第四课　文件系统(下)

一、sync/fsync/fdatasync

1. 大多数磁盘I/O都通过缓冲进行，
   写入文件其实只是写入缓冲区，直到缓冲区满，
   才将其排入写队列。

2. 延迟写降低了写操作的次数，提高了写操作的效率，
   但可能导致磁盘文件与缓冲区数据不同步。

3. sync/fsync/fdatasync用于强制磁盘文件与缓冲区同步。

4. sync将所有被修改过的缓冲区排入写队列即返回，
   不等待写磁盘操作完成。

5. fsync只针对一个文件，且直到写磁盘操作完成才返回。

6. fdatasync只同步文件数据，不同步文件属性。

```
#include <unistd.h>

void sync (void);

int fsync (
    int fd
);
```

成功返回0，失败返回-1。

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

成功返回0，失败返回-1。

```
              +-fwrite-> 标准库缓冲 -fflush-+             sync
应用程序内存 -+                             +-> 内核缓冲 -fdatasync-> 磁盘(缓冲)
              +------------write------------+             fsync
```

二、fcntl

```
#include <fcntl.h>

int fcntl (
    int fd,  // 文件描述符
    int cmd, // 操作指令
    ...      // 可变参数，因操作指令而异
);
```

对fd文件执行cmd操作，某些操作需要提供参数。

1. 常用形式

```
#include <fcntl.h>

int fcntl (int fd, int cmd);
int fcntl (int fd, int cmd, long arg);
```

成功返回值因cmd而异，失败返回-1。

cmd取值：

F_DUPFD - 复制fd为不小于arg的文件描述符。
若arg文件描述符已用，
该函数会选择比arg大的最小未用值，
而非如dup2函数那样关闭之。

范例：dup.c

F_GETFD - 获取文件描述符标志。

F_SETFD - 设置文件描述符标志。

目前仅定义了一个文件描述符标志位FD_CLOEXEC：

0 - 在通过exec函数族所创建的进程中，
该文件描述符依然保持打开。

1 - 在通过exec函数族所创建的进程中，
该文件描述符将被关闭。

F_GETFL - 获取文件状态标志。
不能获取O_CREAT/O_EXCL/O_TRUNC。

注意：O_RDONLY标志的值为0，不能用位与判断：

```
if ((flags & O_ACCMODE) == O_RDONLY)
  // 有只读标志
```

其它标志可以用位与判断：

```
if (flags & O_WRONLY)
  // 有只写标志
```

F_SETFL - 追加文件状态标志。
只能追加O_APPEND/O_NONBLOCK。

范例：flags.c

2. 文件锁

```
#include <fcntl.h>

int fcntl (int fd, int cmd, struct flock* lock);
```

其中：

```
struct flock {
    short int l_type;   // 锁的类型：
                        // F_RDLCK/F_WRLCK/F_UNLCK
                        // (读锁/写锁/无锁)
    short int l_whence; // 偏移起点：
                        // SEEK_SET/SEEK_CUR/SEEK_END
                        // (文件头/当前位置/文件尾)
    off_t     l_start;  // 锁区偏移，从l_whence开始
    off_t     l_len;    // 锁区长度，0表示锁到文件尾
    pid_t     l_pid;    // 加锁进程，-1表示自动设置
};
```

cmd取值：

```
F_GETLK  - 测试lock所表示的锁是否可加。
           若可加则将lock.l_type置为F_UNLCK，
           否则通过lock返回当前锁的信息。

F_SETLK  - 设置锁定状态为lock.l_type，
           成功返回0，失败返回-1。
           若因其它进程持有锁而导致失败，
           则errno为EACCES或EAGAIN。

F_SETLKW - 设置锁定状态为lock.l_type，
           成功返回0，否则一直等待，
           除非被信号打断返回-1。
```

前提：A和B是同一个文件上的两个相互重叠的区域。

第一种情况：在写锁区上加写锁

进程1	进程2
打开文件，准备写A区	打开文件，准备写B区
调用fcntl()，给A区加写锁， fcntl()返回成功，A区被加上写锁	
调用write()，写A区	调用fcntl()，给B区加写锁， fcntl()阻塞或返回失败
调用fcntl()，解锁A区	fcntl()返回成功，B区被加上写锁
	调用write()，写B区
	调用fcntl()，解锁B区
关闭文件	关闭文件

第二种情况：在写锁区上加读锁

进程1	进程2
打开文件，准备写A区	打开文件，准备读B区
调用fcntl()，给A区加写锁， fcntl()返回成功，A区被加上写锁	
调用write()，写A区	调用fcntl()，给B区加读锁， fcntl()阻塞或返回失败
调用fcntl()，解锁A区	fcntl()返回成功，B区被加上读锁
	调用read()，读B区
	调用fcntl()，解锁B区
关闭文件	关闭文件

第三种情况：在读锁区上加写锁

进程1	进程2
打开文件，准备读A区	打开文件，准备写B区
调用fcntl()，给A区加读锁， fcntl()返回成功，A区被加上读锁	
调用read()，读A区	调用fcntl()，给B区加写锁， fcntl()阻塞或返回失败
调用fcntl()，解锁A区	fcntl()返回成功，B区被加上写锁
	调用write()，写B区
	调用fcntl()，解锁B区
关闭文件	关闭文件

第四种情况：在读锁区上加读锁

进程1	进程2
打开文件，准备读A区	打开文件，准备读B区
调用fcntl()，给A区加读锁， fcntl()返回成功，A区被加上读锁	
调用read()，读A区	调用fcntl()，给B区加读锁， fcntl()返回成功，B区被加上读锁
调用fcntl()，解锁A区	调用read()，读B区

| 调用fcntl()，解锁B区
---|---
关闭文件 | 关闭文件

1) 既可以锁定整个文件，也可以锁定特定区域。

2) 读锁(共享锁)、写锁(独占锁/排它锁)、解锁。

图示：rwlock.bmp、flock.bmp

3) 文件描述符被关闭(进程结束)时，自动解锁。

4) 劝谏锁(协议锁)、强制锁。

范例：lock1.c、lock2.c

5) 文件锁仅在不同进程间起作用。

6) 通过锁同步多个进程对同一个文件的读写访问。

范例：wlock.c、rlock.c

```
# wlock 达内科技 | # wlock 有限公司
wlock.txt
<乱码>

# wlock 达内科技 -l | # wlock 有限公司 -l
wlock.txt
达内科技有限公司

-----------------------------------------

# wlock 达内科技有限公司 | # rlock
<乱码>

# wlock 达内科技有限公司 -l | # rlock -l
达内科技有限公司
```

三、stat/fstat/lstat

获取文件属性。

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

int stat (
    const char*  path, // 文件路径
    struct stat* buf   // 文件属性
);

int fstat (
    int          fd, // 文件描述符
    struct stat* buf // 文件属性
);
```

```
int lstat (
    const char*  path, // 文件路径
    struct stat* buf   // 文件属性
);
```

成功返回0，失败返回-1。

stat函数跟踪软链接，lstat函数不跟踪软链接。

```
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;     // 用户ID
    gid_t     st_gid;     // 组ID
    dev_t     st_rdev;    // 特殊设备ID
    off_t     st_size;    // 总字节数
    blksize_t st_blksize; // I/O块字节数
    blkcnt_t  st_blocks;  // 占用块(512字节)数
    time_t    st_atime;   // 最后访问时间
    time_t    st_mtime;   // 最后修改时间
    time_t    st_ctime;   // 最后状态改变时间
};
```

st_mode(0TTSUGO)为以下值的位或：

```
S_IFDIR           - 目录         \
S_IFREG           - 普通文件      |
S_IFLNK           - 软链接        |
S_IFBLK           - 块设备        > TT (S_IFMT)
S_IFCHR           - 字符设备      |
S_IFSOCK          - Unix域套接字  |
S_IFIFO           - 有名管道     /
-------------------------------
S_ISUID           - 设置用户ID  \
S_ISGID           - 设置组ID     > S
S_ISVTX           - 粘滞        /
-------------------------------
S_IRUSR(S_IREAD)  - 用户可读    \
S_IWUSR(S_IWRITE) - 用户可写     > U (S_IRWXU)
S_IXUSR(S_IEXEC)  - 用户可执行  /
-------------------------------
S_IRGRP           - 同组可读    \
S_IWGRP           - 同组可写     > G (S_IRWXG)
S_IXGRP           - 同组可执行  /
-------------------------------
S_IROTH           - 其它可读    \
S_IWOTH           - 其它可写     > O (S_IRWXO)
S_IXOTH           - 其它可执行  /
```

1. 有关S_ISUID/S_ISGID/S_ISVTX的说明

1) 具有S_ISUID/S_ISGID位的可执行文件，
   其有效用户ID/有效组ID，
   并不取自由其父进程(比如登录shell)所决定的，

实际用户ID/实际组ID，
而是取自该可执行文件的用户ID/组ID。
如：/usr/bin/passwd

2) 具有S_ISUID位的目录，
其中的文件或目录除root外，
只有其用户可以删除。

3) 具有S_ISGID位的目录，
在该目录下所创建的文件，继承该目录的组ID，
而非其创建者进程的有效组ID。

4) 具有S_ISVTX位的可执行文件，
在其首次执行并结束后，
其代码区将被连续地保存在磁盘交换区中，
而一般磁盘文件中的数据块是离散存放的。
因此，下次执行该程序可以获得较快的载入速度。
现代Unix系统大都采用快速文件系统，
已不再需要这种技术。

5) 具有S_ISVTX位的目录，
只有对该目录具有写权限的用户，
在满足下列条件之一的情况下，
才能删除或更名该目录下的文件或目录：

A. 拥有此文件；
B. 拥有此目录；
C. 是超级用户。

如：/tmp

任何用户都可在该目录下创建文件，
任何用户对该目录都享有读/写/执行权限，
但除root以外的任何用户在目录下，
都只能删除或更名属于自己的文件。

2. 常用以下宏辅助分析st_mode

S_ISDIR() - 是否目录
S_ISREG() - 是否普通文件
S_ISLNK() - 是否软链接
S_ISBLK() - 是否块设备
S_ISCHR() - 是否字符设备
S_ISSOCK() - 是否Unix域套接字
S_ISFIFO() - 是否有名管道

范例：stat.c

四、access

```
#include <unistd.h>

int access (
    const char* pathname, // 文件路径
    int         mode      // 访问模式
```

```
);
```

1. 按实际用户ID和实际组ID(而非有效用户ID和有效组ID)，
   进行访问模式测试。

2. 成功返回0，失败返回-1。

3. mode取R_OK/W_OK/X_OK的位或，
   测试调用进程对该文件，
   是否可读/可写/可执行，
   或者取F_OK，测试该文件是否存在。

范例：access.c

五、umask

可以用umask命令查看/修改当前shell的文件权限屏蔽字：

```
# umask
0022

# umask 0033
# umask
0033
```

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

mode_t umask (
    mode_t cmask // 屏蔽字
);
```

1. 为进程设置文件权限屏蔽字，并返回以前的值，
   此函数永远成功。

2. cmask由9个权限宏位或组成(直接写八进制整数形式亦可，
   如0022 - 屏蔽同组和其它用户的写权限)：

```
S_IRUSR(S_IREAD)  - 用户可读
S_IWUSR(S_IWRITE) - 用户可写
S_IXUSR(S_IEXEC)  - 用户可执行
-----------------------------
S_IRGRP           - 同组可读
S_IWGRP           - 同组可写
S_IXGRP           - 同组可执行
-----------------------------
S_IROTH           - 其它可读
S_IWOTH           - 其它可写
S_IXOTH           - 其它可执行
```

3. 设上屏蔽字以后，此进程所创建的文件，
   都不会有屏蔽字所包含的权限。

范例：umask.c

六、chmod/fchmod

修改文件的权限。

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

int chmod (
    const char* path, // 文件路径
    mode_t      mode  // 文件权限
);

int fchmod (
    int    fd,  // 文件描述符
    mode_t mode // 文件权限
);
```

成功返回0，失败返回-1。

mode为以下值的位或(直接写八进制整数形式亦可，
如07654 - rwSr-sr-T)：

```
S_ISUID           - 设置用户ID
S_ISGID           - 设置组ID
S_ISVTX           - 粘滞
-----------------------------
S_IRUSR(S_IREAD)  - 用户可读
S_IWUSR(S_IWRITE) - 用户可写
S_IXUSR(S_IEXEC)  - 用户可执行
-----------------------------
S_IRGRP           - 同组可读
S_IWGRP           - 同组可写
S_IXGRP           - 同组可执行
-----------------------------
S_IROTH           - 其它可读
S_IWOTH           - 其它可写
S_IXOTH           - 其它可执行
```

范例：chmod.c

注意：chmod/fchmod同样受umask权限屏蔽字影响。

七、chown/fchown/lchown

```
# chown <uid>:<gid> <file>
```

修改文件的用户和组。

```
#include <unistd.h>

int chown (
    const char* path,  // 文件路径
    uid_t       owner, // 用户ID
    gid_t       group  // 组ID
);
```

```
int fchown (
    int   fildes, // 文件描述符
    uid_t owner,  // 用户ID
    gid_t group   // 组ID
);

int lchown (
    const char* path,  // 文件路径(不跟踪软链接)
    uid_t       owner, // 用户ID
    gid_t       group  // 组ID
);
```

成功返回0，失败返回-1。

注意：

1. 用户ID/组ID取-1表示不修改。

2. 超级用户可以修改任何文件的用户和组，
   普通用户只能修改自己文件的用户和组。

八、truncate/ftruncate

修改文件的长度，截短丢弃，加长添零。

```
#include <unistd.h>

int truncate (
    const char* path,  // 文件路径
    off_t       length // 文件长度
);

int ftruncate (
    int   fd,     // 文件描述符
    off_t length // 文件长度
);
```

成功返回0，失败返回-1。

范例：trunc.c、mmap.c

注意：对于文件映射，
私有映射(MAP_PRIVATE)将数据写到缓冲区而非文件中，
只有自己可以访问。
而对于内存映射，
私有(MAP_PRIVATE)和公有(MAP_SHARED)没有区别，
都是仅自己可以访问。

九、link/unlink/remove/rename
---

link: 创建文件的硬链接(目录条目)。

unlink: 删除文件的硬链接(目录条目)。
只有当文件的硬链接数降为0时，文件才会真正被删除。

若该文件正在被某个进程打开，
其内容直到该文件被关闭才会被真正删除。

remove: 对文件同unlink，
对目录同rmdir (不能删非空目录)。

rename: 修改文件/目录名。

```
#include <unistd.h>

int link (
    const char* path1, // 文件路径
    const char* path2  // 链接路径
);

int unlink (
    const char* path // 链接路径
);

#include <stdio.h>

int remove (
    const char* pathname // 文件/目录路径
);

int rename (
    const char* old, // 原路径名
    const char* new  // 新路径名
);
```

成功返回0，失败返回-1。

注意：硬链接只是一个文件名，即目录中的一个条目。
软链接则是一个独立的文件，
其内容是另一个文件的路径信息。

十、symlink/readlink
---

symlink: 创建软链接。目标文件可以不存在，
也可以位于另一个文件系统中。

readlink: 获取软链接文件本身(而非其目标)的内容。
open不能打开软链接文件本身。

```
#include <unistd.h>

int symlink (
    const char* oldpath, // 文件路径(可以不存在)
    const char* newpath  // 链接路径
);
```

成功返回0，失败返回-1。

```
ssize_t readlink (
    const char* restrict path,    // 软链接文件路径
```

```
    char* restrict       buf,     // 缓冲区
    size_t               bufsize // 缓冲区大小
                                 // 以字节为单位，不含结尾空字符
);
```

成功返回实际拷入缓冲区buf中软链接文件内容的字节数，
失败返回-1。

注意：readlink在将软链接文件内容拷入缓冲区buf时，
不负责追加结尾空字符。

范例：slink.c

十一、mkdir/rmdir
---

mkdir: 创建一个空目录。

rmdir: 删除一个空目录。

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

int mkdir (
    const char* path, // 目录路径
    mode_t      mode  // 访问权限，
                      // 目录的执行权限(x)表示可进入
);

#include <unistd.h>

int rmdir (
    const char* path // 目录路径
);
```

成功返回0，失败返回-1。

十二、chdir/fchdir/getcwd
---

chdir/fchdir: 更改当前工作目录。
工作目录是进程的属性，只影响调用进程本身。

getcwd: 获取当前工作目录。

```
#include <unistd.h>

int chdir (
    const char* path // 工作目录路径
);

int fchdir (
    int fildes // 工作目录描述符(由open函数返回)
);
```

成功返回0，失败返回-1。

```
char* getcwd (
    char*  buf, // 缓冲区
    size_t size // 缓冲区大小，
                // 以字节为单位，含结尾空字符
);
```

成功返回当前工作目录字符串指针，失败返回NULL。

注意：getcwd在将当前工作目录字符串拷入缓冲区buf时，
会自动追加结尾空字符。

范例：dir.c

十三、opendir/fdopendir/closedir/readdir/rewinddir/telldir/seekdir
---

opendir/fdopendir: 打开目录流。

closedir: 关闭目录流。

readdir: 读取目录流。

rewinddir: 复位目录流。

telldir: 获取目录流位置。

seekdir: 设置目录流位置。

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

DIR* opendir (
    const char* name // 目录路径
);

DIR* fdopendir (
    int fd // 目录描述符(由open函数返回)
);
```

成功返回目录流指针，失败返回NULL。

```
int closedir (
    DIR* dirp // 目录流指针
);
```

成功返回0，失败返回-1。

```
struct dirent* readdir (
    DIR* dirp // 目录流指针
);
```

成功返回下一个目录条目结构体的指针，
到达目录尾(不置errno)或失败(设置errno)返回NULL。

```
struct dirent {
    ino_t          d_ino;       // i节点号
```

```
    off_t          d_off;       // 下一条目在目录流中的位置
    unsigned short d_reclen;    // 记录长度
    unsigned char  d_type;      // 文件类型
    char           d_name[256]; // 文件名
};
```

d_type取值：

```
DT_DIR     - 目录
DT_REG     - 普通文件
DT_LNK     - 软链接
DT_BLK     - 块设备
DT_CHR     - 字符设备
DT_SOCK    - Unix域套接字
DT_FIFO    - 有名管道
DT_UNKNOWN - 未知
```

图示：de.bmp

范例：list.c

练习：打印给定路径下的目录树。

代码：tree.c

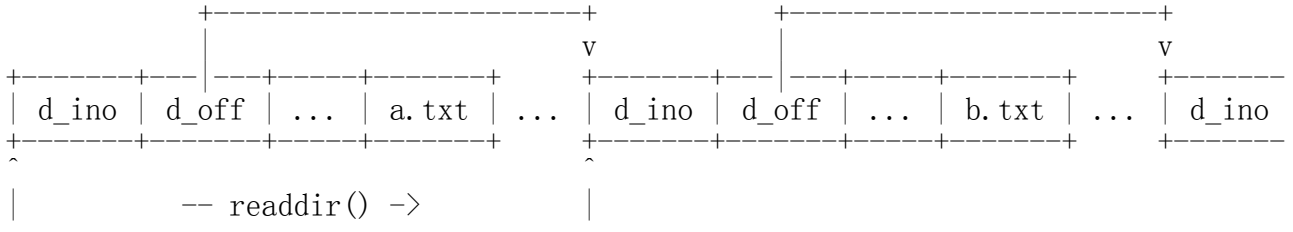
```
void rewinddir (
    DIR* dirp // 目录流指针
);

long telldir (
    DIR* dirp // 目录流指针
);
```

成功返回目录流位置，失败返回-1。

```
void seekdir (
    DIR* dirp,  // 目录流指针
    long offset // 目录流位置
);
```

目录流：



范例：seek.c