三、檔案與目錄I/O

File I/O

- Basic five functions
  - open
  - read
  - write
  - lseek
  - close
- Unbuffered I/O
  - Each function invokes a system call

File Descriptors

- All open files are referred to by the file descriptors in the kernel.
- A non-negative integer
  - File descriptor 0: standard input
  - File descriptor 1: standard output
  - File descriptor 2: standard error
    (Reference: /usr/include/unistd.h)

open Function

```
#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);
```

open Function(flags 1/2)

- O_RDONLY
  - 開啟一個唯讀的檔案
- O_WRONLY
  - 開啟一個唯寫的檔案
- O_RDWR
  - 開啟一個可讀可寫的檔案
- O_CREAT
  - 建立新檔案
- O_EXCL
  - 與O_CREAT一起使用，當所要建立的檔案已經存在時產生錯誤
- O_TRUNC
  - 與O_WRONLY或O_RDWR一起使用，檔案長度變為零
- O_APPEND
  - 從檔案的尾端接續寫入

open Function(flags 2/2)

- O_NONBLOCK or O_NDELAY
  - I/O不做block的動作
- O_SYNC
  - 所有寫入的資料都要等到真正完成後才返回
- O_NOFOLLOW
  - 開啟的檔案如果是symbolic link則產生錯誤
- O_DIRECT
  - 降低cache的影響
- O_ASYNC
  - 當I/O可讀或可寫時產生SIGIO signal
- O_LARGEFILE
  - 支援超過 4G大小的檔案

open Function(mode)

- S_IRWXU 700
- S_IRUSR 400
- S_IWUSR 200
- S_IXUSR 100
- S_IRWXG 070
- S_IRGRP 040
- S_IWGRP 020
- S_IXGRP 010
- S_IRWXO 007
- S_IROTH 004
- S_IWOTH 002
- S_IXOTH 001

creat Function

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

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

- Is equivalent to

  open (pathname, O_WRONLY | O_CREATE |OTRUNC, mode)

close Function

```
#include <unistd.h>

    int close(int fd);
```

- 關閉檔案，釋放所有record locks
- 當行程結束時，所有開啟的檔案都會自動關閉

lseek function

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

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

- 每一個開啟的檔案都有一個檔案指標(current file offset)，它是一個非負的整數，代表與開頭間隔多少位元組
- 開啟檔案時檔案指標的值預設為零，除非使用O_APPEND
- 檔案指標的值可以由lseek函式改變
  - SEEK_SET: 把檔案指標移到由檔案開頭算起offset的位置
  - SEEK_CUR:把檔案指標移到由目前算起offset的位置
  - SEEK_END:把檔案指標移到由到檔案尾端算起offset的位置

read function

```
#include <unistd.h>

    ssize_t read(int fd, void *buf, size_t count);
```

- 傳回讀到多少位元組，如果讀到檔尾則回傳0
- 每次讀取成功後檔案指標會往後移動回傳值
- 傳回值與count不同的可能原因
  - 讀到EOF
  - 讀取的裝置為terminal device
  - 讀取的裝置為網路裝置
  - 讀取的裝置為record-oriented device，像是磁帶

write function

```
#include <unistd.h>

    ssize_t write(int fd, const void *buf, size_t count);
```

- 由目前檔案指標的位置開始寫入，如果在open時指定O_APPEND，則每次write都是接在檔案的最末端
- 每次寫入成功後檔案指標會往後移動count
- 傳回值通常跟count相同，如果不相同代表有錯誤產生，通常造成錯誤的原因為
  - 磁碟滿了
  - 超出該行程對於檔案大小的限制

I/O Efficiency(1/2)

```
#include "ourhdr.h"
#define BUFFSIZE 8192

int main(void) {
        int n;
        char buf[BUFFSIZE];
        while ( (n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
                if (write(STDOUT_FILENO, buf, n) != n)
                        err_sys("write error");
        if (n < 0) err_sys("read error");
        exit(0);
}
```

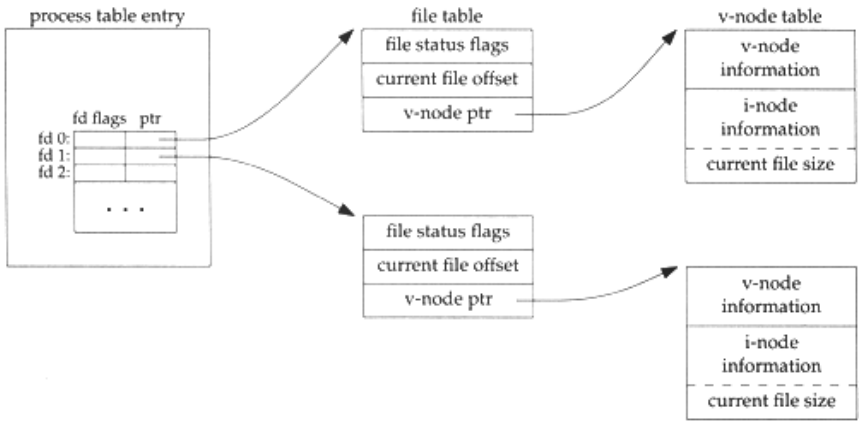

I/O Efficiency(2/2)

BUFFSIZE	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)	#loops
1	23.8	397.9	423.4	1468802
2	12.3	202.0	215.2	734401
4	6.1	100.6	107.2	367201
8	3.0	50.7	54.0	183601
16	1.5	25.3	27.0	91801
32	0.7	12.8	13.7	45901
64	0.3	6.6	7.0	22950
128	0.2	3.3	3.6	11475
256	0.1	1.8	1.9	5738
512	0.0	1.0	1.1	2869
1024	0.0	0.6	0.6	1435
2048	0.0	0.4	0.4	718
4096	0.0	0.4	0.4	359
8192	0.0	0.3	0.3	180
16384	0.0	0.3	0.3	90
32768	0.0	0.3	0.3	45
65536	0.0	0.3	0.3	23
131072	0.0	0.3	0.3	12

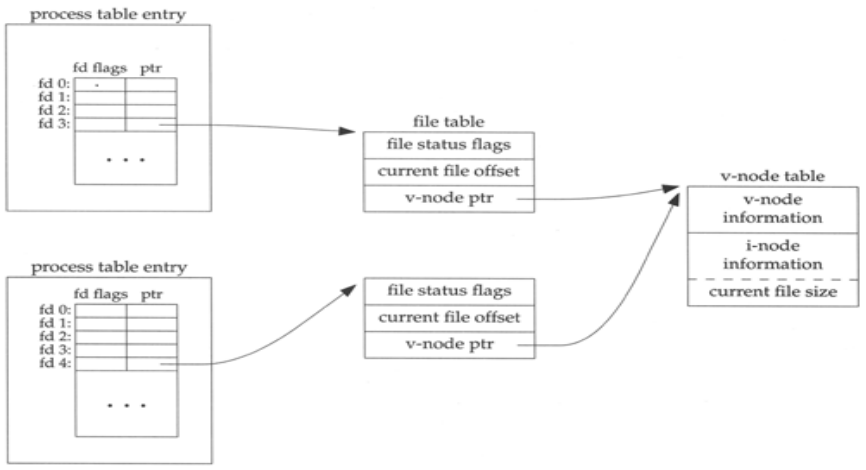
File Sharing(1/3)

- Kernel中的三個資料結構
  - Process table
    - 每一個process都會有一個entry在process table 中
    - 每一個process entry有該process開啟的檔案資訊
      - 檔案的flag (read, write, append, sync, nonblocking等等)
      - 指向file table的指標
  - File table
    - 每一個開啟的檔案都會有一個entry在file table中
      - 檔案的狀態
      - 檔案目前的指標位置
      - 指向i-node table的指標
  - i-node table

File Sharing(2/3)



File Sharing(3/3)



Atomic Operations(1/2)

- 不可切割執行的操作稱為atomic operations
- 假設下面程式片段

```
lseek(fd,0L,2);          /*移到檔尾*/
write(fd,buf,100);       /*寫入*/
```

- 如果該檔案只由一個行程操作則沒有問題，但是如果同時有許多行程操作就會產生問題
- 在開檔時加入O_APPEND就可以解決上述問題

Atomic Operations(2/2)

- 另一個例子
  - open檔案時同時使用O_CREAT與O_EXCL，如果建立的檔案已經存時在就會失敗
  - 上述整個流程也為一個atomic operation
- 如果沒有atomic operation，只能用嘗試的方法，如下(但還是可能有問題)：

```
if ((fd=open(pathname, O_WRONLY)) <0 )
        if(errno==ENOENT){
                if ((fd=creat(pathname,mode))<0)
                        err_sys("create error");
        }else
                err_sys("open error");
```


dup and *dup2* Functions(1/3)

```
#include <unistd.h>

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

- dup將oldfd複製到目前最小可用的file descriptor
- dup2將oldfd複製到newfd，如果newfd已經開啟，則newfd會先被關閉
- 新的fd將與oldfd共用同一份file table

dup and *dup2* Functions(2/3)

- dup(filefd) 與下列函式相同

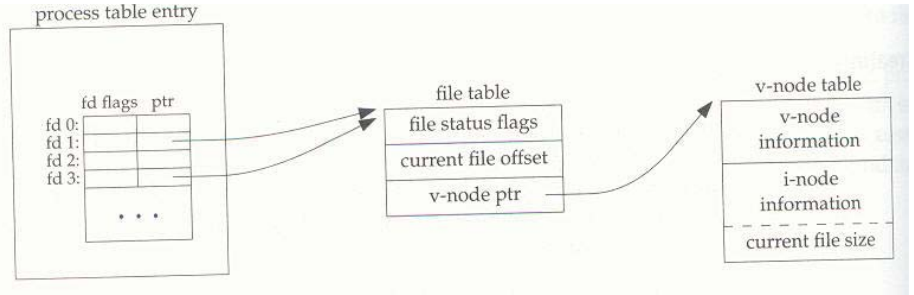
  fcntl(filefd, F_DUPED, 0);

- dup2(filefd,filefd2)與下列函式相同
  (嚴格來說dup2為atomic operation，而下兩個函式並非atomic operation)

  close(filefd2);

  fcntl(filefd, F_DUPED, filefd2);

dup and *dup2* Functions(3/3)



fcntl Function(1/2)

```
#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);
```

- fcntl的功用
  - 複製已經存在的file descriptor(cmd=F_DUPFD)
  - 取得/設定file descriptor的flag(cmd=F_GETFD/F_SETFD)
  - 取得/設定file descriptor flag的狀態(cmd=F_GETFL/F_SETFL)
  - 取得/設定非同步I/O的擁有者 (cmd=F_GETOWN/F_SETOWN)
  - 取得/設定record locks(cmd=F_GETLK, F_SETLK, F_SETLKW)

fcntl Function(2/2)

- F_DUPFD
  - 複製 file descriptor
- F_GETFD
  - 取得file descriptor的flag
- F_SETFD
  - 設定file descriptor的 flag
- F_GETFL
  - 取得file descriptor的status flag
- F_SETFL
  - 設定file descriptor的status flag
- F_GETOWN
  - 取得目前SIGIO signal是送到那個pid
- F_SETOWN
  - 設定目前SIGIO signal要送到那個pid
- F_GETSIG
  - 取得目前是送哪一種signal
- F_SETSIG
  - 設定目前要送哪一種signal

ioctl Function(1/2)

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

    int ioctl(int d, int request, ...)
```

- 所有不能用其它函式處理的I/O操作
  - 例如控制磁帶機
    - 寫入end-of-file mark
    - 倒帶
    - 往前轉

ioctl Function(2/2)

類別	常數名稱
Disk labels	DIOxxx
File I/O	FIOxxx
Mag tape I/O	MTIOxxx
Socket I/O	SIOxxx
Terminal I/O	TIOxxx

/dev/fd

- 開啟/dev/fd/n 就跟dup n是相同的功用

  fd = open("/dev/fd/0", mode);

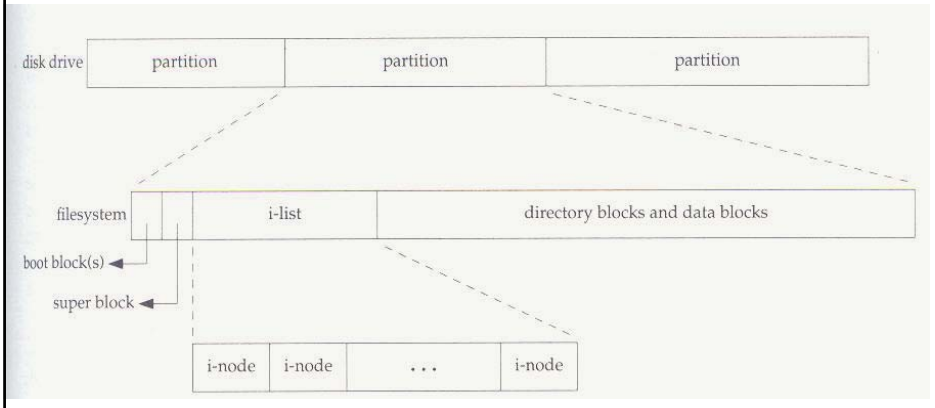
  fd = dup(0);

- mode必須為原開啟檔案模式的子集，例如原本的檔案是以唯讀方式開啟，則mode只能是唯讀而不能寫。
- /dev/fd通常是shell在用的

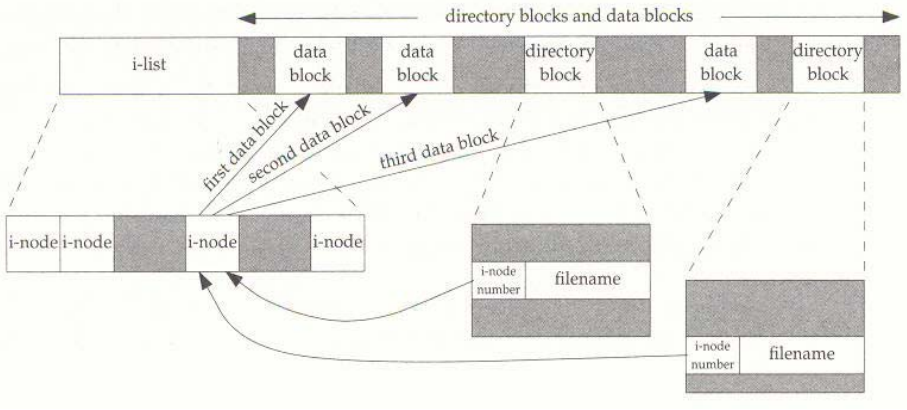
  filter file2 | cat file1 – file3 ||pr

  filter file2 | cat file1 /dev/fd/0 file3 ||pr

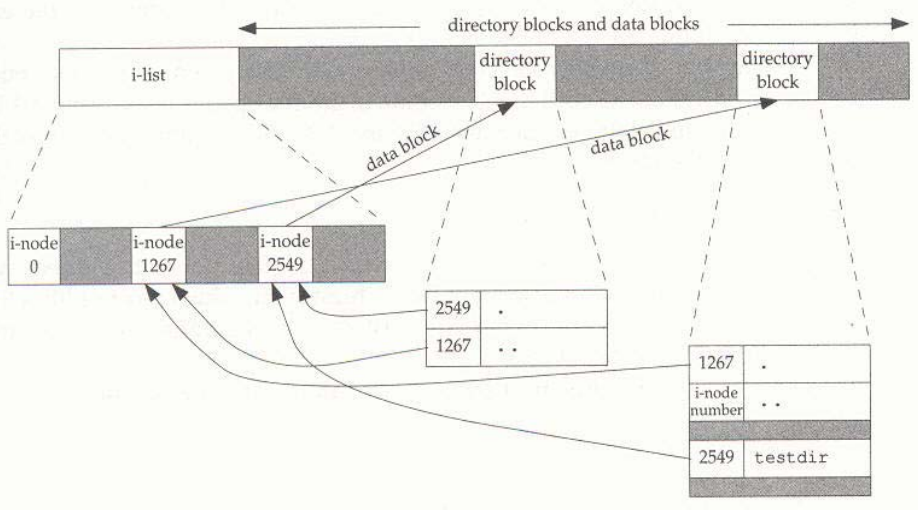
檔案系統(1/3)



檔案系統(2/3)



檔案系統(3/3)



取得檔案相關資訊

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

    int stat(const char *file_name, struct stat *buf);
    int fstat(int filedes, struct stat *buf);
    int lstat(const char *file_name, struct stat *buf);
```

- lstat與stat十分相似，其間只有一個差異，當檔案是symbolic link時，lstat會得到該symbolic link的資訊，而stat會得到該symbolic link所指檔案的資訊。
- 第二個參數buf所需的空間必須由我們自己配置

struct stat

```
struct stat {
        dev_t       st_dev;     /* device */
        ino_t       st_ino;     /* inode */
        mode_t      st_mode;    /* protection */
        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 type (if inode device) */
        off_t       st_size;    /* total size, in bytes */
        blksize_t   st_blksize; /* blocksize for filesystem I/O */
        blkcnt_t    st_blocks;  /* number of blocks allocated */
        time_t      st_atime;   /* time of last access */
        time_t      st_mtime;   /* time of last modification */
        time_t      st_ctime;   /* time of last change */
};
```

可以在linux下指令：man 2 stat看更詳細的說明

檔案的類型

- Regular file (S_ISREG())
  - 對於kernel來說，並不管檔案內容是text或是binary，而是由應用程式自己決定，大部分的檔案都為此類。
- Directory file (S_ISDIR())
  - 其檔案內容擺放該目錄下所在檔案的名稱與指向這些檔案的指標
- Character special file (S_ISCHR())
- Block special file (S_ISBLK())
- FIFO (S_ISFIFO())
- Socket (S_ISSOCK())
- Symbolic link (S_ISLNK())
  - 其檔案內容擺放指向檔案的名稱

判斷檔案類型

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

int main(void){
        struct stat buf;
        lstat("test.txt",&buf);
        if(S_ISREG(buf.st_mode)) printf("Regular\n");
}
```

- 可以用S_ISREG等巨集來測試檔案是否屬於該類型，像S_ISREG就是用來測試檔案是否為regular file
- S_ISDIR的巨集定義如下：
  #define S_ISDIR(mode) (((mode) & S_IFMT) == S_IFDIR)

Set-User-ID與Set-Group-ID

- 每個行程(process)擁有四種ID，這些ID用來判斷使用者是否有權限存取某些檔案
  - Real user ID (uid)
    - 執行使用者的真實身份
  - Real group ID (gid)
    - 執行使用者所屬的群組
  - Effective user ID (euid)
    - 通常與uid相同，除非檔案有set-user-id，如果有則執行時的euid會變成該檔案的擁有者
  - Effective group ID (egid)
    - 通常與gid相同，除非檔案有set-group-id，如果有則執行時的egid會變成該檔案的擁有群組
- Set-user-id與set-group-id的使用時機為，當某程式要存取原使用者無法存取的檔案時，例如/etc/passwd只有root能夠寫，但是為了讓全部的使用者可以改密碼，因此/usr/bin/passwd就必須set-user-id

檔案的存取權限(1/2)

- 當要開啟任何檔案時，必須符合下列條件
  - 必須擁有該檔所在目錄的執行權限，例如要開啟檔案/1/2/test，則必須擁有/，/1與/1/2的執行權限，因此可以稱這個權限為search bit。
  - 必須擁有該檔案的相關權限，如果是唯獨則必須有讀的權限，如果是寫則必須擁有寫的權限，依此類推。
- 目錄的執行權限與讀取權限所代表的意義不同
  - 執行權限用來代表可否進入該目錄
  - 讀取權限用來代表可否列出該目錄下的檔案

檔案的存取權限(2/2)

- 當要建立任何檔案時，必須符合下列條件：
  - 必須擁有該檔所屬目錄的寫入與執行權限
- 當要刪除任何檔案時，必須符合下列條件：
  - 必須擁有該檔所屬目錄的寫入與執行權限
  - 不需要擁有該檔的讀取或寫入權限
- 當要執行任何檔案時，必須符合下列條件：
  - 必須擁有該檔的執行權限，而且檔案必須為regular file
- 擁有權限的判斷是以effective uid和effective gid為準

新建檔案與新建目錄的擁有者

- 當一個新檔或新目錄被產生時，它的擁有者設定方式如下：
  - 新檔的擁有者會設為effective user id
  - 新檔的群組則有兩種方式
    - 設為effective group id
    - 設為該檔所屬目錄的擁有群組
      (將所屬目錄set-group-id即可)

判斷是否擁有某種權限

```
#include <unistd.h>

    int access(const char *pathname, int mode);
```

- 系統是使用euid和egid來做權限判斷，而access是用真實的uid和gid來做權限判斷，因此如果希望某些檔案在set-user-id後還是不能存取的話，可以用access這個函式來判斷。
- mode可以為R_OK, W_OK, X_OK和F_OK，分別判斷讀，寫，執行，與檔案是否存在。

檔案建立遮罩

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

    mode_t umask(mode_t mask);
```

- 用來設定檔案建立遮罩，維持安全性。
- 檔案的原有建立權限，在扣掉遮罩後才為真正的權限。例如原來我們要建立一個權限為777(rwxrwxrwx)的檔案，如果用umask設定遮罩為333(-wx-wx-wx)，則最後檔案的權限成為444(r--r--r--)

改變檔案權限

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

    int chmod(const char *path, mode_t mode);
    int fchmod(int fildes, mode_t mode);
```

- 可以用chmod或fchmod來改變檔案權限，chmod可以直接給檔名，而fchmod是用來改變已經開啟的檔案(也就是有file descriptor)
- 行程的euid必須要是該檔案的擁有者(或是superuser)才可以改變該檔權限
- 只有superuser可以設定sticky bit(S_ISVTX)
- 只有當檔案的所屬group與行程的egid相同時，才可以set-group-id

Sticky Bit (S_ISVTX)

- 如果一個可執行的檔案被設sticky bit，則作業系統會儘可能的將該檔案保留在記憶體中，以減少載入的時間。
- 如果sticky bit設在目錄上，則該目錄下的檔案只有在下列情況時才能被砍除或改名(例如/tmp的權限就為如此)：
  - 必須擁有該目錄寫的權限
  - 另為必須符合下列三條件之一
    - 擁有該檔案
    - 擁有該目錄
    - 為superuser

改變檔案的擁有者

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

    int chown(const char *path, uid_t owner, gid_t group);
    int fchown(int fd, uid_t owner, gid_t group);
    int lchown(const char *path, uid_t owner, gid_t group);
```

- 如果_POSIX_CHOWN_RESTRICTED生效的話，只有superuser可以改變檔案的擁有者，非superuser只能改變擁有群組
- Set-user-id和set-group-id在這些函式呼叫後會自動被清除。

檔案大小

- 在struct stat中的st_size紀錄著檔案的大小
  - 對於regular file來說，檔案大小為0是允許的
  - 對於directory來說，檔案大小為一整數(4096)的倍數
  - 對於symbolic link來說，檔案大小為指向該檔案的路徑與檔名長度
- st_blksize紀錄喜好的緩衝區大小
- st_blocks紀錄檔案佔據多少個512-bytes的區塊，其中st_size/512並不一定與st_blocks相等，因為檔案中可能存在空洞

截斷檔案

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

    int truncate(const char *path, off_t length);
    int ftruncate(int fd, off_t length);
```

- 這兩個函式用來將檔案縮減到指定的大小，truncate可以直接給檔名，而ftruncate用來處理已開啟的檔案(也就是有file descriptor)

連結, 刪除與搬移檔案(1/2)

```
#include <unistd.h>

    int link(const char *oldpath, const char *newpath);
    int unlink(const char *pathname);
```

- link函式用來建立hard link，也就是建立新的directory entry。只有superuser可以建立目錄的hard link，因為這個動作很容易讓檔案系統產生迴圈
- unlink函式用來刪除現存的directory entry，並會將i-node中的link count減一，當link count未歸零前，該檔案還是可以透過別的link存取到
- 要刪除一個檔案必須符合下列條件
  - 必須擁有該目錄的寫入與執行權限
  - 如果該目錄有設定sticky bit，則必須擁有該目錄寫的權限，並且要符合下列三個條件之一 1. 擁有該檔案 2. 擁有該目錄 3. 為superuser
- 在檔案還在開啟時，unlink並不會刪除該檔案的內容，所以有些程式為了避免有些暫存檔案忘記或是異常時沒有砍掉，常常在open之後馬上unlink，等到程式結束時(檔案被關閉)，檔案才會真正被砍掉
- 在unlink中，如果pathname指到的是symbolic link，則是symbolic link被unlink，而不是所指的檔案

連結, 刪除與搬移檔案(2/2)

```
#include <stdio.h>

    int remove(const char *pathname);
    int rename(const char *oldpath, const char *newpath);
```

- remove用來unlink檔案或目錄，如果參數是檔案，則跟unlink函式相同，如果是目錄則跟rmdir函式相同
- rename可以用來將檔案或目錄重新命名(搬移)

符號連結(symbolic link)

```
#include <unistd.h>

    int symlink(const char *oldpath, const char *newpath);
    int readlink(const char *path, char *buf, size_t bufsiz);
```

- 符號連結是一種間接的指標，不像hard link是直接的指標
- Hard link擁有下列限制，而符號連結沒有
  - link與連結的檔案只能在同一個檔案系統中
  - 只有superuser可以建立目錄的hard link
- 注意各個函式遇到符號連結的處理方式(是存取符號連結本身，或是存取所指檔案)，例如stat會存取符號連結所指的檔案，而lstat會存取符號連結本身
- symlink函式用來建立符號連結
- readlink函式用來讀取符號連結，因為open會讀到符號連結所指的檔案，所以特別有readlink這個函式來讀取符號連結本身

檔案的三種時間

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

    int utime(const char *filename, struct utimbuf *buf);
```

- Last access time (st_atime)
- Last modification time (st_mtime)
- Last change time (st_ctime)

目錄的建造與刪除

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

    int mkdir(const char *pathname, mode_t mode);

#include <unistd.h>

    int rmdir(const char *pathname);
```

- mkdir函式用來建立目錄，其權限為 (mode & ~umask)
- rmdir函式用來刪除目錄，被刪除的目錄必須是空的才行(除了.與..之外沒有其它directory entry)

讀取目錄

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

    DIR *opendir(const char *name);
    struct dirent *readdir(DIR *dir);
    void rewinddir(DIR *dir);
    int closedir(DIR *dir);
```

- 目錄也為檔案的一種，只是其檔案內容所包含的是一些檔名與指標，因此如果用open與read函式來讀取，則必須自行處理這些資料結構，所以有另一套函式專門處理這類工作。要列出一個目錄下的檔案有下列步驟：
  - 用opendir函式來開啟目錄
  - 用readdir函式來讀取目錄(或是用rewinddir函式將要讀取的指標移到最前頭的directory entry)
  - 用closedir函式來關閉目錄

切換目錄

```
#include <unistd.h>

    int chdir(const char *path);
    int fchdir(int fd);
    char *getcwd(char *buf, size_t size);
```

- 前兩個函式用來更改目前的所在目錄(current directory)，chdir可以直接給檔名，而fchdir用來切換到已開啟的目錄(也就是有file descriptor)
- getcwd函式用來取得目前的工作路徑

將緩衝區的資料寫到硬碟

```
#include <unistd.h>

    void sync(void);
    int fsync(int fd);
    int fdatasync(int fd);
```

- Linux檔案系統可能不會馬上將在記憶體中的資料真正的寫到磁碟上，我們可以透過這些函式強制作業系統寫入。
- sync函式就跟指令sync相同，要求作業系統將所有檔案緩衝區的資料寫入磁碟。呼叫這個函式只會讓作業系統安排這件事(不見得馬上做)，因此這個函式不會等做完才返回。因此這個函式執行完不代表緩衝區的資料都寫入磁碟了
- fsync函式可以指定要將那個檔案緩衝區寫入磁碟，這個函式會等真正寫入後才返回
- fdatasync與fsync相似，不過只將data block寫入，i-node的資料則不寫入(可以降低disk I/O的次數)

EXT2 檔案系統特有的屬性

CODE. 3-1

- ext2檔案系統上的檔案可以設定一些特別屬性，列舉如下：
  - 該檔不修改access time
  - 該檔所有write呼叫都為synchronous
  - 該檔只能附加(append only)
  - 該檔自動做壓縮
  - 該檔不做備份
  - 該檔不能修改，刪除，改名(immutable)
  - 該檔必須安全砍除
- 可以參考chattr與lsattr這兩個指令
- 可以用ioctl函式來更改這些設定(參考/usr/include/linux/ext2_fs.h中的inodes flags)