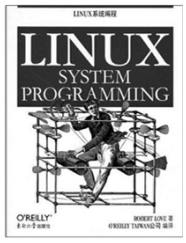
联航精英训练营
UNIGRESS ELITE TRAINING CAMP

Linux高级编程（一）

张勇涛

参考书目

Linux程序设计Neil Matthew / Richard Stones	人民邮电出版社
UNIX环境高级编程W.Richard Stevens / Stephen A.Rago	人民邮电出版社
LINUX系统编程Robert Love	东南大学出版社
Linux内核设计与实现Robert Love	机械工业出版社
Linux/UNIX系统编程手册Michael Kerrisk	人民邮电出版社



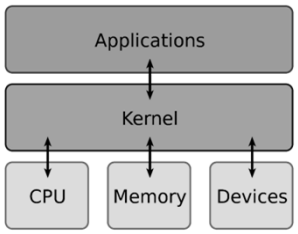
联航精英训练营

什么是操作系统?

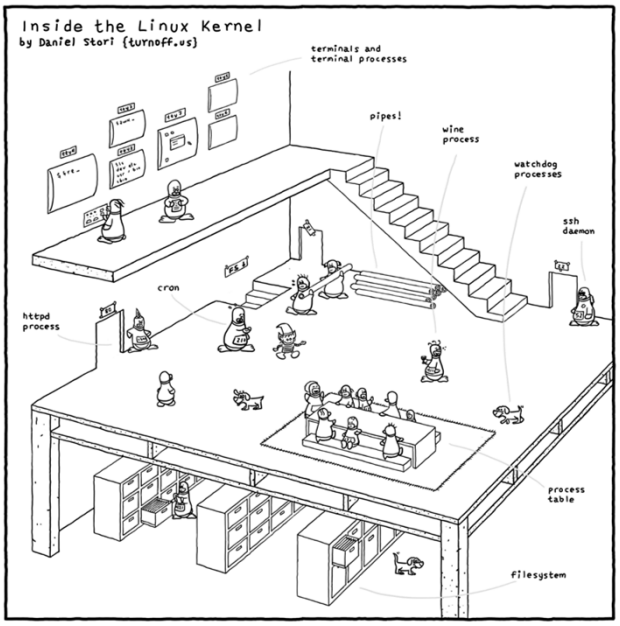
■ 操作系统（Operating System）：

An operating system (OS) is system software that manages computer hardware, software resources, and provides common services for computer programs.

操作系统扮演的两个角色:
1. 魔术师角色
2. 管理者角色

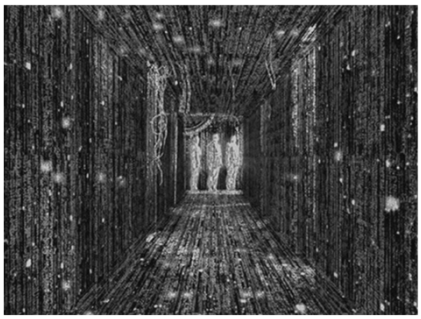


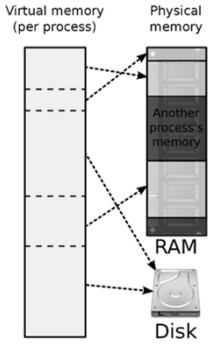
Inside the Linux kernel



用户程序和操作系统的关系

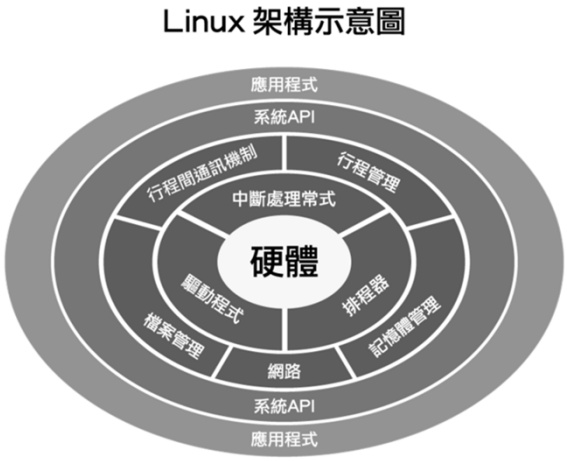
- 操作系统为用户程序提供了一个**虚拟**机器界面,而应用程序运行在这个界面之上。
- 程序和程序之间的关系：无非是调用和被调用的关系。





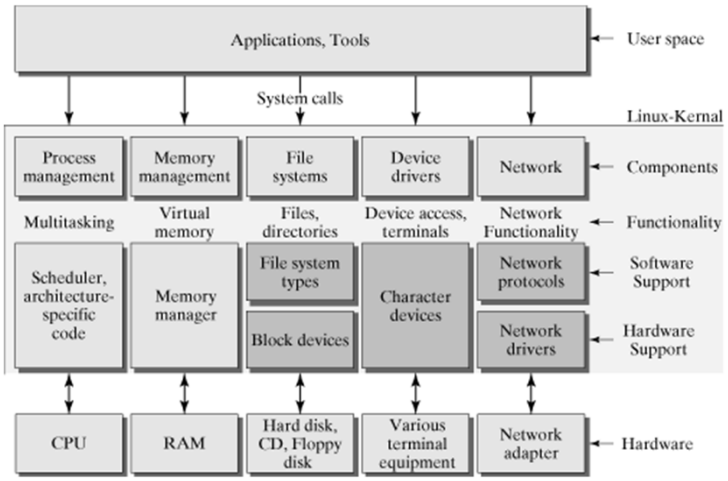
文件I/O操作

LINUX操作系统的体系结构



联航精英训练营

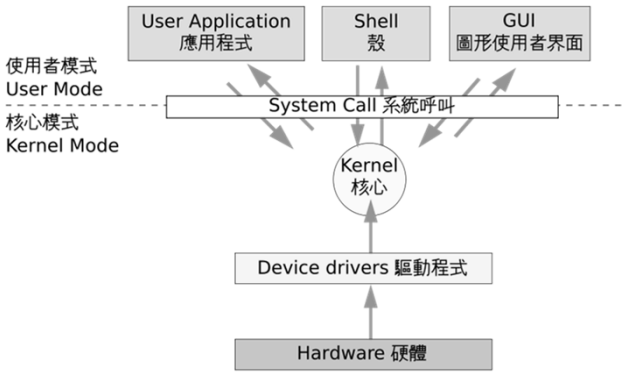
LINUX操作系统的体系结构



联航精英训练营

系统调用（system call）

- 系统调用就是用在程序中调用操作系统所提供的一些子功能
- 指操作系统提供给用户程序的一组“特殊”接口，用户程序可以通过这组“特殊”接口来获得操作系统内核提供的特殊服务。



联航精英训练营

系统调用

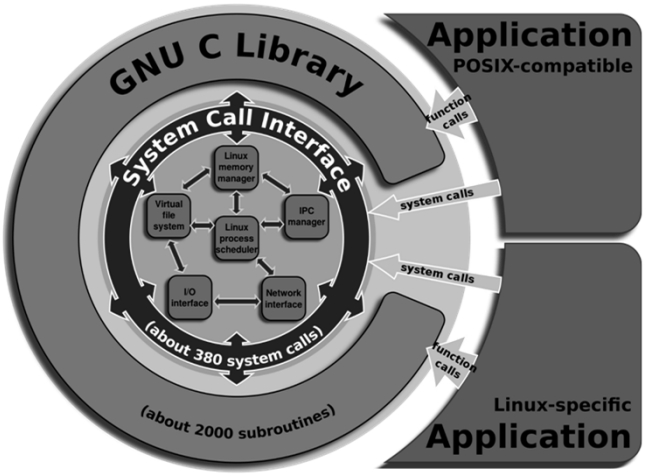
- 所谓系统调用是指操作系统提供给用户程序调用的**一组“特殊”接口**，用户程序可以通过这组“特殊”接口来获得操作系统内核提供的服务。例如用户可以通过进程控制相关的系统调用来创建进程、实现进程调度、进程管理等。

- 为什么用户程序不能直接访问系统内核提供的服务呢？这是由于在Linux中，为了更好地保护内核空间，将程序的运行空间分为内核空间和用户空间（也就是常称的**内核态**和**用户态**），它们分别运行在不同的级别上，在逻辑上是相互隔离的。因此，**用户进程在通常情况下不允许访问内核数据，也无法使用内核函数，它们只能在用户空间操作用户数据，调用用户空间的函数。**

- 但是，在有些情况下，用户空间的进程需要获得一定的系统服务（调用内核空间程序），这时操作系统就必须利用系统提供给用户的“特殊接口”——系统调用规定用户进程进入内核空间的具体位置。进行系统调用时，程序运行空间需要从用户空间进入内核空间，处理完后再返回到用户空间。

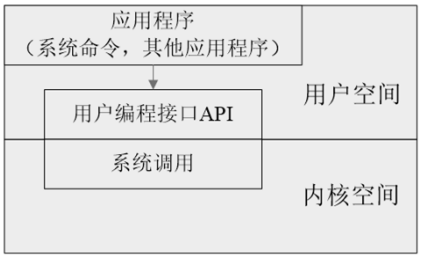
联航精英训练营

系统调用（system call）



用户编程接口（API）

- 前面讲到的**系统调用**并**不是直接**与程序员进行交互的，它仅仅是一个通过**软中断机制**向内核提交请求，以获取内核服务的接口。在实际使用中程序员调用的通常是用户编程接口——API
- 系统命令相对API更高了一层，它实际上一个可执行程序，它的内部引用了用户编程接口（API）来实现相应的功能。



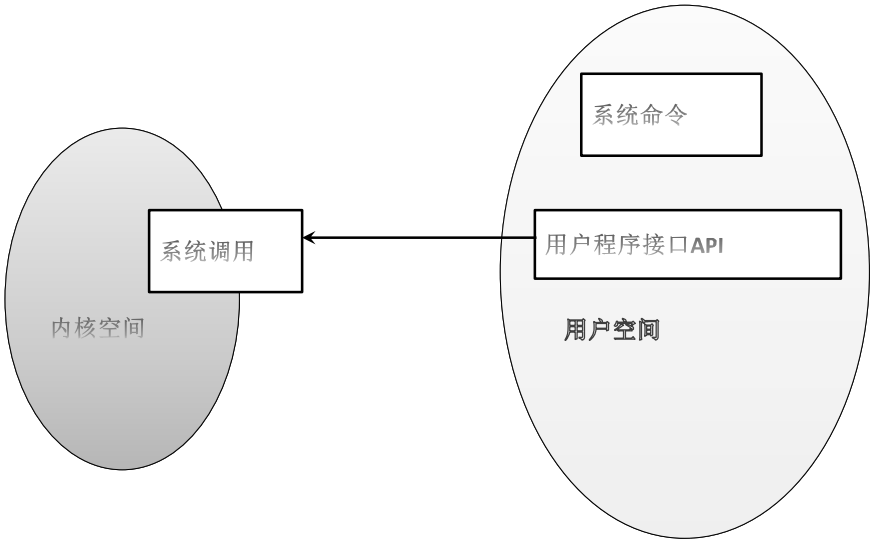
用户编程接口(API)

- 系统调用并不直接与程序员进行交互,它仅仅通过软中断的形式向内核提交请求, 以获得内核服务的接口.
- **用户编程接口（API）**其实是一个函数定义，说明了如何获得一个给定的服务。
- 在linux中用户编程接口（API）遵循了在UNIX中最流行的应用编程界面标准—POSIX标准。这些系统调用编程接口主要通过C库（libc）实现的。

系统命令

- 系统命令实际上是可执行程序.

系统调用、API、系统命令的关系



errno

无论什么时候系统调用失败了，都将errno设置成一组预定义的错误值中的一个。

strerror（）：

功能：以字符串方式打印错误信息。

用法：

#include <string.h>

char *strerror(int *errnum*);

返回：指向消息字符串的指针。

perror（）

功能：在标准错误上产生一条基于其参数串和errno的当前值出错消息。

用法：

#include <stdio.h>

void perror(const char * *msg*);

输出：首先输出由msg指向的字符串，然后是一个冒号，一个空格，然后是对应于errno值的出错信息，然后是一个新行符。

perror和strerror例子

Perror例子：

```
int main()
{
    FILE *fp;
    if((fp=fopen("1.txt","r")) == NULL)
    {
        perror("fopen");
        exit(0);
    }
    perror("fopen");
}
```

```
fopen: No such file or directory
fopen: Success
```

Strerror例子：

```
main()
{
   int i;
   for(i=0;i<10;i++)
   printf("%d : %s\n",i,strerror(i));
}
```

执行：

0 : Success
1 : Operation not permitted
2 : No such file or directory
3 : No such process
4 : Interrupted system call
5 : Input/output error
6 : Device not configured
7 : Argument list too long
8 : Exec format error
9 : Bad file descriptor

备注：好像共有132个

联航精英训练营

UNIX哲学之一：一切皆为文件

- Linux文件种类：

1. 普通文件
2. 目录文件
3. 链接文件
4. 设备文件

联航精英训练营

文件描述符

- 内核如何区分和引用特定的文件呢?

  这里用到了一个重要的概念——**文件描述符。**
- 对于Linux而言，所有对设备和文件的操作都是使用文件描述符来进行的。**文件描述符是一个非负的整数，它是一个索引值，并指向在内核中每个进程打开文件的记录表。**
- 当打开一个现存文件或创建一个新文件时，内核就向进程返回一个文件描述符；当需要读写文件时，也需要把文件描述符作为参数传递给相应的函数。
- 通常，一个进程启动时，都会打开3个文件：**标准输入、标准输出和标准出错处理。**这3个文件分别对应文件描述符为**0**、**1**和**2**（也就是宏替换**STDIN_FILENO**、**STDOUT_FILENO**和**STDERR_FILENO**）。

包含在<unistd.h>
文件描述符的范围是0 ~ OPEN_MAX
Linux 下OPEN_MAX为1048576

底层文件I/O操作

open()函数是用于**打开或创建**文件，在打开或创建文件时可以指定文件的属性及用户的权限等各种参数。

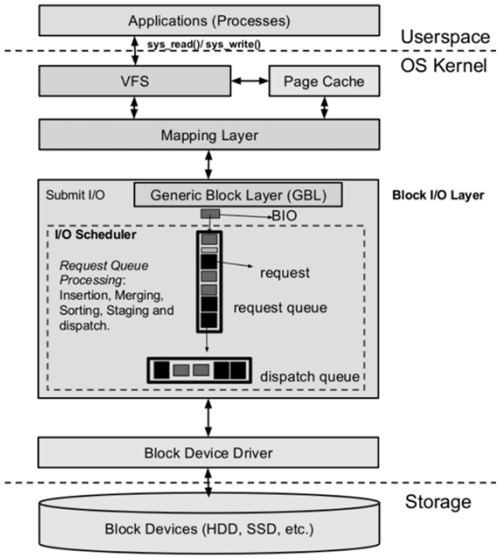
close()函数是用于**关闭**一个被打开的文件。当一个进程终止时，所有被它打开的文件都由内核自动关闭，很多程序都使用这一功能而不显示地关闭一个文件。

read()函数是用于**将从指定的文件描述符中读出的数据放到缓存区中，并返回实际读入的字节数。**若返回0，则表示没有数据可读，即已达到文件尾。读操作从文件的当前指针位置开始。当从终端设备文件中读出数据时，通常一次最多读一行。

write()函数是**用于向打开的文件写数据，写操作从文件的当前指针位置开始。**对磁盘文件进行写操作，若磁盘已满或超出该文件的长度，则write()函数返回失败。

lseek()函数是**用于在指定的文件描述符中将文件指针定位到相应的位置。**

底层文件I/O操作



联航精英训练营

open

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

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

返回值：成功返回新分配的文件描述符，出错返回-1并设置errno

联航精英训练营

参数含义

- pathname是要打开或创建的文件的名字。
- oflag参数可用来说明此函数的多个选择项。
- mode 对于open函数而言，仅当创建新文件时才使用第三个参数。

- 以下三个常数中必须指定一个，且仅允许指定一个(这些常数定义在<fcntl.h>头文件中)：

O_RDONLY 只读打开。
O_WRONLY 只写打开。
O_RDWR 读、写打开。

可选项

- 以下可选项可以同时指定0个或多个, 和必选项按位或起来作为flags参数。

O_APPEND 每次写时都加到文件的尾端。
O_CREAT 若此文件不存在则创建它。使用此选择项时，需同时说明第三个参数mode，用其说明该新文件的存取许可权位。
O_EXCL 如果同时指定了O_CREAT，而文件已经存在，则出错。这可测试一个文件是否存在，如果不存在则创建此文件成为一个原子操作。
O_TRUNC 如果此文件存在，而且为只读或只写成功打开，则将其长度截短为0。
O_NONBLOCK 如果pathname指的是一个块特殊文件或一个字符特殊文件，则此选择项为此文件的本次打开操作和后续的I／O操作设置非阻塞方式。

mode

- 第三个参数mode指定文件权限，可以用八进制数表示，比如0644表示-rw-r--r--，也可以用S_IRUSR、S_IWUSR等宏定义按位或起来表示。

mode取值	含义
S_IRUSR	文件所有者的读权限
S_IWUSR	文件安所有者的写权限
S_IXUSR	文件所有者的执行权限
S_IRGRP	文件所有者同组用户的读权限
S_IWGRP	文件所有者同组用户的写权限
S_IXGRP	文件所有者同组用户的执行权限
S_IROTH	其他用户的读权限
S_IWOTH	其他用户的写权限
S_IXOTH	其他用户的执行权限

creat

- 功能：创建一个新文件
- 用法：
  - #include <sys/types.h>
  - #include <sys/stat.h>
  - #include <fcntl.h>
  - int creat(const char *_name_, mode_t _mode_) ;
- 返回：
  - 成功为只写打开的文件描述符；出错为-1。
  - 出错时errno被设置。
- 注意：此函数等效于：
  - open (_name_, O_WRONLY | O_CREAT | O_TRUNC, _mode_) ;

  creat函数现在已经很少使用

close

可用close函数关闭一个打开文件：

```
#include <unistd.h>

int close (int filedes);
```

返回：若成功为0，若出错为- 1

当一个进程终止时，它所有的打开文件都由内核自动关闭。很多程序都使用这一功能而不显式地用close关闭打开的文件。

练习：

- 打开文件./hello.txt 用于写操作,以追加方式打开
- 打开文件./hello.txt 用于写操作,如果该文件不存在则创建
- 打开文件./hello.txt 用于写操作,如果该文件已存在则截断它,如果该文件不存在则创建它
- 打开文件./hello.txt 用于写操作,如果该文件已存在则报错退出,如果不存在则创建它

例子

- 打开文件test.txt
- 例子open.c

例子

- 打开文件的出错情况
- 例子：file_error.c

read函数

- 用read函数从打开文件中读数据

#include <unistd.h>

ssize_t read(int *feledes*, void **buff*, size_t *nbytes*) ;

返回：读到的字节数，若已到文件尾为0，若出错为- 1。

如read成功，则返回读到的字节数。如已到达文件的尾端，则返回0。

read 从当前文件偏移量处读入指定大小的文件内容。

实际读到的字节数少于要求读字节数

1. 读普通文件时，在读到要求字节数之前已到达了文件尾端。例如，若在到达文件尾端之前还有30个字节，而要求读100个字节，则read返回30，下一次再调用read时，它将返回0 (文件尾端)。

2. 当从终端设备读时，通常一次最多读一行。

3. 当从网络读时，网络中的缓冲机构可能造成返回值小于所要求读的字节数。

write函数

- 用write函数向打开文件写数据。

#include <unistd.h>

ssize_t write(int *filedes*, const void * *buff*, size_t *nbytes*) ;
返回：若成功为已写的字节数，若出错为- 1。

其返回值通常与参数*nbytes*的值相同，否则表示出错。
write出错的一个常见原因是：磁盘已写满，或者超过了对一个给定进程的文件长度限制。

文件的读写操作

- 见例子my_cp.c

堵塞与非堵塞

- 读常规文件是不会堵塞的,不管读多少字节,read一定会在有限的时间内返回.
- 从终端和网络读则不一定,如果终端输入不足一行,read一个终端设备就会堵塞.
- 如果在open一个设备时指定了O_NONBLOCK, read和write就不会堵塞.
- 以read为例,如果 设备没有数据到达就返回-1,同时置errno位

非堵塞工作模式

```
while(1)
{
   非堵塞 read(设备1);
    if(设备1有数据到达)
        处理数据;
   非堵塞 read(设备2);
    if(设备2有数据到达)
        处理数据;
     …
    {sleep(n);}
}
```

问题如何解决？

- select可以堵塞地同时监视多个设备,还可以设定堵塞等待的超时时间从而圆满的解决这个问题.

堵塞读终端例子

非堵塞读终端的例子

lseek函数

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

off_t lseek(int filesdes, off_t offset, int whence) ;
```

返回：若成功为新的文件位移，若出错为- 1。

对参数*offset* 的解释与参数*whence*的值有关。

若*whence*是SEEK_SET，则将该文件的位移量设置为距文件开始处 *offset* 个字节。

若*whence*是SEEK_CUR ， 则将该文件的位移量设置为其当前值加*offset, offset*可为正或负。

若*whence*是SEEK_END ， 则将该文件的位移量设置为文件长度加*offset, offset*可为正或负。

如何测试文件位移量

若lseek成功执行，则返回新的文件位移量，为此可以用下列方式确定一个打开文件的当前位移量：

```
off_t curr_pos;
curr_pos = lseek(fd, 0, SEEK_CUR);
```

一般从当前文件偏移量处写入，但如果打开时使用了O_APPEND，那么无论当前文件偏移量在哪里,都会移动到文件末尾写入

思考题：

如何迅速创建一个大文件?

例子

- 见lseek.c

fcntl函数

fcntl函数可以改变已经打开文件的性质。

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
int fcntl(int fd, int cmd, struct flock *lock);
```

返回：若成功则依赖于*cmd*(见下)，若出错为- 1。

fcntl函数改变一个已打开的文件的属性，可以重新设置读、写、追加、非阻塞等标志（这些标志称为File Status Flag），而不必重新open文件。

fcntl函数的五种功能

复制一个现存的描述符, 新文件描述符作为函数值返(cmd = F_DUPFD）。

获得/设置文件描述符标记,对应于*filedes* 的文件描述符标志作为函数值返回. （cmd = F_GETFD或F_SETFD）。

获得/设置文件状态标志，对应于*filedes* 的文件状态标志作为函数值返回。（cmd = F_GETFL或F_SETFL）。

获得/设置异步I / O有权（cmd = F_GETOWN或F_SETOWN)。

获得/设置记录锁（cmd = F_SETLK , F_SETLKW）。

fcntl例子

- 用fcntl改变File Status Flag的例子

联航精英训练营

fcntl的作用

- 通过fcntl设置的都是当前进程如何设置访问设备和文件的访问控制属性,如读、写、追加、非堵塞、加锁等
- 但是不设置文件或设备本身的属性，例如文件的读写权限、串口波特率等。

联航精英训练营

ioctl

用于向设备发控制或配置命令。

```
#include <unistd.h> /* SVR4 */
#include <sys/ioctl.h> /* 4.3+BSD * /

int ioctl(int filedes, int request, . . . ) ;
```

返回：若出错则为- 1，若成功则为其他值。

联航精英训练营

ioctl的特点

ioctl 函数是I/O操作的杂物箱。不能用本章中其他函数表示的I / O操作通常都能用ioctl表示。

终端I/O是ioctl 的最大使用方面（POSIX.1已经用新的函数代替ioctl进行终端I / O操作）。

ioctl更多的是用于设备控制，比如磁盘的格式化，MODEM设备，磁带的快进、快倒等。

ioctl甚至可以读写一些数据，但是这些数据是不能用read、write读写的，称为out-of-band数据。

read\write读写的是in-band数据,是I/O操作的主体,而ioctl命令传送的是控制信息,其中的数据是辅助的数据。

联航精英训练营

ioctl例子

- 见ioctl例子

mmap

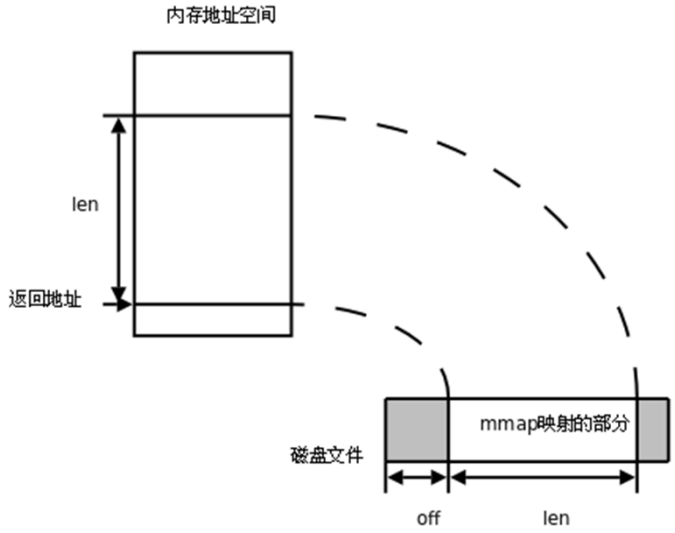
#include <sys/mman.h>

void *mmap(void *addr, size_t len, int prot, int flag, int filedes, off_t off);

int munmap(void *addr, size_t len);

mmap可以把磁盘文件的一部分直接映射到内存，这样文件中的位置直接就有对应的内存地址，对文件的读写可以直接用指针来做而不需要read/write函数

mmap函数



mmap

- 如果addr参数为NULL，内核会自己在进程地址空间中选择合适的地址建立映射。
- 如果addr不是NULL，则给内核一个提示，应该从什么地址开始映射，内核会选择addr之上的某个合适的地址开始映射。
- 建立映射后，真正的映射首地址通过返回值可以得到。len参数是需要映射的那一部分文件的长度。off参数是从文件的什么位置开始映射，必须是页大小的整数倍（在32位体系统结构上通常是4K）。filedes是代表该文件的描述符。

prot参数有四种取值：

PROT_EXEC表示映射的这一段可执行，例如映射共享库
PROT_READ表示映射的这一段可读
PROT_WRITE表示映射的这一段可写
PROT_NONE表示映射的这一段不可访问

flag参数有很多种取值，这里只讲两种，其它取值可查看mmap(2)
MAP_SHARED多个进程对同一个文件的映射是共享的，一个进程对映射的内存做了修改，另一个进程也会看到这种变化。
MAP_PRIVATE多个进程对同一个文件的映射不是共享的，一个进程对映射的内存做了修改，另一个进程并不会看到这种变化，也不会真的写到文件中去。

如果mmap成功则返回映射首地址，如果出错则返回常数MAP_FAILED。当进程终止时，该进程的映射内存会自动解除，也可以调用munmap解除映射。munmap成功返回0，出错返回-1。

mmap例子

文件截断

#include <unistd.h>
#include <sys/types.h>
int truncate(const char *path, off_t length);
int ftruncate(int fd, off_t length);
truncate中path表示文件的路径
length表示将文件截断的字节数

截断的例子
见truncate.c

sync

#include <unistd.h>

int fsync(int filedes) 把指定文件的数据和属性写入到磁盘。
int fdatasync(int filedes) 把指定文件的数据部分写到磁盘。

void sync(void) 把修改部分排入磁盘写队列，但并不意味着已经写入磁盘。

文件同步

- 见fsyn.c

文件和目录操作的系统函数

- chmod
- chown
- mkdir/rmdir
- chdir/getcwd

文件读写作业

text1.txt

begin
10 11 12
20 21 22
30 31 32
end

text2.txt

begin
15 16 17
25 26 27
35 36 37
end

text3.txt

begin
25 27 29
45 47 49
65 67 69
end

手动创建两个文本文件text1.txt,text2.txt,要求编程创建text3.txt,实现text1.txt和text2.txt文件中除去首行和末尾对应的数据相加,三个文本的内容如上

标准I/O开发

C语言文件概述

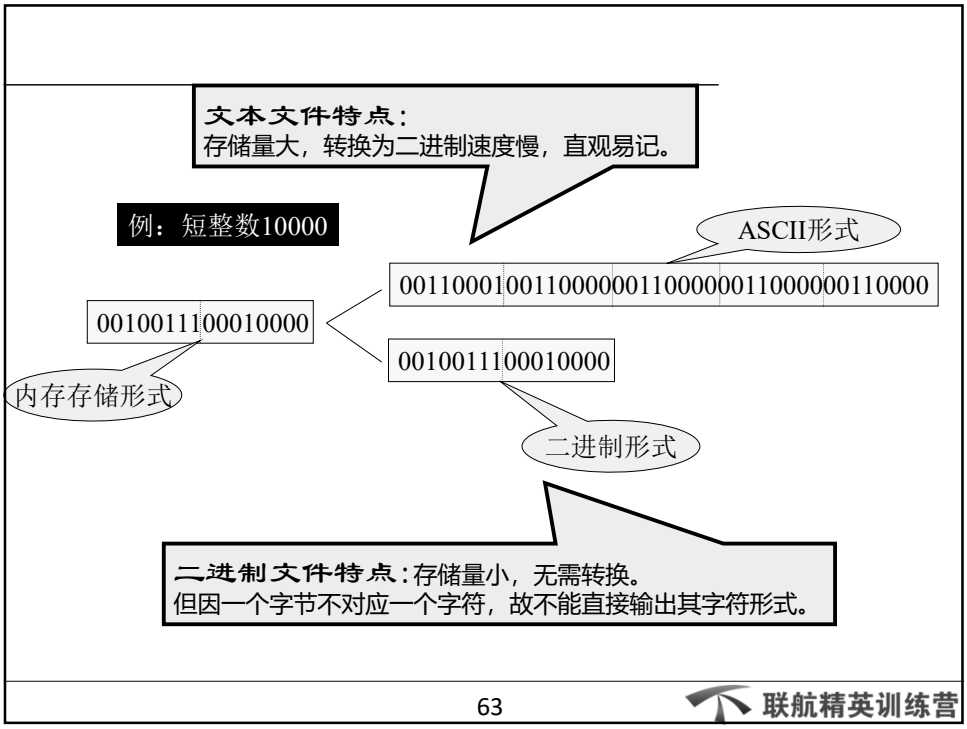
文件：存储在外部介质上数据的集合,是操作系统数据管理的单位。

文件分类：

- 按文件的逻辑结构：
  - 记录文件：由具有一定结构的记录组成（定长和不定长）。
  - 流式文件：由一个个字符（字节）数据顺序组成。
- 按存储介质：
  - 普通文件：存储介质文件（磁盘、磁带等）。
  - 设备文件：非存储介质（键盘、显示器、打印机等）。
- 按数据的组织形式：
  - ASCII文件(文本文件)：每个字节存放ASCII码，表示一个字符。
  - 二进制文件：数据按其在内存中的存储形式原样存放。

文件类型

- 文件可分为文本文件（Text File）和二进制文件（Binary File）两种。
- 源文件是文本文件，而目标文件、可执行文件和库文件是二进制文件。
- od -tx1 -tc -Ax filename



od命令

- od (octal dump)和 xd(hexdump)命令可以以十进制、八进制、十六进制和ASCII码来显示文件或者流，它们对于访问或可视地检查文件中不能直接显示在终端上的字符很有用。

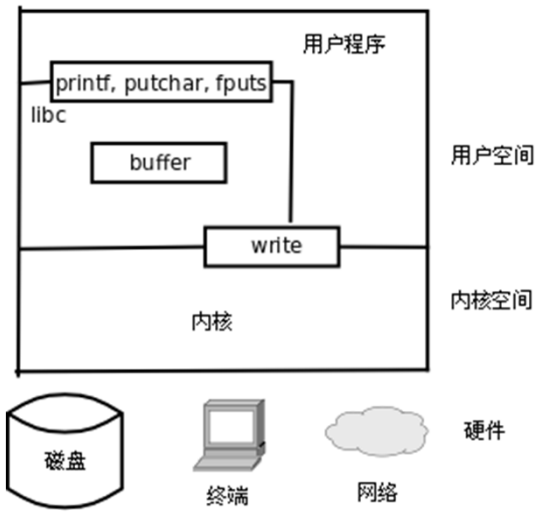
  语法：od ［选项］ 文件…

  命令中各选项的含义：

  - A 指定地址基数，包括：
  d 十进制
  o 八进制（系统默认值）
  x 十六进制
  n 不打印位移值
  - t 指定数据的显示格式，主要的参数有：
  c ASCII字符或反斜杠序列
  d 有符号十进制数
  f 浮点数
  o 八进制（系统默认值为02）
  u 无符号十进制数
  x 十六进制数

  除了选项c以外的其他选项后面都可以跟一个十进制数n，指定每个显示值所包含的字节数。

库函数与系统调用的层次关系



打开文件fopen

#include <stdio.h>

FILE *fopen(const char *path, const char *mode);

返回值：成功返回文件指针，出错返回NULL并设置errno
path是文件的路径名，mode表示打开方式。

FILE

- FILE是C标准库中定义的结构体类型，其中包含该文件在内核中标识、I/O缓冲区和当前读写位置等信息
- 像FILE *这样的指针称为不透明指针（Opaque Pointer）或者叫句柄（Handle），FILE *指针就像一个把手（Handle），抓住这个把手就可以打开门或抽屉，但用户只能抓这个把手，而不能直接抓门或抽屉。

FILE

文件类型结构体FILE

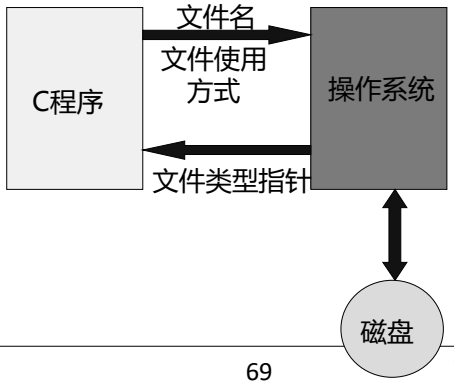
- 缓冲文件系统为每个正使用的文件在内存开辟文件信息区。
- 文件信息用系统定义的名为FILE的结构体描述。
- FILE定义在stdio.h中。

```
typedef struct
 {
 short    level;             /* 缓冲区满/空程度  */
 unsigned  flags;             /* 文件状态标志 */
 char  fd;                    /* 文件描述符 */
 unsigned char  hold;         /* 若无缓冲区不读取字符 */
 short  bsize;                /* 缓冲区大小 */
 unsigned char  *buffer;      /* 数据传送缓冲区位置 */
 unsigned char  *curp;        /* 当前读写位置 */
 unsigned  istemp;            /* 临时文件指示 */
 short  token;                /* 用作无效检测 */
} FILE ;                      /* 结构体类型名 FILE  */
```

FILE

- FILE *文件结构指针名；
- 用法：
  - 文件打开时，系统自动建立文件结构体，并把指向它的指针返回来，程序通过这个指针获得文件信息,访问文件。
  - 文件关闭后，它的文件结构体被释放。

```
FILE  *fp;
```



联航精英训练营

mode

- mode参数是一个字符串，由rwatb+六个字符组合而成，r表示读，w表示写，a表示追加（Append），在文件末尾追加数据使文件的尺寸增大。
- t表示文本文件，b表示二进制文件，有些操作系统的文本文件和二进制文件格式不同，而在UNIX系统中，无论文本文件还是二进制文件都是由一串字节组成，t和b没有区分，用哪个都一样。

联航精英训练营

mode

- “r” :只读，文件必须已存在
- “w “:只写，如果文件不存在则创建，如果文件已存在则把文件长度截断（Truncate）为0字节再重新写，也就是替换掉原来的文件内容
- “a”:只能在文件末尾追加数据，如果文件不存在则创建
- “r+ “:允许读和写，文件必须已存在
- “w+ “:允许读和写，如果文件不存在则创建，如果文件已存在则把文件长度截断为0字节再重新写
- "a+”:允许读和追加数据，如果文件不存在则创建

fopen的使用

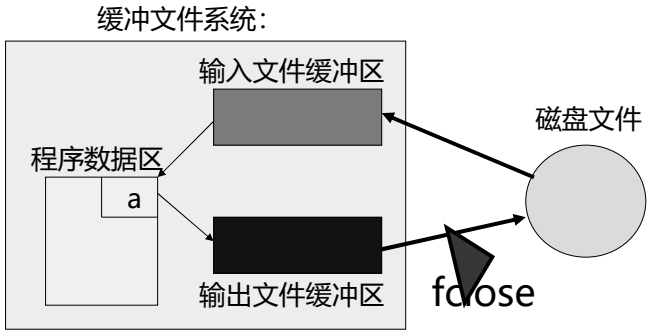
- 在打开一个文件时如果出错，fopen将返回NULL并设置errno。在程序中应该做出错处理，通常这样写：

```
FILE *fp;
if ( (fp = fopen("/tmp/file1", "r")) == NULL)
 {
    perror("error open file /tmp/file1");
   exit(1);
}
```

fclose

#include <stdio.h>

int fclose(FILE *fp);
返回值：成功返回0，出错返回EOF并设置errno



stdin/stdout/stderr

- UNIX的传统是Everything is a file，键盘、显示器、串口、磁盘等设备在/dev目录下都有一个特殊的设备文件与之对应，这些设备文件也可以像普通文件一样打开、读、写和关闭，使用的函数接口是相同的。

那为什么printf和scanf不用打开就能对终端设备进行操作呢?

练习：

打开一个没有访问权限的文件。

```
fp = fopen("/etc/shadow", "r");
if (fp == NULL)
 {
    perror("Open /etc/shadow");
    exit(1);
}
```

fopen也可以打开一个目录，传给fopen的第一个参数目录名末尾可以加/也可以不加/，但只允许以只读方式打开。试试如果以可写的方式打开一个存在的目录会怎么样呢?

```
fp = fopen("/home/zyt/", "r+");
if (fp == NULL)
{
    perror("Open /home/zyt");
    exit(1);
}
```


以字节为单位的I/O函数

```
#include <stdio.h>

int fgetc(FILE *stream);
int getchar(void);
```

返回值：成功返回读到的字节，出错或者读到文件末尾时返回EOF

从终端设备读还有一个特点，用户输入一般字符并不会使getchar函数返回，仍然阻塞着，只有当用户输入回车或者到达文件末尾时getchar才返回

说明：

- 要用fgetc函数读一个文件，该文件的打开方式必须是可读的。
- 系统对于每个打开的文件都记录着当前读写位置在文件中的地址（或者说距离文件开头的字节数），也叫偏移量（Offset）。
- fgetc成功时返回读到一个字节，本来应该是unsigned char型的，但由于函数原型中返回值是int型，所以这个字节要转换成int型再返回。

fputc与putchar

```
#include <stdio.h>

int fputc(int c, FILE *stream);
int putchar(int c);
```

返回值：成功返回写入的字节，出错返回EOF

说明：

- 要用fputc函数写一个文件，该文件的打开方式必须是可写的（包括追加）。
- 每调用一次fputc，读写位置向后移动一个字节，因此可以连续多次调用fputc函数依次写入多个字节。但如果文件是以追加方式打开的，每次调用fputc时总是将读写位置移到文件末尾然后把要写入的字节追加到后面。

例子

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

int main(void)
{
    FILE *fp;
    int ch;

    if ( (fp = fopen("file2", "w+")) == NULL) {
        perror("Open file file2\n");
        exit(1);
    }
    while ( (ch = getchar()) != EOF)
        fputc(ch, fp);
    rewind(fp);
    while ( (ch = fgetc(fp)) != EOF)
        putchar(ch);
    fclose(fp);
    return 0;
}
```

练习：

- 编写一个简单的文件复制程序。

$ ./mycp dir1/fileA dir2/fileB运行这个程序可以把dir1/fileA文件拷贝到dir2/fileB文件。

- 注意各种出错处理。

操作读写位置的函数

```
#include <stdio.h>

int fseek(FILE *stream, long offset, int whence);
返回值：成功返回0，出错返回-1并设置errno

long ftell(FILE *stream);
返回值：成功返回当前读写位置，出错返回-1并设置errno

void rewind(FILE *stream);
```

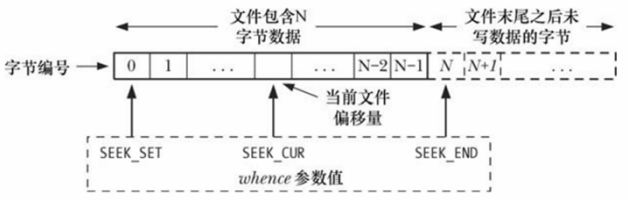
whence

fseek的whence和offset参数共同决定了读写位置移动到何处，whence参数的含义如下：

SEEK_SET 从文件开头移动offset个字节

SEEK_CUR 从当前位置移动offset个字节

SEEK_END 从文件末尾移动offset个字节



联航精英训练营

以字符串为单位的I/O函数

```
#include <stdio.h>

char *fgets(char *s, int size, FILE *stream);
char *gets(char *s);
```

返回值：成功时s指向哪返回的指针就指向哪，出错或者读到文件末尾时返回NULL

fgets从指定的文件中读一行字符到调用者提供的缓冲区中

gets从标准输入读一行字符到调用者提供的缓冲区中。

联航精英训练营

warning 1：

- gets函数的存在只是为了兼容以前的程序，我们写的代码都不应该调用这个函数。gets函数的接口设计得很有问题，就像strcpy一样，用户提供一个缓冲区，却不能指定缓冲区的大小，很可能导致缓冲区溢出错误，这个函数比strcpy更加危险，strcpy的输入和输出都来自程序内部，只要程序员小心一点就可以避免出问题，而gets读取的输入直接来自程序外部，用户可能通过标准输入提供任意长的字符串，程序员无法避免gets函数导致的缓冲区溢出错误，所以唯一的办法就是不要用它。

warning 2：

- 对于fgets来说，'\n'是一个特别的字符，而'\0'并无任何特别之处，如果读到'\0'就当作普通字符读入。如果文件中存在'\0'字符（或者说0x00字节），调用fgets之后就无法判断缓冲区中的'\0'究竟是从文件读上来的字符还是由fgets自动添加的结束符，所以fgets只适合读文本文件而不适合读二进制文件，并且文本文件中的所有字符都应该是可见字符，不能有'\0'。

fputs与puts

```
#include <stdio.h>

int fputs(const char *s, FILE *stream);
int puts(const char *s);
```

返回值：成功返回一个非负整数，出错返回EOF

以记录为单位的I/O函数

```
#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t nmemb,
FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
FILE *stream);
```

返回值：读或写的记录数，成功时返回的记录数等于nmemb，出错或读到文件末尾时返回的记录数小于nmemb，也可能返回0

以记录为单位的I/O函数

- 参数size指出一条记录的长度，而nmemb指出要读或写多少条记录，这些记录在ptr所指的内存空间中连续存放，共占size * nmemb个字节，fread从文件stream中读出size * nmemb个字节保存到ptr中，而fwrite把ptr中的size * nmemb个字节写到文件stream中。
- nmemb是请求读或写的记录数，fread和fwrite返回的记录数有可能小于nmemb指定的记录数。

联航精英训练营

例子：

- writerec.c
- readrec.c

联航精英训练营

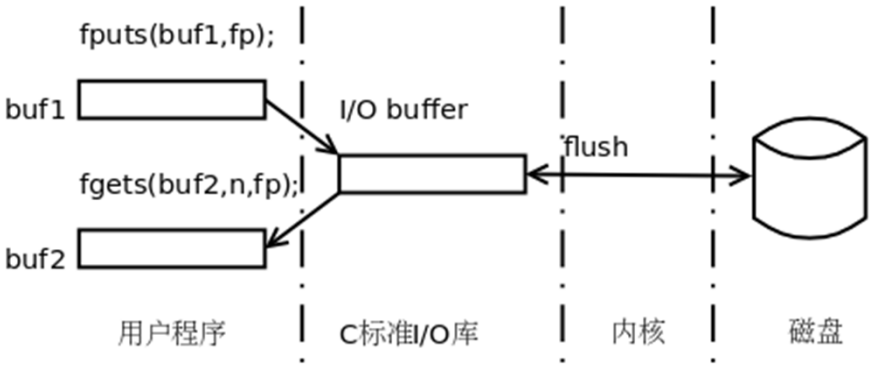
格式化I/O函数

```
#include <stdio.h>

int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);
```

返回值：成功返回格式化输出的字节数（不包括字符串的结尾'\0'），出错返回一个负值

C标准库的I/O缓冲区



标准I/O提供3种类型的缓冲

1.全缓冲

如果缓冲区写满了就写回内核。常规文件通常是全缓冲的。

2.行缓冲

如果用户程序写的数据中有换行符就把这一行写回内核，或者如果缓冲区写满了就写回内核。标准输入和标准输出对应终端设备时通常是行缓冲的。

3. 不带缓冲

用户程序每次调库函数做写操作都要通过系统调用写回内核。标准错误输出通常是无缓冲的，这样用户程序产生的错误信息可以尽快输出到设备。

例子：

```
#include <stdio.h>

int main()
{
   printf("hello world");
   while(1);
   return 0;
}
```

fflush

#include <stdio.h>

int fflush(FILE *stream);
返回值：成功返回0，出错返回EOF并设置errno

fflush函数用于确保数据写回了内核，以免进程异常终止时丢失数据。

联航精英训练营
UNIGRESS ELITE TRAINING CAMP

专业铸就品质 梦想成就未来
The Specialty Casts Quality,the Dream Gains Future.