

# **CS241 System Programming**

## **File System III**

Klara Nahrstedt

Lecture 22

3/13/2006



# Content

- Reading and Writing in UNIX
- Opening and Closing Files
- Select Function
- File Representation in UNIX
- Summary

# Administrative

- R: Ch 4 pp92-135
- MP3 is posted, due April 3, 2006
- Quiz 6 is March 17, 2006
  - Topics covered for Quiz 6:
    - Tanenbaum: 3.1-3.3 (I/O and Deadlock)
    - Tanenbaum: 5.1-5.2 (File and Directories)
    - R&R: 4.1., 4.2, 4.3, 4.4, 4.6.1, 4.6.2
    - Lecture Notes up to Monday (March 13)

# Reading and Writing

- Sequential access to files and other devices give by **read** and **write** functions
- A read operation
  - Attempts to retrieve `nbytes` from the file or device `files` into the user variable `buf`
  - May return fewer bytes than requested if, e.g. it reaches end-of-file before completely satisfying the request.
  - Returns 0 to indicate end-of-file for a regular file

# Sequential Files

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

```
ssize_t read(int fildes, void *buf,  
             size_t nbyte);
```

- Returns number of bytes read in buf
- \*buf must be initialized.
- read on pipe may return < nbytes
  - pipe – simplest UNIX inter-process communication mechanism, represented by a special file

# Standard files

- `stdin` – standard input, file descriptor `STDIN_FILENO` (legacy code standard `fd = 0`)
- `stdout` – standard output, file descriptor `STDOUT_FILENO` (`fd = 1`)
- `stderr` – standard output, file descriptor `STDERR_FILENO` (`fd = 2`)
- Use always symbolic names rather than the numeric values

# Sequential write operation

```
#include <unistd.h>
ssize_t write(int fildes, const void *buf,
size_t nbyte);
```

- Returns number of bytes actually written
- Can only write to end of file (or file truncates)

```
#define BLKSIZE 1024
char buf[BLKSIZE];
read(STDIN_FILENO, buf, BLKSIZE);
write(STDOUT_FILENO, buf, BLKSIZE);
```

What can go wrong with the above code segment?

# Copyfile

```
#include <errno.h>
#include <unistd.h>
#define BLKSIZE 1024
int copyfile(int fromfd, int tofd) {
    char *bp;
    char buf[BLKSIZE];
    int bytesread;
    int byteswritten = 0;
    int totalbytes = 0;
    for ( ; ; ) {
        while (((bytesread = read(fromfd, buf, BLKSIZE)) == -1) &&
              (errno == EINTR)) ; /* handle interruption by signal */
        if (bytesread < 0) /* real error or end-of-file on fromfd */
            break;
        bp = buf;
```

# Copyfile

```
while (bytesread > 0) {
    while(((byteswritten = write(tofd, bp, bytesread)) == -1 ) &&
        (errno == EINTR)) ;           /* handle interruption by signal */
    if (byteswritten <= 0)             /* real error on tofd */
        break;
    totalbytes += byteswritten;
    bytesread -= byteswritten;
    bp += byteswritten;
}
if (byteswritten == -1)                /* real error on tofd */
    break;
}
return totalbytes;
}
```

# Use the restart library for read/write

## P99 – e.g. r\_write

```
ssize_t r_write(int fd, void *buf, size_t size) {
    char *bufp;
    size_t bytestowrite;
    ssize_t byteswritten;
    size_t totalbytes;
    for (bufp = buf, bytestowrite = size, totalbytes = 0;
        bytestowrite > 0;
        bufp += byteswritten, bytestowrite -= byteswritten) {
        byteswritten = write(fd, bufp, bytestowrite);
        if ((byteswritten) == -1 && (errno != EINTR))
            return -1;
        if (byteswritten == -1)
            byteswritten = 0;
        totalbytes += byteswritten;
    }
    return totalbytes;
}
```

# Restart functions make programs simpler!

```
#include <unistd.h>
#include "restart.h"
#define BLKSIZE 1024
int copyfile(int fromfd, int tofd) {
    char buf[BLKSIZE];
    int bytesread, byteswritten;
    int totalbytes = 0;
    for ( ; ; ) {
        if ((bytesread = r_read(fromfd, buf, BLKSIZE)) <= 0)
            break;
        if ((byteswritten = r_write(tofd, buf, bytesread)) == -1)
            break;
        totalbytes += byteswritten;
    }
    return totalbytes;
}
```

# Use 'readblock' function

- Read a specific number of bytes
  - Return 0 if end-of-file occurs before any bytes are read
  - Return `size` if 'readblock' successful
  - Return -1 in case of error and sets `errno`
- Use it to read structures
- Either works or returns error

# Opening and Closing Files

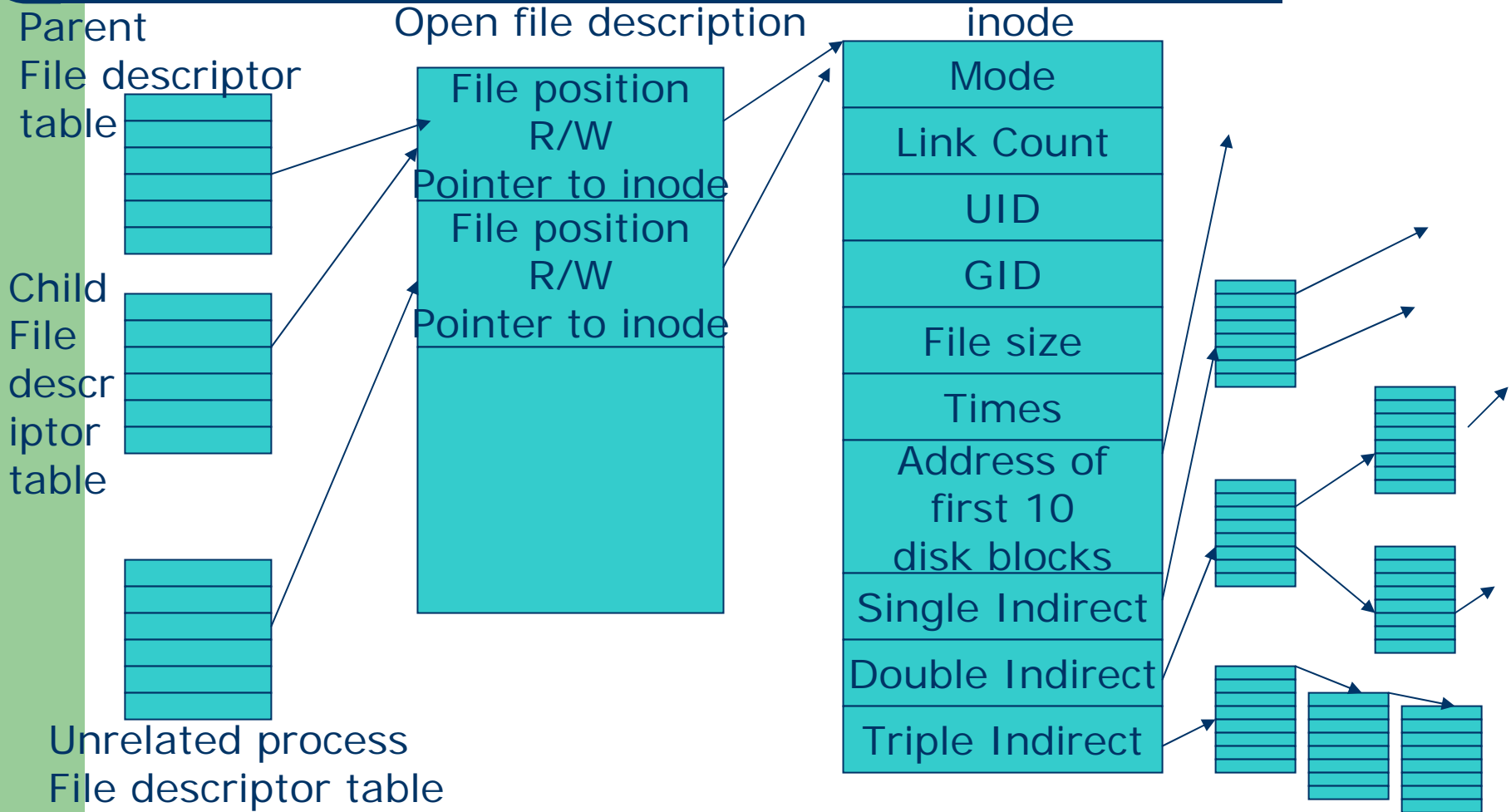
- Open associates a file descriptor with a file or physical device
- File descriptor is an index into a file descriptor table per process and can be inherited to child processes allowing sharing
- One should close open files
- Oflag access modes and status

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

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

```
int open(const char *path, int oflag, ...)
```

# UNIX file structure implementation



# File Access Example

```
int fd;  
mode_t fdmode = (S_IRUSR | S_IWUSR |  
S_IRGRP | S_IROTH);
```

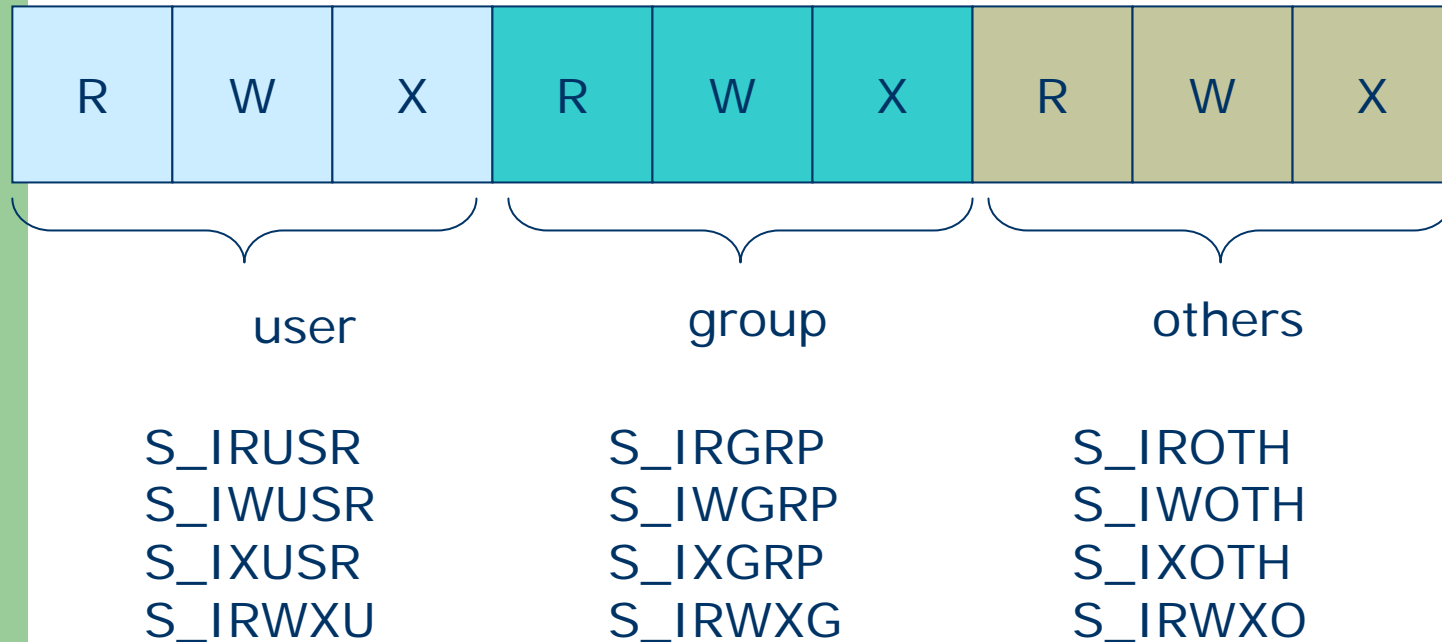
```
If ((fd = open("info.dat", O_RDWR | O_CREAT,  
fdmode)) == -1) perror("Failed to open info.dat");
```

Opens a file info.dat in current directory rewriting any existing file data if present.

# Additional Oflags

- O\_APPEND – file offset to EOF
- O\_CREAT – need to give permissions
- O\_EXCL – use with CREATE detects file
- O\_NOCTTY – prevents device from becoming controlling terminal
- O\_NONBLOCK – return immediately
- O\_TRUNC – file to beginning for write

# Access Control sys/stat.h



S\_ISUID – set user ID on execution  
S\_ISGID – set group ID on execution

# Close

```
#include <errno.h>
#include <unistd.h>
int r_close(int fd) {
    int retval;
    while ((retval = close(fd)) == -1 &&
           errno == EINTR) ;
    return retval;
}
```

# SELECT Function

- Handling of I/O from multiple sources is an important problem
  - Examples
    - Program may want to overlap terminal I/O with reading input from a disk or with printing
    - Program expects input from two different sources, but it does not know which input will be available first
      - **Problem: if we read from resource A and input comes on resource B, process blocks**
      - **Solution: Need to block until input from either source becomes available**
- First method of monitoring multiple file descriptors: - **use a separate process (monitorfork.c)**
- Second method of monitoring multiple file descriptors from a single process : - **use Select statement**
  - can select input from one of a number of different files

# Select Function (2)

- `#include <sys/select.h>`

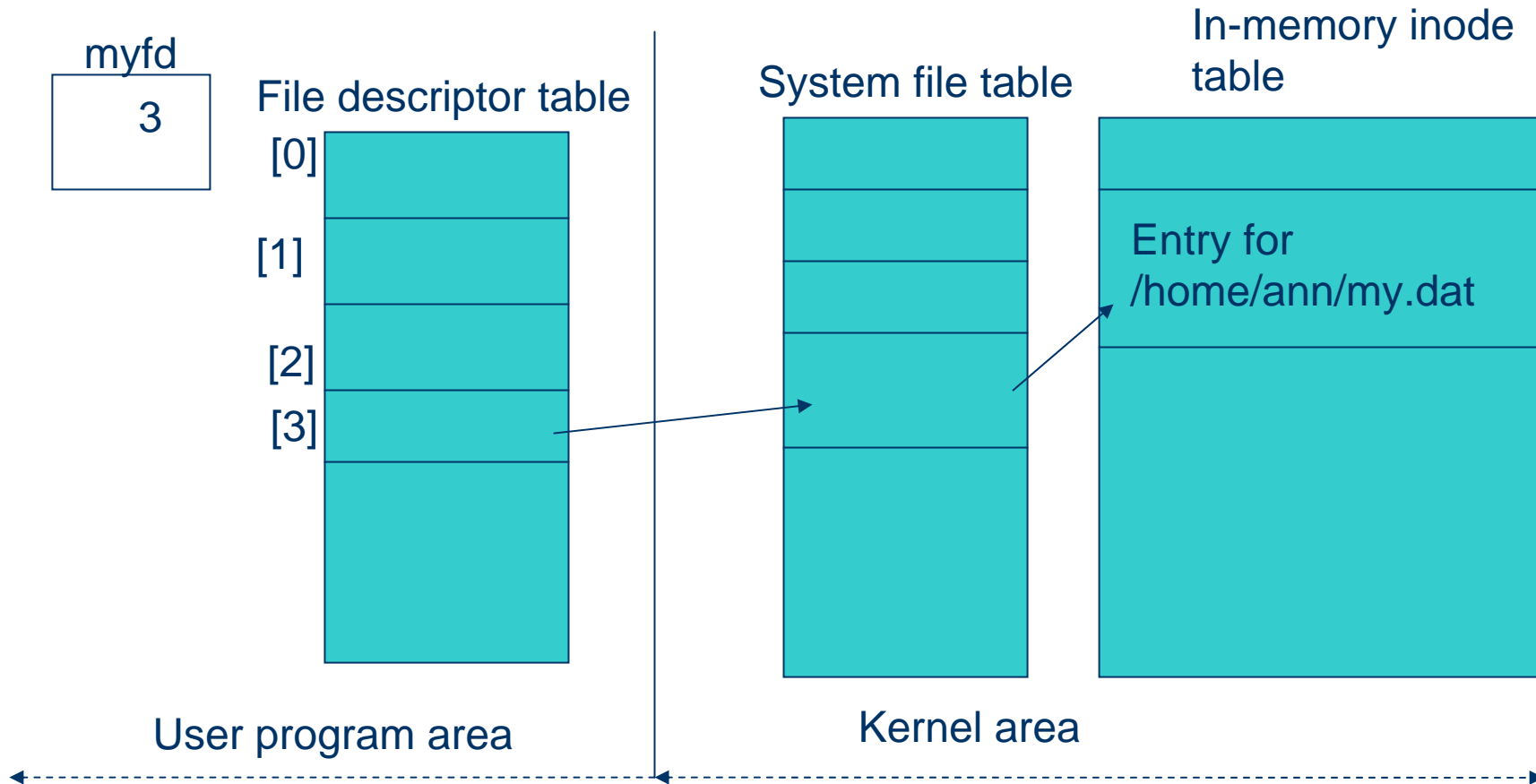
```
int select(int nfd,  
          fd_set *restrict readfds,  
          fd_set *restrict writefds,  
          fd_set *restrict errorfds,  
          struct timeval *restrict timeout);
```

# File Representation

- Files are designated within C programs either by file pointers or by file descriptors
- The UNIX I/O library functions (**open, read, write, close, ioctl**)
  - **use file descriptor**
- The standard I/O library function (**fopen, fscanf, fprintf, fread, fwrite, fclose,...**)
  - **use file pointers.**

# File Descriptors

```
Myfd = open("/home/ann/my.dat", O_RDONLY);
```

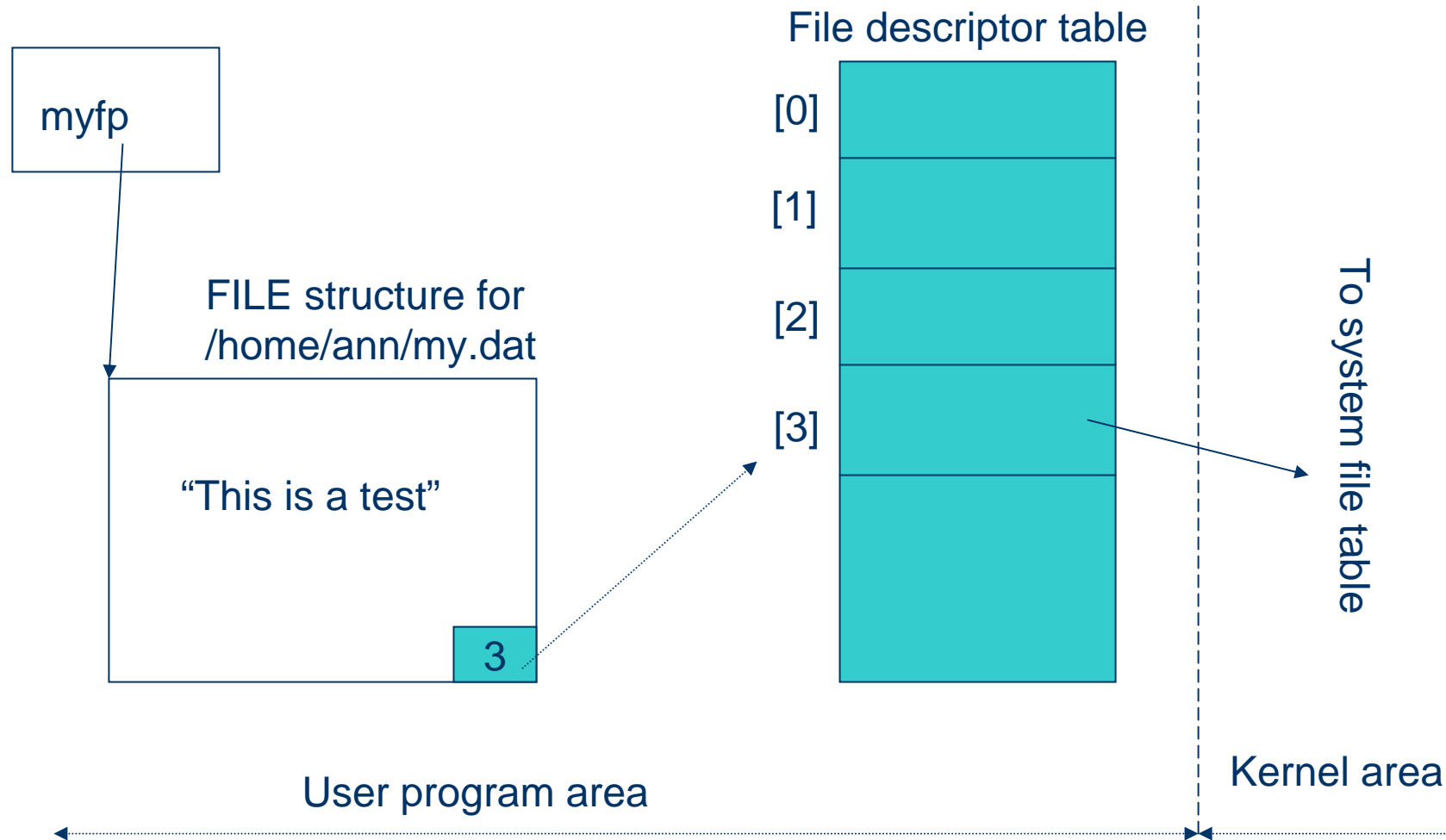


# File Pointers and buffering

- ISO C standard I/O Library uses file pointers not file descriptors (p122)
- File pointer points to a data structure called a **FILE** structure in the user area of the process
- “This is a test” message is written into a buffer in FILE structure (if buffer fills up, the I/O subsystem calls write with the file descriptor to write to the disk)

```
FILE *myfp;  
if ((myfp = fopen("/home/ann/my.dat", "w")) == NULL  
    perror("Failed to open /home/ann/my.dat");  
else  
    fprintf(myfp, "This is a test");
```

# Schematic Handling of a file pointer after `fopen`



# Problems with FILE pointers

- Buffered data may get lost during program crash
- How can a program avoid the effect of buffering?
- Use `fflush` - force whatever has been buffered in the FILE structure to be written out.
- Even more subtle problems due to
  - OS Buffer cache

# Summary

- UNIX example
- File descriptors vs File pointers
- Tables for Files