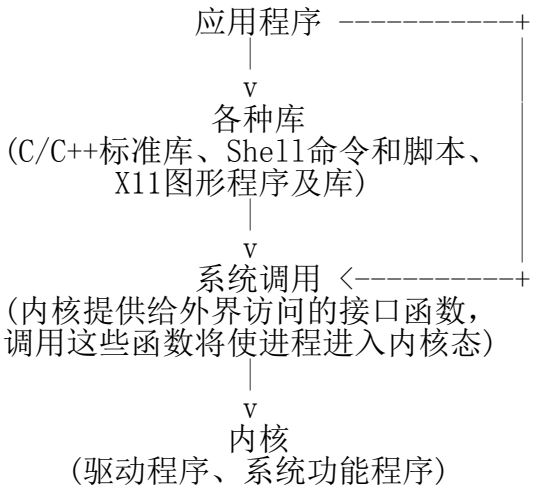
第三课　文件系统(上)

一、系统调用



1. Unix/Linux大部分系统功能是通过系统调用实现的。
   如：open/close。

2. Unix/Linux的系统调用已被封装成C函数的形式，
   但它们并不是标准C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。

4. 程序员自己编写的代码也可以调用系统调用，
   与操作系统内核交互，进入内核态。
   如：brk/sbrk/mmap/munmap。

5. 系统调用在内核中实现，其外部接口定义在C库中。
   该接口的实现借助软中断进入内核。

time命令：测试运行时间

real : 总执行时间
user : 用户空间执行时间
sys : 内核空间执行时间

strace命令：跟踪系统调用

二、一切皆文件

1. Linux环境中的文件具有特别重要的意义，
   因为它为操作系统服务和设备，
   提供了一个简单而统一的接口。
   在Linux中，(几乎)一切皆文件。

2. 程序完全可以象访问普通磁盘文件一样，
   访问串行口、网络、打印机或其它设备。

3. 大多数情况下只需要使用五个基本系统调用：
   open/close/read/write/ioctl，
   即可实现对各种设备的输入和输出。

4. Linux中的任何对象都可以被视为某种特定类型的文件，
   可以访问文件的方式访问之。

5. 广义的文件

1) 目录文件

```
# vim day01
```

2) 设备文件

A. 控制台：/dev/console
B. 声卡：/dev/audio
C. 标准输入输出：/dev/tty
D. 空设备：/dev/null

例如：

```
# cat /dev/tty
Hello, World !
Hello, World !

# echo Hello, World ! > /dev/tty
Hello, World !

# echo Hello, World ! > test.txt
# cat test.txt
Hello, World !
# cat /dev/null > test.txt
# cat test.txt

# find / -name perl 2> /dev/null
```

三、文件相关系统调用

```
open   - 打开/创建文件
creat  - 创建空文件
close  - 关闭文件
read   - 读取文件
write  - 写入文件
lseek  - 设置读写位置
fcntl  - 修改文件属性
unlink - 删除硬链接
rmdir  - 删除空目录
remove - 删除硬链接(unlink)或空目录(rmdir)
```

注意：

1. 如果被unlink/remove删除的是文件的最后一个硬链接，
   并且没有进程正打开该文件，
   那么该文件在磁盘上的存储区域将被立即标记为自由。
   反之，如果有进程正打开该文件，
   那么该文件在磁盘上的存储区域，
   将在所有进程关闭该文件之后被标记为自由。

```
  a -> +-----+
X b -> | ... |
X c -> +-----+
```

2. 如果被unlink/remove删除的是一个软链接文件，
   那么仅软链接文件本身被删除，其目标不受影响。

```
       +-----+
  a -> | ... |
       +-----+
       +-----+
X b -> |  a  |
       +-----+
       +-----+
X c -> |  a  |
       +-----+
```

四、文件描述符
---

1. 非负的整数。

2. 表示一个打开的文件。

3. 由系统调用(open)返回，
   被内核空间(后续系统调用)引用。

4. 内核缺省为每个进程打开三个文件描述符：

0 - 标准输入
1 - 标准输出
2 - 标准出错

在unistd.h中被定义为如下三个宏：

```
#define STDIN_FILENO  0
#define STDOUT_FILENO 1
#define STDERR_FILENO 2
```

范例：redir.c

```
# a.out 0<i.txt 1>o.txt 2>e.txt
```

5. 文件描述符的范围介于0到OPEN_MAX之间，
   传统Unix中OPEN_MAX宏被定义为63，
   现代Linux使用更大的上限。

五、open/creat/close

```
#include <fcntl.h>

int open (
    const char* pathname, // 路径
    int         flags,    // 模式
    mode_t      mode      // 权限(仅创建文件有效)
);  // 创建/读写文件时都可用此函数

int creat (
    const char* pathname, // 路径
    mode_t      mode      // 权限
);  // 常用于创建文件

int open (
    const char* pathname, // 路径
    int         flags     // 模式
);  // 常用于读写文件
```

成功返回文件描述符，失败返回-1。

flags为以下值的位或：

```
O_RDONLY   - 只读。\
                   |
O_WRONLY   - 只写。 > 只选一个
                   |
O_RDWR     - 读写。/

O_APPEND   - 追加。

O_CREAT    - 创建。不存在即创建(已存在即直接打开，
             并保留原内容，除非...)，
             有此位mode参数才有效。

O_EXCL     - 排斥。已存在即失败。\
                                  > 只选一个，
O_TRUNC    - 清空。已存在即清空   /  配合O_CREAT使用
             (有O_WRONLY/O_RDWR)。

O_NOCTTY   - 若pathname指向一个终端设备，
             则该终端不会成为调用进程的控制终端。

O_NONBLOCK - 非阻塞。若pathname指向FIFO/块/字符文件，
             则该文件的打开及后续操作均为非阻塞模式。

O_SYNC     - 写同步。write等待数据和属性，
             被物理地写入底层硬件后再返回。

O_DSYNC    - 数据写同步。write等待数据，
             被物理地写入底层硬件后再返回。

O_RSYNC    - 读同步。read等待对所访问区域的所有写操作，
             全部物理地写入底层硬件后，再读取并返回。

O_ASYNC    - 异步读写。当文件描述符可读/写时，
```

向调用进程发送SIGIO信号。

open/creat所返回的一定是当前未被使用的，
最小文件描述符。

一个进程可以同时打开的文件描述符个数，
受limits.h中定义的OPEN_MAX宏的限制，
POSIX要求不低于16，传统Unix是63，现代Linux是256。

```
#include <unistd.h>

int close (
    int fd // 文件描述符
);
```

成功返回0，失败返回-1。

范例：open.c

操作系统可通过权限掩码(当前为0022)，
屏蔽程序所创建文件的某些权限位。如：

0666 (rw-rw-rw-) & ~0022 = 0644 (rw-r--r--)

creat函数是通过调用open实现的。

```
int creat (const char* pathname, mode_t mode) {
    return open (pathname,
        O_WRONLY | O_CREAT | O_TRUNC, mode);
}
```

六、write

```
#include <unistd.h>

ssize_t write (
    int         fd,    // 文件描述符
    const void* buf,   // 缓冲区
    size_t      count // 期望写入的字节数
);
```

成功返回实际写入的字节数(0表示未写入)，失败返回-1。

size_t: unsigned int，无符号整数
ssize_t: int，有符号整数

范例：write.c

七、read

```
#include <unistd.h>

ssize_t read (
    int    fd,    // 文件描述符
```

```
    void*  buf,  // 缓冲区
    size_t count // 期望读取的字节数
);
```

成功返回实际读取的字节数(0表示读到文件尾)，
失败返回-1。

范例：read.c

二进制读写和文本读写。

范例：binary.c、text.c

练习：带覆盖检查的文件复制。

代码：copy.c

八、系统I/O与标准I/O
---

1. 当系统调用函数被执行时，需要切换用户态和内核态，
   频繁调用会导致性能损失。

2. 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。

范例：sysio.c、stdio.c

```
# time ./sysio

real    0m17.442s
user    0m0.000s
sys     0m0.284s

# time ./stdio

real    0m0.056s
user    0m0.000s
sys     0m0.009s
```

九、lseek
---

1. 每个打开的文件都有一个与其相关的“文件位置”。

2. 文件位置通常是一个非负整数，
   用以度量从文件头开始计算的字节数。

3. 读写操作都从当前文件位置开始，
   并根据所读写的字节数，增加文件位置。

4. 打开一个文件时，除非指定了O_APPEND，
   否则文件位置一律被设为0。

5. lseek函数仅将文件位置记录在内核中，
   并不引发任何I/O动作。

6. 在超越文件尾的文件位置写入数据，
   将在文件中形成空洞。

7. 文件空洞不占用磁盘空间，但被算在文件大小内。

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

off_t lseek (
    int   fd,     // 文件描述符
    off_t offset, // 偏移量
    int   whence  // 起始位置
);
```

成功返回当前文件位置，失败返回-1。

whence取值：

SEEK_SET - 从文件头
           (文件的第一个字节)。

SEEK_CUR - 从当前位置
           (上一次读写的最后一个字节的下一个位置)。

SEEK_END - 从文件尾
           (文件的最后一个字节的下一个位置)。

范例：seek.c

思考：既然lseek系统调用相当于标C库函数fseek，
那么是否存在与标C库函数ftell相对应的系统调用？

不存在，
因为通过lseek(fd,0,SEEK_CUR)就可以获得当前文件位置。

思考：如何获取文件的大小？

通过lseek(fd,0,SEEK_END)可以获得文件的大小。

十、打开文件的内核数据结构

通过ls -i可查看文件的i节点号。
i节点记录了文件的属性和数据在磁盘上的存储位置。
目录也是文件，存放路径和i节点号的映射表。

图示：open.bmp

范例：bad.c

十一、dup/dup2

```
#include <unistd.h>

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

成功返回文件描述符oldfd的副本，失败返回-1。

1. 复制一个已打开的文件描述符。

2. 返回的一定是当前未被使用的最小文件描述符。

3. dup2可由第二个参数指定描述符的值。
   若指定描述符已打开，则先关闭之。

4. 所返回的文件描述符副本，
   与源文件描述符，对应同一个文件表。

图示：dup.bmp

范例：dup.c

注意区分通过dup获得的文件描述符副本，
和两次open同一个文件的区别：

dup只复制文件描述符，不复制文件表。

```
fd1 \
     > 文件表 -> v节点 -> i节点
fd2 /
```

open创建新文件表，并为其分配新文件描述符。

```
fd1 -> 文件表1 \
                > v节点 -> i节点
fd2 -> 文件表2 /
```

图示：same.bmp

范例：same.c

作业：学生管理系统登录模块。
注册 - 增加用户名和密码，
登录 - 验证用户名和密码，
用户信息保存在文件中。

代码：mis.c