Systems Programming

Prof. Shih-Wei Li

Department of Computer Science and Information Engineering
National Taiwan University

What we will cover today?

- File I/Os
- Also, you can find the source code from the textbook here:
  - http://www.apuebook.com/src.3e.tar.gz

Disk Structure

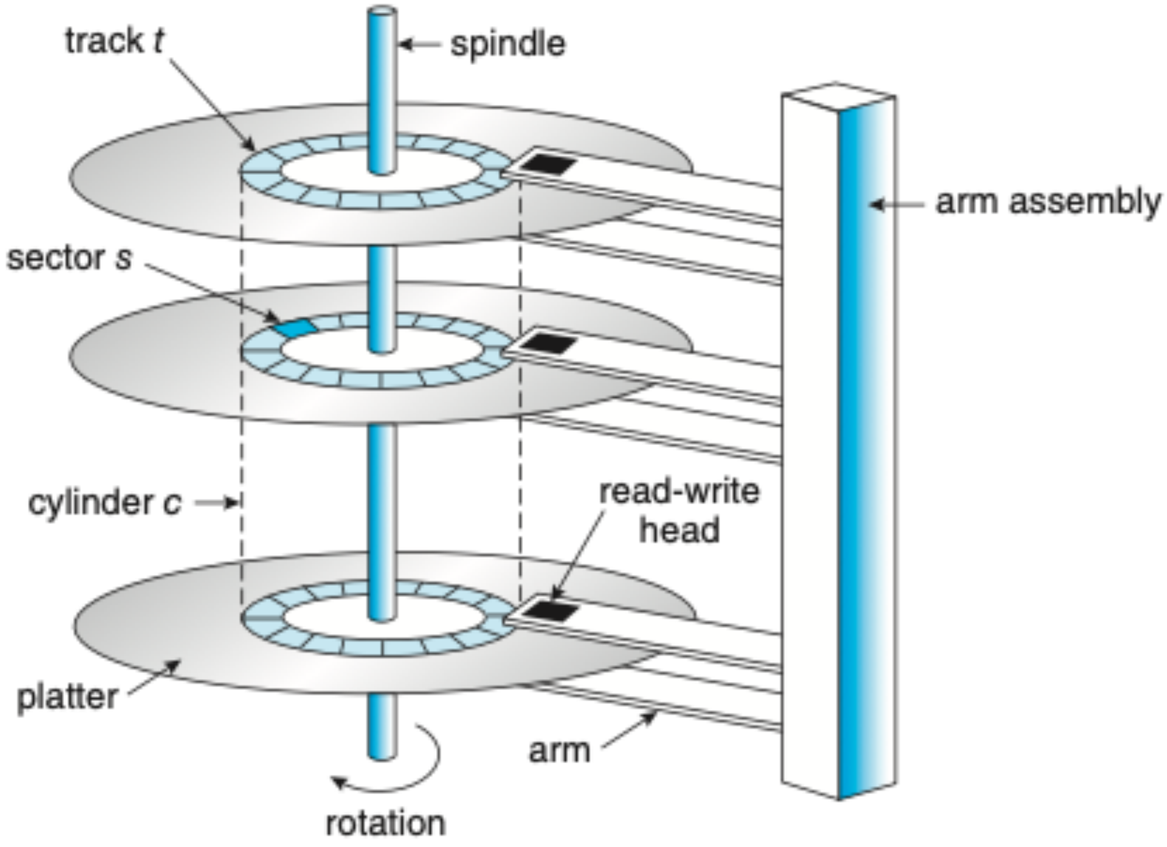


Figure 11.1 HDD moving-head disk mechanism.

File I/O in Unix

- File I/Os are the operations involved to use a file
- File I/Os in Unix are categorized into *unbuffered I/Os* and *(buffered) standard I/Os*
  - (Buffered) Standard I/Os: functions accumulate results in intermediate buffers, not making system calls each time (e.g., fread/fwrite)
  - Unbuffered I/Os: functions invoke system calls to the kernel each time (e.g., read(), write() in Unix) -> will be discussed in the lecture

File I/O: deeper look

- What's involved when a process opens, reads, writes, .. a file? A few facts
  - The following is stored on the disk:
    - The actual *file contents*
    - The file *metadata*: file permissions and where the file is stored on the disk
- Key facts about Unix File I/O:
  - In Unix, a process must ***open*** the file first before read/write to the file
  - The exact process may open a given file multiple times
  - Many processes may access the same file at the same time

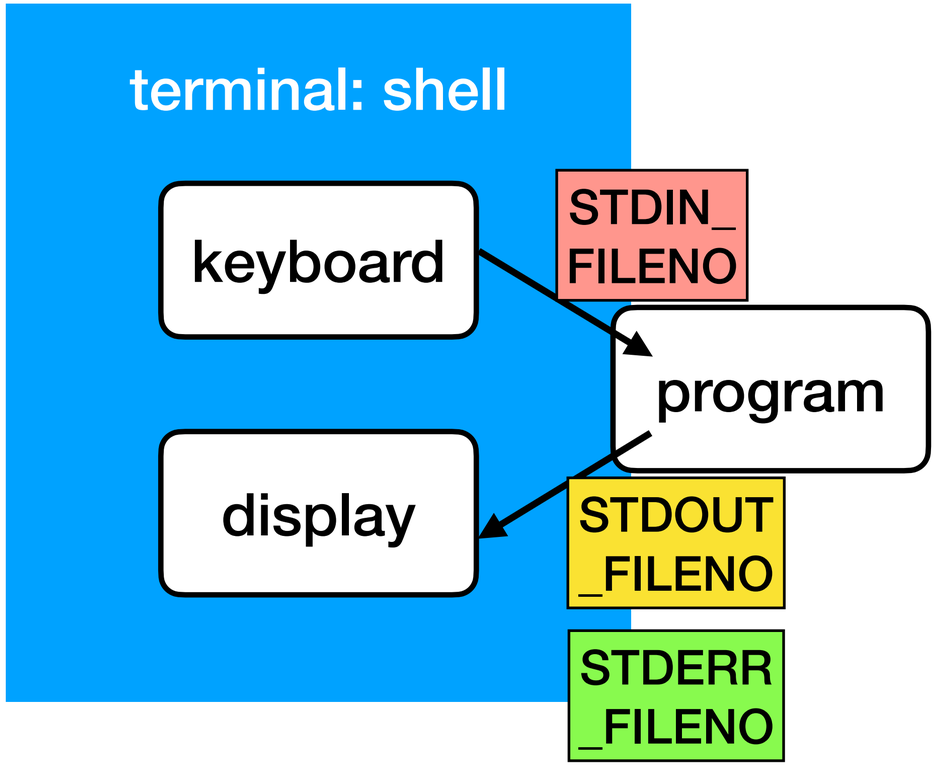
File I/O: deeper look

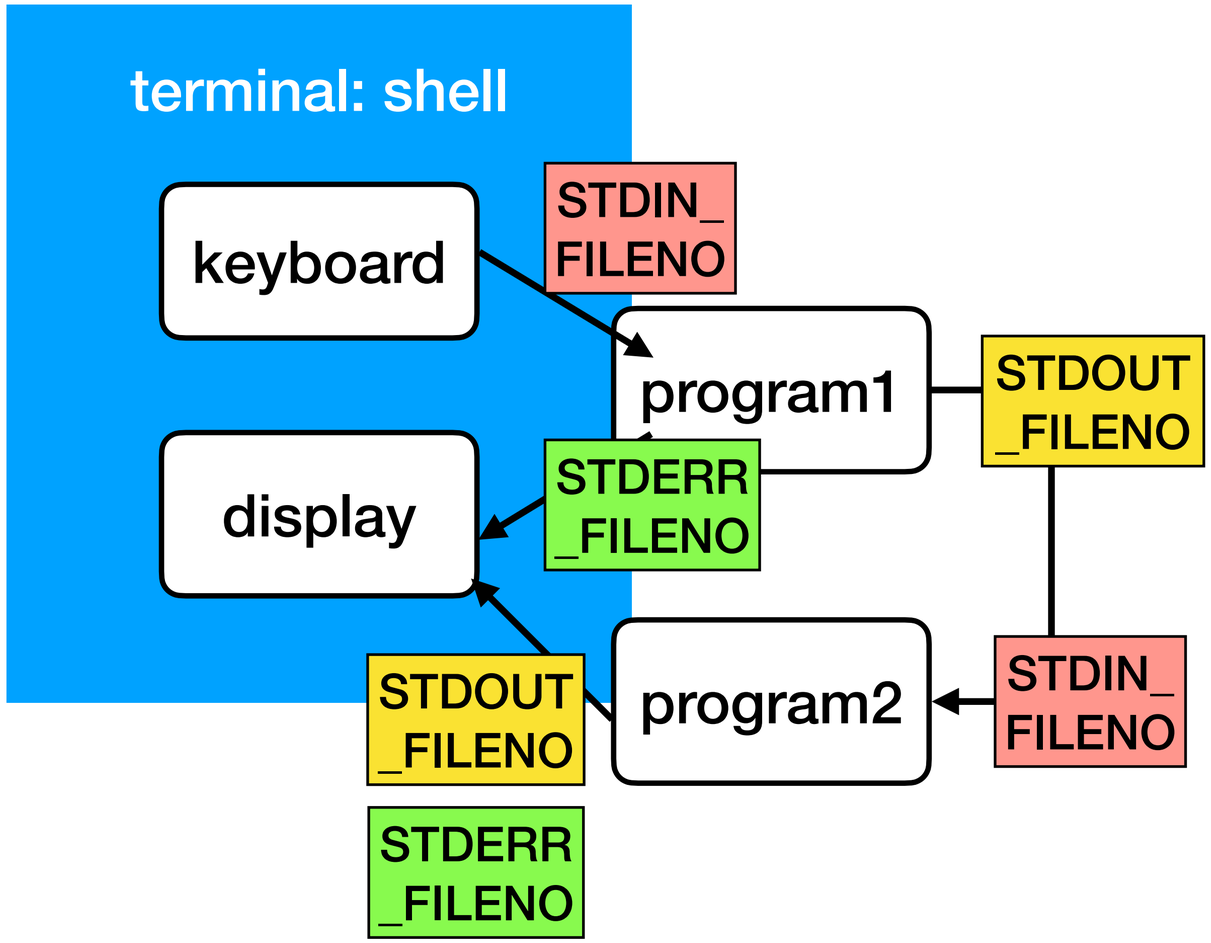
- The OS kernel keeps a record for a process' connection to open files; the process should tell the kernel which file-connection it uses
  - Intuitive solution: use the file path (ex: a.out), any issues here?

File Descriptors

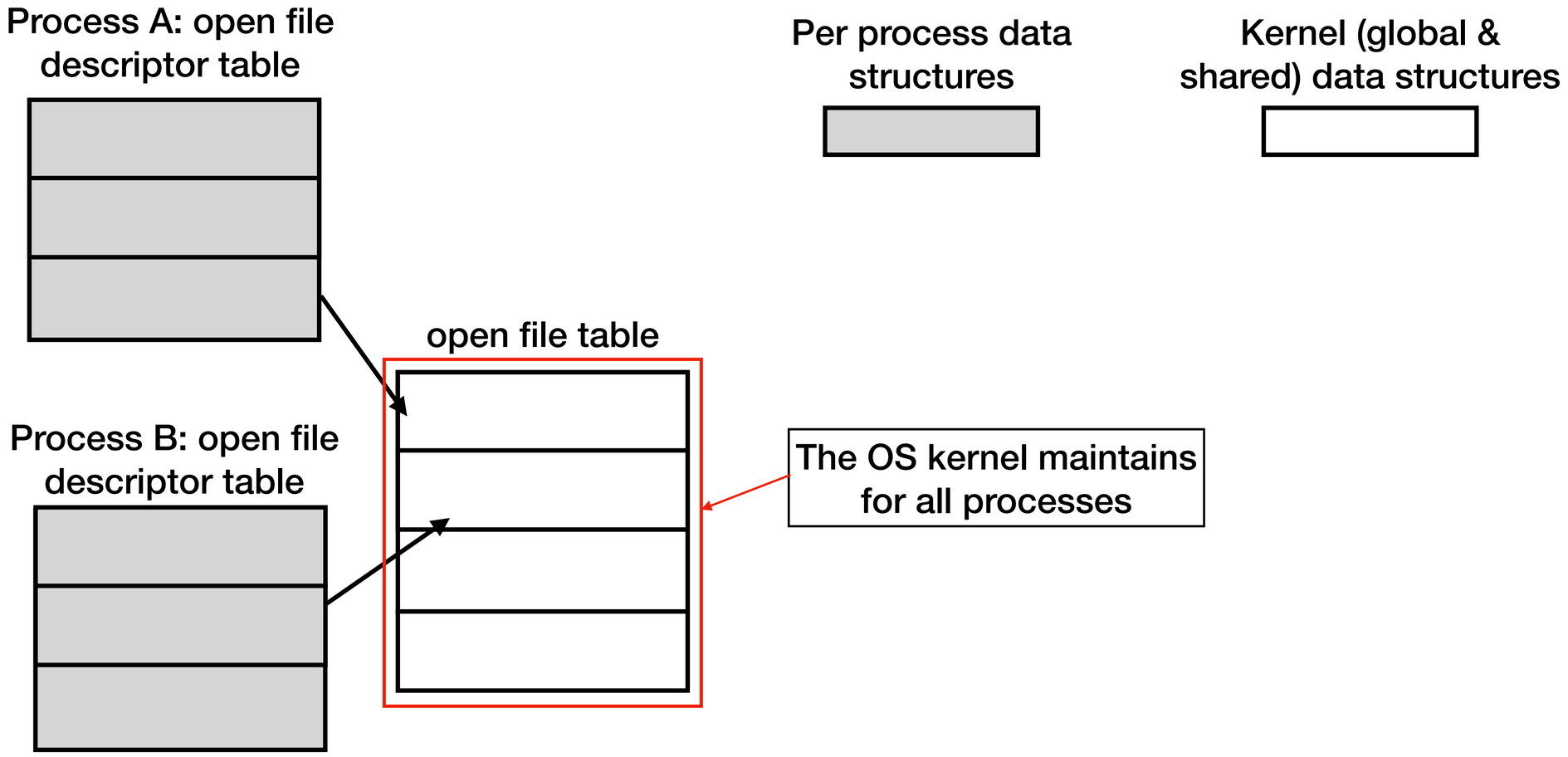
- The Unix kernel refers to an open file using a *file descriptor:*
  - A non-negative integer ranges from 0 to OPEN_MAX-1; OPEN_MAX specifies the maximum number of files that one process can open at any time
  - The kernel manages a *file descriptor table* for each process and allocates the file descriptor
  - The file descriptors are *per-process*; different processes may have the same file descriptors
- By convention, Unix associates the numbers *0*, *1*, and *2* with *standard input*, *standard output*, and *standard error,* respectively
  - The numbers 0, 1, and 2, also map to constants: STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO — defined by the POSIX.1 standard
  - The constants are defined in the <unistd.h> header

File Descriptors





File Descriptors



File Descriptors: code example

Q: name a use case example of the program.

```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;
}
```

File Descriptors: code example

Q: name a use case example of the program.

```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;
}
```

Copy standard input to standard output

File I/O: open and openat

```
#include <fcntl.h>
//For both functions, the file descriptor is returned on a successful call; and -1 is returned on an error
int open(const char *path, int oflag, … /* mode_t mode */ );
int openat(int fd, const char *path, int oflag, … /* mode_t mode */)
```

- A process can open or create a file by calling either the ***open*** or ***openat*** function
- On a successful call, the returned file descriptor will be the lowest-numbered file descriptor not currently opened by the caller process

File I/O: open

```
#include <fcntl.h>
//For both functions, the file descriptor is returned on a successful call; and -1 is returned on an error
int open(const char *path, int oflag, … /* mode_t mode */ );
int openat(int fd, const char *path, int oflag, … /* mode_t mode */)
```

- open() parameters:
  - *path* is the path of the file to open or create: could be an absolute or relative path
  - *oflag* is to specify the ***access modes*** (how the program intends to access the files, e.g., making it read-only)

File I/O: open

```
#include <fcntl.h>
//For both functions, the file descriptor is returned on a successful call; and -1 is returned on an error
int open(const char *path, int oflag, … /* mode_t mode */ );
int openat(int fd, const char *path, int oflag, … /* mode_t mode */)
```

- The *oflag* must include one (and the only one) of the access modes:
  - O_RDONLY: for read-only
  - O_WRONLY: for write-only
  - O_RDWR: for read and write
- The *oflag* can *optionally* get ***OR'd*** with the file creation and file status flags:
  - O_APPEND: append to the end of the file for each write
  - O_TRUNC: truncate the file size to 0
  - O_CREAT: create if the file does not exist
  - O_NONBLOCK: for non-blocking mode
  - O_SYNC, O_DSYNC, O_RSYNC: data synchronization, wait for I/O completion

File I/O: open — O_CREAT flag

```
#include <fcntl.h>

int open(const char *path, flag | O_CREAT, mode_t mode);
```

- The parameter *mode* specifies the user file permissions (omitted if not creating a new file)
- The mode applies to future access of the newly created file — the resulting mode of the file is: *mode subtracts umask*

$ umask 0x22 (root user)

$ created file: 0x777

Resulting permission: 0x755

The following constants are provided for mode

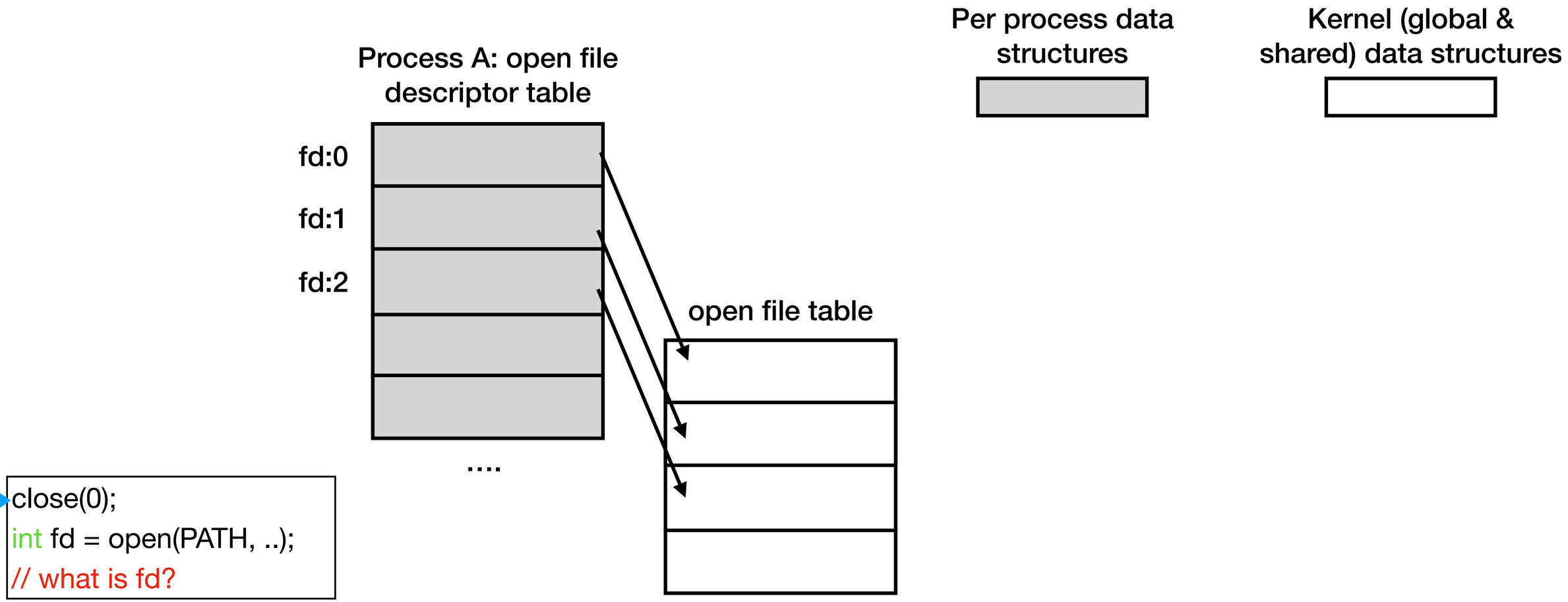
S_IRWXU	00700	user (file owner) has read, write, and execute permission
S_IRUSR	00400	user has read permission
S_IWUSR	00200	user has write permission
S_IXUSR	00100	user has execute permission
S_IRWXG	00070	group has read, write, and execute permission
S_IRGRP	00040	group has read permission
S_IWGRP	00020	group has write permission
S_IXGRP	00010	group has execute permission
S_IRWXO	00007	others have read, write, and execute permission
S_IROTH	00004	others have read permission
S_IWOTH	00002	others have write permission
S_IXOTH	00001	others have execute permission

File I/O: close

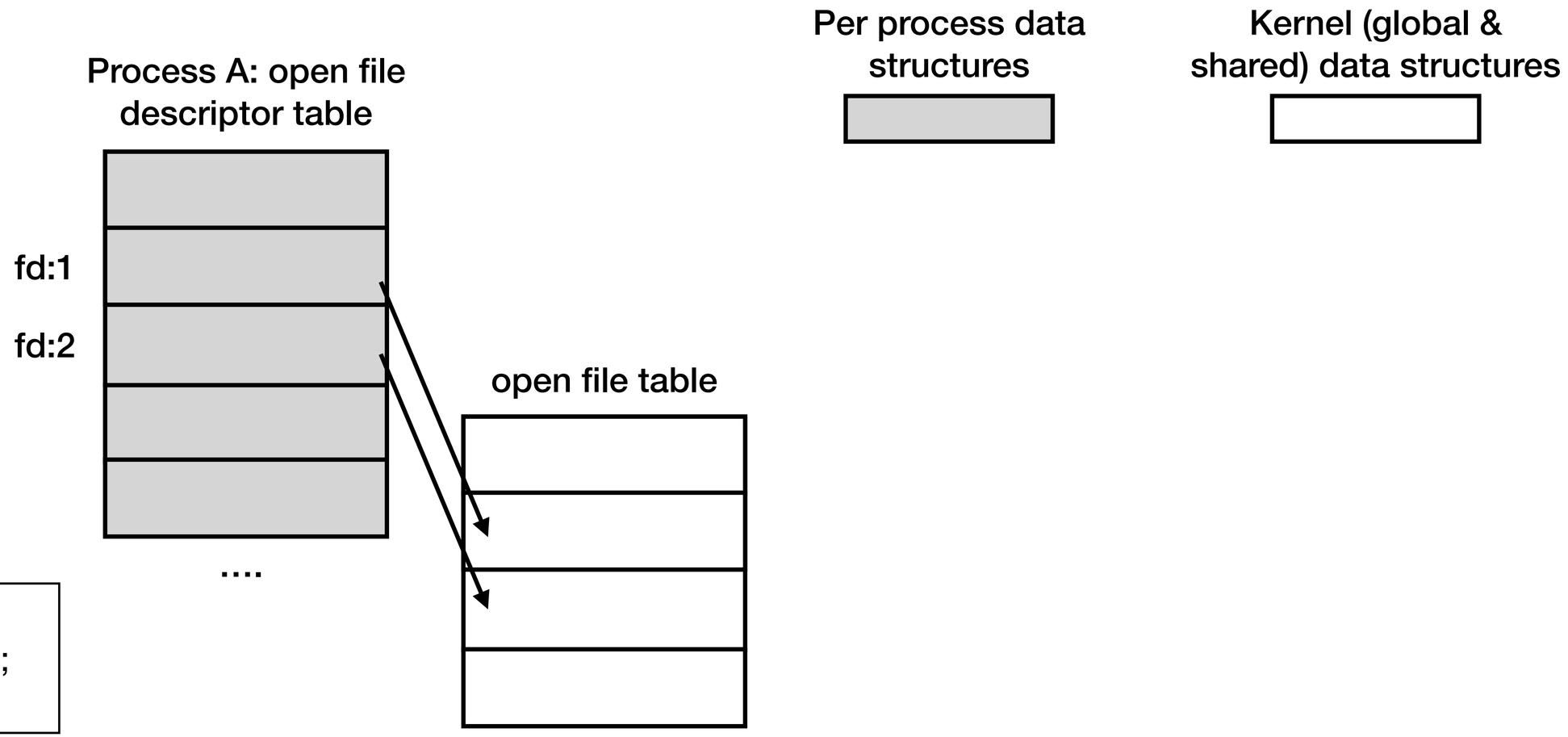
```
#include <fcntl.h>
//0 is returned on a successful call; and -1 is returned on an error
int close(int fd);
```

- An open file is closed by calling the ***close*** function
- When a process terminates, all of its open files are closed automatically by the kernel
  - Many programs take advantage of this fact and do not explicitly close open files
  - Closing a file descriptor releases any record locks on that file (will discuss more about this when we introduce file record locking)

File I/O: example

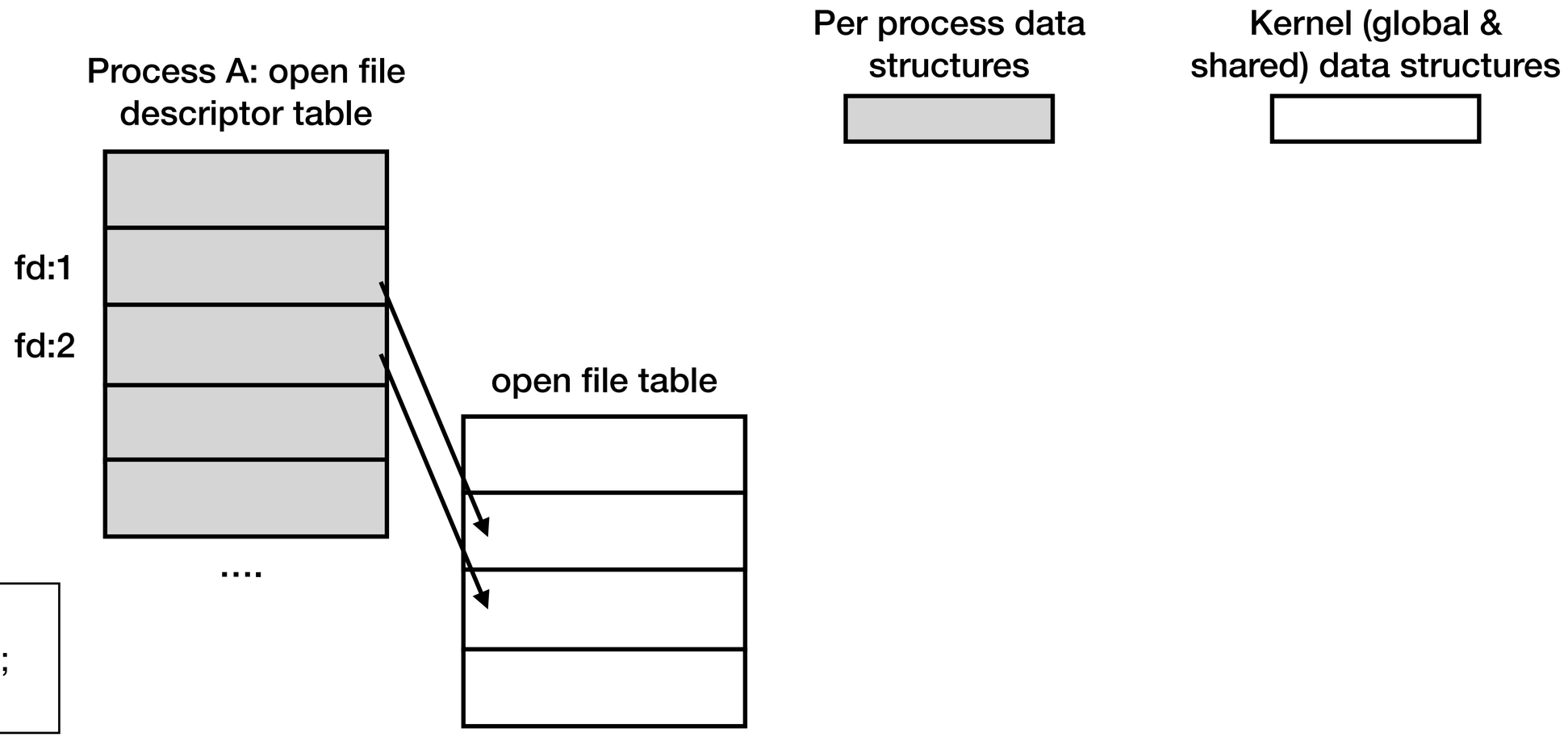


File I/O: example




```
close(0);
int fd = open(PATH, ..);
// what is fd?
```

File I/O: example




```
close(0);
int fd = open(PATH, ..);
// what is fd?
```

File I/O: example

```
#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);
}
```

What is the output of the program?

Hint: open always returns the lowest unopened descriptor

File I/O: openat

```
#include <fcntl.h>
//For both functions, the file descriptor is returned on a successful call; and -1 is returned on an error
int open(const char *path, int oflag, … /* mode_t mode */ );
int openat(int fd, const char *path, int oflag, … /* mode_t mode */)
```

- openat() operates exactly the same way as *open()* except that the ***fd*** parameter is used in conjunction with the ***path*** argument:
  - If path is absolute, the parameter fd is ignored
  - If path is relative:
    - If fd is the special value ***AT_FDCWD***, then path is interpreted relative to the current working directory of the calling process
    - If fd is referred to a given directory, then path is interpreted relative to the directory respective to the directory

File I/O: openat

```
#include <fcntl.h>
//For both functions, the file descriptor is returned on a successful call; and -1 is returned on an error
int open(const char *path, int oflag, … /* mode_t mode */ );
int openat(int fd, const char *path, int oflag, … /* mode_t mode */)
```

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

is equivalent to

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

File I/O: openat

```
#include <fcntl.h>
//For both functions, the file descriptor is returned on a successful call; and -1 is returned on an error
int open(const char *path, int oflag, … /* mode_t mode */ );
int openat(int fd, const char *path, int oflag, … /* mode_t mode */)
```

- openat() supports file open in paths relative to a pre-opened fd:
  - open() supports file open relative to the current working directory
  - openat() supports file open relative to the paths derived from the file descriptor

File I/O: creat

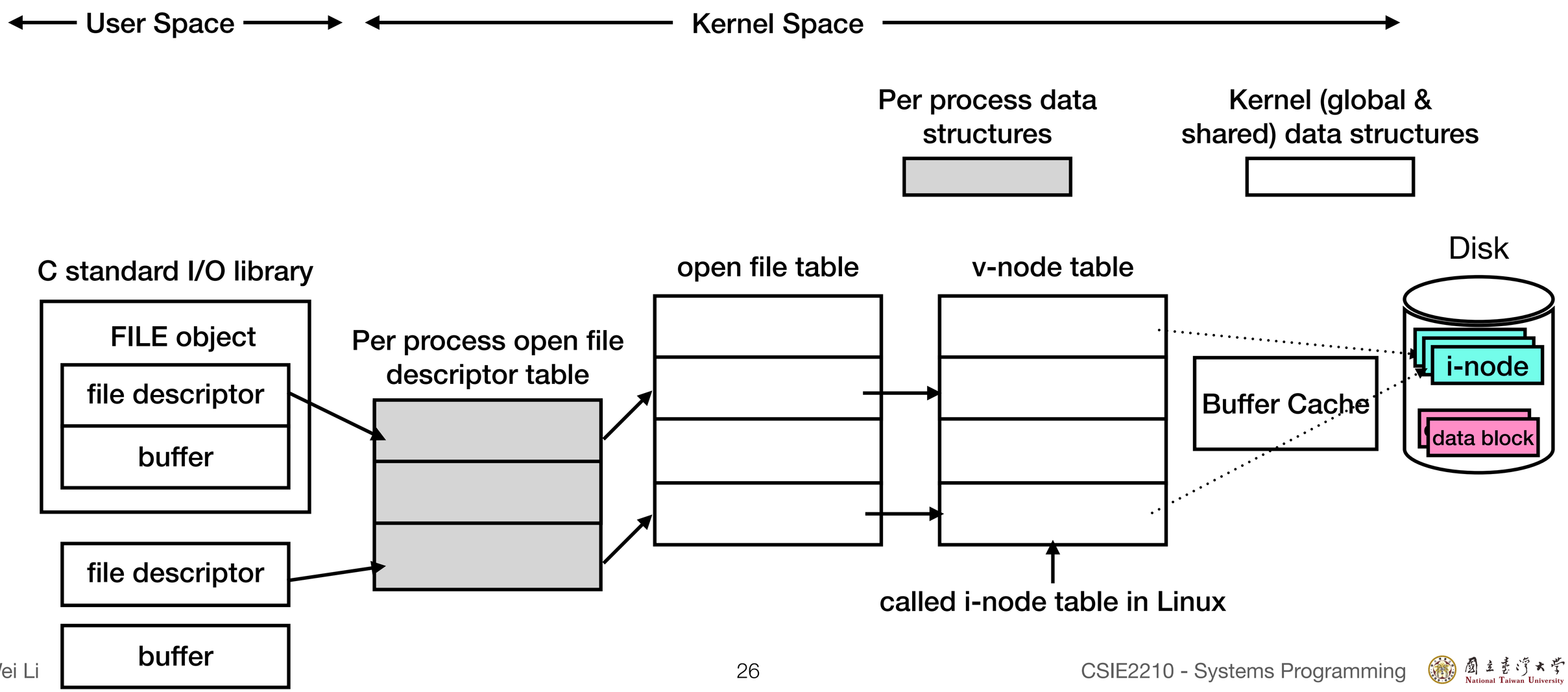
```
#include <fcntl.h>
//Returns a file descriptor on a successful call; and -1 on an error
int creat(const char *path, mode_t mode);
```

- A file is *created* and *opened* ***write-only*** by calling the ***creat*** function
- This function provides duplicated functionality as *open()*
  - *open(path, O_WRONLY | O_CREAT | OTRUNC, mode)* works the same
  - It is mostly obsolete now
- Question: what if you want to create a file and open for read-write using creat?

```
O_TRUNC
       If the file already exists and is a regular file and the
       access mode allows writing (i.e., is O_RDWR or O_WRONLY)
       it will be truncated to length 0.  If the file is a FIFO
       or terminal device file, the O_TRUNC flag is ignored.
       Otherwise, the effect of O_TRUNC is unspecified.
```

File I/O Support in Unix kernel

Unix kernel support for File I/O



Unix kernel support for File I/O

- The Unix OS kernel uses *three* data structures to represent an open file — the relationships among them determine the effect one process has on another (for file sharing)
  - Open file descriptor table (per process)
  - System open file table (shared for all open files in the system)
  - V-node table (shared for all open files in the system)
- The arrangement has existed since the early Unix versions

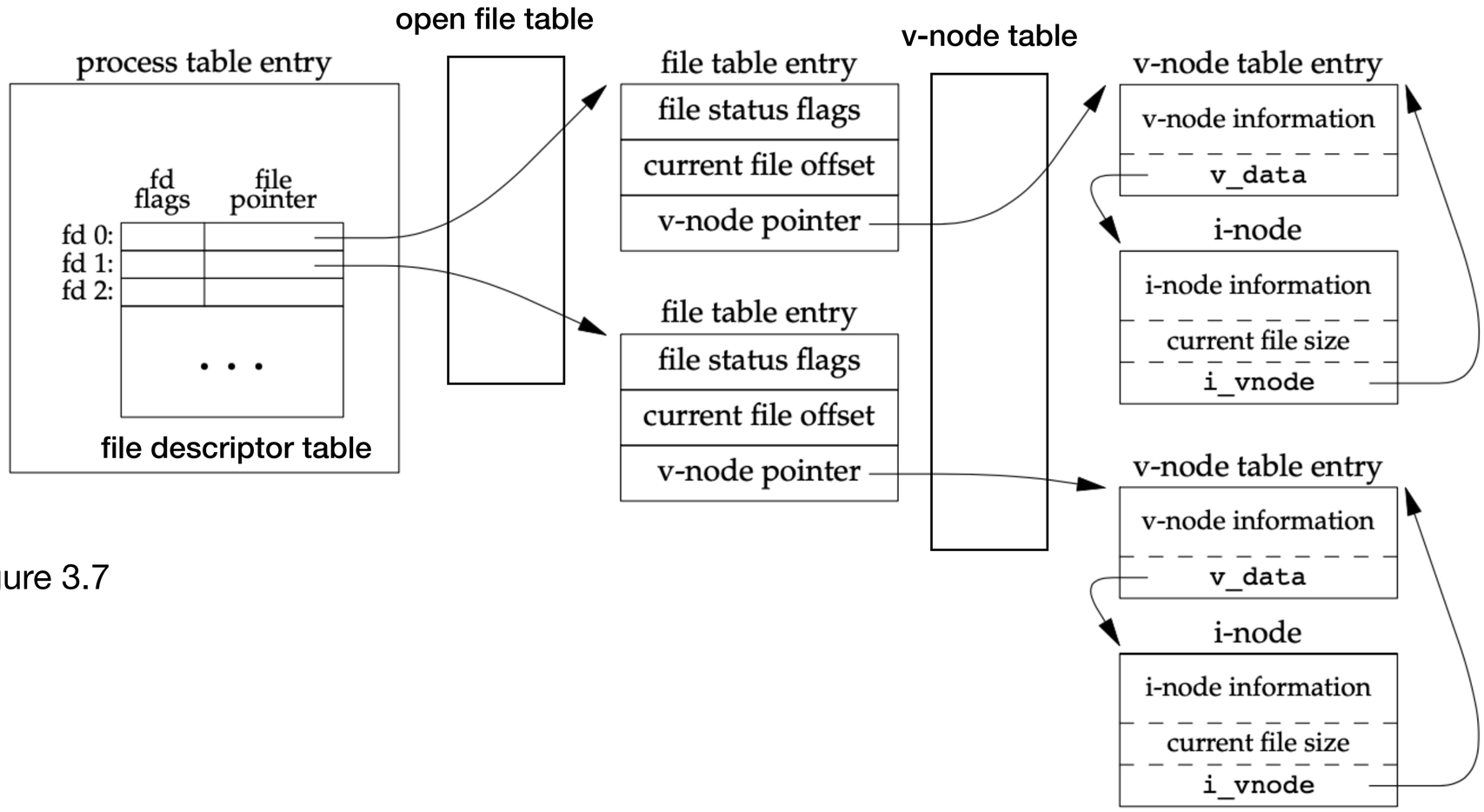
Unix kernel support for File I/O

- ***Open file descriptor table***; one entry per file descriptor; each entry contains:
  - The *file descriptor flag* (I will discuss it later)
  - A pointer to a system open file table entry
- ***Open file table***; each entry contains:
  - The *file status flag* for the file (readable/writable/append/sync/nonblocking)
  - The *current file offset* (I will discuss it later)
  - A pointer to the v-node table entry for the file
- ***V-node table***; each entry contains a V-node data structure that contains:
  - The pointer to the i-node structure of the file of the respective file
  - V-node information

Unix kernel support for File I/O

- ***V-node*** is an in-memory structure for each open file:
  - Invented to support multiple file system types on a single computer system
  - V-node information: the type of file and pointers to functions that operate on the file
- ***I-node*** is both stored physically on the storage device and in memory:
  - Contains the metadata about the file: *file owner*, *file size*, *residing device*, *protection information*, and *locations of the data blocks comprising a file*, .. (you see some of them from "*ls*")
  - The OS kernel reads the I-node from the disk to memory when the associated file is opened; why?
- Remarks:
  - Linux does not have a v-node; a generic i-node (conceptually the same as the v-node) is used

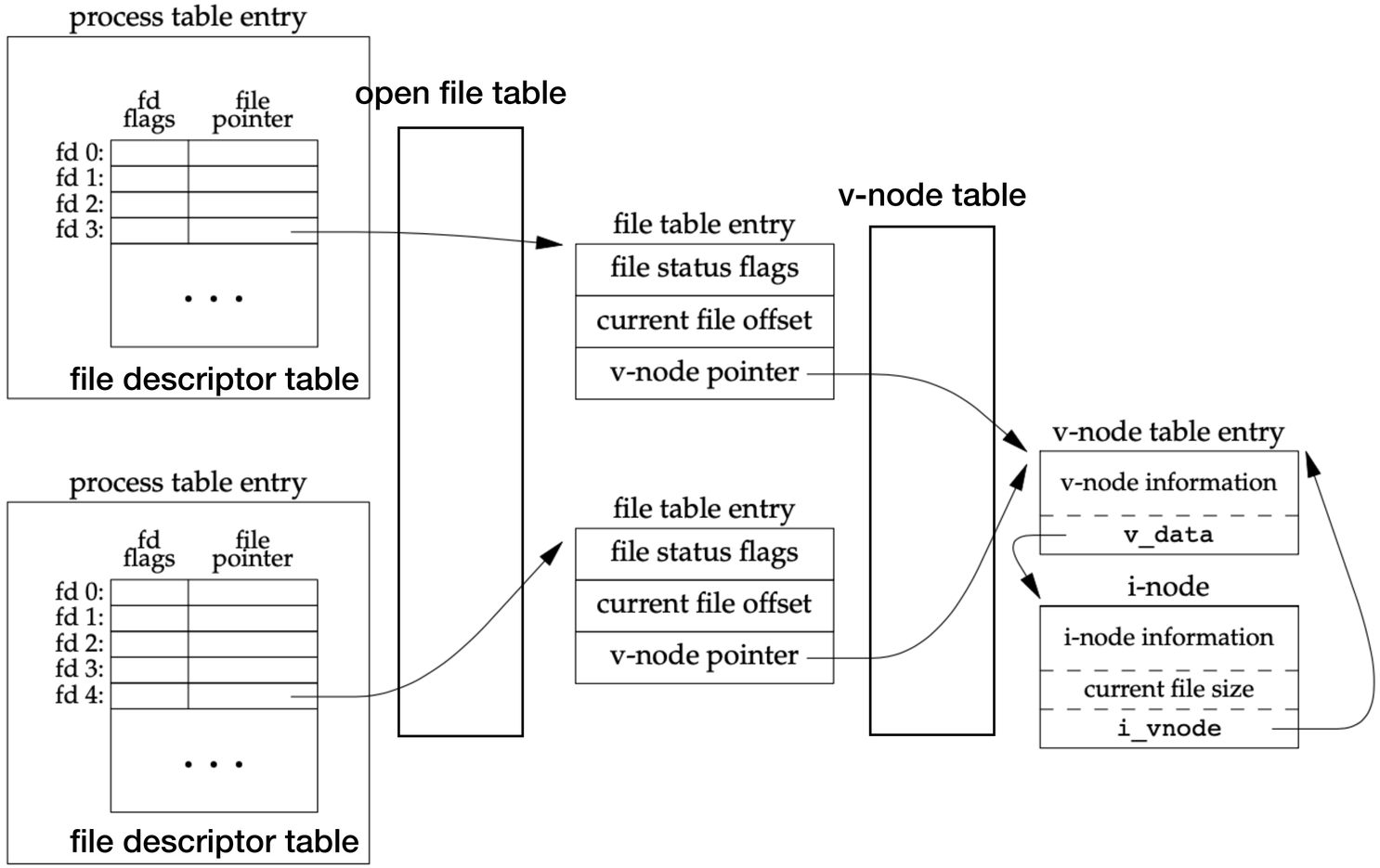
Unix kernel support for File I/O



From APUE 3rd Edition: Figure 3.7

Figure 3.7 Kernel data structures for open files

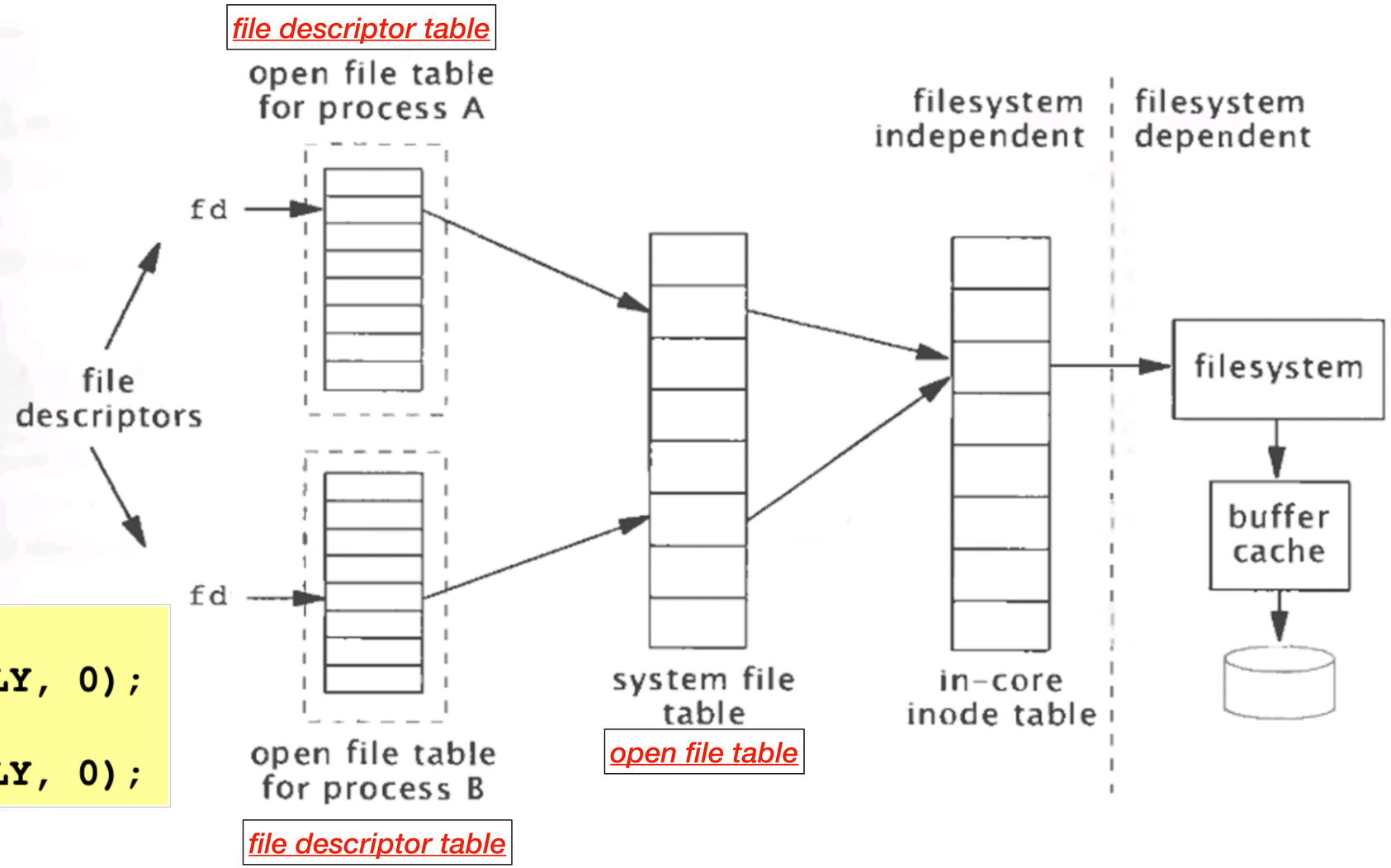
Unix kernel support for File I/O



From APUE 3rd Edition: Figure 3.8

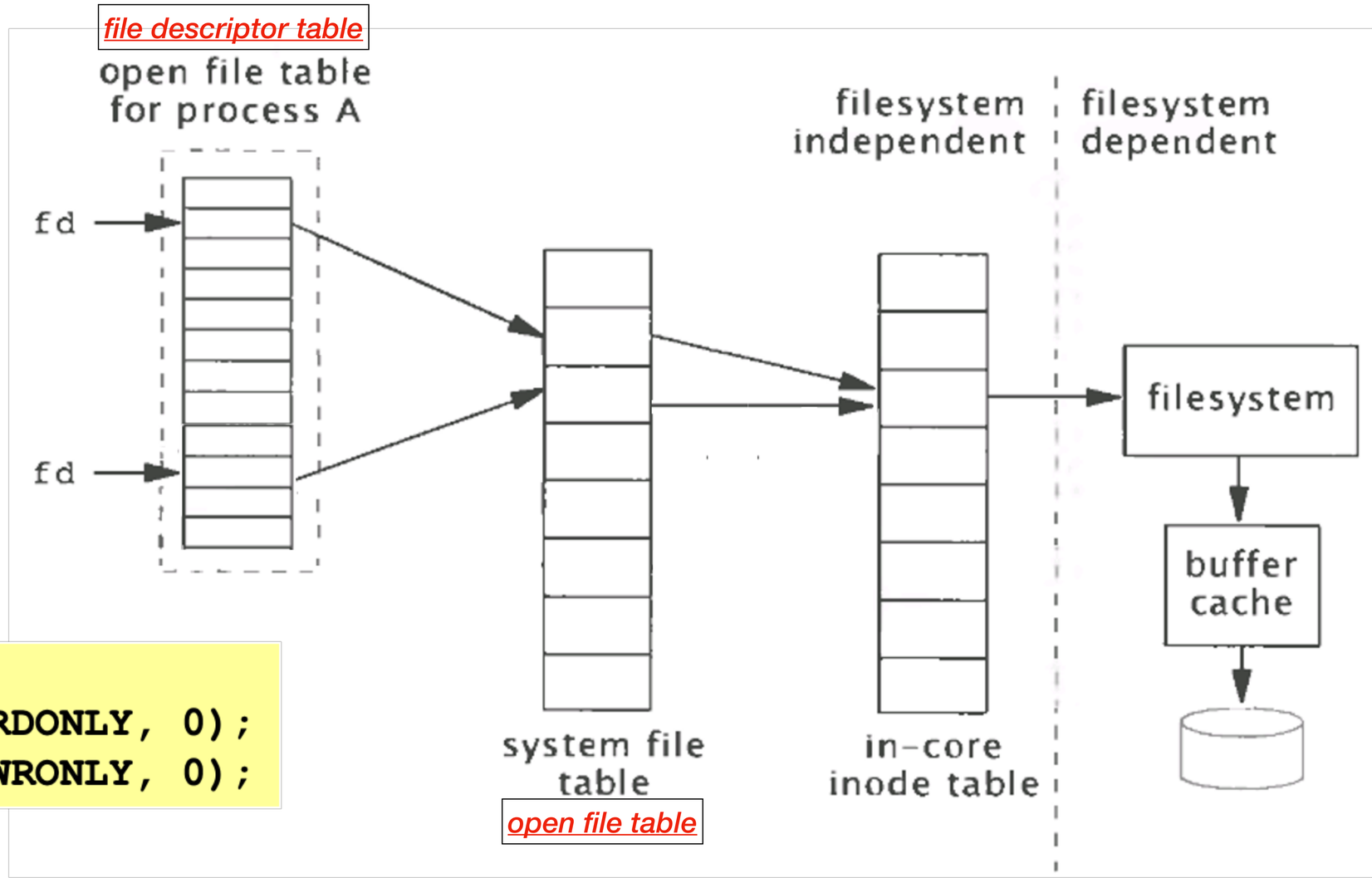
Figure 3.8 Two independent processes with the same file open

File I/O in Linux




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

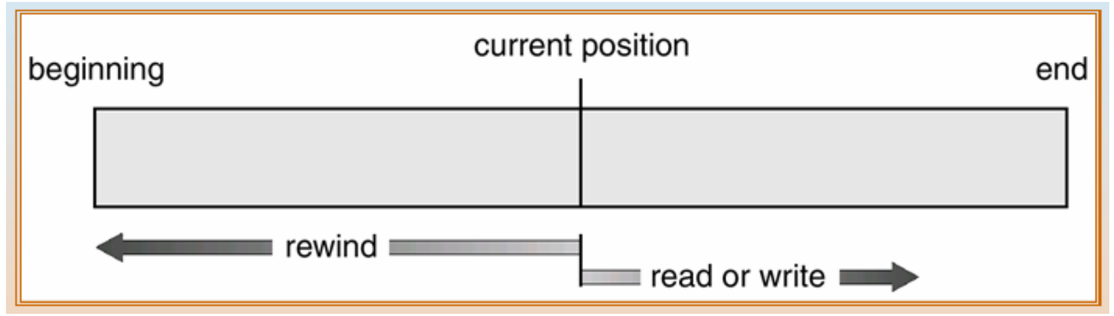
File I/O in Linux




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

File Offset

- Every open file in Unix has an associated "(*current) file offset*":
  - An integer that counts the number of bytes from the beginning of the file
  - The value of *current file offset* must be positive for regular files; certain devices could allow negative offets
- By default, this offset is *initialized to 0 when a file is opened*, unless the *O_APPEND* option is specified (in the *oflag* parameter)
- Read and write operations cause the offset to be incremented by the number of bytes read or written!




```
O_APPEND
       The file is opened in append mode.  Before each write(2),
       the file offset is positioned at the end of the file, as
```

File I/O: lseek

```
//off_t: signed int type
#include <unistd.h>
//Returns the new file offset on a success call, -1 on error
off_t lseek(int fd, off_t offset, int whence);
```

- An open file's *current file offset* can be set by calling *lseek()*
- *lseek()* parameters:
  - *fd* is the file descriptor of the open file
  - *whence* determines the interpretation of the *offset* parameter, can be set to three values:
    - *SEEK_SET*: the file's offset is set to *offset* bytes from the beginning of the file
    - *SEEK_CUR*: the file's offset is set to its current value plus the *offset* (offset can be positive or negative)
    - *SEEK_END*: the file's offset is set to the size of the file plus the *offset* (offset can be positive or negative)

File I/O: lseek

Q: how do you get the current file offset using lseek?

```
#include <unistd.h>
//Returns the new file offset on a success call, -1 on error
off_t lseek(int fd, off_t offset, int whence);
```

- *whence* can be set to three values:
  - *SEEK_SET*: the file's offset is set to *offset* bytes from the beginning of the file
  - *SEEK_CUR*: the file's offset is set to its current value plus the *offset* (offset can be positive or negative)
  - *SEEK_END*: the file's offset is set to the size of the file plus the *offset* (offset can be positive or negative)

File I/O: lseek

A: get the current file offset using lseek:

```
#include <unistd.h>
//Returns the new file offset on a success ca
off_t lseek(int fd, off_t offset, int whence);
```

```
off_t currpos;
currpos = lseek(fd, 0, SEEK_CUR);
```

- *whence* can be set to three values:
  - *SEEK_SET*: the file's offset is set to *offset* bytes from the beginning of the file
  - ***SEEK_CUR*: the file's offset is set to its current value plus the *offset* (offset can be positive or negative)**
  - *SEEK_END*: the file's offset is set to the size of the file plus the *offset* (offset can be positive or negative)

File I/O: lseek

```
#include <unistd.h>
//Returns the new file offset on a success call, -1 on error
off_t lseek(int fd, off_t offset, int whence);
```

```
#include "apue.h"

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

Figure 3.1 Test whether standard input is capable of seeking

From APUE 3rd Edition: Figure 3.1

If the fd *refers to* a pipe, FIFO, or socket, *lseek()* returns -1, and sets the *errno*

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

File I/O: lseek

- The <errno.h> header file defines the integer variable ***errno*** — set by system calls or some library functions in the event of an error to indicate what went wrong

ERRORS top

EBADF *fd* is not an open file descriptor.

EINVAL *whence* is not valid. Or: the resulting file offset would be negative, or beyond the end of a seekable device.

ENXIO *whence* is **SEEK_DATA** or **SEEK_HOLE**, and *offset* is beyond the end of the file, or *whence* is **SEEK_DATA** and *offset* is within a hole at the end of the file.

EOVERFLOW
The resulting file offset cannot be represented in an *off_t*.

ESPIPE *fd* is associated with a pipe, socket, or FIFO.

From https://man7.org/linux/man-pages/man2/lseek.2.html

```
#include "apue.h"

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

Figure 3.1 Test whether standard input is capable of seeking

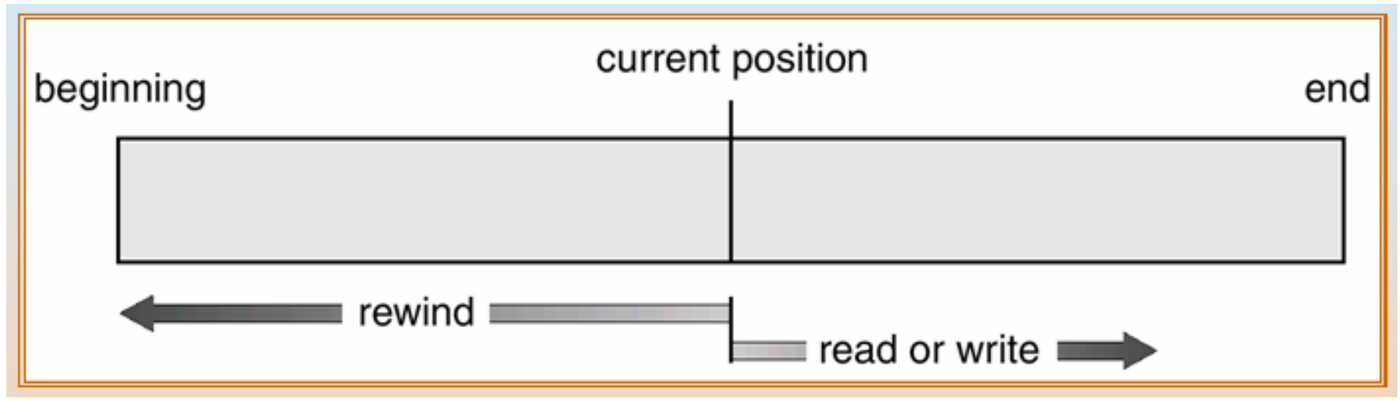
From APUE 3rd Edition: Figure 3.1

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

File I/O: lseek

```
#include <unistd.h>
//Returns the new file offset on a success call, -1 on error
off_t lseek(int fd, off_t offset, int whence);
```

- You can use *lseek()* to do the following:
  - Seek to a negative offset
  - Seek 0 bytes from the current position
  - Seek past the end of the file

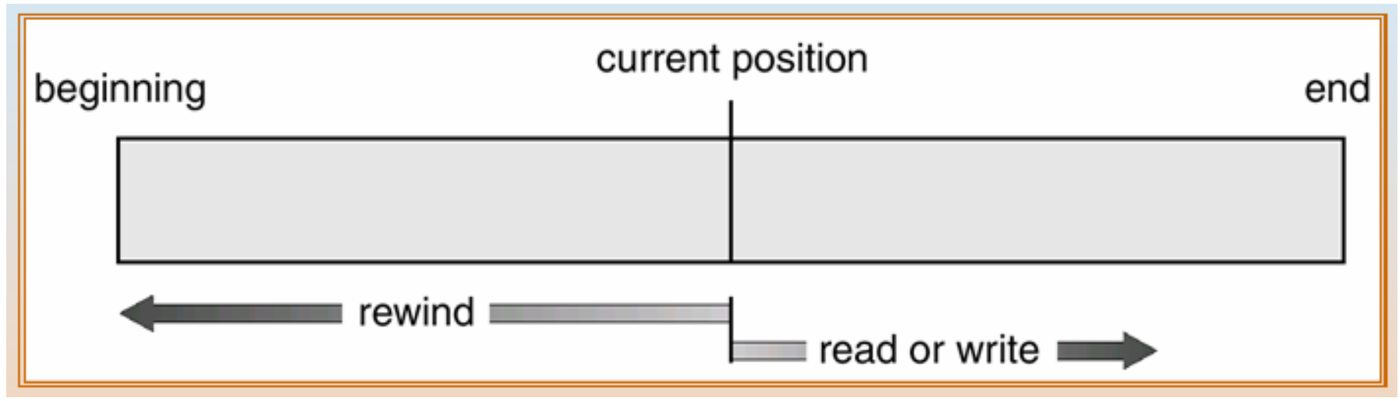


File I/O: lseek

```
#include <unistd.h>
//Returns the new file offset on a success call, -1 on error
off_t lseek(int fd, off_t offset, int whence);
```

What if lseek() uses a negative offset that goes beyond the beginning of the file?

- You can use *lseek()* to do the following:
  - Seek to a negative offset
  - Seek 0 bytes from the current position
  - Seek past the end of the file



File I/O: lseek

```
#include <unistd.h>
//Returns the new file offset on a success call, -1 on error
off_t lseek(int fd, off_t offset, int whence);
```

- *lseek()* only records the current file offset within the kernel
  - No actual I/O takes place
  - The updated offset is used by the next file operation (e.g., *read* or *write*)
- The file's offset can be greater than the file's current size — the next *write()* to the file will extend the file
  - Reads from any bytes in a file that have not been written are read back as 0

File I/O: read

```
//ssize_t: signed int type; size_t: unsigned int type
#include <unistd.h>
//Returns the number of bytes read on a success call, -1 on error, 0 if end of file (EOF)
is encountered
ssize_t read(int fd, void *buf, size_t nbytes);
```

- Data is read from an open file via the file descriptor *fd* into the memory buffer *buf* (*buf* is provided by the caller)
- The read operation starts at *the file's current offset*; before a successful return, the file offset is *increased* by the number of bytes read

File I/O: read

```
#include <unistd.h>
//Returns the number of bytes read on a success call, -1 on error, 0 if end of file (EOF)
is encountered
ssize_t read(int fd, void *buf, size_t nbytes);
```

- There are several cases in which the number of bytes read is ***less than*** the amount requested:
  - When reading from a regular file, if the end of the file is reached before the requested number of bytes has been read:
    - Q: what if we read 100 bytes from a file that has 30 bytes remaining until the end of the file? What will return for the next time that we call read?
  - More cases described in the textbook (e.g., read from pipe, FIFO) — we will discuss later

File I/O: write

```
#include <unistd.h>
//Returns the number of bytes written on a success call, -1 on error
ssize_t write(int fd, void *buf, size_t nbytes);
```

- Data stored in the buffer *buf* is written to an open file
- The write operation starts at the file's current offset
  - After a successful write, the file's offset is increased by the number of bytes written
- Common write errors: filling up a disk or exceeding the file size limit for a given process

File I/O: write

```
#include <unistd.h>
//Returns the number of bytes written on a success call, -1 on error
ssize_t write(int fd, void *buf, size_t nbytes);
```

- Tricky stuff:
  - If the *O_APPEND* option was specified when the file was opened, the file's offset is set to the current end of the file ***before*** each write operation; what about read()?

Remarks: Unix read() and write()

- write() can go beyond the EOF
  - A write() that starts at or goes beyond the EOF will extend the file
- read() stops when it reaches the EOF
  - A read() that reaches the EOF returns 0, indicating no more data to read.

Example: Create a hole in a file

```
#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);
}
```

```
$ ./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
```

od –c: dump files in the format of ASCII characters or backslash escapes (in octal format)

$ cat < file.hole > file.nohole

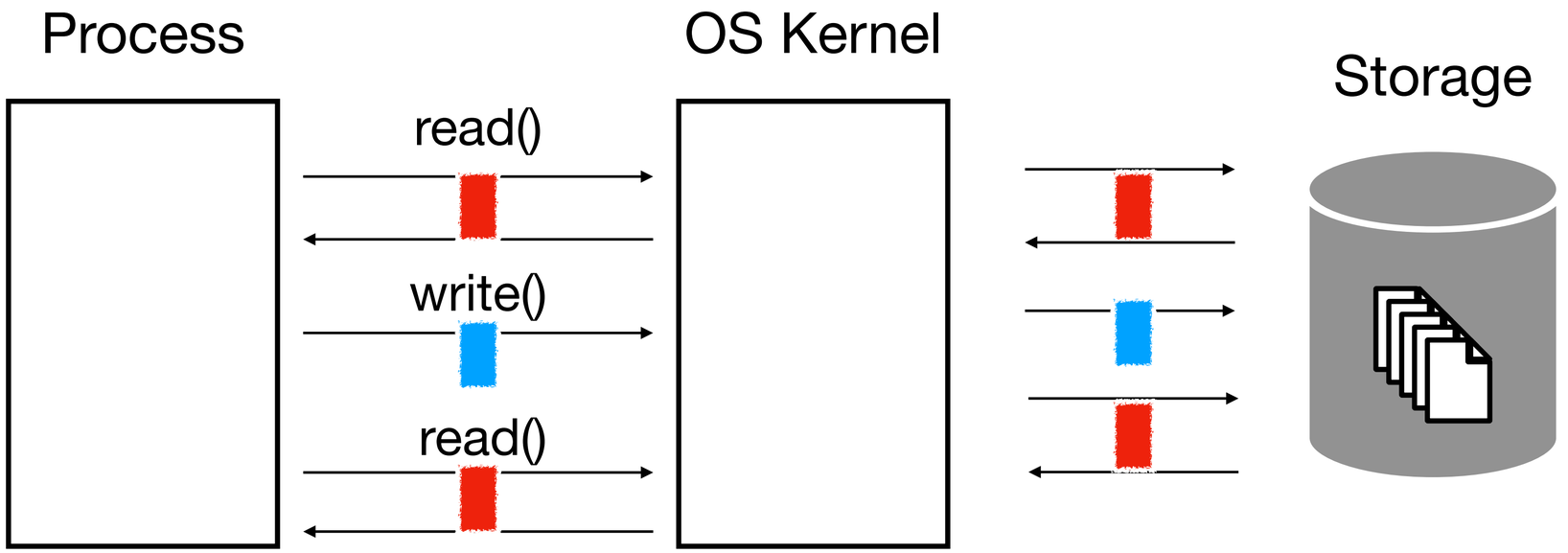
```
$ ls -ls file.hole file.nohole     compare sizes
 8 -rw-r--r--  1 sar       16394 Nov 25 01:01 file.hole
20 -rw-r--r--  1 sar       16394 Nov 25 01:03 file.nohole
```

# of disk blocks

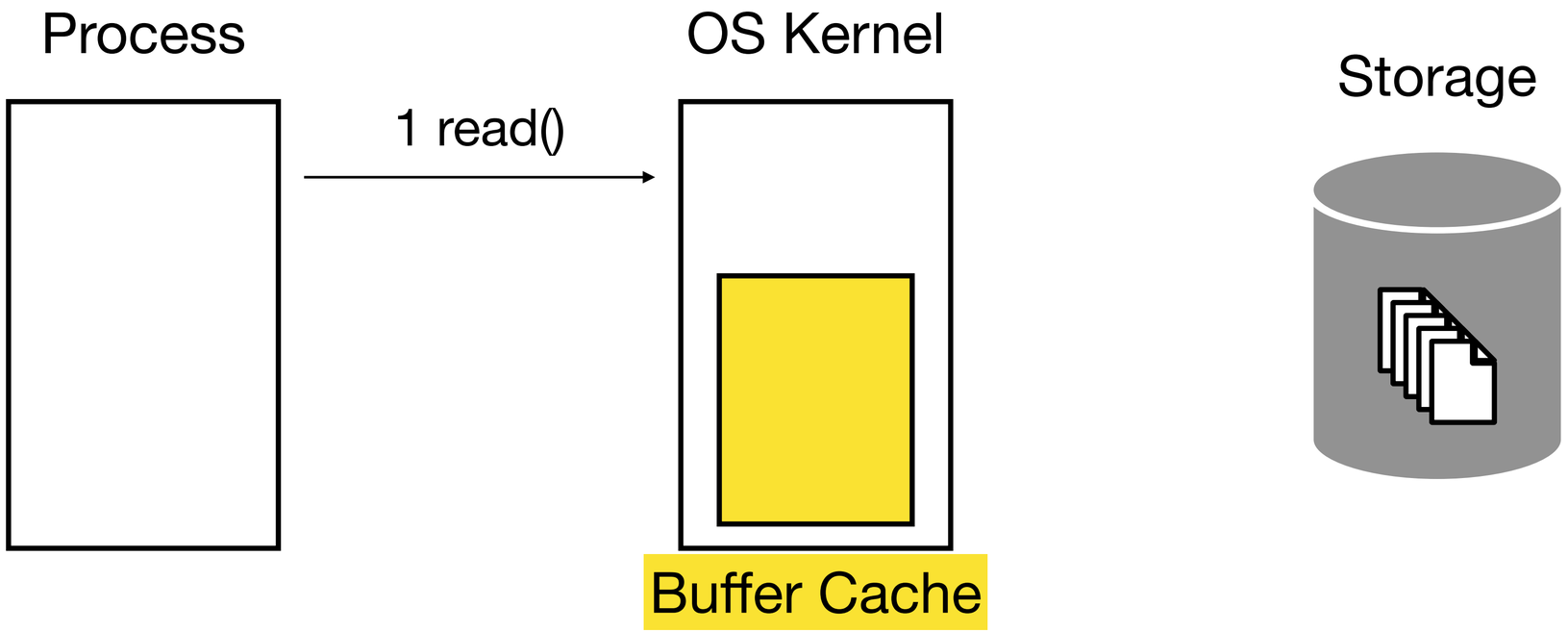
File size

From APUE 3rd Edition: Figure 3.2

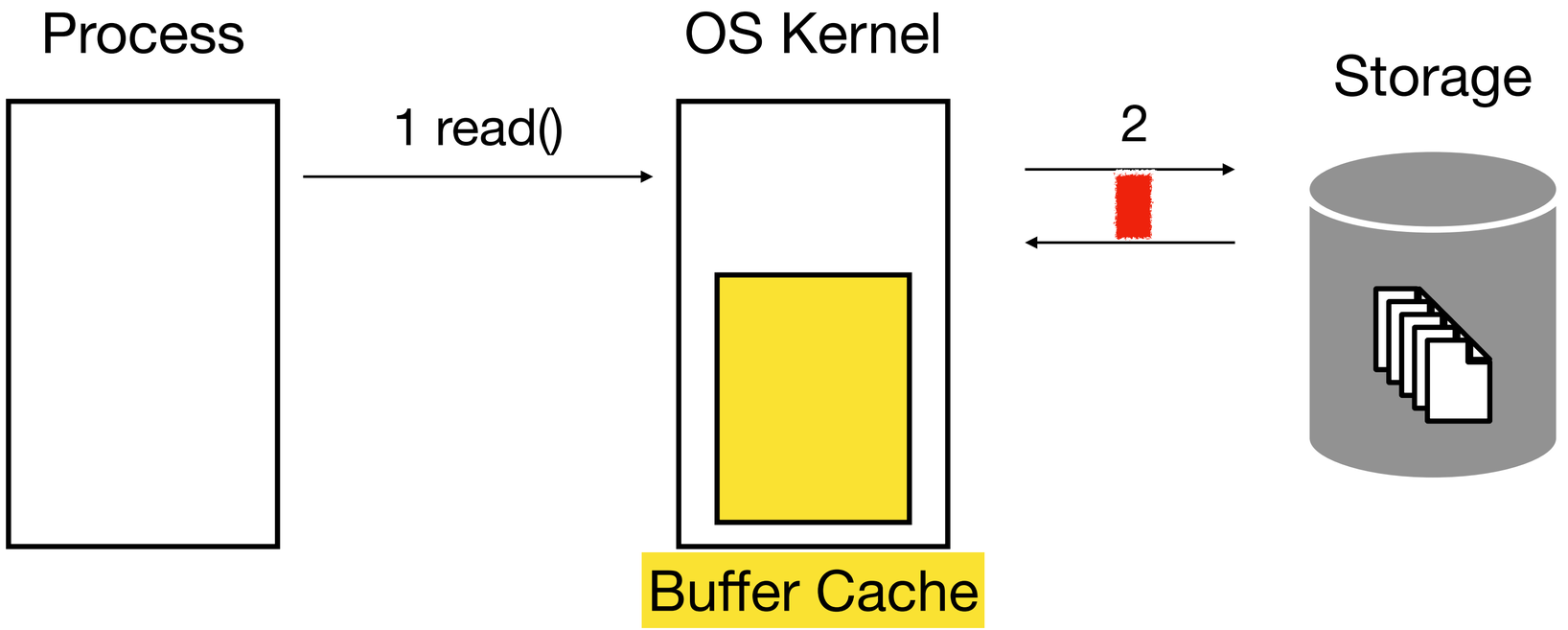
How is file I/O conducted in Unix?



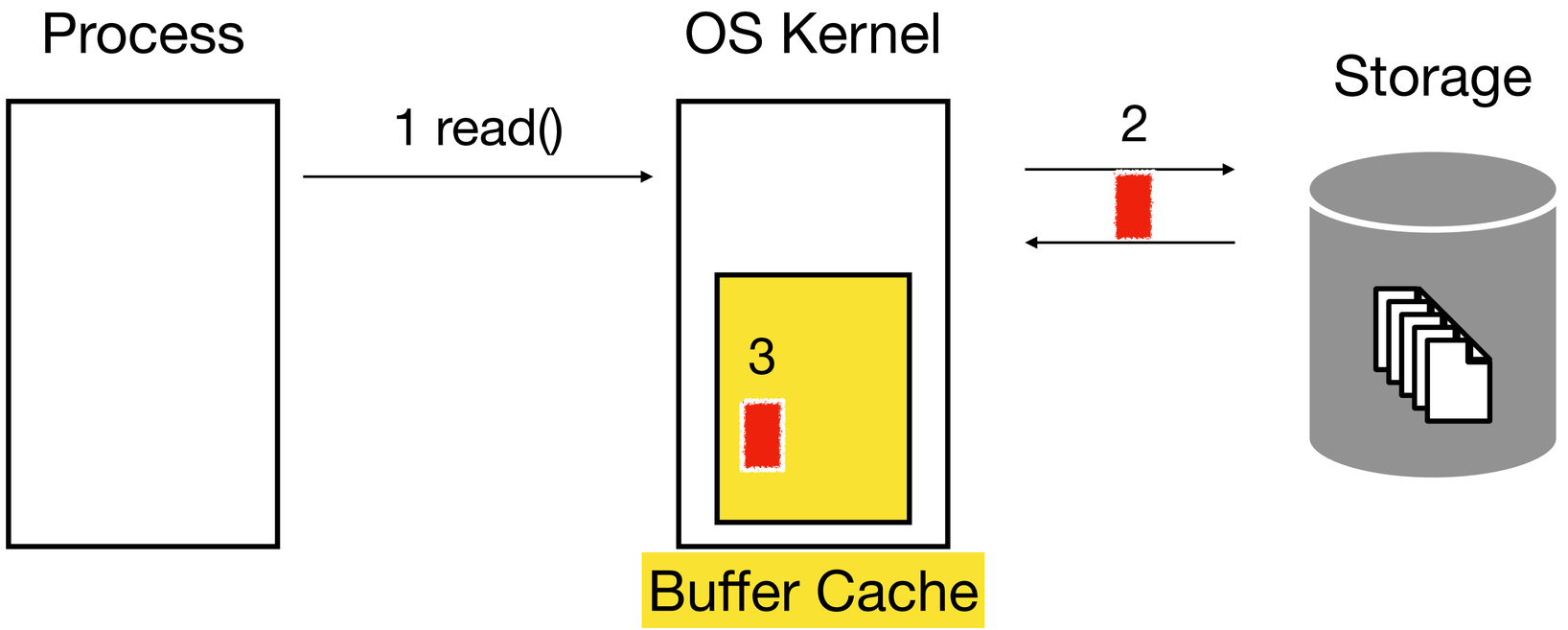
How is file I/O conducted in Unix?



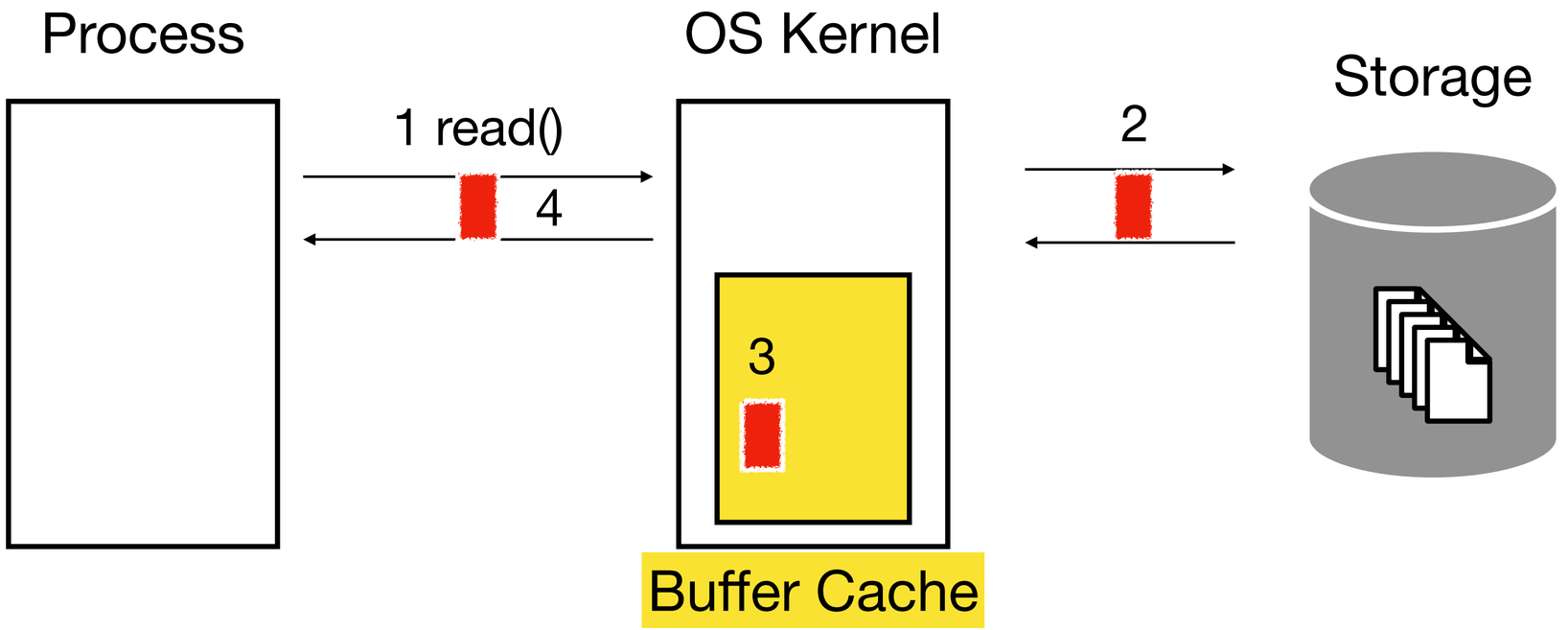
How is file I/O conducted in Unix?



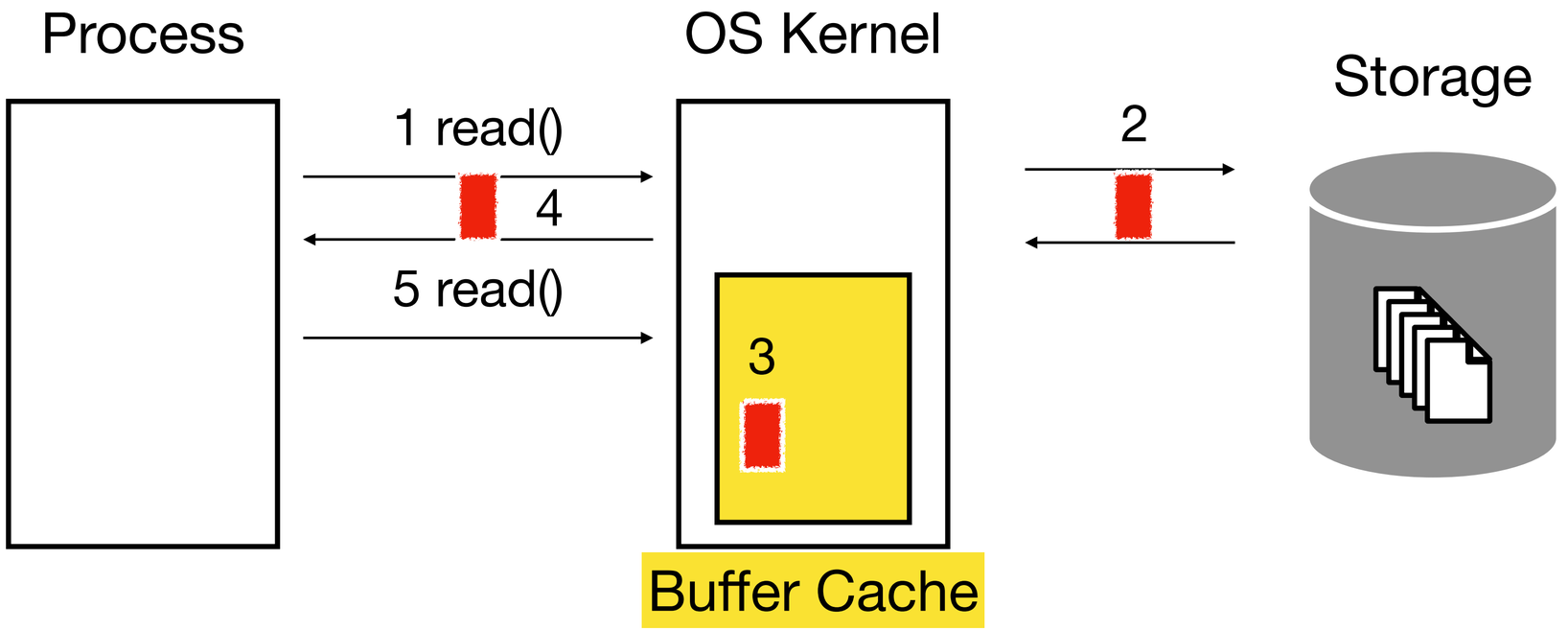
How is file I/O conducted in Unix?



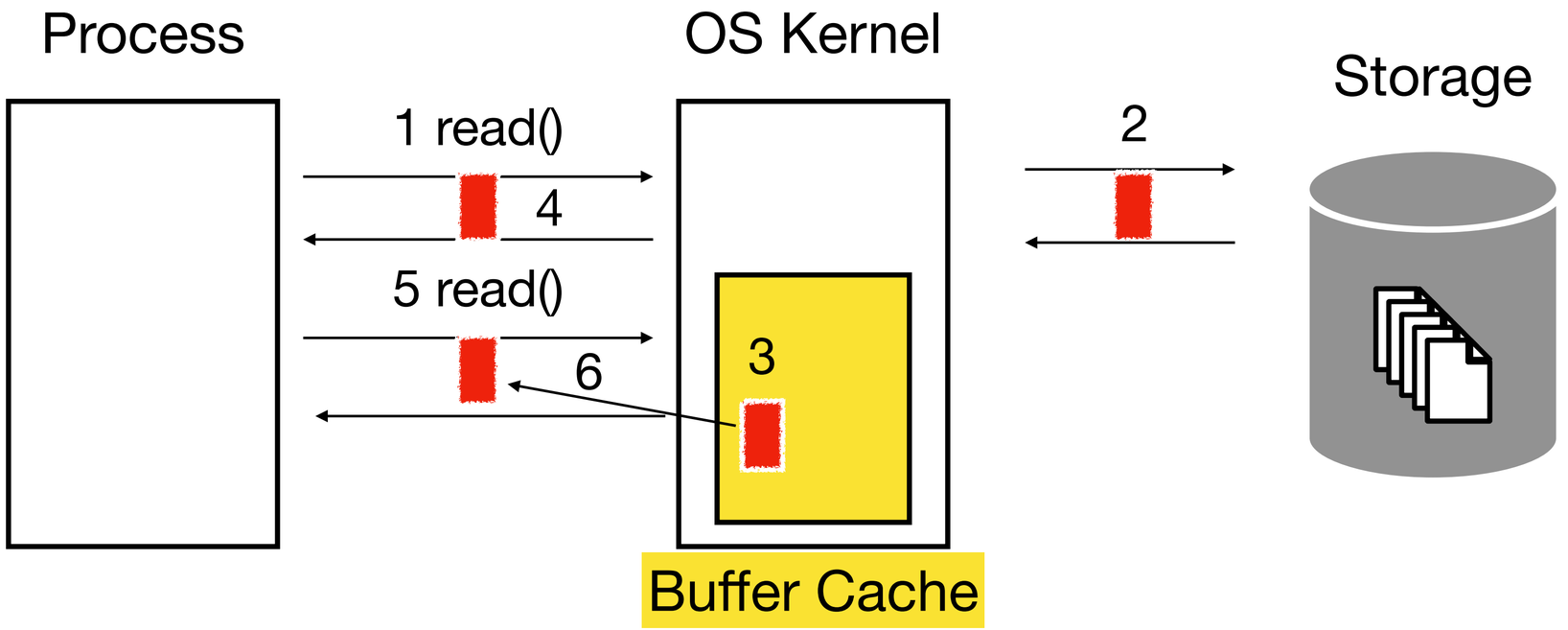
How is file I/O conducted in Unix?



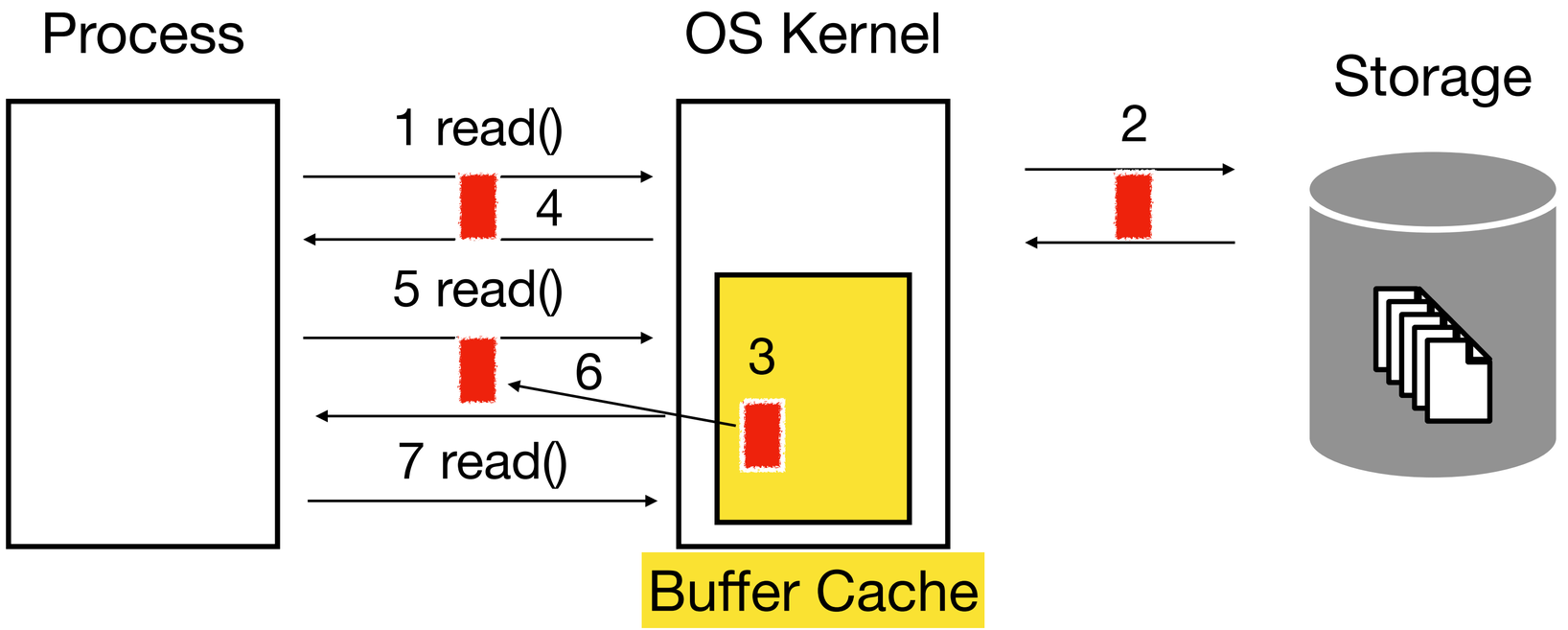
How is file I/O conducted in Unix?



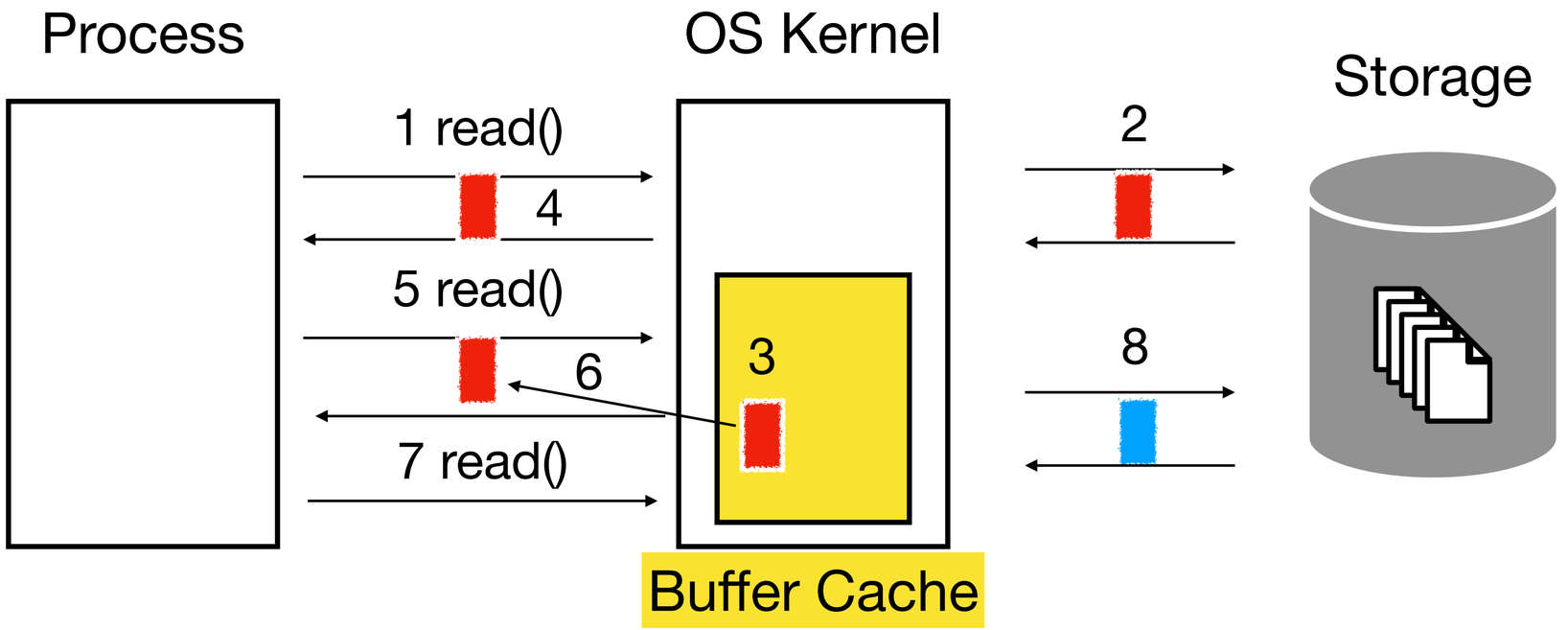
How is file I/O conducted in Unix?



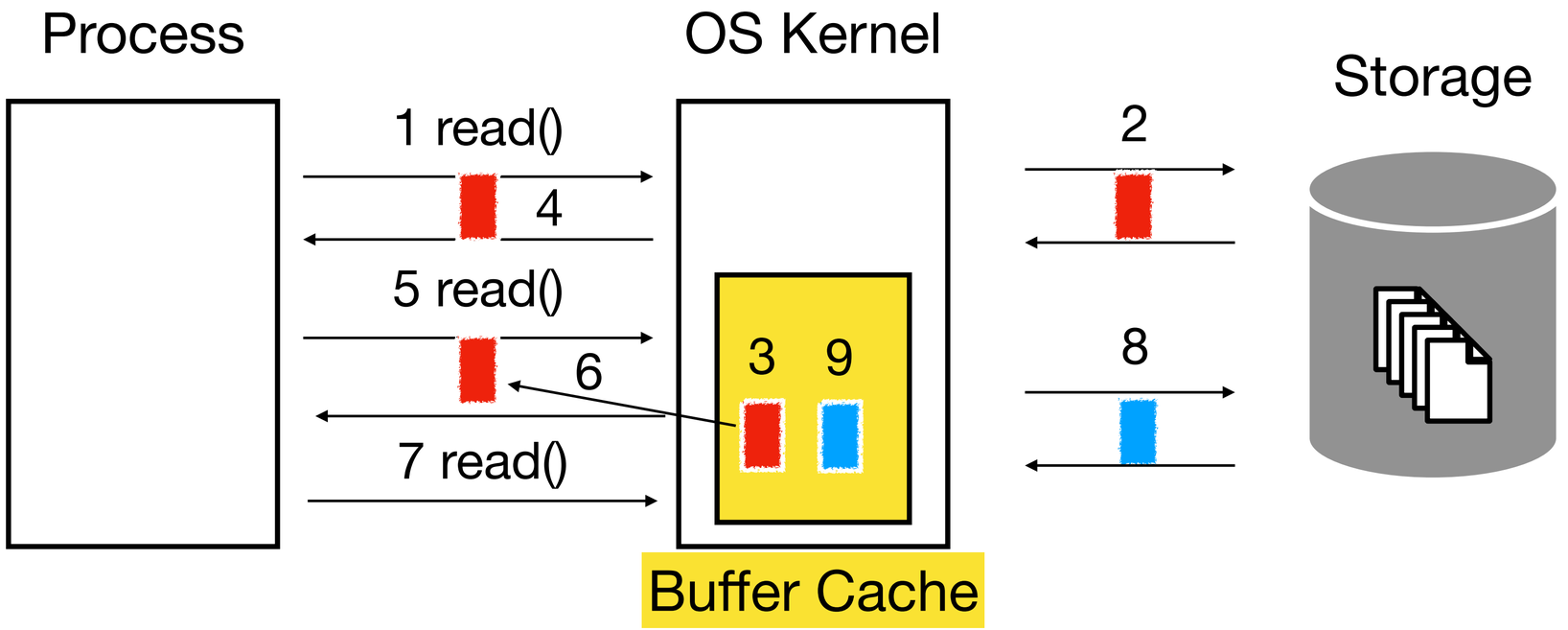
How is file I/O conducted in Unix?



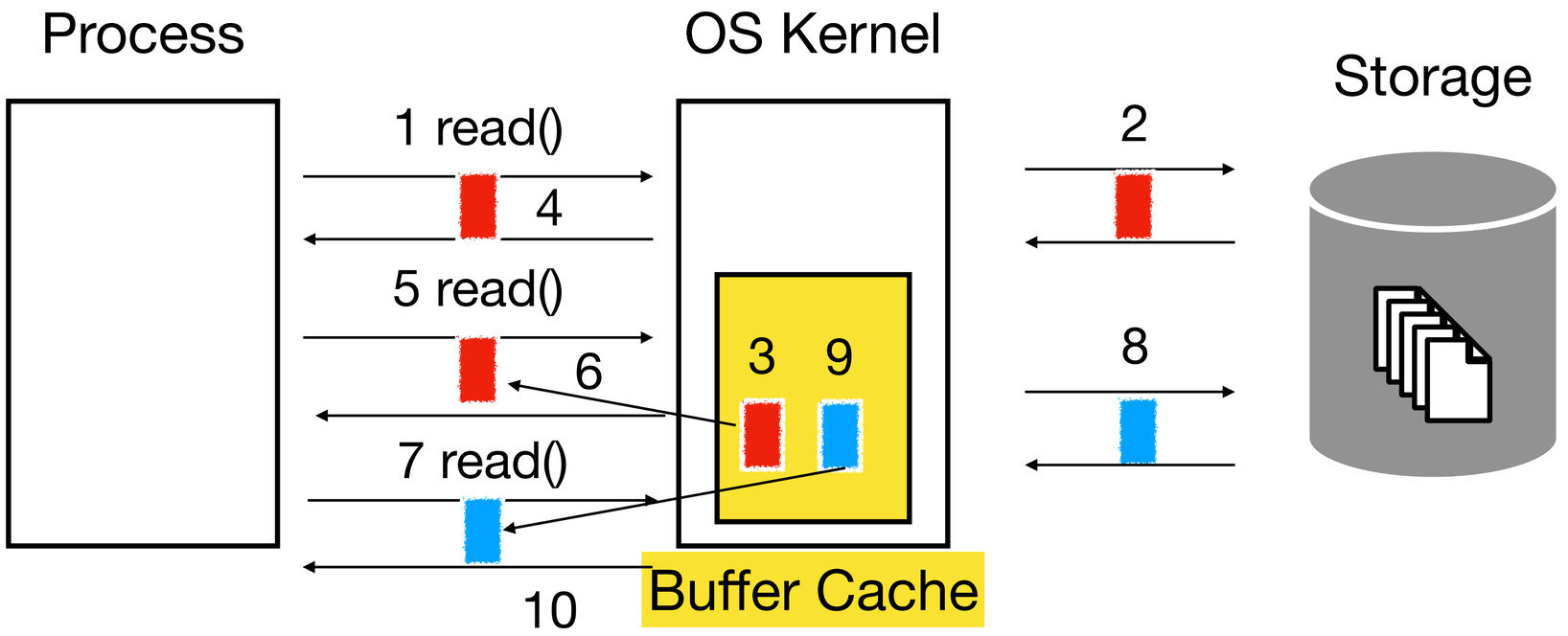
How is file I/O conducted in Unix?



How is file I/O conducted in Unix?

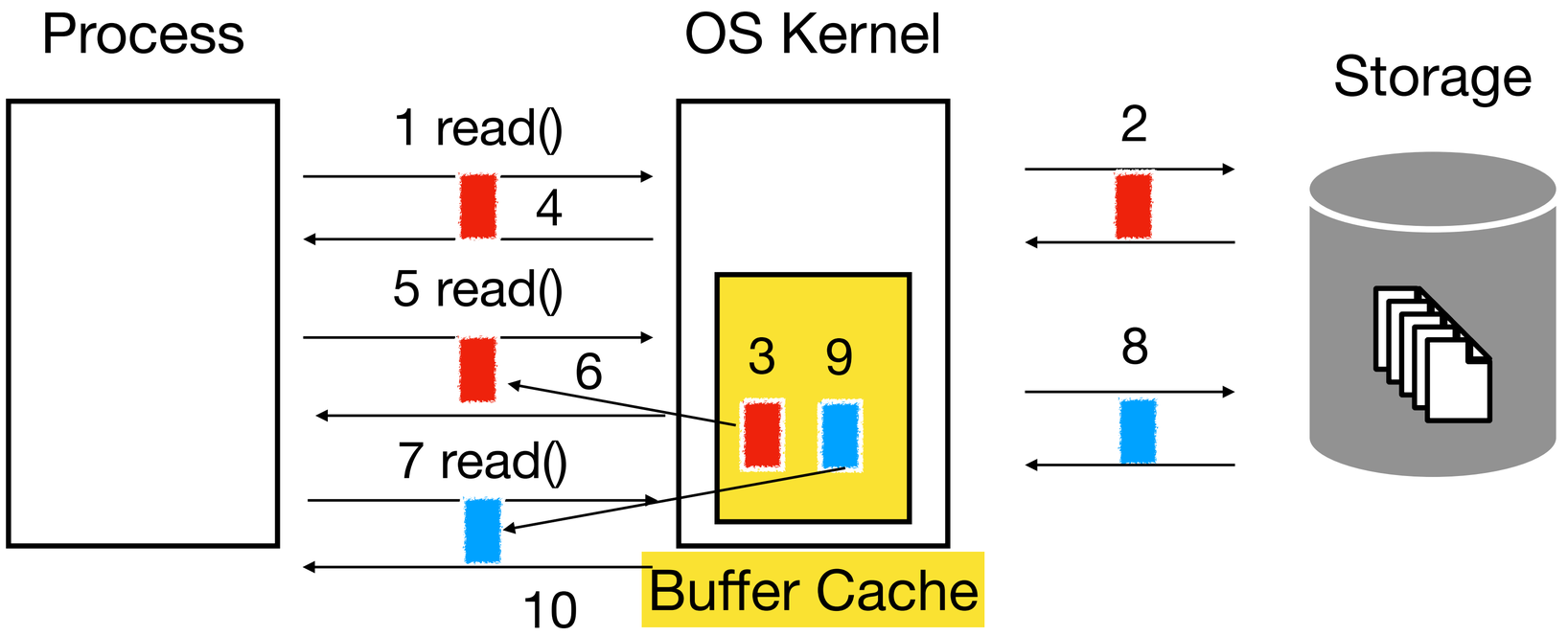


How is file I/O conducted in Unix?



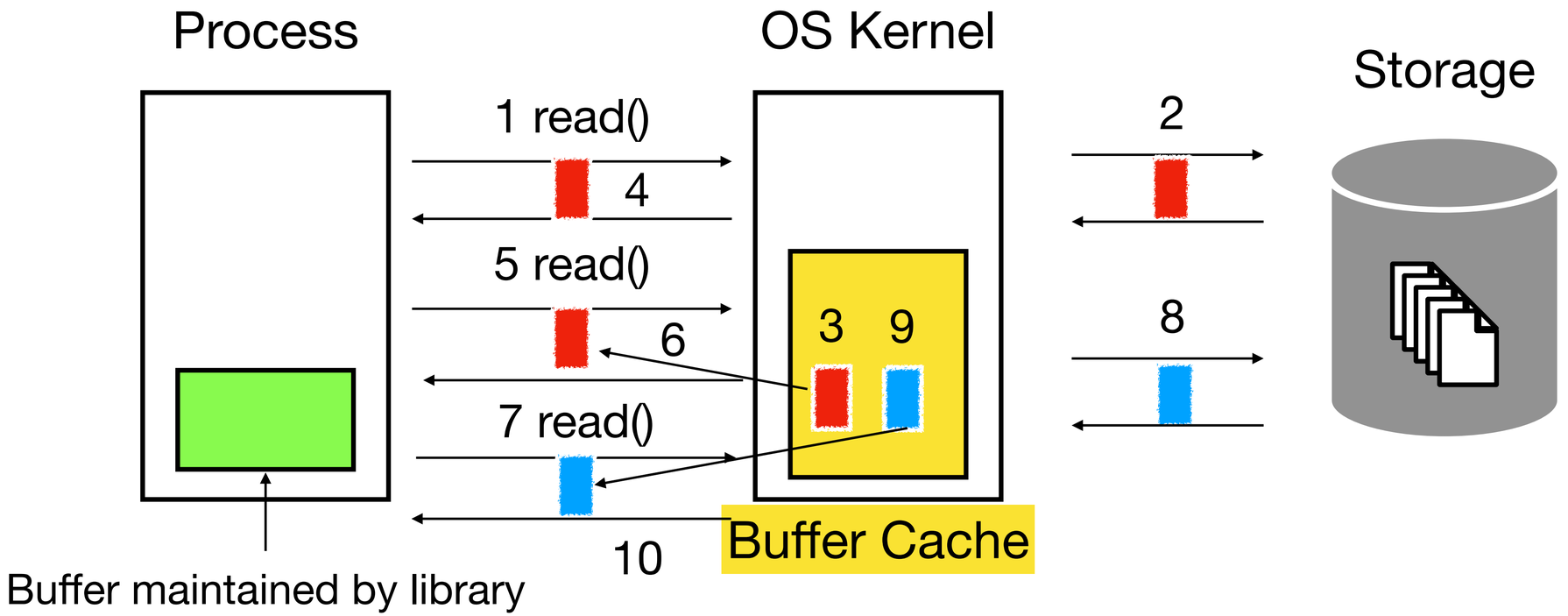
How is file I/O conducted in Unix?

Q: how does write() work?



Remarks: buffered v.s. unbuffered I/O

Q: how does buffered/unbuffered I/O differ?



Remarks: buffered v.s. unbuffered I/O

```
f#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <time.h>

#define NUM_OPERATIONS 100000

void buffered_io() {
    FILE *file = fopen("buffered.txt", "w");
    for (int i = 0; i < NUM_OPERATIONS; i++) {
        fprintf(file, "X");
    }
    fclose(file);
}

void unbuffered_io() {
    int fd = open("unbuffered.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    for (int i = 0; i < NUM_OPERATIONS; i++) {
        write(fd, "X", 1);
    }
    close(fd);
}
```

```
int main() {
    clock_t start, end;
    double cpu_time_used;

    start = clock();
    buffered_io();
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Buffered I/O took %f seconds\n", cpu_time_used);

    start = clock();
    unbuffered_io();
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Unbuffered I/O took %f seconds\n", cpu_time_used);

    return 0;
}
```

I/O Efficiency

What have you observed?

- Results for reading a 516,581,760-byte file ($FILE) on the Linux ext4 file system, using 20 different buffer sizes

```
#include "apue.h"

#define BUFFSIZE    4096

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);
}
```

Changing BUFFSIZE to different values

Figure 3.5 Copy standard input to standard output

BUFFSIZE	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)	Number of loops
1	20.03	117.50	138.73	516,581,760
2	9.69	58.76	68.60	258,290,880
4	4.60	36.47	41.27	129,145,440
8	2.47	15.44	18.38	64,572,720
16	1.07	7.93	9.38	32,286,360
32	0.56	4.51	8.82	16,143,180
64	0.34	2.72	8.66	8,071,590
128	0.34	1.84	8.69	4,035,795
256	0.15	1.30	8.69	2,017,898
512	0.09	0.95	8.63	1,008,949
1,024	0.02	0.78	8.58	504,475
2,048	0.04	0.66	8.68	252,238
4,096	0.03	0.58	8.62	126,119
8,192	0.00	0.54	8.52	63,060
16,384	0.01	0.56	8.69	31,530
32,768	0.00	0.56	8.51	15,765
65,536	0.01	0.56	9.12	7,883
131,072	0.00	0.58	9.08	3,942
262,144	0.00	0.60	8.70	1,971
524,288	0.01	0.58	8.58	986

Time spent running in user mode

Time spent running in kernel mode

Total time spent

You can get these time via the “time” command

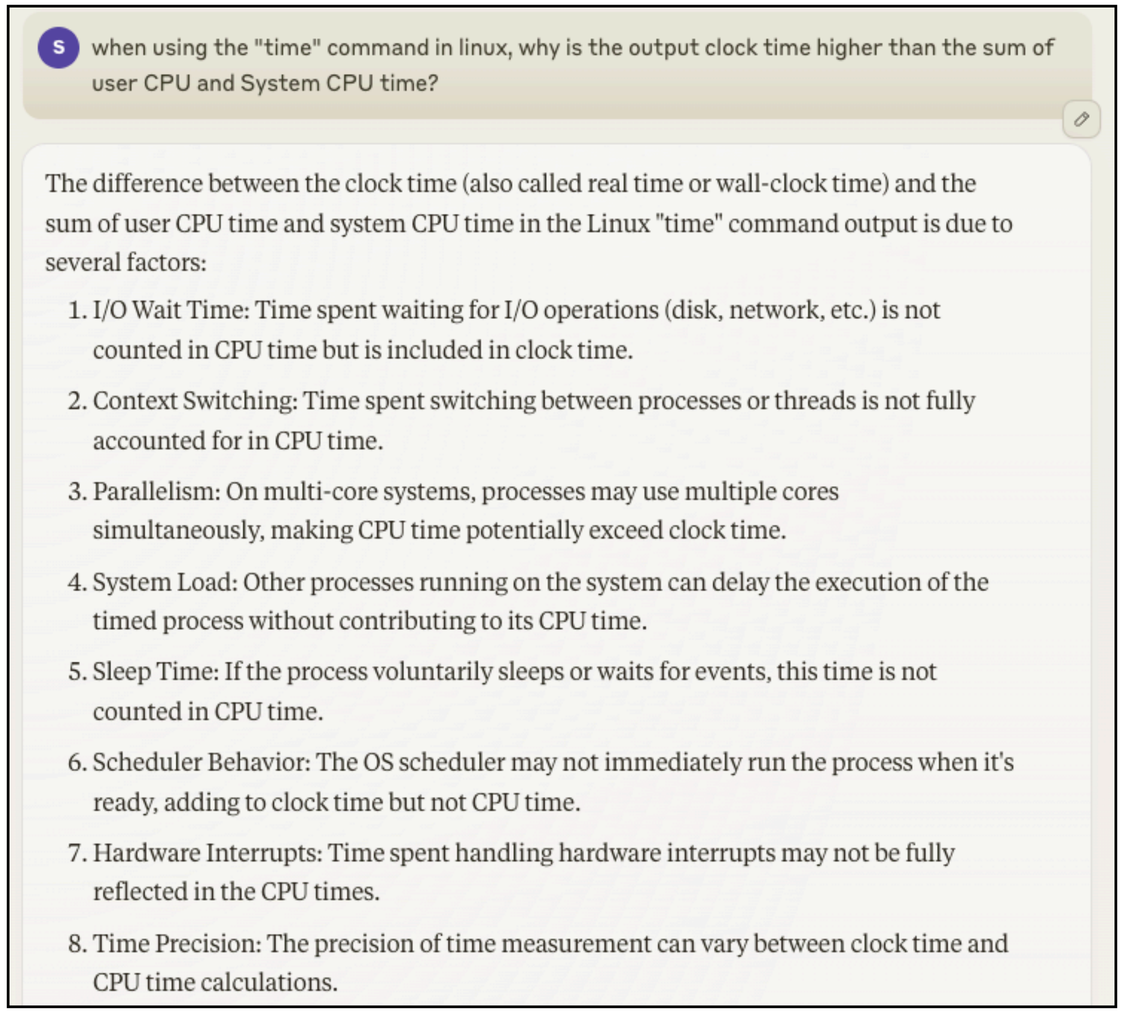
Figure 3.6 Timing results for reading with different buffer sizes on Linux

From APUE 3rd Edition: Figure 3.6

$ time /dev/zero < ./a.out < cat $FILE

What is /dev/zero? You can think of it as a black hole — takes input but output nothing!

I/O Efficiency



I/O Efficiency

User CPU Time + Sys CPU Time <= Clock Time

- Results for reading a 516,581,760-byte file on the Linux ext4 file system, using 20 different buffer sizes

```
#include "apue.h"

#define BUFFSIZE    4096

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);
}
```

Figure 3.5 Copy standard input to standard output

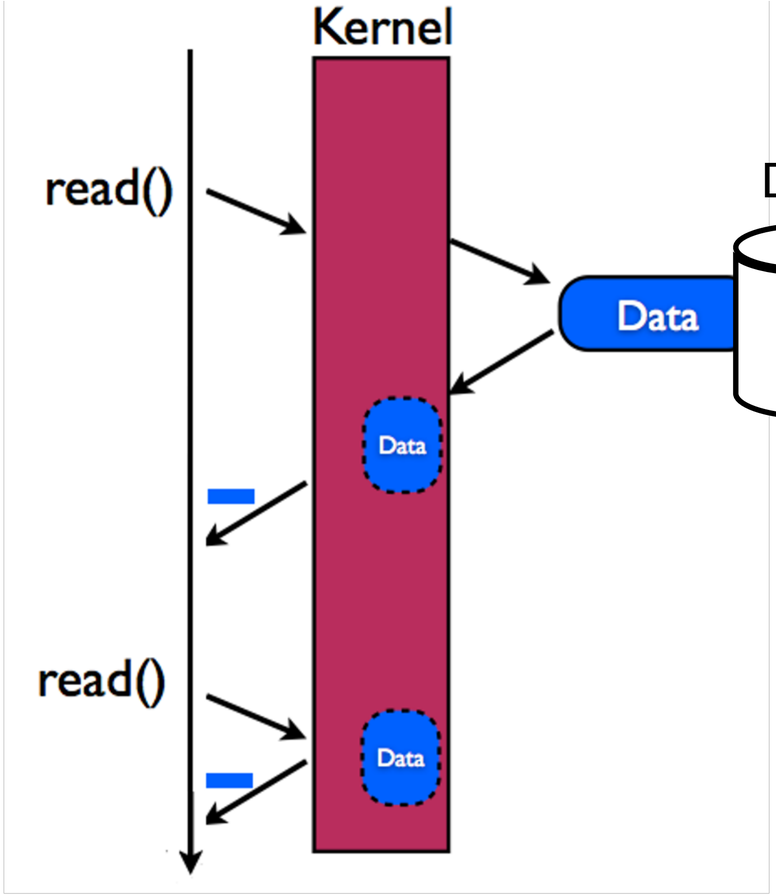
the elapsed clock time for buffer sizes as small as 32 bytes is as good as the elapsed time for larger buffer sizes, why?

BUFFSIZE	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)	Number of loops
1	20.03	117.50	138.73	516,581,760
2	9.69	58.76	68.60	258,290,880
4	4.60	36.47	41.27	129,145,440
8	2.47	15.44	18.38	64,572,720
16	1.07	7.93	9.38	32,286,360
32	0.56	4.51	8.82	16,143,180
64	0.34	2.72	8.66	8,071,590
128	0.34	1.84	8.69	4,035,795
256	0.15	1.30	8.69	2,017,898
512	0.09	0.95	8.63	1,008,949
1,024	0.02	0.78	8.58	504,475
2,048	0.04	0.66	8.68	252,238
4,096	0.03	0.58	8.62	126,119
8,192	0.00	0.54	8.52	63,060
16,384	0.01	0.56	8.69	31,530
32,768	0.00	0.56	8.51	15,765
65,536	0.01	0.56	9.12	7,883
131,072	0.00	0.58	9.08	3,942
262,144	0.00	0.60	8.70	1,971
524,288	0.01	0.58	8.58	986

Figure 3.6 Timing results for reading with different buffer sizes on Linux

$ time /dev/zero < ./a.out < cat $FILE

I/O Efficiency - Read-ahead



Sequential reads are detected, the system tries to read more data than an application requests, so future reads do not have to go to the disk

BUFFSIZE	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)	Number of loops
1	20.03	117.50	138.73	516,581,760
2	9.69	58.76	68.60	258,290,880
4	4.60	36.47	41.27	129,145,440
8	2.47	15.44	18.38	64,572,720
16	1.07	7.93	9.38	32,286,360
32	0.56	4.51	8.82	16,143,180
64	0.34	2.72	8.66	8,071,590
128	0.34	1.84	8.69	4,035,795
256	0.15	1.30	8.69	2,017,898
512	0.09	0.95	8.63	1,008,949
1,024	0.02	0.78	8.58	504,475
2,048	0.04	0.66	8.68	252,238
4,096	0.03	0.58	8.62	126,119
8,192	0.00	0.54	8.52	63,060
16,384	0.01	0.56	8.69	31,530
32,768	0.00	0.56	8.51	15,765
65,536	0.01	0.56	9.12	7,883
131,072	0.00	0.58	9.08	3,942
262,144	0.00	0.60	8.70	1,971
524,288	0.01	0.58	8.58	986

Figure 3.6 Timing results for reading with different buffer sizes on Linux

I/O Efficiency

User CPU Time + Sys CPU Time <= Clock Time

- Results for reading a 516,581,760-byte file on the Linux ext4 file system, using 20 different buffer sizes

```
#include "apue.h"

#define BUFFSIZE    4096

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);
}
```

Figure 3.5 Copy standard input to standard output

Minimum in the system time occurring at the few timing measurements starting around a BUFFSIZE of 4,096: the size of the disk I/O block

BUFFSIZE	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)	Number of loops
1	20.03	117.50	138.73	516,581,760
2	9.69	58.76	68.60	258,290,880
4	4.60	36.47	41.27	129,145,440
8	2.47	15.44	18.38	64,572,720
16	1.07	7.93	9.38	32,286,360
32	0.56	4.51	8.82	16,143,180
64	0.34	2.72	8.66	8,071,590
128	0.34	1.84	8.69	4,035,795
256	0.15	1.30	8.69	2,017,898
512	0.09	0.95	8.63	1,008,949
1,024	0.02	0.78	8.58	504,475
2,048	0.04	0.66	8.68	252,238
4,096	0.03	0.58	8.62	126,119
8,192	0.00	0.54	8.52	63,060
16,384	0.01	0.56	8.69	31,530
32,768	0.00	0.56	8.51	15,765
65,536	0.01	0.56	9.12	7,883
131,072	0.00	0.58	9.08	3,942
262,144	0.00	0.60	8.70	1,971
524,288	0.01	0.58	8.58	986

Figure 3.6 Timing results for reading with different buffer sizes on Linux

```
$ time /dev/zero < ./a.out < cat $FILE
```

Remarks: what happens with each call?

- lseek() updates the file offset of the file (stored in the system's open file table), no actual I/O is performed
  - Recall the special behavior for file writes to the files opened with the O_APPEND flag set
- For read() and write() from/to a file, the file's current file offset is incremented by the amount of bytes read from the or write to the file

Duplicating file descriptors

```
$> ls -la | more
$> cat file | wc
$> man ksh | grep “history”
$> ls -l | grep “bowman” | wc
$> who | sort > current_users
```

How are “|” and “>” implemented?

- Analogous to creating an alias for an open file
- Unix provides system calls (dup/dup2) to support file descriptor duplication, allows programs to efficiently redirect file input/output

File I/O: dup and dup2

```
#include <unistd.h>
//Returns the new file descriptor on a successful call; and -1 on an error
int dup(int fd);
int dup2(int fd, int fd2);
```

- *dup()* and *dup2()* create a copy of an existing file descriptor and return the new file descriptor
  - Copy as “both file descriptors refer to the same open file” —> sharing the same open file table entry
  - Impact: if the file offset is modified by using *lseek()* on one of the file descriptors, the offset is also changed for the other file descriptor; why?

File I/O: dup and dup2

- Assume that the next available descriptor is 3
- After dup(1) is executed, both descriptor 1 and 3 point to the same open file table entry

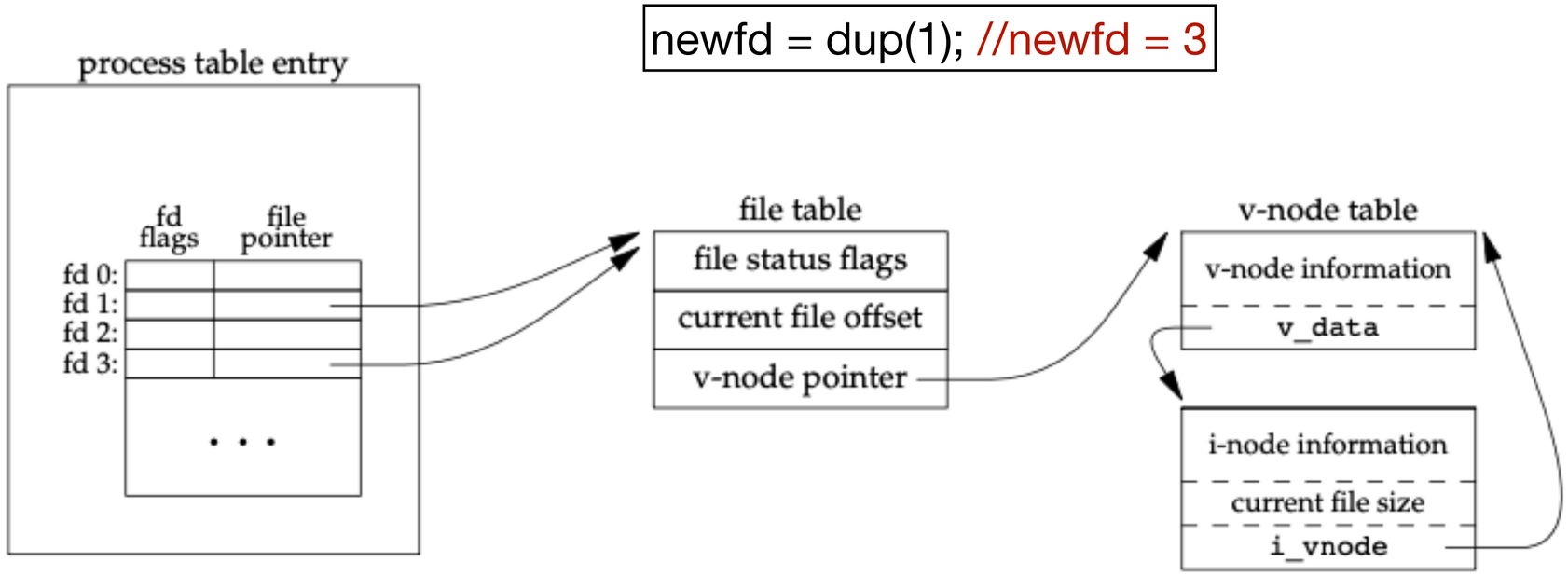


Figure 3.9 Kernel data structures after `dup(1)`

File I/O: dup and dup2

```
#include <unistd.h>
//Returns the new file descriptor on a successful call; and -1 on an error
int dup(int fd);
int dup2(int fd, int fd2);
```

- *dup*: copies *fd* to a newly allocated file descriptor
  - Assign the lowest-numbered available to the new file descriptor
- *dup2*: specify a destination *fd2* and copies *fd* to *fd2*
  - If *fd2* is open when the call is made, the Unix kernel closes *fd2* first before being reused

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 descriptor 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

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(fd2, &c, 1);
    dup2(fd2, fd1);
    read(fd1, &c, 1);
    printf("c = %c\n", c);
    exit(0);
}
```

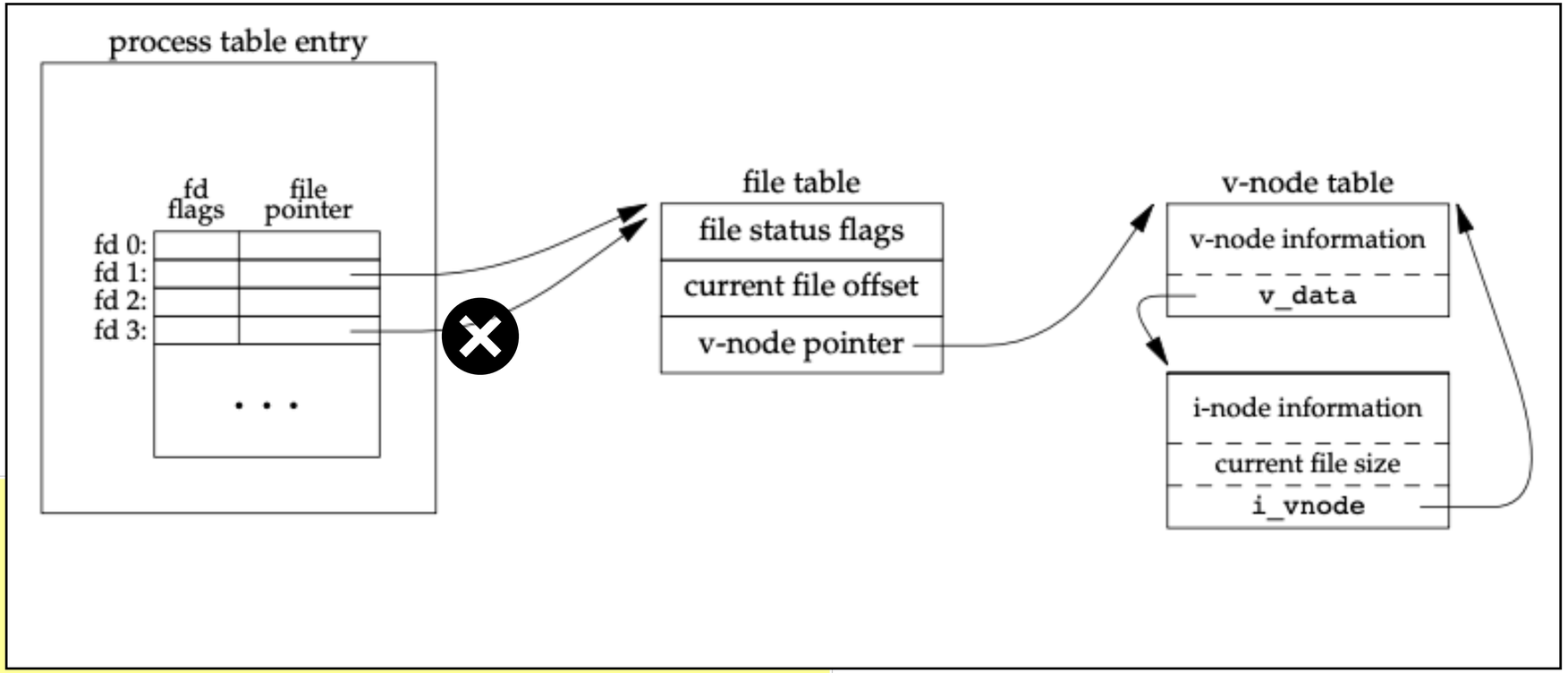
We are redirecting fd1 to fd2 (fd1 now points to the same open file table entry as fd2). So the second read uses the file position offset of fd2. c='o'.

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, the printf will output "Hello %d" to the file.

What is the output of the following program?

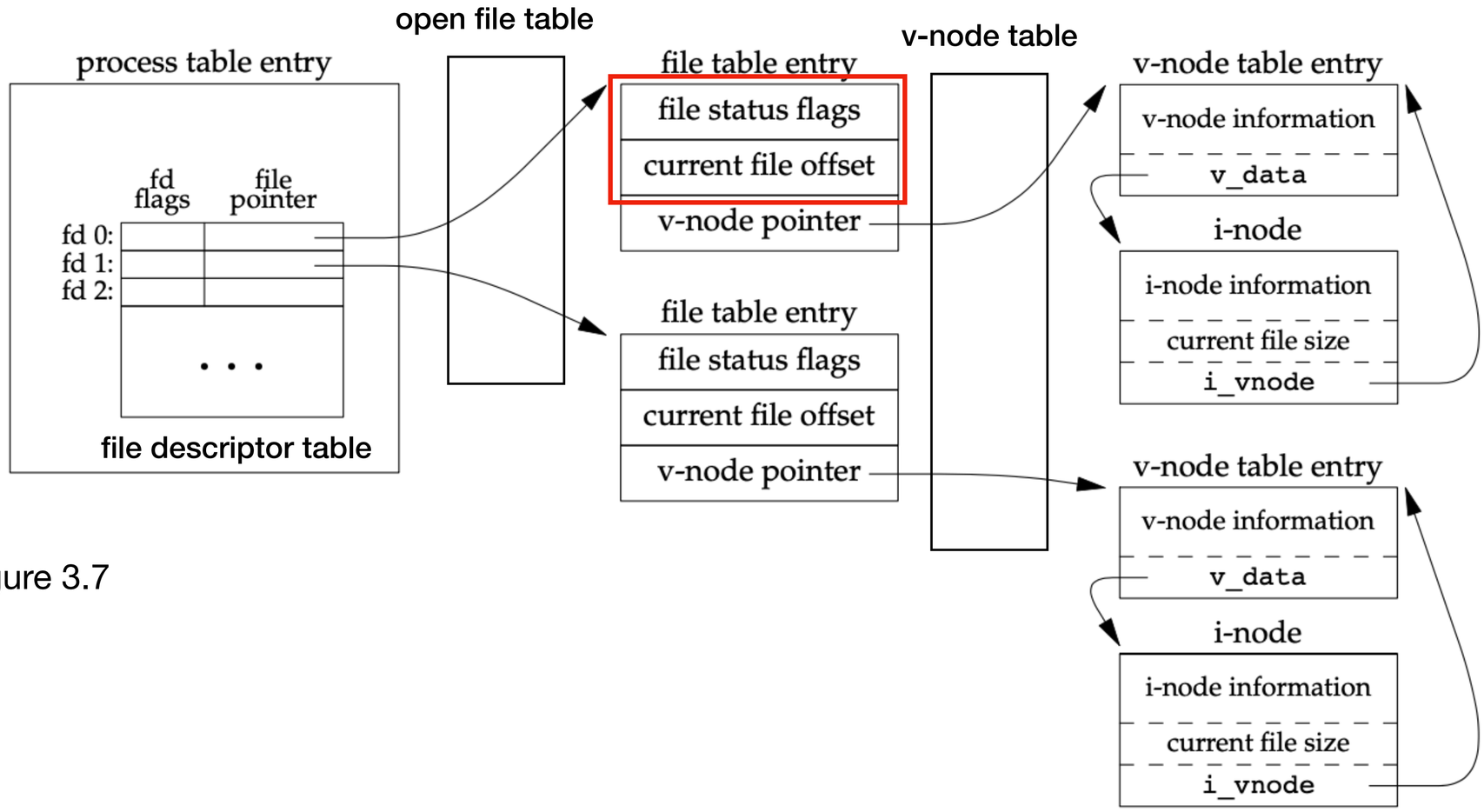



```
int main()
{
  int fd;
  char *s;
  fd = open("file", O_WRONLY | O_CREAT | O_TRUNC, 0666);
  dup2(fd,1);
  close(fd);
  printf("Hello %d\n", fd);
}
```

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);
}
```

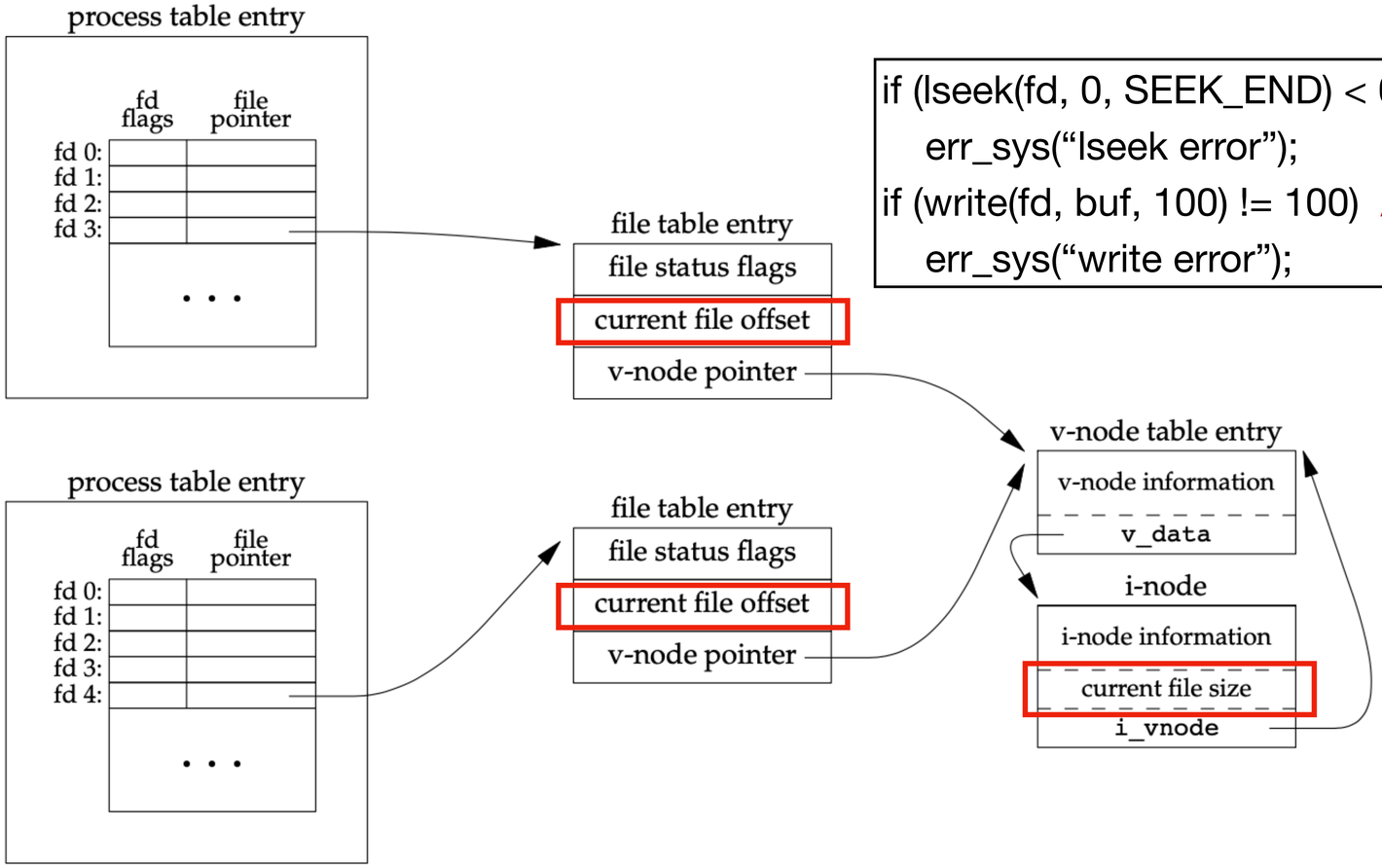
Rewind: Unix kernel support for File I/O



From APUE 3rd Edition: Figure 3.7

Figure 3.7 Kernel data structures for open files

File I/O - Atomic Operations




```
if (lseek(fd, 0, SEEK_END) < 0)  /* update current file offset position to EOF */
    err_sys("lseek error");
if (write(fd, buf, 100) != 100)  /* write to the file, and update the file size */
    err_sys("write error");
```

Here we assume fd is opened without the O_APPEND flag

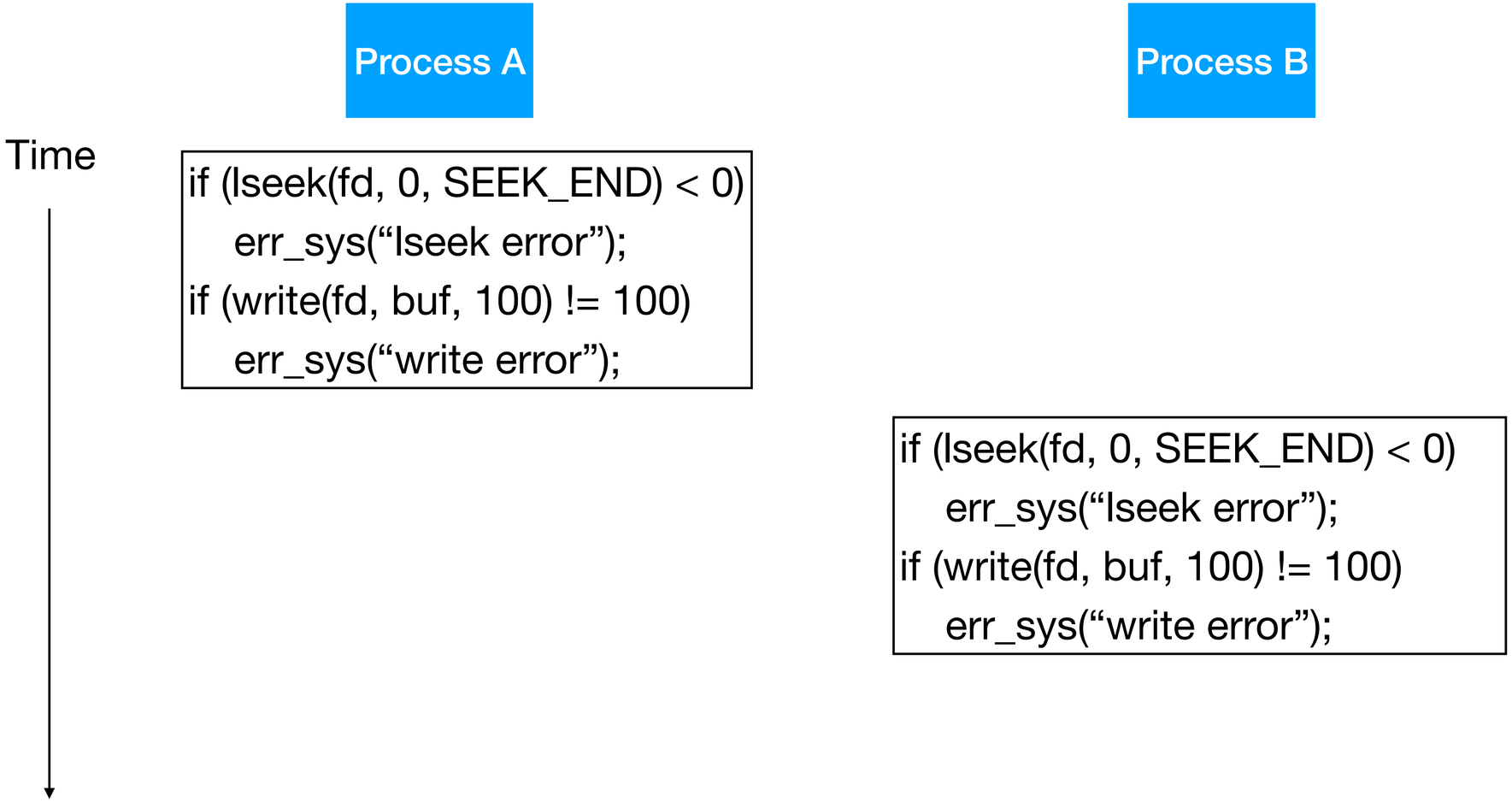
Figure 3.8 Two independent processes with the same file open

File I/O - Atomic Operations

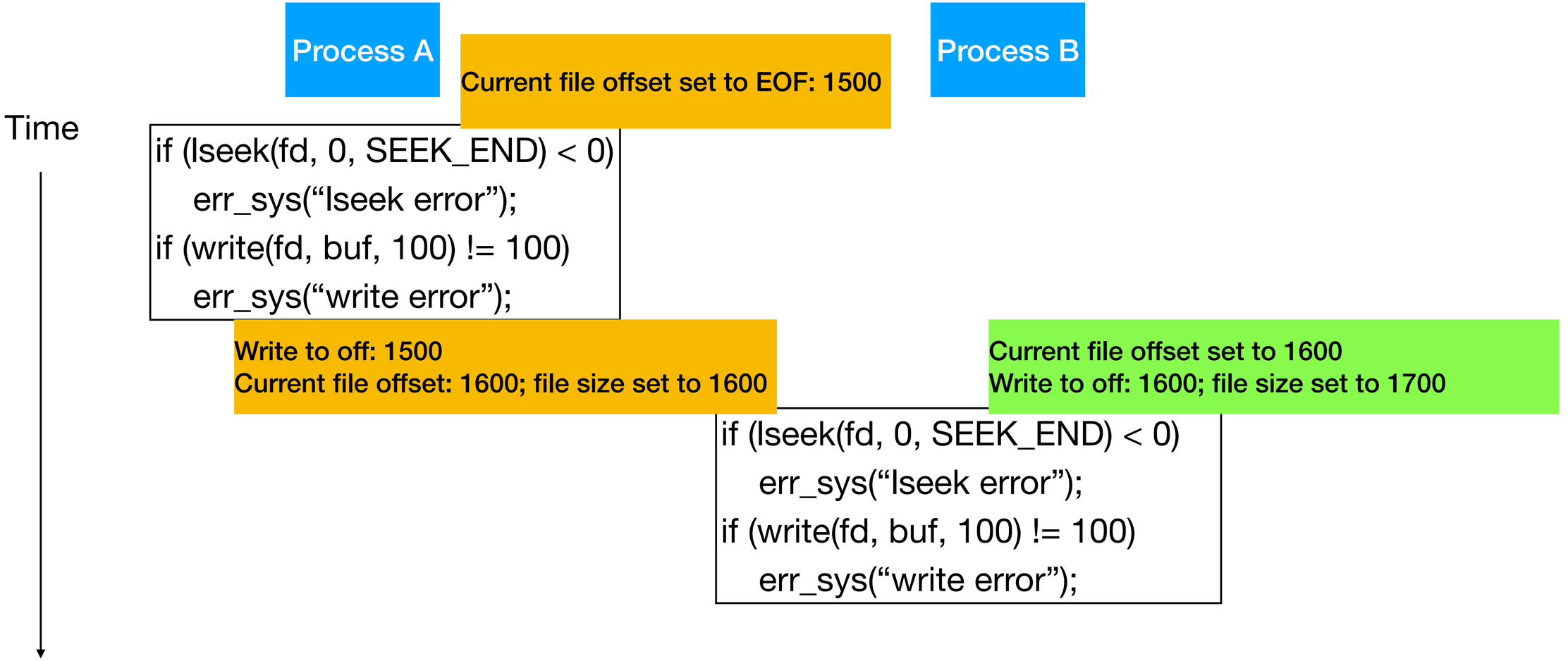
```
// Appending to a file
if (lseek(fd, 0, SEEK_END) < 0)  /* position to EOF */
    err_sys("lseek error");
if (write(fd, buf, 100) != 100)  /* and write */
    err_sys("write error");
```

- Consider two processes, A and B, execute the same code above at the same time and write to the same file
- Q: what will the file look like after A and B execute the code?

File I/O - Atomic Operations



File I/O - Atomic Operations



File I/O - Atomic Operations

Process A

Process B

Time

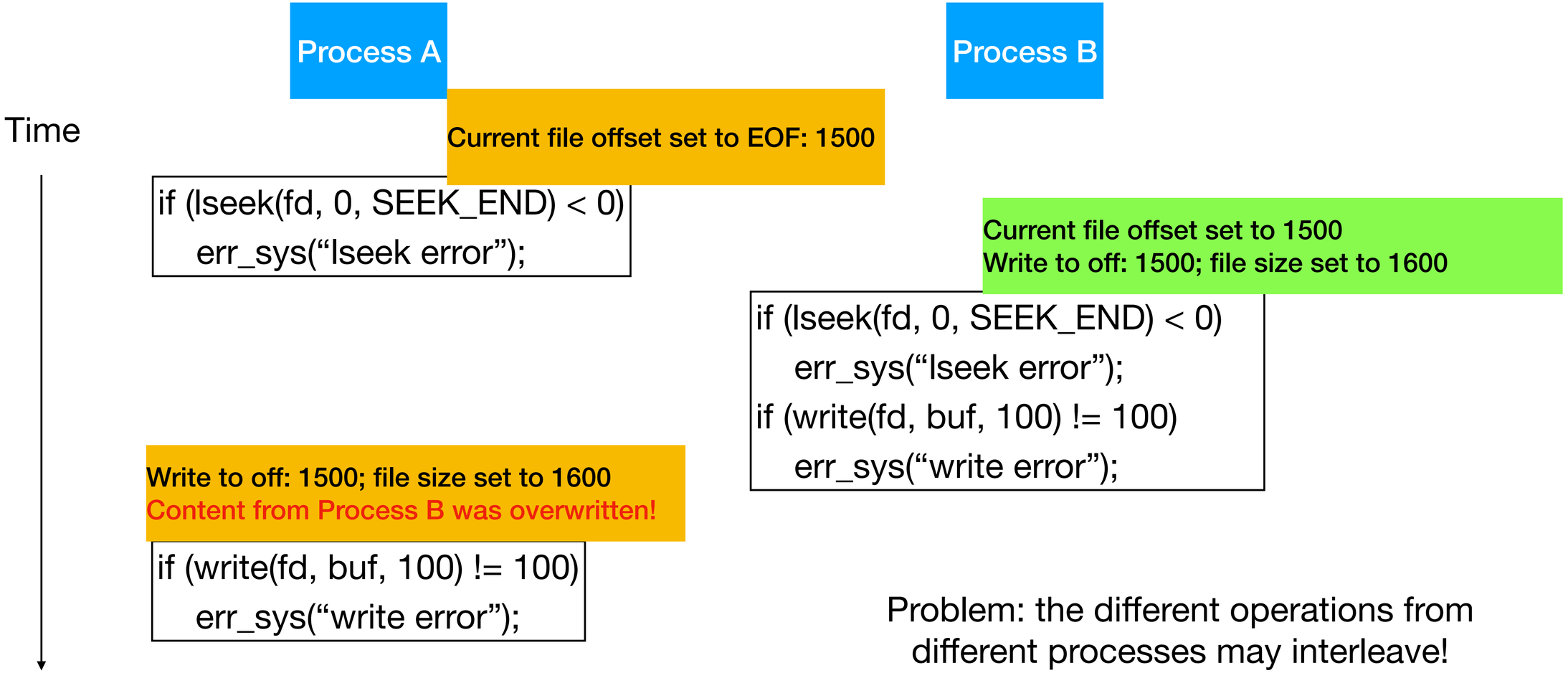
```
if (lseek(fd, 0, SEEK_END) < 0)
    err_sys("lseek error");
```

```
if (write(fd, buf, 100) != 100)
    err_sys("write error");
```

Question: why do we have to consider the case here?

```
if (lseek(fd, 0, SEEK_END) < 0)
    err_sys("lseek error");
if (write(fd, buf, 100) != 100)
    err_sys("write error");
```

File I/O - Atomic Operations



Problem: the different operations from different processes may interleave!
What can we do?

File I/O - Atomic Operations

- Atomic operations (from osdev.org)
  - An atomic operation is always executed without any other process being able to read or change the state that is read or changed during the operation—it is *effectively executed as a single step*…
- When should an operation be atomic?
  - When the result of one operation depends on the values of any shared resources, including global variables, file status, etc, accessible by many other running programs
- Unix supports atomic operations:
  - Recall the flag *O_APPEND* that positions the file to its current end of file before each write; in the previous example, if the file was open with the *O_APPEND* flag set, there's no need to call the lseek() before each *write()*

Review: File I/O: creat

```
#include <fcntl.h>
//Returns a file descriptor on a successful call; and -1 on an error
int creat(const char *path, mode_t mode);
```

- A file is *created and opened* ***write-only*** by calling the ***creat*** function

```
O_TRUNC
       If the file already exists and is a regular file and the
       access mode allows writing (i.e., is O_RDWR or O_WRONLY)
       it will be truncated to length 0.  If the file is a FIFO
       or terminal device file, the O_TRUNC flag is ignored.
       Otherwise, the effect of O_TRUNC is unspecified.
```

- This functio…*en()*
  - *open(path, O_WRONLY | O_CREAT | O_TRUNC, mode)* works the same
- Question: what if you want to create a file for read and write using creat?

File I/O - Atomic Operations

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

ENOENT O_CREAT is not set and the named file does not exist.

- Q: what's problematic here?

From APUE 3rd Edition: Section 3.11

File I/O - Atomic Operations

Process A

Process B

Time

open(path, O_WRONLY);

O_CREAT: If the file exists, this flag has no effect… Otherwise, the file shall be created

https://linux.die.net/man/3/open

```
fd = open(path, O_WRONLY | O_CREAT);
if (fd != -1)
    write(path, buf, sizeof(buf));
```

creat(path, mode);

- *open(path, O_WRONLY | O_CREAT | OTRUNC, mode)* works the same as *creat(path, mode)*

What can we do? We'll discuss next week.