



系統程式設計

Systems Programming

鄭卜壬教授
臺灣大學資訊工程系

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng ©2007
Department of Computer Science and Information Engineering
Graduate Institute of Multimedia and Networking, National Taiwan University



Contents

1. OS Concept & Intro. to UNIX
2. UNIX History, Standardization & Implementation
3. File I/O
4. Standard I/O Library
5. Files and Directories
6. System Data Files and Information
7. Environment of a Unix Process
8. Process Control
9. Signals
10. Inter-process Communication
11. Thread Programming
12. Networking



Example of Unbuffered I/O

```
#include <unistd.h>

int main(void)
{
    char buf[100];
    ssize_t n;
    while ( (n=read( STDOUT_FILENO, buf, 100 )) != 0 )
        write( STDOUT_FILENO, buf, n );
    return 0;
}
```

Unbuffered I/O

~~UNIX有兩種標準：ANSI C & POSIX~~

Copy standard input to standard output

- **UNIX Standardization** C是為了寫unix誕生的
 - ANSI C (stdio.h, stdlib.h, string.h, math.h, time.h, ...)
 - Provide portability of conforming C programs to a wide variety of OS's
 - POSIX (unistd.h, pwd.h, dirent.h, grp.h, fcntl.h, ...)
 - Define the application programming interface for software compatible with variants of OS's



Unbuffered I/O

(x) 沒有buffer的I/O
kernel 提供的function: system call
c-library 提供的function: function call

User Process



read()

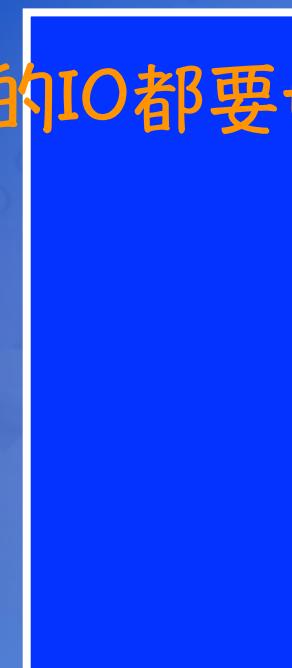
每一次的IO都要track in進去system

read()

:

read()

Kernel



Storage

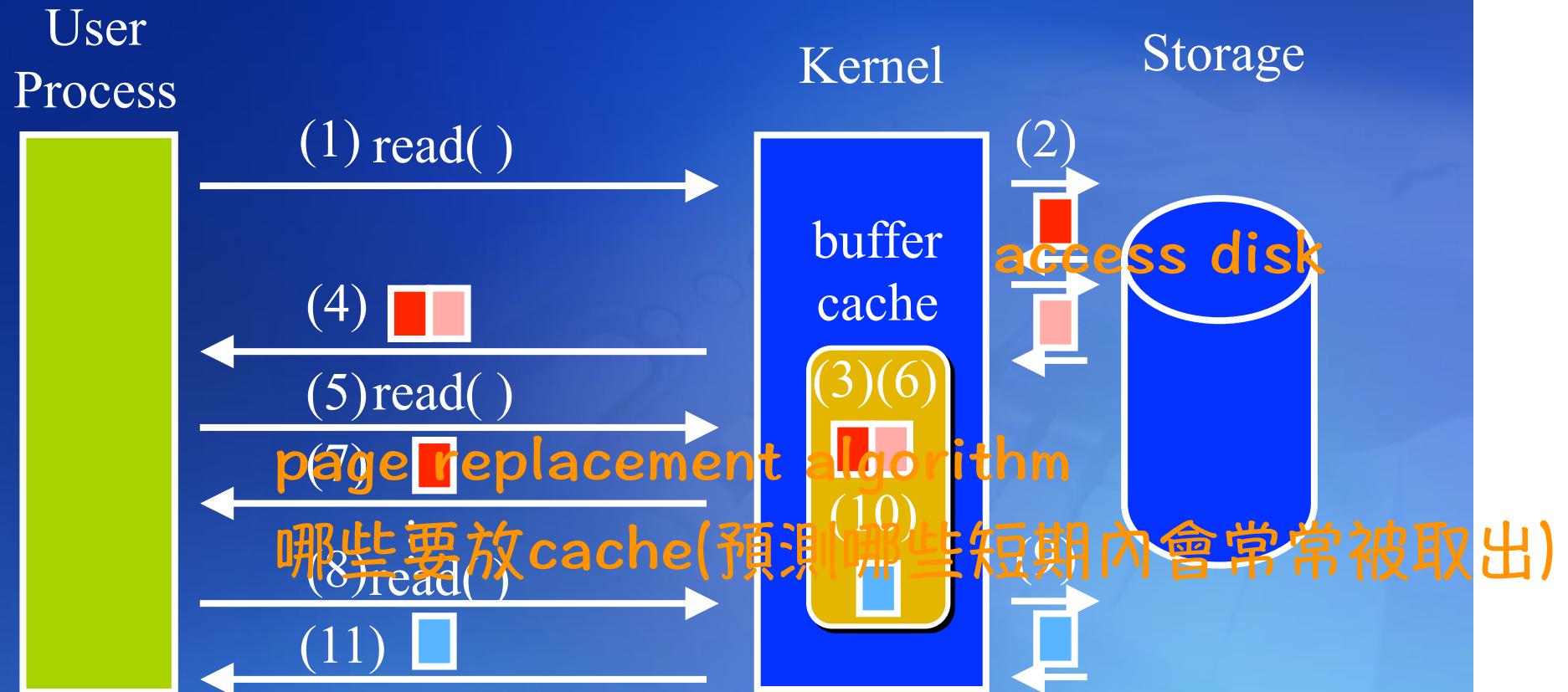
進去system



Each read() and write() invokes a system call
in the kernel (part of POSIX.1)!



Unbuffered I/O



Buffer cache: cache of recently used disk blocks
It's not necessary to trigger disk I/O for unbuffered I/O



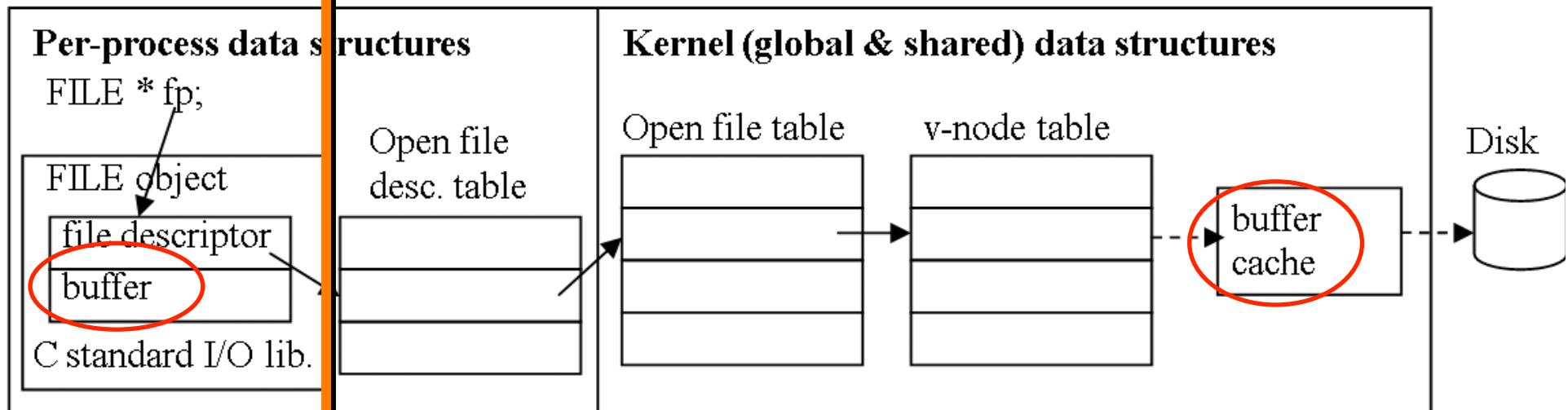
每一個process，都有這兩個DS

kernel What “Unbuffered” refer to?

pre-process

buffered I/O

unbuffered I/O (operate on file descriptors)



file pointer 指向 buffer I/O 的 pre-process DS 的 buffer

• Buffer in user's process is provided by the C standard library.

in user process
• Buffer cache is provided by the operating system

kernel mode

-> kernel space



File I/O (or Unbuffered I/O)

- **Unbuffered I/O**

- Popular functions: open, close, read, write, lseek, dup, fcntl, ioctl
- Each read() and write() invokes a system call!
- All data are processed as a series of bytes (no format)
- All disk I/O goes through the kernel's block buffers (also called the kernel's buffer cache)
- The term “unbuffered I/O” refers to the lack of automatic buffering in the user process



File I/O

c header: stdio.h
unix: unistd.h
limits.h

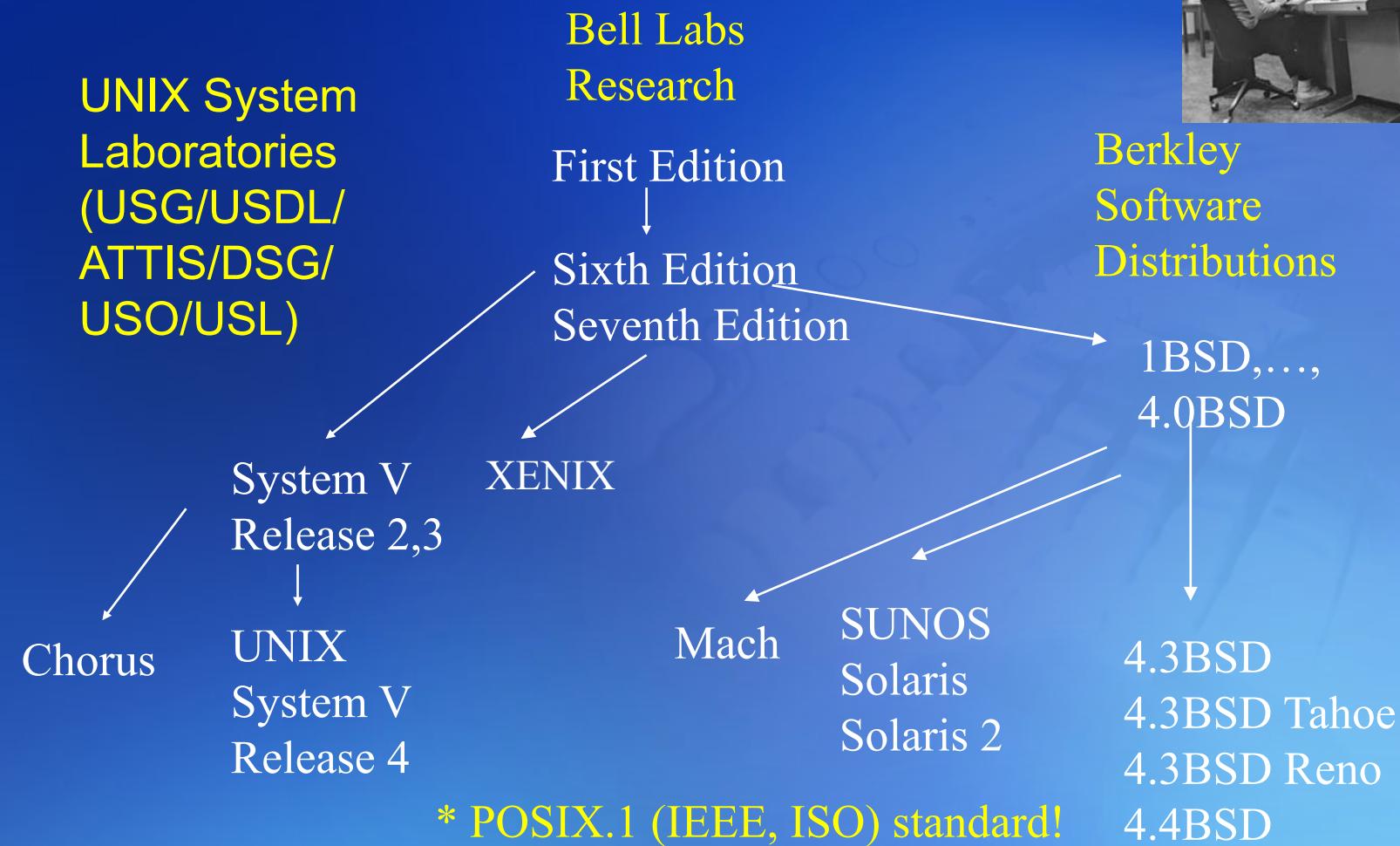
- **File Descriptor** 文件描述符

- Non-negative integer returned by open() or creat(): 0 .. OPEN_MAX (compiler-time limits)
 - Virtually un-bounded for SVR4 & 4.3+BSD
- Referenced by the kernel (next slide)
- Per-process base
- POSIX.1 – 0: STDIN_FILENO, 1: STDOUT_FILENO, 2: STDERR_FILENO
 - <unistd.h>
 - Convention employed by the Unix shells and applications



UNIX

Created by Ken Thompson & Dennis Ritchie
at Bell Laboratories in 1969.



D. Ritchie and K. Thompson, 『The *UNIX Time-Sharing System*』, CACM Vol.17, No.7, pp.365-375, 1974



File I/O - open, openat

`open`: 開檔，可以用絕對或相對路徑

Path name

Absolute path name (/xxx)

Relative path name (./xxx)

(relative to working directory)

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

```
int open(const char* pathname, int oflag, .../* mode_t mode */);
```

- **File/Path Name** 偷偷truncate掉name的後面char，
 - PATH_MAX, NAME_MAX 因為最多14個char
 - _POSIX_NO_TRUNC -> ENAMETOOLONG if error occurs ERNO (global var)
 - **O_RDONLY, O_WRONLY, O_RDWR, O_EXEC** (file access modes)
 - **O_CREAT, O_TRUNC, O_EXCL** (file creation)
 - **O_APPEND**, (append to the end of the file for each write)
 - **O_NONBLOCK** (non-blocking)
 - **O_DSYNC, O_RSYNC, O_SYNC** (data synchronization)

ex: `open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode)`



int openat(int dirfd, const char* pathname, int oflag, ...);

- **pathname**

- Absolute pathname -> openat() == open()

- Relative pathname

- dirfd == AT_FDCWD -> openat() == open()

- otherwise, relative to directory file descriptor *dirfd*

- **Time-of-check-to-time-of-use (TOCTTOU)**

- Any part of the path of a file could be changed in parallel to a call to *open()*

- openat()* is atomic and guarantee the opened file is located relative to the desired directory at 代表ATOMIC

```
int dirfd = open ("..", O_RDONLY);
int fd = openat( dirfd, "test", O_RDWR | O_CREAT, 0644 );
```

is equivalent to

```
int fd = openat( AT_FDCWD, "../test", O_RDWR | O_CREAT, 0644 );
```

O_RDONLY, O_WRONLY, O_RDWR

- Reasons to distinguish the access modes
 - Performance
 - Read ahead, buffer swapping, pointer synchronization
 - Integrity/Security/Privacy
 - Process may only have read permission for a file, i.e., /etc/passwd
 - Opening files without proper permission leads to integrity/ security/privacy issue.

pipeline: 不是第一個cmd處理完、再處理第二個cmd

因為cmd裡面有好幾個steps

-> 同步走好几个cmd



suspend = 不讓process佔CPU

Blocking vs. Synchronization

不希望CPU被浪費

content switch

non-block: 系統一直被煩，process不會被系統亂停掉

Block/non-block: the behavior of a function call (Ch14.2)

block: 系統避免被浪費自己的時間，但要等IO

- A blocking call returns after the requested operations complete
block mode: e.g. 呼叫scanf()
- A non-blocking call returns STDIN 等user 來打字，等不到就很慘
 - Ack if the system receives and starts to process the request
 - Error if the system cannot process the request
- **Synchronized/Asynchronized IO: the behavior of data movement:**
read, write
 - Synchronized IO moves the data to the targeted devices and returns.
原本在CPU準備處理，但遇到後踢到IO queue排隊
預設是「不同步」
 - An asynchronized IO buffers the data and moves the data to the targeted device later.
人：希望系統永遠跑我寫的code，不要亂停掉
sys：希望爭取CPU 同時間內被充分利用，該停的就停、跑重要的



File I/O - creat and close

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

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

- open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode)
- This interface is made obsolete by open
- Only for write-access.
- Q: What if you want to create a file for READ and WRITE?

sys執行的第一個function，不是main()
而是放有argv...etc的那個

```
#include <unistd.h>
```

```
int close(int filedes);
```

- All open files are automatically closed by the kernel when a process terminates
- Closing a file descriptor releases any record locks on that file (see Ch14.3 for file locking)



mode Flags for O_CREAT

- **Defined in <sys/stat.h>**
 - S_IRUSR: owner read permit
 - S_IWUSR: owner write permit
 - S_IXUSR: owner execute permit
 - S_IRGRP: group read permit
 - S_IWGRP: group write permit
 - S_IXGRP: group execute permit
 - S_IROTH: others read permit
 - S_IWOTH: owners write permit
 - S_IXOTH: others execute permit
- **open("myfile", O_CREAT, S_IRUSR | S_IXOTH)**
- **Set umask variable for default file permit**

預防開出不想開的權限

666 rw rw

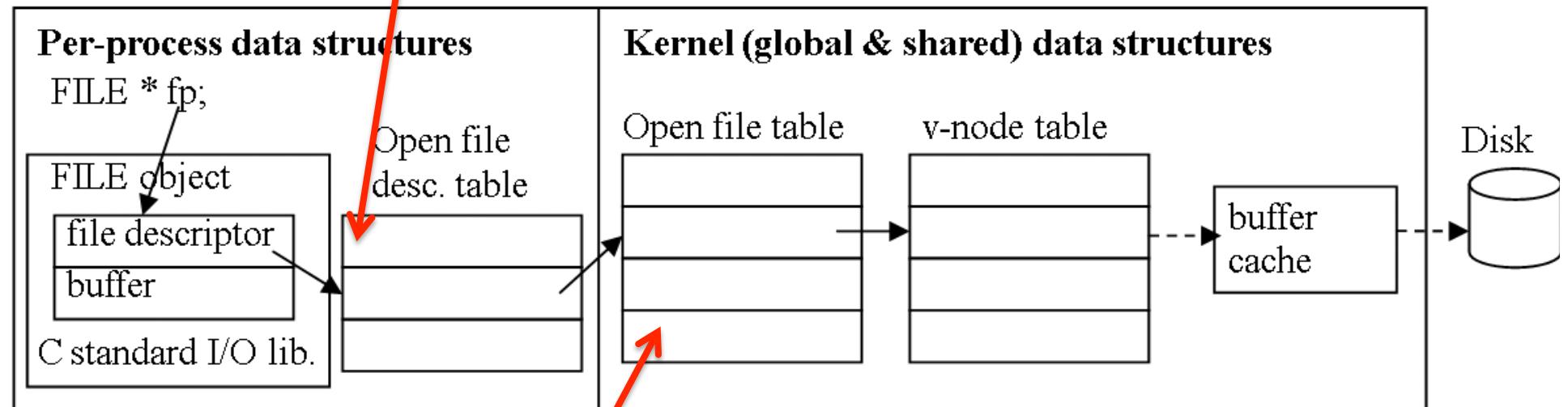
\$umask

644 r- r-



Where to store file descriptor & file status flags?

File descriptor

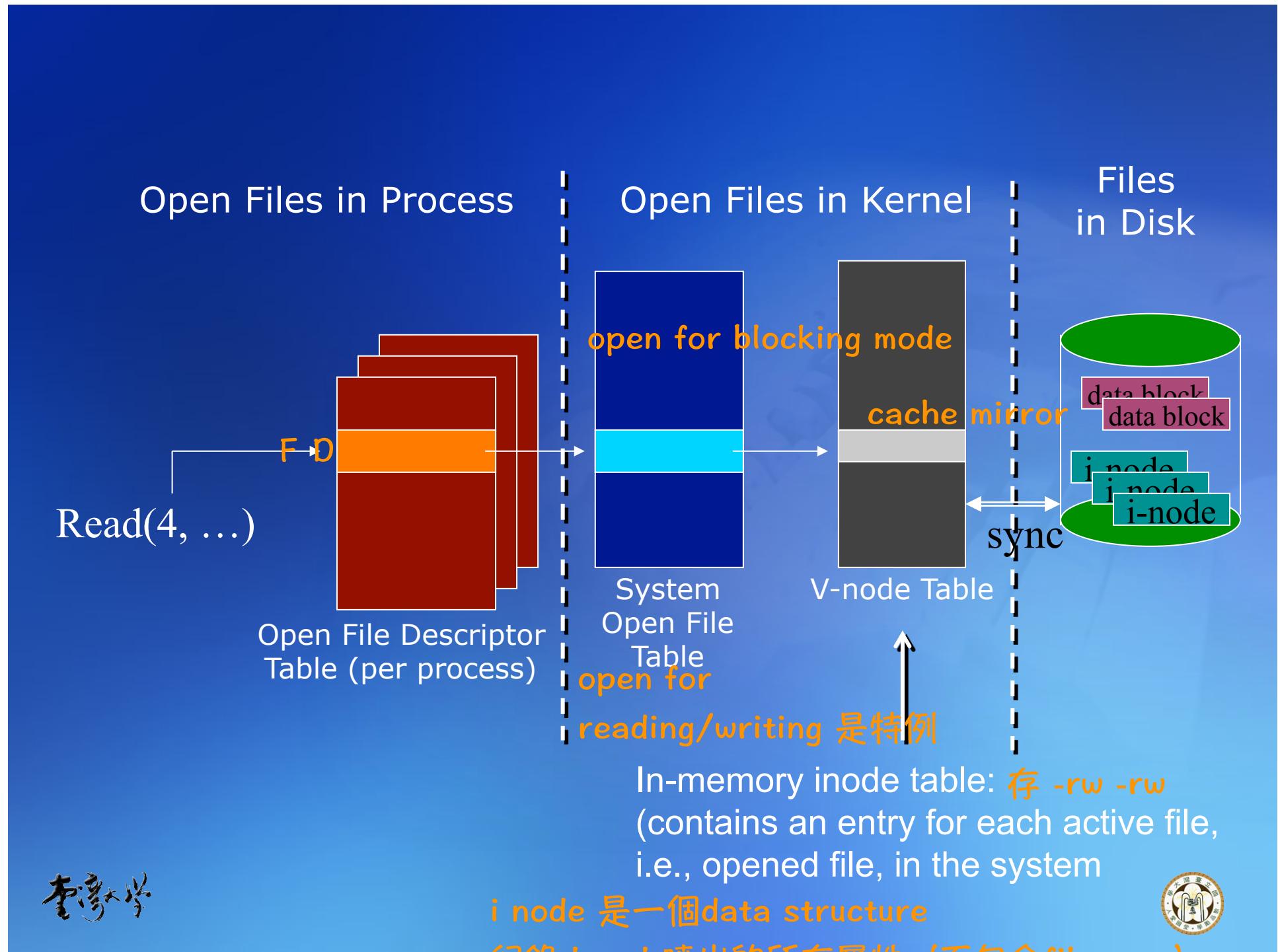


被指的人，都有一個reference count

不管是同process or 不同process

open 屬性一定會多開一個 (open file table多開一格)

File status flags such as read, write, append, sync,
and nonblocking



● Open File Descriptor Table

- Each process has its own
- A file descriptor contains
 - the file descriptor flags **COPY_ON_EXEC**
 - a pointer to a system open file table entry
- Child inherits from parents

process 會生小孩、並繼承父母 (fork, ch 8)

0, 1, 2 是預設的 parent

stdout(1) 建有的 ptr，如果被重新導向到 > out
就是 ptr 被導向的意思

● System Open File Table

- A set of all open files
- Shared by all processes
- Reference count of number of file descriptors pointing to each entry
- Each file table entry contains
 - the file's status flags for the file, the current file offset, &a pointer to the v-node table entry for the file
- Each open file corresponds to an entry
 - a disk file may be opened multiple times



- **V-node table** 直接記i-node就好

- Each file consists of file metadata (or attributes) and file content
- V-node table stores the metadata while buffer cache stores the content
- Each open file has a v-node structure
- Shared by all processes 一個檔案只有一個v-node，開好幾次仍是一個。
(跟sys open file table不同，每次都開新的)
- V-node
 - Invented by Peter Weinberger (Bell Lab)/Bill Joy (Sun) to support multiple file system types on a single computer system
 - No v-node in linux. A generic i-node is used. In SVR4, i-node contains/is replaced with v-node. **VFS: virtual file system**
 - i-node contains
 - file owner, file size, residing device, block
 - Each disk file corresponds to an entry when we open it



What's the output of the following program?

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int fd1, fd2;
    fd1 = open("foo1.txt", O_RDONLY, 0);
    close(fd1);
    fd2 = open("foo2.txt", O_RDONLY, 0);
    printf("fd2 = %d\n", fd2);
    exit(0);
}
```

open always returns *lowest* unopened descriptor



File I/O - lseek

c.f. fseek : 類似的function in c lib

是LSEEK，不是”I”SEEK

```
#include <sys/types>
```

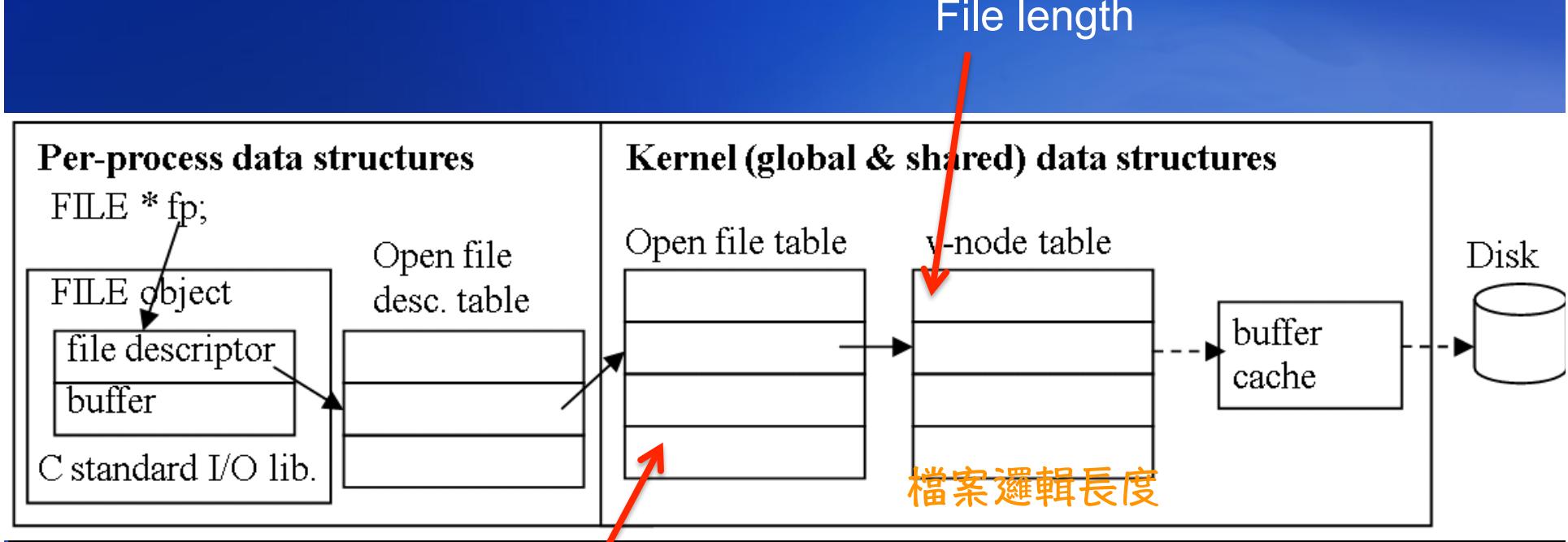
```
#include <unistd.h>
```

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

- Current file offset: number of bytes from the beginning of the file
- whence: SEEK_SET, SEEK_CUR, SEEK_END 從頭到目前，是第幾個byte
- Example
 - curpos = lseek(fd, 0, SEEK_CUR)
 - EPIPE for a pipe or a FIFO
 - off_t: **typedef long off_t; /* 2^{31} bytes */**
 - or **typedef longlong_t off_t; /* 2^{63} bytes */**
 - Negative for /dev/kmem on SVR4
 - **No I/O takes place until next read or write.**

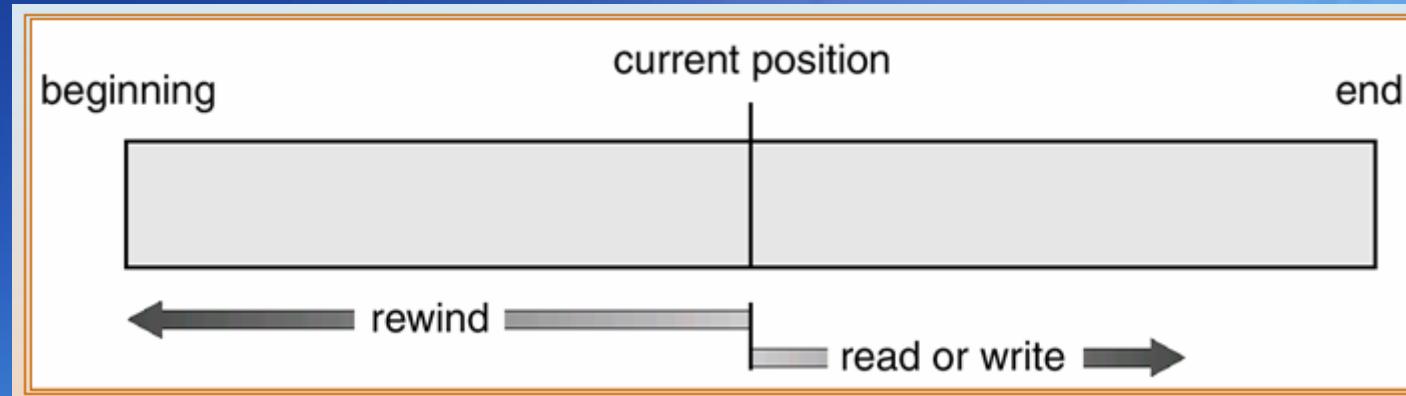


Where to store current file offset & file length?



“Weird” things you can do using lseek

- Seek to a negative offset
- Seek 0 bytes from the current position
- Seek past the end of the file



Example of Seeking

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>      檢查filetype可不可以lseek

int main(void)
{
    if (lseek(STDIN_FILENO, 0, SEEK_CUR) == -1)
        printf("cannot seek\n");
    else
        printf("seek OK\n");
    exit(0);
}
```

shell做：

```
$ ./a.out < /etc/motd      標準輸入STD變成/etc/motd ,
seek OK                      open file table的[0] ptr被轉向
$ cat < /etc/motd | ./a.out  pipe 本身就不能random access
cannot seek
$ ./a.out < /var/spool/cron/FIFO
cannot seek
```



Example of Creating a Hole

```
#include "apue.h"
#include <fcntl.h>

char    buf1[] = "abcdefghijkl";
char    buf2[] = "ABCDEFGHIJ";

int
main(void)
{
    int      fd;

    if ((fd = creat("file.hole", FILE_MODE)) < 0)
        err_sys("creat error");

    if (write(fd, buf1, 10) != 10)
        err_sys("buf1 write error");
    /* offset now = 10 */

    if (lseek(fd, 16384, SEEK_SET) == -1)
        err_sys("lseek error");
    /* offset now = 16384 */

    if (write(fd, buf2, 10) != 10)
        err_sys("buf2 write error");
    /* offset now = 16394 */

    exit(0);
}
```



od -c : dump files in the format of ASCII characters or backslash escapes

```
$ ./a.out
$ ls -l file.hole          check its size
-rw-r--r-- 1 sar           16394 Nov 25 01:01 file.hole
$ od -c file.hole          let's look at the actual contents
00000000  a  b  c  d  e  f  g  h  i  j  \0  \0  \0  \0  \0  \0
00000020  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
00400000  A  B  C  D  E  F  G  H  I  J
0040012
```

```
$ ls -ls file.hole file.nohole compare sizes
  8 -rw-r--r-- 1 sar           16394 Nov 25 01:01 file.hole
  20 -rw-r--r-- 1 sar           16394 Nov 25 01:03 file.nohole
```

of disk blocks

file size



File I/O - read and write

#include <unistd.h>

ssize_t read(int filedes, void *buf, size_t nbytes);

- **Less than nbytes of data are read:**

- EOF(0), terminal device (line-input), network buffering, record-oriented devices (e.g., tape), signal
- -1: error
- Offset is increased for every read() – SSIZE_MAX

#include <unistd.h>

ssize_t write(int filedes, const void *buf, size_t nbytes);

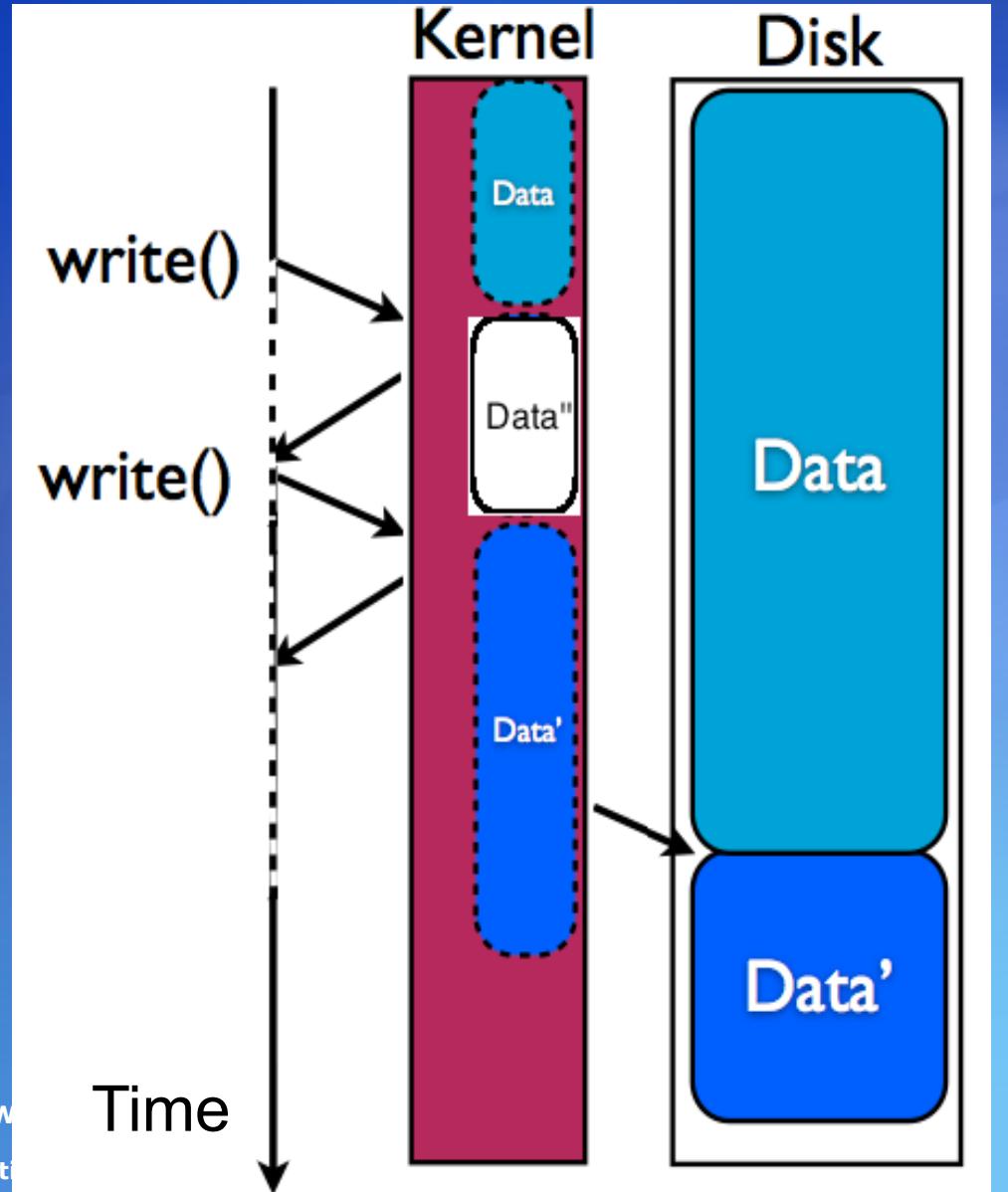
- Write errors for disk-full or file-size-limit causes.
- When O_APPEND is set, the file offset is set to the end of the file before each write operation.



Delayed Write

- Keep the data buffered so that multiple writes do not require multiple disk accesses.
- The buffer is queued for writing to disk at some later time.

每次寫都動到disk會很慢
所以用delayed write加速write



I/O Efficiency

How do you move from home to dorm?



Example of File I/O Efficiency

Copy standard input to standard output

```
#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);
}
```

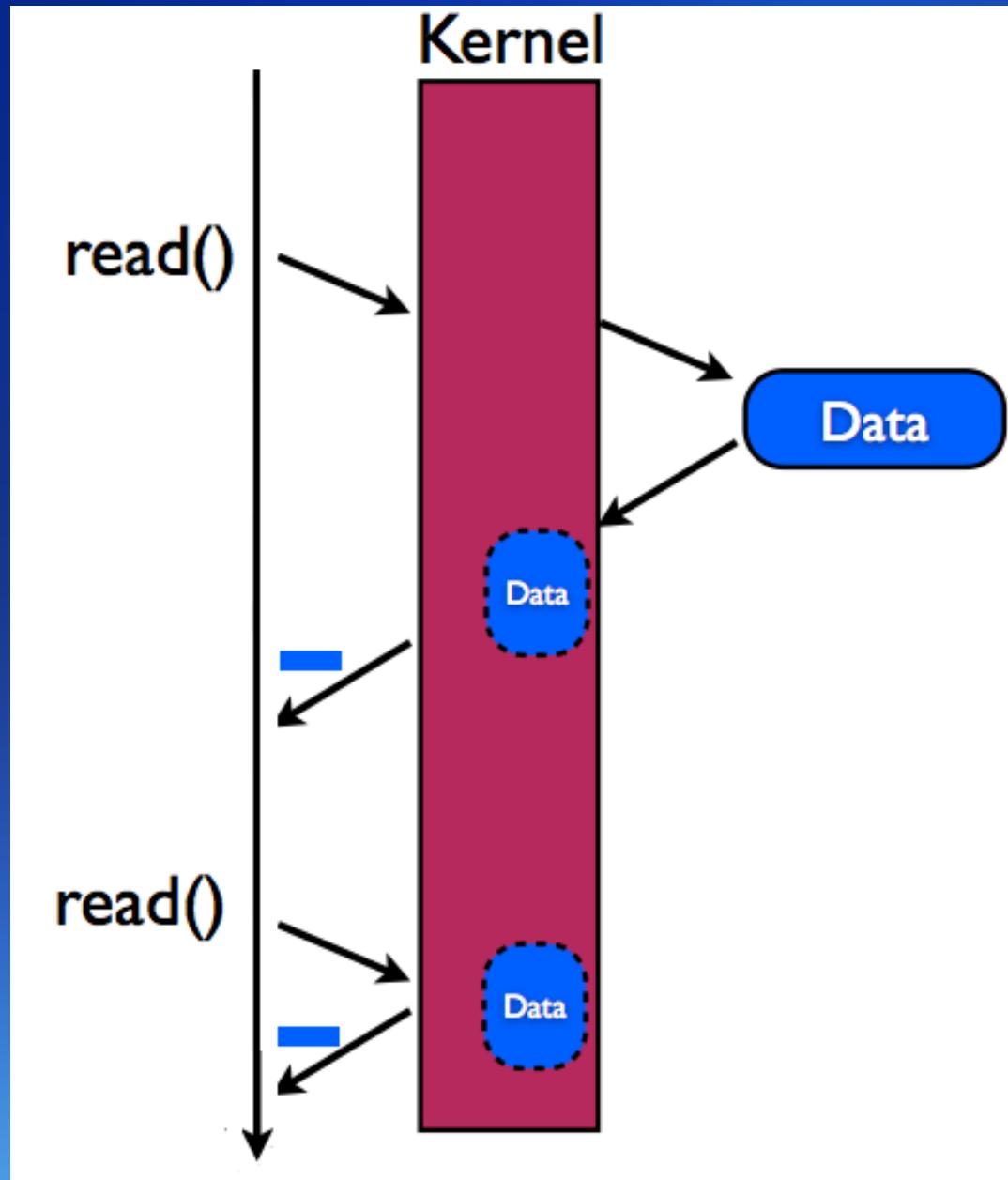


File I/O - Efficiency

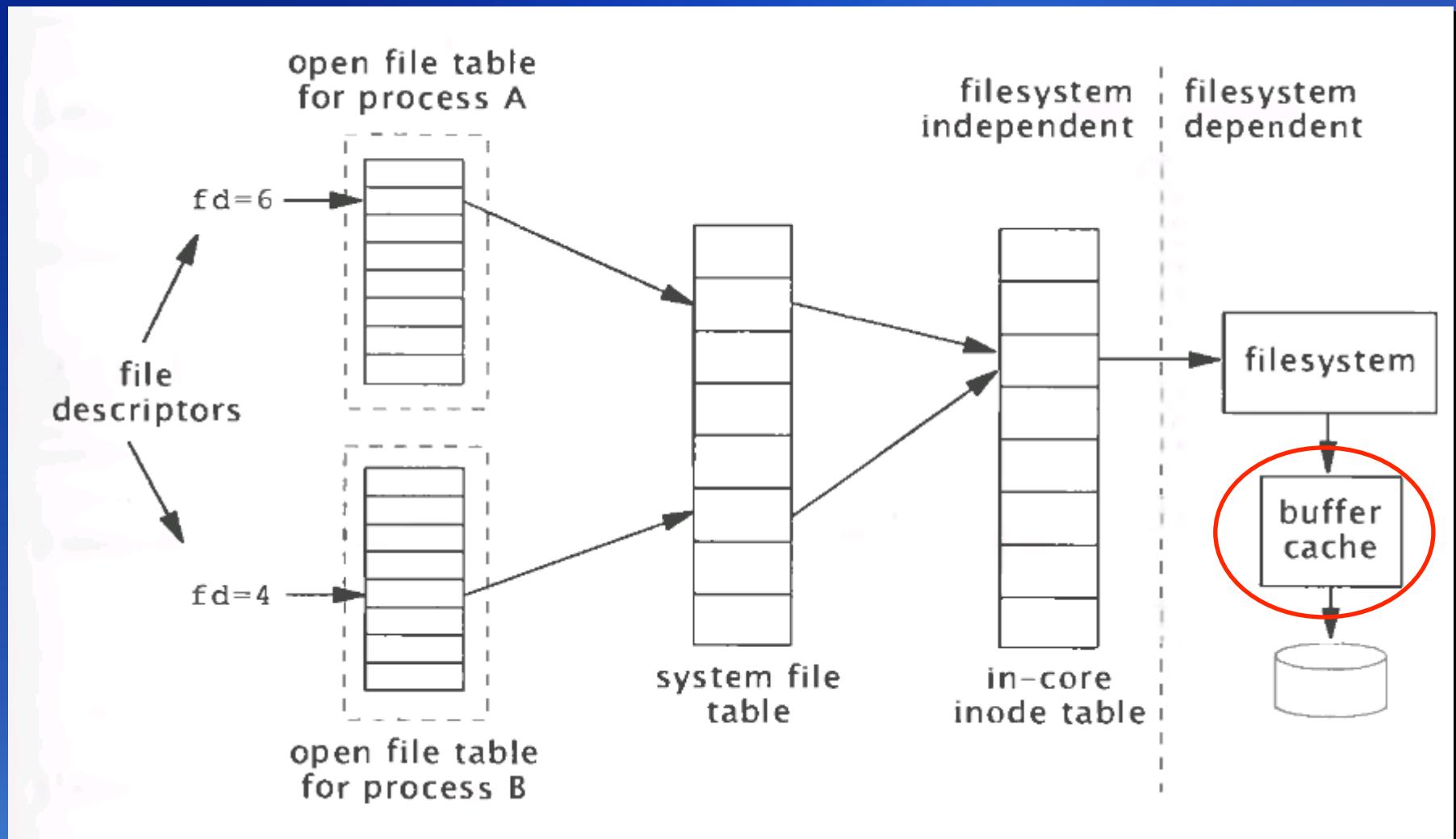
- No needs to open/close standard input/output
- Copy stdin to stdout (> /dev/null)
- Work for “text” and “binary” files since there is no such distinction in the UNIX kernel
- How do we know the optimal BUFSIZE?
- Try I/O redirection in reading an 103Mb file

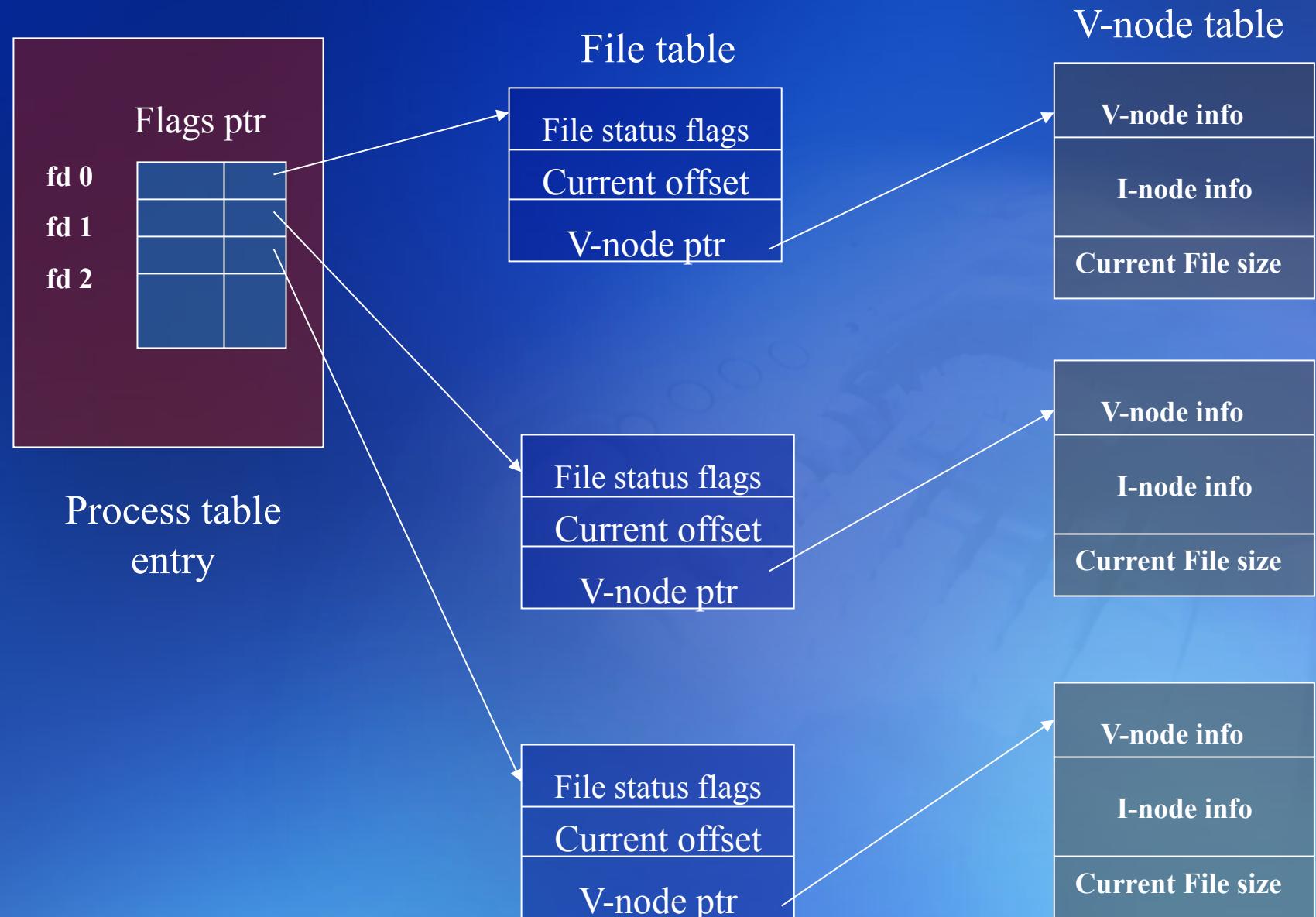
Buffer size	size of disk block		read-ahead	
	UsrCPU	SysCPU	Clock	#loops
1	124.89	161.65	288.64	103,316,352
64	2.11	2.48	6.76	1,614,318
512	0.27	0.41	7.03	201,789
1024	0.17	0.23	7.14	100,894
8192	0.01	0.18	6.67	12,611
131072	0	0.16	6.7	3,152





Increasing the buffer size beyond `st_blksize` (introduced in Ch4) has little positive effect.





What Happens with Each Call

- After each write completes, the current file offset in the file table entry is incremented. (If current file offset > current file size, change current file size in i-node table entry.)
- If file was opened O_APPEND, set corresponding flag in file status flags in file table. For each write, current file offset is first set to current file size from the i-node entry.
- Iseek simply adjusts current file offset in file table entry
- To Iseek to the end of a file, just copy current file size into current file offset.
- File descriptor flags versus file status flags
- dup() and fork() causes the sharing of entries in the (system) open file table. (Will discuss later)
 - After fork() the child process is a duplicate of the parent process in both memory and file descriptors.
- Reading a file simultaneously is no problem
- How about writing?



File I/O - Atomic Operations

- Appending to a file (**O_APPEND** in `open()`)
- Seeking and reading/writing (`pread()`, `pwrite()`)
- Creating a file (**O_CREAT|O_EXCL** in `open()`)



File I/O – Atomic Operations

- **Atomic Operation**
 - Composed of multiple steps?
- **Example – File Appending**

```
if (lseek(fd, 0L, SEEK_END) < 0) err_sys("lseek err");
if (write(fd, buf, 10) != 10) err_sys("wr err");
```

```
if (lseek(fd, 0L, 2) < 0) err_sys("lseek err");
if (write(fd, buf, 10) != 10) err_sys("wr err");
```

```
if (lseek(fd, 0L, 2) < 0) err_sys("lseek err");
if (write(fd, buf, 10) != 10) err_sys("wr err");
```



File I/O – Atomic Operations

- **Atomic Operation**
 - Composed of multiple steps?
- **Example – File Appending**

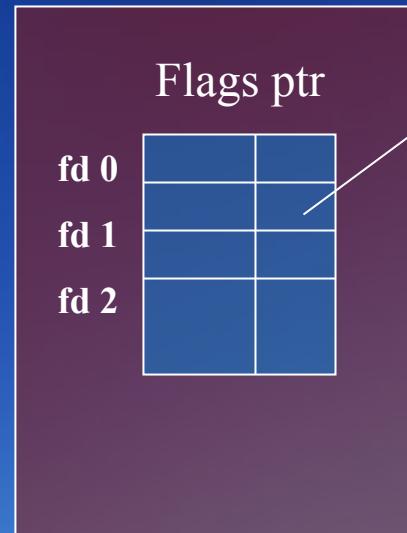
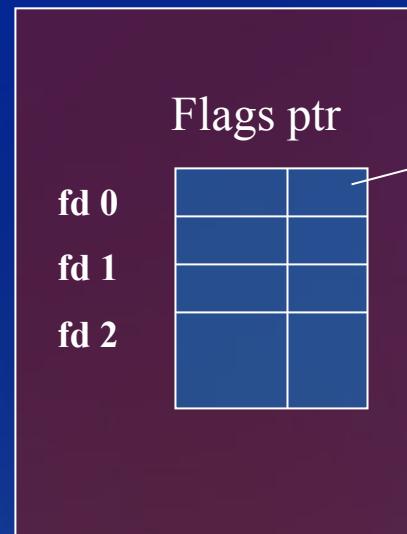
```
if (lseek(fd, 0L, SEEK_END) < 0) err_sys("lseek err");
if (write(fd, buf, 10) != 10) err_sys("wr err");
```

```
if (lseek(fd, 0L, 2) < 0) err_sys("lseek err");
```

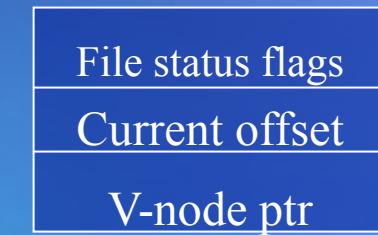
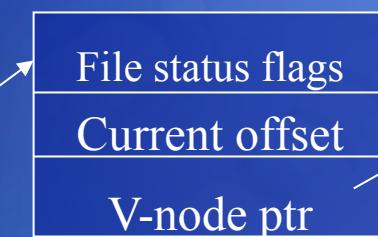
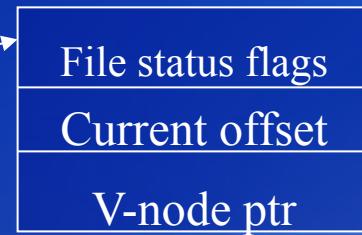
```
if (lseek(fd, 0L, 2) < 0) err_sys("lseek err");
if (write(fd, buf, 10) != 10) err_sys("wr err");
```

```
if (write(fd, buf, 10) != 10) err_sys("wr err");
```

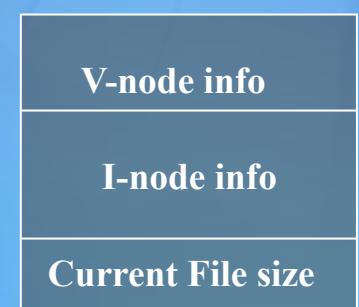




File table



V-node table



Atomic seek and read/write

#include <unistd.h>

ssize_t pread(int filedes, void *buf, size_t nbytes, off_t offset);

- Same to call lseek (from the start of the file) followed by read.
- Cannot interrupt pread

#include <unistd.h>

ssize_t pwrite(int filedes, const void *buf, size_t nbytes, off_t offset);

- Same to call lseek followed by write.
- Note that file offset is not affected by the two functions and the file referred by filedes should be seekable.**



File I/O – Atomic Operations

- **Example – File Creation**

```
if ((fd=open(pathname, O_WRONLY)) < 0)
    if (errno == ENOENT) {
        if ((fd = creat(pathname, mode)) < 0)
            err_sys("creat err");
    } else err_sys("open err");
    creat() rewrites and truncates any existing file.
```

- **When should the operations be atomic?**

- The result of one operation depends on the values of any shared resources including global variables, static variables, file status, etc, accessible by other processes.



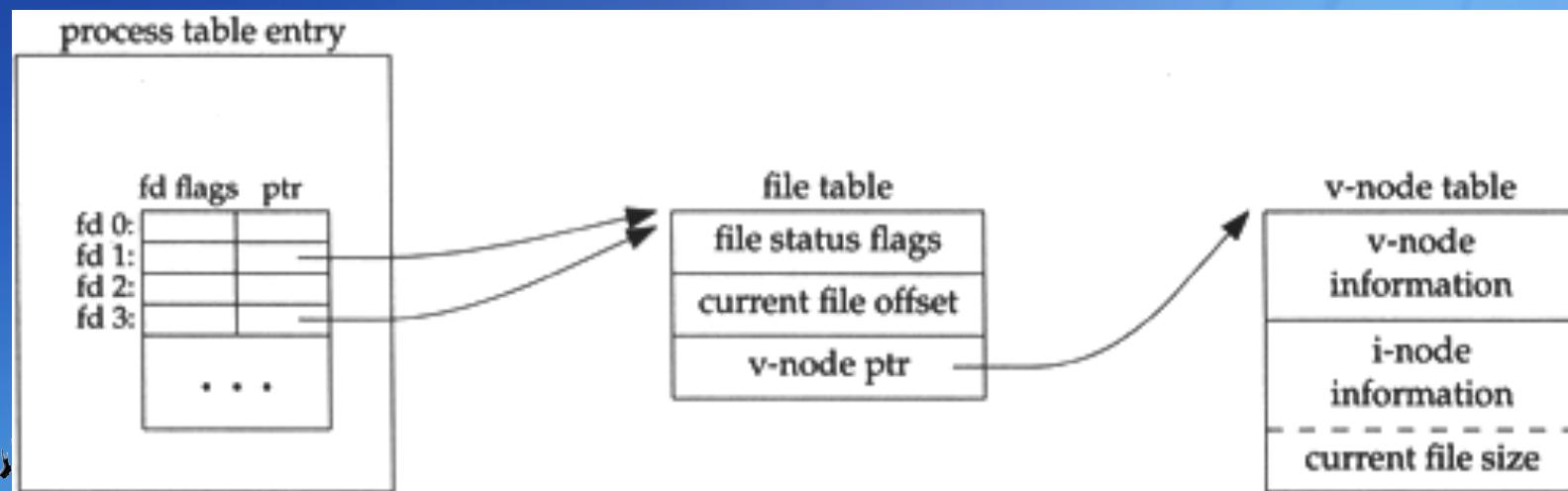
File I/O - dup and dup2

```
#include <unistd.h>
```

```
int dup(int filedes);
```

```
int dup2(int filedes, int newfiledes);
```

- Create a copy of the file descriptor
- dup() returns the lowest available file descriptor.
 - fcntl(filedes, F_DUPFD, 0);
- dup2() is atomic and from Version 7, ...SVR3.2
 - close(newfiledes); fcntl(filedes, F_DUPFD, newfiledes);



Suppose that `foobar.txt` consists of the 6 ASCII characters "foobar". Then what is the output of the following program?

```
int main()
{
    int fd1, fd2;
    char c;
    fd1 = open("foobar.txt", O_RDONLY, 0);
    fd2 = open("foobar.txt", O_RDONLY, 0);
    read(fd1, &c, 1);
    read(fd2, &c, 1);
    printf("c = %c\n", c);
    exit(0);
}
```

The descriptors `fd1` and `fd2` each have their own open file table entry, so each descriptor has its own file position for `foobar.txt`. Thus, the read from `fd2` reads the first byte of `foobar.txt`



As before, suppose foobar.txt consists of 6 ASCII characters "foobar". Then what is the output of the following program?

```
int main()
{
    int fd;
    char c;
    fd = open("foobar.txt", O_RDONLY, 0);
    if(fork() == 0)
        {read(fd, &c, 1); exit(0);}
    wait(NULL);
    read(fd, &c, 1);
    printf("c = %c\n", c);
    exit(0);
}
```

Child inherit's the parent's descriptor table. So child and parent share an open file table entry. Hence they share a file position. c='o'



Assuming that foobar.txt consists of 6 ASCII characters “foobar”. Then what is the output of the following program?

```
int main()
{
    int fd1, fd2;
    char c;
    fd1 = open("foobar.txt", O_RDONLY, 0);
    fd2 = open("foobar.txt", O_RDONLY, 0);
    read(fd2, &c, 1);
    dup2(fd2, fd1);
    read(fd1, &c, 1);
    printf("c = %c\n", c);
    exit(0);
}
```

We are redirecting **fd1** to **fd2**. (**fd1** now points to the same open file table entry as **fd2**). So the second **read** uses the file position offset of **fd2**. **c='o'**.



What is the output of the following program?

```
int main()
{
    int fd;
    char *s;
    fd = open("file", O_WRONLY | O_CREAT | O_TRUNC, 0666);
    dup2(fd,1);
    close(fd);
    printf("Hello %d\n", fd);
}
```

We are redirecting **stdout** to **fd**. So whenever your program writes to standard output, it will write to **fd**.



What is the output of “a.out file1 file2”?

```
int main(int argc, char **argv, char **envp)
{
    int fd1, fd2;
    int dummy;
    char *newargv[2];

    fd1 = open( argv[1], O_RDONLY);
    dup2(fd1, 0);
    close(fd1);

    fd2 = open( argv[2], O_WRONLY | O_TRUNC | O_CREAT, 0644);
    dup2(fd2, 1);
    close(fd2);

    newargv[0] = "cat";
    newargv[1] = (char *) 0;
    execve("/bin/cat", newargv, envp);

    exit(0);
}
```



File I/O: sync(), fsync(), fdatasync()

Kernel maintains a buffer cache between apps & disks.

#include <unistd.h>

int fsync(int filedes); // data + attr sync

int fdatasync(int filedes); // data sync

void sync();

// 1. queues all kernel modified block buffers

// for writing and returns immediately

// 2. called by daemon update & command sync

Apps write()

Kernel buffer
cache

Disks

Comparison with open()'s options

O_DSYNC

Have each `write` wait for physical I/O to complete, but don't wait for file attributes to be updated if they don't affect the ability to read the data just written.

O_RSYNC

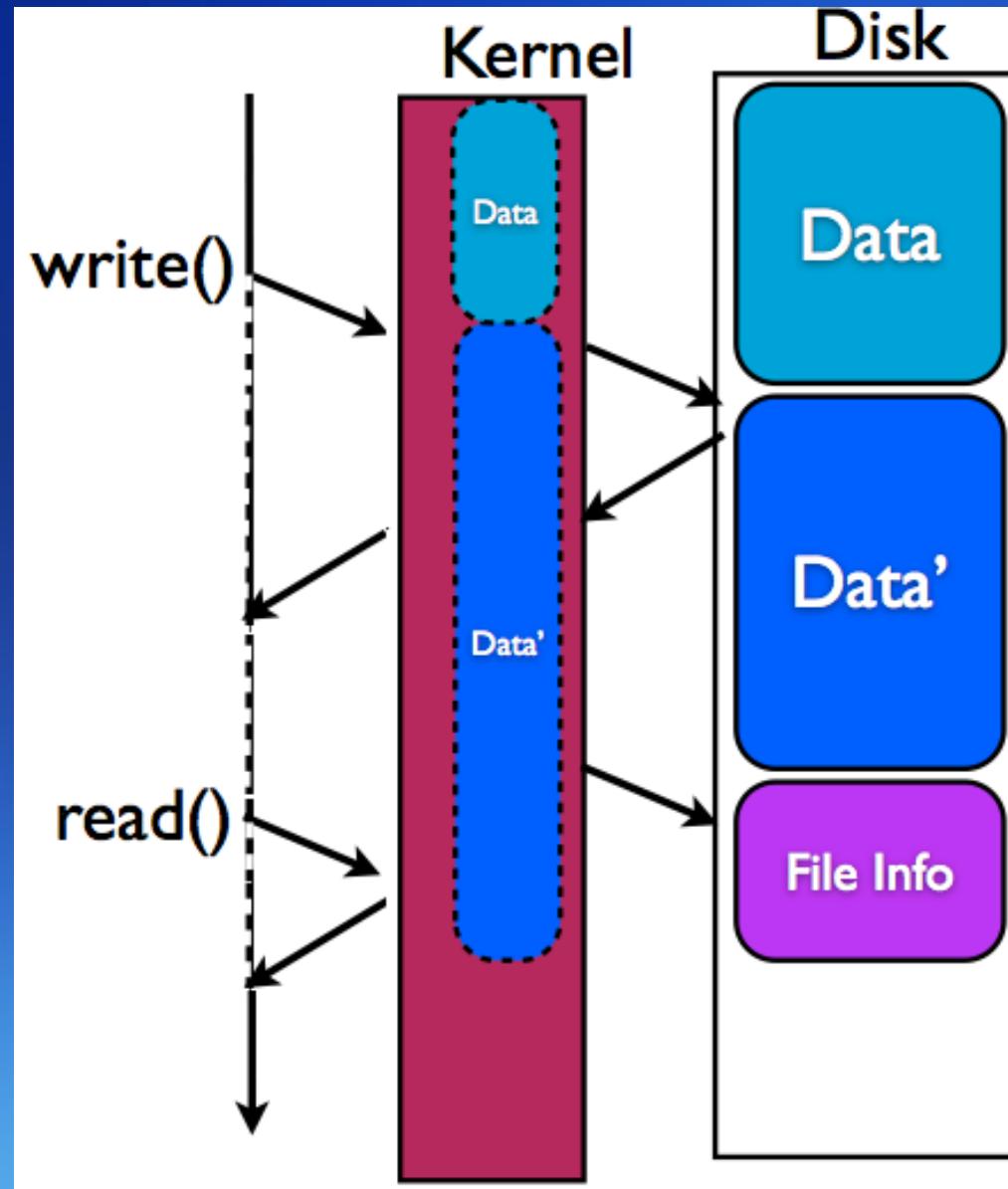
Have each `read` operation on the file descriptor wait until any pending writes for the same portion of the file are complete.

O_SYNC

Have each `write` wait for physical I/O to complete, including I/O necessary to update file attributes modified as a result of the `write`. We use this option in [Section 3.14](#).



Open with O_DSYNC



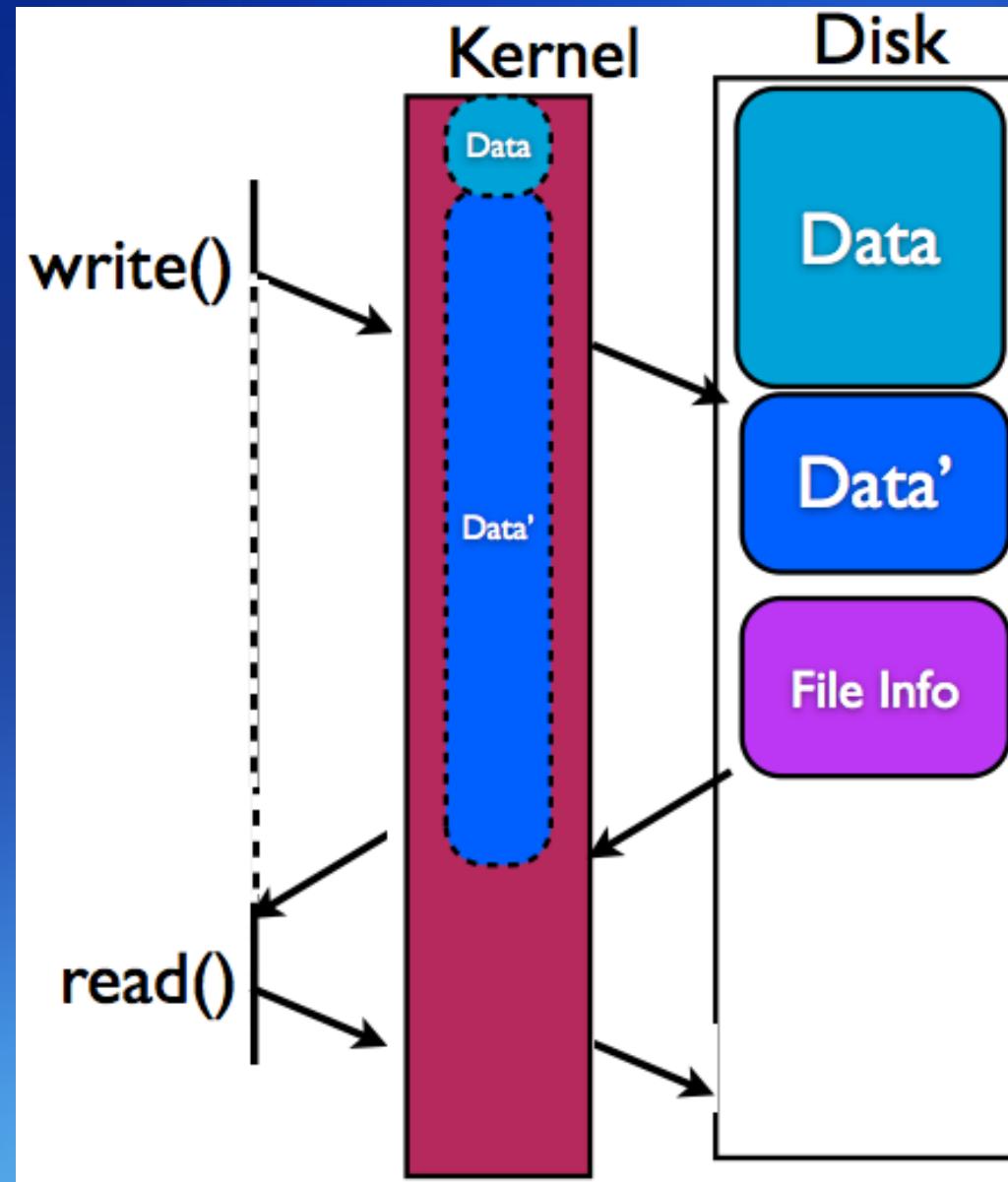
臺灣大學

Graduate Institute of Multimedia and Networking, National Taiwan University

Pu-Jen Cheng ©2007
Information Engineering



Open with O_SYNC



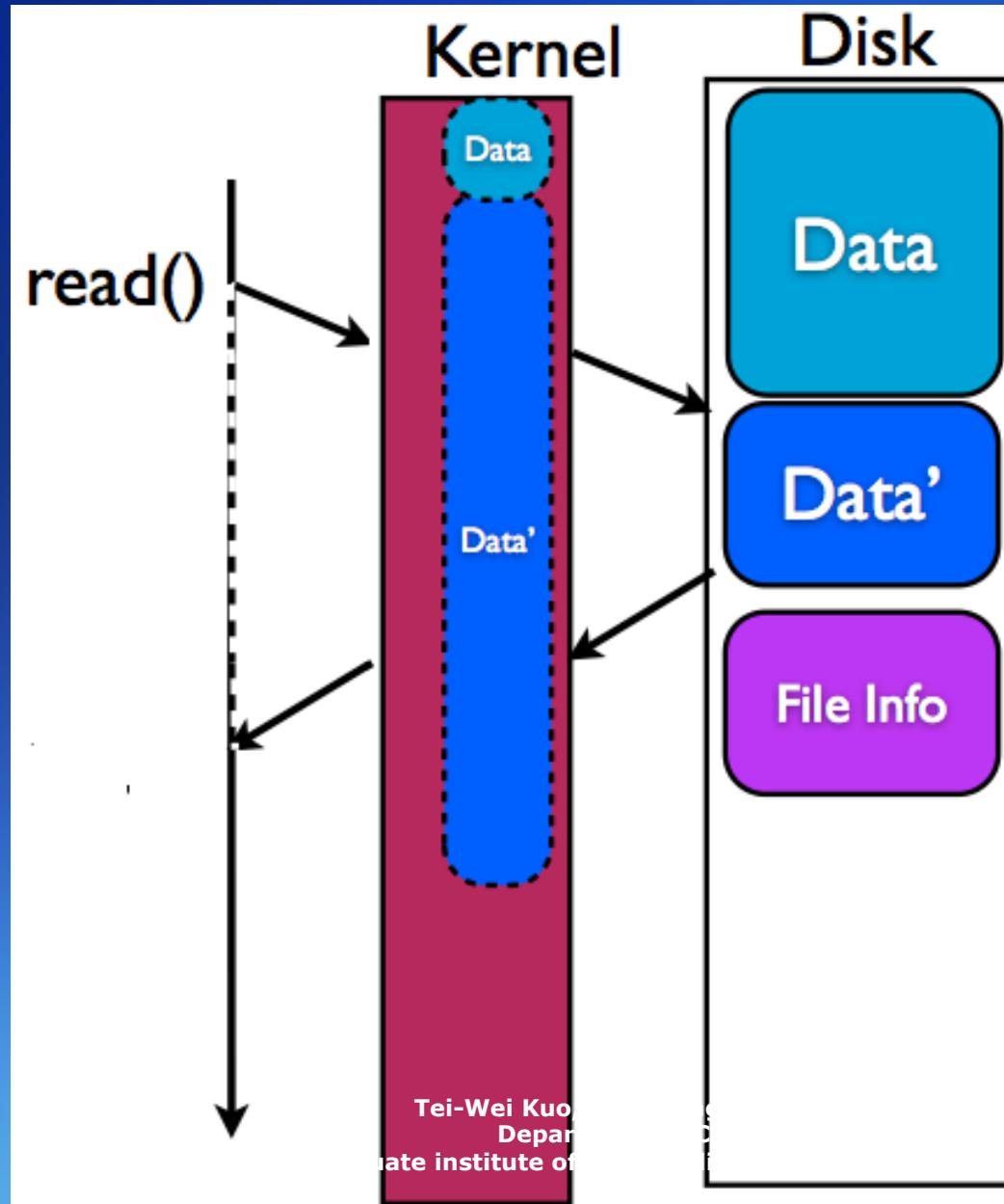
臺灣大學

Graduate Institute of Multimedia and Networking, National Taiwan University

Pu-Jen Cheng ©2007
Information Engineering



Open with O_RSYNC



Linux ext2 timing results using various synchronization mechanisms

1. Delayed write
2. The Linux file system isn't honoring the O_SYNC flag
(O_SYNC should increase system and clock time)

	Operation	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)
1	read time from Figure 3.5 for BUFSIZE = 4,096 (write to /dev/null)	0.03	0.16	6.86
2	normal <code>write</code> to disk file	0.02	0.30	6.87
	<code>write</code> to disk file with <code>O_SYNC</code> set	0.03	0.30	6.83
	<code>write</code> to disk followed by <code>fflush</code>	0.03	0.42	18.28
	<code>write</code> to disk followed by <code>fsync</code>	0.03	0.37	17.95
	<code>write</code> to disk with <code>O_SYNC</code> set followed by <code>fsync</code>	0.05	0.44	17.95



File I/O - fcntl

int fcntl(int filedes, int cmd, ... /* int arg */);

- Changes the properties of opened files
- **F_DUPFD:** duplicate an existing file descriptor (\geq arg).
 - FD_CLOEXEC (close-on-exec) is cleared (for exec()).
- **F_GETFD, F_SETFD:** file descriptor flag, e.g., **FD_CLOEXEC**
F_GETFL, F_SETFL: file status flags
 - O_APPEND, O_NONBLOCK, O_SYNC, O_ASYNC, O_RDONLY, O_WRONLY, RDWR
- **F_GETOWN, F_SETOWN:** ownership, + proID, -groupID
 - SIGIO, SIGURG – I/O possible on a filedes/urgent condition on I/O channel
- **F_GETLK, F_SETLK, F_SETLKW**
 - File lock



Print file flags for a specified descriptor

```
#include <sys/types.h>
#include <fcntl.h>
#include "ourhdr.h"

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

    if (argc != 2)
        err_quit("usage: a.out <descriptor#>");

    if ((val = fcntl(atoi(argv[1]), F_GETFL, 0)) < 0)
        err_sys("fcntl error for fd %d", atoi(argv[1]));

    accmode = val & O_ACCMODE;
    if (accmode == O_RDONLY)    printf("read only");
    else if (accmode == O_WRONLY) printf("write only");
    else if (accmode == O_RDWR)   printf("read write");
    else err_dump("unknown access mode");

    if (val & O_APPEND)          printf(", append");
    if (val & O_NONBLOCK)         printf(", nonblocking");
#ifndef _POSIX_SOURCE && defined(O_SYNC)
    if (val & O_SYNC)            printf(", synchronous writes");
#endif
    putchar('\n');
    exit(0);
}
```

```
$ ./a.out 0 < /dev/tty
read only
$ ./a.out 1 > temp.foo
$ cat temp.foo
write only
$ ./a.out 2 2>>temp.foo
write only, append
$ ./a.out 5 5<>temp.foo
read write
```



Turn on one or more flags

val &= ~flags: clear the flag.
set_fl(STDOUT_FILENO, O_SYNC);

```
#include      <fcntl.h>
#include      "ourhdr.h"

void
set_fl(int fd, int flags) /* flags are file status flags to turn on */
{
    int          val;

    if ((val = fcntl(fd, F_GETFL, 0)) < 0)
        err_sys("fcntl F_GETFL error");

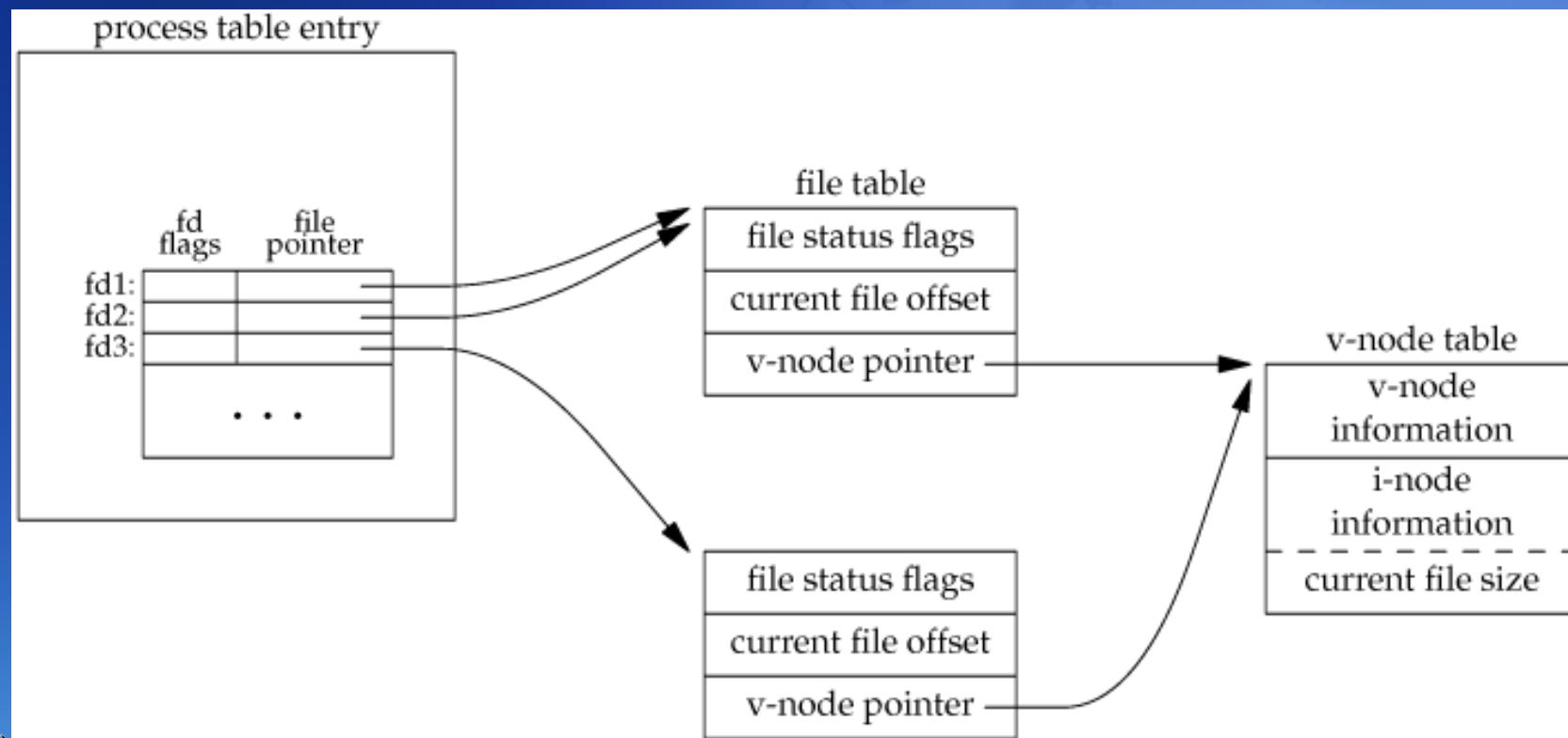
    val |= flags;           /* turn on flags */

    if (fcntl(fd, F_SETFL, val) < 0)
        err_sys("fcntl F_SETFL error");
}
```



Which descriptors are affected by an fcntl on fd1 with F_SETFD and F_SETFL?

```
fd1 = open(pathname, oflags);  
fd2 = dup(fd1);  
fd3 = open(pathname, oflags);
```



File I/O - ioctl

#include <unistd.h>

#include <sys/ioctl.h>

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

- **Catchall for I/O operations**
 - E.g., setting of the size of a terminal's window.
- **More headers could be required:**
 - Disk labels (<disklabel.h>), file I/O (<ioctl.h>), mag tape (<mtio.h>), socket I/O (<ioctl.h>), terminal I/O (<ioctl.h>)



File I/O - /dev/fd

- **/dev/fd/n**

- open("/dev/fd/n", mode) → duplicate descriptor *n* (assuming that *n* is open)
 - fd=open("/dev/fd/0", mode) == fd=dup(0)
 - The new mode is ignored or a subset of that of the referenced file.
- Uniformity and Cleanliness!
 - Not POSIX.1, but supported by SVR4 and 4.3+BSD
 - /dev/stdin -> ./fd/0
 - cat /dev/fd/0



What You Should Know

- **Unbuffered I/O**
- **Kernel data structures for unbuffered I/O**
- **File manipulation**
- **Redirection by dup(), dup2()**
- **Atomic operations**
- **I/O efficiency**
 - Buffer size, synchronization



Error Handling

- **Error Handling**
 - errno in <errno.h> (sys/errno.h)
 - E.g., 15 error numbers for open()
 - #define ENOTTY 25 /* Inappropriate ioctl for device */
 - Never cleared if no error occurs
 - No value 0 for any error number
- **Functions**
 - char *strerror (int errnum) (<string.h>)
 - void perror(const char *msg) (<stdio.h>)



```
#include      <errno.h>
#include      "apue.h"

int
main(int argc, char *argv[])
{
    fprintf(stderr, "EACCES: %s\n",
    strerror(EACCES));

    errno = ENOENT;
    perror(argv[0]);

    exit(0);
}
```

```
$ a.out
EACCES: Permission denied
a.out: No such file or directory
```



Error Recovery

- Fatal error: no recovery action
- Nonfatal error: delay and try again
 - Improves robustness by avoiding an abnormal exit
- Examples
 - **EAGAIN, ENFILE, ENOBUFS, ENOLCK, ENOSPC, ENOSR, EWOULDBLOCK, ENOMEM**

