檔案輸入與輸出

中正大學，作業系統實驗室
羅習五 陽春副教授
shiwulo@gmail.com

單元介紹

- Linux內的「檔案」與file hole
  - 範例：最簡單的mycp
- lseek與file hole
  - 範例：支援file hole的mycp2
- file lock　(flock & lockf)
- sync & fsync & fdatasync

檔案在Linux內是什麼樣子

OS 傳輸 LBA (Logical Block Address) number 給 disk,
disk 會回傳一個 block (4K)

檔案（file）

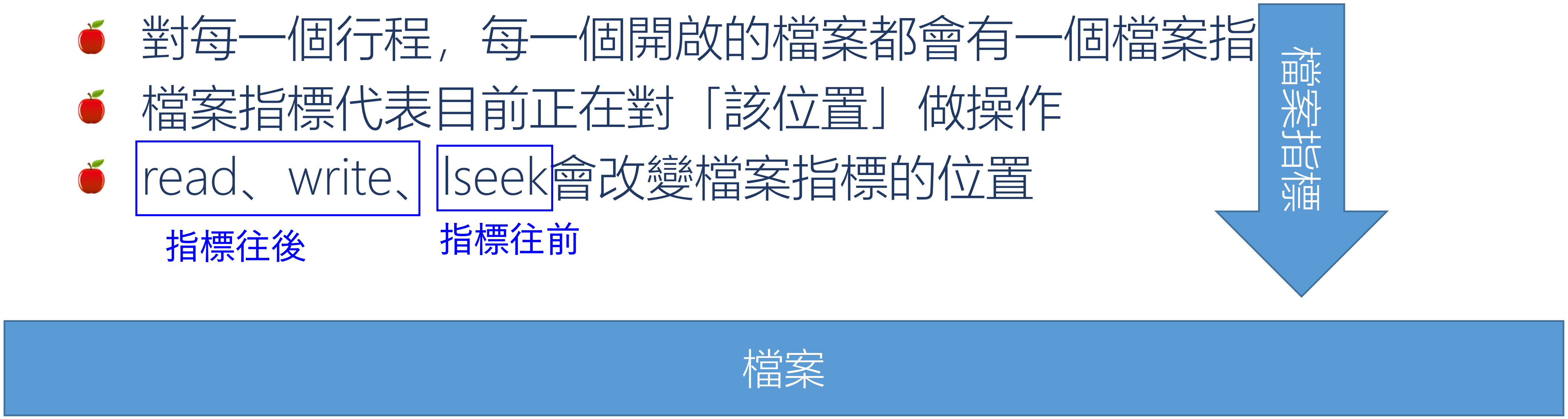
UNIX system 使用 inode (Index Node) 對檔案進行編號

- 檔案是一堆數據的有序集合
- 對作業系統而言，可以由「目錄系統」找到一個檔案在硬碟上的位置
- 對程式而言，必須先告訴作業系統，準備「使用」哪些檔案，作業系統會「開啟」該檔案，並給該檔案一個代碼（file descriptor），隨後該程式使用該「代碼」操作該檔案

檔案（file）

- 對每一個行程，每一個開啟的檔案都會有一個檔案指
- 檔案指標代表目前正在對「該位置」做操作
- read、write、lseek會改變檔案指標的位置

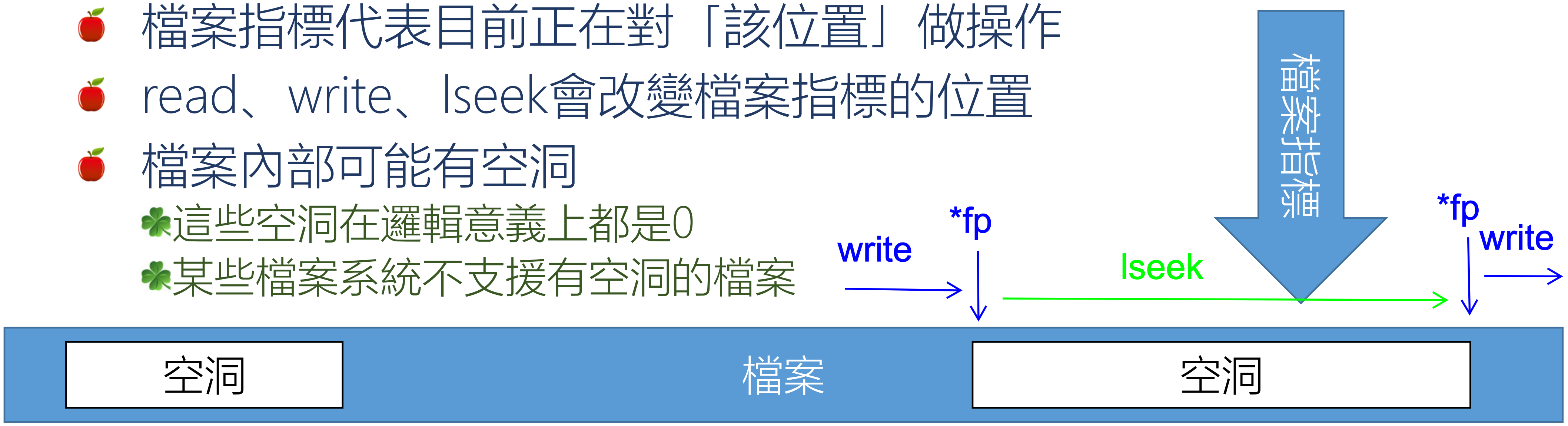
指標往後　　指標往前



檔案（file）

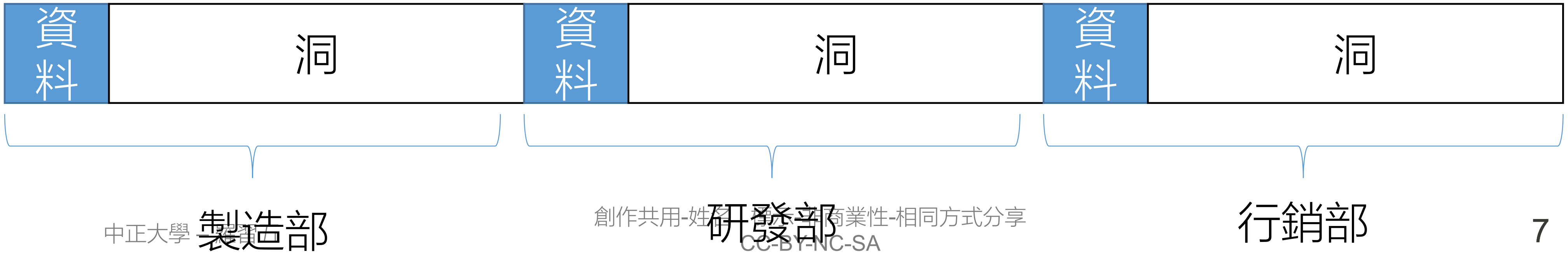
DOS, FAT32 不支援 file hole

- 對每一個行程，每一個開啟的檔案都會有一個檔案指標
- 檔案指標代表目前正在對「該位置」做操作
- read、write、lseek會改變檔案指標的位置
- 檔案內部可能有空洞
  - 這些空洞在邏輯意義上都是0
  - 某些檔案系統不支援有空洞的檔案



為什麼檔案系統需要支援「空洞」

- 例如一間公司，員工編號共五碼，第1XXXX代表製造部、2XXXX代表研發部、3XXXX代表行銷部
- 如果檔案系統支援「洞」，那麼可以直接使用員工編號當index，而不需要擔心浪費磁碟空間的問題，如：



先『不考慮』file hole
用一個例子開始：mycp

mycp.c

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

#define BUF_SIZE 4096

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

    int inputFd, outputFd;
    ssize_t numIn, numOut;
    char buffer[BUF_SIZE];

    inputFd = open (argv [1], O_RDONLY);
    if (inputFd == -1) {
        perror ("cannot open the file for read"); exit(1); }

    outputFd = open(argv[2], O_WRONLY | O_CREAT, S_IRUSR| S_IWUSR);
    if(outputFd == -1){
        perror("canot open the file for write"); exit(1); }

    while((numIn = read (inputFd, buffer, BUF_SIZE)) > 0){
        numOut = write (outputFd, buffer, (ssize_t) numIn);
        if(numIn != numOut){ perror("numIn != numOut"); exit(1); }
    }
    close (inputFd); close (outputFd);
    return (EXIT_SUCCESS);
}
```

一堆的#include <xxx.h>

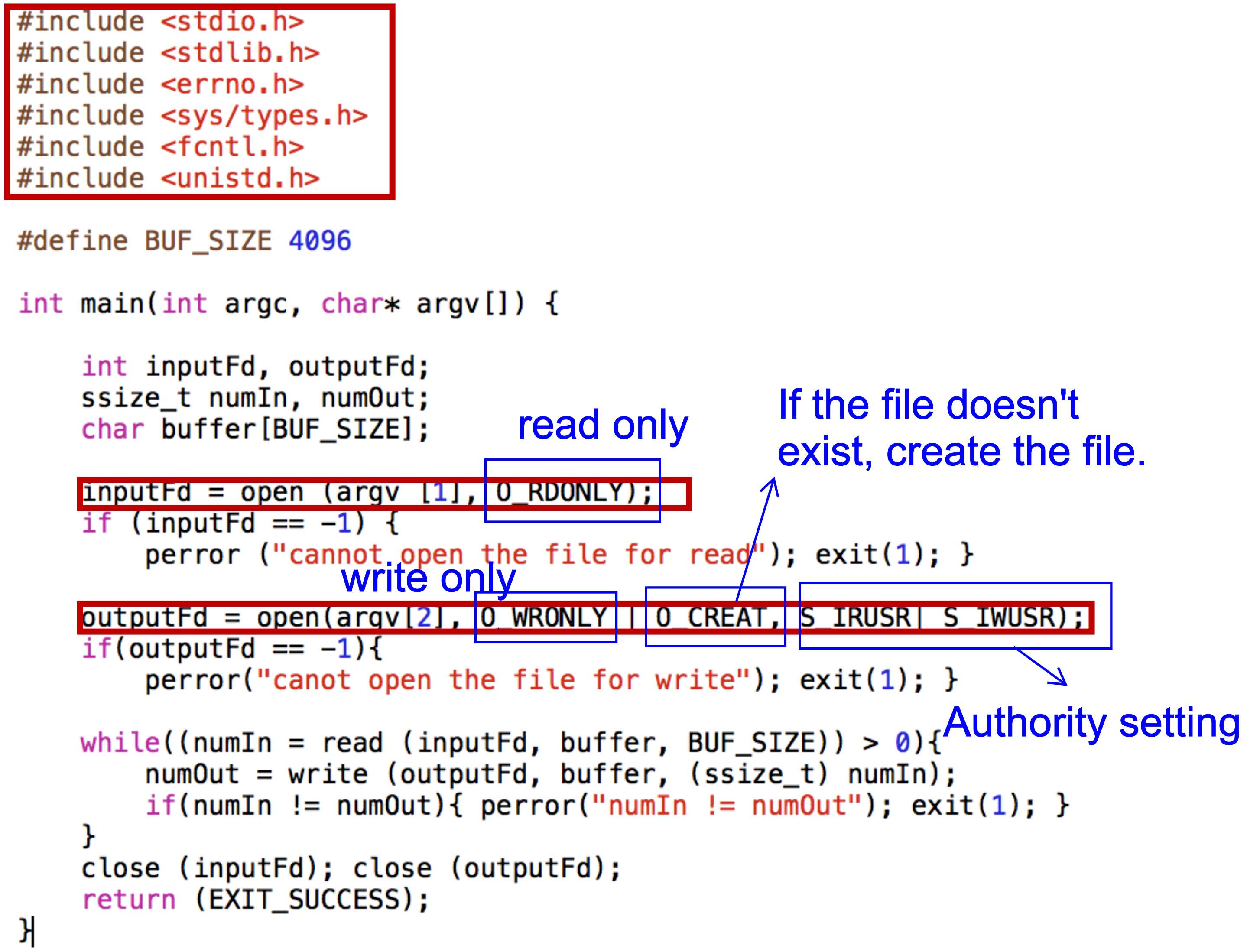
問題

- 記得函數的名稱就好
- 如果忘記或者不知道include某個.h檔案，編譯器會告訴你某函數未定義
- 針對該函數使用man查詢他需要include哪些

舉例

```
$man perror
NAME
       perror - print a system
error message

SYNOPSIS
       #include <stdio.h>

       void perror(const char *s);
```

mycp.c



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

#define BUF_SIZE 4096

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

    int inputFd, outputFd;
    ssize_t numIn, numOut;
    char buffer[BUF_SIZE];

    inputFd = open (argv [1], O_RDONLY);
    if (inputFd == -1) {
        perror ("cannot open the file for read"); exit(1); }

    outputFd = open(argv[2], O_WRONLY | O_CREAT, S_IRUSR| S_IWUSR);
    if(outputFd == -1){
        perror("canot open the file for write"); exit(1); }

    while((numIn = read (inputFd, buffer, BUF_SIZE)) > 0){
        numOut = write (outputFd, buffer, (ssize_t) numIn);
        if(numIn != numOut){ perror("numIn != numOut"); exit(1); }
    }
    close (inputFd); close (outputFd);
    return (EXIT_SUCCESS);
}
```

0, 2, 3, 4 have been used by OS, then
1. open -> 1; open -> 5

2, open -> 1; close (1); open -> 1

open

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

- open的傳回值是file descriptor（檔案描述子），在系統中從0開始編號
- 如果前面的號碼有缺號，open會優先使用最小的號碼當file descriptor
  - 如：系統已經使用了0, 2, 3, 4，當使用open再開啟一個檔案時，file descriptor會是「1」
- 通常0是stdin，1是stdout，2是stderr

stdout: OS 不會馬上印出，可優化
stderr: OS 會馬上印出

- 一個行程能夠開啟的檔案是有限的
  - 可以使用getrlimit()的RLIMIT_FSIZE查看
- 當回傳值為-1代表發生了錯誤，例如：超出RLIMIT_FSIZE

open

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

- 當flags設定為O_CREAT時mode的意義如下
  - owner, group, others的權限
  - set-user-ID、 set-group-ID及sticky bit
  - 介紹檔案系統會再介紹權限的相關意義

open()

```
  mycp file1 file2
argv [0]   [1]   [2]
```

為了讀

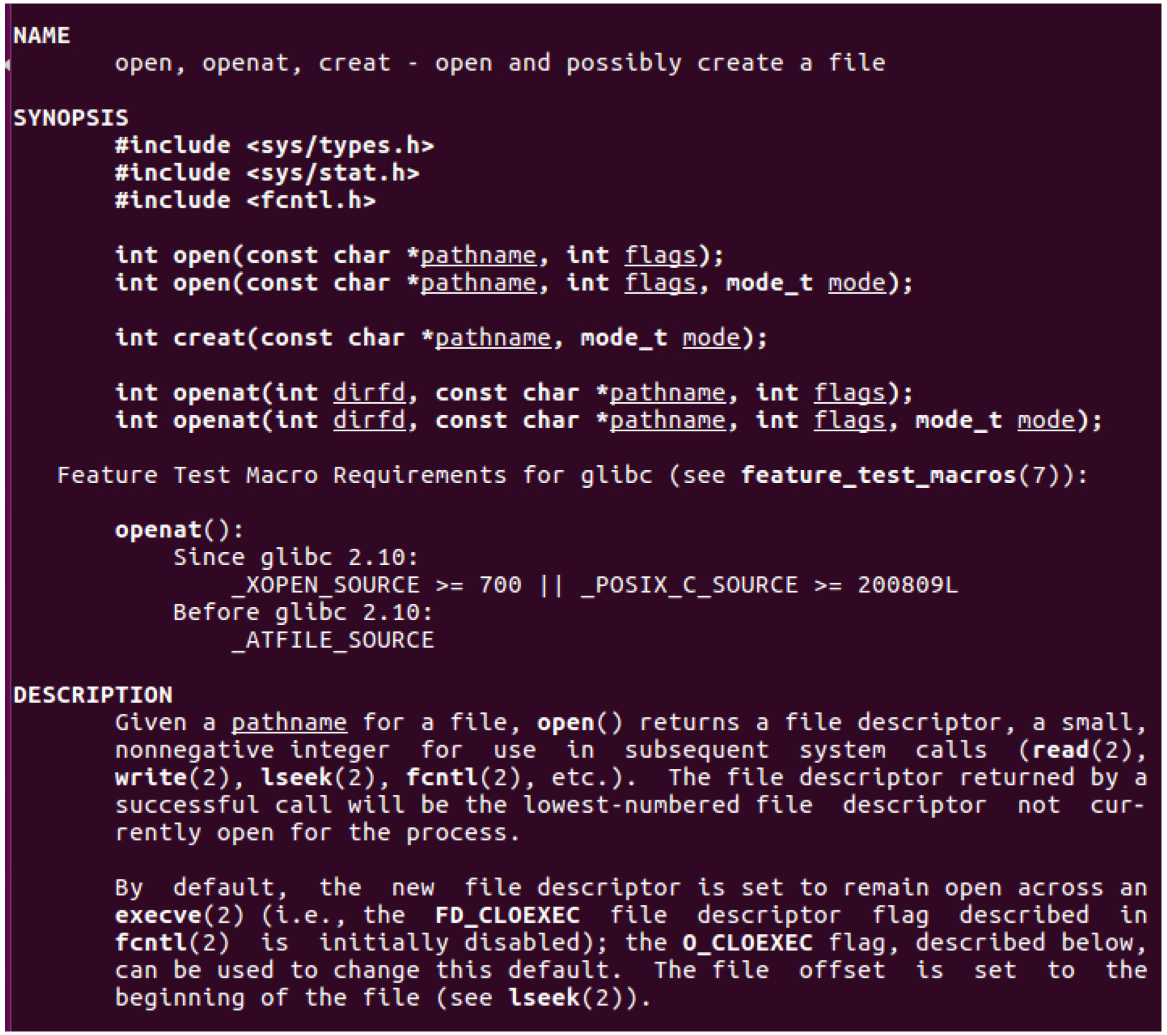
- open (argv [1], O_RDONLY);
- 第一個參數是"路徑名"
- 第二個參數告訴OS開啟這個檔案的目的是「只讀取」

為了寫

- open(argv[2], O_WRONLY | O_CREAT, S_IRUSR| S_IWUSR);
- 第一個參數是"路徑名"
- 第二個參數告訴OS這個檔案只用來寫入（O_WRONLY），如果檔案不存在，就建立檔案（O_CREAT）
- 第三個參數代表新建立的檔案的讀寫屬性（owner可讀寫）

自學open

- man 2 open
- 2代表系統裡面的第二本書，system call



```
NAME
       open, openat, creat - open and possibly create a file

SYNOPSIS
       #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);

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

       int openat(int dirfd, const char *pathname, int flags);
       int openat(int dirfd, const char *pathname, int flags, mode_t mode);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       openat():
           Since glibc 2.10:
               _XOPEN_SOURCE >= 700 || _POSIX_C_SOURCE >= 200809L
           Before glibc 2.10:
               _ATFILE_SOURCE

DESCRIPTION
       Given a pathname for a file, open() returns a file descriptor, a small,
       nonnegative integer  for  use  in  subsequent  system  calls  (read(2),
       write(2), lseek(2), fcntl(2), etc.).  The file descriptor returned by a
       successful call will be the lowest-numbered file  descriptor  not  cur-
       rently open for the process.

       By  default,  the  new  file descriptor is set to remain open across an
       execve(2) (i.e., the  FD_CLOEXEC  file  descriptor  flag  described  in
       fcntl(2)  is  initially disabled); the O_CLOEXEC flag, described below,
       can be used to change this default.  The file  offset  is  set  to  the
       beginning of the file (see lseek(2)).
```

open重要參數

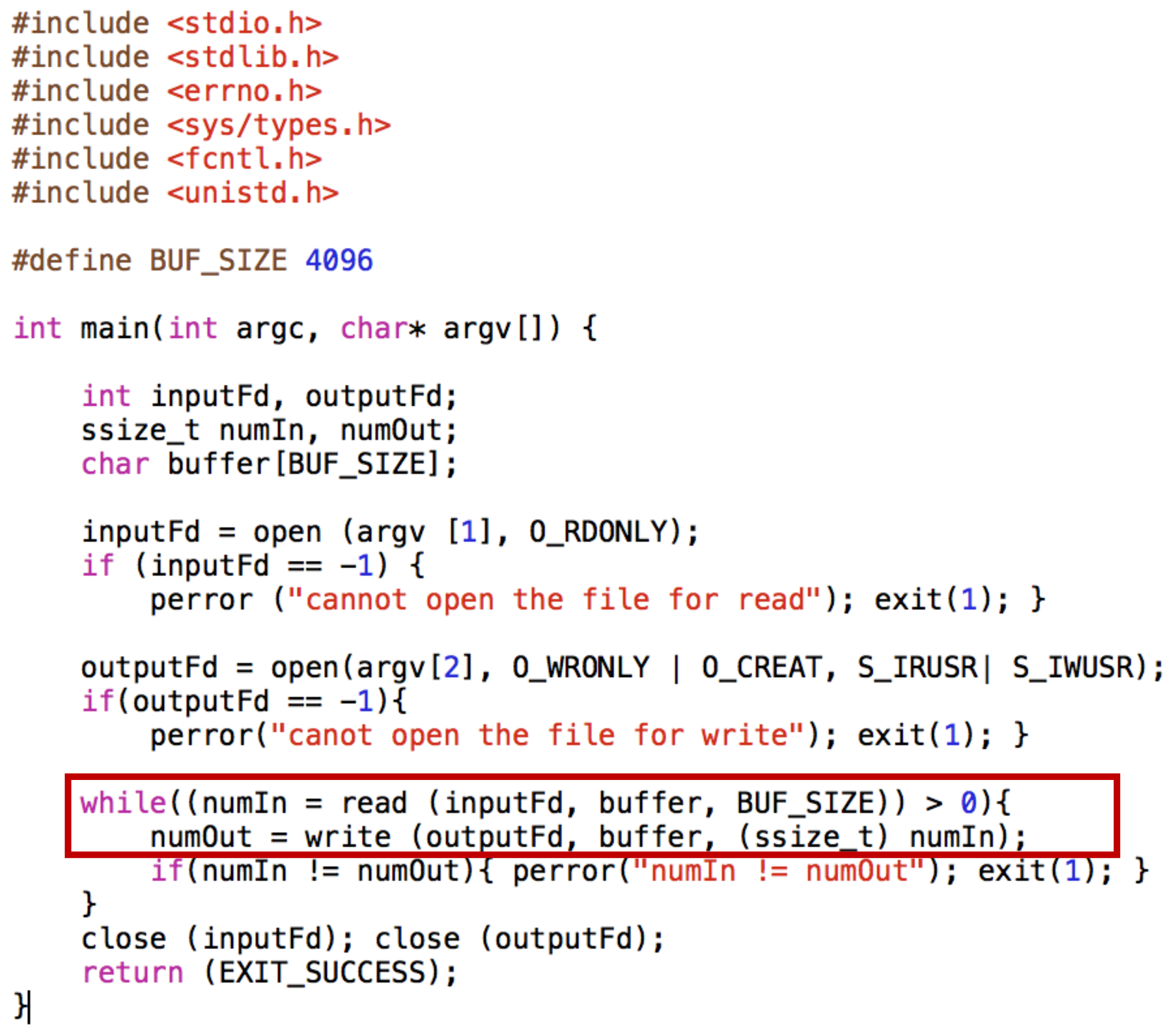
int open(const char *pathname, int flags);

- O_APPEND
  - 每次都會將資料加到檔案的最尾巴，就算是多個行程同時寫入，也能保證原子性（完整性）的加到最尾巴
- O_TRUNC
  - 將檔案大小歸為零，我們在設計存檔功能時，通常需要加上這個參數才可以保證不會有舊資料
- O_CLOEXEC
  - 使用execve時自動關閉檔案（execve後面會介紹）
  - 避免另外一個程序存取原程序所開啟的檔案

自我學習

- 先打開檔案，讀取後再儲存
  - int truncate(const char *path, off_t length);
  - int ftruncate(int fd, off_t length);

mycp.c



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

#define BUF_SIZE 4096

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

    int inputFd, outputFd;
    ssize_t numIn, numOut;
    char buffer[BUF_SIZE];

    inputFd = open (argv [1], O_RDONLY);
    if (inputFd == -1) {
        perror ("cannot open the file for read"); exit(1); }

    outputFd = open(argv[2], O_WRONLY | O_CREAT, S_IRUSR| S_IWUSR);
    if(outputFd == -1){
        perror("canot open the file for write"); exit(1); }

    while((numIn = read (inputFd, buffer, BUF_SIZE)) > 0){
        numOut = write (outputFd, buffer, (ssize_t) numIn);
        if(numIn != numOut){ perror("numIn != numOut"); exit(1); }
    }
    close (inputFd); close (outputFd);
    return (EXIT_SUCCESS);
}
```

read()

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

- 會從fd所代表的檔案讀取「最多」count個byte到指標buf所指向的記憶體
- 當回傳值大於1，代表讀取了多少個byte
- 回傳值等於0代表讀到了EOF（檔案結尾）
- 回傳值-1，代表讀取發生了錯誤

write()

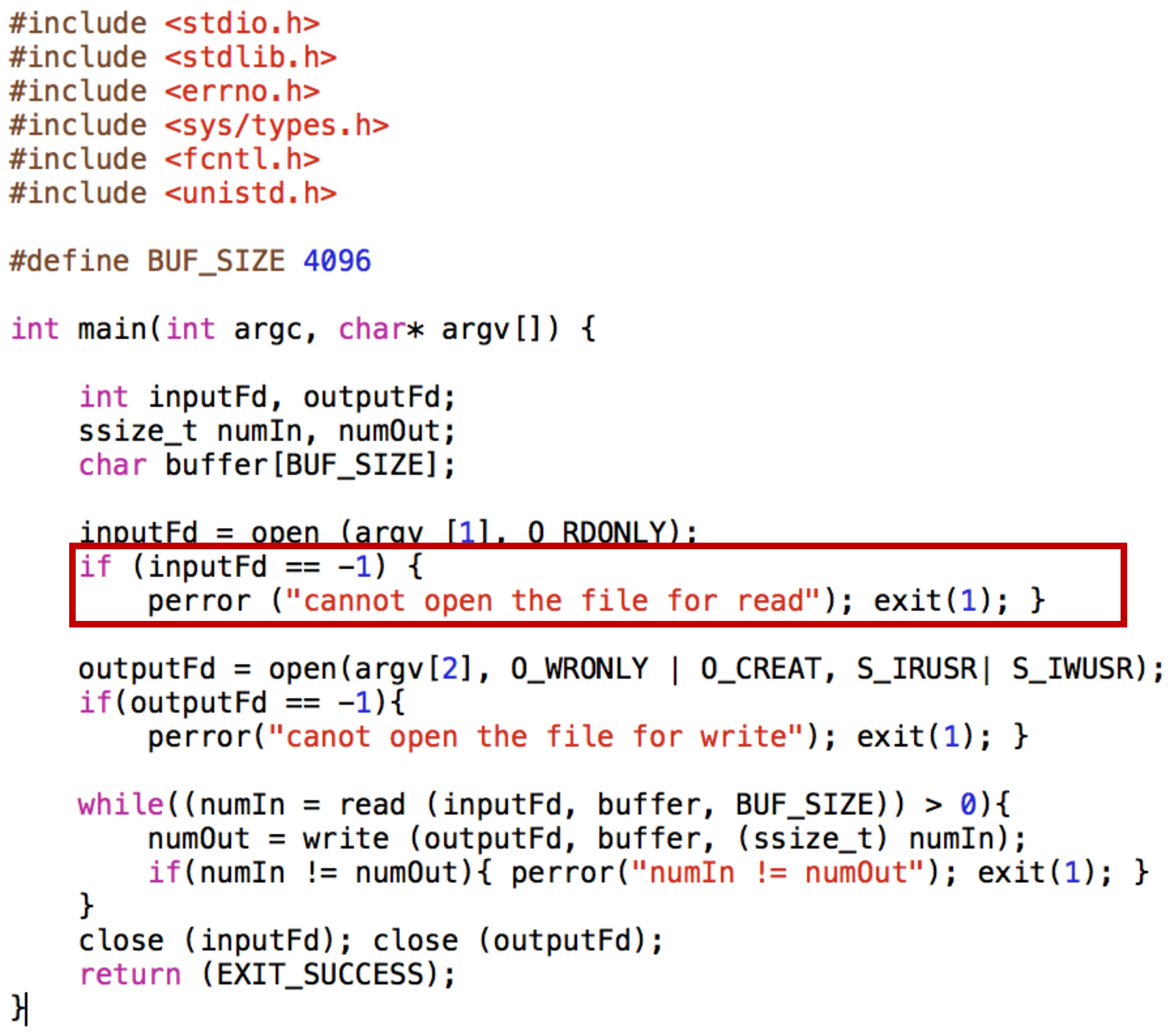
ssize_t: 有正負；size_t: 非負

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

- 將buf指向的資料共count個byte，寫入fd所代表的檔案
- 傳回值代表總共寫入了多少個byte
- 當傳回值為-1，代表發生了錯誤

mycp.c

perror 會在 programmer 寫的字串
後面再加上 OS 的詳細錯誤資訊



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

#define BUF_SIZE 4096

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

    int inputFd, outputFd;
    ssize_t numIn, numOut;
    char buffer[BUF_SIZE];

    inputFd = open (argv [1], O_RDONLY);
    if (inputFd == -1) {
        perror ("cannot open the file for read"); exit(1); }

    outputFd = open(argv[2], O_WRONLY | O_CREAT, S_IRUSR| S_IWUSR);
    if(outputFd == -1){
        perror("canot open the file for write"); exit(1); }

    while((numIn = read (inputFd, buffer, BUF_SIZE)) > 0){
        numOut = write (outputFd, buffer, (ssize_t) numIn);
        if(numIn != numOut){ perror("numIn != numOut"); exit(1); }
    }
    close (inputFd); close (outputFd);
    return (EXIT_SUCCESS);
}
```

perror

```
void perror(const char *s);
```

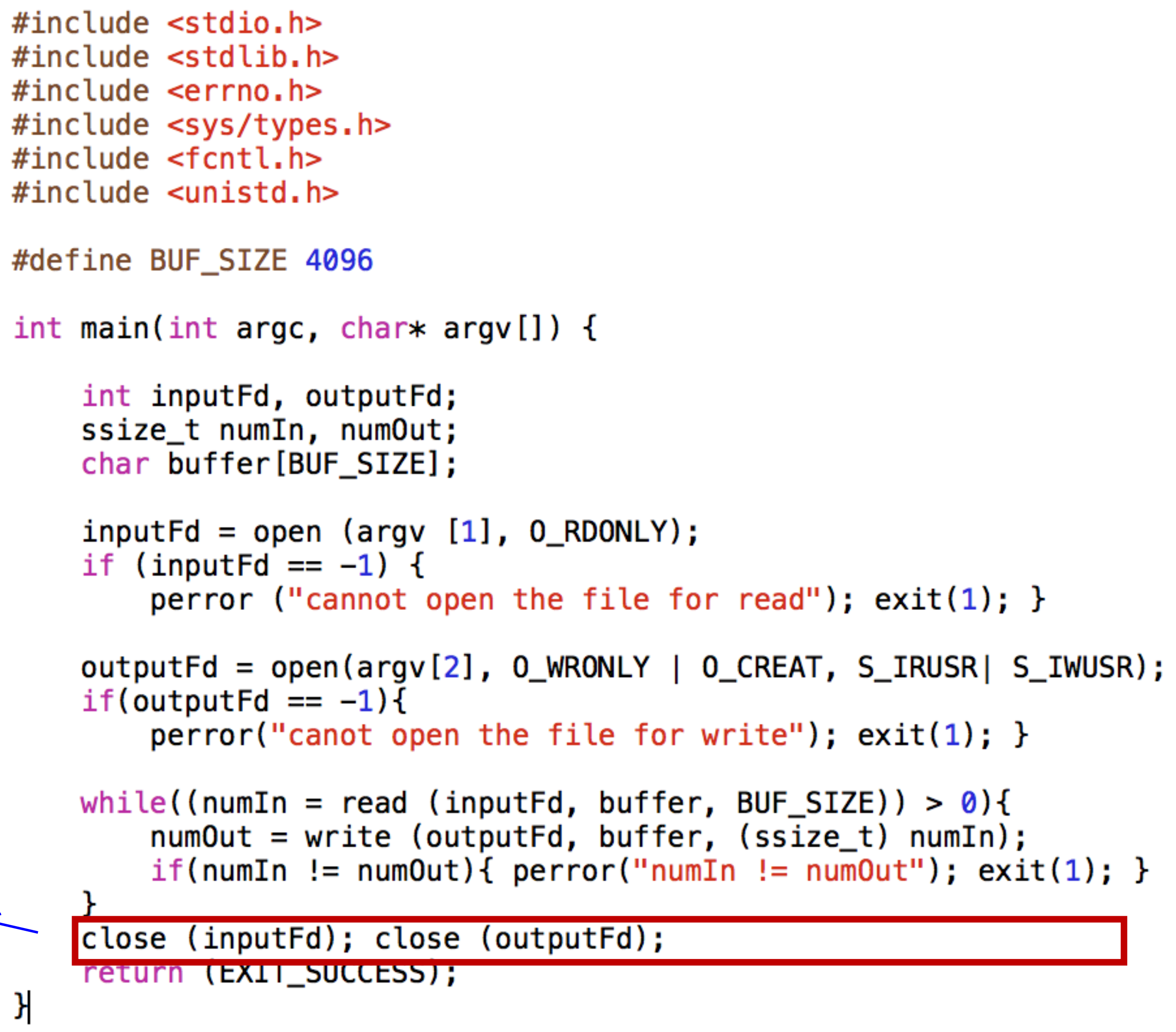
- 依照1. 依照errno印出訊息 2. 字串S
- 假設errno是1，perror(“the error is”)會印出「the error is: Operation not permitted」

什麼是errno

error number
一個應用程式只有一個 errno

- errno是系統內的錯誤訊息代碼
- 如果呼叫一個C函數時發生了錯誤，則errno會被設定為該錯誤所代表的號碼
- 如果呼叫一個C函數並且未發生任何錯誤，errno無意義
- 所有errno對應的錯誤訊息在sys_errlist

mycp.c



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

#define BUF_SIZE 4096

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

    int inputFd, outputFd;
    ssize_t numIn, numOut;
    char buffer[BUF_SIZE];

    inputFd = open (argv [1], O_RDONLY);
    if (inputFd == -1) {
        perror ("cannot open the file for read"); exit(1); }

    outputFd = open(argv[2], O_WRONLY | O_CREAT, S_IRUSR| S_IWUSR);
    if(outputFd == -1){
        perror("canot open the file for write"); exit(1); }

    while((numIn = read (inputFd, buffer, BUF_SIZE)) > 0){
        numOut = write (outputFd, buffer, (ssize_t) numIn);
        if(numIn != numOut){ perror("numIn != numOut"); exit(1); }
    }
    close (inputFd); close (outputFd);
    return (EXIT_SUCCESS);
}
```

1. 釋放 OS 資源
2. OS 會把資料清空寫出去

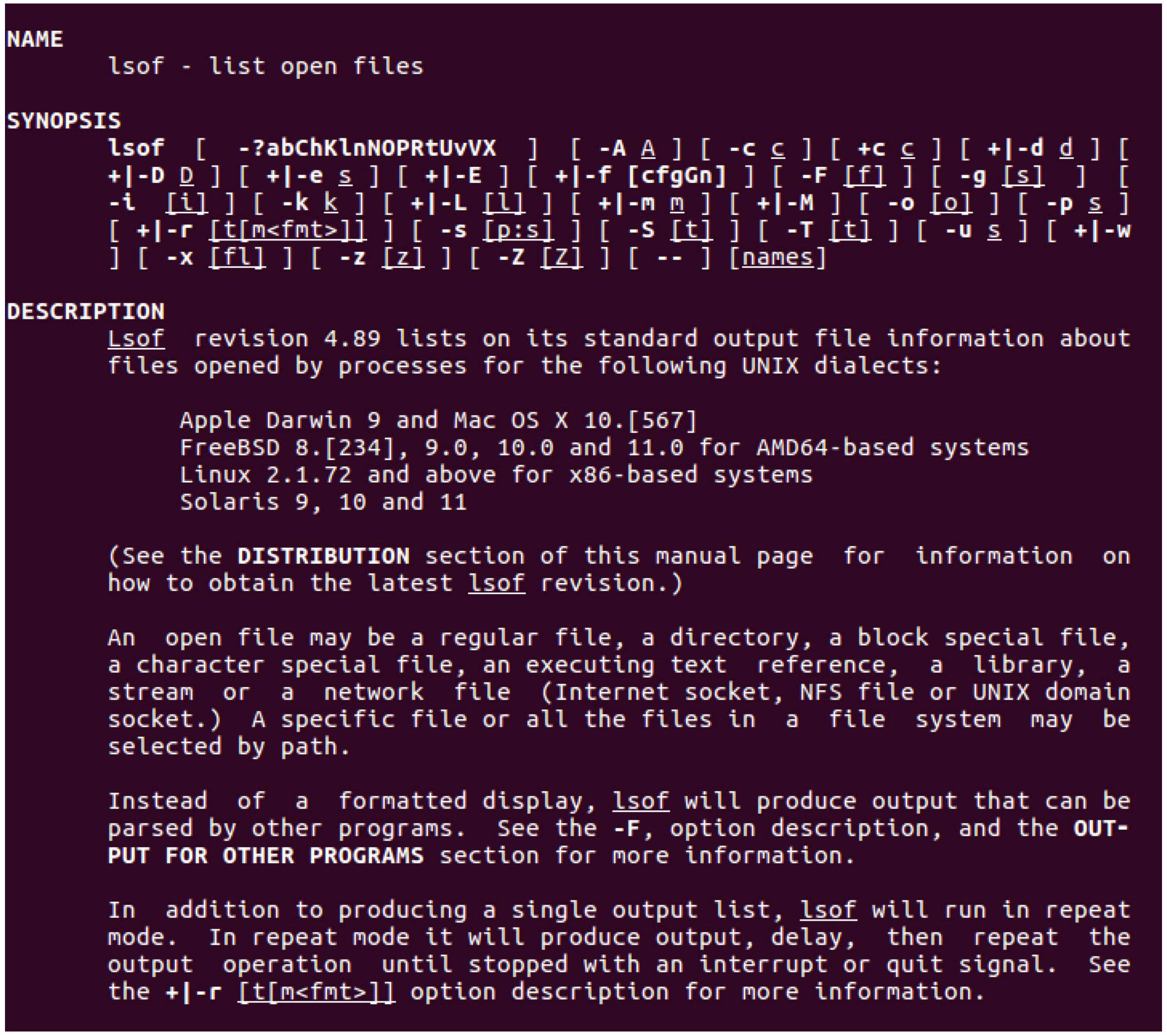
close()

```
int close(int fd);
```

- 使用完一個檔案，使用close告訴作業系統使用完畢
- 作業系統會依照當時的狀況（最後一個存取該檔案的行程），決定是否釋放相關資源
- 成功回傳0，失敗回傳-1

如果忘記close()

- 程式執行結束時，作業系統會自動幫忙關閉檔案
- 但如果程式會執行很久呢？ daemon?
- 使用lsof查看到底哪些檔案還沒關閉
  - 重要參數-p PROCESS_ID



```
NAME
       lsof - list open files

SYNOPSIS
       lsof  [  -?abChKlnNOPRtUvVX  ]  [ -A A ] [ -c c ] [ +c c ] [ +|-d d ] [
       +|-D D ] [ +|-e s ] [ +|-E ] [ +|-f [cfgGn] ] [ -F [f] ] [ -g [s]  ]  [
       -i  [i] ] [ -k k ] [ +|-L [l] ] [ +|-m m ] [ +|-M ] [ -o [o] ] [ -p s ]
       [ +|-r [t[m<fmt>]] ] [ -s [p:s] ] [ -S [t] ] [ -T [t] ] [ -u s ] [ +|-w
       ] [ -x [fl] ] [ -z [z] ] [ -Z [Z] ] [ -- ] [names]

DESCRIPTION
       Lsof  revision 4.89 lists on its standard output file information about
       files opened by processes for the following UNIX dialects:

            Apple Darwin 9 and Mac OS X 10.[567]
            FreeBSD 8.[234], 9.0, 10.0 and 11.0 for AMD64-based systems
            Linux 2.1.72 and above for x86-based systems
            Solaris 9, 10 and 11

       (See the DISTRIBUTION section of this manual page  for  information  on
       how to obtain the latest lsof revision.)

       An  open file may be a regular file, a directory, a block special file,
       a character special file, an executing text  reference,  a  library,  a
       stream  or  a  network  file  (Internet socket, NFS file or UNIX domain
       socket.)  A specific file or all the files in  a  file  system  may  be
       selected by path.

       Instead  of  a  formatted display, lsof will produce output that can be
       parsed by other programs.  See the -F, option description, and the OUT-
       PUT FOR OTHER PROGRAMS section for more information.

       In  addition to producing a single output list, lsof will run in repeat
       mode.  In repeat mode it will produce output, delay,  then  repeat  the
       output  operation  until stopped with an interrupt or quit signal.  See
       the +|-r [t[m<fmt>]] option description for more information.
```

自我學習

- 原子性的讀取和寫入
  - ssize_t pread(int fd, void *buf, size_t count, off_t offset);
  - ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset);

小節

- 初步了解open、read、write，並用這幾個函數設計了簡單的cp
- open可以接很多參數，同學們應該主動學習

lseek & file holes

hole.c

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

int main() {
    int fd;
    fd = open("./myHole", O_RDWR| O_CREAT, S_IRUSR| S_IWUSR);
    if (fd <0)
        perror("open");
    write(fd, "1", sizeof("1"));
    lseek(fd, 100000, SEEK_SET);
    write(fd, "2", sizeof("2"));
    lseek(fd, 100000, SEEK_CUR);
    write(fd, "3", sizeof("3"));
    close(fd);
    return 0;
}
```

lseek()

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

- 將檔案fd的檔案指標移動到從whence起算，偏移offset的位置
- 傳統上UNIX支援的whence有三種選擇
  - SEEK_SET：絕對位置
  - SEEK_CUR：從現在位置起算
  - SEEK_END：從結束位置起算
- 傳回值為從檔案開始的偏移值
- 錯誤時：
  - 在執行lseek前先將errno設定為0
  - 檢查傳回值是否等於-1「並且」errno不為0

hole.c

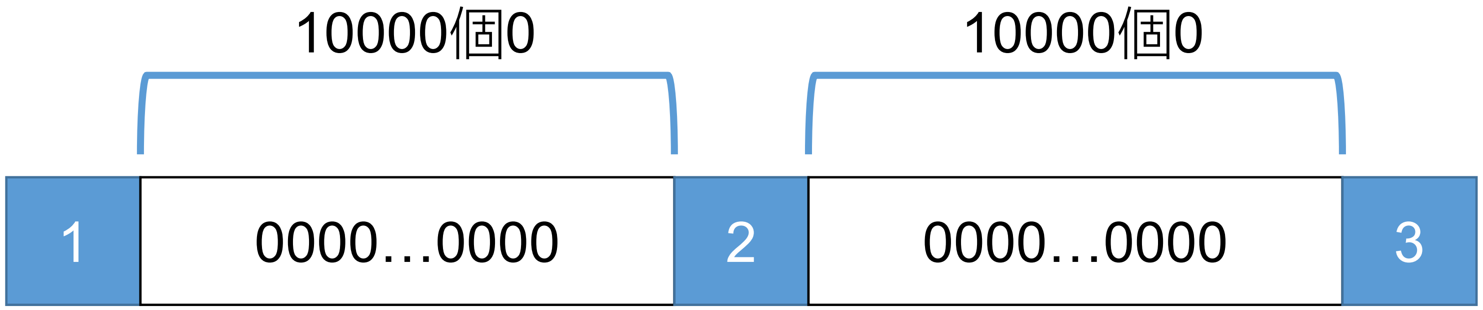
- 因此hole.c會產生一個名為myHole的檔案，在開始位置寫入1，往後移動10000 byte在寫入2，往後移動10000 byte在寫入3

```
$ls myHole -lhs
12K -rw------- 1 shiwulo shiwulo
196K Jan 13 04:24 myHole
/*檔案大小為196K，佔據磁碟空間12K*/
```

使用mycp複製myhole

```
$ ./mycp myHole myHole2
$ ls myH* -lhs
 12K -rw------- 1 shiwulo shiwulo 196K Jan 13 04:24 myHole
196K -rw------- 1 shiwulo shiwulo 196K Jan 13 04:31 myHole2
/*檔案大小都是196K，但是myHole2佔據磁碟空間196K而非12K*/
$cmp myHole myHole2
/*使用cmp比較二者無差異*/
```

myHole內部構造



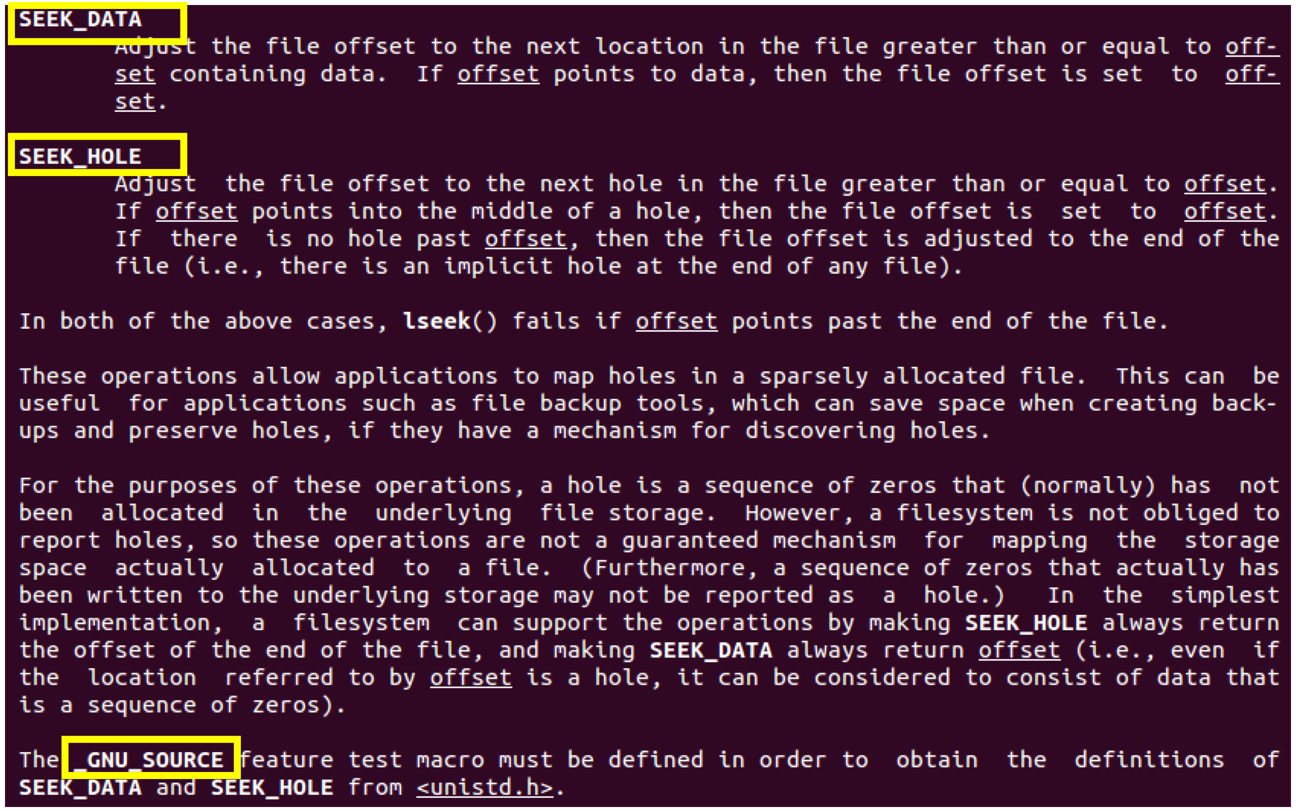
進階版的mycp.c，mycp2.c（第一部分）

```
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <unistd.h>

#define BUF_SIZE 4096

int main(int argc, char* argv[]) {
    int inputFd, outputFd;
    ssize_t numIn, numOut;
    char buffer[BUF_SIZE];
    off_t begin=0, end=0;
    int fileSize, blockSize, pos=0;

    inputFd = open (argv [1], O_RDONLY);
    outputFd = open(argv[2], O_WRONLY | O_CREAT, S_IRUSR| S_IWUSR);
    ftruncate(outputFd, 0);

    fileSize = lseek(inputFd, 0, SEEK_END);
    lseek(inputFd, 0, SEEK_SET);
```

一開始要宣告
_GNU_SOURCE才可以使用
進階版的lseek()

要改成long long

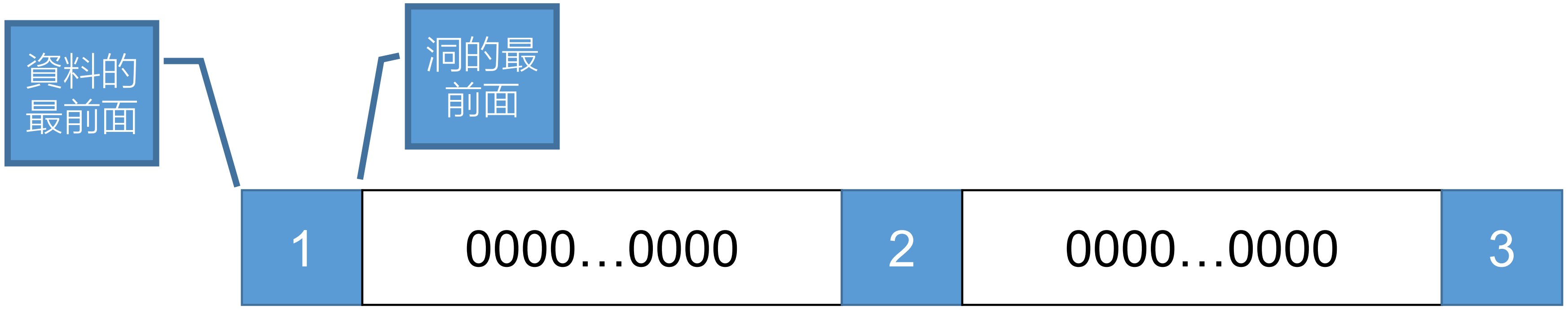
man lseek



```
SEEK_DATA
       Adjust the file offset to the next location in the file greater than or equal to off‐
       set containing data.  If offset points to data, then the file offset is set  to  off‐
       set.

SEEK_HOLE
       Adjust  the file offset to the next hole in the file greater than or equal to offset.
       If offset points into the middle of a hole, then the file offset is  set  to  offset.
       If  there  is no hole past offset, then the file offset is adjusted to the end of the
       file (i.e., there is an implicit hole at the end of any file).

In both of the above cases, lseek() fails if offset points past the end of the file.

These operations allow applications to map holes in a sparsely allocated file.  This can  be
useful  for applications such as file backup tools, which can save space when creating back‐
ups and preserve holes, if they have a mechanism for discovering holes.

For the purposes of these operations, a hole is a sequence of zeros that (normally) has  not
been  allocated  in  the  underlying  file storage.  However, a filesystem is not obliged to
report holes, so these operations are not a guaranteed mechanism  for  mapping  the  storage
space  actually  allocated  to  a file.  (Furthermore, a sequence of zeros that actually has
been written to the underlying storage may not be reported as  a  hole.)   In  the  simplest
implementation,  a  filesystem  can support the operations by making SEEK_HOLE always return
the offset of the end of the file, and making SEEK_DATA always return offset (i.e., even  if
the  location  referred to by offset is a hole, it can be considered to consist of data that
is a sequence of zeros).

The _GNU_SOURCE feature test macro must be defined in order to  obtain  the  definitions  of
SEEK_DATA and SEEK_HOLE from <unistd.h>.
```

SEEK_HOLE & SEEK_DATA

- 在新版的UNIX提供這二個新的選項，但必須手動打開，即 #define _GNU_SOURCE
- SEEK_HOLE移動到一個洞的最前面
- SEEK_DATA移動到一個資料的最前面



進階版的mycp.c，mycp2.c（第二部分）

```
while (1) {
    pos = lseek(inputFd, pos, SEEK_DATA);
    begin = pos;
    pos = lseek(inputFd, pos, SEEK_HOLE);
    end = pos;
    blockSize=end-begin;
    lseek(inputFd, begin, SEEK_SET);
    lseek(outputFd, begin, SEEK_SET);
    while((numIn = read (inputFd, buffer, BUF_SIZE)) > 0) {
        numOut = write (outputFd, buffer, (ssize_t) numIn);
        if (numIn != numOut) perror("numIn != numOut");
        blockSize-=numIn;
        if (blockSize == 0) break;
    }
    if (lseek(outputFd, 0, SEEK_CUR) == fileSize) break;
}
close (inputFd);
close (outputFd);

return (EXIT_SUCCESS);
}
```

取得每個資料區段的位置及大小

移動到該區段的開頭位置

進行該區段的複製

問題

- cp2假設一開始是資料
- 上述程式如果「洞」出現在一開始的位置會發生什麼樣的情況?
  - begin > 0， end = 0。因此 blockSize = end – begin < 0;

結果

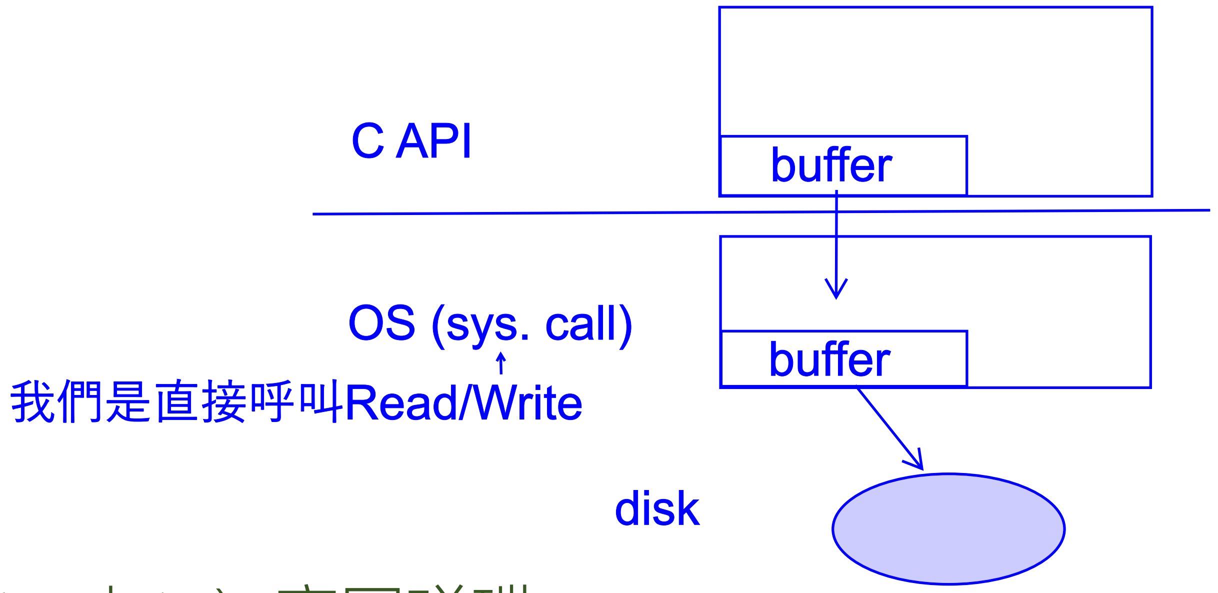
```
$./mycp myHole myHole2
$ ./mycp2 myHole myHole3
$ ls myH* -lhs
 12K -rw------- 1 shiwulo shiwulo 196K Jan 13 04:24 myHole
196K -rw------- 1 shiwulo shiwulo 196K Jan 13 05:08 myHole2
 12K -rw------- 1 shiwulo shiwulo 196K Jan 13 05:09 myHole3
```

小節

- 對於Linux及大多數的UNIX而言，「洞」並不會佔據空間
- 讀取「洞」，裡面的值都是0，因此第一個版本的cp會讓「洞」佔據空間
- 使用Linux的系統擴充，SEEK_DATA及SEEK_DATA可以找出洞，複製時可以跳過這些洞

確保寫入
sync & fsync & fdatasync

三個類似的函數



- void sync(void);
  - 將所有的資料（包含meta-data）寫回磁碟
- int fsync(int fd);
  - 將fd代表的檔案的所有的資料（包含meta-data）寫回磁碟
- int fdatasync(int fd);
  - 將fd代表的檔案的所有的資料（「不」包含meta-data）寫回磁碟

什麼是meta-data 檔案屬性

- 例如：檔案的修改日期、檔案的大小、檔案的瀏覽日期等等，這些外加的資料都「附屬」於該檔案，因此稱之為meta-data

sync.c

```c
int main() {
        int fd;
        int num;
        fd = open("./hello1",O_WRONLY | O_CREAT, 0644);
        for(num=0; num <=100000; num++) {
                write(fd, "1234", sizeof("1234"));
                fsync(fd);
                if (num%10000==1) {
                        write(1, "*", sizeof("*"));
                        fsync(1);
                }
        }
        return 0;
}
```

datasync.c

```c
int main() {
        int fd;
        int num;
        fd = open("./hello3",O_WRONLY | O_CREAT, 0644);
        for(num=0; num <=100000; num++) {
                write(fd, "1234", sizeof("1234"));
                fdatasync(fd);
                if (num%10000==1) {
                        write(1, "*", sizeof("*"));
                        fsync(1);
                }
        }
        return 0;
}
```

nosync.c

```c
int main() {
        int fd;
        int num;
        fd = open("./hello2",O_WRONLY | O_CREAT, 0644);
        for(num=0; num <=100000; num++) {
                write(fd, "1234", sizeof("1234"));

                if (num%10000==1) {
                        write(1, "*", sizeof("*"));
                        fsync(1);
                }
        }
        return 0;
}
```

sync.c的執行結果

```
$ time ./sync
**********
real    0m21.215s
user    0m0.136s
sys     0m5.272s
```

比較

strace -c ./sync
可以看到呼叫幾次 system call
ltrace -c ./sync
可以看到呼叫幾次 C API

	sync	fdatasync	no sync
real	0m21.215s	0m17.545s	0m0.076s
user	0m0.136s	0m0.048s	0m0.004s
sys	0m5.272s	0m3.980s	0m0.068s

鎖定檔案flock – 良心鎖 advisory locking

鎖的種類

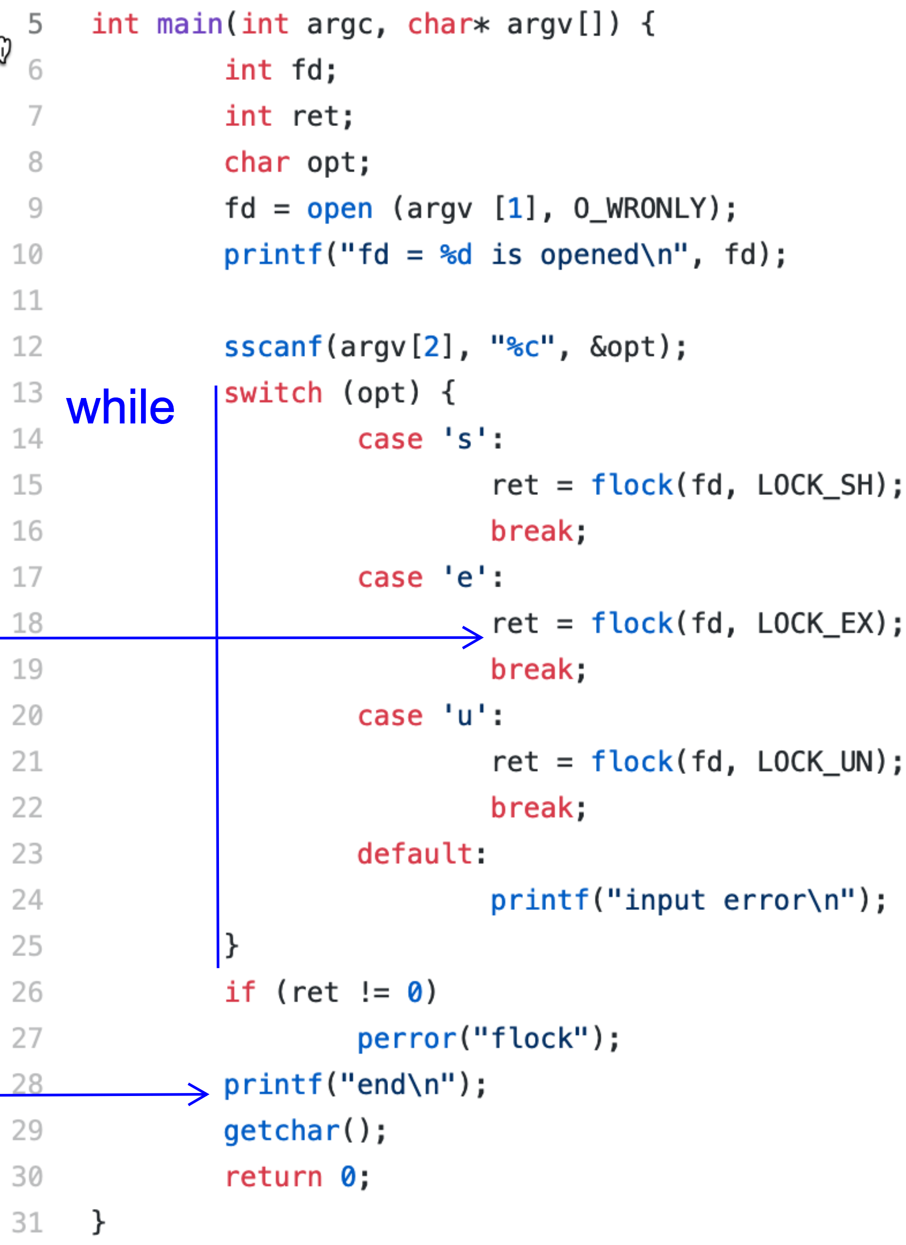
良心鎖

強制鎖

圖片來源：
http://tinyurl.com/y5byr6t2
https://mall.pchome.com.tw/store/QCAI5Z

良心鎖

flock.c



```c
int main(int argc, char* argv[]) {
        int fd;
        int ret;
        char opt;
        fd = open (argv [1], O_WRONLY);
        printf("fd = %d is opened\n", fd);

        sscanf(argv[2], "%c", &opt);
        switch (opt) {
                case 's':
                        ret = flock(fd, LOCK_SH);
                        break;
                case 'e':
                        ret = flock(fd, LOCK_EX);
                        break;
                case 'u':
                        ret = flock(fd, LOCK_UN);
                        break;
                default:
                        printf("input error\n");
        }
        if (ret != 0)
                perror("flock");
        printf("end\n");
        getchar();
        return 0;
}
```

while

share lock (RD)

exclusive lock (WR)

unlock

第二個程式無法 lock， 會停留在這裡等待

第一個程式成功取得權限，成功 lock， 會停在這裡

執行結果

先執行

```
$ ./lock myHole e
fd = 3 is opened
end
```

後執行

```
$ ./lock myHole e
fd = 3 is opened
/*被鎖住了，除非另外一個行程
unlock或者結束*/
```

flock()

```
int flock(int fd, int operation);
```

第一個三數是檔案描述子，operation可以接三個選項，分別是

- LOCK_SH：分享鎖，除了互斥鎖，可以多個人同時編譯
- LOCK_EX：互斥鎖，只可以這個行程進行編譯
- LOCK_UN：解開這個鎖

- 請注意，如果另外一個行程並未使用flock，那麼另一個行程就不需要遵照這些「鎖」

鎖定檔案flock – 強制鎖 mandatory locking

強制鎖（mandatory lock），預備動作

如果要每次開機都自動有這個動作
sudo vim /etc/fstab

- `sudo mount -oremount,mand /` 將根目錄重新 mount 一次

change mode　　set

- `chmod g+s system-programming.txt` ← 我們要鎖定的檔案

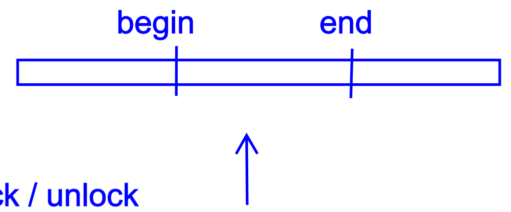
- `chmod g-x system-programming.txt`

execute

- Blocking（F_LOCK)
  - Flock 回傳的時候，就一定上鎖了
- Nonblocking（F_TLOCK)
  - Flock 回傳的時候，可能「目前有人已經鎖定檔案」>> Error
  - 另一種，順利上鎖

lockf.c

```c
int main(int argc, char* argv[]) {
        int fd;
        int ret;
        char opt;
        off_t begin, end;

        if (argc == 1) printf("lockf [file] [type(l/u)] [begin] [end]\n");

        fd = open (argv [1], O_WRONLY);
        printf("fd = %d is opened\n", fd);

        sscanf(argv[2], "%c", &opt);
        sscanf(argv[3], "%ld", &begin);
        sscanf(argv[3], "%ld", &end);

        switch (opt) {
                case 'l':
                        lseek(fd, begin, SEEK_SET);
                        ret = lockf(fd, F_LOCK, end - begin + 1);
                        break;
                case 'u':
                        lseek(fd, begin, SEEK_SET);
                        ret = lockf(fd, F_UNLCK, end - begin + 1);
                        break;
                default:
                        printf("input error\n");
        }
        if (ret != 0)
                perror("flock");
        printf("end\n");
        getchar();
        return 0;
}
```



lock / unlock

可以不用 lock 整個檔案

小結

```
./flock ./system-programming.txt e
vim ./system-programming.txt
```
vim 是可以修改檔案的，因為 vim 不遵守 flock 機制

- 當多個程式讀取檔案時，可以用flock上鎖，但先決條件是所有的程式在讀取之前都先使用flock
- 如果要用強制鎖需要有root的權限（因為mount指令只有root可以執行）
- 檔案的寫入會變更檔案的屬性，此外檔案內容的變化是否先暫存在記憶體（buffer）呢。「同步更新」的東西越多，速度越慢，但也越安全（例如系統突然斷電）

作業

- 使用強置鎖，設計底下二個程式
  - 第一個程式不斷的詢問使用者要對檔案做什麼樣的操作，lock或者unlock，lock或者是unlock在檔案的什麼區域
  - 因此可以透過第一個程式對檔案對檔案同時上好幾個鎖
  - 第二個程式不斷地詢問使用者要對檔案做什麼樣的操作，read或者write，要讀寫哪個地方？