文件系统

目的：类ls的实现，如myls

```
ls -n
ls -l
对应 /etc/passwd 和 /etc/group两个函数
```

1.目录和文件

1.1 获取文件属性

- stat：通过文件路径获取属性，面对符号链接文件时获取的是所指向的目标文件。
- fstat：通过文件描述符获取属性
- lstat：面对符号链接文件时获取的是符号链接文件的属性
- stat 、fstat、lstat函数

```
NAME
       stat, fstat, lstat, fstatat - get file status

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

       int stat(const char *pathname, struct stat *statbuf);
       int fstat(int fd, struct stat *statbuf);
       int lstat(const char *pathname, struct stat *statbuf);
DESCRIPTION
       These  functions return information about a file, in the buffer
pointed to by statbuf.  No permissions are required on the file itself, but-
in the case of stat(), fstatat(), and lstat()-execute (search) permission is
required on all of the directories in pathname that lead to the file.
       stat() retrieve information about the file pointed to by pathname;
       lstat() is identical to stat(), except that if pathname is a symbolic
link, then it returns information about the link itself, not the file that it
refers to.
       fstat() is identical to stat(), except that the file about which
information is to be retrieved is specified by the file descriptor fd。

The stat structure
    struct stat {
                dev_t     st_dev;         /* ID of device containing file */
                ino_t     st_ino;         /* Inode number */
                mode_t    st_mode;        /* File type and mode */
                nlink_t   st_nlink;       /* Number of hard links */
                uid_t     st_uid;         /* User ID of owner */
                gid_t     st_gid;         /* Group ID of owner */
                dev_t     st_rdev;        /* Device ID (if special file) */
```

```
               off_t     st_size;        /* Total size, in bytes */
               blksize_t st_blksize;     /* Block size for filesystem I/O */
               blkcnt_t  st_blocks;      /* Number of 512B blocks allocated
*/

               /* Since Linux 2.6, the kernel supports nanosecond
                  precision for the following timestamp fields.
                  For the details before Linux 2.6, see NOTES. */

               struct timespec st_atim;  /* Time of last access */
               struct timespec st_mtim;  /* Time of last modification */
               struct timespec st_ctim;  /* Time of last status change */

               #define st_atime st_atim.tv_sec      /* Backward compatibility
*/
               #define st_mtime st_mtim.tv_sec
               #define st_ctime st_ctim.tv_sec
           };
```

小功能：利用stat结构体重的 st_size来得到一个文件的大小

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

/**
 *  通过stat structure，返回文件的长度
*/
static off_t flen(const char *fname)
{
    struct stat statres;

    if(stat(fname,&statres) < 0)
    {
        perror("stat()");
    }

    return statres.st_size;
}
int main(int argc,char** argv)
{
    if(argc < 2)
    {
        fprintf(stderr,"Usage:%s <filename>...\n",argv[0]);
        exit(1);
    }

    printf("%ld\n",flen(argv[1]));
    exit(0);
}
```

```
marz@ubuntu1:~/cpp/io/filesystem$ ls
flen  flen.c  makefile
marz@ubuntu1:~/cpp/io/filesystem$ ./flen flen.c
552
```

- 通过stat命令也可以得到文件的信息：

```
marz@ubuntu1:~/cpp/io/filesystem$ stat flen.c
  File: flen.c
  Size: 552             Blocks: 8          IO Block: 4096   regular file
Device: 805h/2053d      Inode: 657150      Links: 1
Access: (0664/-rw-rw-r--)  Uid: ( 1000/    marz)   Gid: ( 1000/    marz)
Access: 2023-03-29 06:59:43.654497589 -0700
Modify: 2023-03-29 06:59:43.631800857 -0700
Change: 2023-03-29 06:59:43.626497011 -0700
 Birth: -
marz@ubuntu1:~/cpp/io/filesystem$
```

size是st_size的数值，我们的小功能函数也验证了这一点。**值得注意的是**，st_size仅仅一个参数，类似st_ino这样，文件的实际存储大小是由

st_blocks*st_blksize的大小决定的，这和文件系统有关系（linux是这样的，win不太一样）。

文件系统块大小：

```
marz@ubuntu1:~/cpp/io/filesystem$ sudo fdisk -l
Disk /dev/loop0: 4 KiB, 4096 bytes, 8 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
```

小功能：用程序生一个st_size非常大，但是磁盘空间非常小的文件。

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

/**
 *
 * 做一个totalsize非常大，但是磁盘空间非常小的文件
 * 做一个5G大小的文件
*/
int main(int argc,char **argv)
{
    int fd;
    off_t res;
    if(argc < 2)
    {
        fprintf(stderr,"Usage:%s <filename>\n",argv[0]);
        exit(0);
    }

    fd = open(argv[1],O_WRONLY|O_CREAT|O_TRUNC,0600);
    if(fd < 0)
    {
```

```
        perror("open");
        exit(1);
    }

    res = lseek(fd,5L*1024L*1024L*1024L-1L,SEEK_SET);
    if(res< 5L*1024L*1024L*1024L-1L)
    {
        perror("lseek");
        exit(1);
    }

    write(fd,"",1);
    close(fd);
    exit(0);
}
```

```
marz@ubuntu1:~/cpp/io/filesystem$ ./big /tmp/filename
marz@ubuntu1:~/cpp/io/filesystem$ ls -l /tmp/filename
-rw------- 1 marz marz 5368709120 Mar 29 07:57 /tmp/filename
marz@ubuntu1:~/cpp/io/filesystem$ stat /tmp/filename
  File: /tmp/filename
  Size: 5368709120      Blocks: 8          IO Block: 4096   regular file
Device: 805h/2053d      Inode: 2359522     Links: 1
Access: (0600/-rw-------)  Uid: ( 1000/    marz)   Gid: ( 1000/    marz)
Access: 2023-03-29 07:56:34.693377820 -0700
Modify: 2023-03-29 07:57:01.121556240 -0700
Change: 2023-03-29 07:57:01.121556240 -0700
 Birth: -
```

这样就生成了一个size很大，但是占用内存仅4k的一个文件。

这个程序的注意点：

> 输入5*1024*1024*1024-1 会报数据溢出的警告，所以将数据扩张到Long类型，所以就需要在后面跟上L
> 5L*1024L*1024L*1024L-1L

总结：linux环境下，size值只是一个属性而已。

1.2 st_mode

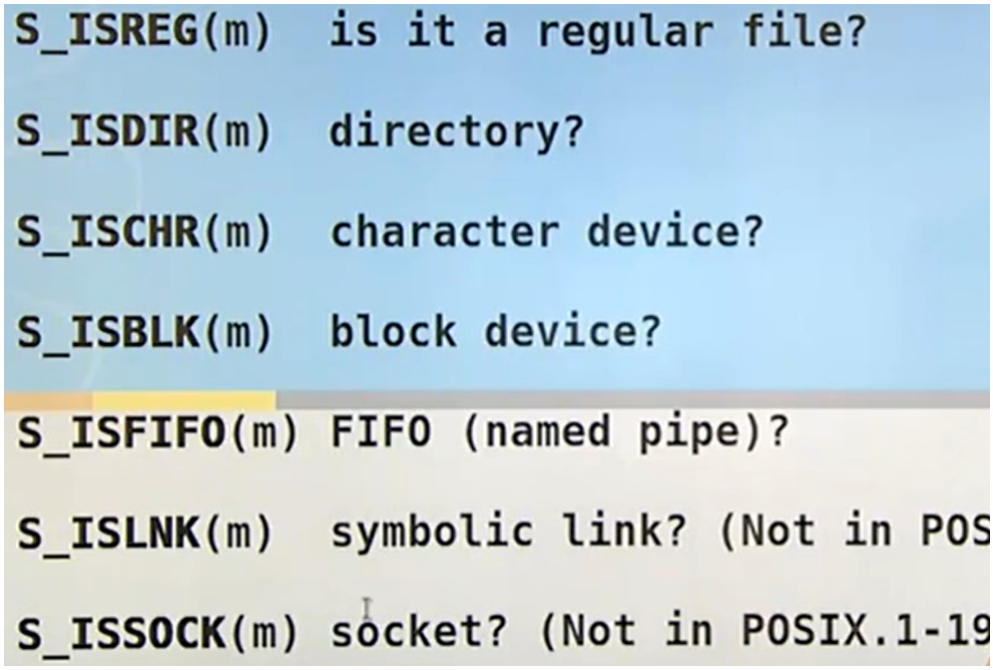
st_mode是一个16位的位图，用于表示文件类型、文件访问权限及特殊权限位。

```
marz@ubuntu1:~$ ll
total 124
drwxrwxrwx 19 marz marz  4096 Mar 30 05:26 ./
drwxr-xr-x  3 root root  4096 Feb 23 00:25 ../
-rwxrwxrwx  1 marz marz  9206 Mar 29 08:12 .bash_history
-rwxrwxrwx  1 marz marz   220 Feb 23 00:25 .bash_logout*
-rwxrwxrwx  1 marz marz  3847 Mar 11 18:59 .bashrc*
drwxrwxr-x  3 marz marz  4096 Mar 25 07:49 c/
drwxrwxrwx 13 marz marz  4096 Mar  4 19:36 .cache/
drwxrwxrwx 11 marz marz  4096 Feb 23 05:19 .config/
```

前面的权限就是由st_mode来决定的。

文件类型：dcb-lsp

d:directory
c:字符设备文件
b:block，块设备文件
-：普通文件（regular fil
l:link，符号链接(symbol link)文件
s:socket file，网络套接字文件
p:pipe，管道文件，在这里特指的是匿名管道文件，匿名管道文件在磁盘上看不到。




```
S_IFMT     0170000   bit mask for the file type bit field

S_IFSOCK   0140000   socket

S_IFLNK    0120000   symbolic link
S_IFREG    0100000   regular file
S_IFBLK    0060000   block device
S_IFDIR    0040000   directory
S_IFCHR    0020000   character device
S_IFIFO    0010000   FIFO
```

```
The following mask values are defined for the file mode component of the st_mode field:

    S_ISUID     04000   set-user-ID bit (see execve(2))
    S_ISGID     02000   set-group-ID bit (see below)
    S_ISVTX     01000   sticky bit (see below)

    S_IRWXU     00700   owner has read, write, and execute permission
    S_IRUSR     00400   owner has read permission
    S_IWUSR     00200   owner has write permission
    S_IXUSR     00100   owner has execute permission

    S_IRWXG     00070   group has read, write, and execute permission
    S_IRGRP     00040   group has read permission
    S_IWGRP     00020   group has write permission
    S_IXGRP     00010   group has execute permission

    S_IRWXO     00007   others  (not  in group) have read, write, and
                        execute permission
    S_IROTH     00004   others have read permission
    S_IWOTH     00002   others have write permission
    S_IXOTH     00001   others have execute permission
```

1.3 umask

作用：为了防止产生权限过松的文件。

1.4 文件权限的更改/管理

- chmod
- fchmod

1.5 粘住位

最原始的定义：给某一个可执行的二进制文件设置t位，在内存中保留它使用的痕迹。

t位：(了解)

```
marz@ubuntu1:/$ ls -all
total 1918448
drwxr-xr-x  20 root root       4096 Feb 23 00:24 .
drwxr-xr-x  20 root root       4096 Feb 23 00:24 ..
lrwxrwxrwx   1 root root          7 Feb 23 00:19 bin -> usr/bin
drwxr-xr-x   4 root root       4096 Mar 28 06:47 boot
drwxrwxr-x   2 root root       4096 Feb 23 00:24 cdrom
drwxr-xr-x  19 root root       4200 Apr  4 08:08 dev
drwxr-xr-x 133 root root      12288 Mar 28 06:47 etc
drwxr-xr-x   3 root root       4096 Feb 23 00:25 home
lrwxrwxrwx   1 root root          7 Feb 23 00:19 lib -> usr/lib
lrwxrwxrwx   1 root root          9 Feb 23 00:19 lib32 -> usr/lib32
lrwxrwxrwx   1 root root          9 Feb 23 00:19 lib64 -> usr/lib64
lrwxrwxrwx   1 root root         10 Feb 23 00:19 libx32 -> usr/libx32
drwx------   2 root root      16384 Feb 23 00:19 lost+found
drwxr-xr-x   3 root root       4096 Aug 30  2022 media
drwxr-xr-x   2 root root       4096 Aug 30  2022 mnt
drwxr-xr-x   3 root root       4096 Mar  5 04:33 opt
dr-xr-xr-x 362 root root          0 Apr  4 08:08 proc
drwx------   8 root root       4096 Mar 27 06:06 root
drwxr-xr-x  34 root root        940 Apr  4 23:21 run
lrwxrwxrwx   1 root root          8 Feb 23 00:19 sbin -> usr/sbin
drwxr-xr-x  10 root root       4096 Feb 23 01:38 snap
drwxr-xr-x   2 root root       4096 Aug 30  2022 srv
-rw-------   1 root root 1964400640 Feb 23 00:19 swapfile
dr-xr-xr-x  13 root root          0 Apr  4 08:08 sys
drwxrwxrwt  19 root root       4096 Apr  4 23:18 tmp
drwxr-xr-x  14 root root       4096 Aug 30  2022 usr
drwxr-xr-x  14 root root       4096 Aug 31  2022 var
```

1.6 文件系统：FAT，UFS

文件系统：文件或数据的存储和管理。

1.7硬链接，符号链接

```
marz@ubuntu1:/tmp$ stat bigfile
  File: bigfile
  Size: 114             Blocks: 8          IO Block: 4096   regular file
Device: 805h/2053d      Inode: 2359897     Links: 1
Access: (0664/-rw-rw-r--)  Uid: ( 1000/    marz)   Gid: ( 1000/    marz)
Access: 2023-04-05 08:09:07.166590232 -0700
Modify: 2023-04-05 08:09:18.638365807 -0700
Change: 2023-04-05 08:09:18.638365807 -0700
 Birth: -
```

- 硬链接

```
ln [参数] [源文件或目录] [目标文件或目录]
命令的功能是为某一个文件在另外一个位置建立一个同步的链接
```

使用 “ln bigfile bigfile_link” 创建连接

```
marz@ubuntu1:/tmp$ stat bigfile
  File: bigfile
  Size: 114             Blocks: 8          IO Block: 4096   regular file
Device: 805h/2053d      Inode: 2359897     Links: 2
Access: (0664/-rw-rw-r--)  Uid: ( 1000/    marz)   Gid: ( 1000/    marz)
Access: 2023-04-05 08:09:07.166590232 -0700
Modify: 2023-04-05 08:09:18.638365807 -0700
Change: 2023-04-05 08:09:57.365608207 -0700
 Birth: -
```

```
marz@ubuntu1:/tmp$ stat bigfile_link
  File: bigfile_link
  Size: 114             Blocks: 8          IO Block: 4096   regular file
Device: 805h/2053d      Inode: 2359897     Links: 2
Access: (0664/-rw-rw-r--)  Uid: ( 1000/    marz)   Gid: ( 1000/    marz)
Access: 2023-04-05 08:09:07.166590232 -0700
Modify: 2023-04-05 08:09:18.638365807 -0700
Change: 2023-04-05 08:09:57.365608207 -0700
 Birth: -
```

通过ln bigfile bigfile_link,删除bigfile源文件，bigfile_link依然可以正常使用。

硬链接：两个指针指向同一个文件。是**目录项**的同义词，且建立硬链接有限制，不能给分区简历，不能给目录简历

- 符号（symbol）连接

  符号链接：可以跨分区，可以给目录简历。

  ```
  ln -s [源文件或目录] [目标文件或目录]
  ```

```
lrwxrwxrwx  1 marz marz    4 Apr  6 07:02 te_S -> test
-rw-rw-r--  1 marz marz   94 Apr  6 07:01 test
```

```
marz@ubuntu1:/tmp$ stat test
  File: test
  Size: 94              Blocks: 8          IO Block: 4096   regular file
Device: 805h/2053d      Inode: 2359700     Links: 1
Access: (0664/-rw-rw-r--)  Uid: ( 1000/    marz)   Gid: ( 1000/    marz)
Access: 2023-04-06 07:01:46.059226077 -0700
Modify: 2023-04-06 07:01:54.047270753 -0700
Change: 2023-04-06 07:01:54.047270753 -0700
 Birth: -
marz@ubuntu1:/tmp$ stat te_S
  File: te_S -> test
  Size: 4               Blocks: 0          IO Block: 4096   symbolic link
Device: 805h/2053d      Inode: 2359898     Links: 1
Access: (0777/lrwxrwxrwx)  Uid: ( 1000/    marz)   Gid: ( 1000/    marz)
Access: 2023-04-06 07:02:34.299518035 -0700
Modify: 2023-04-06 07:02:32.791508149 -0700
Change: 2023-04-06 07:02:32.791508149 -0700
 Birth: -
```

  删除源文件后，符号链接变得不可用。

```
lrwxrwxrwx  1 marz marz    4 Apr  6 07:02 te_S -> test
```

涉及函数：

- link
- unlink
- remove
- rename

1.8 utime

可以更改文件最后读的时间和最后修改的时间

1.9 目录的创建和销毁

涉及函数：

- mkdir()
- rmdir()

1.10 更改当前工作路径

涉及函数：

```
- chdir(),cd函数是有该函数封装得到的。
- fchdir()
- getcwd(), 封装出来的命令 pwd
```

1.11分析目录和读取目录内容

- opendir
- closedir
- readdir
- rewinddir
- seekdir
- telldir
- glob：解析模式/通配符

  glob，可以实现上面函数的功能。

小功能：

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

int  main(int argc,int **argv)
{
    printf("argc = %d\n",argc);

    exit(0);
}
```

```
marz@ubuntu1:~/cpp/io/filesystem$ ./main
argc = 1
marz@ubuntu1:~/cpp/io/filesystem$ ./main hello world 123 908
argc = 5
marz@ubuntu1:~/cpp/io/filesystem$ ls
big.c  flen.c  ftype  ftype.c  main  main.c  makefile
marz@ubuntu1:~/cpp/io/filesystem$ ./main *.c
argc = 5
marz@ubuntu1:~/cpp/io/filesystem$ ./main big.c flen.c ftype.c main.c
```

这里有一个问题，统计数据参数个数的函数在输入 "*.c"的时候，其结果是多少?

“*”就是一个通配符。

小功能：通过glob函数读取/etc/下的文件

```
#include<stdio.h>
#include<stdlib.h>
#include<glob.h>
//#define PATTERN "/etc/a*.conf"
#define PATTERN "/etc/*"
// 统计/etc目录下有多少以a*.conf文件

#if 0
int errfunc(cosnt char* epath,int eerrno)
{
    puts(epath);
    fprintf(strerr,"ERR MSG:%s",strerror(eerrno));
    return errno;
}

#endif
int main(int argc,char **argv)
{
    glob_t globers;
    int err = glob(PATTERN,0,NULL,&globers);
    if(err)
    {// 出错
        printf("Error code = %d\n",err);
        exit(1);
    }

    for(int i=0;i<globers.gl_pathc;i++)
    {
        puts(globers.gl_pathv[i]);
    }

    globfree(&globers);
    exit(0);
}
```

小功能：用opendir、readdir、closedir实现/etc/下文件的读取

```
#include<stdio.h>
#include<stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <dirent.h>


int main()
{
    DIR* dir;
    struct dirent* rdd;
    dir = opendir("/etc/");
    if(dir == NULL)
    {
        closedir(dir);
        printf("opendir failed!!!\n");
        fprintf(stderr,"opendir failed,%s",strerror(errno));
        exit(1);
    }

    int count = 0;
    while((rdd = readdir(dir)) != NULL)
    {
        printf("%s\n",rdd->d_name);
        count++;
    }

    printf("======================\n");
    fprintf(stdout,"total files number is %d\n",count);

    closedir(dir);
    exit(0);
}
```

2.系统数据文件和信息

2.1 /etc/passwd文件

```
相关的函数：
    getpwuid()
    getpwnam()
```

```
NAME
       getpwnam, getpwnam_r, getpwuid, getpwuid_r - get password file entry

SYNOPSIS
       #include <sys/types.h>
       #include <pwd.h>

       struct passwd *getpwnam(const char *name);
       struct passwd *getpwuid(uid_t uid);
通过uid和用户名查询用户的所有信息
```

不同的系统并不一定有/etc/passwd文件，所以这个还是要看具体系统的存放方式，linux系统中是有的。

```
marz@ubuntu1:~/cpp/io/filesystem$ vim /etc/passwd

  1 root:x:0:0:root:/root:/bin/bash
 45 sssd:x:126:131:SSSD system user,,,:/var/lib/sss:/usr/sbin/nologin
 46 marz:x:1000:1000:marz,,,:/home/marz:/bin/bash
 47 systemd-coredump:x:999:999:systemd Core Dumper:/:/usr/sbin/nologin
```

x代表passwd，是通过加密后的信息。

小功能：通过uid来获取用户名

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

// 从命令行输入uid，打印出username
int main(int argc,char **argv)
{
    struct  passwd *pwdline;
    if(argc < 2)
    {
        fprintf(stderr,"Error,Usage:%s <uid>\n",argv[0]);
        exit(1);
    }

    pwdline = getpwuid(atoi(argv[1]));
    puts(pwdline->pw_name);
    exit(0);
}
```

```
marz@ubuntu1:~/cpp/io/filesystem$ ./username 1000
marz
marz@ubuntu1:~/cpp/io/filesystem$ ./username 0
root
```

2.2 /etc/group

```
相关函数
    getgrgid();
    getgrgrnam();
```

```
NAME
       getgrnam, getgrnam_r, getgrgid, getgrgid_r - get group file entry

SYNOPSIS
       #include <sys/types.h>
       #include <grp.h>

       struct group *getgrnam(const char *name);
       struct group *getgrgid(gid_t gid);
```

2.3 /etc/shadow

```
相关函数：
    getspnam()
    getspent()
    crypt()加密函数
    getpass()
```

```
GETSPNAM(3)                                                Linux Programmer's Manual                                               GETSPNAM(3)

NAME
       getspnam, getspnam_r, getspent, getspent_r, setspent, endspent, fgetspent, fgetspent_r, sgetspent, sgetspent_r, putspent, lckp-
       wdf, ulckpwdf - get shadow password file entry

SYNOPSIS
       /* General shadow password file API */
       #include <shadow.h>

       struct spwd *getspnam(const char *name);
       struct spwd *getspent(void);
```

2.4 时间戳

```
涉及函数：
    time()
    gmtime
    localtime()
    mktime()
    strftime()
时间戳的类型：time_t、char *、struct tm
上面的时间函数是在这三个类型中切换的函数
```

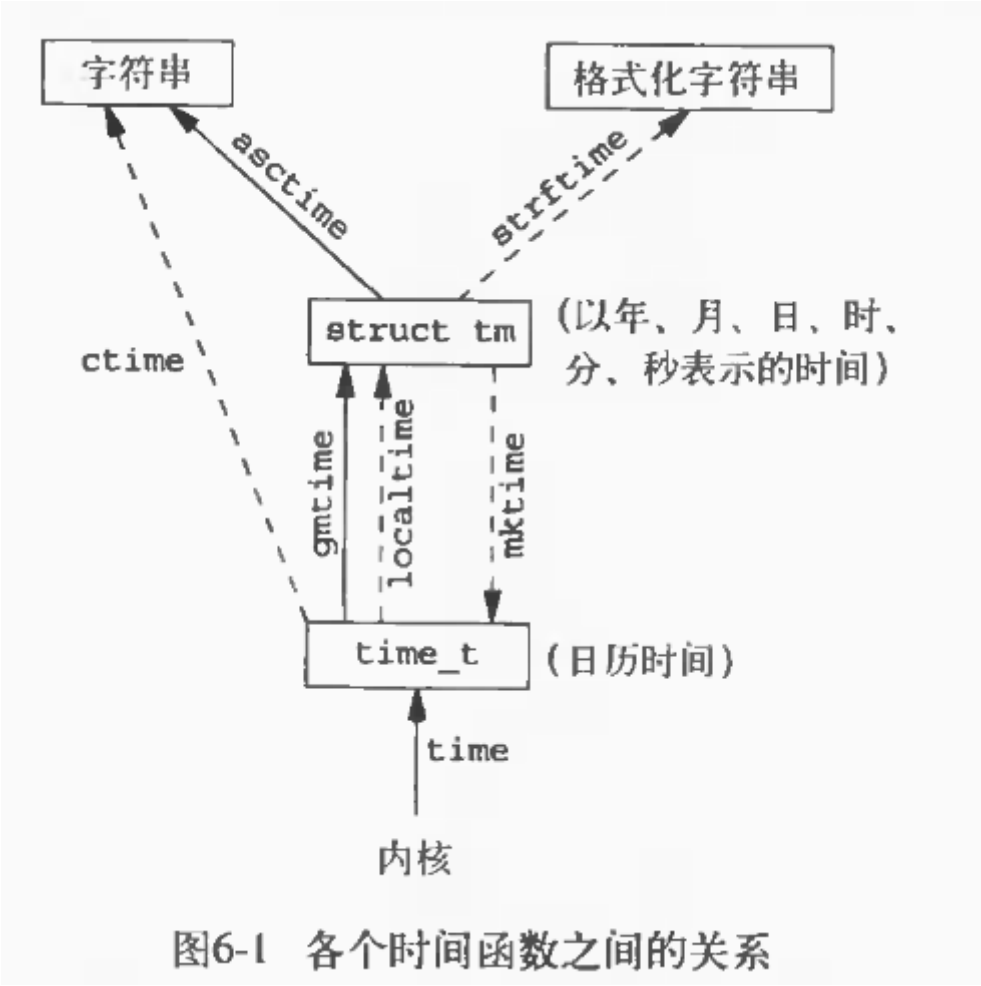


图6-1 各个时间函数之间的关系

这里在首次调用./timelog的时候，去查看/tmp/out文件，查看不到任何内容，这是因为：

在IO中，除了终端设备，其他都是全缓冲模式，所以fprintf中的"\n"不能起到刷新缓冲区的作用了。

3.进程环境

3.1 main函数

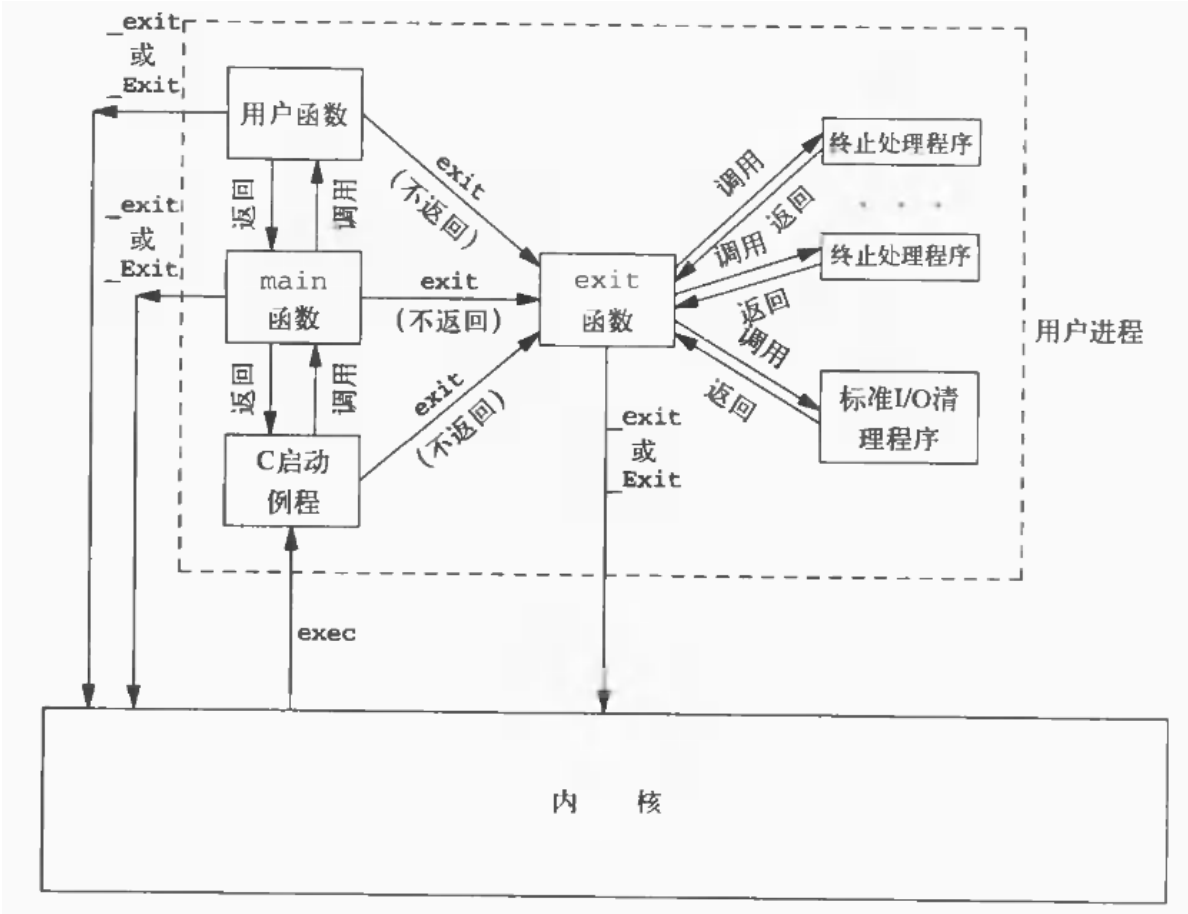
```
int main(int argc,char*argv[] )
```

3.2 进程终止

- 正常终止：
  - 从main函数返回
  - 调用exit
  - 调用_exit或_EXIT
  - 最后一个线程从其启动例程返回
  - 最后一个线程调用pthread_exit
- 异常终止：
  - 调用abort函数
  - 接到一个信号并终止
  - 最后一个线程对其取消请求作出响应

```
exit与_exit的区别：
exit是库函数，而_exit是系统函数。
```

- **exit与_exit的区别：**



图片解读：调用_exit会导致当前进程终止，而调用exit，会先终止处理程序（比如钩子函数）和标准IO清理程序，然后才会调用_exit函数 。

- exit与_exit调用时机的问题：

```
当出现错误的时候，不需要或者不敢做任何操作的时候，我们就调用_Exit函数。
当出现错误的时候，我们需要保存数据，刷新文件等等操作的时候，我们就调用exit函数。
```

3.2.1 钩子函数atexit(3)

进程正常终止的时候，在钩子函数会被调用。

类似c++中的析构函数

```
NAME
       atexit - register a function to be called at normal process termination
SYNOPSIS
       #include <stdlib.h>
       int atexit(void (*function)(void));
```

案例：

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

static void f1(void)
{
    puts("f1() is working!!!");
}

static void f2(void)
{
    puts("f2() is working");
}

static void f3(void)
{
    puts("f3() is working");
}
int main()
{
    puts("Begin!");

    atexit(f1);
    atexit(f2);
    atexit(f3);

    puts("End!");
    exit(0);

}
```

```
marz@ubuntu:~/cpp/io/filesystem$ make atexit
cc -D_FILE_OFFSET_BITS=64  -lcrypt  atexit.c   -o atexit
marz@ubuntu:~/cpp/io/filesystem$ ./atexit
Begin!
End!
f3() is working
f2() is working
f1() is working!!!
```

钩子函数的使用场景:

当打开多个文件的时候，如果文件打开是失败，在fd100处要关闭很多进程。这个时候我们就可以使用钩子函数

```c
fd1 = open();
if(fd1<0)
{
    perror();
    exit(1);
}
fd2 = open();
if(fd2 <0)
{
    close(fd1);
```

```
    perror();
    exit(1);
}
...
fd100 = open();
if(fd100<0)
{
    close(fd1);
    close(fd2);
    ...
    close(fd99);
    perror();
    exit(1);
}
```

改进

```
fd1 = open();
if(fd1<0)
{
    perror();
    exit(1);
}
atexit();     --->   close(fd1)
fd2 = open();
if(fd2 <0)
{
    perror();
    exit(1);
}
atexit();     --->   close(fd2)
...
fd100 = open();
if(fd100<0)
{
    perror();
    exit(1);
}

atexit();     --->   close(fd2)
```

除了open，还有malloc等等，只要用到申请资源的函数，下面就可以挂上钩子函数。

3.3 命令行参数的分析

相关函数：

```
getopt()
getopt_long()
```

```
NAME
       getopt, getopt_long, getopt_long_only, optarg, optind, opterr, optopt -
Parse command-line options

SYNOPSIS
       #include <unistd.h>
       int getopt(int argc, char * const argv[],const char *optstring);
       extern char *optarg;
       extern int optind, opterr, optopt;
描述：
    optstring是包含合法的选项字符的字符串，如果这样的字符后门跟这":",说明这个选项需要一个参数，所以在optarg中，getopt()函数在argv参数中用指针指向接下来的内容，或者是接下来的argv-element的文本。两个":"意味着一个选项带着一个可选参数。如果当前argv参数中
```

3.4 环境变量

环境变量类似于全局变量。

- 输出自己系统的环境变量

```c
#include<stdio.h>
#include<stdlib.h>
extern char **environ;
int main()
{

    for(int i=0;environ[i] != NULL;i++)
    {
        puts(environ[i]);
    }

    exit(0);
}
```

相关函数：

```
getenv()
setenv()
putenv()
```

3.5 c程序的存储空间布局

pmap

3.6 库

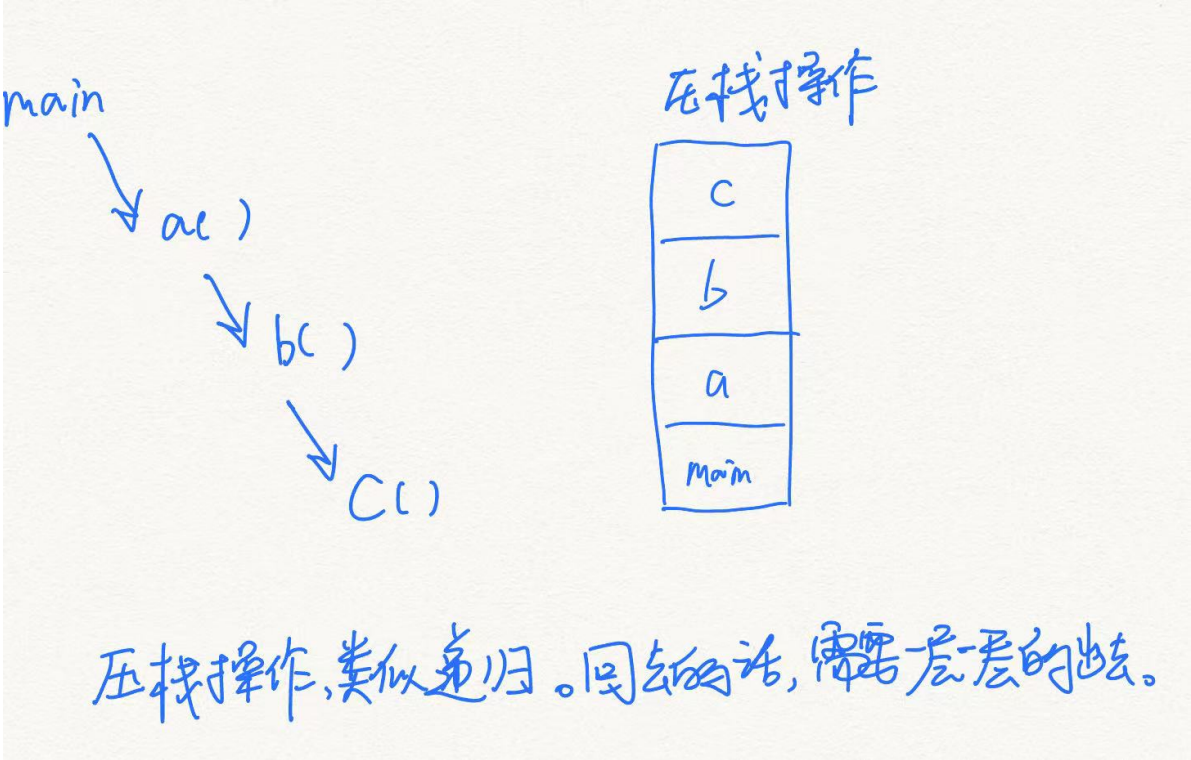
```
动态库
静态库
手工装载库
dlopen()
dlclose()
dlerror()
dlsym()
```

3.7 函数跳转

类似于goto，但是goto无法夸函数跳转。比如c++中，抛出异常，就需要夸函数跳转。

```
setjmp();  // 设置跳转点
longjmp(); // 从某个位置回到跳转点
// 这两个函数可以实现夸函数跳转
```

模拟实现压栈的功能:




```c
#include<stdio.h>
#include<stdlib.h>
#include <setjmp.h>
static void d(void)
{
    printf("%s():Begin.\n",__FUNCTION__);
    printf("%s():Jump now!\n",__FUNCTION__);
    printf("%s():End.\n",__FUNCTION__);
}
static void c(void)
{
    printf("%s():Begin.\n",__FUNCTION__);
    printf("%s():Call d().\n",__FUNCTION__);
```

```
    d();
    printf("%s():d() returned.\n",__FUNCTION__);
    printf("%s():End.\n",__FUNCTION__);
}
static void b(void)
{
    printf("%s():Begin.\n",__FUNCTION__);
    printf("%s():Call c().\n",__FUNCTION__);
    c();
    printf("%s():c() returned.\n",__FUNCTION__);
    printf("%s():End.\n",__FUNCTION__);

}

static void a(void)
{
    printf("%s():Begin.\n",__FUNCTION__);
    printf("%s():Call b().\n",__FUNCTION__);
    b();
    printf("%s():b() returned.\n",__FUNCTION__);
    printf("%s():End.\n",__FUNCTION__);
}

int main()
{
    printf("%s():Begin.\n",__FUNCTION__);
    printf("%s():Call a().\n",__FUNCTION__);
    a();
    printf("%s():a() returned.\n",__FUNCTION__);
    printf("%s():End.\n",__FUNCTION__);

    exit(0);
}
```

```
marz@ubuntu:~/cpp/io/filesystem$ make setjmp
cc -D_FILE_OFFSET_BITS=64   -lcrypt   setjmp.c    -o setjmp
marz@ubuntu:~/cpp/io/filesystem$ ./setjmp
main():Begin.
main():Call a().
a():Begin.
a():Call b().
b():Begin.
b():Call c().
c():Begin.
c():Call d().
d():Begin.
d():End.
c():d() returned.
c():End.
b():c() returned.
b():End.
a():b() returned.
a():End.
main():a() returned.
main():End.
```

使用setjmp进行跳转,(在函数a中设置跳转点，在函数d中跳转)

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

static jmp_buf save;

static void d(void)
{
    printf("%s():Begin.\n",__FUNCTION__);
    printf("%s():Jump now!\n",__FUNCTION__);
    longjmp(save,6);
    printf("%s():End.\n",__FUNCTION__);
}


static void c(void)
{
    printf("%s():Begin.\n",__FUNCTION__);
    printf("%s():Call d().\n",__FUNCTION__);
    d();
    printf("%s():d() returned.\n",__FUNCTION__);
    printf("%s():End.\n",__FUNCTION__);

}

static void b(void)
{
    printf("%s():Begin.\n",__FUNCTION__);
    printf("%s():Call c().\n",__FUNCTION__);
    c();
```

```
    printf("%s():c() returned.\n",__FUNCTION__);
    printf("%s():End.\n",__FUNCTION__);

}

static void a(void)
{
    int ret;

    printf("%s():Begin.\n",__FUNCTION__);
    ret = setjmp(save);
    if(ret == 0)
    {
        printf("%s():Call b().\n",__FUNCTION__);
        b();
        printf("%s():b() returned.\n",__FUNCTION__);
    }
    else
    {
        printf("%s():Jumped back here with code %d\n.",__FUNCTION__,ret);
    }

    printf("%s():End.\n",__FUNCTION__);

}

int main()
{
    printf("%s():Begin.\n",__FUNCTION__);
    printf("%s():Call a().\n",__FUNCTION__);
    a();
    printf("%s():a() returned.\n",__FUNCTION__);
    printf("%s():End.\n",__FUNCTION__);

    exit(0);
}
```

```
marz@ubuntu:~/cpp/io/filesystem$ make setjmp
cc -D_FILE_OFFSET_BITS=64  -lcrypt  setjmp.c   -o setjmp
marz@ubuntu:~/cpp/io/filesystem$ ./setjmp
main():Begin.
main():Call a().
a():Begin.
a():Call b().
b():Begin.
b():Call c().
c():Begin.
c():Call d().
d():Begin.
d():Jump now!
a():Jumped back here with code 6
.a():End.
main():a() returned.
main():End.
```

3.8 资源的获取及控制

```
getrlimit()
setrlimit()
```

1. main函数
2. 进程的终止
3. 命令行参数的分析
4. 环境变量
5. c程序的存储空间布局
6. 库
7. 函数跳转
8. 资源的获取及控制