h>
 5 int h = 0, m = 0, s = 0, ms = 0;
 6 void sigalrm(int signum) {
 7         printf("\r%02d:%02d:%02d.%03d", h, m, s, ms);
 8         if (++ms == 1000) {
 9                 ms = 0;
10                 if (++s == 60) {
11                         s = 0;
12                         if (++m == 60) {
13                                 m = 0;
14                                 ++h;
15                         }
16                 }
17         }
18 }
19 void sigint(int signum) {
20         static int run = 1;
21         struct itimerval it = {};
22         if (!run) {
23                 printf("\n");
24                 h = m = s = ms = 0;
25                 it.it_value.tv_usec = 1;
26                 it.it_interval.tv_usec = 1000;
27         }
28         setitimer(ITIMER_REAL, &it, NULL);
29         run ^= 1;
30 }
31 int main(void) {
32         setbuf(stdout, NULL);
33         if (signal(SIGALRM, sigalrm) == SIG_ERR) {
34                 perror("signal");
35                 return -1;
36         }
37         if (signal(SIGINT, sigint) == SIG_ERR) {
38                 perror("signal");
39                 return -1;
40         }
41         struct itimerval it = {{0, 1000}, {5, 0}};
```

```
42              if (setitimer(ITIMER_REAL, &it, NULL) == -1) {
43                      perror("setitimer");
44                      return -1;
45              }
46              struct itimerval cv;
47              if (getitimer(ITIMER_REAL, &cv) == -1) {
48                      perror("getitimer");
49                      return -1;
50              }
51              printf("初始间隔：%ld 秒%ld 微秒\n",
52                      cv.it_value.tv_sec, cv.it_value.tv_usec);
53              printf("重复间隔：%ld 秒%ld 微秒\n",
54                      cv.it_interval.tv_sec, cv.it_interval.tv_usec);
55              getchar();
56              return 0;
57 }
```

十、进程间通信

1.为什么需要进程间通信?

对每个进程而言，除去内核空间以外，都有各自独立的内存映射表，记录着各自虚拟内存和物理内存的映射关系。即使是同一个虚拟内存地址，在不同的进程中，也会被映射到完全不同的物理内存区域，因此在多个进程之间通过交换虚拟内存地址的方式交换数据是不可能的。鉴于进程之间天然存在的内存壁垒，要想实现多个进程间的数据交换，就必须提供一种专门的机制，这就是所谓的进程间通信(InterProcess Communication, IPC)。



2.进程间通信的种类

1)命令行参数 \
2)环境变量 |
3)退出码 > 基本进程间通信，通信能力较弱，表现力不强
4)内存映射文件 |
5)信号 /
6)有名管道 \ 经典进程间通信，兼容性最好，功能十分强大
7)无名管道 /
8)共享内存 \
9)消息队列 > XSI 进程间通信，违背一切皆文件的设计理念
10)信号量(灯) /
11)本地套接字 - BSD 模仿网络通信的接口实现有名管道的功能

3.有名管道

```
                        | +--------+ |
进程 1<->文件描述符<->|有名管道|<->文件描述符<->进程 2
                        | +--------+ |
                        +------------+
文件描述符表
0 文件描述符标志 | * -> 文件表项
1 文件描述符标志 | * -> 文件表项
2 文件描述符标志 | * -> 文件表项
...
7 文件描述符标志 | * -> 文件表项
...                          v 节点指针 -> v 节点
                                                i 节点信息
                                                  元数据
                                                  * -> 有名管道
```

命令中通讯：
Shell 终端命令:makefifo + 有名管道文件
一个进程的 shell 终端：echo ‘hello’ > 有名管道文件名
另一个进程的 shell 终端：cat 有名管道文件名

有名管道借助于文件系统使多个进程汇合，这些进程可以没有父子或兄弟等任何直接关系，只要可以访问文件系统，就可以象读写文件一样，读写有名管道，借以实现进程间的数据交换。

创建有名管道(对象/文件)：

```
#include <sys/stat.h>//加权限用的
int mkfifo(const char* pathname, mode_t mode);
```

成功返回 0，失败返回-1。
pathname: 文件路径
mode: 权限模式
删除用：unlink
代码：wfifo.c

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <unistd.h>
 4 #include <fcntl.h>
 5 #include <sys/stat.h>
 6 #define FIFO_FILE "myfifo"
 7 int main(void) {
 8         printf("创建管道...\n");
 9         if (mkfifo(FIFO_FILE, 0666) == -1) {
10                 perror("mkfifo");
11                 return -1;
12         }
13         printf("打开管道...\n");
14         int fd = open(FIFO_FILE, O_WRONLY);
15         if (fd == -1) {
16                 perror("open");
17                 return -1;
```

```
18              }
19              printf("发送数据...\n");
20              for (;;) {
21                      printf("> ");
22                      char buf[1024];
23                      fgets(buf, sizeof(buf) / sizeof(buf[0]), stdin);
24                      if (!strcmp(buf, "!\n"))
25                              break;
26                      if (write(fd, buf, strlen(buf) * sizeof(buf[0])) == -1) {
27                              perror("write");
28                              return -1;
29                      }
30              }
31              printf("关闭管道...\n");
32              if (close(fd) == -1) {
33                      perror("close");
34                      return -1;
35              }
36              printf("删除管道...\n");
37              if (unlink(FIFO_FILE) == -1) {
38                      perror("unlink");
39                      return -1;
40              }
41              return 0;
42      }
```

rfifo.c

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <fcntl.h>
4  #define FIFO_FILE "myfifo"
5  int main(void) {
6               printf("打开管道...\n");
7               int fd = open(FIFO_FILE, O_RDONLY);
8               if (fd == -1) {
9                       perror("open");
10                      return -1;
11              }
12              printf("读取数据...\n");
13              for (;;) {
14                      char buf[1024] = {};
15                      ssize_t rb = read(fd, buf, sizeof(buf) - sizeof(buf[0]));
16                      if (rb == -1) {
17                              perror("read");
18                              return -1;
19                      }
20                      if (!rb)//对方把管道关了
```

```
 21                                      break;
 22                              printf("< %s", buf);
 23              }
 24              printf("关闭管道...\n");
 25              if (close(fd) == -1) {
 26                              perror("close");
 27                              return -1;
 28              }
 29              return 0;
 30 }
```

4.无名管道(在父子进程或者在兄弟进程中使用)

创建无名管道:

```
#include <unistd.h>
int pipe(int pipefd[2]);
```

成功返回 0，失败返回-1。

pipefd: 输出两个文件描述符，其中 pipefd[0]用于从无名管道读取数据，而 pipefd[1]用于向无名管道写入数据。

```
                                +--系统内核--+
                                | +--------+ |
父进程<->写端文件描述符-->|无名管道|-->读端文件描述符->子进程
                                | +--------+ |
                                +------------+

(父)进程       /pipefd[0]-读端文件描述符<-关闭
pipe->pipefd
      |fork     \pipefd[1]-写端文件描述符<-数据
      v
子进程          /pipefd[0]-读端文件描述符->数据
         pipefd
                \pipefd[1]-写端文件描述符<-关闭
-----------------------------------------
(父)进程       /pipefd[0]-读端文件描述符
pipe->pipefd
      |fork     \pipefd[1]-写端文件描述符
      v
兄进程          /pipefd[0]-读端文件描述符<-关闭
         pipefd
                \pipefd[1]-写端文件描述符<-数据

弟进程          /pipefd[0]-读端文件描述符->数据
         pipefd
                \pipefd[1]-写端文件描述符<-关闭
```

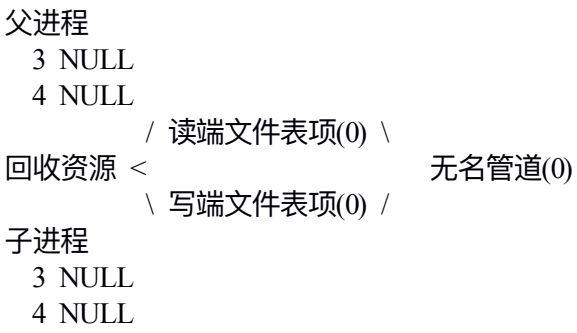
注意：基于有名管道的进程间通信是全双工通信，而基于无名管道的进程间通信是半双工通信，如果需要在亲属进程之间建立双向通信，可以创建两个无名管道，一个从父(兄)流向子(弟)，一个从子(弟)流向父(兄)。

代码：pipe.c

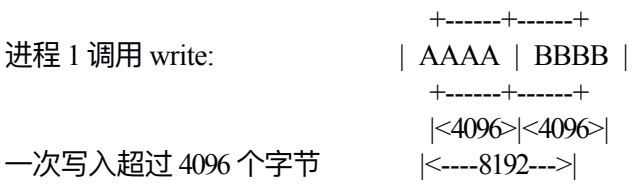
```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <unistd.h>
 4 #include <sys/wait.h>
 5 int main(void) {
 6         printf("父进程：创建管道...\n");
 7         int pipefd[2];
 8         if (pipe(pipefd) == -1) { // [1, 1]
 9                 perror("pipe");
10                 return -1;
11         }
12 // pipefd[0]：用于读取无名管道的文件描述符，称为该管道的读端，用于接收数据
13 // pipefd[1]：用于写入无名管道的文件描述符，称为该管道的写端，用于发送数据
14         printf("管道读端：%d\n 管道写端：%d\n", pipefd[0], pipefd[1]);
15         printf("父进程：创建进程...\n");
16         pid_t pid = fork(); // [2, 2]
17         if (pid == -1) {
18                 perror("fork");
19                 return -1;
20         }
21         if (pid == 0) {
22                 printf("子进程：关闭写端...\n");
23                 close(pipefd[1]); // [2, 1]
24                 printf("子进程：接收数据...\n");
25                 for (;;) {
26                         char buf[1024] = {};
27                         ssize_t rb = read(pipefd[0], buf, sizeof(buf) - sizeof(buf[0]));
28                         if (rb == -1) {
29                                 perror("read");
30                                 return -1;
31                         }
32                         if (!rb)
33                                 break; // 发送方关闭了管道的写端
34                         fputs(buf, stdout);
35                 }
36                 printf("子进程：关闭读端...\n");
37                 close(pipefd[0]); // [1, 1]
38                 return 0;
39         }
40         printf("父进程：关闭读端...\n");
41         close(pipefd[0]); // [0(释放读端文件描述符所对应的文件表项), 1]
42         printf("父进程：发送数据...\n");
43         for (;;) {
44                 char buf[1024];
45                 fgets(buf, sizeof(buf) / sizeof(buf[0]), stdin);
46                 if (!strcmp(buf, "!\n"))
```

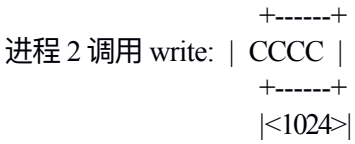
```
47                              break;
48                      if (write(pipefd[1], buf, strlen(buf) * sizeof(buf[0])) == -      1) {
49                              perror("write");
50                              return -1;
51                      }
52              }
53              printf("父进程：关闭写端...\n");
54              close(pipefd[1]); // 导致接收方的 read 返回 0
55                                      // [0, 0(释放写端文件描述符所对应的文件表项)]
56              if (wait(NULL) == -1) {
57                      perror("write");
58                      return -1;
59              }
60              return 0;
61 }
```

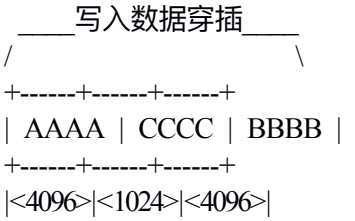
```
父进程
  3 * ---+
  4 * -+ |
       | +->读端文件表项(1) \
       |                    无名管道(2)
       +--->写端文件表项(1) /
----------------------------------------
父进程
| 3 * ---+
| 4 * -+ |
|      | +->读端文件表项(2) \
| fork | |                    无名管道(2)
v      +--->写端文件表项(2) /
子进程 | |
  3 * -|-+
  4 * -+
----------------------------------------
父进程
  3 NULL
  4 * -+
       | +->读端文件表项(1) \
       | |                  无名管道(2)
       +--->写端文件表项(1) /
子进程   |
  3 * ---+
  4 NULL
----------------------------------------
```



基于(有名和无名)管道的进程间通信所有的特殊情况：
1)试图读一个写文件描述符已被关闭的管道：只要管道中还有没来得及读取的数据，依然可以正常读取，一直读到管道中没有数据了，这时 read 函数会返回 0(既不是返回-1，也不是阻塞)，指示读到文件尾。
2)试图写一个读文件描述符已被关闭的管道：会直接触发 SIGPIPE(13)信号。该信号的默认操作是杀掉执行写操作的进程。如果实现已将该信号设置为忽略或捕获，则执行写操作的 write 调用会返回-1，且置 errno 为 EPIPE。
3)系统为每个管道都会维护一个内存缓冲区，其大小通常为 4096 个字节。如果写管道时发现缓冲区中的空闲空间不足以容纳此次 write 所要写入的字节数，则 write 函数会阻塞，直到缓冲区中的空闲空间变得足够大为止。
4)如果同时有多个进程向同一个管道写入数据，而每次调用 write 函数写入的字节数都不超过该管道缓冲区的大小，则这些 write 操作不会相互穿插，反之单次写入的字节数超过了该管道缓冲区的大小，则它们的 write 操作可能会相互穿插。







5)读一个写文件描述符打开但缓冲区中没有数据的管道，将导致阻塞，直到缓冲区中有别的进程写入的数据了，再读取这些数据并返回。
6)如果通过 fcntl 函数为表示管道的文件描述符追加了 O_NONBLOCK 状态标志位，那么前面所有发生阻塞的地方都不会再阻塞，而是直接返回失败(-1)，并置 errno 为 EAGAIN。

5.管道符号

Shell 通过管道符号"|"将前一个命令的输出作为后一个命令的输入。

```
ifconfig |                              # 列出网络配置
      grep inet |                       # 选出带有 inet 的字符串
           grep -v 127 |                # 选出不带 127 的字符串
                awk '{print $2}' |  # 打印每一行的第 2 个子串
                     cut -d ":" -f 2 # 按冒号切分取第 2 项
$ cmd1 | cmd2
Shell
 +->显式提示符
 |  读取命令行 <- cmd1 | cmd2
 |  调用 pipe 函数创建无名管道，获得其读写两端：pipefd[2]
 |  调用 vfork 函数创建子进程
 |       关闭管道的读端：close(pipefd[0])
 |       将管道的写端复制到标准输出：dup2(pipefd[1], 1)
 |       调用 exec 函数创建 cmd1 进程
 |            cmd1: printf -> pipefd[1] -> 管道 ---------+
 |  调用 vfork 函数创建子进程                                    |
 |       关闭管道的写端：close(pipefd[1])                       |
 |       将管道的读端复制到标准输入：dup2(pipefd[0], 0) |
 |       调用 exec 函数创建 cmd2 进程                            |
 |            cmd2:  scanf <- pipefd[0] <- 管道 <--------+
 +--调用 waitpid 回收两个子进程，获得其终止状态
```

代码：cmd1.c

```
1 #include <stdio.h>
2 int main(void) {
3           int x = 123, y = 456;
4           printf("%d %d\n", x, y);
5           return 0;
6 }
```

cmd2.c

```
1 #include <stdio.h>
2 int main(void) {
3           int x, y;
4           scanf("%d%d", &x, &y);
5           printf("%d+%d=%d\n", x, y, x + y);
6           return 0;
7 }
```

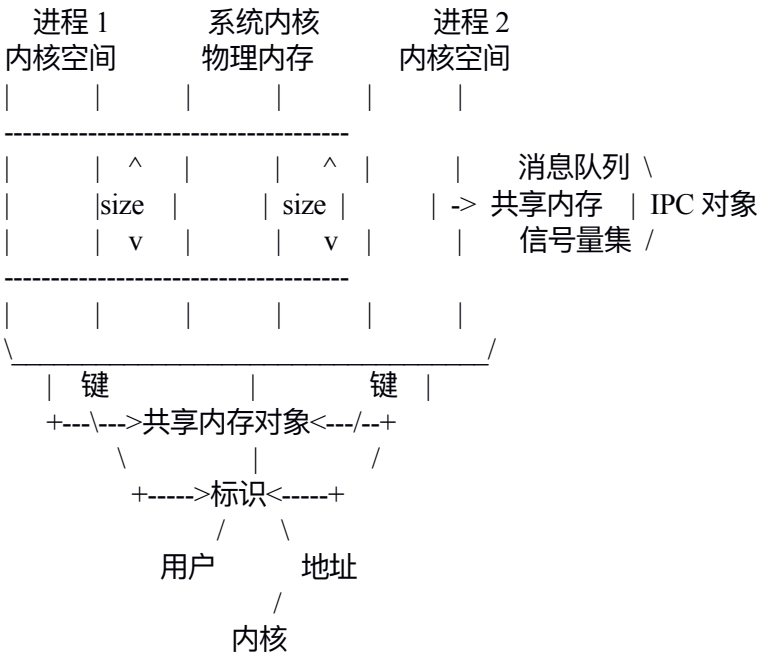
shell.c

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <sys/wait.h>
4 #include <errno.h>
5 int main(void) {
```

```
6       int pipefd[2];
7       if (pipe(pipefd) == -1) {
8               perror("pipe");
9               return -1;
10      }
11      pid_t pid = vfork();
12      if (pid == -1) {
13              perror("vfork");
14              return -1;
15      }
16      if (pid == 0) {
17              close(pipefd[0]);
18              if (dup2(pipefd[1], STDOUT_FILENO) == -1) {
19                      perror("dup2");
20                      _exit(-1);
21              }
22              if (execl("./cmd1", "cmd1", NULL) == -1) {
23                      perror("execl");
24                      _exit(-1);
25              }
26      }
27      if ((pid = vfork()) == -1) {
28              perror("vfork");
29              return -1;
30      }
31      if (pid == 0) {
32              close(pipefd[1]);
33              if (dup2(pipefd[0], STDIN_FILENO) == -1) {
34                      perror("dup2");
35                      _exit(-1);
36              }
37              if (execl("./cmd2", "cmd2", NULL) == -1) {
38                      perror("execl");
39                      _exit(-1);
40              }
41      }
42      close(pipefd[0]);
43      close(pipefd[1]);
44      for (;;)
45              if (wait(NULL) == -1) {
46                      if (errno != ECHILD) {
47                              perror("wait");
48                              return -1;
49                      }
50                      break;
51              }
52      return 0;
```

```
53 }
```

6.共享内存



1)生成 IPC 对象的键(消息队列，信号量集，共享内存)，负责管理内核

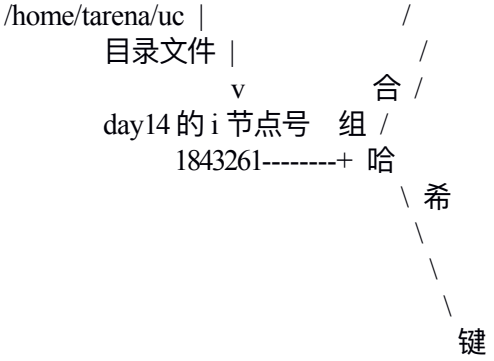
```
#include <sys/ipc.h>
key_t ftok(const char* pathname, int proj_id);
```

成功返回 IPC 对象的键，失败返回-1。

pathname: 一个真实存在的文件或目录的路径

proj_id: 项目 ID，仅低 8 位有效，取 0-255 之间的值

ftok("/home/tarena/uc/day14", 1000)



2)创建(新的)或获取(已有的)共享内存

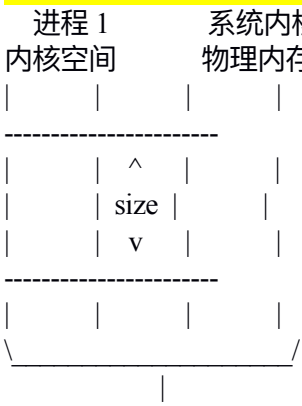
```
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflg);  //shm：share memory
```

成功返回共享内存对象的标识，失败返回-1。

key: 共享内存的键(名字)

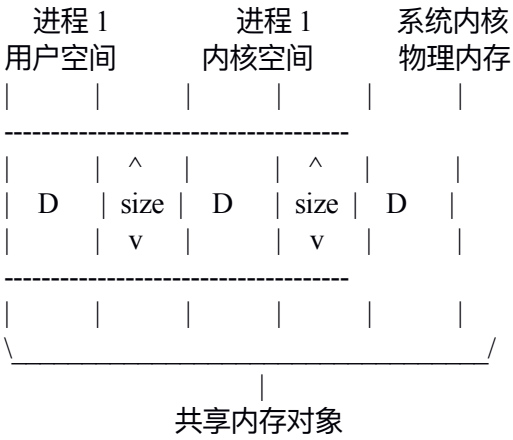
size: 共享内存的字节数，自动向上圆整为 4096 的整数倍。

若只为获取共享内存，该参数取 0 即可。
shmflg: [三位八进制整数表示的读写权限 |] 以下值：
0 - 纯获取，不存在即失败
IPC_CREAT - 创建或获取，不存在即创建，已存在即获取
IPC_EXCL - 排它，与 IPC_CREAT 结合，不存在即创建，已存在即失败
Shell 终端查看共享内存对象命令：ipcs -m



3)加载共享内存
#include <sys/shm.h>
void* shmat(int shmid, const void* shmaddr, int shmflg);
成功返回用户空间共享内存的起始地址，失败返回-1。
shmid: 共享内存对象的标识。
shmaddr: 用户空间共享内存的起始地址，可置 NULL，自动选择。
shmflg: 加载标志，可取以下值：
0 - 可读可写
SHM_RDONLY - 只读
SHM_RND - 只在 shmaddr 参数非 NULL 时起作用，
表示对该参数自动向下圆整至页边界

一旦获得了该函数所返回的用户空间空间共享内存的起始地址，就可以象访问普通内存一样访问共享内存中的数据。系统内核中的共享内存对象包含一个引用计数，每一个针对该共享内存的加载操作都会令其引用计数加一。



4)卸载共享内存

#include <sys/shm.h>
int shmdt(const void* shmaddr);
成功返回 0，失败返回-1。
shmaddr: 用户空间共享内存的起始地址。
每一个针对共享内存的卸载操作都会令其引用计数减一。
5)销毁共享内存
#include <sys/shm.h>
int shmctl(int shmid, IPC_RMID, NULL);
成功返回 0，失败返回-1。
shmid: 共享内存对象的标识。
所谓销毁共享内存，其实未必真的销毁，而只是做一个标记，禁止任何进程对该共享内存形成新的加载，但已有的加载依然保留。只有当使用者纷纷卸载，当其引用计数变为 0 时，才会真的被销毁。
Shell 终端删除共享内存命令：ipcrm -m shmid 号
6)编程模型

进程 1		进程 2
获得共享内存的键	ftok	获得共享内存的键
创建共享内存	shmget	获取共享内存
加载共享内存	shmat	加载共享内存
使用共享内存	strcpy/memcpy/sprintf *pointer	使用共享内存
卸载共享内存	shmdt	卸载共享内存
销毁共享内存	shmctl	

代码：wshm.c

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <sys/shm.h>
 4 int main(void) {
 5         printf("%d 进程：获得共享内存的键\n", getpid());
 6         key_t key = ftok(".", 100);
 7         if (key == -1) {
 8                 perror("ftok");
 9                 return -1;
10         }
11         printf("%d 进程：共享内存的键%#x\n", getpid(), key);
12         printf("%d 进程：创建共享内存\n", getpid());
13         int shmid = shmget(key, 4096, 0644 | IPC_CREAT | IPC_EXCL);
14         if (shmid == -1) {
15                 perror("shmget");
16                 return -1;
17         }
18         printf("%d 进程：共享内存标识%d\n", getpid(), shmid);
19         printf("%d 进程：加载共享内存\n", getpid());
20         void* shmaddr = shmat(shmid, NULL, 0);
21         if (shmaddr == (void*)-1) {
```

```
22                      perror("shmat");
23                      return -1;
24              }
25              printf("%d 进程：共享内存地址%p\n", getpid(), shmaddr);
26              printf("%d 进程：写入共享内存\n", getpid());
27              sprintf(shmaddr, "我是%d 进程写入共享内存的字符串", getpid());
28              printf("%d 进程：卸载共享内存\n", getpid());
29              if (shmdt(shmaddr) == -1) {
30                      perror("shmdt");
31                      return -1;
32              }
33              getchar();
34              printf("%d 进程：销毁共享内存\n", getpid());
35              if (shmctl(shmid, IPC_RMID, NULL) == -1) {
36                      perror("shmctl");
37                      return -1;
38              }
39              return 0;
40 }
```

rshm.c

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/shm.h>
4  int main(void) {
5               printf("%d 进程：获得共享内存的键\n", getpid());
6               key_t key = ftok(".", 100);
7               if (key == -1) {
8                       perror("ftok");
9                       return -1;
10              }
11              printf("%d 进程：共享内存的键%#x\n", getpid(), key);
12              printf("%d 进程：获取共享内存\n", getpid());
13              int shmid = shmget(key, 0, 0);
14              if (shmid == -1) {
15                      perror("shmget");
16                      return -1;
17              }
18              printf("%d 进程：共享内存标识%d\n", getpid(), shmid);
19              printf("%d 进程：加载共享内存\n", getpid());
20              void* shmaddr = shmat(shmid, NULL, 0);
21              if (shmaddr == (void*)-1) {
22                      perror("shmat");
23                      return -1;
24              }
25              printf("%d 进程：共享内存地址%p\n", getpid(), shmaddr);
26              printf("%d 进程：读取共享内存\n", getpid());
27              printf("%d 进程：读取到的内容\"%s\"\n", getpid(), (char*)shmaddr);
```

```
28              printf("%d 进程：卸载共享内存\n", getpid());
29              if (shmdt(shmaddr) == -1) {
30                      perror("shmdt");
31                      return -1;
32              }
33              return 0;
34 }
```

利用共享内存实现进程间通信，避免了在用户空间和内核空间的内存复制，因此效率最高，速度最快，但是共享内存缺乏必要同步性，参与数据交换的诸进程并不知道什么时候满足可读或可写的条件，往往还要依赖于其它 IPC 机制建立这种同步。因此共享内存主要用于在不同进程之间偶然交换大量数据。

7.消息队列

1)逻辑模型：单向线性链表队列

```
  消息      +->  消息      +->  ...
--------  |  --------  |
消息类型  |  消息类型  |
数据长度  |  数据长度  |
消息数据  |  消息数据  |
消息指针--+  消息指针--+
```

```
消息类型:1  消息类型:3  消息类型:1  消息类型:3
数据长度:1  数据长度:1  数据长度:1  数据长度:1
消息数据:A  消息数据:B  消息数据:C  消息数据:D
消息指针:*->消息指针:*->消息指针:*->消息指针:NULL
   |                                        |
  尾                 先入先出                首
   ^                 队列模型                |
   |                                         v
发送消息                                  接收消息
```

不区分类型：D-C-B-A

区分类型-3：D-B

区分类型-1：C-A

2)系统限制

最大可发送消息字节数 ： 8192(8K)字节

最大队列总消息数 ： 16384(16K)个消息

最大系统消息队列数 ： 16 个队列

最大系统总消息数 ： 262144(256K)个消息

3)生成 IPC 对象的键

4)创建(新的)或获取(已有的)消息队列

```
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
```

成功返回消息队列对象的标识，失败返回-1。

key: 消息队列的键(名字)

msgflg: [三位八进制整数表示的读写权限 |] 以下值：

0 - 纯获取，不存在即失败

IPC_CREAT - 创建或获取，不存在即创建，已存在即获取

IPC_EXCL - 排它，与 IPC_CREAT 结合，不存在即创建，
已存在即失败

5)发送消息

#include <sys/msg.h>
int msgsnd(int msgid, const void* msgp, size_t msgsz,int msgflg);
成功返回 0，失败返回-1。

msgid: 消息队列对象的标识。

```
          +----------+-----------------+
msgp -> | 消息类型 |       消息数据       |
          +----------+-----------------+
          |--4 字节--|-----  msgsz  ------|
```

msgflg: 发送标志，一般取 0。

如果系统内核中的消息未达上限，则 msgsnd 函数会将欲发送消息加入指定的消息队列并立即返回 0，否则该函数会阻塞，直到系统内核允许加入新消息为止(比如有消息因被接受而离开消息队列)。如果 msgflg 参数中包含 IPC_NOWAIT 标志位，则 msgsnd 函数在系统内核中的消息已达上限的情况下不会阻塞，而是返回-1，且置 errno 为 EAGAIN。

6)接收消息

#include <sys/msg.h>
ssize_t msgrcv(int msgid, void* msgp, size_t msgsz,long msgtyp, int msgflg);
成功返回所接收到的消息的字节数，失败返回-1。

msgid: 消息队列对象的标识。

```
                          _____返回值_____
                         /                \
          +----------+-----------------+----------+
msgp -> | 消息类型 |       消息数据       |          |
          +----------+-----------------+----------+
          |--- 4 ----|----------- msgsz -----------|
```

msgtyp: 消息类型，可取以下值：

正整数 - 获取特定类型的消息
0 - 不区分类型获取消息

msgflg: 接收标志，一般取 0。

若消息队列存在满足条件的消息，但其数据长度大于 msgsz 参数，且 msgflg 参数包含 MSG_NOERROR 标志位，则只截取该消息数据的前 msgsz 字节返回，剩余部分直接丢弃；但如果 msgflg 参数不包含 MSG_NOERROR 标志位，则不处理该消息，直接返回-1，同时置 errno 为 E2BIG。若消息队列中有可接收消息，则 msgrcv 函数会将该消息移出消息队列，并立即返回所接收到的消息数据的字节数，表示接收成功，否则此函数会阻塞，直到消息队列中有可接收消息为止。若 msgflg 参数中包含 IPC_NOWAIT 标志位，则 msgrcv 函数在消息队列中没有可接收消息的情况下不会阻塞，而是返回-1，且置 errno 为 EAGAIN。

5)销毁消息队列内存

#include <sys/msg.h>
int msgctl(int msgid, IPC_RMID, NULL);
成功返回 0，失败返回-1。

msgid: 消息队列对象的标识。

6)编程模型

```
        进程 1                                  进程 2
获得消息队列的键          ftok          获得消息队列的键
  创建消息队列            msgget          获取消息队列
```

发送接收消息　　msgsnd/msgrcv　　接收发送消息
销毁消息队列　　msgctl
消息队列采用链式存储，空间性能优于管道，但是其时间性能略逊于共享内存。总体性能相对中庸，应用最为广泛。消息队列具有天然的同步机制，即流量控制。

Shell 终端查看消息队列命令：ipcs -q

代码：wmsg.c

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/msg.h>
int main(void) {
        printf("%d 进程：获得消息队列的键\n", getpid());
        key_t key = ftok(".", 100);
        if (key == -1) {
                perror("ftok");
                return -1;
        }
        printf("%d 进程：消息队列的键%#x\n", getpid(), key);
        printf("%d 进程：创建消息队列\n", getpid());
        int msgid = msgget(key, 0644 | IPC_CREAT | IPC_EXCL);
        if (msgid == -1) {
                perror("msgget");
                return -1;
        }
        printf("%d 进程：消息队列标识%d\n", getpid(), msgid);
        printf("%d 进程：向消息队列发送数据...\n", getpid());
        for (;;) {
                printf("> ");
                struct {
                        long type;          // 消息类型
                        char data[1024]; // 消息数据
                }       msg = {1234, ""};
                fgets(msg.data, sizeof(msg.data) / sizeof(msg.data[0]), stdi      n);
                if (!strcmp(msg.data, "!\n"))
                        break;
                if (msgsnd(msgid, &msg, strlen(msg.data) * sizeof(msg.data[0]), 0) == -1)
        {
                        perror("msgsnd");
                        return -1;
                }
        }
        printf("%d 进程：销毁消息队列\n", getpid());
        if (msgctl(msgid, IPC_RMID, NULL) == -1) {
                perror("msgctl");
                return -1;
```

```
39              }
40              return 0;
41  }
```

rmsg.c

```
 1  #include <stdio.h>
 2  #include <string.h>
 3  #include <unistd.h>
 4  #include <errno.h>
 5  #include <sys/msg.h>
 6  int main(void) {
 7              printf("%d 进程：获得消息队列的键\n", getpid());
 8              key_t key = ftok(".", 100);
 9              if (key == -1) {
10                      perror("ftok");
11                      return -1;
12              }
13              printf("%d 进程：消息队列的键%#x\n", getpid(), key);
14              printf("%d 进程：获取消息队列\n", getpid());
15              int msgid = msgget(key, 0);
16              if (msgid == -1) {
17                      perror("msgget");
18                      return -1;
19              }
20              printf("%d 进程：消息队列标识%d\n", getpid(), msgid);
21              printf("%d 进程：从消息队列接收数据...\n", getpid());
22              for (;;) {
23                      struct {
24                              long type;          // 消息类型
25                              char data[1024]; // 消息数据
26                      }       msg = {};
27                      ssize_t msgsz = msgrcv(msgid,
                                &msg, sizeof(msg.data) - sizeo      f(msg.data[0]),
28                              1234, MSG_NOERROR/* | IPC_NOWAIT*/);
29                      if (msgsz == -1)
30                              if (errno == EIDRM) {
31                                      printf("%d 进程：消息队列已被销毁\n", getpid(      ));
32                                      break;
33                              }
34                              else if (errno == ENOMSG) {
35                                  printf("%d 进程：暂时没有消息，空闲处理\n", g      etpid());
36                                      sleep(1);
37                              }
38                              else {
39                                      perror("msgrcv");
40                                      return -1;
41                              }
```

```
 42                          else
 43                                  printf("%04ld 字节< %s", msgsz, msg.data);
 44              }
 45              return 0;
 46 }
```

8.信号量集

1)信号量并非用于进程间的数据交换，而是用于在多个进程中分享有限的资源。使用资源的进程多于资源总数，但是每个进程使用(持有)资源的时间是有限的，当一个进程获得某个资源时，其它进程就无法再获得该资源，直到拥有该资源的进程主动放弃其使用权，其它进程才有机会竞争该资源。理想情况下，每个进程获得资源的概率是均等的。

```
+----------------+
|       图书馆       |
|------------+---+
|《三国演义》| 0 |
+------------+---+
                    |
                  信号量
                资源计数器
          表示可分配资源的数量
----------+----+----+----->
借阅者 1    x      o      o
借阅者 2    o      -      o
借阅者 3    x      x      o
借阅者 4    o      -      x
借阅者 5    o      -      x
借阅者 6    x      o      o
借阅者 7    o      o      -
借阅者 8    x      o      o
借阅者 9    o      o      -
```

2)为了获得共享资源，进程需按以下步骤执行：

A.测试控制该资源的信号量；

B.若信号量的值大于 0，说明还有可分配资源，则进程获得该资源，同时将信号量的值减 1，表示可分配资源少了一个；

C.若信号量的值等于 0，说明没有可分配资源，则进程进入睡眠态，

直到信号量的值再度大于 0，这时处于睡眠状态的进程中会有一个被系统内核唤醒，它将返回执行步骤 A，而其它未被唤醒的进程则继续在睡眠中等待；

D.当进程不再使用所获得的资源时，应将控制该资源的信号量的值加 1，表示可分配资源多了一个。此时那些正在睡眠中等待资源的进程中的一个将被系统内环唤醒。

```
+----------------+
|       图书馆       |
|------------+---+
|《三国演义》| 5 | \ 每个信号量表示一种资
|《水 浒 传》| 5 | \ 源的可分配数，多个信
|《红 楼 梦》| 5 | / 号量组成的信号量集表
|《西 游 记》| 5 | / 示多种资源的可分配数
+------------+---+
```

3)生成 IPC 对象的键
4)创建(新的)或获取(已有的)信号量集
```
#include <sys/sem.h>
int semget(key_t key, int nsems, int semflg);
```
成功返回信号量集对象的标识，失败返回-1。
key: 信号量集的键(名字)
msgflg: [三位八进制整数表示的读写权限 |] 以下值:
0 - 纯获取，不存在即失败
IPC_CREAT - 创建或获取，不存在即创建，已存在即获取
IPC_EXCL - 排它，与 IPC_CREAT 结合，不存在即创建，已存在即失败
5)操作信号量集
```
#include <sys/sem.h>
int semop(int semid, struct sembuf* sops, size_t nsops);
```
成功返回 0，失败返回-1。
semid: 信号量集对象的标识
sops: 操作结构数组
nsops: 操作结构数组的长度，即数组中操作结构的个数
```
struct sembuf {
    unsigned short sem_num; // 信号量在信号量集中的下标
    short          sem_op;  // 操作数
    short          sem_flg; // 操作标志
};
```
sops 指针所指向操作结构数组中的每个元素通过其 sem_num 字段与信号量集中的一个特定的信号量相对应，表示对该信号量的操作。semop 函数对 sops 指向的包含 nsops 个元素的操作结构数组中的每个元素执行如下操作:
A.若 sem_op 大于 0，则将其加到 semid 信号量集第 sem_num 个信号量的值上，以表示对资源的释放;
B.若 sem_op 小于 0，则从 semid 信号量集第 sem_num 个信号量的值中减去其绝对值，以表示对资源的获取。如果不够减(信号量的值不能为负)，则此函数会阻塞，直到够减为止，以表示对资源的等待。如果 sem_flg 包含 IPC_NOWAIT 位，则即使不够减也不会阻塞，而是返回-1，同时置 errno 为 EAGAIN，以便在等待资源时还可以做其它处理;
C.若 sem_op 等于 0,则直到 semid 信号量集第 sem_num 个信号量的值为 0 时才返回,除非 sem_flg 含 IPC_NOWAIT 位。
```
semid -semop(semid, sops, nsops)-> semid
                  _________|          |
0: 15             |                   |       0: 14
1: 21             +-->{3, -1, 0}\ |       1: 21
2: 33                 {0, -1, 0} >3       2: 34
3: 42                 {2,  1, 0}/         3: 41
```
6)销毁信号量集
```
int semctl(int semid, 0, IPC_RMID);
```
成功返回 0，失败返回-1。
semid: 信号量集对象的标识。
7)控制信号量集
A.整体设值
```
unsigned short array[信号量数] = {
```

信号量 1 的值, 信号量 2 的值, ...};
semctl(semid, 0, SETALL, array) // array->semid
B.整体取值
unsigned short array[信号量数];
semctl(semid, 0, GETALL, array) // semid->array
C.具体设值
semctl(semid, 信号量下标, SETVAL, 信号量的值);
D.具体取值
semctl(semid, 信号量下标, GETVAL);->信号量的值
8)编程模型

进程 1		进程 2
获得信号量集的键	ftok	获得信号量集的键
创建信号量集	semget	获取信号量集
初始化信号量集	semctl+SETALL	
操作信号量集	semop	操作信号量集
销毁信号量集	semctl+IPC_RMID	

代码：csem.c

```
 1 #include <stdio.h>
 2 #include <errno.h>
 3 #include <sys/sem.h>
 4 int pmenu(void) {
 5         printf("-----------------\n");
 6         printf("       迷你图书馆\n");
 7         printf("-----------------\n");
 8         printf("[1] 借《三国演义》\n");
 9         printf("[2] 还《三国演义》\n");
10         printf("[3] 借《水浒传》\n");
11         printf("[4] 还《水浒传》\n");
12         printf("[5] 借《红楼梦》\n");
13         printf("[6] 还《红楼梦》\n");
14         printf("[7] 借《西游记》\n");
15         printf("[8] 还《西游记》\n");
16         printf("[0] 退出\n");
17         printf("-----------------\n");
18         printf("请选择：");
19         int sel = -1;
20         scanf("%d", &sel);
21         return sel;
22 }
23 int pleft(int semid, unsigned short semnum) {
24         int val = semctl(semid, semnum, GETVAL);
25         if (val == -1) {
26                 perror("semctl");
27                 return -1;
28         }
29         printf("还剩%d 册。\n", val);
```

```
30              return 0;
31  }
32  int borrow(int semid, unsigned short semnum) {
33              struct sembuf sops = {semnum, -1, /*0*/IPC_NOWAIT};
34              if (semop(semid, &sops, 1) == -1) {
35                      if (errno != EAGAIN) {
36                              perror("semop");
37                              return -1;
38                      }
39                      printf("暂时没书。\n");
40                      return 0;
41              }
42              printf("借阅成功。\n");
43              return pleft(semid, semnum);
44  }
45  int revert(int semid, unsigned short semnum) {
46              struct sembuf sops = {semnum, 1, 0};
47              if (semop(semid, &sops, 1) == -1) {
48                      perror("semop");
49                      return -1;
50              }
51              printf("归还成功。\n");
52              return pleft(semid, semnum);
53  }
54  int main(void) {
55              key_t key = ftok(".", 100);
56              if (key == -1) {
57                      perror("ftok");
58                      return -1;
59              }
60              int semid = semget(key, 4, 0644 | IPC_CREAT | IPC_EXCL);
61              if (semid == -1) {
62                      perror("semget");
63                      return -1;
64              }
65              unsigned short semarr[] = {5, 5, 5, 5};
66              if (semctl(semid, 0, SETALL, semarr) == -1) {
67                      perror("semctl");
68                      return -1;
69              }
70              int quit = 0;
71              while (!quit) {
72                      int sel = pmenu();
73                      switch (sel) {
74                              case 0:
75                                      quit = 1;
76                                      break;
```

```
 77                                  case 1: // 借《三国演义》-> 0 -+
 78                                  case 3: // 借《水浒传》   -> 1  | 对信号量
 79                                  case 5: // 借《红楼梦》   -> 2  | 做-1 操作
 80                                  case 7: // 借《西游记》   -> 3 -+
 81                                          if (borrow(semid, sel / 2) == -1)
 82                                                  return -1;
 83                                          break;
 84                                  case 2: // 还《三国演义》-> 0 -+
 85                                  case 4: // 还《水浒传》   -> 1  | 对信号量
 86                                  case 6: // 还《红楼梦》   -> 2  | 做+1 操作
 87                                  case 8: // 还《西游记》   -> 3 -+
 88                                          if (revert(semid, (sel - 1) / 2) == -1)
 89                                                  return -1;
 90                                          break;
 91                                  default:
 92                                          printf("无效选择！\n");
 93                                          scanf("%*[^\n]");
 94                                          scanf("%*c");
 95                                          break;
 96                          }
 97                  }
 98                  if (semctl(semid, 0, IPC_RMID) == -1) {
 99                          perror("semctl");
100                          return -1;
101                  }
102                  return 0;
103  }
```

gsem.c

```
 1  #include <stdio.h>
 2  #include <errno.h>
 3  #include <sys/sem.h>
 4  int pmenu(void) {
 5          printf("-----------------\n");
 6          printf("      迷你图书馆\n");
 7          printf("-----------------\n");
 8          printf("[1] 借《三国演义》\n");
 9          printf("[2] 还《三国演义》\n");
10          printf("[3] 借《水浒传》\n");
11          printf("[4] 还《水浒传》\n");
12          printf("[5] 借《红楼梦》\n");
13          printf("[6] 还《红楼梦》\n");
14          printf("[7] 借《西游记》\n");
15          printf("[8] 还《西游记》\n");
16          printf("[0] 退出\n");
17          printf("-----------------\n");
18          printf("请选择：");
```

```
19              int sel = -1;
20              scanf("%d", &sel);
21              return sel;
22  }
23  int pleft(int semid, unsigned short semnum) {
24              int val = semctl(semid, semnum, GETVAL);
25              if (val == -1) {
26                      perror("semctl");
27                      return -1;
28              }
29              printf("还剩%d 册。\n", val);
30              return 0;
31  }
32  int borrow(int semid, unsigned short semnum) {
33              struct sembuf sops = {semnum, -1, /*0*/IPC_NOWAIT};
34              if (semop(semid, &sops, 1) == -1) {
35                      if (errno != EAGAIN) {
36                              perror("semop");
37                              return -1;
38                      }
39                      printf("暂时没书。\n");
40                      return 0;
41              }
42              printf("借阅成功。\n");
43              return pleft(semid, semnum);
44  }
45  int revert(int semid, unsigned short semnum) {
46              struct sembuf sops = {semnum, 1, 0};
47              if (semop(semid, &sops, 1) == -1) {
48                      perror("semop");
49                      return -1;
50              }
51              printf("归还成功。\n");
52              return pleft(semid, semnum);
53  }
54  int main(void) {
55              key_t key = ftok(".", 100);
56              if (key == -1) {
57                      perror("ftok");
58                      return -1;
59              }
60              int semid = semget(key, 0, 0);
61              if (semid == -1) {
62                      perror("semget");
63                      return -1;
64              }
65              unsigned short semarr[] = {5, 5, 5, 5};
```

```
66          if (semctl(semid, 0, SETALL, semarr) == -1) {
67                  perror("semctl");
68                  return -1;
69          }
70          int quit = 0;
71          while (!quit) {
72                  int sel = pmenu();
73                  switch (sel) {
74                          case 0:
75                                  quit = 1;
76                                  break;
77                          case 1: // 借《三国演义》-> 0 -+
78                          case 3: // 借《水浒传》   -> 1  | 对信号量
79                          case 5: // 借《红楼梦》   -> 2  | 做-1 操作
80                          case 7: // 借《西游记》   -> 3 -+
81                                  if (borrow(semid, sel / 2) == -1)
82                                          return -1;
83                                  break;
84                          case 2: // 还《三国演义》-> 0 -+
85                          case 4: // 还《水浒传》   -> 1  | 对信号量
86                          case 6: // 还《红楼梦》   -> 2  | 做+1 操作
87                          case 8: // 还《西游记》   -> 3 -+
88                                  if (revert(semid, (sel - 1) / 2) == -1)
89                                          return -1;
90                                  break;
91                          default:
92                                  printf("无效选择！\n");
93                                  scanf("%*[^\n]");
94                                  scanf("%*c");
95                                  break;
96                  }
97          }
98          return 0;
99  }
```

9.IPC 命令

1)查看 IPC 对象

ipcs -m (memory, 共享内存)

ipcs -q (queue, 消息队列)

ipcs -s (semphore, 信号量)

ipcs -a (all, 所有的)

2)删除 IPC 对象

ipcrm -m <共享内存对象的标识>

ipcrm -q <消息队列对象的标识>

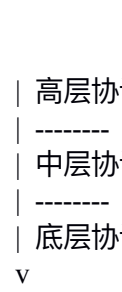
ipcrm -s <信号量集对象的标识>

十一、网络通信

1.网络与网络协议

1)网络协议模型————ISO/OSI 网络协议模型

A.什么是计算机网络?
计算机网络是指将地理位置不同的具有独立功能的多台计算机或具有计算能力的设备，通过通信线路或电磁场连接起来，在网络操作系统、网络管理软件及网络通信协议的管理和协调下，实现资源共享和信息传递的计算机系统。
主体：多台功能独立的计算设备
介质：通信线路或电磁场
软件：网络操作系统、网络管理软件、网络通信协议
功能：资源共享、信息传递
B.什么是网络通信协议?
网络协议的本质就是规则，即各种硬件和软件都必须遵守的共同守则。网络协议本身并不是一套单独的软件，它融合于其它所有和网络通信有关的软件和硬件之中，因此可以说网络协议就是网络通信的标准，渗透到网络中的方方面面。
C.什么是网络协议栈?
大多数网络协议都采用了分层设计的方法。所谓分层设计，就是按照信息的流动过程将网络的整体功能分解为一个个相对独立的功能层，不同机器上的同等功能层之间采用相同的协议，同一机器上相邻功能层之间通过接口进行信息传递。各层的协议和接口统称为协议栈。



D.ISO/OSI 网络协议模型
开放系统互联(Open System Interconnection)
国际标准化组织(International Standardization Organization)

应用层	业务逻辑，如：http、ftp、telnet
表示层	数据格式，如：ASCII、JPEG、MPEG
会话层	互访顺序
传输层	从端到端，如：TCP、UDP
网络层	网中找路，如：IP、SPX、路由协议
数据链路层	物理寻址，如：MAC、网桥协议
物理层	硬件驱动

应用层 > 应用层

2)TCP/IP 协议族
A.传输层协议
传输控制协议：TCP
用户数据报协议：UDP
B.网络层协议
网际控制消息协议：ICMP[v4]、ICMPv6
网际组管理协议：IGMP
网际协议：IP[v4]、IPv6

C.数据链路层协议
地址解析协议：ARP
逆地址解析协议：RARP

以太网头 - 数据链路层 \
-------- |
IP 头 - 网络层 \ |
-------- | | 以太网帧 <-> 物理层
TCP 头 - 传输层 \ | IP 包 |
-------- | TCP 包| |
HTTP 包头 \ 应用层 | | |
HTTP 包体 / 用户数据 / / |
-------- |
以太网尾 - 数据链路层 /

3)IP 地址
文件系统中的文件
/ \
i 节点号----路径
目录

网络中的计算机
/ | \
MAC 地址---IP 地址---域名
ARP DNS

IPv4 中的 IP 地址是计算机在互联网中的唯一标识，网络(大端)字节序 32 位无符号整数。

0x12345678
L H
|0x78|0x56|0x34|0x12| - 小端字节序，低位低址
L H
|0x12|0x34|0x56|0x78| - 大端字节序，即网络字节序，低位高址

Intel: x86/amd64，小端字节序
ARM: 大端字节序

小端机器---------- >大端机器
0x12345678
|0x78|0x56|0x34|0x12| - 主机序 - |0x78|0x56|0x34|0x12|
0x78563412

0x12345678
|0x78|0x56|0x34|0x12| - 主机序
|0x12|0x34|0x56|0x78| - 网络序 - |0x12|0x34|0x56|0x78|
主机序 - |0x12|0x34|0x56|0x78|
0x12345678

十六进制整数形式的 IP 地址：0x01020304
点分十进制字符串形式的 IP 地址：1.2.3.4
2.编程接口
1)套接字
进程表项
  文件描述符表
  0 文件描述符标志 | * -> 文件表项 -> v 节点 -> i 节点信息
  1 文件描述符标志 | * -> 文件表项 -> v 节点 -> i 节点信息
  2 文件描述符标志 | * -> 文件表项 -> v 节点 -> i 节点信息
  ...
  7 文件描述符标志 | * -> 文件表项 -> 套接字对象 -> 网络
  ^
  |
  +--套接字文件描述符
文件 I/O：
        应用程序   - 业务
            | read/write
磁盘文件的文件描述符 - 接口
            |
        文件系统   - 内核
            |
        磁盘文件   - 硬件

网络 I/O：
        应用程序   - 业务
            | read/write
 套接字的文件描述符   - 接口
            |
       网络协议栈   - 内核
            |
        网络设备   - 硬件
如同读写磁盘文件一样，收发网络数据。
2)创建套接字
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
成功返回套接字文件描述符，失败返回-1。
domain - 通信域，即协议族，可取以下值：
            AF_LOCAL/AF_UNIX - 本地通信，即进程间通信
            AF_INET   - 基于 IPv4 协议的网络通信
            AF_INET6   - 基于 IPv6 协议的网络通信
            AF_PACKET   - 面向数据链路层或物理层的通信
type - 套接字类型，可取以下值：
        SOCK_STREAM - 流式套接字，基于 TCP 传输层协议
                      可靠：不丢包、不重复、不乱序
                      速度慢
        SOCK_DGRAM - 数据报套接字，基于 UDP 传输层协议

不可靠：丢包、重复、乱序
速度快
SOCK_RAW - 原始套接字，基于 IP 及其底层协议
protocol - 特殊协议，对于流式和数据报套接字，取 0 即可。
3)将套接字和由 IP 地址及端口号描述的网络设备绑定

```
 ftp  telnet  web
+-|--+--|--+--|-+
| 21 | 23   | 80 |
+----+-----+----+
|192.168.225.139|
+---------------+
```

```
#include <netinet/in.h>
struct sockaddr { ... }; // 仅用于函数传参
struct sockaddr_in {
    sa_family_t     sin_family; // 地址族(AF_INET)
    in_port_t       sin_port;   // 端口号(网络字节序)
    struct in_addr  sin_addr;   // IP 地址(网络字节序)
};
struct in_addr {
    in_addr_t s_addr;(INADDR_ANY)//绑定任意 IP 与 domian 参数选取的协议族填写一致
};
typedef uint16_t in_port_t; // unsigned short
typedef uint32_t in_addr_t; // unsigned long
```

所谓绑定就是将套接字逻辑对象与用 sockaddr_in 结构所描述的 IP 地址和端口号表示的物理设备建立关联。

```
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr* addr, socklen_t addrlen);
```

成功返回 0，失败返回-1。
sockfd: 套接字文件描述符
addr: 自己的地址结构
addrlen: 地址结构的字节数
主机 A
应用程序<->套接字-bind-网络设备(IP+端口)
4)连接本机的网络设备与对方的网络设备

```
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr* addr, socklen_t addrlen);
```

成功返回 0，失败返回-1。
sockfd: 套接字文件描述符
addr: 对方的地址结构
addrlen: 地址结构的字节数
主机 A
应用程序<->套接字-bind-网络设备(IP+端口
主机 B | connect
应用程序<->套接字-bind-网络设备(IP+端口)
5)发送数据

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

或
#include <sys/socket.h>
ssize_t send(int sockfd, const void* buf, size_t count, int flags);
flags: 发送标志，取 0 等价于 write。可取以下值：
MSG_DONTWAIT - 非阻塞
MSG_OOB - 带外数据
MSG_DONTROUTE - 不路由
6)接收数据
#include <unistd.h>
ssize_t read(int sockfd, void* buf, size_t len);
或
#include <sys/socket.h>
ssize_t recv(int sockfd, void* buf, size_t len, int flags);
flags: 接收标志，取 0 等价于 read。可取以下值：
MSG_DONTWAIT - 非阻塞
MSG_OOB - 带外数据
MSG_PEEK - 不从接收缓冲区中删除数据
MSG_WAITALL - 等待收满 len 个字节再返回
3.基于流式(TCP 协议)套接字通信的特殊性
1)TCP 连接的三路握手
客户机 服务器
| |
+----------SYN(N)->+第一路
| |
+<-ACK(N+1)/SYN(M)--+第二路
| |
+---------ACK(M+1)->+第三路
| |
2)侦听套接字
在已经完成绑定的套接字上启动侦听，使之成为被动套接字，可以感知远程主机的连接请求，即三路握手中的第一路。
#include <sys/socket.h>
int listen(int sockfd, int backlog);
成功返回 0，失败返回-1。
sockfd: 已经完成绑定的套接字描述符。
backlog: 未决连接请求队列最大长度，取 1024 以上的值。
max: backlog
|--------------------|
客户机 3 客户机 4
/ 连接请求 3 <- 连接请求 4 - 未决连接请求队列
服务器 <
\ 连接请求 2 -> 连接请求 1 - 已决连接请求队列
客户机 2 客户机 1
3)等待并接受连接请求
侦听套接字
|
int accept(int sockfd, struct sockaddr* addr,

socklen_t* addrlen);
成功返回连接套接字描述符，失败返回-1。
^^^^^^^^^^
|
用于后续通信
sockfd: 侦听套接字描述符。
addr: 输出连接请求发起方的地址信息。
addrlen: 输入/输出连接请求发起方地址结构的字节。

客户机 服务器
socket socket
主动打开 bind
listen
被动打开
---
accept
阻塞
connect----------SYN(N)->将连接请求
阻塞 压入未决连
connect<-ACK(N+1)+SYN(M)--接请求队列
返回 ---------ACK(M+1)->将连接请求从未决
队列移入已决队列
accept
返回连接套接字

主动套接字<--read/write 等-->连接套接字
4)TCP 服务器
A.迭代式服务器
处理完一个客户机的业务再处理下一个客户机的业务

创建套接字(socket)
准备自己的地址结构并绑定(bind)
启动侦听(listen)
+->等待并接受客户机的连接请求(accept)
| <-发起连接请求(connect)
|+->在连接套接字上接收来自客户机的业务请求(read/recv)
|| <-发送业务请求|(write/send)
|| 业务处理
|+--在连接套接字上向客户机发送业务响应(write/send)
| ->接收业务响应(read/recv)
+--业务处理结束(终止报文或客户机断开连接)
B.并发式服务器
“同时”处理多个客户机的业务

创建套接字(socket)
准备自己的地址结构并绑定(bind)
启动侦听(listen)

```
+->等待并接受客户机的连接请求(accept)
|                                    <-发起连接请求(connect)
+--创建子进程(fork)
      +->在连接套接字上接收来自客户机的业务请求(read/recv)
      |                              <-发送业务请求|(write/send)
      |  业务处理
      +--在连接套接字上向客户机发送业务响应(write/send)
                                      ->接收业务响应(read/recv)
      业务处理结束(终止报文或客户机断开连接)，子进程终止
```

Shell 终端命令查看网卡地址：ifconfig
本地网卡：127.0.0.1

代码：tcpsvr.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <unistd.h>
 4 #include <signal.h>
 5 #include <wait.h>
 6 #include <errno.h>
 7 #include <sys/socket.h>
 8 #include <netinet/in.h>
 9 #include <arpa/inet.h>
10 void sigchld(int signum) {
11         for (;;) {
12                 pid_t pid = waitpid(-1, NULL, WNOHANG);
13                 if (pid == -1) {
14                         if (errno != ECHILD) {
15                                 perror("waitpid");
16                                 exit(-1);
17                         }
18                         printf("服务器：全部子进程都已退出\n");
19                         break;
20                 }
21                 if (pid)
22                         printf("服务器：发现%d 子进程退出\n", pid);
23                 else {
24                         printf("服务器：暂时没有子进程退出\n");
25                         break;
26                 }
27         }
28 }
29 int do_client(int connfd) {
30         // 业务循环：每循环一次处理一个业务请求
31         for (;;) {
32                 printf("%d: 接受请求\n", getpid());
33                 char buf[1024];
```

```
34                      ssize_t rb = recv(connfd, buf, sizeof(buf), 0);
35                      if (rb == -1) {
36                              perror("recv");
37                              return -1;
38                      }
39                      if (rb == 0) {
40                              printf("%d: 客户机已关闭\n", getpid());
41                              break;
42                      }
43                      printf("%d: 发送响应\n", getpid());
44                      if (send(connfd, buf, rb, 0) == -1) {
45                              perror("send");
46                              return -1;
47                      }
48              }
49              return 0;
50  }
51  int main(int argc, char* argv[]) {
52              if (argc < 2) {
53                      fprintf(stderr, "用法：%s <端口号>\n", argv[0]);
54                      return -1;
55              }
56              if (signal(SIGCHLD, sigchld) == SIG_ERR) {
57                      perror("signal");
58                      return -1;
59              }
60              printf("服务器：创建套接字\n");
61              int sockfd = socket(PF_INET, SOCK_STREAM, 0);
62              if (sockfd == -1) {
63                      perror("socket");
64                      return -1;
65              }
66              printf("服务器：准备地址并绑定\n");
67              struct sockaddr_in addr;
68              addr.sin_family = AF_INET;
69              addr.sin_port = htons(atoi(argv[1])); // 将 16 位整数从主机字节序转成网络字节序
home(net) to net(home) short(long)
70              addr.sin_addr.s_addr = INADDR_ANY;
71              if (bind(sockfd, (struct sockaddr*)&addr, sizeof(addr)) == -1) {
72                      perror("bind");
73                      return -1;
74              }
75              printf("服务器：侦听套接字\n");
76              if (listen(sockfd, 1024) == -1) {
77                      perror("listen");
78                      return -1;
79              }
```

```
80          // 连接循环：每循环一次处理一个客户机的连接
81          for (;;) {
82                  printf("服务器：等待连接\n");
83                  struct sockaddr_in addrcli = {};
84                  socklen_t addrlen = sizeof(addrcli);
85                  int connfd = accept(sockfd, (struct sockaddr*)&addrcli, &addrlen);
86                  if (connfd == -1) {
87                          perror("accept");
88                          return -1;
89                  }
90                  printf("服务器：客户机%s:%hu\n",
91                          inet_ntoa(addrcli.sin_addr), // 将整数形式的 IP 地址转成点分十
进制字符串 n:nomber a:ASCII
92                          ntohs(addrcli.sin_port));       // 将 16 位整数从网络字节序转成主
机字节序
93                  pid_t pid = fork();
94                  if (pid == -1) {
95                          perror("fork");
96                          return -1;
97                  }
98                  if (pid == 0) {
99                          printf("%d: 关闭侦听套接字\n", getpid());
100                         if (close(sockfd) == -1) {
101                                 perror("close");
102                                 return -1;
103                         }
104                         printf("%d: 处理客户机业务\n", getpid());
105                         if (do_client(connfd) == -1)
106                                 return -1;
107                         printf("%d: 关闭连接套接字\n", getpid());
108                         if (close(connfd) == -1) {
109                                 perror("close");
110                                 return -1;
111                         }
112                         printf("%d: 子进程退出\n", getpid());
113                         return 0;
114                 }
115         printf("服务器：关闭连接套接字\n");
116         if (close(connfd) == -1) {
117                 perror("close");
118                 return -1;
119                 }
120          }
121          printf("服务器：关闭侦听套接字\n");
122          if (close(sockfd) == -1) {
123                  perror("close");
124                  return -1;
```

```
125             }
126             return 0;
127 }
```

tcpcli.c

```
  1 #include <stdio.h>
  2 #include <stdlib.h>
  3 #include <string.h>
  4 #include <unistd.h>
  5 #include <errno.h>
  6 #include <sys/socket.h>
  7 #include <netinet/in.h>
  8 #include <arpa/inet.h>
  9 int main(int argc, char* argv[]) {
 10             if (argc < 3) {
 11                     fprintf(stderr, "用法：%s <IP 地址> <端口号>\n", argv[0]);
 12                     return -1;
 13             }
 14             printf("客户机：创建套接字\n");
 15             int sockfd = socket(PF_INET, SOCK_STREAM, 0);
 16             if (sockfd == -1) {
 17                     perror("socket");
 18                     return -1;
 19             }
 20             printf("客户机：准备地址并连接\n");
 21             struct sockaddr_in addr;
 22             addr.sin_family = AF_INET;
 23             addr.sin_port = htons(atoi(argv[2]));
 24             addr.sin_addr.s_addr = inet_addr(argv[1]);
 25             if (connect(sockfd, (struct sockaddr*)&addr, sizeof(addr)) == -1) {
 26                     perror("connect");
 27                     return -1;
 28             }
 29             for (;;) {
 30                     printf("> ");
 31                     char buf[1024];
 32                     fgets(buf, sizeof(buf) / sizeof(buf[0]), stdin);
 33                     if (!strcmp(buf, "!\n"))
 34                             break;
 35                     printf("客户机：发送请求\n");
 36                     if (send(sockfd, buf, strlen(buf) * sizeof(buf[0]), 0) == -1) {
 37                             perror("send");
 38                             return -1;
 39                     }
 40                     printf("客户机：接收响应\n");
 41                     ssize_t rb = recv(sockfd, buf, sizeof(buf) - sizeof(buf[0]), 0);
```

```
42                      if (rb == -1) {
43                              perror("recv");
44                              return -1;
45                      }
46                      buf[rb / sizeof(buf[0])] = '\0';
47                      printf("< %s", buf);
48              }
49              printf("客户机：关闭套接字\n");
50              if (close(sockfd) == -1) {
51                      perror("close");
52                      return -1;
53              }
54              return 0;
55      }
```

```
C/S: 客户机/服务器 _________C/C++/Java/Python_________
          |       |  /                                    \
          |       +--接收请求->业务处理->调整格式->发送响应
          |              \_______                _________/
          |                      \              /
          +--输入->调整格式->发送请求->接收响应->输出
              \__________C/C++/Java/Python__________/
```

```
B/S: 浏览器/服务器 _________C/C++/Java/Python_________
          |       |  /                                    \
          |       +--接收请求->业务处理->调整格式->发送响应
          |              \_______                _________/
          |                      \              /
          +--输入->调整格式->发送请求->接收响应->输出
              \________HTML5/JavaScript/前端________/
```

4.基于数据报(UDP 协议)套接字通信的特殊性

无连接、无三路握手、无侦听、无等待、无并发。

```
            / 子进程 - 套接字 - 客户机
TCP 服务器 - 子进程 - 套接字 - 客户机
            \ 子进程 - 套接字 - 客户机
                      / 客户机
UDP 服务器 - 套接字 - 客户机
                      \ 客户机
```

```
#include <sys/socket.h>
ssize_t sendto(int sockfd, const void* buf, size_t count, int flags, const struct sockaddr* dest_addr,
                socklen_t addrlen);
```

dest_addr: 接收者地址结构

addrlen: 接收者地址结构的字节数

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

```
ssize_t recvfrom(int sockfd, void* buf, size_t len, int flags, struct sockaddr* src_addr, socklen_t*
                addrlen);
```
src_addr: 输出发送者地址结构
addrlen: 输入/输出发送者地址结构的字节数

代码：udpsvr.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <unistd.h>
 4 #include <sys/socket.h>
 5 #include <netinet/in.h>
 6 #include <arpa/inet.h>
 7 int main(int argc, char* argv[]) {
 8         if (argc < 2) {
 9                 fprintf(stderr, "用法：%s <端口号>\n", argv[0]);
10                 return -1;
11         }
12         printf("服务器：创建套接字\n");
13         int sockfd = socket(PF_INET, SOCK_DGRAM, 0);
14         if (sockfd == -1) {
15                 perror("socket");
16                 return -1;
17         }
18         printf("服务器：准备地址并绑定\n");
19         struct sockaddr_in addr;
20         addr.sin_family = AF_INET;
21         addr.sin_port = htons(atoi(argv[1]));//将 16 位整数从主机字节序转成网络字节序
22         addr.sin_addr.s_addr = INADDR_ANY;
23         if (bind(sockfd, (struct sockaddr*)&addr, sizeof(addr)) == -1) {
24                 perror("bind");
25                 return -1;
26         }
27         // 连接循环：每循环一次处理一个来自任意客户机的业务请求
28         for (;;) {
29                 printf("服务器：接收请求\n");
30                 char buf[1024];
31                 struct sockaddr_in addrcli = {};
32                 socklen_t addrlen = sizeof(addrcli);
33                 ssize_t rb = recvfrom(sockfd, buf, sizeof(buf), 0,
34                         (struct sockaddr*)&addrcli, &addrlen);
35                 if (rb == -1) {
36                         perror("recvfrom");
37                         return -1;
38                 }
39                 printf("服务器：向%s:%hu 发送响应\n",
40                         inet_ntoa(addrcli.sin_addr), ntohs(addrcli.sin_port));
41                 if (sendto(sockfd, buf, rb, 0, (struct sockaddr*)&addrcli, addrlen) == -1){
```

```
42                                  perror("sendto");
43                                  return -1;
44                          }
45                  }
46                  printf("服务器：关闭套接字\n");
47                  if (close(sockfd) == -1) {
48                          perror("close");
49                          return -1;
50                  }
51                  return 0;
52  }
```

udpcli.c

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <string.h>
 4  #include <unistd.h>
 5  #include <sys/socket.h>
 6  #include <netinet/in.h>
 7  #include <arpa/inet.h>
 8  int main(int argc, char* argv[]) {
 9          if (argc < 3) {
10                  fprintf(stderr, "用法：%s <IP 地址> <端口号>\n", argv[0]);
11                  return -1;
12          }
13          printf("客户机：创建套接字\n");
14          int sockfd = socket(PF_INET, SOCK_DGRAM, 0);
15          if (sockfd == -1) {
16                  perror("socket");
17                  return -1;
18          }
19          printf("客户机：准备地址\n");
20          struct sockaddr_in addr;
21          addr.sin_family = AF_INET;
22          addr.sin_port = htons(atoi(argv[2]));
23          addr.sin_addr.s_addr = inet_addr(argv[1]);
24          if (connect(sockfd, (struct sockaddr*)&addr, sizeof(addr)) == -1) {
25                  perror("connect");
26                  return -1;
27          }//connect 的作用：起到缓存作用，以便后面调用 send 函数时写的形参少，实则是
                                    在 send 函数中调用 sendto
28          for (;;) {
29                  printf("> ");
30                  char buf[1024];
31                  fgets(buf, sizeof(buf) / sizeof(buf[0]), stdin);
32                  if (!strcmp(buf, "!\n"))
33                          break;
```

```
34                      printf("客户机：发送请求\n");
35                      if (send(sockfd, buf, strlen(buf) * sizeof(buf[0]), 0) == -1) {
36                              perror("send");
37                              return -1;
38                      }
39                      /*
40                      if (sendto(sockfd, buf, strlen(buf) * sizeof(buf[0]), 0,
41                              (struct sockaddr*)&addr, sizeof(addr)) == -1) {
42                              perror("sendto");
43                              return -1;
44                      }
45                      */
46                      printf("客户机：接收响应\n");
47                      ssize_t rb = recv(sockfd, buf, sizeof(buf) - sizeof(buf[0]), 0);
48                      if (rb == -1) {
49                              perror("recv");
50                              return -1;
51                      }
52                      buf[rb / sizeof(buf[0])] = '\0';
53                      printf("< %s", buf);
54              }
55              printf("客户机：关闭套接字\n");
56              if (close(sockfd) == -1) {
57                      perror("close");
58                      return -1;
59              }
60              return 0;
61      }
```

针对报文套接字的 connect 函数与流式套接字截然不同，既无三路握手，亦无虚电路，而仅仅是将传递给该函数的对方地址结构缓存在套接字对象中。此后发送数据时，可以不用 sendto 指明接收者地址，而是直接调用 send 函数发送数据，该函数会从套接字对象中获取关于目的地址的缓存，并以之为参数调用 sendto 函数完成发送。

5. 基于本地(UNIX 域)套接字的进程间通信

6.

```
int sockfd = socket(PF_LOCAL/FP_UNIX, SOCK_DGRAM, 0);
        |
        v
  本地套接字
        |bind
        v
  套接字文件
```

```
#include <sys/un.h>
struct sockaddr_un {
    sa_family_t sun_family; // 地址族(AF_LOCAL/AF_UNIX)
    char        sun_path[]; // 套接字文件路径
```

};
进程 1-本地套接字-bind-套接字文件-bind-本地套接字-进程 2
进程 1-文件描述符-open-管道　文件-open-文件描述符-进程 2

代码：locsvr.c

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <sys/socket.h>
 4 #include <sys/un.h>
 5 #define SOCK_FILE "mysock"
 6 int main(void) {
 7         printf("服务器：创建套接字\n");
 8         int sockfd = socket(PF_LOCAL, SOCK_DGRAM, 0);
 9         if (sockfd == -1) {
10                 perror("socket");
11                 return -1;
12         }
13         printf("服务器：准备地址并绑定\n");
14         struct sockaddr_un addr;
15         addr.sun_family = AF_LOCAL;
16         strcpy(addr.sun_path, SOCK_FILE);
17         if (bind(sockfd, (struct sockaddr*)&addr, sizeof(addr)) == -1) {
18                 perror("bind");
19                 return -1;
20         }
21         printf("服务器：接收数据\n");
22         for (;;) {
23                 char buf[1024] = {};
24                 ssize_t rb = read(sockfd, buf, sizeof(buf) - sizeof(buf[0]));
25                 if (rb == -1) {
26                         perror("read");
27                         return -1;
28                 }
29                 if (!strcmp(buf, "!\n"))
30                         break;
31                 printf("< %s", buf);
32         }
33         printf("服务器：关闭套接字\n");
34         if (close(sockfd) == -1) {
35                 perror("close");
36                 return -1;
37         }
38         if (unlink(SOCK_FILE) == -1) {
39                 perror("unlink");
40                 return -1;
41         }
42         return 0;
```

```
43 }
```

loccli.c

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <sys/socket.h>
 4 #include <sys/un.h>
 5 #define SOCK_FILE "mysock"
 6 int main(void) {
 7         printf("客户机：创建套接字\n");
 8         int sockfd = socket(PF_LOCAL, SOCK_DGRAM, 0);
 9         if (sockfd == -1) {
10                 perror("socket");
11                 return -1;
12         }
13         printf("客户机：准备地址并连接\n");
14         struct sockaddr_un addr;
15         addr.sun_family = AF_LOCAL;
16         strcpy(addr.sun_path, SOCK_FILE);
17         if (connect(sockfd, (struct sockaddr*)&addr, sizeof(addr)) == -1) {
18                 perror("connect");
19                 return -1;
20         }
21         printf("客户机：发送数据\n");
22         for (;;) {
23                 printf("> ");
24                 char buf[1024];
25                 fgets(buf, sizeof(buf) / sizeof(buf[0]), stdin);
26                 if (write(sockfd, buf, strlen(buf) * sizeof(buf[0])) == -1) {
27                         perror("write");
28                         return -1;
29                 }
30                 if (!strcmp(buf, "!\n"))
31                         break;
32         }
33         printf("客户机：关闭套接字\n");
34         if (close(sockfd) == -1) {
35                 perror("close");
36                 return -1;
37         }
38         return 0;
39 }
```

6.域名解析

DNS 服务器	TMOOC 服务器
Domain Name Service	HTTP Service
域名服务	Web 服务

```
                ^   |                                ^
www.tmooc.cn |   | 111.202.83.186            |
                |   v                                |
              浏览器<----------------------+
                                111.202.83.186
```

```
#include <netdb.h>
struct hostent* gethostbyname(const char* name);
成功返回主机条目结构体指针，失败返回 NULL。
name: 主机域名
hostent
  h_name -> xxx\0 - 主机官方名
  h_aliases -> |*|*|*|...|NULL| - 别名表
                  |
                  +--> xxx\0 - 别名
  h_addrtype: AF_INET - 地址类型
  h_length:    4          - 地址字节数
  h_addr_list -> |*|*|*|...|NULL| - 地址表
                    |
                    +--> in_addr - IPv4 地址
struct in_addr {
    in_addr_t s_addr;
};
typedef uint32_t in_addr_t; // unsigned long
```

代码：dns.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <string.h>
 4 #include <unistd.h>
 5 #include <sys/socket.h>
 6 #include <netinet/in.h>
 7 #include <arpa/inet.h>
 8 #include <netdb.h>
 9 int main(void) {
10         struct hostent* host = gethostbyname("www.tmooc.cn");
11         if (!host) {
12                 perror("gethostbyname");
13                 return -1;
14         }
15         printf("主机官方名：\n");
16         printf("\t%s\n", host->h_name);
17         printf("主机别名表：\n");
18         for (char** pp = host->h_aliases; *pp; ++pp)
19                 printf("\t%s\n", *pp);
20         if (host->h_addrtype == AF_INET) {
21                 printf("主机地址表：\n");
```

```
22                      for (struct in_addr** pp = (struct in_addr**)host->h_addr_list; *pp; ++pp)
23                              printf("\t%s\n", inet_ntoa(**pp));
24              }
25              int sockfd = socket(PF_INET, SOCK_STREAM, 0);
26              if (sockfd == -1) {
27                      perror("socket");
28                      return -1;
29              }
30              struct sockaddr_in addr;
31              addr.sin_family = AF_INET;
32              addr.sin_port = htons(80);
33              addr.sin_addr = **(struct in_addr**)host->h_addr_list;
34              if (connect(sockfd, (struct sockaddr*)&addr, sizeof(addr)) == -1) {
35                      perror("connect");
36                      return -1;
37              }
38              char request[1024];
39              sprintf(request,
40                      "GET / HTTP/1.0\r\n"
41                      "Host: www.tmooc.cn\r\n"
42                      "Accept: text/html\r\n"
43                      "Connection: close\r\n"
44                      "User-Agent: Mozilla/5.0\r\n"
45                      "Referer: www.tmooc.cn\r\n\r\n");
46              if (send(sockfd, request, strlen(request) * sizeof(request[0]), 0) == -1) {
47                      perror("send");
48                      return 0;
49              }
50              for (;;) {
51                      char respond[1024] = {};
52                      ssize_t rlen = recv(sockfd, respond,
53                              sizeof(respond) - sizeof(respond[0]), 0);
54                      if (rlen == -1) {
55                              perror("recv");
56                              return -1;
57                      }
58                      if (!rlen)
59                              break;
60                      printf("%s", respond);
61              }
62              printf("\n");
63              close(sockfd);
64              return 0;
65      }
```

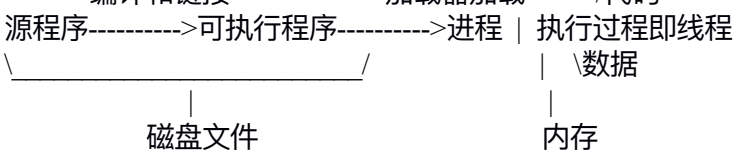
Shell 终端命令：

./dns > tmooc_head.txt ---> 包头

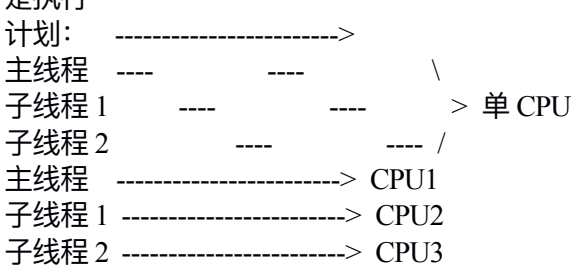
./tmooc_default.html ---->包体
gedit 文件名 --->以文本方式查看文件

十二、线程

1.线程的基本概念
1)线程是程序的执行路线，即进程内部的控制序列，或者说是进程的子任务。



2)一个进程可以同时拥有多个线程，即同时被系统调度的多条执行路线，但至少要有一个主线程。



3)一个进程中的所有线程都共享进程的代码区、数据区、堆区、环境变量和命令行参数、文件描述符、信号处理函数、当前工作目录、各种用户 ID 和组 ID；每个线程都拥有独立的线程 ID(所谓进程 ID 其实就是该进程的主线程的线程 ID)、CPU 的寄存器状态、栈内存、调度策略和优先级、信号掩码、errno 以及线程私有数据(相当于线程级的全局变量)等。
4)线程调度
A.系统内核中专门负责线程调度的处理单元被称为调度器；
B.调度器将所有处于就绪状态(没有阻塞在任何系统调用上)的线程排成一个队列，即就绪队列；
C.调度器从就绪队列中获取队首线程，为其分配一个时间片，并令处理机执行该线程，过了一段时间：
a)该线程的时间片耗尽，调度器立即中止该线程，将其排到就绪队列的尾端，接着从队首获取下一个线程，为其分配时间片并执行；
b)该线程的时间片未耗尽，但需阻塞于某系统调用，比如等待 I/O 或者睡眠。调用器会提前中止该线程，并将其从就绪队列移出至等待队列，直到其等待的条件得到满足后，再被移回调度队列；
D.在低优先级线程执行期间，有高优先级线程就绪，后者会抢占前者的时间片；
E.若就绪队列为空，则系统进程空闲状态，直至其非空。
5)时间片
A.调度器分配给每个线程的时间片长短，对系统的行为和性能影响很大：
a)如果时间片过长，线程必须等待很长时间才能重新获得处理机，这就降低了系统运行的并行性，用户会感觉到明显的响应延迟；
b)如果时间片过短，大量的时间会浪费在线程切换上，同时降低了虚拟内存的存储命中率，线程的时间局部性无法得到保证；
B.某些 Unix 系统倾向于为线程分配较长的时间片，希望通过扩大系统吞吐率来改善其整体表现；而另一些 Unix 系统则更倾向于为线程分配较短的时间片，以提升系统的交互性；
C.Linux 系统根据线程在不同时间的具体表现，为其动态分配时间片，在吞吐率和交互性之间寻

求最佳平衡点：
a)处理机约束：较短的时间片
b)I/O 约束：较长的时间片
2.线程的基本特点
1)线程是进程的一个实体，可作为系统独立调度和分派的基本单位。
2)线程有不同的状态，系统提供了多种线程控制原语，如启动、终止、暂停、继续、取消等等。
3)线程可以使用的大部分资源都是隶属于进程的，即使是在特定线程中动态分配的资源，也同样为进程所有。
4)一个进程中可以有多个线程并发地运行。它们可以执行相同的代码，也可以执行不同的代码。
5)同一个进程的多个线程都在同一个地址空间内活动，因此相比于进程，线程的系统开销要小得多，而且切换速度也快。
6)进程空间内的代码和数据对于该进程的每个线程而言都是共享的，因此同一个进程的不同线程之间不存在通信问题，当然也就不需要类似 IPC 的通信机制。
7)线程之间虽然不存在通信问题但是存在冲突问题。同样是因为数据共享，当一个进程的多个线程"同时"访问一份数据时，线程间的冲突可能造成逻辑甚至系统错误。
8)线程之间存在优先级的差异。即使低优先级线程的时间片尚未耗尽，只要高优先级线程处于就绪状态，就会立即抢夺低优先级线程手中的处理机。
3.POSIX 线程
IEEE POSIX 1003.1c (1995)标准定义了关于线程的同一接口。
包含头文件：#include <pthread.h>
链接线程库：-lpthread

4.创建线程

```
   +-----------------------+
   v 描述一个线程的执行过程 |
线程——执行过程——代码序列——函数
   | 在一个独立的过程中调用 ^
   +-----------------------+
```

main 函数：描述了主线程的执行过程。
线程过程函数：描述了子线程的执行过程。
线程过程函数的形式如下所示：

```
void* 线程过程函数(void* arg) {
    线程的执行过程
}
```

创建线程：

```
#include <pthread.h>
int pthread_create(pthread_t* tid,const pthread_attr_t* attr,
    void* (*start_routine)(void*),
    void* arg);
```

成功返回 0，失败返回错误码。
tid: 输出所创建线程的线程 ID。
pthread_t: unsigned long int
attr: 线程属性，取 NULL 表示缺省属性。
start_routine: 线程过程函数指针。
arg: 传递给线程过程函数的参数。
void* start_routine(void* arg) { ... }

```
int main(void) {
    ...
    pthread_create(..., start_routine, ...);
    ...
}
            pthread_create
主线程：---------+-----------> main              TID=1000
子线程：                \___________> start_routine  TID=1001
```

代码：create.c

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <unistd.h>
 4 #include <pthread.h>
 5 #include <sys/syscall.h>
 6 // 子线程的线程过程函数：子线程干什么?
 7 void* thread_proc(void* arg) {
 8         printf("%lu 线程：%s\n", pthread_self(), (char*)arg);
 9         printf("%ld 线程：%s\n", syscall(SYS_gettid), (char*)arg);
10         return NULL;
11 }
12 // 主线程的线程过程函数：主线程干什么?
13 int main(void) {
14         printf("%d 进程：进程开始\n", getpid());
15         printf("%lu 线程：主线程开始\n", pthread_self());
16         printf("%ld 线程：主线程开始\n", syscall(SYS_gettid));
17         // 创建子线程：开始干!
18         pthread_t tid;
19         int error = pthread_create(&tid, NULL, thread_proc, "Hello, World!");
20         if (error) {
21                 fprintf(stderr, "pthread_create: %s\n", strerror(error));
22                 return -1;
23         }
24         printf("子线程的 TID：%lu\n", tid);
25         sleep(1);
26         return 0;
27 }
```

5.线程并发

1)pthread_create 函数本身并不调用线程过程函数，而是在系统内核中开启独立的线程，并立即返回，在该线程中执行线程过程函数中的代码。

2)pthread_create 函数返回时，所指定的线程过程函数可能尚未执行，也可能正在执行，甚至可能已经执行完毕并返回。

3)如果主线程先于子线程结束，则由于进程结束，所有的子线程都会被直接终止。

4)主线程和通过 pthread_create 函数创建的一到多个子线程，在时间上“同时”运行，如果不附加任何同步条件，则它们一个执行步骤的先后顺序是完全无法预知的，这就叫做自由并发。

代码：concur.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <string.h>
 4 #include <unistd.h>
 5 #include <pthread.h>
 6 void* thread_proc(void* arg) {
 7         for (;;) {
 8                 printf("%lld", (long long)arg);
 9                 usleep((rand() % 100) * 1000);
10         }
11         return NULL;
12 }
13 int main(void) {
14         srand(time(NULL));
15         setbuf(stdout, NULL);
16         for (long long i = 0; i < 5; ++i) {
17                 pthread_t tid;
18                 int error = pthread_create(&tid, NULL, thread_proc, (void*)(i+1));
19                 if (error) {
20                         fprintf(stderr, "pthread_create: %s\n", strerror(err      or));
21                         return -1;
22                 }
23         }
24         getchar();
25         return 0;
26 }
```

6.线程参数

1)传递给线程过程函数的参数是一个泛型指针 void*，它可以指向任何类型的数据：基本类型变量、结构体类型变量或者数组型变量等等，但必须保证在线程过程函数执行期间，该指针所指向的目标变量持久有效，直到线程过程函数不在使用它为止。

2)调用 pthread_create 函数的代码在用户空间，线程过程函数的代码也在用户空间，但偏偏创建线程的动作由系统内核完成。因此传递给线程过程函数的参数也不得不经由系统内核传递给线程过程函数。pthread_create 函数的 arg 参数负责将线程过程函数的参数带入系统内核。

```
用户空间：  pthread_create(... arg) | thread_proc(... arg)
-----------------------------v----------------------^--
内核空间：                        创建线程------>调用线程过程函数
```

代码：arg.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <unistd.h>
 4 #include <pthread.h>
 5 #define PI 3.14159
 6 void* circle_area(void* arg) {
 7         double r = *(double*)arg;
```

```
 8              *(double*)arg = PI * r * r;
 9              return NULL;
10 }
11 /*
12 struct rect {
13              double w, h, s;
14 };
15 void* rect_area(void* arg) {
16              struct rect* p = (struct rect*)arg;
17              p->s = p->w * p->h;
18              return NULL;
19 }
20 */
21 void* rect_area(void* arg) {
22              double* p = (double*)arg;
23              p[2] = p[0] * p[1];
24              printf("%g\n", p[2]);
25              free(p);
26              return NULL;
27 }
28 int main(void) {
29              pthread_t tid;
30              double a = 10;
31              pthread_create(&tid, NULL, circle_area, &a);
32              usleep(10000);
33              printf("%g\n", a);
34              /*
35              struct rect b = {10, 5};
36              pthread_create(&tid, NULL, rect_area, &b);
37              usleep(10000);
38              printf("%g\n", b.s);
39              *//*
40              double b[3] = {10, 5};
41              */
42              double* b = malloc(3 * sizeof(double));
43              b[0] = 10;
44              b[1] = 5;
45              pthread_create(&tid, NULL, rect_area, b);
46              //free(b);
47              //usleep(10000);
48              //printf("%g\n", b[2]);
49              //free(b);
50              getchar();
51              return 0;
52 }
```

7.汇合线程

父线程等待子线程结束后继续后续工作
父线程想知道子线程过程函数的返回值
父线程需要回收子线程的“僵尸”

线程版的 waitpid

```
        pthread_create  pthread_join->9
父线程：-------+--------------+------->
子线程：              \_____________/|
                                    return  9
```

#include <pthread.h>
int pthread_join(pthread_t tid, void** retval);
成功返回 0，失败返回错误码。
tid: 线程 ID。
retval: 输出线程过程函数的返回值。

```
void* thread_proc(void* arg) {
    ...
    return p;
}           |
            v
        +-------+
retval->| void* |
        +-------+
```

在线程过程函数及被线程过程函数直接或间接调用的函数中终止线程
void pthread_exit(void* retval);
无返回值。
retval: 相当于线程过程函数的返回值，可被 pthread_join 函数得到。

代码：join.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <unistd.h>
 4 #include <pthread.h>
 5 #define PI 3.14159
 6 double* calculate(double r) {
 7         double* s = malloc(sizeof(double));
 8         *s = PI * r * r;
 9         pthread_exit(s);//终止线程
10         //return s;
11 }
12 void* circle_area(void* arg) {
13         return calculate(*(double*)arg);
14 }
15 int main(void) {
16         pthread_t tid;
17         double r = 10, *s;
18         pthread_create(&tid, NULL, circle_area, &r);
```

```
19              pthread_join(tid, (void**)&s);
20              printf("%g\n", *s);
21              free(s);
22              return 0;
23  }
```

8.分离线程

```
#include <pthread.h>
int pthread_detach(pthread_t tid);
```

成功返回 0，失败返回错误码。

tid: 线程 ID。

将 tid 所标识的线程设置为分离线程，该线程不可被汇合(调用 pthread_join 函数会返回失败)，而且该线程的“僵尸”会被系统自动回收。

代码：detach.c

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <unistd.h>
4  #include <pthread.h>
5  void* thread_proc(void* arg) {
6               for (int i = 0; i < 200; ++i) {
7                       putchar('-');
8                       usleep(50000);
9               }
10              return NULL;
11 }
12 int main(void) {
13              setbuf(stdout, NULL);
14              pthread_t tid;
15              int error = pthread_create(&tid, NULL, thread_proc, NULL);
16              if (error) {
17                      fprintf(stderr, "pthread_create: %s\n", strerror(error));
18                      return -1;
19              }
20              if ((error = pthread_detach(tid)) != 0) {
21                      fprintf(stderr, "pthread_detach: %s\n", strerror(error));
22                      return -1;
23              }
24              if ((error = pthread_join(tid, NULL)) != 0)
25                      fprintf(stderr, "pthread_join: %s\n", strerror(error));
26              for (int i = 0; i < 200; ++i) {
27                      putchar('+');
28                      usleep(100000);
29              }
30              printf("\n");
31              return 0;
32 }
```

9.比较线程 ID 是否相等

```
#include <pthread.h>
int pthread_equal(pthread_t t1, pthread_t t2);
```

若两个参数线程 ID 相等则返回非零，否则返回 0。

代码：equal.c

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <pthread.h>
 4 pthread_t main_tid;
 5 void foo(void) {
 6         if (pthread_equal(pthread_self(), main_tid))
 7                 printf("主线程执行的代码...\n");
 8         else
 9                 printf("子线程执行的代码...\n");
10 }
11 void* thread_proc(void* arg) {
12         foo();
13         return NULL;
14 }
15 int main(void) {
16         main_tid = pthread_self();
17         foo();
18         pthread_t tid;
19         pthread_create(&tid, NULL, thread_proc, NULL);
20         pthread_join(tid, NULL);
21         return 0;
22 }
```

10.线程冲突

当两个或者更多线程同时访问某些共享资源时，难免会出现数据不一致或不完整的问题。

代码：vie.c

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <pthread.h>
 4 int a = 0;
 5 void* thread_proc(void* arg) {
 6         for (int i = 0; i < 100000000; ++i)
 7                 ++a;
 8         return NULL;
 9 }
10 int main(void) {
11         pthread_t t1, t2;
12         pthread_create(&t1, NULL, thread_proc, NULL);
13         pthread_create(&t2, NULL, thread_proc, NULL);
```

```
 14              pthread_join(t1, NULL);
 15              pthread_join(t2, NULL);
 16              printf("%d\n", a);
 17              return 0;
 18 }
```

不发生冲突情况：

```
线程 1              eax     a       eax             线程 2
movl a, %eax        0 <- 0
addl $1, %eax       1       0
movl %eax, a        1 -> 1
                            1 -> 1              movl a, %eax
                            1       2           addl $1, %eax
                            2 <- 2              movl %eax, a
```

发生冲突情况：

```
线程 1              eax     a       eax             线程 2
movl a, %eax        0 <- 0
                            0 -> 0              movl a, %eax
addl $1, %eax       1       0
                            0       1           addl $1, %eax
movl %eax, a        1 -> 1
                            1 <- 1              movl %eax, a
```

11.互斥锁

创建互斥锁

线程 1

加锁互斥锁

锁区代码块

解锁互斥锁

线程 2

加锁互斥锁

锁区代码块

解锁互斥锁

锁区代码块中的代码在任何时候最多只被一个线程执行。

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;//静态初始化
```

```
pthread_mutex_t m;//动态声明
pthread_mutex_init(&m, NULL);//动态初始化
...
pthread_mutex_destroy(&m);//动态释放
```

```
pthread_mutex_lock(&m);     // 加锁
pthread_mutex_unlock(&m);   // 解锁
```

任何时刻只会有一个线程对特定的互斥锁加锁成功，其它试图对其加锁的线程会在加锁函数的阻塞中等待，直到该互斥锁的持有者将其解锁。对特定互斥锁加锁成功的线程通过调用解锁函数将其加锁，那些阻塞于对该互斥锁加锁的线程中的一个会被唤醒，得到该互斥锁，并从加锁函数中返回，执行锁区代码。

代码：mutex.c

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <pthread.h>
 4 int a = 0;
 5 pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
 6 void* thread_proc(void* arg) {
 7         for (int i = 0; i < 100000000; ++i) {
 8                 pthread_mutex_lock(&m);
 9                 ++a;
10                 pthread_mutex_unlock(&m);
11         }
12         return NULL;
13 }
14 int main(void) {
15         pthread_t t1, t2;
16         pthread_create(&t1, NULL, thread_proc, NULL);
17         pthread_create(&t2, NULL, thread_proc, NULL);
18         pthread_join(t1, NULL);
19         pthread_join(t2, NULL);
20         printf("%d\n", a);
21         return 0;
22 }
```

互斥锁仅用于需要避免冲突的场合，不要滥用，即锁区不宜过大，因为互斥锁会降低程序运行的性能，且破坏并发特性。

12.停等问题和条件变量

边际同步：多事时候线程之间异步地运行，在特定条件下，需要某种形式的彼此等待，这就是停等问题。

```
 生产者    消费者    模型
   |         |
提供资源  使用资源
生产者 -资源-> 资源仓库 -资源-> 消费者
   |                              ^
   +------------>满--------------+
   ^                              |
   +--------------空<-------------+
```

```
pthread_cond_t c = PTHREAD_COND_INITIALIZER;//静态初始化
```

```
pthread_cond_t c;//动态声明
pthread_cond_init(&c, NULL);//动态初始化
...
```

pthread_cond_destroy(&c);//动态释放
---
pthread_cond_wait(&c, &m);
令调用线程睡入条件变量，同时解锁互斥锁。一旦该线程被唤醒，从此函数中返回，即重新获得互斥锁。
pthread_cond_signal(&c);
唤醒在条件变量中睡眠的一个线程。
pthread_cond_broadcast(&c);
唤醒在条件变量中睡眠的所有线程，但其中只有一个线程会得到互斥锁而从 pthread_cond_wait 函数中返回，而其它线程会继续在 pthread_cond_wait 函数中阻塞，等待锁。

代码：cond.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <string.h>
 4 #include <unistd.h>
 5 #include <pthread.h>
 6 #define MAX_STOCK 10 // 仓库容量
 7 char storage[MAX_STOCK]; // 仓库
 8 size_t stock = 0; // 当前库存
 9 pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
10 pthread_cond_t cp = PTHREAD_COND_INITIALIZER;
11 pthread_cond_t cc = PTHREAD_COND_INITIALIZER;
12 // 显示库存
13 void show(const char* who, const char* op, char prod) {
14         printf("%s：", who);
15         for(size_t i = 0; i < stock; ++i)
16                 printf("%c", storage[i]);
17         printf("%s%c\n", op, prod);
18 }
19 // 生产者线程
20 void* producer(void* arg) {
21         const char* who = (const char*)arg;
22         for (;;) {
23                 pthread_mutex_lock(&m);
24                 while (stock >= MAX_STOCK) {
25                         printf("\033[;;32m%s：满仓！\033[0m\n", who);
                                        //用绿色的字显示
26                         pthread_cond_wait(&cp, &m);
27                 }
28                 char prod = 'A' + rand() % 26;
29                 show(who, "<-", prod);
30                 storage[stock++] = prod;
31                 //pthread_cond_signal(&cc);
32                 pthread_cond_broadcast(&cc);
33                 pthread_mutex_unlock(&m);
```

```
34                      usleep((rand() % 100) * 1000);
35              }
36              return NULL;
37  }
38  // 消费者线程
39  void* customer(void* arg) {
40              const char* who = (const char*)arg;
41              for (;;) {
42                      pthread_mutex_lock(&m);
43                      while (!stock) {
44                              printf("\033[;;31m%s：空仓！\033[0m\n", who);
45                              pthread_cond_wait(&cc, &m);
46                      }
47                      char prod = storage[--stock];
48                      show(who, "->", prod);
49                      //pthread_cond_signal(&cp);
50                      pthread_cond_broadcast(&cp);
51                      pthread_mutex_unlock(&m);
52                      usleep((rand() % 100) * 1000);
53              }
54              return NULL;
55  }
56  int main(void) {
57              srand(time(NULL));
58              pthread_t tid;
59              pthread_create(&tid, NULL, producer, "生产者 1");
60              pthread_create(&tid, NULL, producer, "生产者 2");
61              pthread_create(&tid, NULL, customer, "消费者 1");
62              pthread_create(&tid, NULL, customer, "消费者 2");
63              getchar();
64              return 0;
65  }
```

十三、项目：Web 服务器————显示静态页面

1.基本知识：HTTP 协议

1)URL：统一资源定位符，在互联网范围内唯一地标识一个特定的资源，如文件、应用，或者 API 等。

```
http://www.tmooc.cn/files/login.html:8080 .....
\_____/\_________/\_______________/\___/\____________/
   |         |              |            |           |
 协议      主机域名          路径         端口号       参数
   |         |<-DNS         |            |           |
应用层  服务器 IP 地址    Web 服务器解释  区分服务    输入
\_____________________/        \            |          /
          |                      +----------------+
          |----------------------------/          |
      客户机处理                                服务器处理
```

2)HTTP 请求
A.包头
```
方法          路径          协议及版本(客户端)
 |             |                |
GET  /files/login.html  HTTP/1.0\r\n  //  方法、路径和协议
Host:  www.tmooc.cn\r\n                    //  主机域名
Accept:  text/html\r\n                     //  可接受内容类型
Connection:  keep-alive\r\n                //  期望连接模式
User-Agent:  Mozilla/5.0\r\n               //  兼容浏览器
Referer:  www.tmooc.cn\r\n\r\n             //  引用范围
```
B.包体
对于 GET 方法，包体为空。
对于 POST 方法，包体包含需要上传的数据。
3)HTTP 响应
A.包头
```
 协议及   响应码
  版本    及解释
   |        /   \
HTTP/1.1  200  OK\r\n                                  //  协议和响应信息
Date:  Tue,  14  Apr  2020  09:14:49  GMT\r\n  //  处理日期和时间
Content-Type:  text/html\r\n                       //  内容类型
Content-Length:  2048                                 //  内容长度
Connection:  keep-alive\r\n                        //  实际连接模式
Server:  tarena\r\n\r\n                            //  服务器别名
```
B.包体
```
<html>   \
...          >  页面内容或处理结果
</html>  /
```
2.需求分析————明确系统的功能要求
```
    ___URL___      __HTTP 请求__
  |            v  |                 v
用户          浏览器          Web 服务器<-->资源文件
  ^_________|  ^_____________|
    渲染页面        HTTP 响应
```
3.概要设计————设计系统的整体架构
```
                          +-------------+
                          | 主模块(main) |
                          +-------------+
                                 |
                          +-------------+   +-------------+
                          |  服务器模块   |--| 信号处理模块 |
                          +-------------+   +-------------+
                                 |
+-------------+   +-------------+   +-------------+
| 协议处理模块 |--| 客户线程模块 |--| 资源管理模块 |
+-------------+   +-------------+   +-------------+
                                 |
```

```
                          +--------------+
                          | 网络通信模块 |
                          +--------------+
------------------------|-------------------------------
                          +---------------+
                          | 客户机(浏览器)|
                          +---------------+
```

4.详细设计————设计每个模块的内部结构
1)函数的名称、参数表和返回值；
2)类的名称、成员、访问控制属性和继承关系；
3)引入必要的抽象提高代码的伸缩性和复用性；
4)借鉴成熟的设计模式。
5.编写代码————编程实现每个模块的具体功能
设计顺序：自顶向下，逐层细化。
实现顺序：自底向上，逐层抽象。
1)协议处理模块：http.h、http.c -> http.o
2)网络通信模块：sock.h、sock.c -> sock.o
3)资源管理模块：resource.h、resource.c -> resource.o
4)客户线程模块：client.h、client.c -> client.o
5)信号处理模块：signals.h、signals.c -> signals.o
6)服务器模块：server.h、server.c -> server.o
7)主模块(main)：main.c -> main.o
8)构建脚本：Makfile
代码：code/
1)协议处理模块：http.h、http.c -> http.o
http.h

```
 1 // 声明协议处理模块
 2 #ifndef _HTTP_H
 3 #define _HTTP_H
 4 #include <limits.h>//定义一些上限的文件
 5 #include <sys/types.h>
 6 // HTTP 请求
 7 typedef struct tag_HttpRequest {
 8         char method[32];            // 方法
 9         char path[PATH_MAX + 1]; // 路径
10         char protocol[32];          // 协议
11         char connection[32];        // 连接
12 }       HTTP_REQUEST;
13 // 解析请求
14 int parseRequest(const char* req, HTTP_REQUEST* hreq);
15 // HTTP 响应
16 typedef struct tag_HttpRespond {
17         char   protocol[32];    // 协议
18         int    status;          // 状态
19         char   desc[256];       // 描述
20         char   type[32];        // 类型
21         off_t  length;          // 长度
```

```
22              char    connection[32]; // 连接
23 }            HTTP_RESPOND;
24 // 构造响应头
25 void constructHead(HTTP_RESPOND const* hres, char* head);
26 #endif // _HTTP_H
```

http.c

```
 1 // 实现协议处理模块
 2 #include <unistd.h>
 3 #include <sys/syscall.h>
 4 #include <stdio.h>
 5 #define __USE_GNU
 6 #include <string.h>
 7 #include <time.h>
 8 #include "http.h"
 9 // 解析请求
10 int parseRequest(const char* req, HTTP_REQUEST* hreq) {
11              sscanf(req, "%s%s%s", hreq->method, hreq->path, hreq->protocol);
12              char* connection = strcasestr(req, "connection:");//查找字符串
13              if (connection)
14                      sscanf(connection, "%*s%s", hreq->connection);
15              printf("%d.%ld> [%s][%s][%s][%s]\n", getpid(), syscall(SYS_gettid),
16                      hreq->method, hreq->path, hreq->protocol, hreq->connection);
17              if (strcasecmp(hreq->method, "get")) {
18                      printf("%d.%ld> 无效方法\n", getpid(), syscall(SYS_gettid));
19                      return -1;
20              }
21              if (strcasecmp(hreq->protocol, "http/1.0") &&
22                      strcasecmp(hreq->protocol, "http/1.1")) {
23                      printf("%d.%ld> 无效协议\n", getpid(), syscall(SYS_gettid));
24                      return -1;
25              }
26              return 0;
27 }
28 // 构造响应头
29 void constructHead(HTTP_RESPOND const* hres, char* head) {
30              char dateTime[32];
31              time_t now = time(NULL);
32          strftime(dateTime, sizeof(dateTime), "%a, %d %b %Y %T GMT", gmtime(&     now));
33              sprintf(head,
34                      "%s %d %s\r\n"
35                      "Server: Tarena WebServer 1.0\r\n"
36                      "Date: %s\r\n"
37                      "Content-Type: %s\r\n"
38                      "Content-Length: %ld\r\n"
39                      "Connection: %s\r\n\r\n",
40                      hres->protocol, hres->status, hres->desc,
```

```
41                      dateTime,
42                      hres->type,
43                      hres->length,
44                      hres->connection);
45 }
```

2)网络通信模块：sock.h、sock.c -> sock.o

sock.h

```
 1 // 声明网络通信模块
 2 #ifndef _SOCK_H
 3 #define _SOCK_H
 4 // 初始化套接字
 5 int initSocket(short port);
 6 // 接受客户机连接
 7 int acceptClient(void);
 8 // 接收请求
 9 char* recvRequest(int conn);
10 // 发送响应头
11 int sendHead(int conn, char const* head);
12 // 发送响应体
13 int sendBody(int conn, char const* path);
14 // 终结化套接字
15 void deinitSocket(void);
16 #endif // _SOCK_H
```

sock.c

```
 1 // 实现网络通信模块
 2 #include <unistd.h>
 3 #include <fcntl.h>
 4 #include <sys/syscall.h>
 5 #include <sys/socket.h>
 6 #include <netinet/in.h>
 7 #include <arpa/inet.h>
 8 #include <stdio.h>
 9 #include <stdlib.h>
10 #include <string.h>
11 #include "sock.h"
12 static int s_sock = -1; // 侦听套接字
13 // 初始化套接字
14 int initSocket(short port) {
15         printf("%d> 创建套接字\n", getpid());
16         if ((s_sock = socket(PF_INET, SOCK_STREAM, 0)) == -1) {
17                 perror("socket");
18                 return -1;
19         }
20         printf("%d> 设置套接字\n", getpid());
21         int on = 1;
```

```
22          if (setsockopt(s_sock, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on)) == -1){
23                  perror("setsockopt");
24                  return -1;
25          }
26          printf("%d> 绑定套接字\n", getpid());
27          struct sockaddr_in addr;
28          addr.sin_family         = AF_INET;
29          addr.sin_port           = htons(port);
30          addr.sin_addr.s_addr = INADDR_ANY;
31          if (bind(s_sock, (struct sockaddr*)&addr, sizeof(addr)) == -1) {
32                  perror("bind");
33                  return -1;
34          }
35          printf("%d> 侦听套接字\n", getpid());
36          if (listen(s_sock, 1024) == -1) {
37                  perror("listen");
38                  return -1;
39          }
40          return 0;
41 }
42 // 接受客户机连接
43 int acceptClient(void) {
44          printf("%d> 等待客户机连接\n", getpid());
45          struct sockaddr_in addrcli = {};
46          socklen_t addrlen = sizeof(addrcli);
47          int conn = accept(s_sock, (struct sockaddr*)&addrcli, &addrlen);
48          if (conn == -1) {
49                  perror("accept");
50                  return -1;
51          }
52          printf("%d> 已连接客户机%s:%hu\n", getpid(),
53                  inet_ntoa(addrcli.sin_addr), ntohs(addrcli.sin_port));
54          return conn;
55 }
56 // 接收请求
57 char* recvRequest(int conn) {
58          char* req = NULL;
59          ssize_t len = 0;
60          for (;;) {
61                  char buf[1024] = {};
62                  ssize_t rlen = recv(conn, buf, sizeof(buf) - sizeof(buf[0]), 0);
63                  if (rlen == -1) {
64                          perror("recv");
65                          free(req);
66                          return NULL;
67                  }
68                  if (!rlen) {
```

```
69                                  printf("%d.%ld> 客户机关闭连接\n", getpid(), syscall(SYS_gettid));
70                                  free(req);
71                                  return NULL;
72                          }
73                          req = (char*)realloc(req, len + rlen + 1);//内存重新分配
74                          memcpy(req + len, buf, rlen + 1);
75                          len += rlen;
76                          if (strstr(req, "\r\n\r\n"))
77                                  break;
78          }
79          return req;
80 }
81 // 发送响应头
82 int sendHead(int conn, char const* head) {
83          if (send(conn, head, strlen(head) * sizeof(head[0]), 0) == -1) {
84                  perror("send");
85                  return -1;
86          }
87          return 0;
88 }
89 // 发送响应体
90 int sendBody(int conn, char const* path) {
91          int fd = open(path, O_RDONLY);
92          if (fd == -1) {
93                  perror("open");
94                  return -1;
95          }
96          char buf[1024];
97          ssize_t len;
98          while ((len = read(fd, buf, sizeof(buf))) > 0)
99                  if (send(conn, buf, len, 0) == -1) {
100                         perror("send");
101                         return -1;
102                 }
103         if (len == -1) {
104                 perror("read");
105                 return -1;
106         }
107         close(fd);
108         return 0;
109 }
110 // 终结化套接字
111 void deinitSocket(void) {
112         close(s_sock);
113 }
```

3)资源管理模块：resource.h、resource.c -> resource.o

resource.h

```
 1 // 声明资源管理模块
 2 #ifndef _RESOURCE_H
 3 #define _RESOURCE_H
 4 // 搜索资源
 5 int searchResource(char const* path);
 6 // 识别类型
 7 int indentifyType(char const* path, char* type);
 8 #endif // _RESOURCE_H
```

resource.c

```
 1 // 实现资源管理模块
 2 #include <unistd.h>
 3 #include <sys/syscall.h>
 4 #include <stdio.h>
 5 #include <string.h>
 6 #include "resource.h"
 7 #include "mime.h"
 8 // 搜索资源
 9 int searchResource(char const* path) {
10         return access(path, R_OK);
11 }
12 // 识别类型
13 int indentifyType(char const* path, char* type) {
14         char* suffix = strrchr(path, '.');
15         if (!suffix) {
16                 printf("%d.%ld> 无法获取文件扩展名\n", getpid(), syscall(SYS_gettid));
17                 return -1;
18         }
19         for (size_t i = 0; i < sizeof(s_mime) / sizeof(s_mime[0]); ++i)
20                 if (!strcasecmp(suffix, s_mime[i].suffix)) {
21                         strcpy(type, s_mime[i].type);
22                         return 0;
23                 }
24         printf("%d.%ld> 不可识别的资源类型：%s\n", getpid(), syscall(SYS_gettid),
25                 suffix);
26         return -1;
27 }
```

4)客户线程模块：client.h、client.c -> client.o

client.h

```
 1 // 声明客户线程模块
 2 #ifndef _CLIENT_H
 3 #define _CLIENT_H
 4 // 客户线程参数
```

```
 5 typedef struct tag_ClientArgs {
 6         char const* home; // 主目录
 7         int         conn; // 连接套接字
 8 }       CA;
 9 // 客户线程函数
10 void* client(void* arg);
11 #endif // _CLIENT_H
```

client.c

```
 1 // 实现客户线程模块
 2 #include <unistd.h>
 3 #include <sys/syscall.h>
 4 #include <sys/stat.h>
 5 #include <stdio.h>
 6 #include <stdlib.h>
 7 #include <string.h>
 8 #include "client.h"
 9 #include "http.h"
10 #include "sock.h"
11 #include "resource.h"
12 // 客户线程函数
13 void* client(void* arg) {
14         printf("%d.%ld> 客户线程开始\n", getpid(), syscall(SYS_gettid));
15         CA* ca = (CA*)arg;
16         for (;;) {
17                 printf("%d.%ld> 接收请求\n", getpid(), syscall(SYS_gettid));
18                 char* req = recvRequest(ca->conn);
19                 if (!req)
20                         break;
21                 printf("%d.%ld> 请求电文\n\n%s",getpid(), syscall(SYS_gettid), req);
22                 printf("%d.%ld> 解析请求\n", getpid(), syscall(SYS_gettid));
23                 HTTP_REQUEST hreq = {};
24                 if (parseRequest(req, &hreq) == -1) {
25                         free(req);
26                         break;
27                 }
28                 free(req);
29                 // 资源路径
30                 char path[PATH_MAX + 1];
31                 strcpy(path, ca->home);
32                 if (path[strlen(path) - 1] == '/')
33                         path[strlen(path) - 1] = '\0';
34                 strcat(path, hreq.path);
35                 if (!strcmp(hreq.path, "/"))
36                         strcat(path, "index.html");
37                 printf("%d.%ld> 资源路径：%s\n",getpid(), syscall(SYS_gettid), path);
38                 // 搜索资源
```

```
39                  HTTP_RESPOND hres = {"HTTP/1.1", 200, "OK", "text/html"};
40                  if (searchResource(path) == -1) {
41                          hres.status = 404;
42                          strcpy(hres.desc, "Not Found");
43                          strcpy(path, "../home/404.html");
44                  }
45                  else
46                  // 识别类型
47                  if (indentifyType(path, hres.type) == -1) {
48                          hres.status = 404;
49                          strcpy(hres.desc, "Not Found");
50                          strcpy(path, "../home/404.html");
51                  }
52                  // 内容长度
53                  struct stat st;
54                  if (stat(path, &st) == -1) {
55                          perror("stat");
56                          break;
57                  }
58                  hres.length = st.st_size;
59                  // 连接模式
60                  if (strlen(hreq.connection))
61                          strcpy(hres.connection, hreq.connection);
62                  else
63                  if (!strcasecmp(hreq.protocol, "HTTP/1.0"))
64                          strcpy(hres.connection, "close");
65                  else
66                          strcpy(hres.connection, "keep-alive");
67                  printf("%d.%ld> 构造响应\n", getpid(), syscall(SYS_gettid));
68                  char head[1024];
69                  constructHead(&hres, head);
70                  printf("%d.%ld> 响应电文\n\n%s", getpid(),syscall(SYS_gettid), head);
71                  printf("%d.%ld> 发送响应\n", getpid(), syscall(SYS_gettid));
72                  // 发送响应头
73                  if (sendHead(ca->conn, head) == -1)
74                          break;
75                  // 发送响应体
76                  if (sendBody(ca->conn, path) == -1)
77                          break;
78                  // 若为短连接则断链结束服务
79                  if (!strcasecmp(hres.connection, "close"))
80                          break;
81          }
82          close(ca->conn);
83          free(ca);
84          printf("%d.%ld> 客户线程结束\n", getpid(), syscall(SYS_gettid));
85          return NULL;
```

```
 86 }
```

5)信号处理模块：signals.h、signals.c -> signals.o

signals.h

```
  1 // 声明信号处理模块
  2 #ifndef _SIGNALS_H
  3 #define _SIGNALS_H
  4 // 初始化信号
  5 int initSignals(void);
  6 #endif // _SIGNALS_H
```

signals.c

```
  1 // 实现信号处理模块
  2 #include <unistd.h>
  3 #include <signal.h>
  4 #include <stdio.h>
  5 #include "signals.h"
  6 // 初始化信号
  7 int initSignals(void) {
  8         printf("%d> 忽略大部分信号\n", getpid());
  9         for (int signum = 1; signum <= 64; ++signum)
 10                 if (signum != SIGINT && signum != SIGTERM)
 11                         signal(signum, SIG_IGN);
 12         return 0;
 13 }
```

6)服务器模块：server.h、server.c -> server.o

server.h

```
  1 // 声明服务器模块
  2 #ifndef _SERVER_H
  3 #define _SERVER_H
  4 // 初始化服务器
  5 int initServer(short port);
  6 // 运行服务器
  7 int runServer(char const* home);
  8 // 终结化服务器
  9 void deinitServer(void);
 10 #endif // _SERVER_H
```

server.c

```
  1 // 实现服务器模块
  2 #include <pthread.h>
  3 #include <sys/resource.h>
  4 #include <stdio.h>
  5 #include <stdlib.h>
```

```
#include <string.h>
#include "server.h"
#include "signals.h"
#include "sock.h"
#include "client.h"
// 初始化最大文件描述符数
static int initMaxFiles(void) {
        // 资源限制结构
        struct rlimit rl;
        // 获取当前进程最大文件描述符数
        if (getrlimit(RLIMIT_NOFILE, &rl) == -1) {
                perror("getrlimit");
                return -1;
        }
        // 若当前进程最大文件描述符数未达极限
        if (rl.rlim_cur < rl.rlim_max) {
                // 将当前进程最大文件描述符数设为极限
                rl.rlim_cur = rl.rlim_max;
                // 设置当前进程最大文件描述符数
                if (setrlimit(RLIMIT_NOFILE, &rl) == -1) {
                        perror("setrlimit");
                        return -1;
                }
        }
        return 0;
}
// 初始化服务器
int initServer(short port) {
        // 初始化最大文件描述符数
        if (initMaxFiles() == -1)
                return -1;
        // 初始化信号
        if (initSignals() == -1)
                return -1;
        // 初始化套接字
        if (initSocket(port) == -1)
                return -1;
        return 0;
}
// 运行服务器
int runServer(char const* home) {
        for (;;) {
                // 接受客户机的连接
                int conn = acceptClient();
                if (conn == -1)
                        return -1;
                // 处理客户机的业务
```

```
53                      pthread_t tid; // 线程标识
54                      pthread_attr_t attr; // 线程属性
55                      pthread_attr_init(&attr); // 将线程属性初始化为默认值
56                      pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
                                                                // 分离线程
57                      CA* ca = (CA*)malloc(sizeof(CA)); // 线程参数
58                      ca->home = home;
59                      ca->conn = conn;
60                      int error = pthread_create(&tid, &attr, client, ca);
61                      pthread_attr_destroy(&attr); // 销毁线程属性
62                      if (error) {
63                              fprintf(stderr, "pthread_cteate: %s\n", strerror(error));
64                              return -1;
65                      }
66              }
67              return 0;
68 }
69 // 终结化服务器
70 void deinitServer(void) {
71              // 终结化套接字
72              deinitSocket();
73 }
```

7)主模块(main)：main.c -> main.o

main.c

```
 1 // 实现主模块
 2 #include <stdio.h>
 3 #include <stdlib.h>
 4 #include "server.h"
 5 // ./WebServer 80 ../home
 6 // ./WebServer 80
 7 // ./WebServer
 8 int main(int argc, char* argv[]) {
 9              // 初始化服务器
10              if (initServer(argc < 2 ? 80 : atoi(argv[1])) == -1)
11                      return EXIT_FAILURE;
12              // 运行服务器
13              if (runServer(argc < 3 ? "../home" : argv[2]) == -1)
14                      return EXIT_FAILURE;
15              // 终结化服务器
16              deinitServer();
17              return EXIT_SUCCESS;
18 }
```

8)构建脚本：Makfile

```
1 PROJ     = WebServer
```

```
 2 OBJS    = main.o server.o signals.o client.o http.o sock.o resource.o
 3 CC      = gcc
 4 LINK    = gcc
 5 RM      = rm -rf
 6 CFLAGS = -c -Wall -I.
 7 LIBS    = -lpthread
 8 $(PROJ): $(OBJS)
 9         $(LINK) $^ $(LIBS) -o $@
10 .c.o:
11         $(CC) $(CFLAGS) $^
12 clean:
13         $(RM) $(PROJ) $(OBJS) *.gch
```

mime.h

```
 1 ////////////////////////////////////////////////////////////////////////
 2 // mime.h - MIME 类型
 3 //
 4 // 项目名称：WebServer (多线程版)
 5 // 研发单位：达内科技
 6 // 研发人员：闵卫
 7 // 创建时间：2017 年 5 月 17 日
 8 // 修改记录：
 9 //
10 #ifndef _MIME_H
11 #define _MIME_H
12
13 static struct {
14         char suffix[256];
15         char type[256];
16 }       s_mime[] = {
17         {"."          , "application/x-"                                  },
18         {".*"         , "application/octet-stream"                      },
19         {".001"       , "application/x-001"                               },
20         {".301"       , "application/x-301"                               },
21         {".323"       , "text/h323"                                         },
22         {".906"       , "application/x-906"                               },
23         {".907"       , "drawing/907"                                       },
24         {".acp"       , "audio/x-mei-aac"                                 },
25         {".ai"        , "application/postscript"                        },
26         {".aif"       , "audio/aiff"                                        },
27         {".aifc"      , "audio/aiff"                                      },
28         {".aiff"      , "audio/aiff"                                      },
29         {".a11"       , "application/x-a11"                               },
30         {".anv"       , "application/x-anv"                               },
31         {".apk"       , "application/vnd.android.package-archive"},
32         {".asa"       , "text/asa"                                          },
33         {".asf"       , "video/x-ms-asf"                                  },
```

```
34          {".asp"      ,  "text/asp"                                    },
35          {".asx"      ,  "video/x-ms-asf"                              },
36          {".au"       ,  "audio/basic"                                 },
37          {".avi"      ,  "video/avi"                                   },
38          {".awf"      ,  "application/vnd.adobe.workflow"            },
39          {".biz"      ,  "text/xml"                                    },
40          {".bmp"      ,  "application/x-bmp"                           },
41          {".bot"      ,  "application/x-bot"                          },
42          {".c4t"      ,  "application/x-c4t"                          },
43          {".c90"      ,  "application/x-c90"                          },
44          {".cal"      ,  "application/x-cals"                         },
45          {".cat"      ,  "application/vnd.ms-pki.seccat"              },
46          {".cdf"      ,  "application/x-netcdf"                       },
47          {".cdr"      ,  "application/x-cdr"                          },
48          {".cel"      ,  "application/x-cel"                          },
49          {".cer"      ,  "application/x-x509-ca-cert"                 },
50          {".cg4"      ,  "application/x-g4"                           },
51          {".cgm"      ,  "application/x-cgm"                          },
52          {".cit"      ,  "application/x-cit"                          },
53          {".class"    ,  "java/*"                                      },
54          {".cml"      ,  "text/xml"                                    },
55          {".cmp"      ,  "application/x-cmp"                           },
56          {".cmx"      ,  "application/x-cmx"                           },
57          {".cot"      ,  "application/x-cot"                          },
58          {".crl"      ,  "application/pkix-crl"                       },
59          {".crt"      ,  "application/x-x509-ca-cert"                 },
60          {".csi"      ,  "application/x-csi"                          },
61          {".css"      ,  "text/css"                                    },
62          {".cut"      ,  "application/x-cut"                          },
63          {".dbf"      ,  "application/x-dbf"                          },
64          {".dbm"      ,  "application/x-dbm"                           },
65          {".dbx"      ,  "application/x-dbx"                          },
66          {".dcd"      ,  "text/xml"                                    },
67          {".dcx"      ,  "application/x-dcx"                          },
68          {".der"      ,  "application/x-x509-ca-cert"                 },
69          {".dgn"      ,  "application/x-dgn"                           },
70          {".dib"      ,  "application/x-dib"                          },
71          {".dll"      ,  "application/x-msdownload"                   },
72          {".doc"      ,  "application/msword"                          },
73          {".dot"      ,  "application/msword"                          },
74          {".drw"      ,  "application/x-drw"                           },
75          {".dtd"      ,  "text/xml"                                    },
76          {".dwf"      ,  "application/x-dwf"                           },
77          {".dwg"      ,  "application/x-dwg"                           },
78          {".dxb"      ,  "application/x-dxb"                          },
79          {".dxf"      ,  "application/x-dxf"                          },
80          {".edn"      ,  "application/vnd.adobe.edn"                  },
```

```
81          {".emf"      ,  "application/x-emf"                    },
82          {".eml"      ,  "message/rfc822"                       },
83          {".ent"      ,  "text/xml"                             },
84          {".epi"      ,  "application/x-epi"                    },
85          {".eps"      ,  "application/x-ps"                     },
86          {".eps"      ,  "application/postscript"               },
87          {".etd"      ,  "application/x-ebx"                    },
88          {".exe"      ,  "application/x-msdownload"             },
89          {".fax"      ,  "image/fax"                            },
90          {".fdf"      ,  "application/vnd.fdf"                  },
91          {".fif"      ,  "application/fractals"                 },
92          {".fo"       ,  "text/xml"                             },
93          {".frm"      ,  "application/x-frm"                    },
94          {".g4"       ,  "application/x-g4"                     },
95          {".gbr"      ,  "application/x-gbr"                    },
96          {".gif"      ,  "image/gif"                            },
97          {".gl2"      ,  "application/x-gl2"                    },
98          {".gp4"      ,  "application/x-gp4"                    },
99          {".hgl"      ,  "application/x-hgl"                    },
100         {".hmr"      ,  "application/x-hmr"                    },
101         {".hpg"      ,  "application/x-hpgl"                   },
102         {".hpl"      ,  "application/x-hpl"                    },
103         {".hqx"      ,  "application/mac-binhex40"             },
104         {".hrf"      ,  "application/x-hrf"                    },
105         {".hta"      ,  "application/hta"                      },
106         {".htc"      ,  "text/x-component"                     },
107         {".htm"      ,  "text/html"                            },
108         {".html"     ,  "text/html"                            },
109         {".htt"      ,  "text/webviewhtml"                     },
110         {".htx"      ,  "text/html"                            },
111         {".icb"      ,  "application/x-icb"                    },
112         {".ico"      ,  "image/x-icon"                         },
113         {".iff"      ,  "application/x-iff"                    },
114         {".ig4"      ,  "application/x-g4"                     },
115         {".igs"      ,  "application/x-igs"                    },
116         {".iii"      ,  "application/x-iphone"                 },
117         {".img"      ,  "application/x-img"                    },
118         {".ins"      ,  "application/x-internet-signup"        },
119         {".ipa"      ,  "application/vnd.iphone"               },
120         {".isp"      ,  "application/x-internet-signup"        },
121         {".IVF"      ,  "video/x-ivf"                          },
122         {".java"     ,  "java/*"                               },
123         {".jfif"     ,  "image/jpeg"                           },
124         {".jpe"      ,  "image/jpeg"                           },
125         {".jpeg"     ,  "image/jpeg"                           },
126         {".jpg"      ,  "image/jpeg"                           },
127         {".js"       ,  "application/x-javascript"             },
```

```
128             {".jsp"       ,  "text/html"                                  },
129             {".la1"       ,  "audio/x-liquid-file"                      },
130             {".lar"       ,  "application/x-laplayer-reg"             },
131             {".latex"     ,  "application/x-latex"                      },
132             {".lavs"      ,  "audio/x-liquid-secure"                    },
133             {".lbm"       ,  "application/x-lbm"                          },
134             {".lmsff"     ,  "audio/x-la-lms"                             },
135             {".ls"        ,  "application/x-javascript"                 },
136             {".ltr"       ,  "application/x-ltr"                        },
137             {".m1v"       ,  "video/x-mpeg"                                 },
138             {".m2v"       ,  "video/x-mpeg"                                 },
139             {".m3u"       ,  "audio/mpegurl"                                },
140             {".m4e"       ,  "video/mpeg4"                                  },
141             {".mac"       ,  "application/x-mac"                          },
142             {".man"       ,  "application/x-troff-man"                  },
143             {".math"      ,  "text/xml"                                     },
144             {".mdb"       ,  "application/msaccess"                       },
145             {".mfp"       ,  "application/x-shockwave-flash"            },
146             {".mht"       ,  "message/rfc822"                               },
147             {".mhtml"     ,  "message/rfc822"                               },
148             {".mi"        ,  "application/x-mi"                           },
149             {".mid"       ,  "audio/mid"                                    },
150             {".midi"      ,  "audio/mid"                                    },
151             {".mil"       ,  "application/x-mil"                        },
152             {".mml"       ,  "text/xml"                                     },
153             {".mnd"       ,  "audio/x-musicnet-download"                },
154             {".mns"       ,  "audio/x-musicnet-stream"                  },
155             {".mocha"     ,  "application/x-javascript"                 },
156             {".movie"     ,  "video/x-sgi-movie"                          },
157             {".mp1"       ,  "audio/mp1"                                    },
158             {".mp2"       ,  "audio/mp2"                                    },
159             {".mp2v"      ,  "video/mpeg"                                   },
160             {".mp3"       ,  "audio/mp3"                                    },
161             {".mp4"       ,  "video/mpeg4"                                  },
162             {".mpa"       ,  "video/x-mpg"                                  },
163             {".mpd"       ,  "application/vnd.ms-project"               },
164             {".mpe"       ,  "video/x-mpeg"                                 },
165             {".mpeg"      ,  "video/mpg"                                    },
166             {".mpg"       ,  "video/mpg"                                    },
167             {".mpga"      ,  "audio/rn-mpeg"                              },
168             {".mpp"       ,  "application/vnd.ms-project"               },
169             {".mps"       ,  "video/x-mpeg"                                 },
170             {".mpt"       ,  "application/vnd.ms-project"               },
171             {".mpv"       ,  "video/mpg"                                    },
172             {".mpv2"      ,  "video/mpeg"                                   },
173             {".mpw"       ,  "application/vnd.ms-project"               },
174             {".mpx"       ,  "application/vnd.ms-project"               },
```

```
175             {".mtx"       , "text/xml"                                    },
176             {".mxp"       , "application/x-mmxp"                          },
177             {".net"       , "image/pnetvue"                               },
178             {".nrf"       , "application/x-nrf"                           },
179             {".nws"       , "message/rfc822"                              },
180             {".odc"       , "text/x-ms-odc"                               },
181             {".out"       , "application/x-out"                           },
182             {".p10"       , "application/pkcs10"                          },
183             {".p12"       , "application/x-pkcs12"                        },
184             {".p7b"       , "application/x-pkcs7-certificates"            },
185             {".p7c"       , "application/pkcs7-mime"                      },
186             {".p7m"       , "application/pkcs7-mime"                      },
187             {".p7r"       , "application/x-pkcs7-certreqresp"             },
188             {".p7s"       , "application/pkcs7-signature"                 },
189             {".pc5"       , "application/x-pc5"                           },
190             {".pci"       , "application/x-pci"                           },
191             {".pcl"       , "application/x-pcl"                           },
192             {".pcx"       , "application/x-pcx"                           },
193             {".pdf"       , "application/pdf"                             },
194             {".pdx"       , "application/vnd.adobe.pdx"                   },
195             {".pfx"       , "application/x-pkcs12"                        },
196             {".pgl"       , "application/x-pgl"                           },
197             {".pic"       , "application/x-pic"                           },
198             {".pko"       , "application/vnd.ms-pki.pko"                  },
199             {".pl"        , "application/x-perl"                          },
200             {".plg"       , "text/html"                                   },
201             {".pls"       , "audio/scpls"                                 },
202             {".plt"       , "application/x-plt"                           },
203             {".png"       , "image/png"                                   },
204             {".pot"       , "application/vnd.ms-powerpoint"               },
205             {".ppa"       , "application/vnd.ms-powerpoint"               },
206             {".ppm"       , "application/x-ppm"                           },
207             {".pps"       , "application/vnd.ms-powerpoint"               },
208             {".ppt"       , "application/vnd.ms-powerpoint"               },
209             {".ppt"       , "application/x-ppt"                           },
210             {".pr"        , "application/x-pr"                            },
211             {".prf"       , "application/pics-rules"                      },
212             {".prn"       , "application/x-prn"                           },
213             {".prt"       , "application/x-prt"                           },
214             {".ps"        , "application/x-ps"                            },
215             {".ps"        , "application/postscript"                      },
216             {".ptn"       , "application/x-ptn"                           },
217             {".pwz"       , "application/vnd.ms-powerpoint"               },
218             {".r3t"       , "text/vnd.rn-realtext3d"                      },
219             {".ra"        , "audio/vnd.rn-realaudio"                      },
220             {".ram"       , "audio/x-pn-realaudio"                        },
221             {".ras"       , "application/x-ras"                           },
```

```
222             {".rat"       , "application/rat-file"                          },
223             {".rdf"       , "text/xml"                                          },
224             {".rec"       , "application/vnd.rn-recording"                  },
225             {".red"       , "application/x-red"                                 },
226             {".rgb"       , "application/x-rgb"                                 },
227             {".rjs"       , "application/vnd.rn-realsystem-rjs"           },
228             {".rjt"       , "application/vnd.rn-realsystem-rjt"           },
229             {".rlc"       , "application/x-rlc"                                 },
230             {".rle"       , "application/x-rle"                                 },
231             {".rm"        , "application/vnd.rn-realmedia"                  },
232             {".rmf"       , "application/vnd.adobe.rmf"                     },
233             {".rmi"       , "audio/mid"                                             },
234             {".rmj"       , "application/vnd.rn-realsystem-rmj"           },
235             {".rmm"       , "audio/x-pn-realaudio"                              },
236             {".rmp"       , "application/vnd.rn-rn_music_package"       },
237             {".rms"       , "application/vnd.rn-realmedia-secure"       },
238             {".rmvb"      , "application/vnd.rn-realmedia-vbr"          },
239             {".rmx"       , "application/vnd.rn-realsystem-rmx"         },
240             {".rnx"       , "application/vnd.rn-realplayer"                 },
241             {".rp"        , "image/vnd.rn-realpix"                              },
242             {".rpm"       , "audio/x-pn-realaudio-plugin"                   },
243             {".rsml"      , "application/vnd.rn-rsml"                       },
244             {".rt"        , "text/vnd.rn-realtext"                          },
245             {".rtf"       , "application/msword"                                },
246             {".rtf"       , "application/x-rtf"                             },
247             {".rv"        , "video/vnd.rn-realvideo"                        },
248             {".sam"       , "application/x-sam"                                 },
249             {".sat"       , "application/x-sat"                             },
250             {".sdp"       , "application/sdp"                                   },
251             {".sdw"       , "application/x-sdw"                                 },
252             {".sis"       , "application/vnd.symbian.install"             },
253             {".sisx"      , "application/vnd.symbian.install"             },
254             {".sit"       , "application/x-stuffit"                         },
255             {".slb"       , "application/x-slb"                                 },
256             {".sld"       , "application/x-sld"                                 },
257             {".slk"       , "drawing/x-slk"                                     },
258             {".smi"       , "application/smil"                                  },
259             {".smil"      , "application/smil"                                  },
260             {".smk"       , "application/x-smk"                                 },
261             {".snd"       , "audio/basic"                                       },
262             {".sol"       , "text/plain"                                        },
263             {".sor"       , "text/plain"                                        },
264             {".spc"       , "application/x-pkcs7-certificates"            },
265             {".spl"       , "application/futuresplash"                      },
266             {".spp"       , "text/xml"                                              },
267             {".ssm"       , "application/streamingmedia"                    },
268             {".sst"       , "application/vnd.ms-pki.certstore"            },
```

```
269             {".stl"       , "application/vnd.ms-pki.stl"                },
270             {".stm"       , "text/html"                                     },
271             {".sty"       , "application/x-sty"                          },
272             {".svg"       , "text/xml"                                      },
273             {".swf"       , "application/x-shockwave-flash"           },
274             {".tdf"       , "application/x-tdf"                          },
275             {".tg4"       , "application/x-tg4"                          },
276             {".tga"       , "application/x-tga"                          },
277             {".tif"       , "image/tiff"                                    },
278             {".tiff"      , "image/tiff"                                    },
279             {".tld"       , "text/xml"                                      },
280             {".top"       , "drawing/x-top"                                 },
281             {".torrent", "application/x-bittorrent"                 },
282             {".tsd"       , "text/xml"                                      },
283             {".ttf"       , "application/font-woff"                      },
284             {".txt"       , "text/plain"                                    },
285             {".uin"       , "application/x-icq"                          },
286             {".uls"       , "text/iuls"                                     },
287             {".vcf"       , "text/x-vcard"                                  },
288             {".vda"       , "application/x-vda"                          },
289             {".vdx"       , "application/vnd.visio"                      },
290             {".vml"       , "text/xml"                                      },
291             {".vpg"       , "application/x-vpeg005"                      },
292             {".vsd"       , "application/vnd.visio"                      },
293             {".vsd"       , "application/x-vsd"                          },
294             {".vss"       , "application/vnd.visio"                      },
295             {".vst"       , "application/vnd.visio"                      },
296             {".vst"       , "application/x-vst"                          },
297             {".vsw"       , "application/vnd.visio"                      },
298             {".vsx"       , "application/vnd.visio"                      },
299             {".vtx"       , "application/vnd.visio"                      },
300             {".vxml"      , "text/xml"                                      },
301             {".wav"       , "audio/wav"                                     },
302             {".wax"       , "audio/x-ms-wax"                                },
303             {".wb1"       , "application/x-wb1"                          },
304             {".wb2"       , "application/x-wb2"                          },
305             {".wb3"       , "application/x-wb3"                          },
306             {".wbmp"      , "image/vnd.wap.wbmp"                            },
307             {".wiz"       , "application/msword"                         },
308             {".wk3"       , "application/x-wk3"                          },
309             {".wk4"       , "application/x-wk4"                          },
310             {".wkq"       , "application/x-wkq"                          },
311             {".wks"       , "application/x-wks"                          },
312             {".wm"        , "video/x-ms-wm"                                 },
313             {".wma"       , "audio/x-ms-wma"                                },
314             {".wmd"       , "application/x-ms-wmd"                       },
315             {".wmf"       , "application/x-wmf"                          },
```

```
316             {".wml"       ,  "text/vnd.wap.wml"                          },
317             {".wmv"       ,  "video/x-ms-wmv"                              },
318             {".wmx"       ,  "video/x-ms-wmx"                              },
319             {".wmz"       ,  "application/x-ms-wmz"                      },
320             {".wp6"       ,  "application/x-wp6"                         },
321             {".wpd"       ,  "application/x-wpd"                         },
322             {".wpg"       ,  "application/x-wpg"                         },
323             {".wpl"       ,  "application/vnd.ms-wpl"                    },
324             {".wq1"       ,  "application/x-wq1"                         },
325             {".wri"       ,  "application/x-wri"                         },
326             {".wrk"       ,  "application/x-wrk"                         },
327             {".wr1"       ,  "application/x-wr1"                         },
328             {".ws"        ,  "application/x-ws"                          },
329             {".ws2"       ,  "application/x-ws"                          },
330             {".wsc"       ,  "text/scriptlet"                            },
331             {".wsdl"      ,  "text/xml"                                    },
332             {".wvx"       ,  "video/x-ms-wvx"                              },
333             {".xap"       ,  "application/x-silverlight-app"           },
334             {".xdp"       ,  "application/vnd.adobe.xdp"                 },
335             {".xdr"       ,  "text/xml"                                    },
336             {".xfd"       ,  "application/vnd.adobe.xfd"               },
337             {".xfdf"      ,  "application/vnd.adobe.xfdf"              },
338             {".xhtml"     ,  "text/html"                                   },
339             {".xls"       ,  "application/vnd.ms-excel"                },
340             {".xls"       ,  "application/x-xls"                         },
341             {".xlw"       ,  "application/x-xlw"                           },
342             {".xml"       ,  "text/xml"                                      },
343             {".xpl"       ,  "audio/scpls"                                 },
344             {".xq"        ,  "text/xml"                                      },
345             {".xql"       ,  "text/xml"                                    },
346             {".xquery"    ,  "text/xml"                                    },
347             {".xsd"       ,  "text/xml"                                    },
348             {".xsl"       ,  "text/xml"                                    },
349             {".xslt"      ,  "text/xml"                                    },
350             {".xwd"       ,  "application/x-xwd"                           },
351             {".x_b"       ,  "application/x-x_b"                         },
352             {".x_t"       ,  "application/x-x_t"                         }
353 };
354
355 #endif // _MIME_H
```