系統程式設計
Systems Programming

鄭卜壬教授
臺灣大學資訊工程系

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 (Unbuffered I/O)
4. Standard I/O Library (Buffered I/O)
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

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

Example of Unbuffered I/O

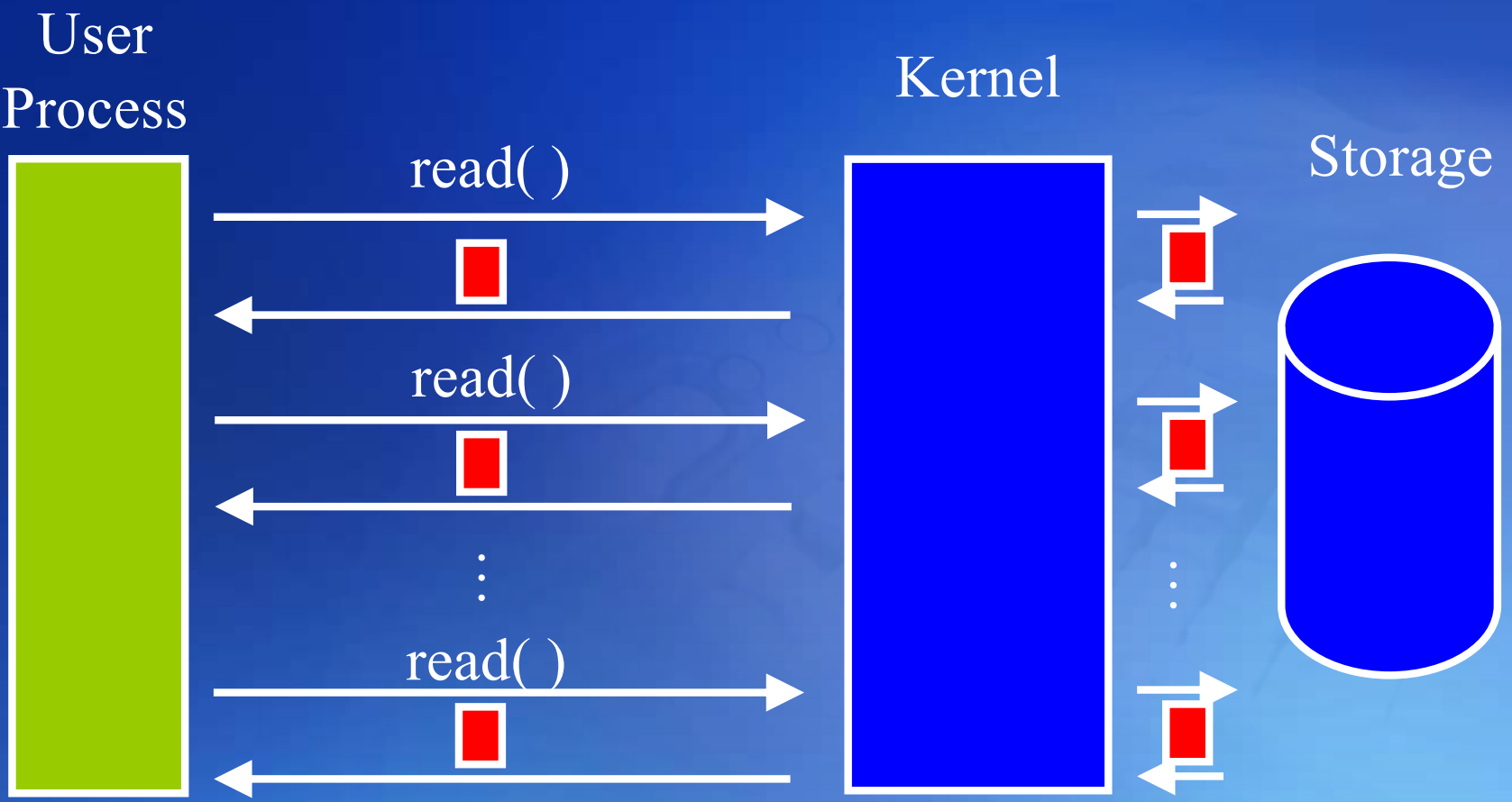
```c
#include <unistd.h>

int main(void)
{
        char buf[100];
        ssize_t n;

        while ( (n=read( STDIN_FILENO, buf, 100 )) != 0 )
                write( STDOUT_FILENO, buf, n );
        return 0;
}
```

Unbuffered I/O

Copy standard input to standard output

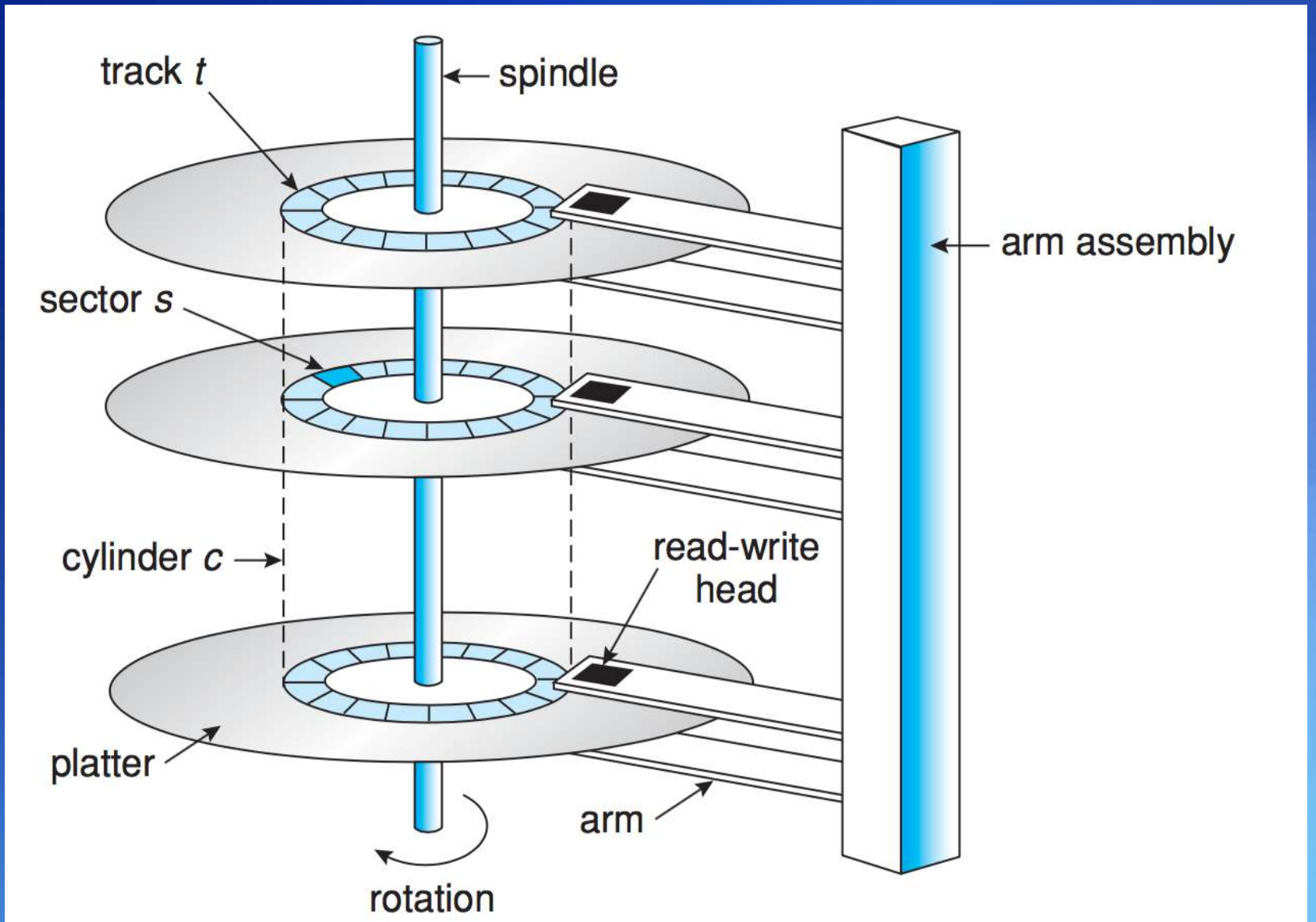
- **UNIX Standardization**
  - ANSI C (stdio.h, stdlib.h, string.h, math.h, time.h, …)
    - Provide portability of conforming C programs to a wide variety of OS's
  - POSIX (unistd.h, pwd.h, dirent.h, grp.h, fcntl.h, …)
    - Define the application programming interface for software compatible with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

Unbuffered I/O

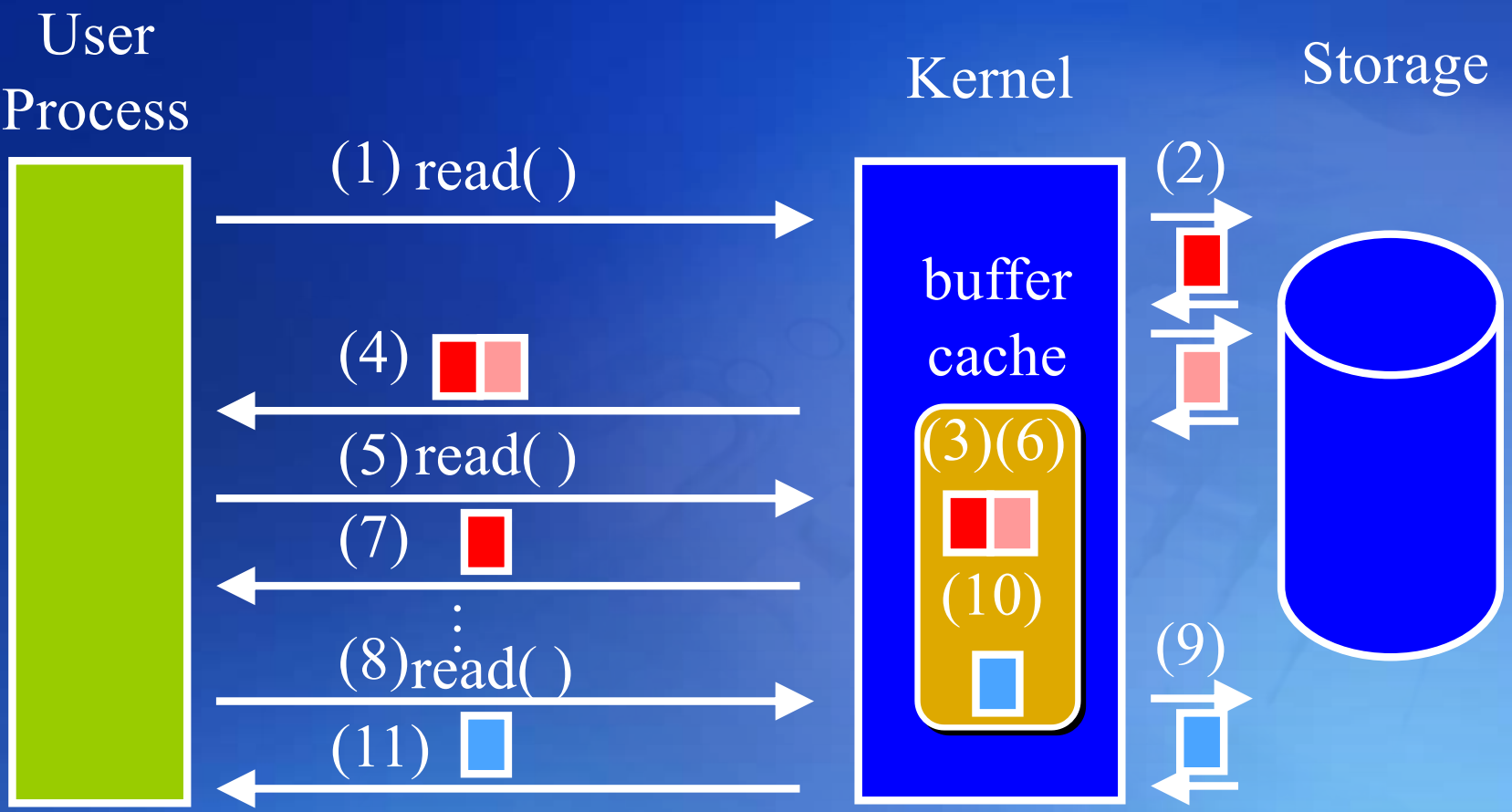


Each read() and write() invokes a system call in the kernel (part of POSIX.1)!

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


track t
spindle
arm assembly
sector s
cylinder c
read-write
head
platter
arm
rotation

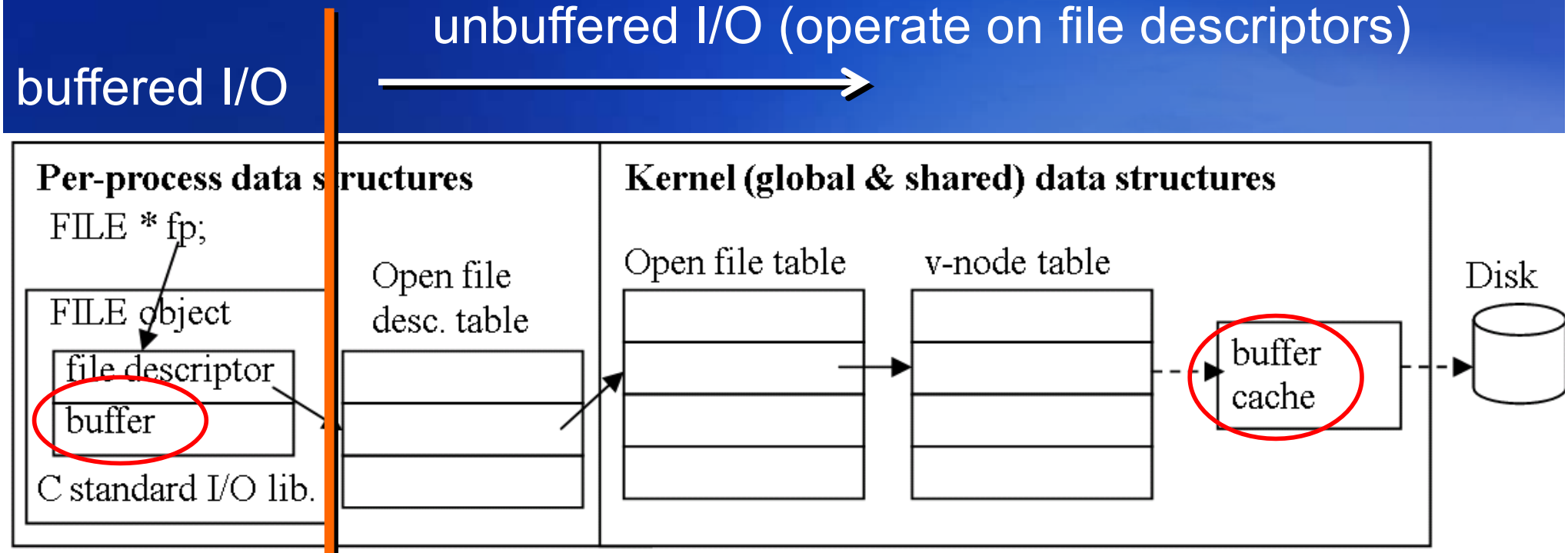
Unbuffered I/O



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

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

What "Unbuffered" refer to?



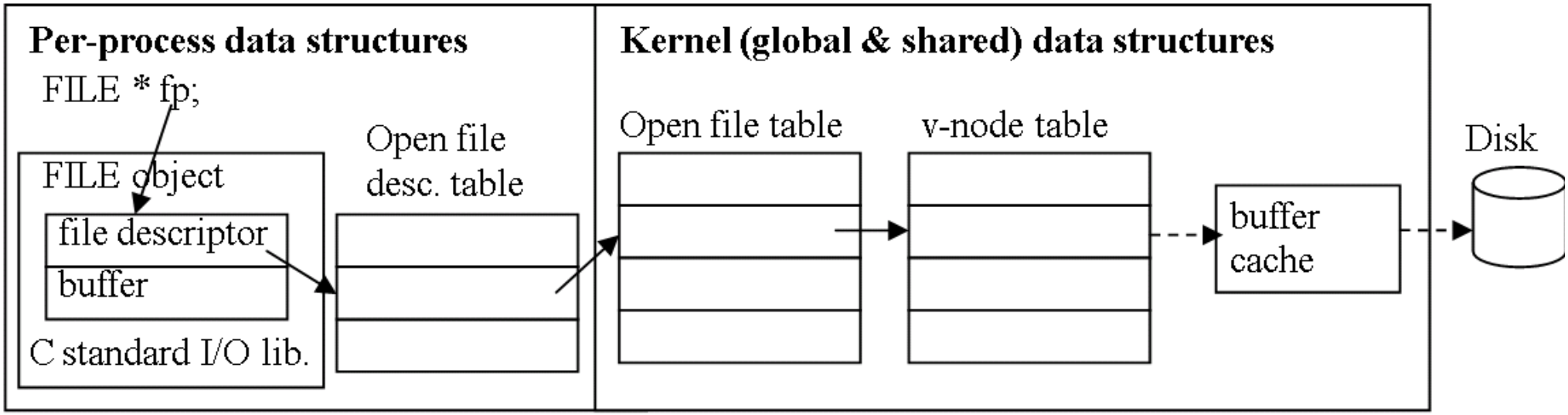
- Buffer in user's process is provided by the C standard library.
- Buffer cache is provided by the operating system

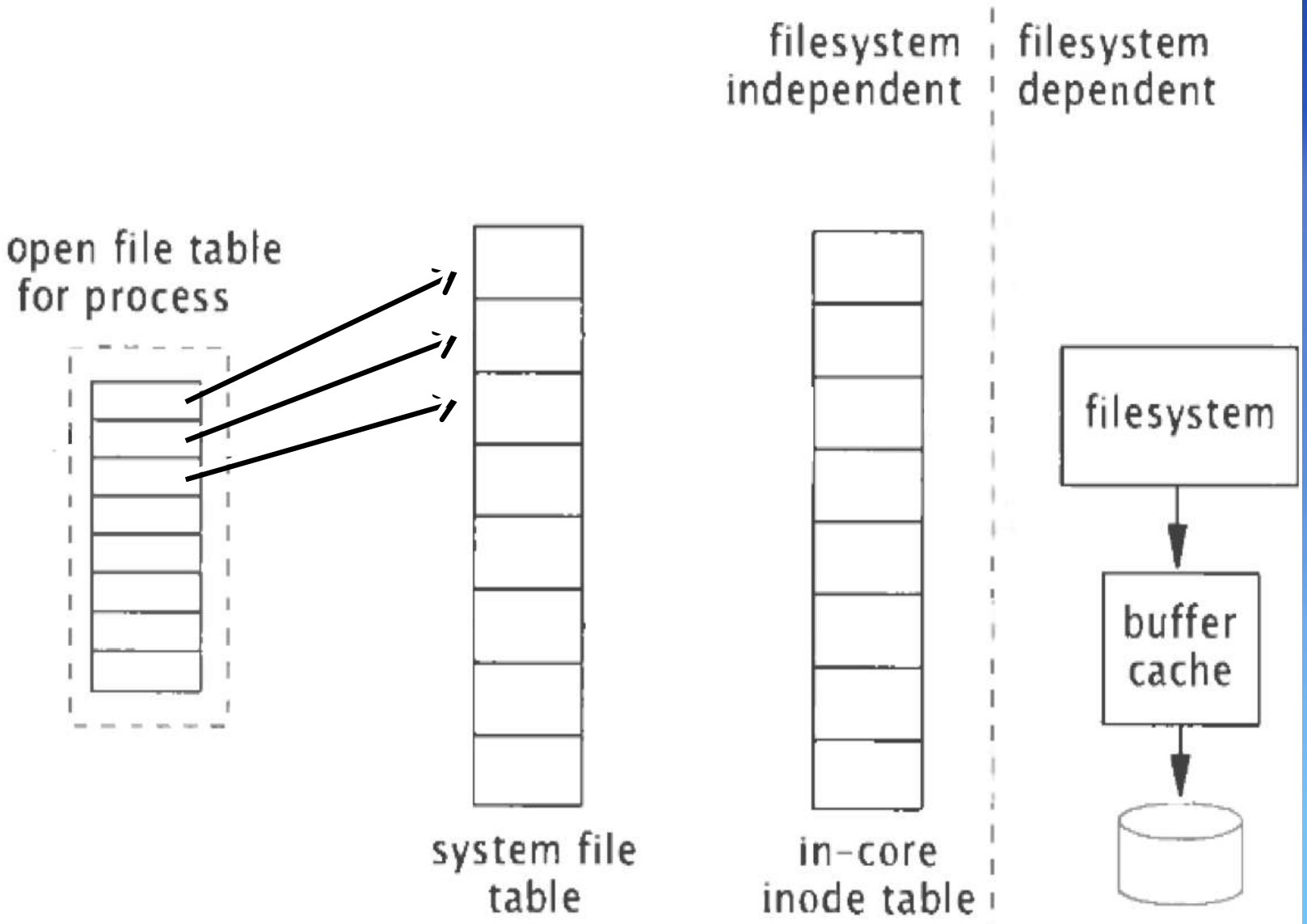
File I/O (or Unbuffered I/O)

- **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

- **File Descriptor**
  - 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
  - Per-process base
  - POSIX.1 – 0: STDIN_FILENO,
    1: STDOUT_FILENO, 2: STDERR_FILENO
    - <unistd.h>
    - Convention employed by the Unix shells and applications


Per-process data structures
FILE * fp;
FILE object
file descriptor
buffer
C standard I/O lib.
Open file
desc. table
Kernel (global & shared) data structures
Open file table
v-node table
buffer
cache
Disk


filesystem independent
filesystem dependent
open file table for process
system file table
in-core inode table
filesystem
buffer cache

File I/O – open, openat

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**
  - PATH_MAX, NAME_MAX
  - _POSIX_NO_TRUNC -> ENAMETOOLONG if error occurs
- **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

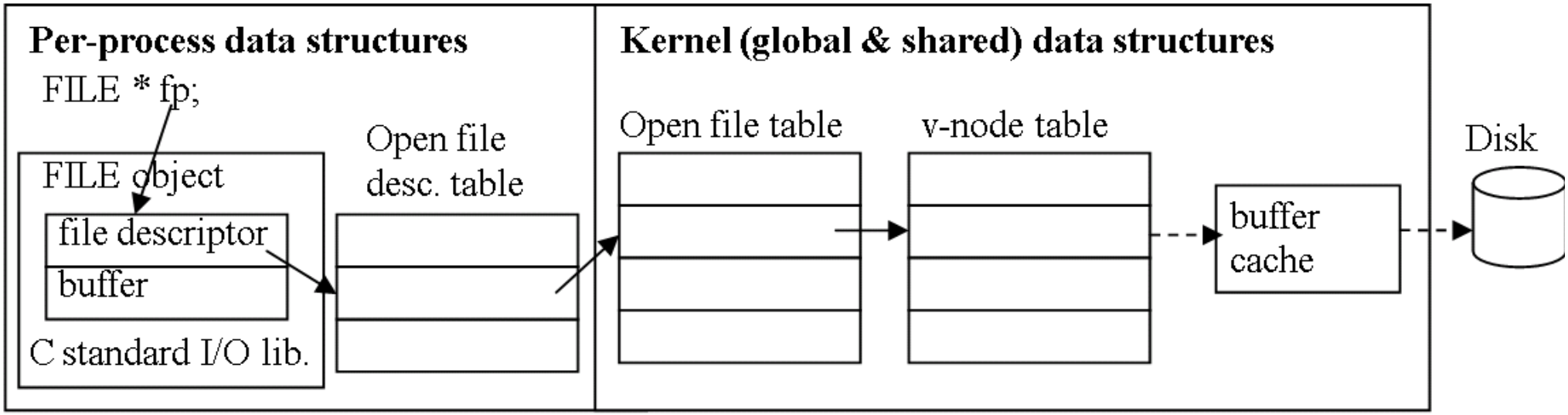
    dirfd == AT_FDCWD -> openat() == open()

    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

```
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
  - 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.


Per-process data structures
FILE * fp;
FILE object
file descriptor
buffer
C standard I/O lib.
Open file
desc. table
Kernel (global & shared) data structures
Open file table
v-node table
buffer
cache
Disk

Blocking vs. Synchronization

- **Block/non-block: the behavior of a function call (Ch14.2)**
  - A blocking call returns after the requested operations complete
  - A non-blocking call returns
    - 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:**
  - Synchronized IO moves the data to the targeted devices and returns.
  - An asynchronized IO buffers the data and moves the data to the targeted device later.

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?

```
#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

$ umask
0022

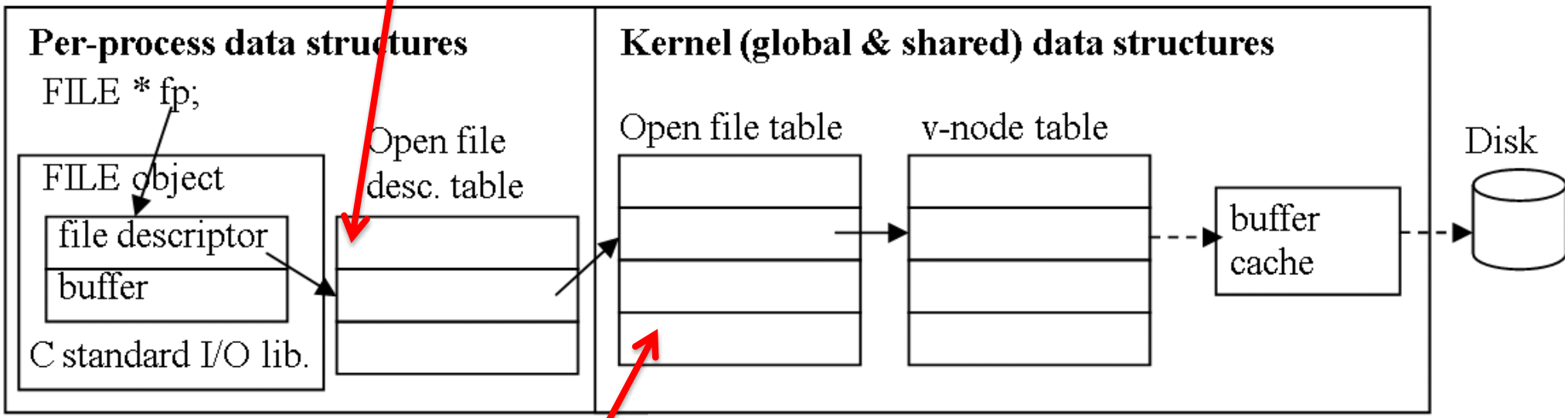
create: 0777
result: 0755

- **open("myfile", O_CREAT, S_IRUSR | S_IXOTH)**
- **Set *umask* variable for default file permit**

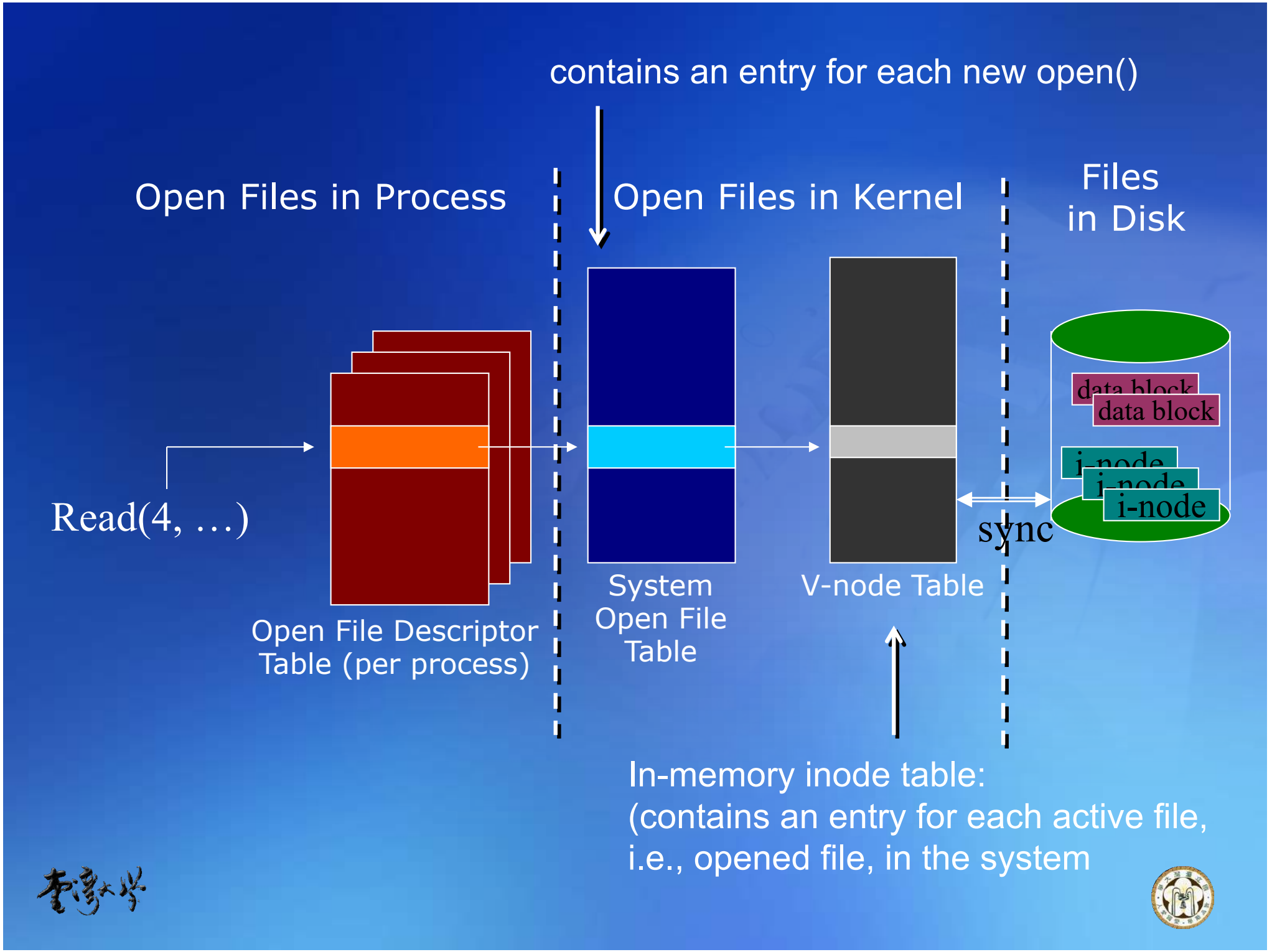
Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Where to store file descriptor & file status flags?

File descriptor



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


contains an entry for each new open()
Open Files in Process
Open Files in Kernel
Files in Disk
Read(4, …)
Open File Descriptor Table (per process)
System Open File Table
V-node Table
sync
data block
data block
i-node
i-node
i-node
In-memory inode table:
(contains an entry for each active file,
i.e., opened file, in the system

- **Open File Descriptor Table**
  - Each process has its own
  - A file descriptor contains
    - the file descriptor flags
    - a pointer to a system open file table entry
  - Child inherits from parents
- **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 files 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**
  - 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
  - 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.
  - 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?

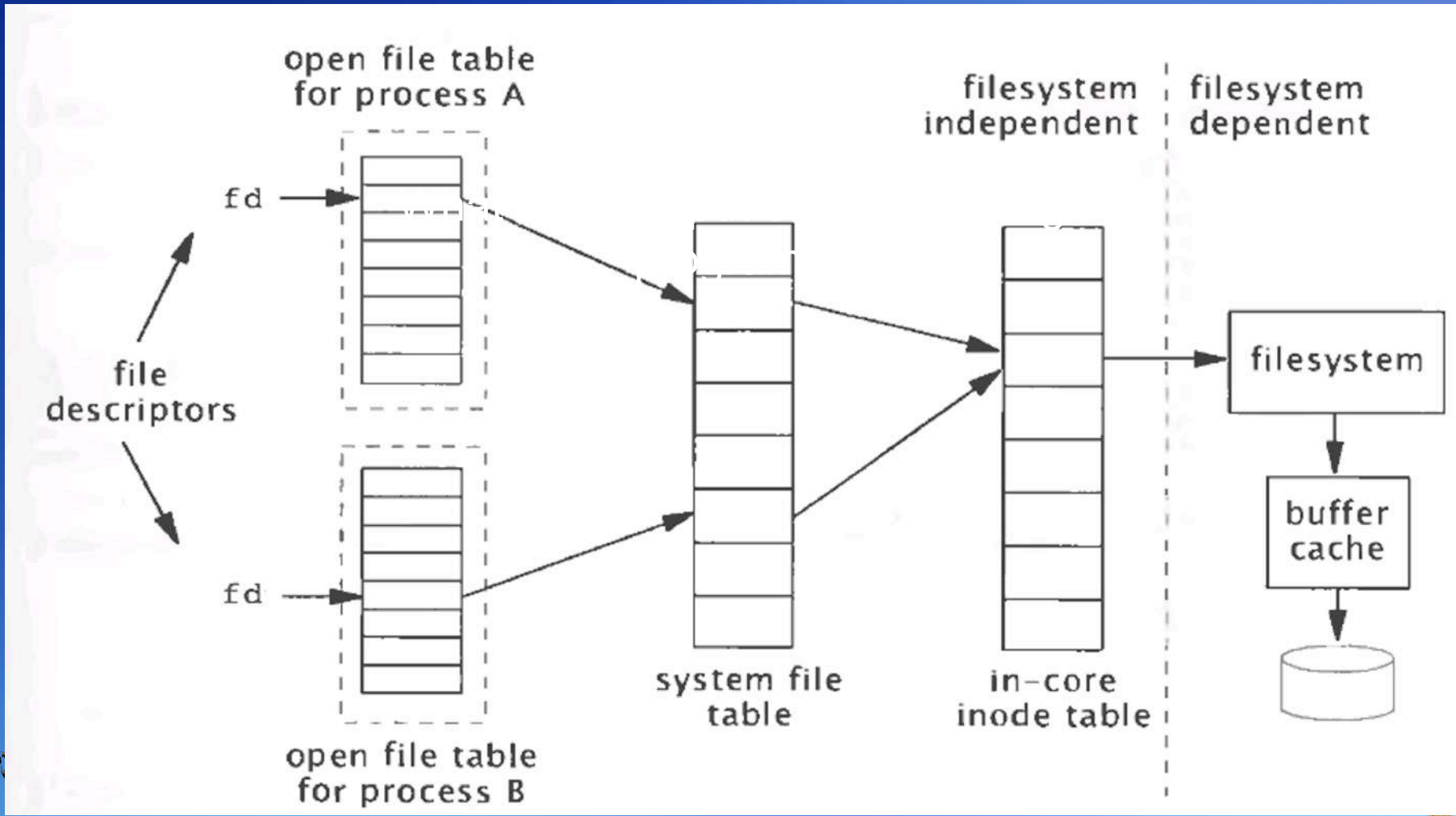
```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int fd1, fd2;
    fd1 = open("foo1.txt", O_RDONLY, 0);
    close(fd1);
    fd2 = open("foo2.txt", O_RDONLY, 0);
    printf("fd2 = %d\n", fd2);
    exit(0);
}
```

open always returns *lowest* unopened descriptor

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

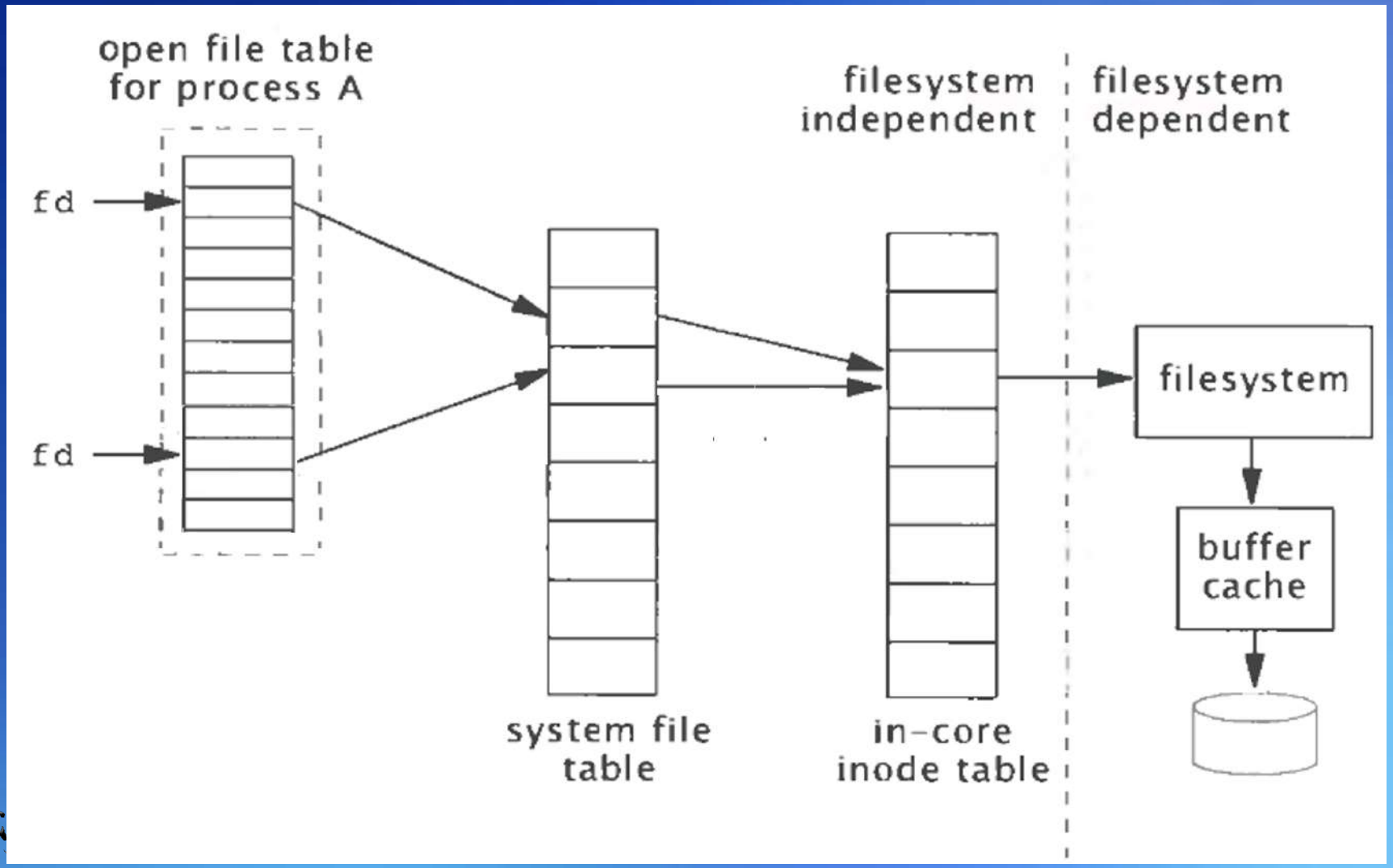
Two processes open the same file

```
Process A:
    open("foo.txt", O_RDONLY, 0);
Process B:
    open("foo.txt", O_RDONLY, 0);
```

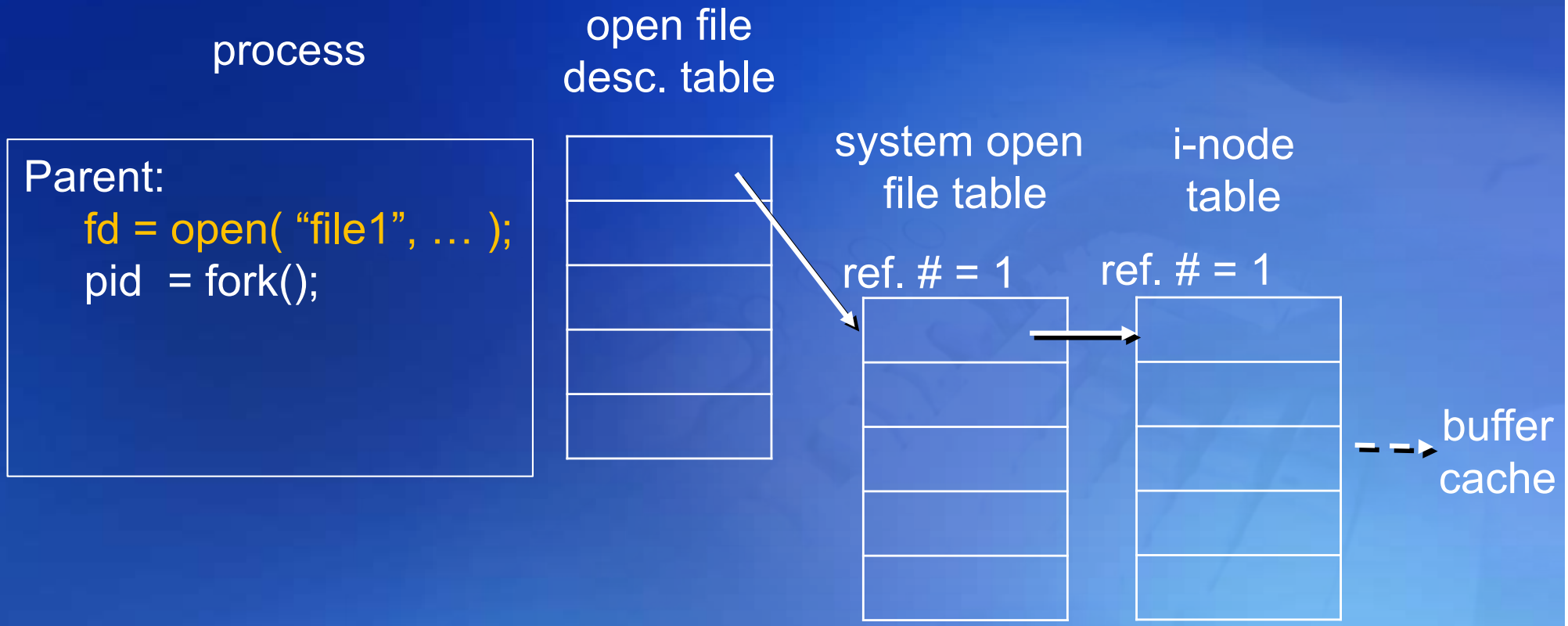


Open process opens the same file twice

```
Process A:
    open("foo.txt", O_RDONLY, 0);
    open("foo.txt", O_WRONLY, 0);
```

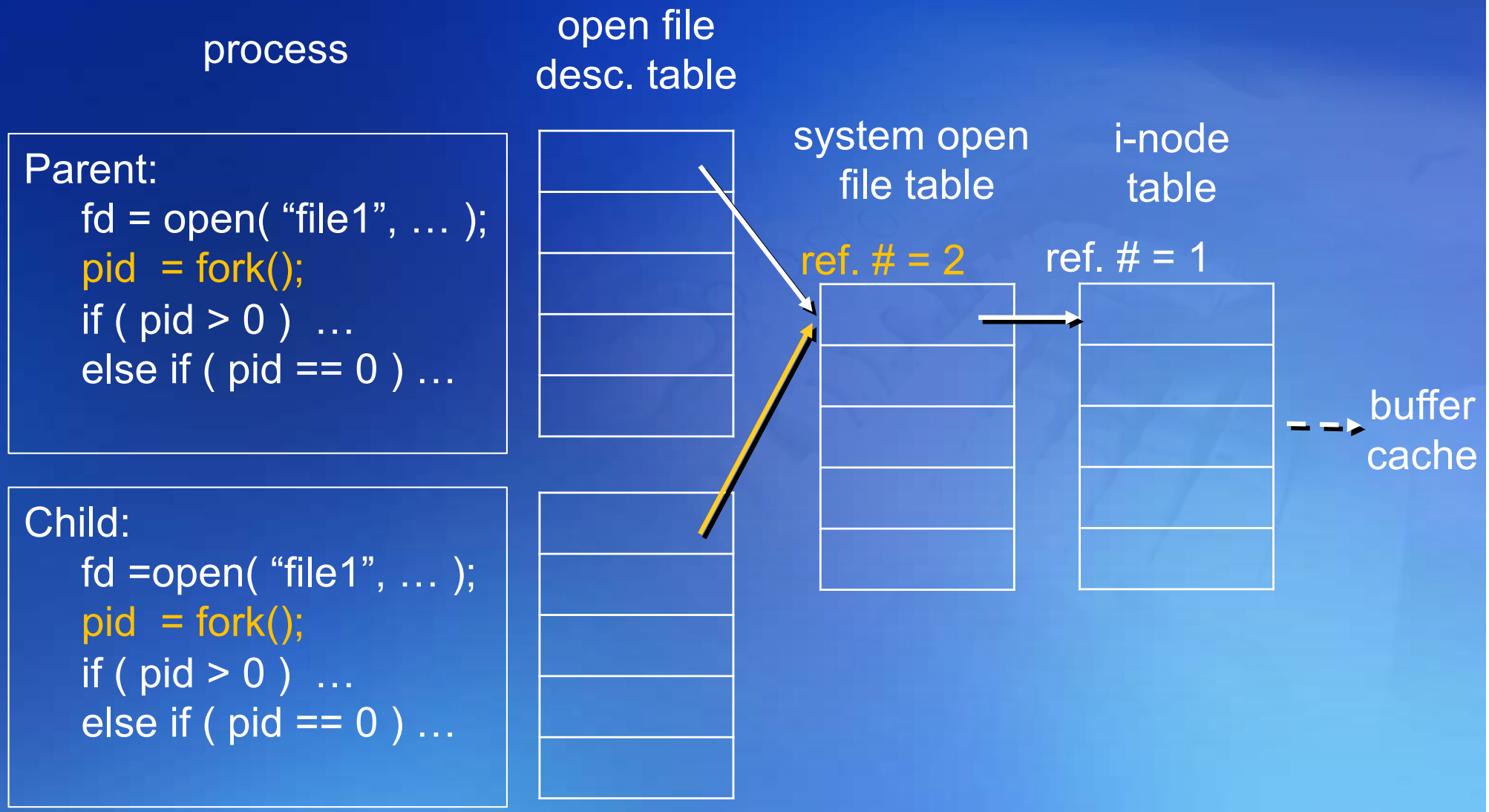


Child inherits from parents



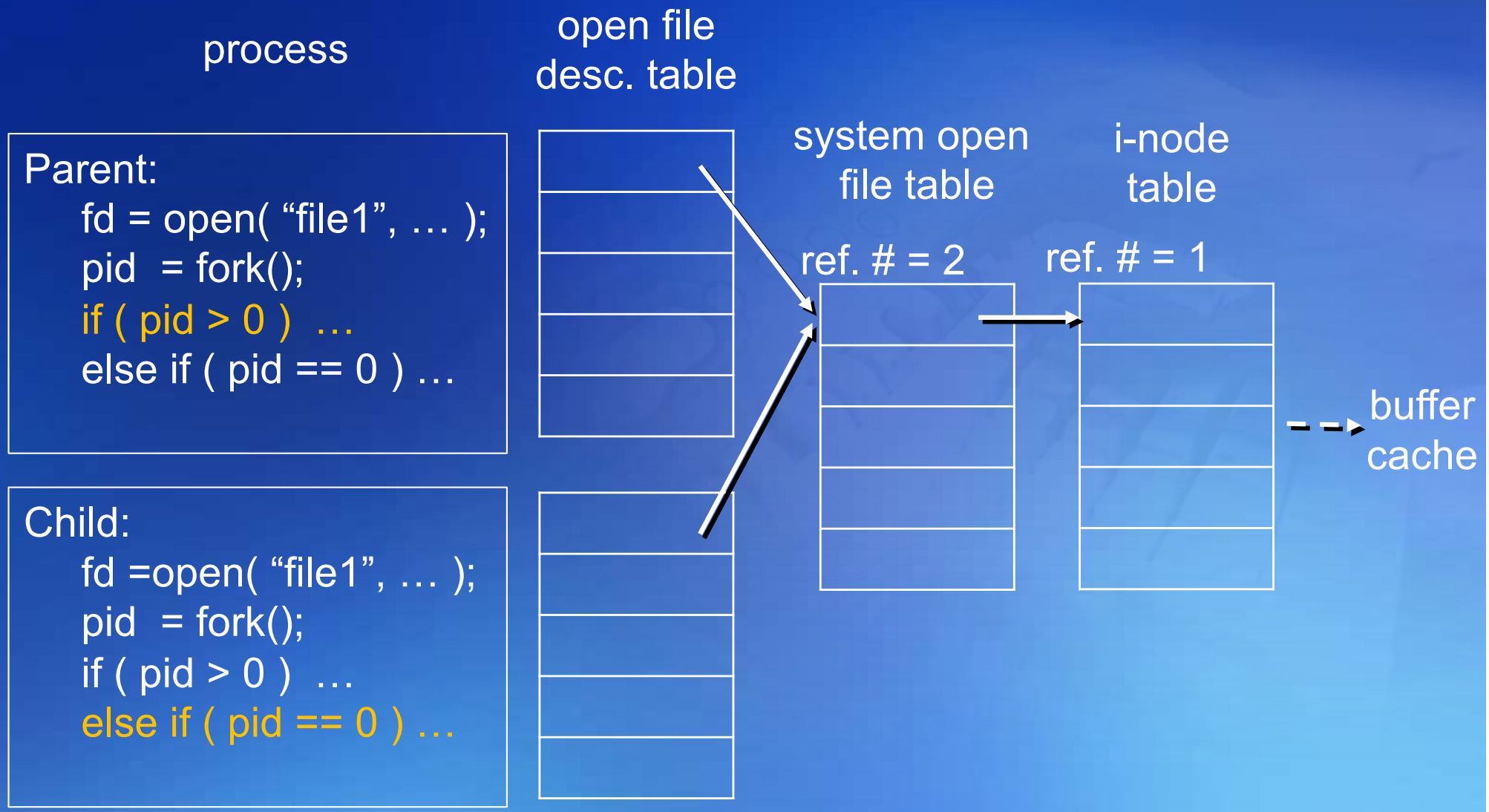
Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Child inherits from parents



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

Child inherits from parents



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

File I/O - lseek

```c
#include <unistd.h>

int main(void)
{
        char buf[100];
        ssize_t n;

        while ( (n=read( STDIN_FILENO, buf, 100 )) != 0 )
                write( STDOUT_FILENO, buf, n );
        return 0;
}
```

#include <sys/types>

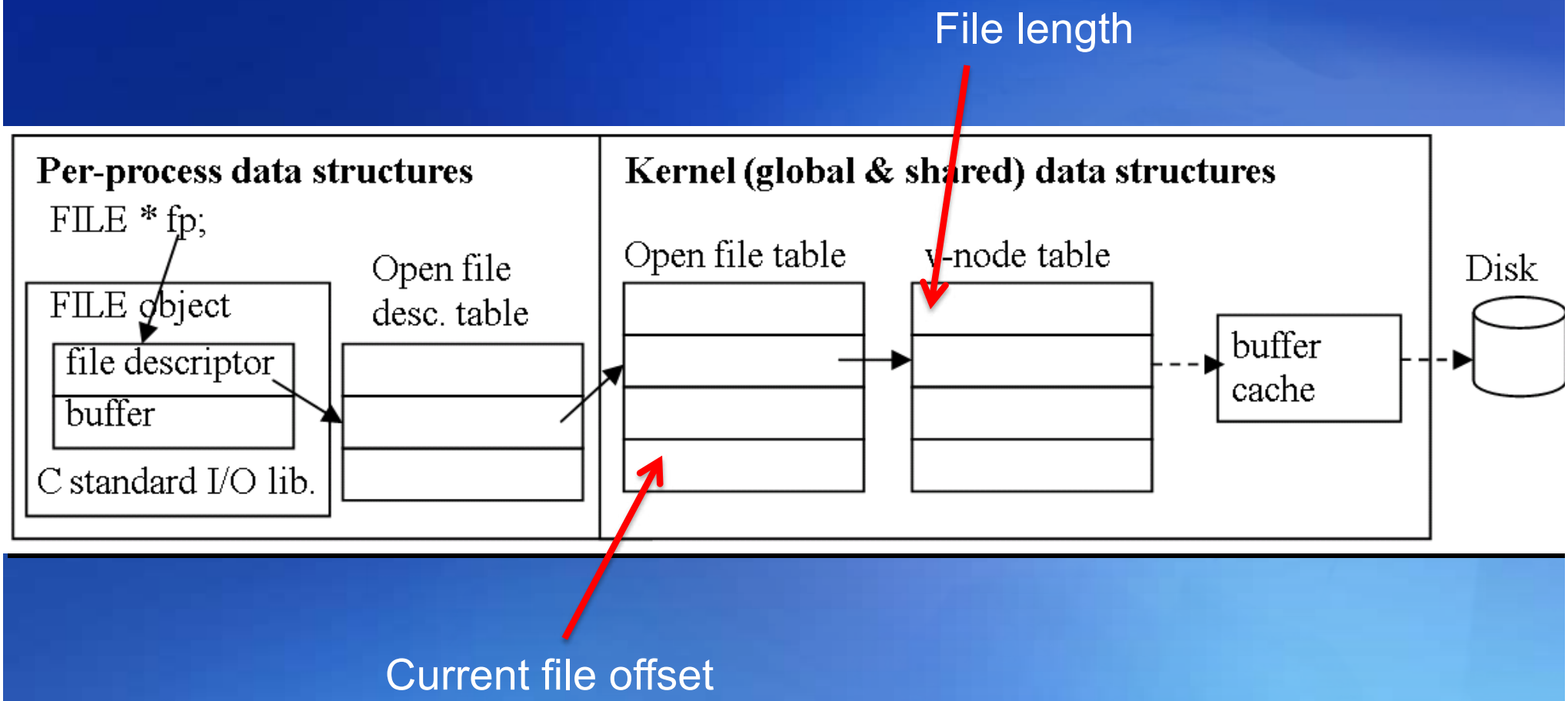
#include <unistd.h>

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**
- **Example**

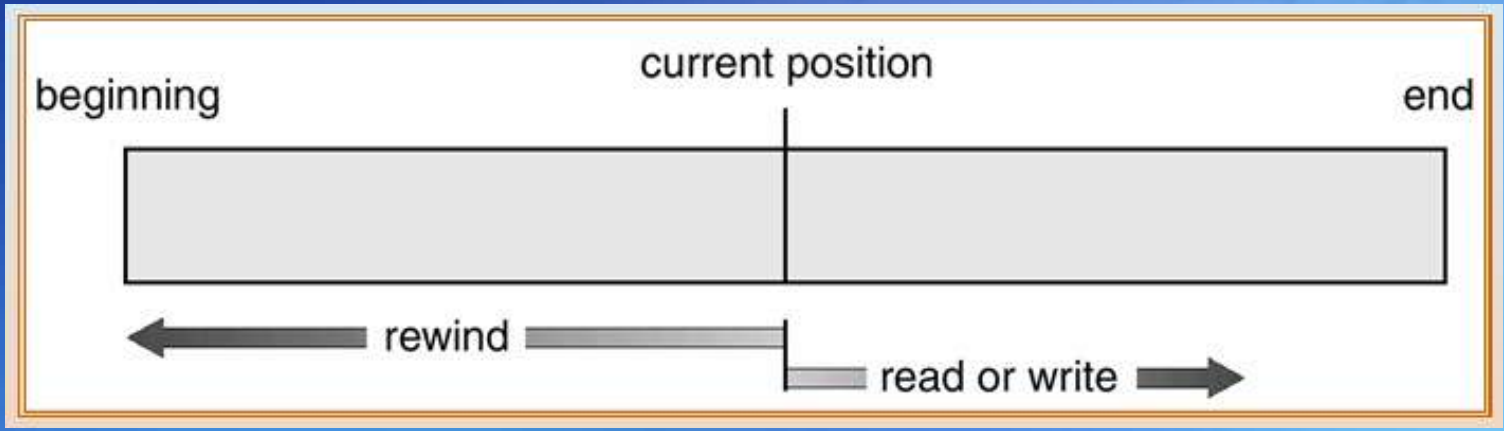
  currpos = lseek(fd, 0, SEEK_CUR)
  - EPIPE for a pipe or a FIFO
- **off_t: typedef long off_t; /* 2^{31} bytes */**
  - or typedef longlong_t off_t; /* 2^{63} bytes */
  - Negative for /dev/kmem on SVR4
- **No I/O takes place until next read or write.**

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

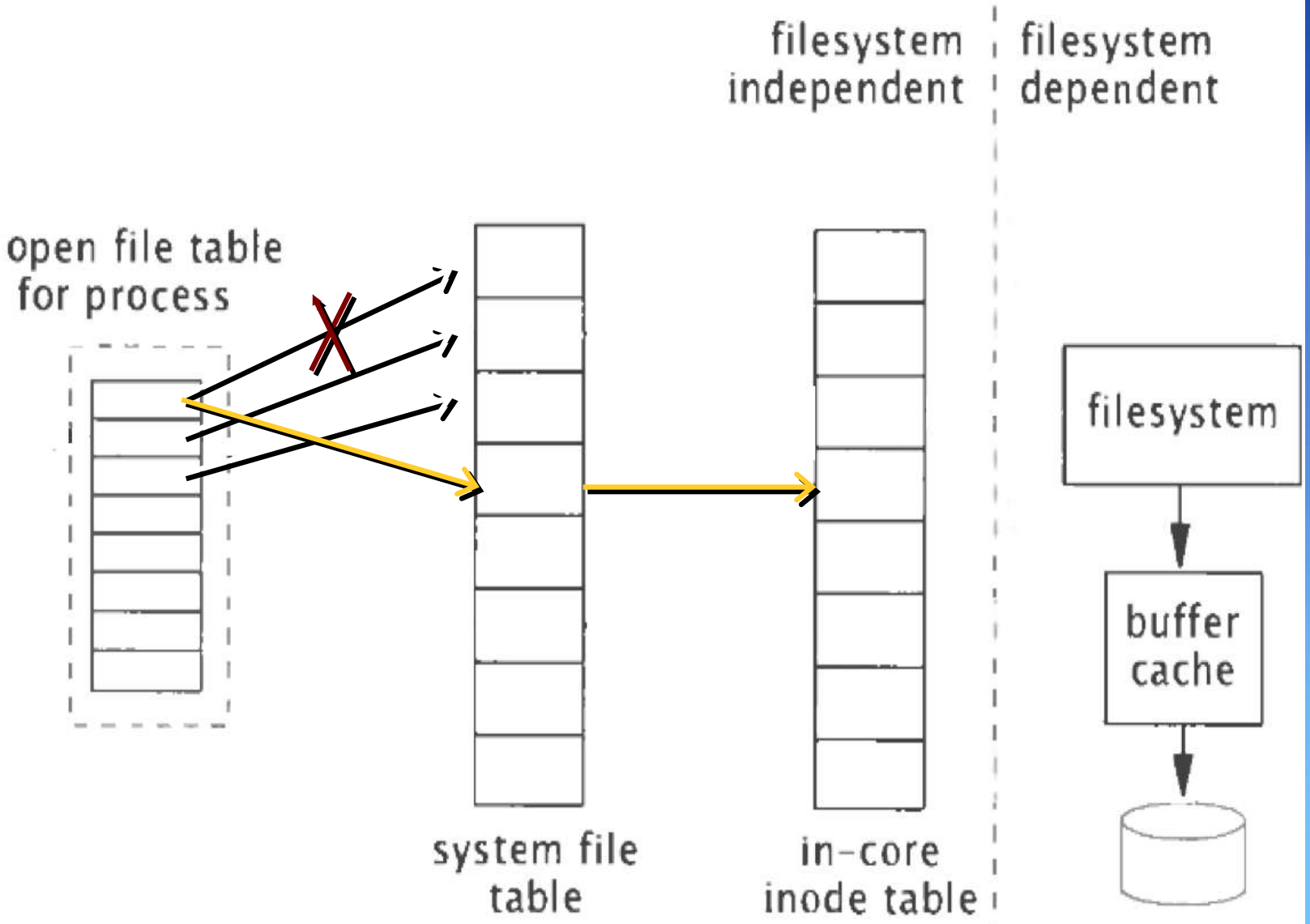


Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

Example of Seeking

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

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

```
$ ./a.out < /etc/motd
seek OK
$ cat < /etc/motd | ./a.out
cannot seek
$ ./a.out < /var/spool/cron/FIFO
cannot seek
```


filesystem independent
filesystem dependent
open file table for process
system file table
in-core inode table
filesystem
buffer cache

Example of Creating a Hole

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

char    buf1[] = "abcdefghij";
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
0000000   a   b   c   d   e   f   g   h   i   j  \0  \0  \0  \0  \0  \0
0000020  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0040000   A   B   C   D   E   F   G   H   I   J
0040012
```

$ cat < file.hole > file.nohole

```
$ 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

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

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

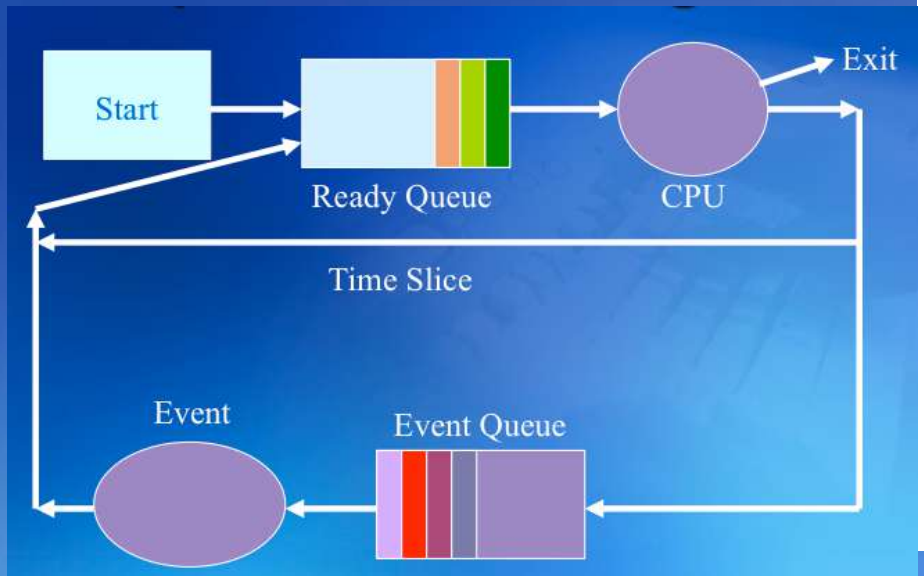
#include <unistd.h>

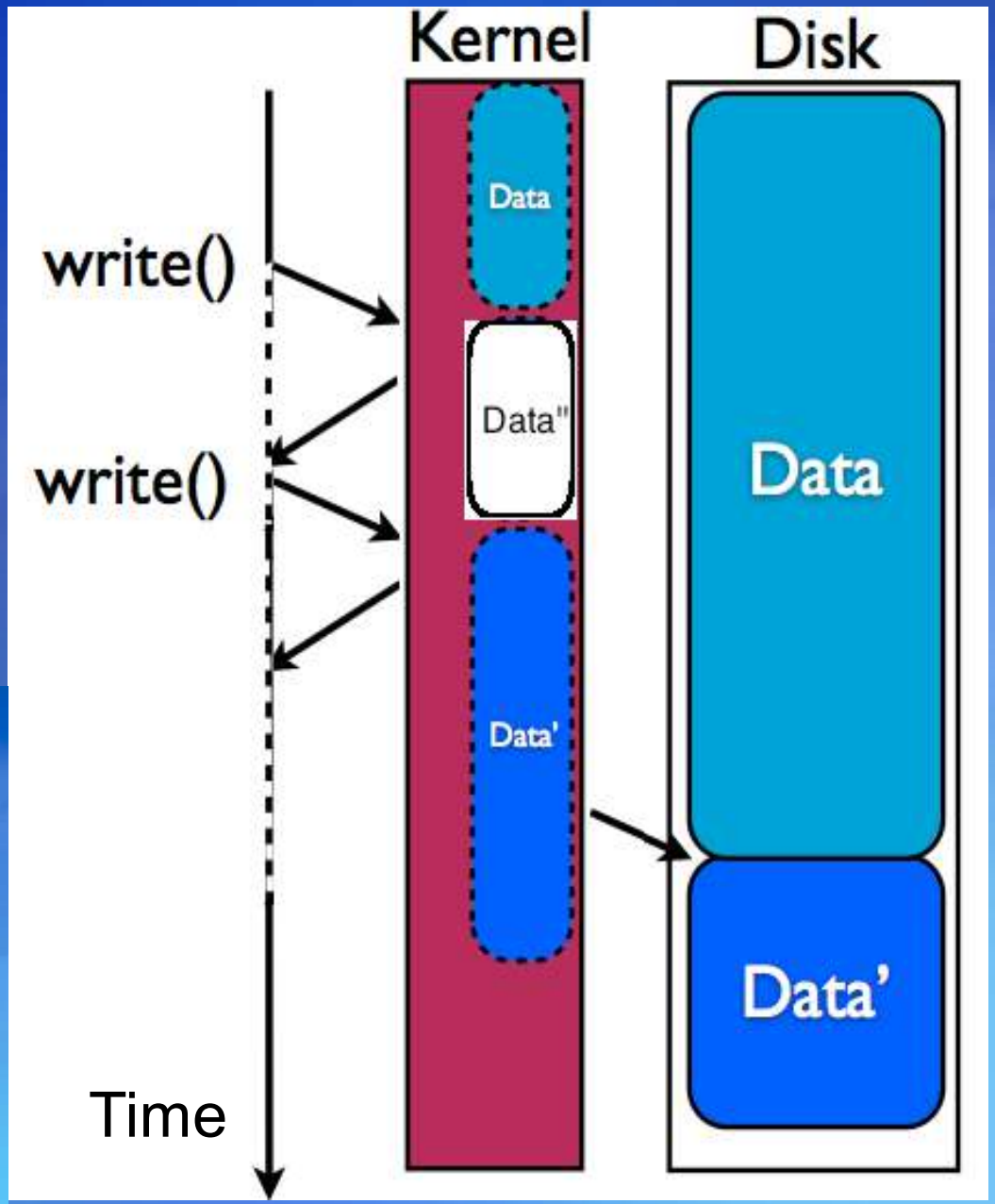
ssize_t write(int filedes, const void *buf, size_t nbytes);

- 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

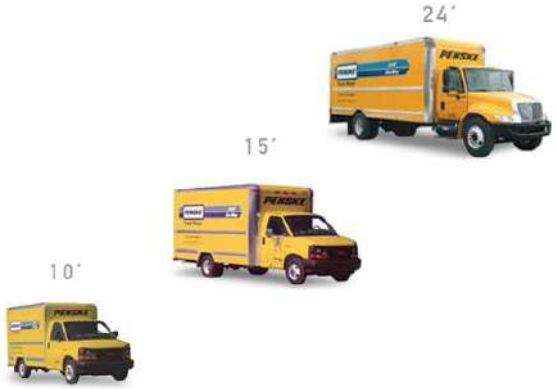
- Keep the data buffered so that multiple writes do not require multiple disk accesses.
- The buffer is queued for writing to disk at some later time.





I/O Efficiency

How do you move from home to dorm?



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

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);
}
```

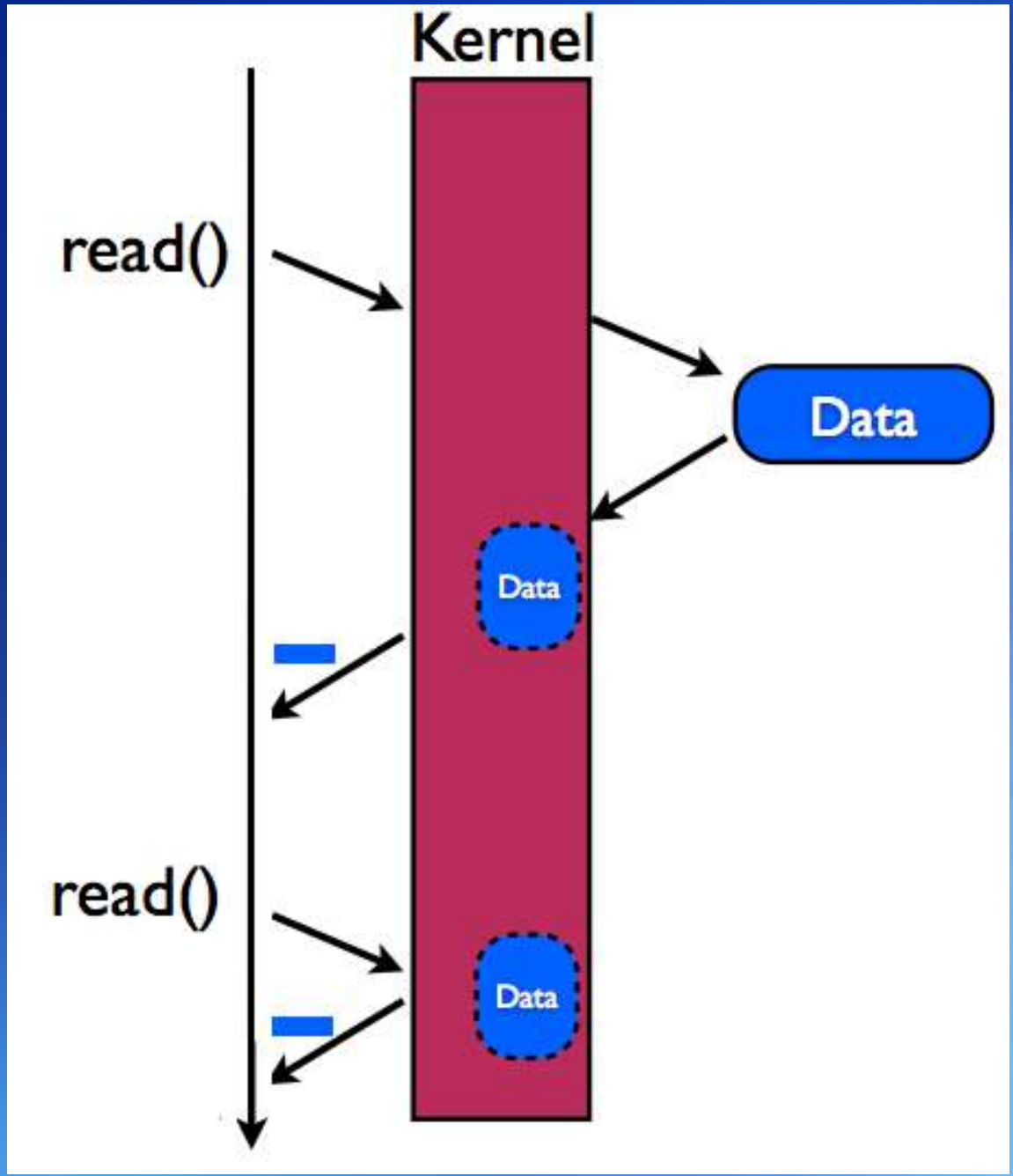
File I/O - Efficiency

- 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 BUFFSIZE?
- Try I/O redirection in reading an 103Mb file

size of disk block **read-ahead**

Buffer size	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

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



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

Effect of Read-ahead

BUFFSIZE	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)	#loops
1	124.89	161.65	288.64	103,316,352
2	63.10	80.96	145.81	51,658,#176
4	31.84	40.00	72.75	25,829,088
8	15.17	21.01	36.85	12,914,544
16	7.86	10.27	18.76	6,457,272
32	4.13	5.01	9.76	3,228,636
64	2.11	2.48	6.76	1,614,318
128	1.01	1.27	6.82	807,159
256	0.56	0.62	6.80	403,579
512	0.27	0.41	7.03	201,789
1,024	0.17	0.23	7.84	100,894

User CPU Time + Sys CPU Time <= Clock Time

- Difference is quite small when the buffer size is small
- The difference increases when the buffer size comes to 128

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.
- lseek simply adjusts current file offset in file table entry
- To lseek 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");
```

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

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.

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

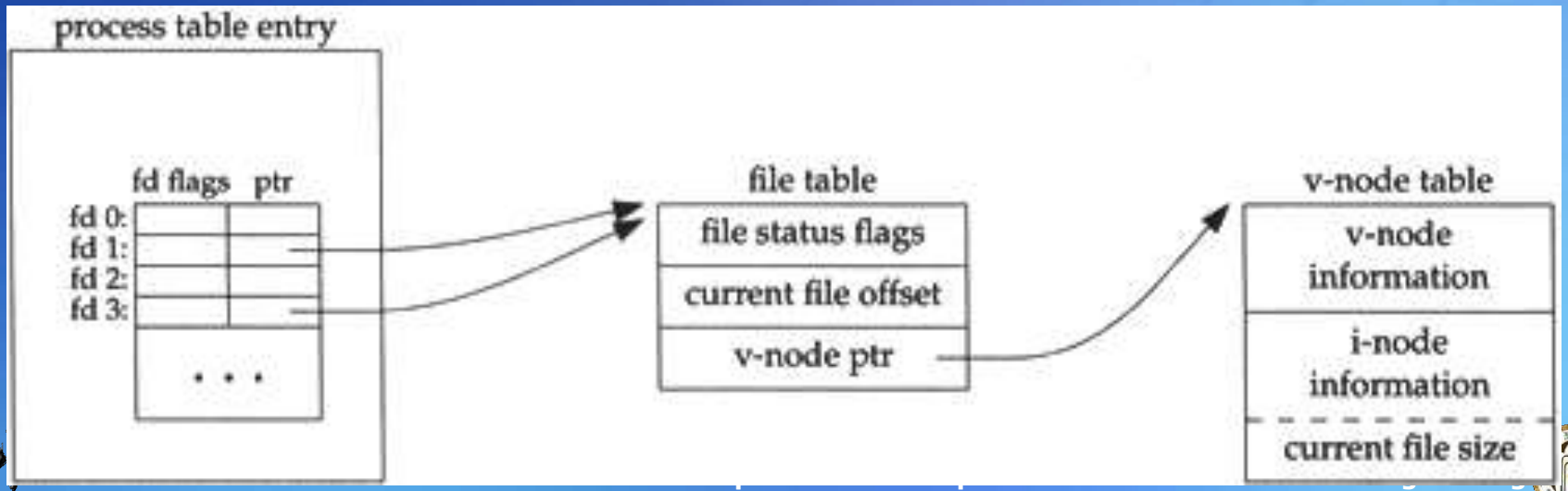
Error Recovery

- 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**

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);

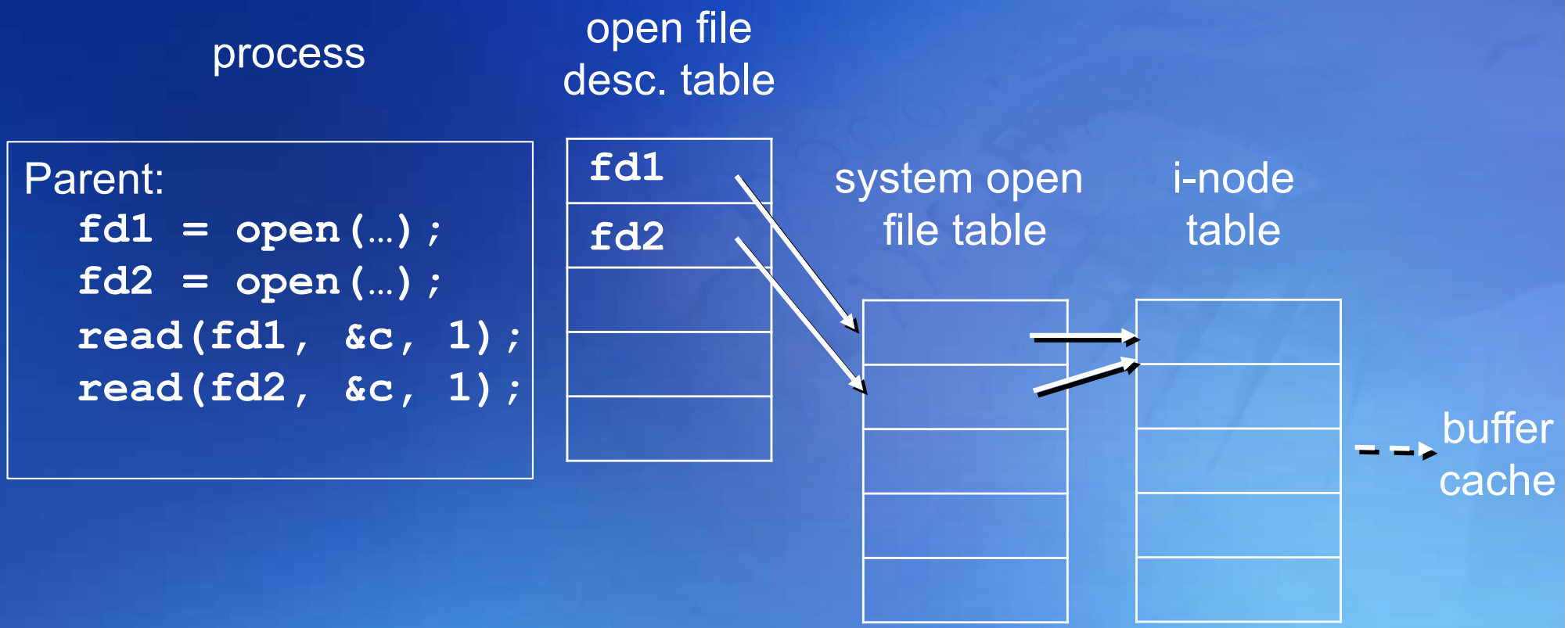


Suppose that `foobar.txt` consists of the 6 ASCII characters "`foobar`". Then what is the output of the following program?

```
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**

Explanation



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

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'

Explanation

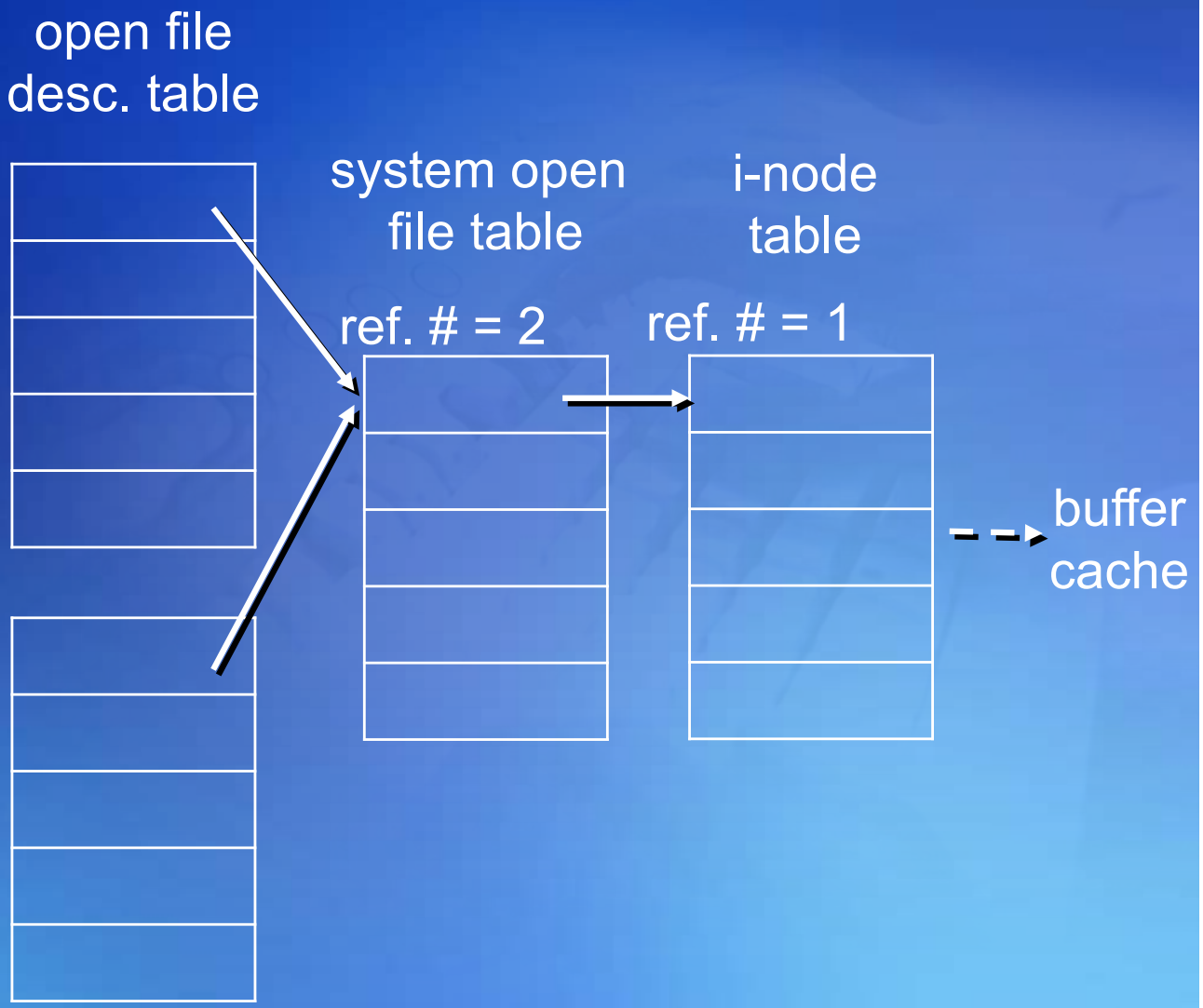
process

Parent:

```
fd = open(…)
if(fork() == 0){
 read(fd, &c, 1);
 exit(0);
}
wait(NULL);
read(fd, &c, 1);
```

Child:

```
fd = open(…)
if(fork() == 0){
 read(fd, &c, 1);
 exit(0);
}
wait(NULL);
read(fd, &c, 1);
```



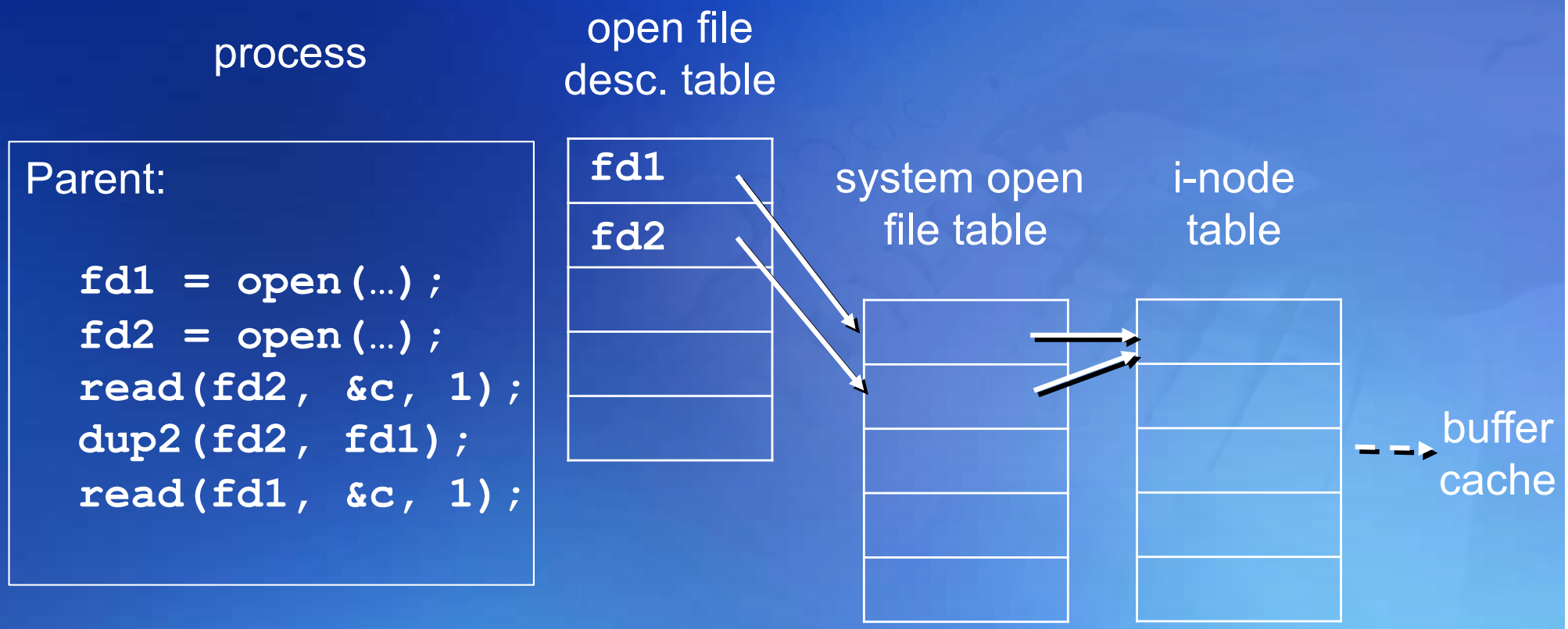
Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

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'.

Explanation



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

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**.

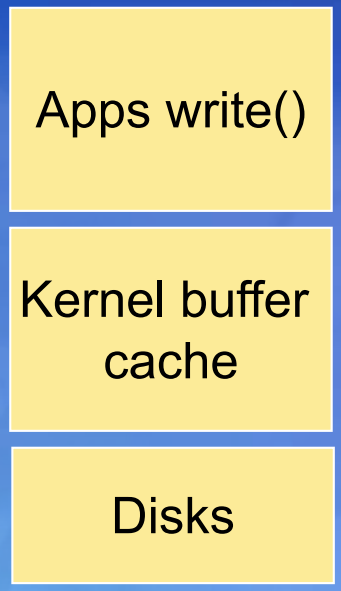
What is the output of “a.out file1 file2”?

```
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.

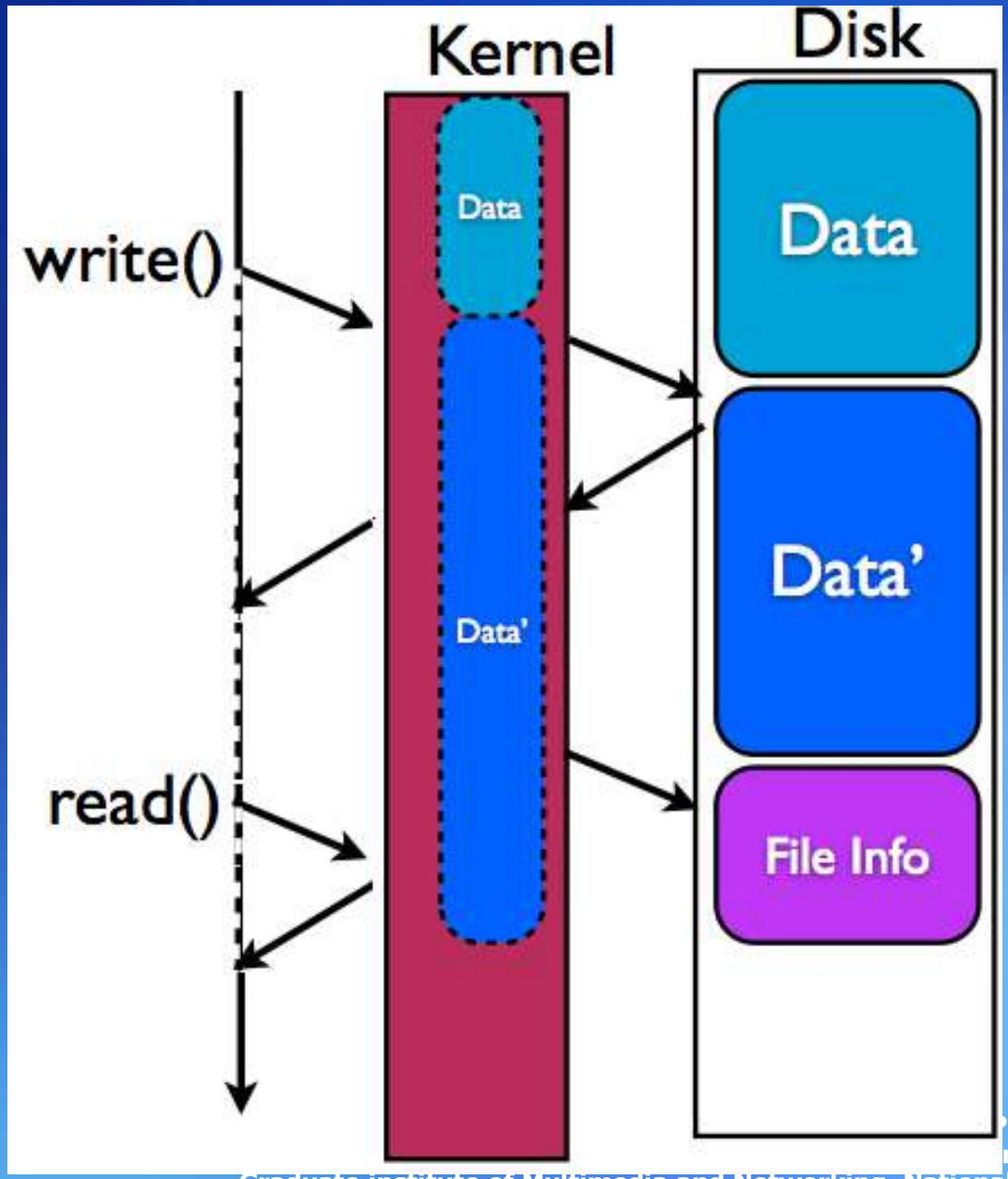
```
#include <unistd.h>
int fsync(int filedes);  // data + attr sync
int fdatasync(int filedes);  // data sync
void sync();
    // 1. queues all kernel modified block buffers
    //    for writing and returns immediately
    // 2. called by daemon update & command sync
```



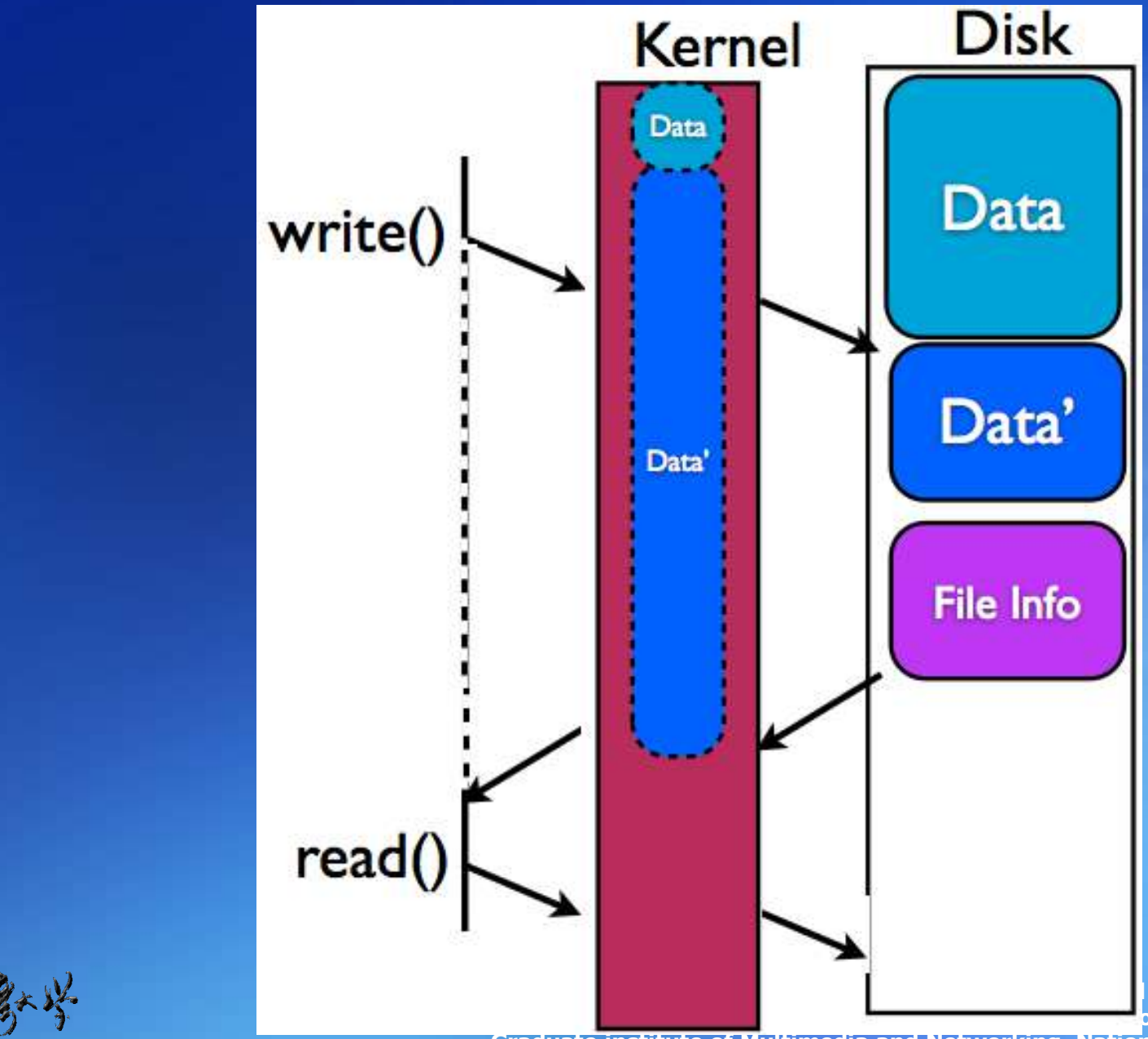
Comparison with open()'s options

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.

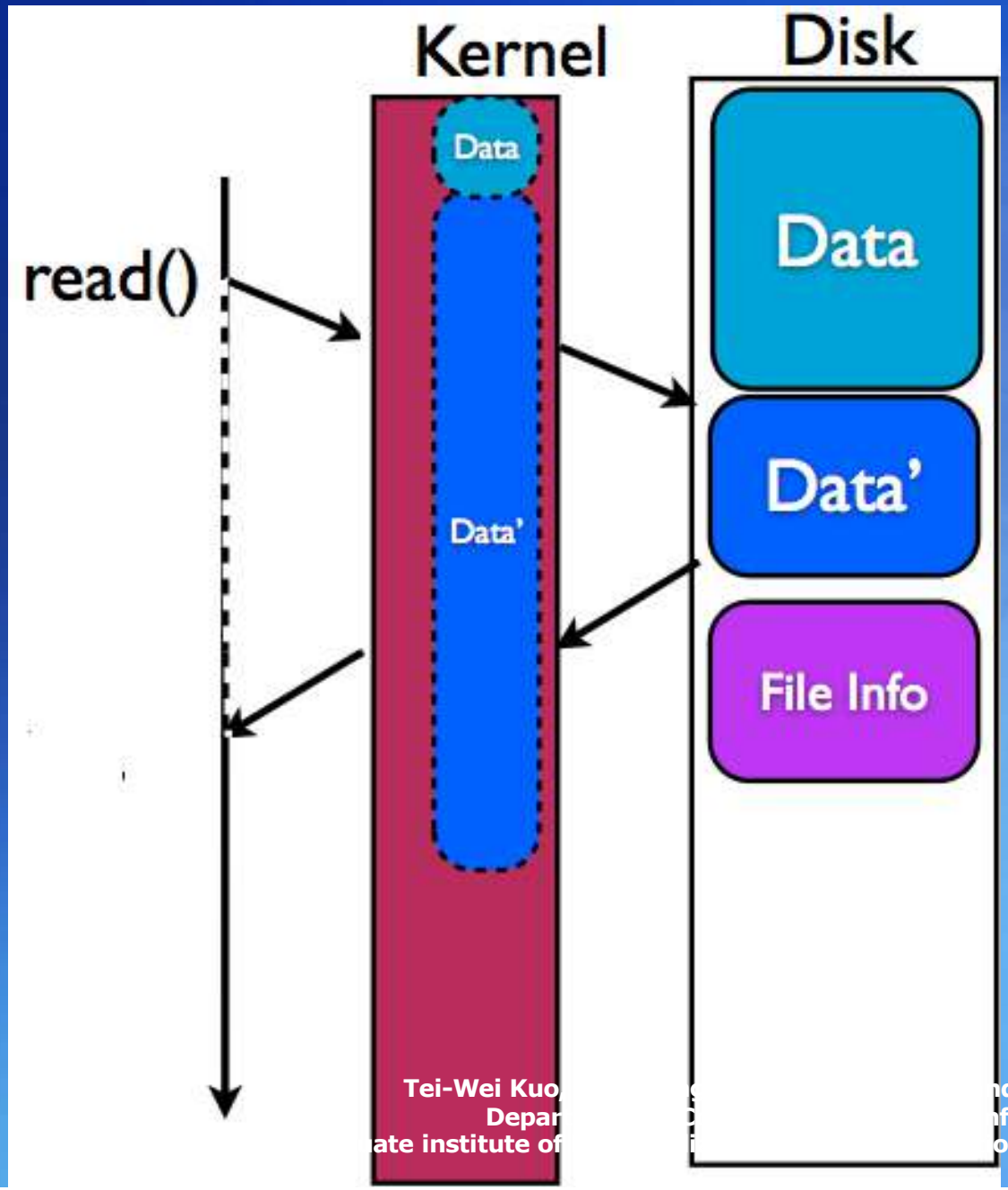
Open with O_DSYNC



Open with O_SYNC



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)
	read time from Figure 3.5 for BUFFSIZE = 4,096 (write to /dev/null)	0.03	0.16	6.86
1	normal write to disk file	0.02	0.30	6.87
2	write to disk file with O_SYNC set	0.03	0.30	6.83
	write to disk followed by fdatasync	0.03	0.42	18.28
	write to disk followed by fsync	0.03	0.37	17.95
	write to disk with O_SYNC set followed by fsync	0.05	0.44	17.95

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 - fcntl

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

- **Changes the properties of opened files**
- **F_DUPFD: duplicate an existing file descriptor (>= 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, + procID, -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");
#if     !defined(_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
```

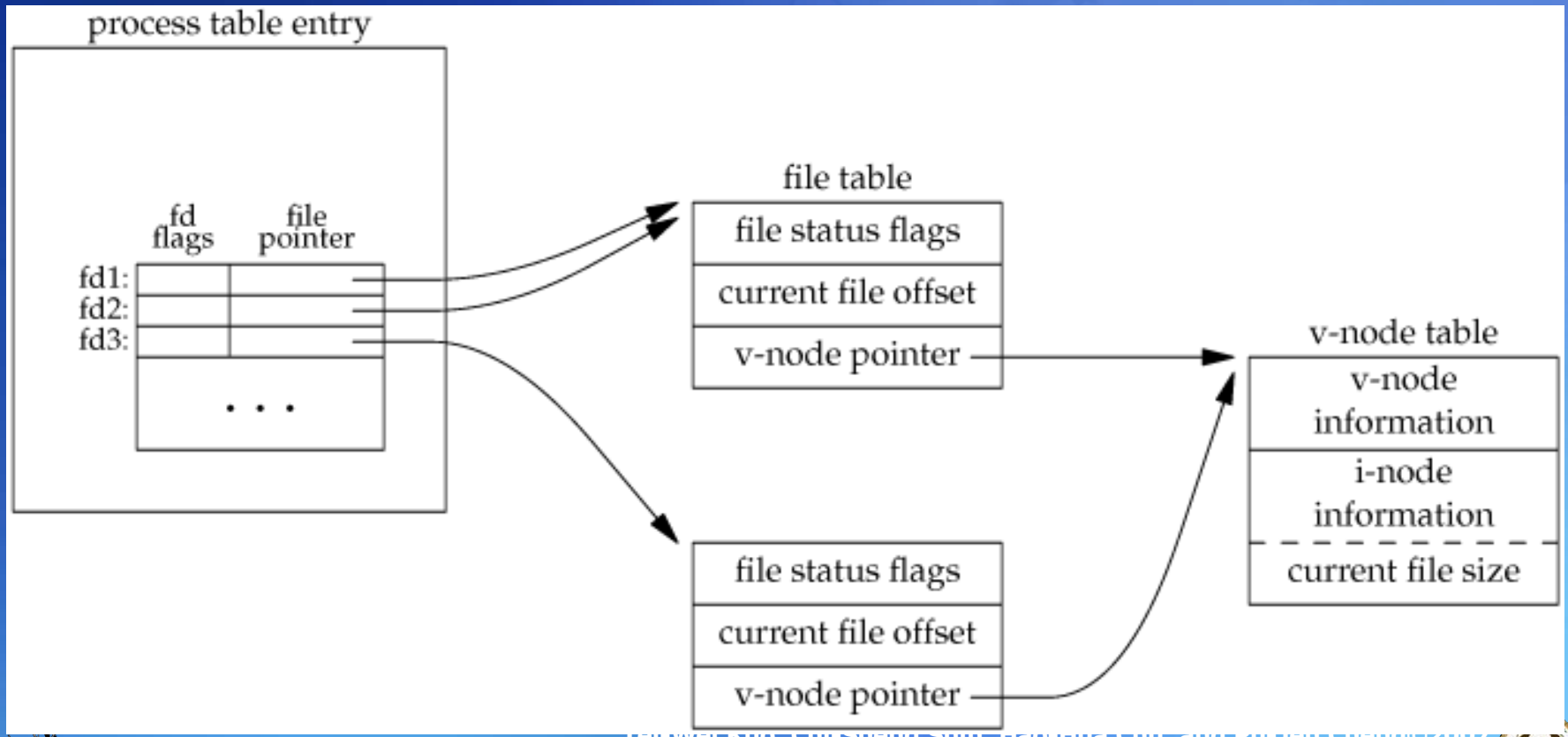
Turn on one or more flags

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/df/*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

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2008
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University