OPerating Systems Laboratory
OPS*Lab*

Programming in the UNIX Environment
(Part 4.2.1)

Prof. Marco Autili
University of L'Aquila

Top-level Makefile

Makefile

```
DIRS = lib chap1-intro
#chap3-fileio chap4-filedir chap5-stdio chap8-proc chap15-ipc1

 # Make with no arguments executes the first rule in the file,
 # hence it executes the following rule "all" as first
 # To substitute a variable's value, write a dollar sign followed
 # by the name of the variable in parentheses or braces:
 # either `$(foo)' or `${foo}' is a valid reference to the variable foo.
 # This special significance of `$' is why you must write `$$' to have
 # the effect of a single dollar sign in a file name or command.
 #
 # N.B.: Recursive make commands should always use the variable MAKE,
 # not the explicit command name 'make'
all:
    for i in $(DIRS); do \
        (cd $$i && echo "making $$i" && $(MAKE) ) || exit 1; \
    done

# Make clean executes the following rule
clean:
    for i in $(DIRS); do \
        (cd $$i && echo "cleaning $$i" && $(MAKE) clean) || exit 1; \
    done
```

make will be called in each subdirectory in DIRS

From Part 2 and Part 3:
run the next command if the preceding command succeeded, and all the commands correctly set exit codes

https://www.gnu.org/software/make/manual/html_node/MAKE-Variable.html

PART 42 - System-Progr — -bash — 117×10

```
iMac:PART 42 - System-Progr marcoautili$ make
for i in lib chap1-intro ; do \
                (cd $i && echo "making $i" && /Applications/Xcode.app/Contents/Developer/usr/bin/make ) || exit 1; \
        done
making lib
make[1]: Nothing to be done for `all'.
making chap1-intro
gcc -I../include -Wall -DMACOS -D_DARWIN_C_SOURCE  mycat.c -o mycat  -L../lib -lapue
iMac:PART 42 - System-Progr marcoautili$
iMac:PART 42 - System-Progr marcoautili$
```

make is going to be called in the directory chap1-intro

Makefile

```
1  ROOT=..
2  PLATFORM=$(shell $(ROOT)/systype.sh)
3  include $(ROOT)/Make.defines.$(PLATFORM)
4
5  # The variable contains the names of all source files to be compiled separated by a blank space,
6  PROGS = mycat
7  # hellofrompid uidgid ls1 shell1 shell2
8
9  # Make with no arguments executes the first rule in the file, hence the followoing rule all
10 all: $(PROGS)
11
12 # The following rule applies to all files.
13 # The rule says that each file depends upon the .c version of the file and
14 # the library LIBAPUE=$(ROOT)/lib/libapue.a (LIBAPUE is defined in the files Make.defines.xxx)
15 %: %.c $(LIBAPUE)
16     $(CC) $(CFLAGS) $@.c -o $@ $(LDFLAGS) $(LDLIBS)
17
18 clean:
19     rm -f $(PROGS) $(TEMPFILES) *.o
20
21 # The following include is used to generate a so called
22 # Automatic Prerequisite with the library LIBAPUE=$(ROOT)/lib/libapue.a
23 # Search "GNU make: Automatic Prerequisites" on the web
24 include $(ROOT)/Make.libapue.inc
25
```

Note that the .c extension is added below

Line 1, Column 1 — Tab Size: 4

Determining the OS type

systype.sh

systype.sh 〉No Selection

```
 # (leading space required for Xenix /bin/sh)

#
# Determine the type of *ix operating system that we're
# running on, and echo an appropriate value.
# This script is intended to be used in Makefiles.
# (This is a kludge.  Gotta be a better way.)
#

case `uname -s` in
"FreeBSD")
    PLATFORM="freebsd"
    ;;
"Linux")
    PLATFORM="linux"
    ;;
"Darwin")
    PLATFORM="macos"
    ;;
"SunOS")
    PLATFORM="solaris"
    ;;
*)
    echo "Unknown platform" >&2
    exit 1
esac
echo $PLATFORM
exit 0

```

chap1-intro — -bash — 79×10

```
[valintina-vaio:chap1-intro marcoautili$
[valintina-vaio:chap1-intro marcoautili$ uname
 Darwin
[valintina-vaio:chap1-intro marcoautili$
[valintina-vaio:chap1-intro marcoautili$
[valintina-vaio:chap1-intro marcoautili$ uname -s
 Darwin
[valintina-vaio:chap1-intro marcoautili$
```

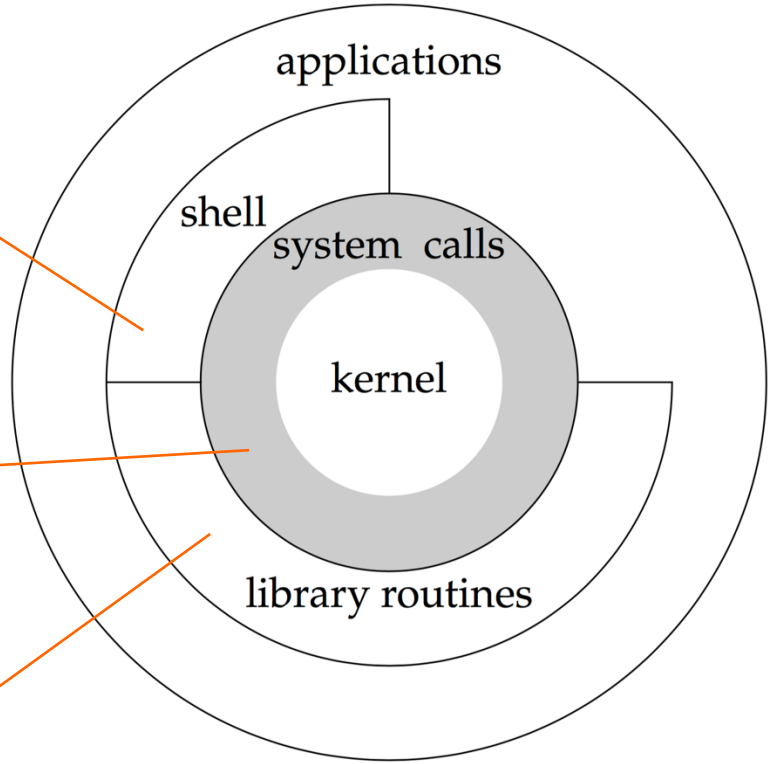
UNIX Architecture (from Part 1)

- An operating system can be defined as the software that controls the hardware resources of the computer and provides an environment for running programs

Special application(s) that provides an interface for running other applications (commands, scripts, …)

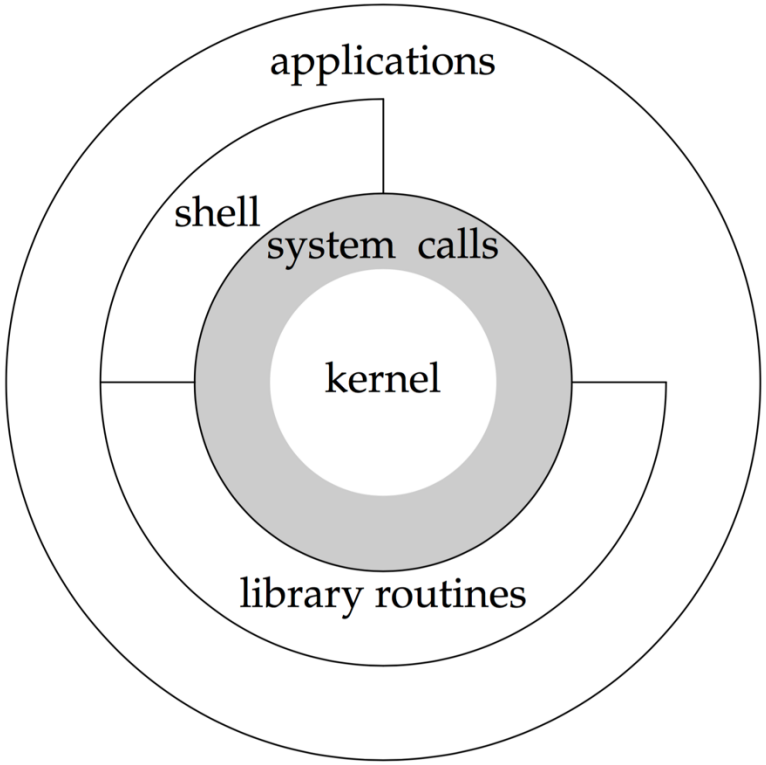
Interface to the kernel

Common functions built on top of the system call interface



In a broader sense… (from Part 1)

- An operating system consists of the kernel and all the other software that makes a computer useful and gives the computer its personality. This other software includes system utilities, applications, shells, libraries of common functions, and so on



- There exist a number of variants of Unix (i.e., UNIX-like systems), notably,
  - GNU/Linux OS: Linux is the kernel used by the GNU operating system
    - often referred to as simply Linux)
  - macOS, developed by Apple Inc., OS family Macintosh – Unix - BSD
    - its core set of components is based upon Apple's open-source Darwin operating system
  - Darwin is a Unix-like operating system

System Calls and Library Functions

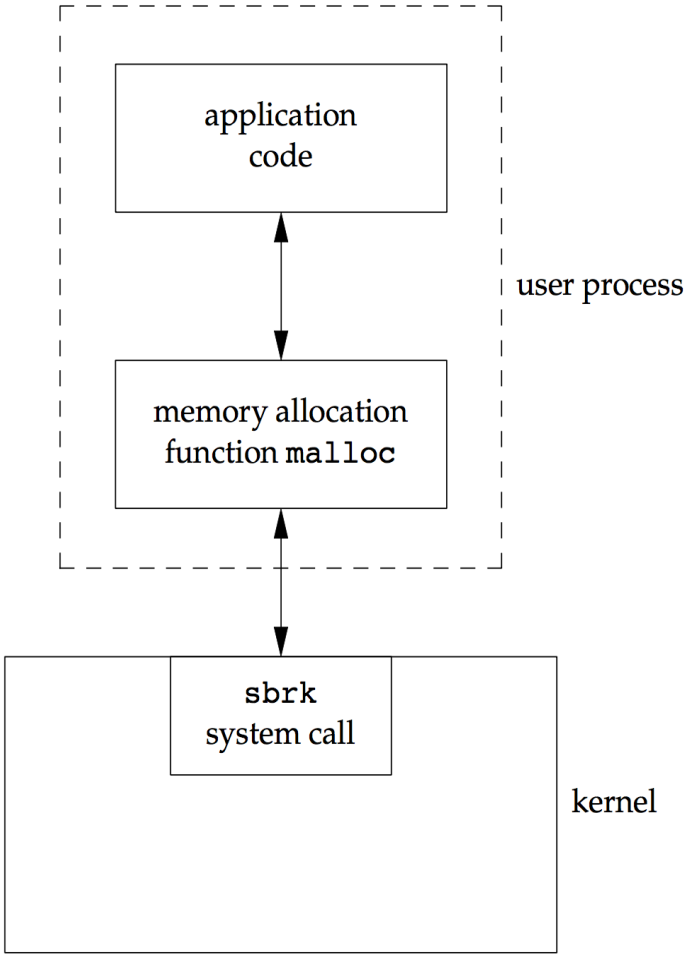


Figure 1.11 Separation of `malloc` function and `sbrk` system call

- The UNIX system call that handles memory allocation, sbrk(2), is not a general-purpose memory manager
  - It increases or decreases the address space of the process by a specified number of bytes
- The memory allocation function, malloc(3), implements one particular type of allocation
  - If we don't like its operation, we can define our own malloc function, which will probably use the sbrk system call
  - In fact, numerous software packages implement their own memory allocation algorithms with the sbrk system call

System Calls and Library Functions

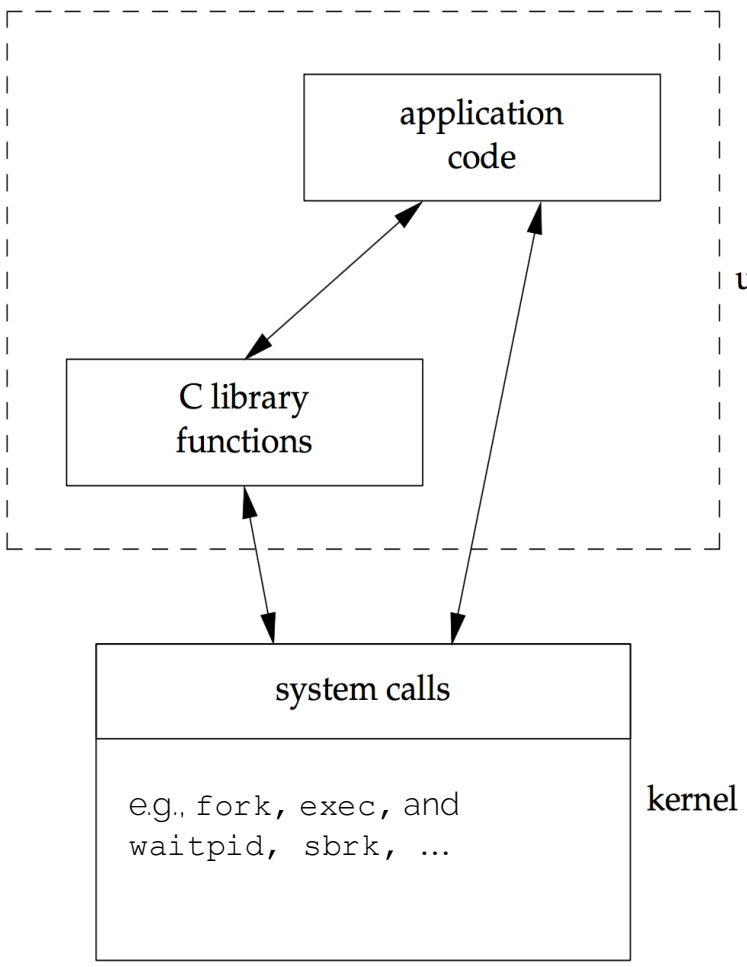


Figure 1.12 Difference between C library functions and system calls

- Another difference between system calls and library functions is that system calls usually provide a minimal interface, whereas library functions often provide more elaborate functionality
  - sbrk(2) VS malloc(3)
  - unbuffered I/O VS standard I/O functions
- Often, the term function refers to both system calls and library functions, except when the distinction is necessary

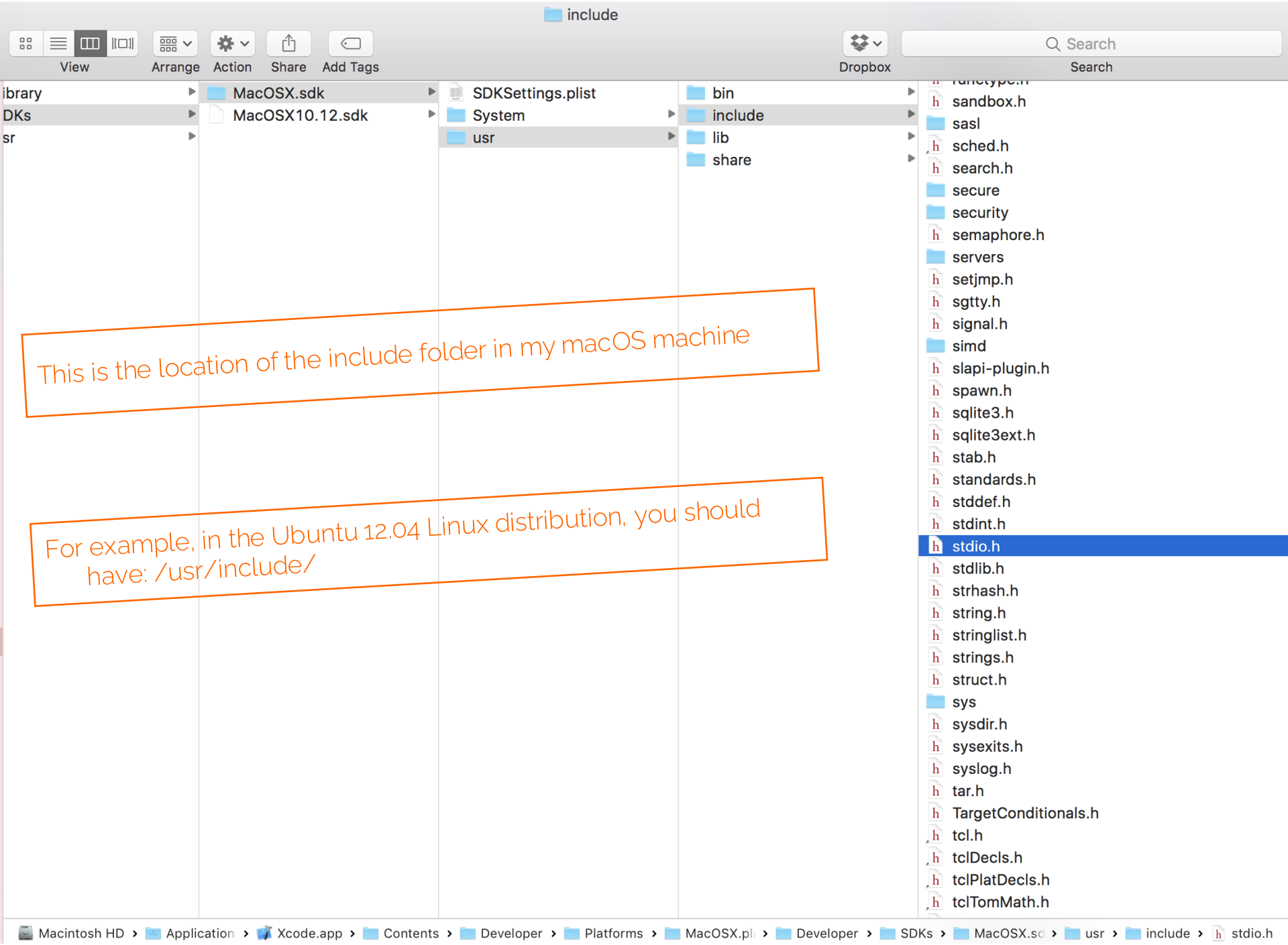
C/C++ Compiler (from PART 4.1)

- gcc --version or cc --version
  or g++ --version or clang++ --version

```
bin — -bash — 113×24
Last login: Fri Sep  9 10:17:57 on ttys001
iMac-2:~ marcoautili$ gcc --version
Configured with: --prefix=/Applications/Xcode.app/Contents/Developer/usr --with-gxx-include-dir=/Applications/Xco
de.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX10.11.sdk/usr/include/c++/4.2.1
Apple LLVM version 7.3.0 (clang-703.0.31)
Target: x86_64-apple-darwin15.6.0
iMac-2:~ marcoautili$ cd /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin
iMac-2:bin marcoautili$ ls -la
total 126624
drwxr-xr-x  71 root  wheel      2414 May  4 13:32 .
drwxr-xr-x   7 root  wheel       238 May  4 13:32 ..
-rwxr-xr-x   1 root  wheel     33488 Mar 16 01:31 ar
-rwxr-xr-x   1 root  wheel     31712 Mar 16 01:31 as
-rwxr-xr-x   1 root  wheel     18080 Mar 16 01:31 asa
-rwxr-xr-x   1 root  wheel    227840 Mar 16 01:31 bison
-rwxr-xr-x   1 root  wheel    148576 Mar 16 01:31 bitcode_strip
lrwxr-xr-x   1 root  wheel         5 Mar 22 13:52 c++ -> clang
-rwxr-xr-x   1 root  wheel     23040 Mar 16 01:31 c89
-rwxr-xr-x   1 root  wheel     23120 Mar 16 01:31 c99
lrwxr-xr-x   1 root  wheel         5 Feb  3  2016 cc -> clang
-rwxr-xr-x   1 root  wheel  49850480 Apr 27 02:02 clang
lrwxr-xr-x   1 root  wheel         5 Feb  3  2016 clang++ -> clang
-rwxr-xr-x   1 root  wheel    118896 Mar 16 01:31 cmpdylib
-rwxr-xr-x   1 root  wheel    144400 Mar 16 01:31 codesign_allocate
```

In my MacOSX machine, header files are located here (as part of the Xcode installation)

(see next slide)


include
View
Arrange
Action
Share
Add Tags
Dropbox
Search
Search
ibrary
DKs
sr
MacOSX.sdk
MacOSX10.12.sdk
SDKSettings.plist
System
usr
bin
include
lib
share
sandbox.h
sasl
sched.h
search.h
secure
security
semaphore.h
servers
setjmp.h
sgtty.h
signal.h
simd
slapi-plugin.h
spawn.h
sqlite3.h
sqlite3ext.h
stab.h
standards.h
stddef.h
stdint.h
stdio.h
stdlib.h
strhash.h
string.h
stringlist.h
strings.h
struct.h
sys
sysdir.h
sysexits.h
syslog.h
tar.h
TargetConditionals.h
tcl.h
tclDecls.h
tclPlatDecls.h
tclTomMath.h
This is the location of the include folder in my macOS machine
For example, in the Ubuntu 12.04 Linux distribution, you should have: /usr/include/
Macintosh HD
Application
Xcode.app
Contents
Developer
Platforms
MacOSX.pl
Developer
SDKs
MacOSX.sd
usr
include
stdio.h

Function prototype declaration

- Many programming languages implement a printf function to output a formatted string.
  It originated from the C programming language, where it has a prototype similar to the following:

```
int printf(const char *format, ...);
```

- The constant string format provides a description of the output, with placeholders marked by % escape characters, to specify both the relative location and the type of output that the function should produce.

- The last argument "..." is the ISO C way to specify that the number and types of the remaining arguments may vary.

- The return value yields the number of printed characters.
  - For example,

```
printf("Color %s, number1 %d, number2 %05d, hex %#x,
        float %5.2f, unsigned value %u.\n",
        "red", 123456,89, 255, 3.14159, 250);
```

  - will print the following line (including new-line character, \n):

```
Color red, number1 123456, number2 00089, hex 0xff, float 3.14, unsigned
value 250.
```

https://en.wikipedia.org/wiki/Printf_format_string

Chapter 1
Chapter 3
Chapter 4

Introduction

File I/O

Files and Directories

Input and Output

File Descriptors

- File descriptors are normally small non-negative integers that the kernel uses to identify the files accessed by a process

- Whenever an existing file is opened or a new file is created, the kernel returns a file descriptor that we use when we want to read or write the file

Unbuffered I/O

- Unbuffered I/O is provided by the functions open, read, write, lseek, and close

- These functions all work with file descriptors
  - most file I/O on a UNIX-like system can be performed using only these functions

Input and Output

- By convention, UNIX System shells associate:
  - file descriptor 0 with the standard input of a process
  - file descriptor 1 with the standard output
  - file descriptor 2 with the standard error

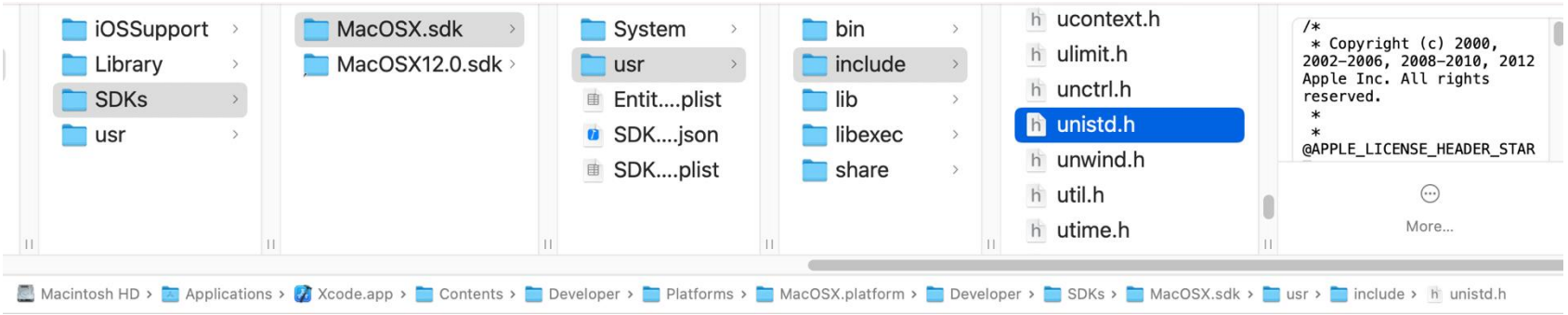
- This convention is used by the shells and many applications; it is not a feature of the UNIX kernel
  - Nevertheless, many applications would break if these associations weren't followed

- Although their values are standardized by POSIX.1, the numbers 0, 1, and 2 should be replaced in POSIX-compliant applications with the symbolic constants STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO to improve readability. These constants are usually defined in the <unistd.h>

See next slide

- File descriptors range from 0 through OPEN_MAX–1

unistd.h



h unistd.h

macOS 12.0 〉 usr/include 〉 h unistd.h 〉 No Selection

```
68 #ifndef _UNISTD_H_
69 #define _UNISTD_H_
70
71 #include <_types.h>
72 #include <sys/unistd.h>
73 #include <Availability.h>
74 #include <sys/_types/_gid_t.h>
75 #include <sys/_types/_intptr_t.h>
76 #include <sys/_types/_off_t.h>
77 #include <sys/_types/_pid_t.h>
78 /* DO NOT REMOVE THIS COMMENT: fixincludes needs to see:
79  * _GCC_SIZE_T */
80 #include <sys/_types/_size_t.h>
81 #include <sys/_types/_ssize_t.h>
82 #include <sys/_types/_uid_t.h>
83 #include <sys/_types/_useconds_t.h>
84 #include <sys/_types/_null.h>
85
86 #define  STDIN_FILENO   0   /* standard input file descriptor */
87 #define STDOUT_FILENO   1   /* standard output file descriptor */
88 #define STDERR_FILENO   2   /* standard error file descriptor */
89
```

Reading a file

- Data is read from an open file with the read function

- If the read is successful, the number of bytes read is returned

- If the End of File is encountered, 0 is returned

```
#include <unistd.h>

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

                                Returns: number of bytes read, 0 if end of file, –1 on error
```

- POSIX.1 changed the prototype for this function in several ways. The "classic" definition is

```
int read(int fd, char *buf, unsigned nbytes)
```

Writing a file

- Data is written to an open file with the write function

- The return value is usually equal to the nbytes argument, otherwise, an error has occurred

- A common cause for a write error is either filling up a disk or exceeding the file size limit for a given process

```
#include <unistd.h>

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

Returns: number of bytes written if OK, –1 on error

Input and Output (Unbuffered I/O sample)

Read from the standard input and write to the standard output

mycat.c

```c
#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);
}
```

Line 1, Column 1 Tab Size: 4 C

Number of bytes that have been just read

Return the number of bytes that have been just written

This example works for both text files and binary files, since there is no difference between the two to the UNIX kernel

The program does not close the input file or output file

Instead, the program uses the feature of the UNIX kernel that closes all open file descriptors in a process when that process terminates

mycat assumes that stdin and stdout have been set up by the shell before this program is executed

chap1-intro — -bash — 78×5

```
desktop-jisqlks:chap1-intro marcoautili$
desktop-jisqlks:chap1-intro marcoautili$
desktop-jisqlks:chap1-intro marcoautili$
desktop-jisqlks:chap1-intro marcoautili$ ./mycat < infile.txt > outfile.txt
desktop-jisqlks:chap1-intro marcoautili$
```

Input and Output (Unbuffered I/O sample)

infile.txt

```
1
2  This is a test file!
3  This is a test file!This is a test file!
4  This is a test file!This is a test file!This is a test file!
5
6
```

chap1-intro — -bash — 81×9

```
valintina-vaio:chap1-intro marcoautili$
valintina-vaio:chap1-intro marcoautili$ ./mycat < infile.txt

This is a test file!
This is a test file!This is a test file!
This is a test file!This is a test file!This is a test file!

valintina-vaio:chap1-intro marcoautili$
```

Redirecting stdin only

chap1-intro — -bash — 81×13

```
valintina-vaio:chap1-intro marcoautili$
valintina-vaio:chap1-intro marcoautili$ ./mycat
sto scrivendo sul terminale... ora faccio INVIO
sto scrivendo sul terminale... ora faccio INVIO


continuo a scrivere sul terminale
continuo a scrivere sul terminale


ora chiudo con CTRL^Dora chiudo con CTRL^D

valintina-vaio:chap1-intro marcoautili$
```

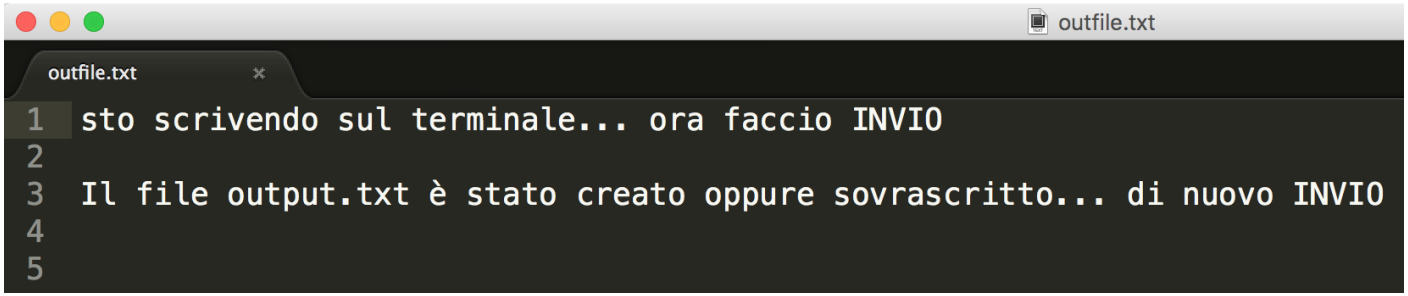
No redirection

Input and Output (Unbuffered I/O sample)

```
chap1-intro — -bash — 81×7
valintina-vaio:chap1-intro marcoautili$
valintina-vaio:chap1-intro marcoautili$ ./mycat > outfile.txt
sto scrivendo sul terminale... ora faccio INVIO

Il file output.txt è stato creato oppure sovrascritto... di nuovo INVIO

valintina-vaio:chap1-intro marcoautili$
```

Redirecting stdout only

N.B.: Obviously, the file outfile.txt is created by the shell (if not already existing) as soon as you type the first [INVIO] to launch ./mycat > outfile.txt



File information structure

```
struct stat {
  mode_t          st_mode;    /* file type & mode (permissions) */
  ino_t           st_ino;     /* i-node number (serial number) */
  dev_t           st_dev;     /* device number (file system) */
  dev_t           st_rdev;    /* device number for special files */
  nlink_t         st_nlink;   /* number of links */
  uid_t           st_uid;     /* user ID of owner */
  gid_t           st_gid;     /* group ID of owner */
  off_t           st_size;    /* size in bytes, for regular files */
  struct timespec st_atim;    /* time of last access */
  struct timespec st_mtim;    /* time of last modification */
  struct timespec st_ctim;    /* time of last file status change */
  blksize_t       st_blksize; /* best I/O block size */
  blkcnt_t        st_blocks;  /* number of disk blocks allocated */
};
```

Structure of information containing all the attributes of a file

Hard links!

st_mode is a number that encodes both the file type and the permissions

More on mode_t later in these slides

We will discuss st_blksize later

Errata corrige:
all of these should be xx_xtime,
e.g., st_atim → st_atime

File access permission to be passed to the mode parameter, used to set/change the st_mode field

st_mode mask	Meaning
S_IRUSR	user-read
S_IWUSR	user-write
S_IXUSR	user-execute
S_IRGRP	group-read
S_IWGRP	group-write
S_IXGRP	group-execute
S_IROTH	other-read
S_IWOTH	other-write
S_IXOTH	other-execute

See the example in later slides ...

Figure 4.6 The nine file access permission bits, from <sys/stat.h>

Opening a file

- A file is opened or created by calling either the open function or the openat function
- Both functions return a file descriptor, which is guaranteed to be the lowest-numbered unused descriptor

file type & file creation mode (permissions) - Section 4.5 – 4.8

```
#include <fcntl.h>

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

int openat(int fd, const char *path, int oflag, ... /* mode_t mode */ );
```

next slide

Both return: file descriptor if OK, –1 on error

- The *oflag* argument is formed by ORing ("|") together one or more constants defined in the <fcntl.h> header file

O_RDONLY	Open for reading only.
O_WRONLY	Open for writing only.
O_RDWR	Open for reading and writing.
O_EXEC	Open for execute only.
O_SEARCH	Open for search only (applies to directories).

- One and only one of this five constants must be specified
- There are many other optional ones (see Chapter 3.3) that can be combined by using "|"

Opening a file

The *fd* parameter distinguishes the openat function from the open function.

There are three possibilities:

1. The *path* parameter specifies an absolute pathname. In this case, the *fd* parameter is ignored and the openat function behaves like the open function

2. The *path* parameter specifies a relative pathname and the *fd* parameter is a file descriptor that specifies the starting location in the file system where the relative pathname is to be evaluated. The *fd* parameter is obtained by opening the directory (next slide) where the relative pathname is to be evaluated

3. The *path* parameter specifies a relative pathname and the *fd* parameter has the special value AT_FDCWD. In this case, the pathname is evaluated starting in the Current Working Directory and the openat function behaves like the open function.

Opening a file

- Obtaining the fd parameter by opening a directory

```
int dirfd = open("/mydir/", O_RDONLY);
```

- Then, to access any file:

```
int fd  = openat(dirfd, "pippo.txt", O_RDWR);
int fd2 = openat(dirfd, "pluto.txt", O_RDWR);
```

Then, you can `read()`/`write()` both `fd` and `fd2`

- In alternative, you can "promote" `fd` to a `FILE *` like so:

```
FILE *fp = fdopen(fd, "r+");
```

and use `fread()`/`fwrite()`

- You can also

```
struct stat sb;
fstatat(dirfd, "pluto.txt", &sb, 0);
```

or

```
fstat(fd2, &sb); // fd2 acquired from openat() by passing dirfd and "pluto.txt" above
```

Back on that later

Creating/opening a file

- A new file can also be created by calling the creat function

Section 3.4 of the textbook

```
#include <fcntl.h>
int creat(const char *path, mode_t mode);
                    Returns: file descriptor opened for write-only if OK, –1 on error
```

Note that this function is equivalent to

```
open(path, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

Historically, in early versions of the UNIX System, the second argument to open could be only 0, 1, or 2. There was no way to open a file that didn't already exist. Therefore, a separate system call, creat, was needed to create new files. With the O_CREAT and O_TRUNC options now provided by open, a separate creat function is no longer needed.

Create the file if it doesn't exist. This option requires the third argument (...) to the open function (i.e., *mode*)

If the file exists and if it is successfully opened for either write-only or read–write, truncate its length to 0

Generate an error if O_CREAT is also specified, and the file already exists

Sample call

```
open("testfile.txt", O_WRONLY|O_CREAT|O_EXCL, S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)
```

File types (1/2)

Most files on a UNIX system are either regular files or directories, but there are additional types of files

- (-) Regular file - The most common type of file, which contains data of some form. There is no distinction to the kernel whether data is text or binary
  - One notable exception to this is with binary executable files. To execute a program, the kernel must understand its format
  - All binary executable files conform to a format that allows the kernel to identify where to load a program's text and data
- (d) Directory file - A file that contains the names of other files and pointers to information on these files (i.e., directory entries)
- (b) Block special (device) file - A type of file providing buffered I/O access in fixed-size units to devices such as disk drives, e.g., /dev/disk0
  - Note that FreeBSD no longer supports block special files. Devices can be accessed through the character special interface

File types (2/2)

- (c) Character special (device) file - A type of file providing unbuffered I/O access in variable-sized units to devices, e.g., /dev/rdisk0, /dev/fd/0, /dev/fd/1 (see later in these slides)

```
iMac-di-User:workspace-1 marcoautili$
iMac-di-User:workspace-1 marcoautili$ file /dev/fd/0
/dev/fd/0: character special (16/0)
iMac-di-User:workspace-1 marcoautili$ file /dev/fd/1
/dev/fd/1: character special (16/0)
iMac-di-User:workspace-1 marcoautili$ file /dev/fd/2
/dev/fd/2: character special (16/0)
iMac-di-User:workspace-1 marcoautili$
```

- (p) FIFO - A type of file used for communication between processes (it is sometimes called a named pipe)

- (s) Socket - A type of file used for network communication between processes

- (l) Symbolic link - A type of file that points to another file (more about symbolic links in Section 4.17)

File types: there may be more... 😮

- '-' regular file
- 'b' block special file
- 'c' character special file
- 'C' high performance ("contiguous data") file
- 'd' directory
- 'D' door (special file for inter-process communication between a client and server)
  https://en.wikipedia.org/wiki/Doors_(computing)
- 'l' symbolic link
- 'm' multiplexed file (7th edition Unix; obsolete)
  https://en.wikipedia.org/wiki/Version_7_Unix#Multiplexed_files
- 'n' network special file (HP-UX)
- 'p' fifo (named pipe)
- 'P' port
- 's' socket
- 'w' whiteout (4.4BSD; obsolete)

Some anticipation…

- Creating a PIPE (see Section 15.2)

  `pipe()`

  ```
  man 2 pipe
  ```

- Creating a FIFO (see Section 15.5)

  `mkfifo()`, `mkfifoat()`, `mknod()`, `mknodat()`

  ```
  man 2 mkfifo
  man 2 mkfifoat
  ```

- Creating a SOCKET (see Section 16.2)

  `socket()`, `socketpair()`

  ```
  man 2 socket
  man 2 socketpair
  ```

- Creating a special file

  `mknod()`, `mknodat()`

  ```
  man 2 mknodat
  ```

The bit mask for file type

S_IFMT is a bit mask for file type (see also man stat)

- Using a bitwise AND (&) directly with mystat.st_mode means considering only the bits involved to determine the file type (regular file, socket, block or char device, etc.)

```
mystat.st_mode & S_IFMT
```

- Using a bitwise AND with the negated bitmask S_IFMT means ignoring the bits for file type and keeping just the ones needed to determine file permission

```
mystat.st_mode & ~S_IFMT
```

File types and st_mode

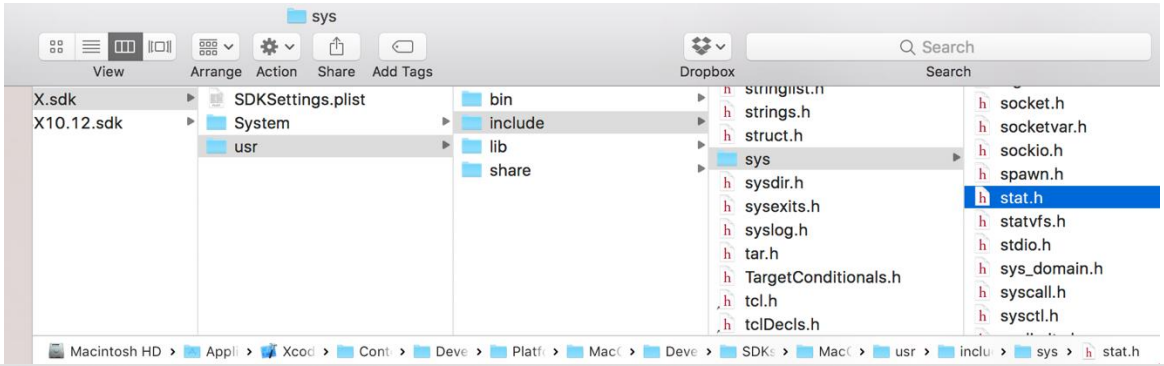
- The type of a file is encoded in the st_mode member of the stat structure
- The file type can also be determined directly with the macros shown in Figure 4.1
- The argument to each of these macros is the st_mode member from the stat structure

Macro	Type of file
S_ISREG()	regular file
S_ISDIR()	directory file
S_ISCHR()	character special file
S_ISBLK()	block special file
S_ISFIFO()	pipe or FIFO
S_ISLNK()	symbolic link
S_ISSOCK()	socket

See next slides ...

Figure 4.1 File type macros in <sys/stat.h>

File types and the stat.h header file



See the example chap4-filedir/filetype.c later in these slides

(Section 4.3 of the textbook)

stat.h

```
243  /*
244   * [XSI] The following macros shall be provided to test whether a file is
245   * of the specified type.  The value m supplied to the macros is the value
246   * of st_mode from a stat structure.  The macro shall evaluate to a non-zero
247   * value if the test is true; 0 if the test is false.
248   */
249  #define S_ISBLK(m)  (((m) & S_IFMT) == S_IFBLK) /* block special */
250  #define S_ISCHR(m)  (((m) & S_IFMT) == S_IFCHR) /* char special */
251  #define S_ISDIR(m)  (((m) & S_IFMT) == S_IFDIR) /* directory */
252  #define S_ISFIFO(m) (((m) & S_IFMT) == S_IFIFO) /* fifo or socket */
253  #define S_ISREG(m)  (((m) & S_IFMT) == S_IFREG) /* regular file */
254  #define S_ISLNK(m)  (((m) & S_IFMT) == S_IFLNK) /* symbolic link */
255  #define S_ISSOCK(m) (((m) & S_IFMT) == S_IFSOCK)    /* socket */
256  #if !defined(_POSIX_C_SOURCE) || defined(_DARWIN_C_SOURCE)
257  #define S_ISWHT(m)  (((m) & S_IFMT) == S_IFWHT) /* OBSOLETE: whiteout */
258  #endif
259
```

stat, fstat, fstatat, and lstat Functions

See next slide

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

int stat(const char *restrict pathname, struct stat *restrict buf);

int fstat(int fd, struct stat *buf);

int lstat(const char *restrict pathname, struct stat *restrict buf);

int fstatat(int fd, const char *restrict pathname,
            struct stat *restrict buf, int flag);
```

All four return: 0 if OK, –1 on error

See Section 4.2 of the textbook

- Given a *pathname*, the stat function obtains in *buf* a structure of information about the named file
- The fstat function obtains information about the file that is already open on the descriptor *fd*
- The lstat function is similar to stat, but when the named file is a symbolic link, lstat returns information about the symbolic link, not the file referenced by the symbolic link
- The fstatat function provides a way to return the file statistics for a pathname relative to an open directory represented by the *fd* argument. The *flag* argument controls whether symbolic links are followed; when the AT_SYMLINK_NOFOLLOW flag is set, fstatat will not follow symbolic links, but rather returns information about the link itself

Curiosity

- restrict is a keyword that can be used in pointer declarations (it is a declaration of intent given by the programmer to the compiler)

- This keyword is used to tell the compiler which pointer references can be optimized, by indicating that the object to which the pointer refers is accessed in the function only via that pointer

- It says that for the lifetime of the pointer, only the pointer itself or a value directly derived from it (such as pointer + 1) will be used to access the object to which it points
  - This limits the effects of pointer aliasing (e.g., when the same memory location can be accessed using different pointers with the same value)
  - This aids optimizations (e.g., minimizing the time taken to execute a program)

- If the declaration of intent is not followed and the object is accessed by an independent pointer, this will result in undefined behavior

https://en.wikipedia.org/wiki/Restrict

File types

We have specifically used the lstat function instead of the stat function to "detect" symbolic links

As already said, lstat returns information about the symbolic link, not the file referenced by the symbolic link

filetype.c

```c
#include "apue.h"

int
main(int argc, char *argv[])
{
    int         i;
    struct stat buf;
    char        *ptr;

    for (i = 1; i < argc; i++) {
        printf("%s: ", argv[i]);
        if (lstat(argv[i], &buf) < 0) {
            err_ret("lstat error");
            continue;
        }
        if (S_ISREG(buf.st_mode))
            ptr = "regular";
        else if (S_ISDIR(buf.st_mode))
            ptr = "directory";
        else if (S_ISCHR(buf.st_mode))
            ptr = "character special";
        else if (S_ISBLK(buf.st_mode))
            ptr = "block special";
        else if (S_ISFIFO(buf.st_mode))
            ptr = "fifo";
        else if (S_ISLNK(buf.st_mode))
            ptr = "symbolic link";
        else if (S_ISSOCK(buf.st_mode))
            ptr = "socket";
        else
            ptr = "** unknown mode **";
        printf("%s\n", ptr);
    }
    exit(0);
}
```

Reaching the definition of mode_t

stat.h

macOS 10.13 〉 usr/include 〉 sys 〉 stat.h 〉 No Selection

Find _darwin_mode 0 matches Aa

```
90 #if !defined(_POSIX_C_SOURCE) || defined(_DARWIN_C_SOURCE)
91 #include <sys/_types/_ino64_t.h>
92 #endif /* !defined(_POSIX_C_SOURCE) || defined(_DARWIN_C_SOURCE) */
93
94 #include <sys/_types/_mode_t.h>
95 #include <sys/_types/_nlink_t.h>
96 #include <sys/_types/_uid_t.h>
97 #include <sys/_types/_gid_t.h>
98 #include <sys/_types/_off_t.h>
99 #include <sys/_types/_time_t.h>
```

_mode_t.h

macOS 10.13 〉 usr/include 〉 sys 〉 _types 〉 _mode_t.h 〉 mode_t

```
22  * FITNESS FOR A PARTICULAR PURPOSE, QUIET ENJOYMENT OR
23  * Please see the License for the specific language gov
24  * limitations under the License.
25  *
26  * @APPLE_OSREFERENCE_LICENSE_HEADER_END@
27  */
28 #ifndef _MODE_T
29 #define _MODE_T
30 #include <sys/_types.h> /* __darwin_mode_t */
31 typedef __darwin_mode_t     mode_t;
32 #endif  /* _MODE_T */
```

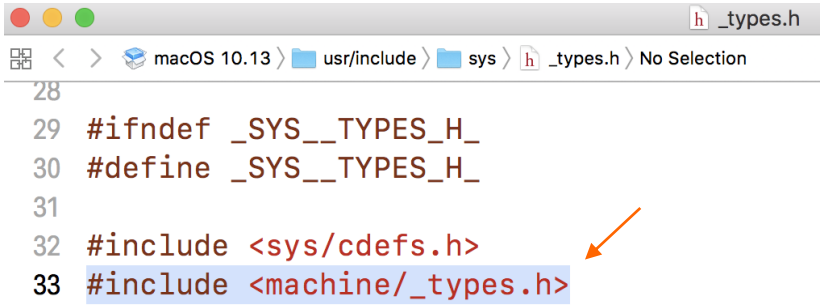
_types.h

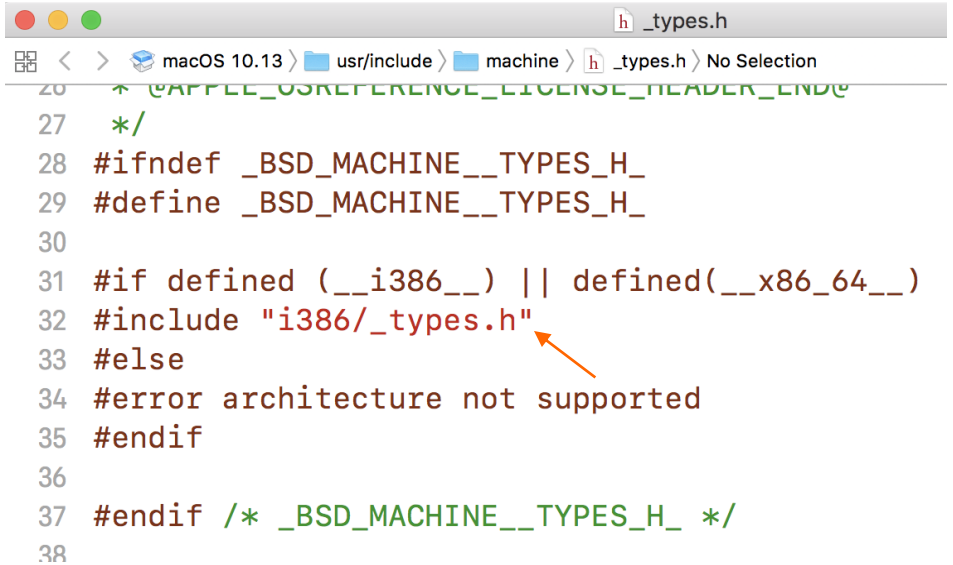
macOS 10.13 〉 usr/include 〉 sys 〉 _types.h 〉 __darwin_mode_t

Find _darwin_mode 1 match Aa Contains

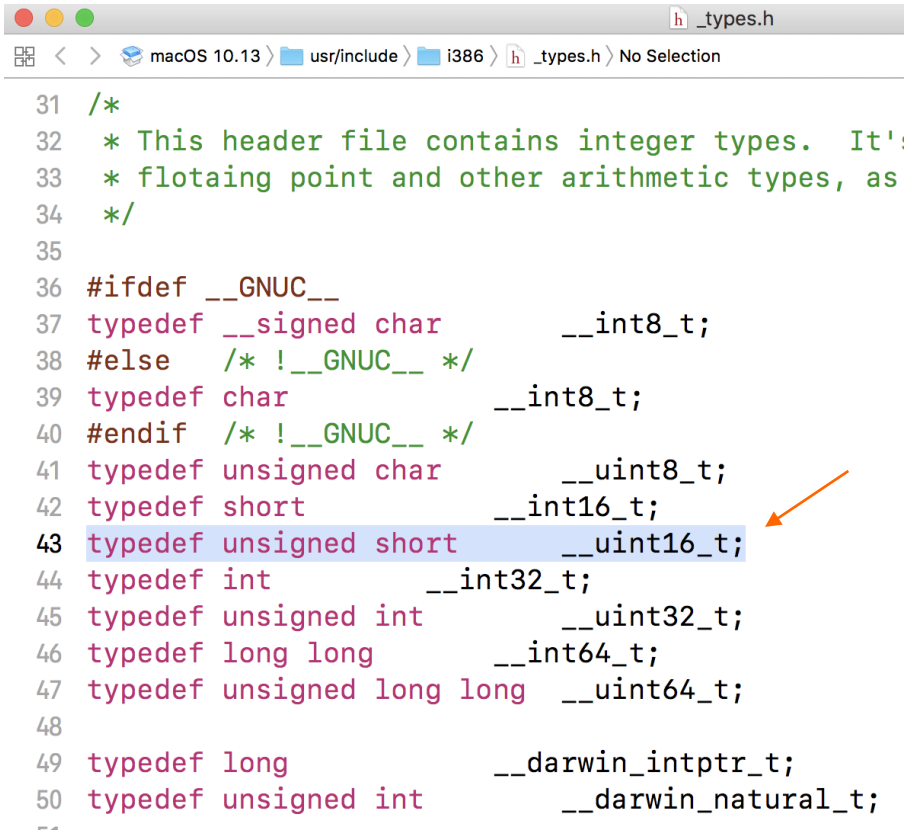
```
66 typedef __uint32_t  __darwin_ino_t;      /* [???] Used for inodes */
67 #endif /* __DARWIN_64_BIT_INO_T */
68 typedef __darwin_natural_t __darwin_mach_port_name_t; /* Used by mach */
69 typedef __darwin_mach_port_name_t __darwin_mach_port_t; /* Used by mach */
70 typedef __uint16_t  __darwin_mode_t;     /* [???] Some file attributes */
71 typedef __int64_t   __darwin_off_t;      /* [???] Used for file sizes */
72 typedef __int32_t   __darwin_pid_t;      /* [???] process and group IDs */
73 typedef __uint32_t  __darwin_sigset_t;   /* [???] signal set */
74 typedef __int32_t   __darwin_suseconds_t;    /* [???] microseconds */
75 typedef __uint32_t  __darwin_uid_t;      /* [???] user IDs */
76 typedef __uint32_t  __darwin_useconds_t;     /* [???] microseconds */
```

Reaching the definition of __uint16_t





… and finally, here it is!



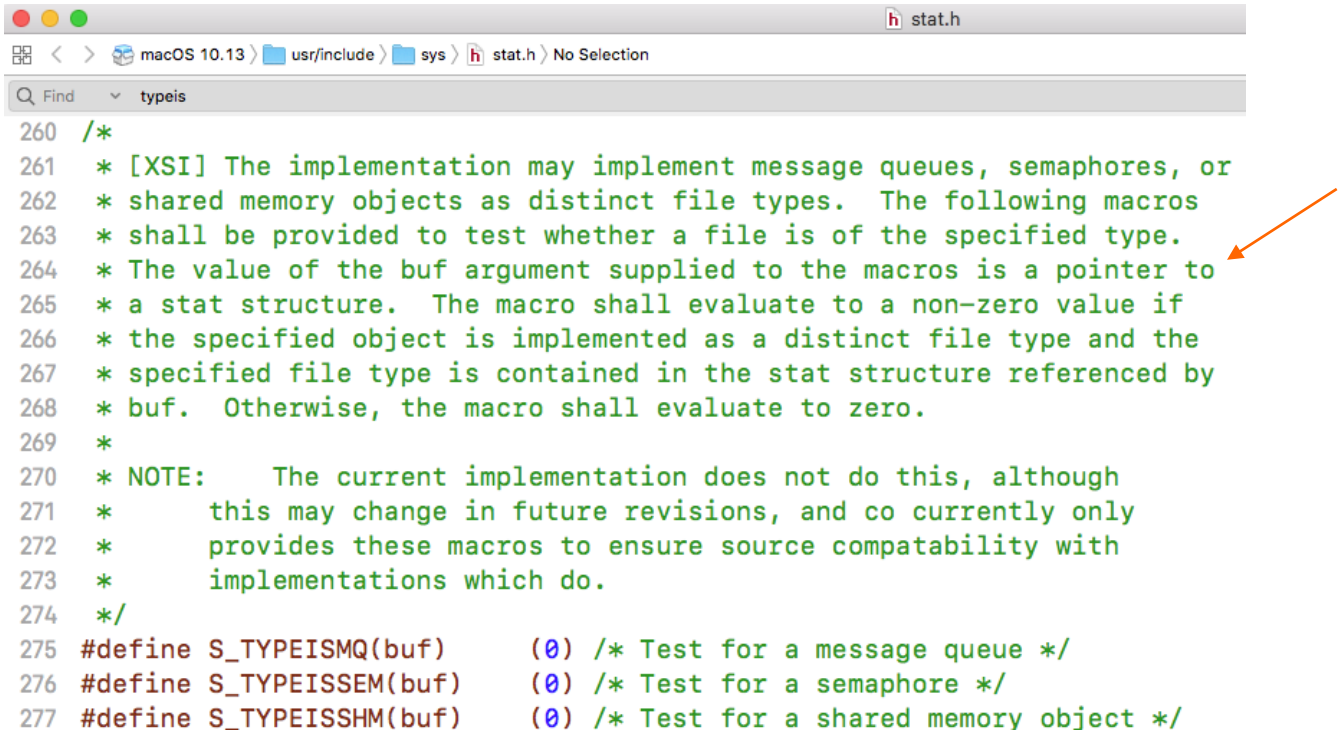
Even more about file types

- POSIX.1 allows implementations to represent interprocess communication (IPC) objects, such as message queues and semaphores, as files
- The macros shown in Figure 4.2 allow us to determine the type of IPC object from the stat structure
- Instead of taking the st_mode member as an argument, these macros differ from those in Figure 4.1 in that their argument is a pointer to the stat structure
- Message queues, semaphores, and shared memory objects are discussed in Chap 15 of the textbook. However, none of the various implementations of the UNIX System discussed in the book represent these objects as files

Macro	Type of object
`S_TYPEISMQ()`	message queue
`S_TYPEISSEM()`	semaphore
`S_TYPEISSHM()`	shared memory object

Figure 4.2 IPC type macros in `<sys/stat.h>`

Even more about file types



See also a stat.h file from Apple opensource projects in next slide...

Summing up

stat.h

```
/*
 * [XSI] The following macros shall be provided to test whether a file is
 * of the specified type.  The value m supplied to the macros is the value
 * of st_mode from a stat structure.  The macro shall evaluate to a non-zero
 * value if the test is true; 0 if the test is false.
 */
#define S_ISBLK(m)      (((m) & S_IFMT) == S_IFBLK)     /* block special */
#define S_ISCHR(m)      (((m) & S_IFMT) == S_IFCHR)     /* char special */
#define S_ISDIR(m)      (((m) & S_IFMT) == S_IFDIR)     /* directory */
#define S_ISFIFO(m)     (((m) & S_IFMT) == S_IFIFO)     /* fifo or socket */
#define S_ISREG(m)      (((m) & S_IFMT) == S_IFREG)     /* regular file */
#define S_ISLNK(m)      (((m) & S_IFMT) == S_IFLNK)     /* symbolic link */
#define S_ISSOCK(m)     (((m) & S_IFMT) == S_IFSOCK)    /* socket */
#if !defined(_POSIX_C_SOURCE) || defined(_DARWIN_C_SOURCE)
#define S_ISWHT(m)      (((m) & S_IFMT) == S_IFWHT)     /* OBSOLETE: whiteout */
#endif

/*
 * [XSI] The implementation may implement message queues, semaphores, or
 * shared memory objects as distinct file types.  The following macros
 * shall be provided to test whether a file is of the specified type.
 * The value of the buf argument supplied to the macros is a pointer to
 * a stat structure.  The macro shall evaluate to a non-zero value if
 * the specified object is implemented as a distinct file type and the
 * specified file type is contained in the stat structure referenced by
 * buf.  Otherwise, the macro shall evaluate to zero.
 *
 * NOTE:        The current implementation does not do this, although
 *              this may change in future revisions, and co currently only
 *              provides these macros to ensure source compatability with
 *              implementations which do.
 */
#define S_TYPEISMQ(buf)         (0)     /* Test for a message queue */
#define S_TYPEISSEM(buf)        (0)     /* Test for a semaphore */
#define S_TYPEISSHM(buf)        (0)     /* Test for a shared memory object */
```

https://opensource.apple.com/source/xnu/xnu-344/bsd/sys/stat.h.auto.html

lseek function

- Every open file has an associated "current file offset," normally a non-negative integer that measures the number of bytes from the beginning of the file
- Read and write operations normally start at the current file offset and cause the offset to be incremented by the number of bytes read or written
- By default, this offset is initialized to 0 when a file is opened, unless the O_APPEND option is specified

```
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

From where?

Returns: new file offset if OK, –1 on error

The interpretation of the *offset* depends on the value of the *whence* argument.

- If *whence* is `SEEK_SET`, the file's offset is set to *offset* bytes from the beginning of the file.
- If *whence* is `SEEK_CUR`, the file's offset is set to its current value plus the *offset*. The *offset* can be positive or negative.
- If *whence* is `SEEK_END`, the file's offset is set to the size of the file plus the *offset*. The *offset* can be positive or negative.

N.B.: bytes!

lseek function

- lseek is a system call that is used to change the location of the read/write pointer of a file descriptor

- Because a successful call to lseek returns the new file offset, we can seek zero bytes from the current position to determine the current offset

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

- lseek only records the current file offset within the kernel

- lseek does not cause any I/O to take place; the offset is then used by the next read or write operation

Open, read, lseek @work together

openreadseek.c

openreadseek.c 〉 No Selection

```c
#include "apue.h"

//#include <unistd.h>
#include <fcntl.h>
//#include <sys/types.h>

int main()
{
    int file=0;

    if((file = open("testfile.txt",O_RDONLY)) < 0) {
        printf("\n fd = %d \n", file);
        return 1;
    }

    char buffer[22];

    if(read(file,buffer,22) != 22)  return 1;
    printf("%s\n",buffer);
    printf("%c\n",buffer[strlen(buffer) - 2]);

    if(lseek(file,10,SEEK_SET) < 0) return 1;

    if(read(file,buffer,22) != 22)  return 1;
    printf("%s\n",buffer);
    printf("%c\n",buffer[strlen(buffer) - 2]);

    return 0;
}
```

testfile.txt

testfile.txt

```
1  This is a test file that will be used
2  to demonstrate the use of lseek, open, and read.
3
```

chap3-fileio — -bash — 59×6

```
desktop-jisqlks:chap3-fileio marcoautili$ ./openreadseek
This is a test file th
t
test file that will be
b
desktop-jisqlks:chap3-fileio marcoautili$
```

Quite easy dealing with characters in a text file...

Alignment of data structures in memory

sizeofstruct.c

sizeofstruct.c 〉 main

```c
#include "apue.h"

int main()
{
    int tot_size;

    struct my_struct {
        char c;
        int i;
        char *bufp;
    } ms;

    printf("\n Actual size of the member ms.c = %d byte", (int)sizeof(ms.c));
    printf("\n Actual size of the memberms.i = %d bytes", (int)sizeof(ms.i));
    printf("\n Actual size of the member ms.bufp = %d bytes \n", (int)sizeof(ms.bufp));

    tot_size = (int)sizeof(ms.c) + (int)sizeof(ms.i) + (int)sizeof(ms.bufp);

    /* N.B.: The compiler may add "padding" for alignment requirements. Note that this applies
     not only to padding between the fields of a struct, but also may apply to the end of the
     struct (so that arrays of the structure type will have each element properly aligned).
     */
    printf("\n Size of the struct ms = %d bytes \n", (int)sizeof(ms));

    // In our case the padding is added into the struct for the char c, just after it
    printf("\n Size of padding added by the compiler = %d bytes \n", (int)sizeof(ms) - tot_size);

    printf("\n");
    return 0;
}
```

```
ste:chap3-fileio marcoautili$
ste:chap3-fileio marcoautili$ ./sizeofstruct

 Actual size of the member ms.c = 1 byte
 Actual size of the memberms.i = 4 bytes
 Actual size of the member ms.bufp = 8 bytes

 Size of the struct ms = 16 bytes

 Size of padding added by the compiler = 3 bytes

ste:chap3-fileio marcoautili$
```

sizeof() is a compile-time unary operator that calculates the size of its operand at compile time

Structures and Alignment

```
struct my_struct {
        char c;
        int i;
        char *bufp;
} *p
```

Note that 8 is the largest field in the structure

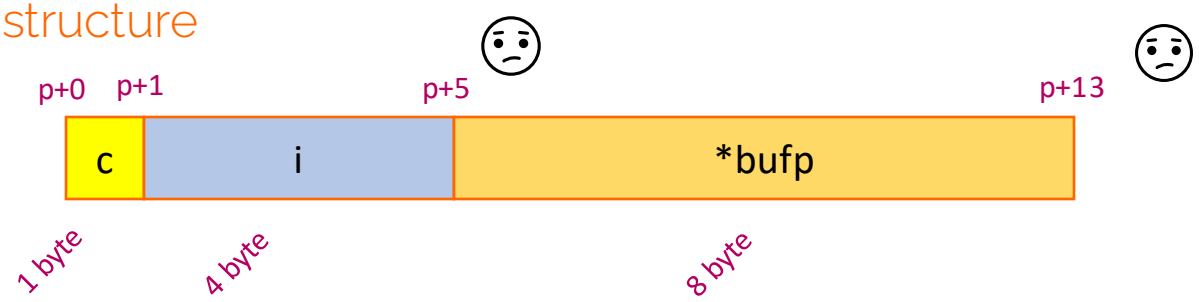
- i.e., a pointer in a 64-bit architecture like mine is 8

  → the multiple for alignment is thus 8

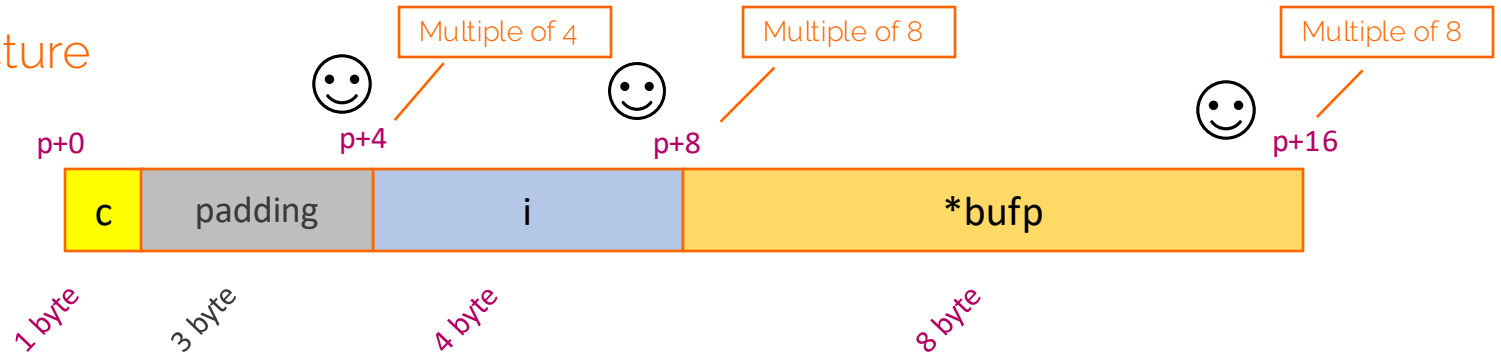
(a pointer in a 32-bit architecture is 4 bytes)

(a pointer in a 64-bit architecture running a 32-bit OS is 4 bytes)

Non-aligned structure



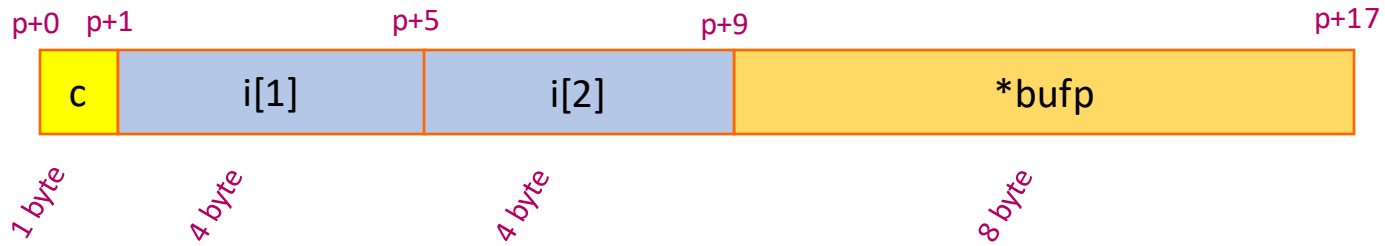
Aligned structure



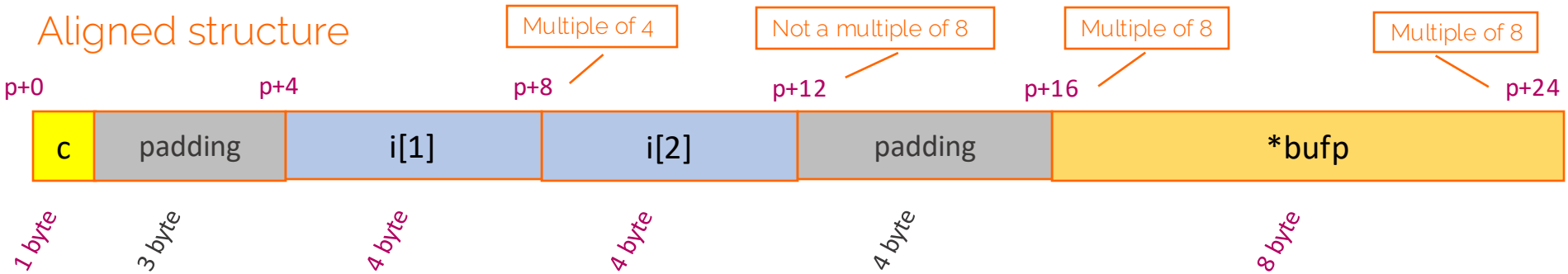
Structures and Alignment

```
struct my_struct {
        char c;
        int i[2];
        char *bufp;
} *p
```

Non-aligned structure



Aligned structure



Alignment principles

- Alignment is required on some systems, and it is advised in others
- It is treated differently in different systems depending on their architecture
- Primitive data types require K bytes, and addresses must be multiple of K
  - K-bytes data must start at addresses divisible by k
  - physical memory is accessed by aligned chunks (e.g., of 4 or 8 bytes -- system dependent)
- The compiler adds padding within structure to ensure correct alignment of the structure fields
  - `sizeof()` should be used to get the actual size of a `struct`
- Alignment leads to fragmentation within the structure (internal fragmentation)

The order of fields makes the difference

```
struct my_struct {
        char *bufp;
        char c;
        int i[2];
} *p
```

Now, you try…

… and what if we have an array of such a struct?

```
struct my_struct {
        char *bufp;
        char c;
        int i[2];
} p[10]
```

Files and Directories (intro)

The "ls" command (a simple implementation of)

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

int
main(int argc, char *argv[])
{
    DIR             *dp;
    struct dirent   *dirp;

    if (argc != 2)
        err_quit("usage: ls directory_name");

    if ((dp = opendir(argv[1])) == NULL)
        err_sys("can't open %s", argv[1]);
    while ((dirp = readdir(dp)) != NULL)
        printf("%s\n", dirp->d_name);

    closedir(dp);
    exit(0);
}
```

Contains the function prototypes for opendir(), readdir(), and the definition of the dirent structure

Files and Directories (intro)

main declaration conforming to the ISO C standard

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

int
main(int argc, char *argv[])
{
    DIR             *dp;
    struct dirent   *dirp;

    if (argc != 2)
        err_quit("usage: ls directory_name");

    if ((dp = opendir(argv[1])) == NULL)
        err_sys("can't open %s", argv[1]);
    while ((dirp = readdir(dp)) != NULL)
        printf("%s\n", dirp->d_name);

    closedir(dp);
    exit(0);
}
```

Pointer to a DIR structure

Pointer to a directory entry within the DIR structure

Files and Directories (intro)

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

int
main(int argc, char *argv[])
{
    DIR             *dp;
    struct dirent   *dirp;

    if (argc != 2)
        err_quit("usage: ls directory_name");

    if ((dp = opendir(argv[1])) == NULL)
        err_sys("can't open %s", argv[1]);
    while ((dirp = readdir(dp)) != NULL)
        printf("%s\n", dirp->d_name);

    closedir(dp);
    exit(0);
}
```

The `dirent` struct is implementation dependent

However, it contains at least the following two members:

```
ino_t d_ino; /* i-node number */
char d_name[]; /* null-terminated filename */
```

Files and Directories (intro)

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

int
main(int argc, char *argv[])
{
    DIR             *dp;
    struct dirent   *dirp;

    if (argc != 2)
        err_quit("usage: ls directory_name");

    if ((dp = opendir(argv[1])) == NULL)
        err_sys("can't open %s", argv[1]);
    while ((dirp = readdir(dp)) != NULL)
        printf("%s\n", dirp->d_name);

    closedir(dp);
    exit(0);
}
```

```
DIR *opendir (const char *pathname);
DIR *fdopendir (int fd)
```

- both return a pointer if OK, NULL on error

REMEMBER:
Obtaining the fd parameter by opening a directory, e.g.,

```
int dirfd = open("/mydir/", O_RDONLY);
```

The pathname of the directory to list

Files and Directories (intro)

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

int
main(int argc, char *argv[])
{
    DIR             *dp;
    struct dirent   *dirp;

    if (argc != 2)
        err_quit("usage: ls directory_name");

    if ((dp = opendir(argv[1])) == NULL)
        err_sys("can't open %s", argv[1]);
    while ((dirp = readdir(dp)) != NULL)
        printf("%s\n", dirp->d_name);

    closedir(dp);
    exit(0);
}
```

`struct dirent *readdir(DIR *dp);`

- When an error is encountered, a **null pointer** is returned and **errno** is set to indicate the error
- When the end of the directory is encountered, a **null pointer** is returned and **errno** is not changed

Name of the (current) directory entry

File Sharing

- The UNIX System supports the sharing of open files among different processes

- The kernel uses three data structures to represent an open file:
  Process table, File table and v-node table (→ next slides)

- The relationships among these structures determine the effect one process has on another with regard to file sharing

- These structures can be implemented in numerous/different ways

The following description is conceptual; it may or may not match a particular implementation. Refer to Bach [1986] for a discussion of these structures in System V. McKusick et al. [1996] describe these structures in 4.4BSD. McKusick and Neville-Neil [2005] cover FreeBSD 5.2. For a similar discussion of Solaris, see McDougall and Mauro [2007]. The Linux 2.6 kernel architecture is discussed in Bovet and Cesati [2006].

Kernel data structure for files

1. Process table - every process has an entry in the process table. Within each process table entry is a table of open file descriptors, which we can think of as a vector, with one entry per descriptor

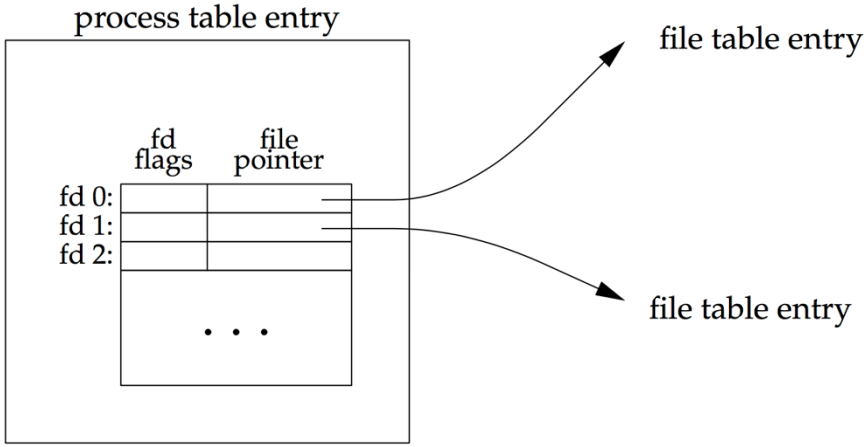
   Associated with each file descriptor are

   (a) The file descriptor flags

   Currently, only one such flag is defined: FD_CLOEXEC, the close-on-exec flag

   - If the FD_CLOEXEC bit is set,
     - the file descriptor will automatically be closed during a successful exec
     - If the exec fails, the file descriptor is left open
   - If the FD_CLOEXEC bit is not set, the file descriptor will remain open across an exec
   - the getter and setter are F_GETFD and F_SETFD (to be used with the fcntl() function)

   (b) A pointer to a file table entry



Kernel data structure for files

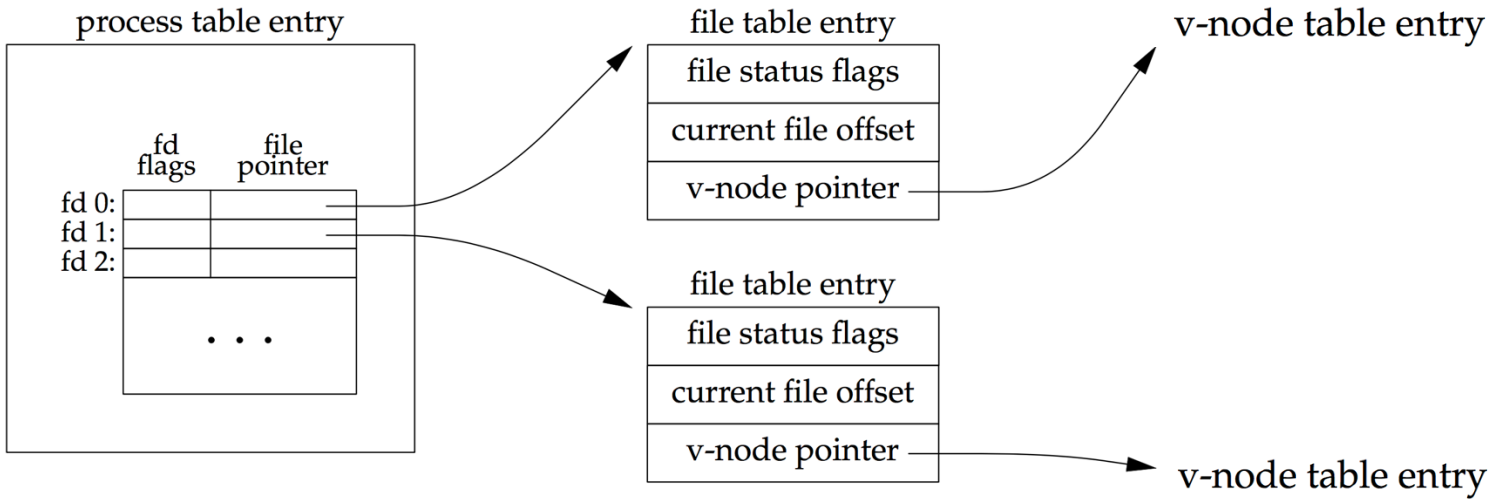
2. File table - the kernel maintains a file table for all open files

   Each file table entry contains

   (a) The file status flags for the file, such as read, write, append, sync and non-blocking (see next SLIDE)

   (b) The current file offset

   (c) A pointer to the v-node table entry for the file



More on file status flags

- File status flags for the function fcntl (see Section 3.14 for more)
  - the fcntl() function provides for control over open files
  - see man fcntl

File status flag	Description
O_RDONLY	open for reading only
O_WRONLY	open for writing only
O_RDWR	open for reading and writing
O_EXEC	open for execute only
O_SEARCH	open directory for searching only
O_APPEND	append on each write
O_NONBLOCK	nonblocking mode
O_SYNC	wait for writes to complete (data and attributes)
O_DSYNC	wait for writes to complete (data only)
O_RSYNC	synchronize reads and writes
O_FSYNC	wait for writes to complete (FreeBSD and Mac OS X only)
O_ASYNC	asynchronous I/O (FreeBSD and Mac OS X only)

Figure 3.10 File status flags for `fcntl`

Kernel data structure for files

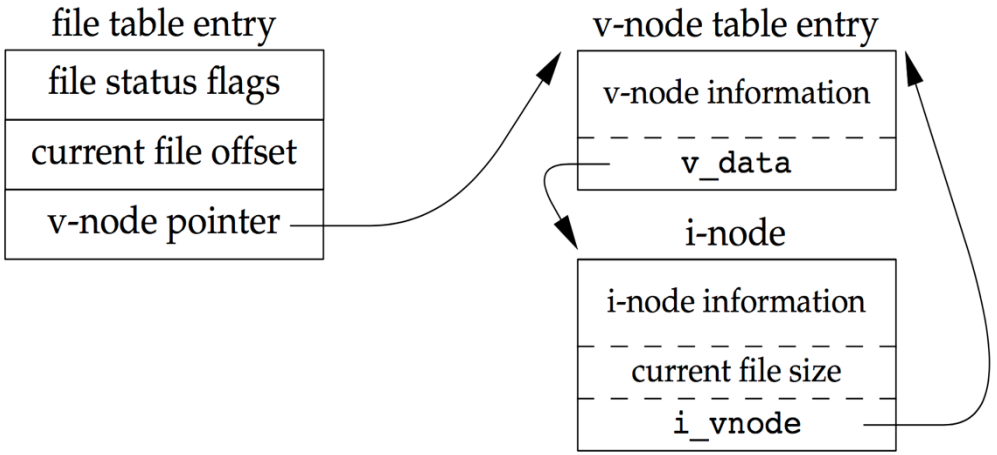
3. v-node table - each open file (or device) has a v-node structure that contains information about the type of file and pointers to functions that operate on the file

For most files, the v-node also contains the i-node (*index node*). This information is read from the disk when the file is opened so that all the pertinent information about the file is readily available

Most of the information in "`struct stat`" is obtained from the i-node

For example, the i-node contains the owner of the file, the size of the file, pointers to where the actual data blocks for the file are located on disk, and so on

- more on i-nodes in next slides and in Section 4.14 where the typical UNIX file system is described in more detail



Linux has no v-node. Instead, a generic i-node structure is used. Although the implementations differ, the v-node is conceptually the same as a generic i-node. Both point to an i-node structure specific to the file system.

Overall picture

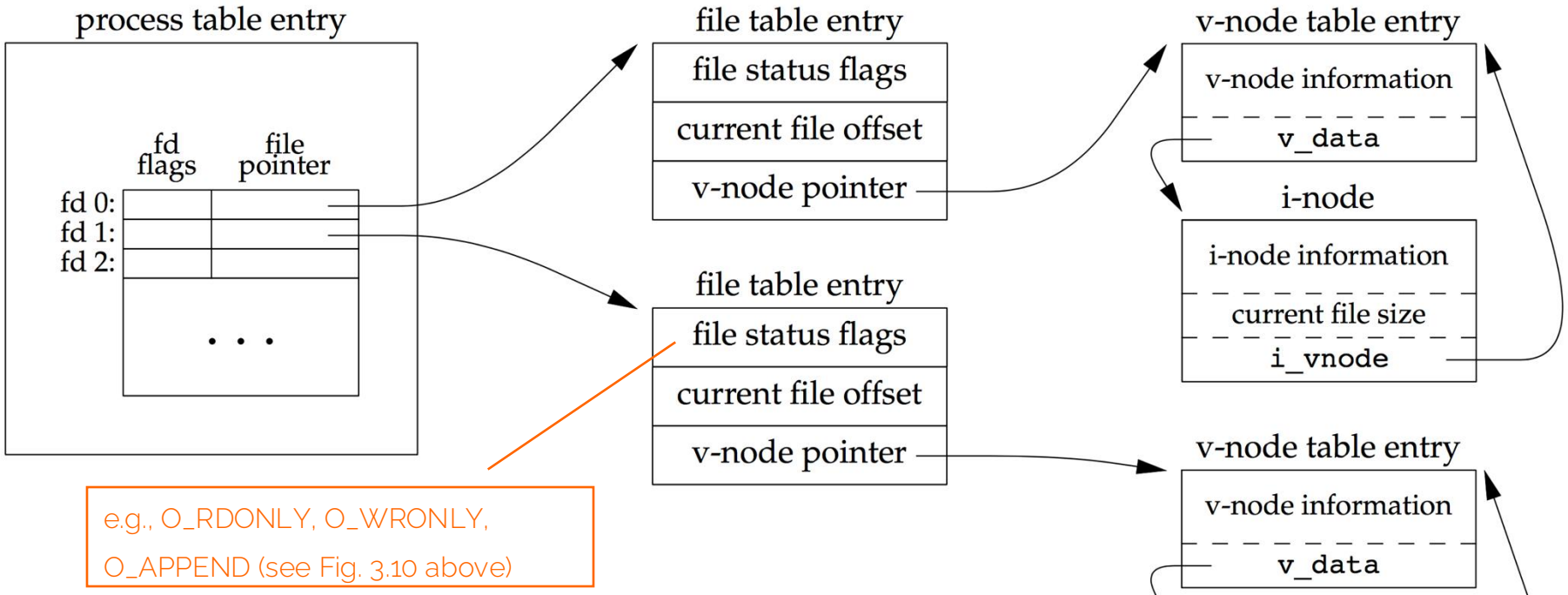
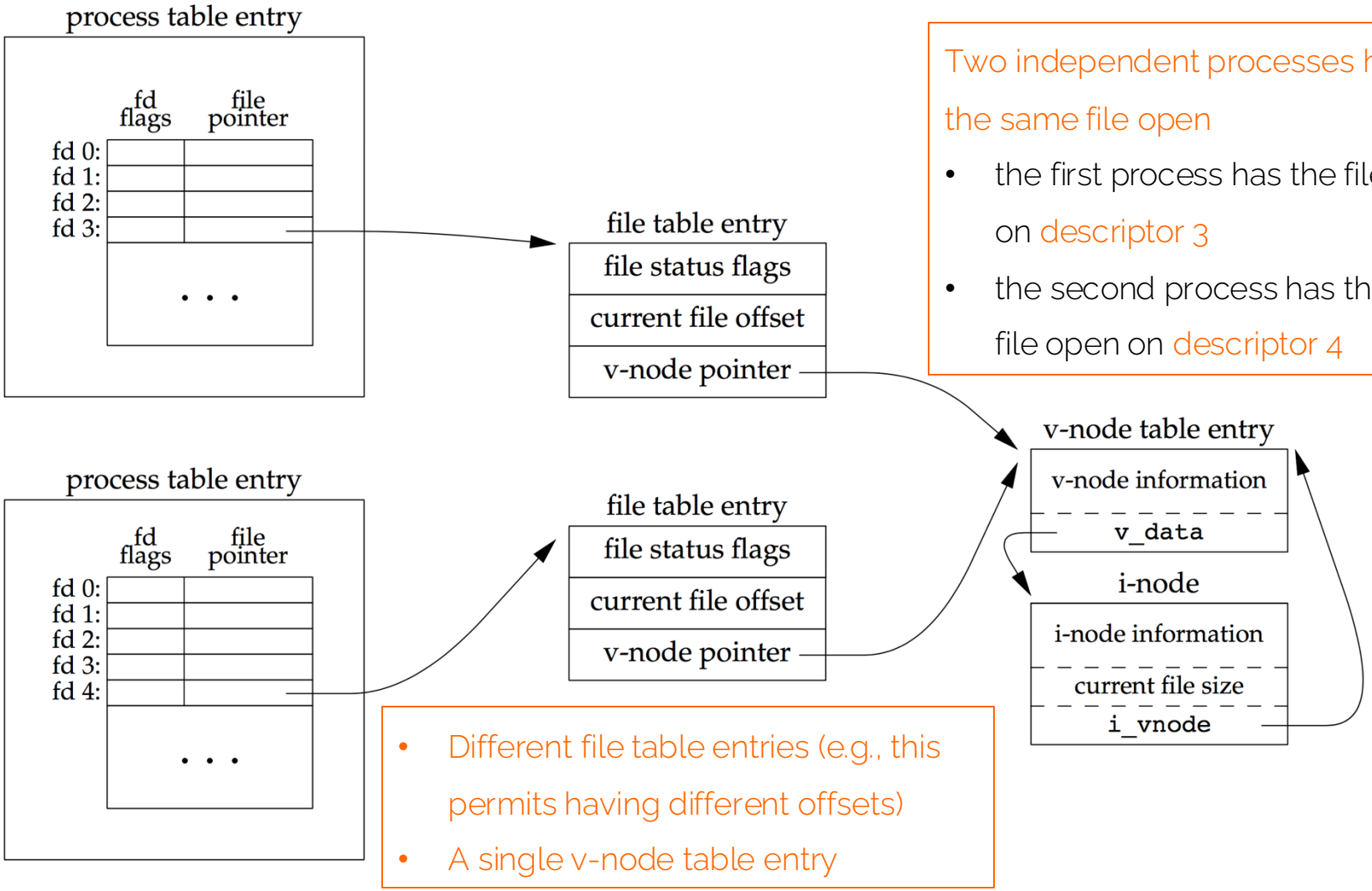


Figure 3.7 Kernel data structures for open files

Kernel data structure for files



Two independent processes having the same file open

- the first process has the file open on descriptor 3
- the second process has the same file open on descriptor 4

- Different file table entries (e.g., this permits having different offsets)
- A single v-node table entry

Figure 3.8 Two independent processes with the same file open

Disk drive, partitions and file system

Conceptual view of the structure of the UNIX file system

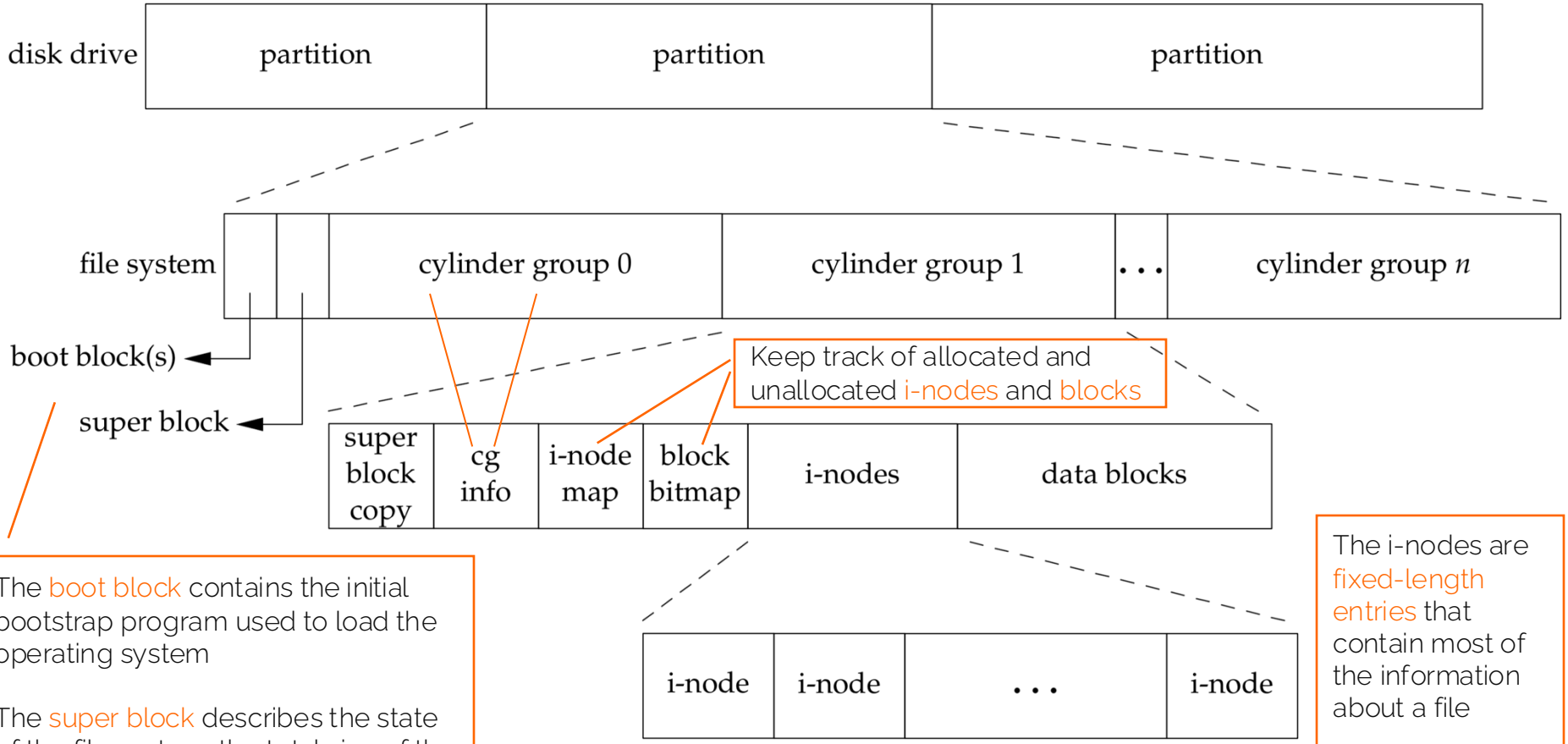


Figure 4.13 Disk drive, partitions, and a file system

The boot block contains the initial bootstrap program used to load the operating system

The super block describes the state of the file system: the total size of the partition, the block size, pointers to a list of free blocks, the i-node number of the root directory

The i-nodes are fixed-length entries that contain most of the information about a file

There is a one-to-one mapping of files to i-nodes and vice versa

i-nodes and data blocks

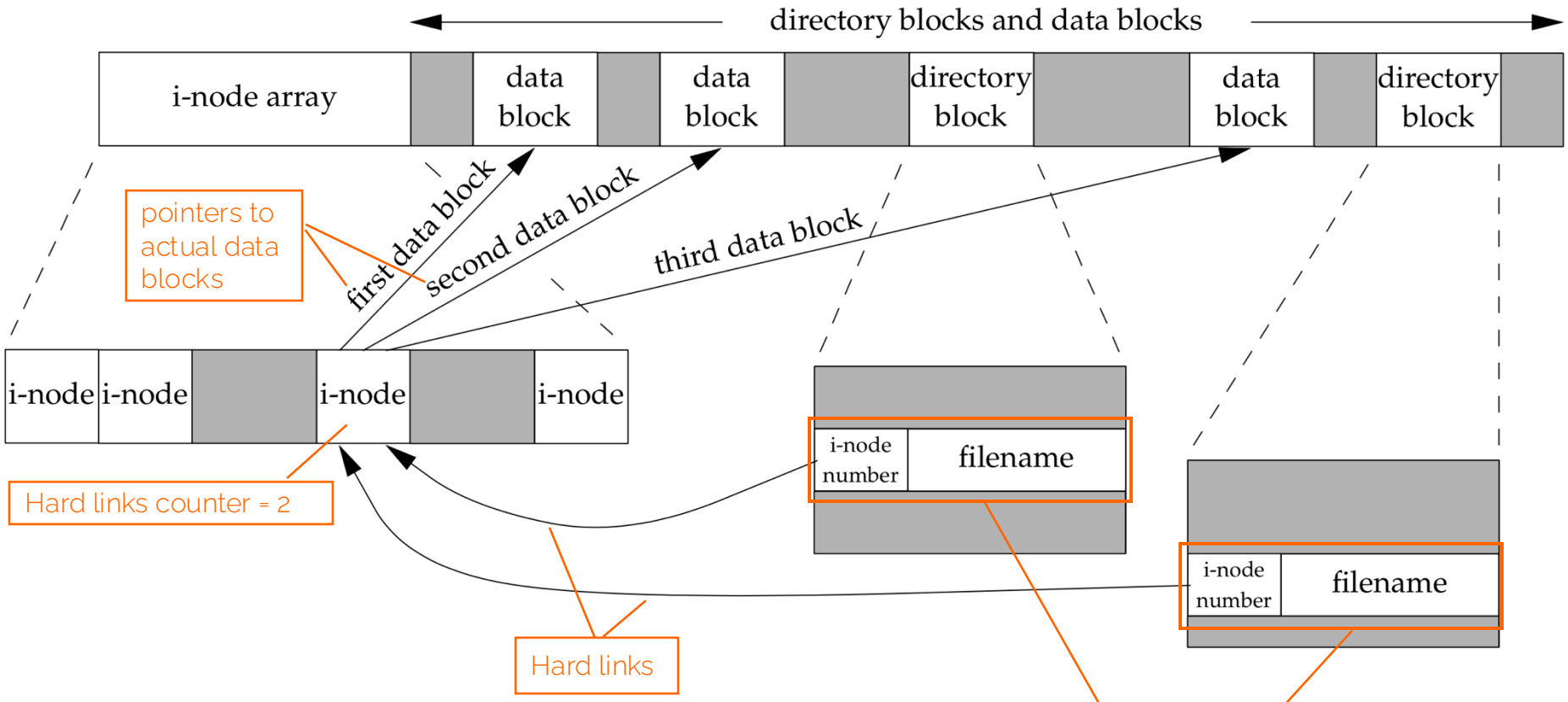


Figure 4.14 Cylinder group's i-nodes and data blocks in more detail

The POSIX.1 constant LINK_MAX specifies the maximum value for a file's link count (Section 2.5.2). Recall that, in the stat structure, the hard link count is the st_nlink member

- Two directory entries that point to the same i-node entry
- Every i-node has a link count that contains the number of directory entries that point to it

i-nodes and directory blocks

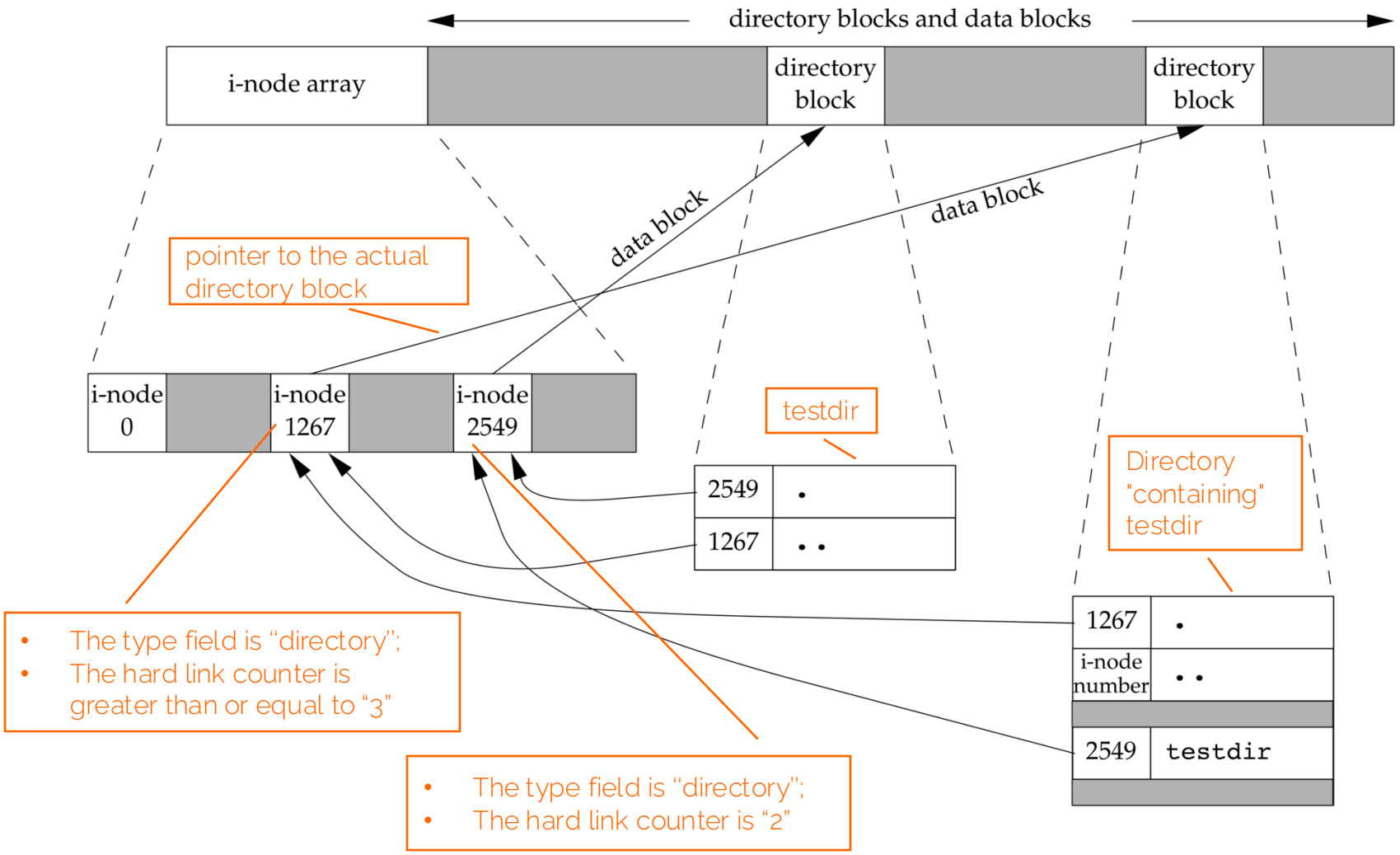


Figure 4.15 Sample cylinder group after creating the directory `testdir`

Summing up

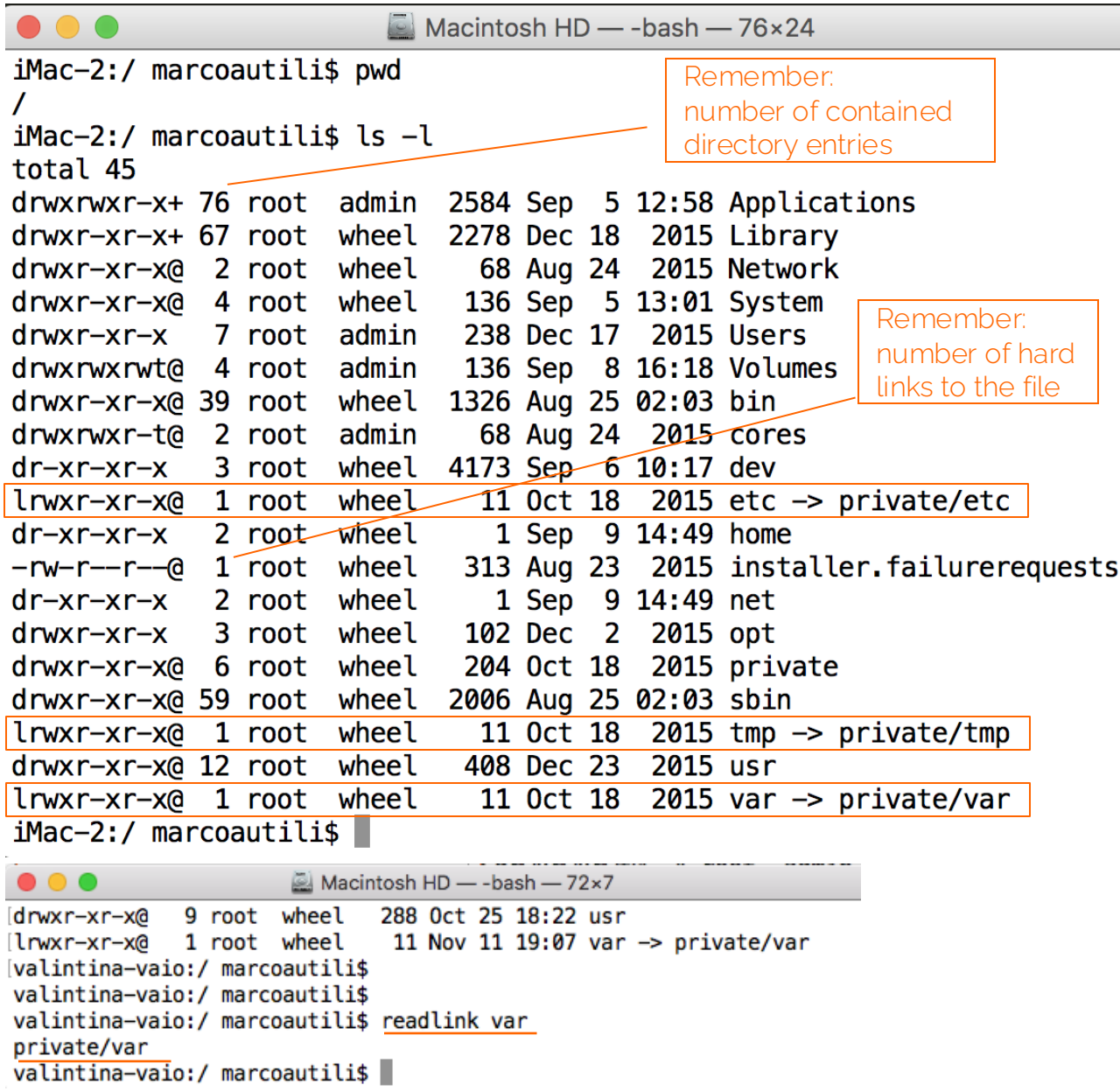
- In Unix-like systems all normal files are hard links
- The inode has a counter for hard links (i.e., the number of directory entries that point to it)
- Each time a new hard link is created, the counter is incremented by one
- When you delete (rm) a link the counter is decremented by one
- Only when the link count goes to 0 the file can be deleted (thereby releasing the data blocks associated with the file)
    - Thus, the operation of unlinking a file does not always mean deleting the blocks associated with the file

Opening a parenthesis on soft links and hard links

Links (from Part 2)

With a symbolic link, the actual contents of the file (i.e., the data blocks) store the name of the file that the symbolic link points to

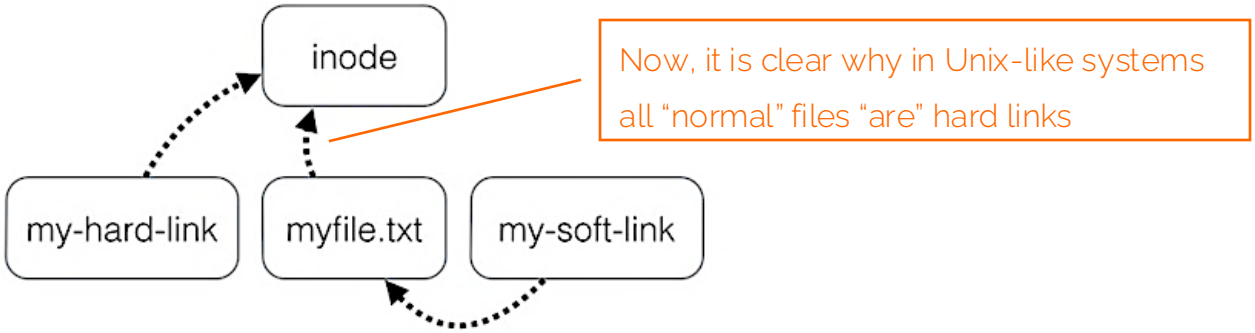
For example, the filename in the directory entry is the three-character string var and the 11 bytes of data in the file are for private/var (one byte for char)



Hard link vs Symbolic link

- A symbolic link is an indirect pointer to (the "name of") a file
- A hard link points directly to the i-node of the file

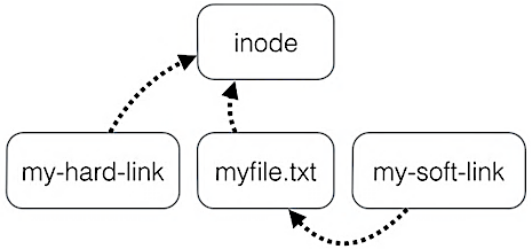
- Symbolic links were introduced to get around the following limitations of hard links
  - Hard links normally require that the link and the file reside in the same file system
  - Only the superuser can create a hard link to a directory (when supported by the underlying file system - see next slide)
  - There are no file system limitations on a symbolic link and what it points to, and anyone can create a symbolic link to a directory



Hard link vs Symbolic link

- Create two files

```
$ touch pippo; touch pluto
```



- Enter some data into them

```
$ echo ”Data contained into pippo" > pippo
$ echo ”Data contained into pluto" > pluto
```

- As expected,

```
$ cat pippo; cat pluto
Data contained into pippo
Data contained into pluto
```

- Create hard and soft links

```
$ ln pippo pippo-hard
$ ln -s pluto pluto-soft
```

Hard link vs Symbolic link

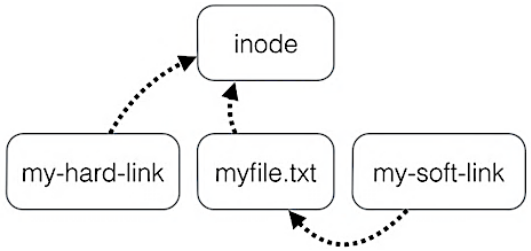
- Let's see what just happened

```
$ ls -l
-rw-r--r--@ 2 marcoautili  staff  26 Nov 29 13:07 pippo
-rw-r--r--@ 2 marcoautili  staff  26 Nov 29 13:07 pippo-hard
-rw-r--r--@ 1 marcoautili  staff  26 Nov 29 13:08 pluto
lrwxr-xr-x  1 marcoautili  staff   5 Nov 29 13:08 pluto-soft -> pluto
```

Note that the number of hard links has been increased for both the files

- Changing the name of pippo does not matter

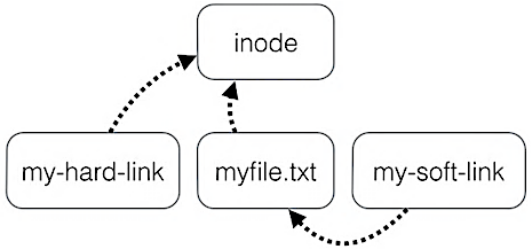
```
$ mv pippo pippo-new
$ cat pippo-hard
Data contained into pippo
```



- In fact, pippo-hard "points to" the inode, and hence to the contents of the file (the content was not changed!)

Hard link vs Symbolic link

```
$ mv pluto pluto-new

$ readlink pluto-soft

pluto

$ cat pluto-soft

cat pluto-soft: No such file or directory
```



- The contents of the file could not be found because the soft link "points" to the name, which was changed, and not to the contents
- Similarly,
  - if pippo is deleted, pippo-hard still holds the contents
  - if pluto is deleted, pluto-soft is just a link to a non-existing file
  - what if a file is placed into the trash? (and not permanently removed)

- Study the functions link, linkat, unlink, unlinkat, and remove in Section 4.15
- Read more on symbolic links in Sections 4.17 and 4.18

→ see next two slides

Directory hardlinks (1/3)

Directory hardlinks can break the filesystem in multiple ways

(1) Directory hardlinks can create loops

- A hard link to a directory can link to a parent of itself, which creates a file system loop

  For example, these commands could create a loop with the back link l

```
mkdir -p /tmp/a/b

cd /tmp/a/b

ln -d /tmp/a l
```

Apple has crippled the ln command (see next slide)

- -d option not available in macOS, see man ln and read -F

```
-F    If the proposed link (link_name) already exists and is a directory,
      then remove it so that the link may occur.  The -F option should be
      used with either -f or -i options.  If none is specified, -f is
      implied.  The -F option is a no-op unless -s option is specified.
```

https://man7.org/linux/man-pages/man1/ln.1.html

```
-d, -F, --directory
       allow the superuser to attempt to hard link directories
       (note: will probably fail due to system restrictions, even
       for the superuser)
```

- A filesystem with a directory loop has infinite depth

```
cd /tmp/a/b/l/b/l/b/l/b/l/b
```

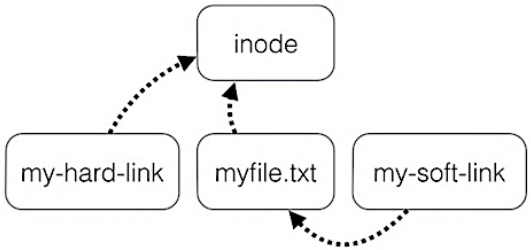
- Avoiding an infinite loop when traversing such a directory structure is somewhat difficult (though for example POSIX requires find to avoid this)

- A file system with this kind of hard link is no longer a tree, because a tree must not contain any loop

man ln on macOS

mydir — less ◂ man ln — 104×33

```
LN(1)                    General Commands Manual                    LN(1)

NAME
     link, ln — make links

SYNOPSIS
     ln [-Ffhinsv] source_file [link_name]
     ln [-Ffhinsv] source_file ... link_dirname
     link source_file link_name

DESCRIPTION
     The ln utility creates a new directory entry (linked file) which has the same modes as the
     original file.  It is useful for maintaining multiple copies of a file in many places at once
     without using up storage for the "copies"; instead, a link "points" to the original copy.
     There are two types of links; hard links and symbolic links.  How a link "points" to a file is
     one of the differences between a hard and symbolic link.

     The options are as follows:

     -F    If the proposed link (link_name) already exists and is a directory, then remove it so
           that the link may occur.  The -F option should be used with either -f or -i options.  If
           none is specified, -f is implied.  The -F option is a no-op unless -s option is
           specified.

     ⋮    ⋮

     By default, ln makes hard links.  A hard link to a file is indistinguishable from the original
     directory entry; any changes to a file are effectively independent of the name used to
     reference the file.  Hard links may not normally refer to directories and may not span file
     systems.

     A symbolic link contains the name of the file to which it is linked.  The referenced file is
     used when an open(2) operation is performed on the link.  A stat(2) on a symbolic link will
     return the linked-to file; an lstat(2) must be done to obtain information about the link.  The
     readlink(2) call may be used to read the contents of a symbolic link.  Symbolic links may span
     file systems and may refer to directories.
```



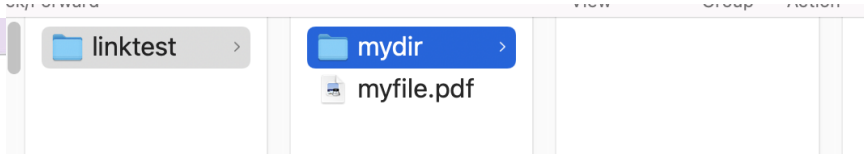
This description is not that clear, at first glance ☹

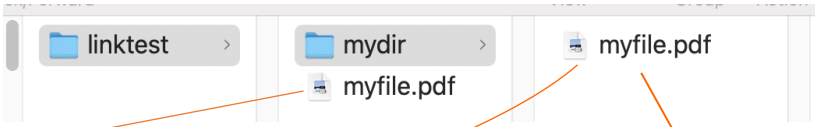
Let's try to clear this up ☺ (next slide)

Directory hardlinks on macOS

The –F option is not needed, because it's no-op when used alone (see next slide)

```
linktest — -zsh — 89×31
marcoautili@iMac linktest %
marcoautili@iMac linktest % ln mydir mydirhardlink
ln: mydir: Is a directory
marcoautili@iMac linktest % ln -F mydir mydirhardlink
ln: mydir: Is a directory
marcoautili@iMac linktest % ln -F myfile.pdf mydir
marcoautili@iMac linktest %
marcoautili@iMac linktest % ls -l myfile.pdf
-rw-r--r--@ 4 marcoautili  staff  863580 Jan 11  2020 myfile.pdf
marcoautili@iMac linktest % ls -l ./mydir/myfile.pdf
-rw-r--r--@ 4 marcoautili  staff  863580 Jan 11  2020 ./mydir/myfile.pdf
marcoautili@iMac linktest %
marcoautili@iMac linktest % ln -F myfile.pdf mydir
ln: mydir/myfile.pdf: File exists
marcoautili@iMac linktest %
marcoautili@iMac linktest % ln -Fs myfile.pdf mydir
marcoautili@iMac linktest %
marcoautili@iMac linktest % ls -l ./mydir/myfile.pdf
lrwxr-xr-x  1 marcoautili  staff  10 Nov 24 13:24 ./mydir/myfile.pdf -> myfile.pdf
marcoautili@iMac linktest %
marcoautili@iMac linktest % readlink ./mydir/myfile.pdf
myfile.pdf
marcoautili@iMac linktest %
```

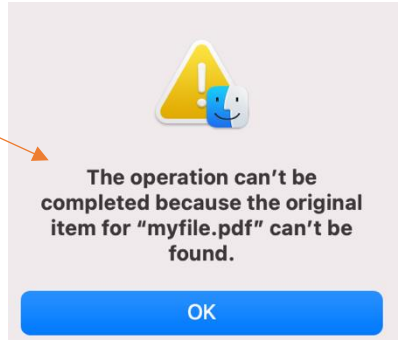




Hardlink inside mydir

Softlink inside mydir

The created softlink does not work because it contains a pathname relative to the current directory even though it has been placed in mydir



Directory hardlinks (2/3)

Directory hardlinks can break the filesystem in multiple ways

(2) Directory hardlinks break the unambiguity of parent directories

- With a filesystem loop, multiple parent directories exist

  cd /tmp/a/b

  cd /tmp/a/b/l/b

- In the first case, /tmp/a is the parent directory of /b

- In the second case, /tmp/a and /tmp/a/b/l are both parent directories of /b

  Thus, /b has two parent directories!

Directory hardlinks (3/3)

Directory hardlinks can break the filesystem in multiple ways

(3) Directory hardlinks can multiply files

- Files are identified by paths (after resolving soft links). Thus, the following would be different files:

  /tmp/a/b/foo.txt

  /tmp/a/b/l/b/foo.txt

- There are infinitely many further paths for the file foo.txt
  Of course, they are the same in terms of their i-node number. But if you do not explicitly expect loops, there is no reason to check for that

- A directory hardlink can also point to a child directory or a directory that is neither child nor parent of any depth. In this case, a file that is a child of the link would be replicated to two files identified by two paths

link() system call on macOS

man 2 link

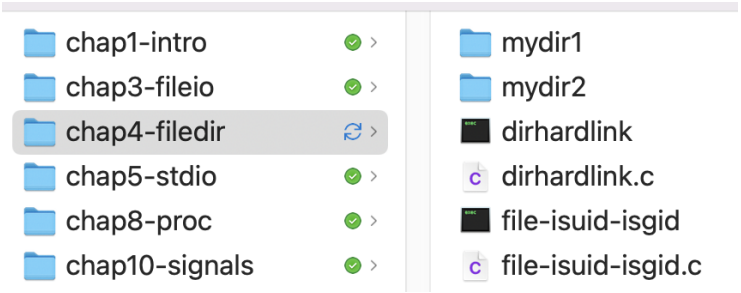
chap4-filedir — less ◂ man 2 link — 109×35

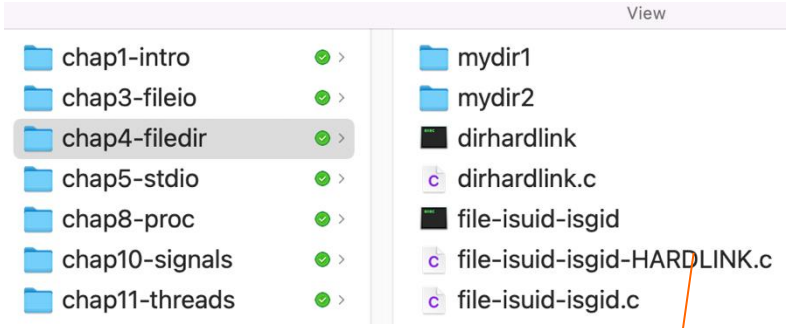
```
LINK(2)                    System Calls Manual                    LINK(2)

NAME
     link, linkat — make a hard file link

SYNOPSIS
     #include <unistd.h>

     int
     link(const char *path1, const char *path2);

     int
     linkat(int fd1, const char *name1, int fd2, const char *name2, int flag);

DESCRIPTION
     The link() function call atomically creates the specified directory entry (hard
     link) path2 with the attributes of the underlying object pointed at by path1.  If
     the link is successful, the link count of the underlying object is incremented;
     path1 and path2 share equal access and rights to the underlying object.

     If path1 is removed, the file path2 is not deleted and the link count of the
     underlying object is decremented.

     In order for the system call to succeed, path1 must exist and both path1 and path2
     must be in the same file system.  As mandated by POSIX.1, path1 may not be a
     directory.
```

Creating hardlinks programmatically

dirhardlink.c

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

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

    if (argc != 3) {
        fprintf(stderr,"Use: hlink <src_dir>
            <target_dir>\n");
        return 1;

    }
    int ret = link(argv[1],argv[2]);

    if (ret != 0)
        perror("link");

    return ret;
}
```





chap4-filedir — -zsh — 97×36

```
marcoautili@iMac chap4-filedir %
marcoautili@iMac chap4-filedir % ./dirhardlink mydir1 mydir1HL
link: Operation not permitted
marcoautili@iMac chap4-filedir %
marcoautili@iMac chap4-filedir % ./dirhardlink mydir1 mydir2/mydir1HL
link: Operation not permitted
marcoautili@iMac chap4-filedir %
marcoautili@iMac chap4-filedir % ./dirhardlink file-isuid-isgid.c file-isuid-isgid-HARDLINK.c
```

Back to writing and reading files

Sharing the same file table entry

- As highlighted on page 77 of the textbook, it is possible for more than one file descriptor to point to the same file table entry, as we'll see later when:
  - Using the dup function (later in this slides and Section 3.12)
  - Using the fork function
    - after a fork when the parent and the child share the same file table entry for each open descriptor (Section 8.3)

Atomic Operations

See Section 3.11

Multiple processes writing the same file

- Problems may arise, e.g., if positioning (lseek) and writing (write) are performed as two separate function calls
- The solution is to have the positioning and the write be an atomic operation with regard to other processes
  - IMPORTANT: Any operation that requires more than one function call cannot be atomic, as there is always the possibility that the kernel might temporarily suspend the process between the two function calls

- Assume that two independent processes, A and B, are appending to the same file. Each has opened the file but *without* the O_APPEND flag
  - This gives us the same picture as Figure 3.8 (see previous slides). Each process has its own file table entry, but they share a single v-node table entry
- The problem here is that the logical operation of "position to the end of file and then write" requires two separate function calls
- The UNIX System provides an atomic way to do this operation if we set the O_APPEND flag when a file is opened
  - In this case no problem arises, since this causes the kernel to position to the end of file before each write, hence calling lseek (before each write) is no longer needed

Atomic Operations

- The functions pread and pwrite allow applications to (1) seek and (2) perform I/O atomically

```
#include <unistd.h>

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

                                Returns: number of bytes read, 0 if end of file, –1 on error

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

                                Returns: number of bytes written if OK, –1 on error
```

Calling `pread` is equivalent to calling `lseek` followed by a call to `read`, with the following exceptions.

- There is no way to interrupt the two operations that occur when we call `pread`.
- The current file offset is not updated.

Calling `pwrite` is equivalent to calling `lseek` followed by a call to `write`, with similar exceptions.

Duplicating a File Descriptor

An existing file descriptor is duplicated by either of the following functions

```
#include <unistd.h>

int dup(int fd);

int dup2(int fd, int fd2);
```

Both return: new file descriptor if OK, –1 on error

The new file descriptor returned by **dup** is guaranteed to be the lowest-numbered available file descriptor. With **dup2**, we specify the value of the new descriptor with the *fd2* argument. If *fd2* is already open, it is first closed. If *fd* equals *fd2*, then **dup2** returns *fd2* without closing it. Otherwise, the **FD_CLOEXEC** file descriptor flag is cleared for *fd2*, so that *fd2* is left open if the process calls **exec**.

- The new file descriptor that is returned as the value of the functions shares the same file table entry as the *fd* argument (next slide)

man -s2 dup

marcoautili — less ‹ man -s2 dup — 94×39

DUP(2) System Calls Manual DUP(2)

NAME

dup, **dup2** — duplicate an existing file descriptor

SYNOPSIS

```
#include <unistd.h>

int
dup(int fildes);

int
dup2(int fildes, int fildes2);
```

DESCRIPTION

dup() duplicates an existing object descriptor and returns its value to the calling process (fildes2 = **dup**(fildes)). The argument fildes is a small non-negative integer index in the per-process descriptor table. The value must be less than the size of the table, which is returned by getdtablesize(2). The new descriptor returned by the call is the lowest numbered descriptor currently not in use by the process.

The object referenced by the descriptor does not distinguish between fildes and fildes2 in any way. Thus if fildes2 and fildes are duplicate references to an open file, read(2), write(2) and lseek(2) calls all move a single pointer into the file, and append mode, non-blocking I/O and asynchronous I/O options are shared between the references. If a separate pointer into the file is desired, a different object reference to the file must be obtained by issuing an additional open(2) call. The close-on-exec flag on the new file descriptor is unset.

In **dup2**(), the value of the new descriptor fildes2 is specified. If fildes and fildes2 are equal, then **dup2**() just returns fildes2; no other changes are made to the existing descriptor. Otherwise, if descriptor fildes2 is already in use, it is first deallocated as if a close(2) call had been done first.

RETURN VALUES

Upon successful completion, the new file descriptor is returned. Otherwise, a value of -1 is returned and the global integer variable errno is set to indicate the error.

Duplicating a File Descriptor

Executing newfd = dup(1);

- Assume that the next available descriptor is 3 (which "probably" is, since 0, 1, and 2 are opened by the shell)
- Because both descriptors point to the same file table entry, they share the same file status flags (read, write, append, and so on) and the same current file offset

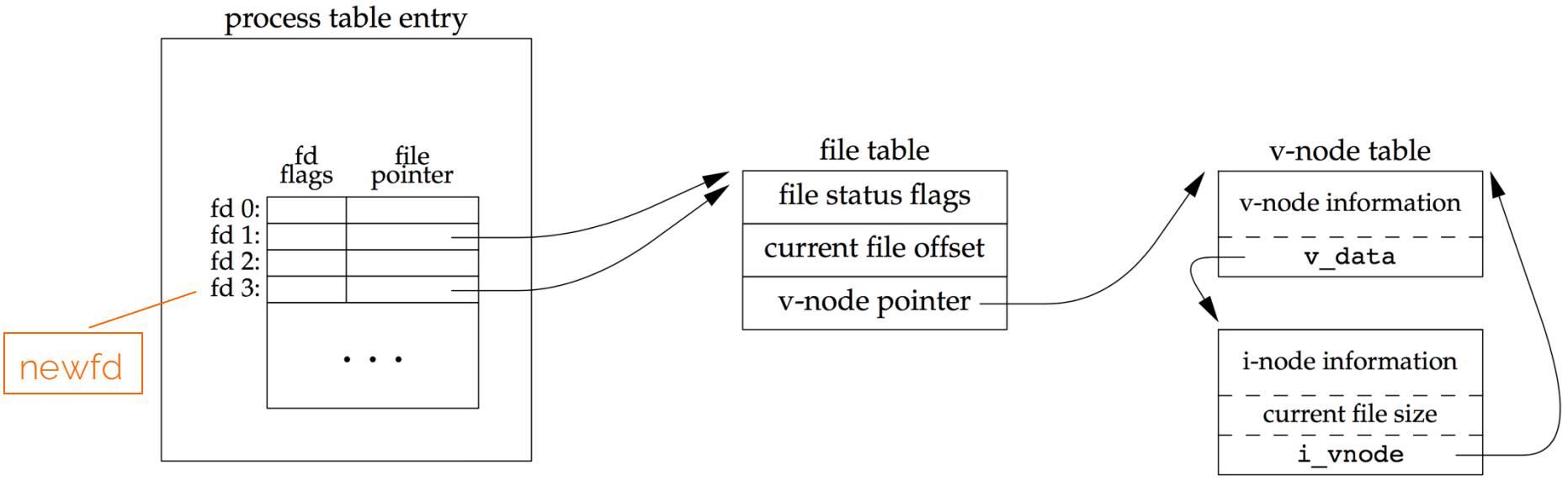


Figure 3.9 Kernel data structures after dup(1)

dup and dup2 practical examples

A typical usage example is I/O redirection

- For this, you fork a child process and close either the stdin or stdout file descriptor (either 0 or 1) and then you call a dup() on another file descriptor of your choice which will now be mapped to the lowest available file descriptor, which is in this case either 0 or 1
- In this way, you can now exec any child process (which is possibly unaware of your application), and whenever the child writes on the stdout (or reads from stdin, whatever you configured) the data gets written on the provided file descriptor instead

- Recall the example

  ls -l pluto.txt minnie.txt > output-file.txt 2>&1

  from PART 2, where minnie.txt was a non-existing file

  - The redirection > can be implemented by using dup() or dup2()
- Shells use this to also implement commands with pipes (e.g., /bin/ls | more) by connecting the stdout of one process to the stdin of the other

dup and dup2 VS fcntl

Another way to duplicate a descriptor is with the fcntl function (described in Section 3.14)

dup(fd);

is equivalent to

fcntl(fd, F_DUPFD, 0);

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

```
#include <fcntl.h>

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

Duplicating a file descriptor
    F_DUPFD (int)
          Duplicate the file descriptor fd using the lowest-numbered
          available file descriptor greater than or equal to arg.
          This is different from dup2(2), which uses exactly the
          file descriptor specified.

          On success, the new file descriptor is returned.

          See dup(2) for further details.
```

Similarly, the call

dup2(fd, fd2);

"can be equivalent" to

close(fd2);

fcntl(fd, F_DUPFD, fd2);

IMPORTANT:

- In this case, the dup2 IS NOT exactly the same as a close followed by an fcntl

dup2 is an atomic operation, whereas the alternate form involves two function calls!

https://digilander.libero.it/uzappi/C/librerie/funzioni/fcntl.html

/dev/fd

- The dev directory contains files that represent various devices

  → more on device files later on these slides when we will introduce the Standard I/O Library

- Newer systems provide the directory /dev/fd whose entries are files named 0, 1, 2, and so on
- Opening the file /dev/fd/*n* is equivalent to duplicating descriptor *n*, assuming that the descriptor *n* is open

It allows programs that use pathname arguments to handle standard input and standard output in the same manner as other pathnames

- In the function call

  fd = open("/dev/fd/0", mode);

  - most systems ignore the specified mode, whereas others require that it be a subset of the mode used when the referenced file (standard input, in this case) was originally opened

- Since the previous open is equivalent to

  fd = dup(0);

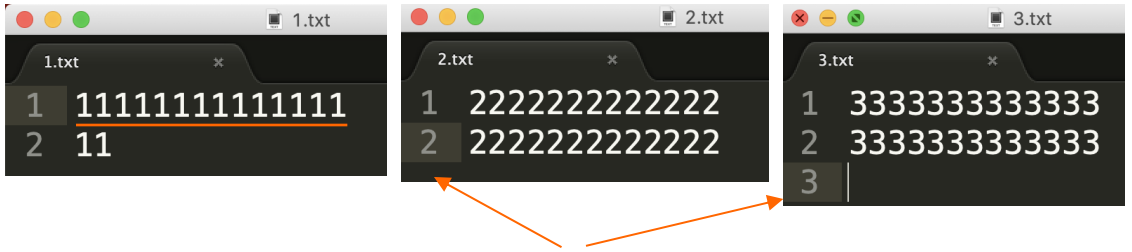
  - the descriptors 0 and fd share the same file table entry for standard input (see Figure 3.9 in previous slide)

/dev/fd

Some systems provide the pathnames

- /dev/stdin
  equivalent to /dev/fd/0
- /dev/stdout
  equivalent to /dev/fd/1
- /dev/stderr
  equivalent to /dev/fd/2

The main use of the /dev/fd files is from the shell

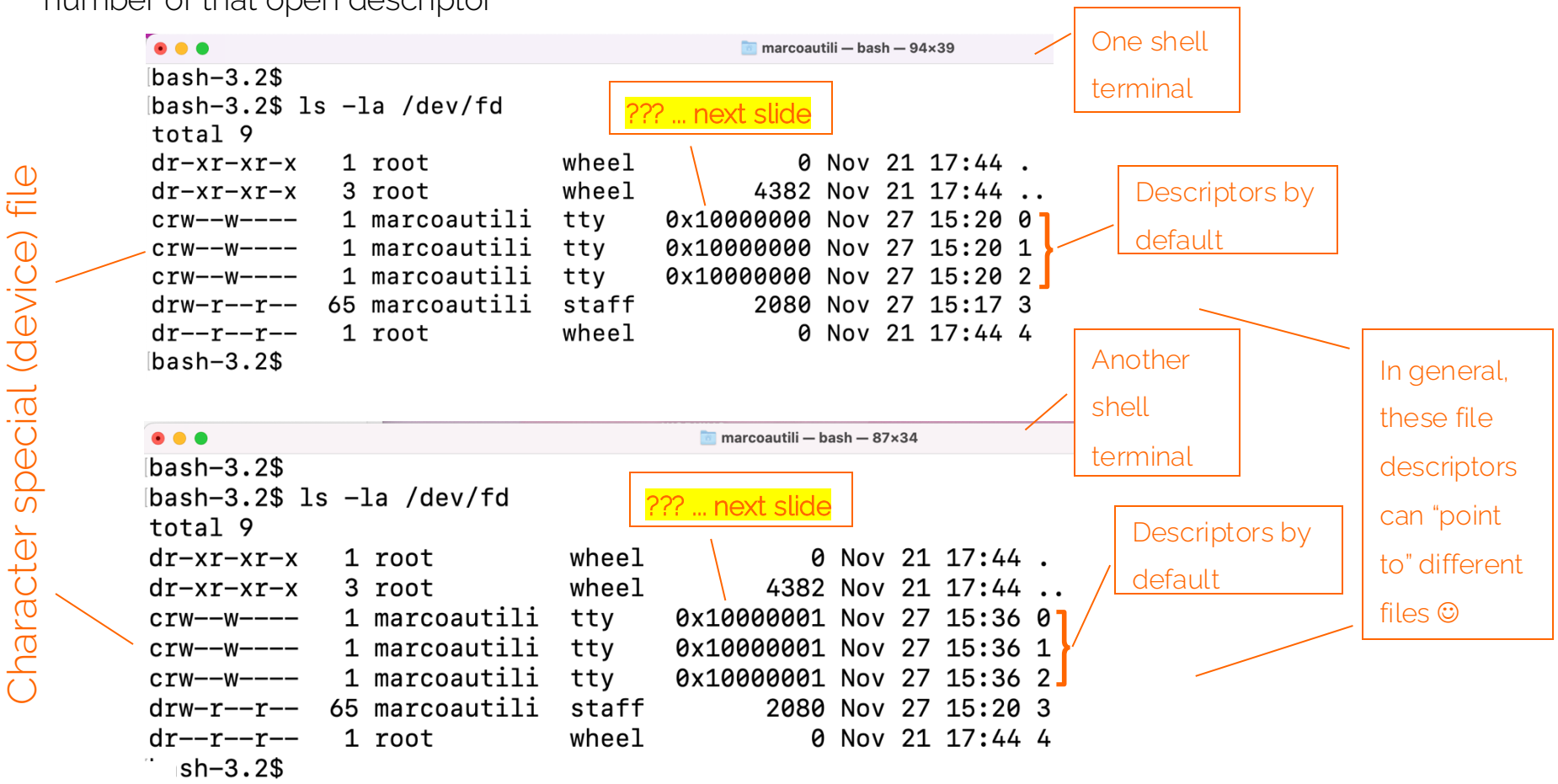



```
ppp — -bash — 78×22
iMac-001:ppp marcoautili$
iMac-001:ppp marcoautili$ egrep '111111' 1.txt | cat - 2.txt 3.txt
11111111111111
2222222222222
22222222222223333333333333
3333333333333
iMac-001:ppp marcoautili$ egrep '111111' 1.txt | cat 2.txt - 3.txt
2222222222222
22222222222211111111111111
3333333333333
3333333333333
iMac-001:ppp marcoautili$ egrep '111111' 1.txt | cat /dev/fd/0 2.txt 3.txt
11111111111111
2222222222222
22222222222223333333333333
3333333333333
iMac-001:ppp marcoautili$ egrep '111111' 1.txt | cat 2.txt /dev/fd/0 3.txt
2222222222222
22222222222211111111111111
3333333333333
3333333333333
iMac-001:ppp marcoautili$
```

"-" to refer to the standard input (or standard output)

/dev/fd

- If two running processes (e.g., two different instance of '/bin/bash') look into "/dev/fd", they could see different things
- Each will see one file for each file descriptor that it has open, and the name of the file will be the number of that open descriptor



/dev/fd

marcoautili — less ‹ man ls — 87×34

If the file is a character special or block special file, the device number for the file is displayed in the size field. If the file is a symbolic link the pathname of the linked-to file is preceded by "->".

The listing of a directory's contents is preceded by a labeled total number of blocks used in the file system by the files which are listed as the directory's contents (which may or may not include . and .. and other files which start with a dot, depending on other options).

The default block size is 512 bytes. The block size may be set with option **-k** or environment variable BLOCKSIZE. Numbers of blocks in the output will have been rounded up so the numbers of bytes is at least as many as used by the corresponding file system blocks (which might have a different size).

The file mode printed under the **-l** option consists of the entry type and the permissions. The entry type character describes the type of file, as follows:

-	Regular file.
b	Block special file.
c	Character special file.
d	Directory.
l	Symbolic link.
p	FIFO.
s	Socket.
w	Whiteout.

Curiosity: older version of my OS

chap4-filedir — -bash — 64×12

```
ste:chap4-filedir marcoautili$
ste:chap4-filedir marcoautili$ ls -la /dev/fd
total 9
dr-xr-xr-x  1 root         wheel          0 Nov 26 08:52 .
dr-xr-xr-x  3 root         wheel       4310 Nov 26 08:52 ..
crw--w----  1 marcoautili  tty       16,   0 Nov 27 15:26 0
crw--w----  1 marcoautili  tty       16,   0 Nov 27 15:26 1
crw--w----  1 marcoautili  tty       16,   0 Nov 27 15:26 2
drw-r--r--  7 marcoautili  staff       224 Nov 27 13:27 3
dr--r--r--  1 root         wheel          0 Nov 26 08:52 4
ste:chap4-filedir marcoautili$
ste:chap4-filedir marcoautili$
```

fd — -ba

```
ste:fd marcoautili$
ste:fd marcoautili$ ls -la /dev/fd
total 9
dr-xr-xr-x  1 root         wheel          0 Nov 26 08:52 .
dr-xr-xr-x  3 root         wheel       4310 Nov 26 08:52 ..
crw--w----  1 marcoautili  tty       16,   1 Nov 27 15:26 0
crw--w----  1 marcoautili  tty       16,   1 Nov 27 15:26 1
crw--w----  1 marcoautili  tty       16,   1 Nov 27 15:26 2
dr--r--r--  1 root         wheel          0 Nov 26 08:52 3
dr--r--r--  1 root         wheel          0 Nov 26 08:52 4
ste:fd marcoautili$
```

For character special (device) files, the information in the size field was different

file /dev/fd/x

marcoautili — bash — 40×39

```
bash-3.2$
bash-3.2$ file /dev/fd/0
/dev/fd/0: character special (16/0)
bash-3.2$ file /dev/fd/1
/dev/fd/1: character special (16/0)
bash-3.2$ file /dev/fd/2
/dev/fd/2: character special (16/0)
bash-3.2$
```

One terminal

major device number

minor device number

Another terminal

marcoautili — bash — 43×34

```
bash-3.2$
bash-3.2$ file /dev/fd/0
/dev/fd/0: character special (16/1)
bash-3.2$ file /dev/fd/1
/dev/fd/1: character special (16/1)
bash-3.2$ file /dev/fd/2
/dev/fd/2: character special (16/1)
bash-3.2$
```

marcoautili — less ‹ man mknod — 116×36

MKNOD(8) System Manager's Manual MKNOD(8)

NAME

mknod — make device special file

SYNOPSIS

mknod [**-F** format] name [**c** | **b**] major minor
mknod [**-F** format] name [**c** | **b**] major unit subunit
mknod name [**c** | **b**] number
mknod name **w**

DESCRIPTION

The **mknod** command creates device special files.

To make nodes manually, the required arguments are:

name Device name, for example “sd” for a SCSI disk on an HP300 or a “pty” for pseudo-devices.

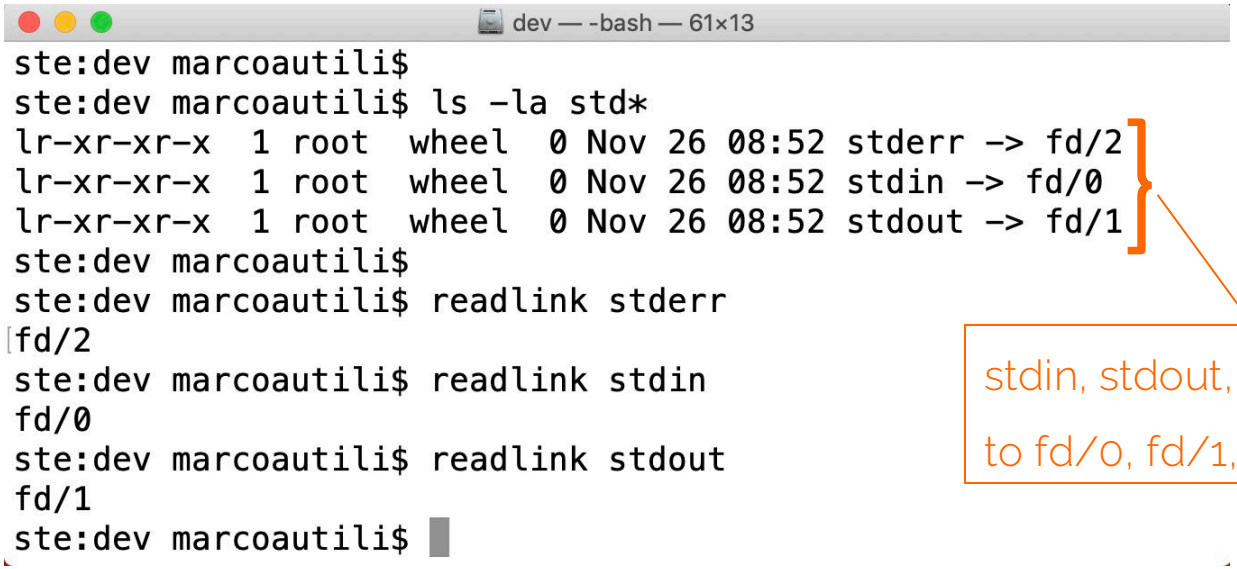
b | **c** | **w**
Type of device. If the device is a block type device such as a tape or disk drive which needs both cooked and raw special files, the type is **b**. Whiteout nodes are type **w**. All other devices are character type devices, such as terminal and pseudo devices, and are type **c**.

major The major device number is an integer number which tells the kernel which device driver entry point to use.

minor The minor device number tells the kernel which one of several similar devices the node corresponds to; for example, it may be a specific serial port or pty.

Check yourself

/dev/fd



stdin, stdout, stderr are symbolic links to fd/0, fd/1, fd/2, respectively ☺

- As a further curiosity, note that, under my macOS, I cannot open the /dev directory in Finder, I can't even see it (even if hidden files are enabled)

Before introducing the Standard I/O Library, it is worth to have a more general discussion on device files…

man fd

marcoautili — less ‹ man fd — 120×46

```
FD(4)                          Device Drivers Manual                          FD(4)

NAME
     fd, stdin, stdout, stderr — file descriptor files

DESCRIPTION
     The files /dev/fd/0 through /dev/fd/# refer to file descriptors which can be accessed through the file system.
     If the file descriptor is open and the mode the file is being opened with is a subset of the mode of the existing
     descriptor, the call:

           fd = open("/dev/fd/0", mode);

     and the call:

           fd = fcntl(0, F_DUPFD, 0);

     are equivalent.

     Opening the files /dev/stdin, /dev/stdout and /dev/stderr is equivalent to the following calls:

           fd = fcntl(STDIN_FILENO,  F_DUPFD, 0);
           fd = fcntl(STDOUT_FILENO, F_DUPFD, 0);
           fd = fcntl(STDERR_FILENO, F_DUPFD, 0);

     Flags to the open(2) call other than O_RDONLY, O_WRONLY and O_RDWR are ignored.

FILES
     /dev/fd/#
     /dev/stdin
     /dev/stdout
     /dev/stderr

SEE ALSO
     tty(4)

macOS 13.5                      June 9, 1993                      macOS 13.5
(END)
```

Device files

- In Unix-like operating systems, a device file or special file is an interface to a device driver that appears in a file system as if it were an ordinary file (located in the /dev directory)
  (There are also special files in DOS, OS/2, and Windows)
- These special files allow an application program to interact with a device by using its device driver via standard input/output system calls (thus, Device Files are not Drivers ☺)
- Using standard system calls simplifies many programming tasks and leads to consistent user-space I/O mechanisms regardless of device features and functions
- Device files usually provide simple interfaces to standard devices (such as printers and serial ports) but can also be used to access specific unique resources on those devices, such as disk partitions
- There are two general kinds of device files in Unix-like operating systems, known as character special files and block special file
  - The difference between them lies in how much data is read and written by the operating system and hardware

https://en.wikipedia.org/wiki/Device_file#Node_creation

Device files (more on)

Character devices - Character special files or character devices provide unbuffered, direct access to the hardware device

- They do not necessarily allow programs to read or write single characters at a time; that is up to the device in question. The character device for a hard disk, for example, will normally require that all reads and writes be aligned to block boundaries and most certainly will not allow reading a single byte

Block devices - Block special files or block devices provide buffered access to hardware devices and provide some abstraction from their specifics. Unlike character devices, block devices will always allow the programmer to read or write a block of any size (including single characters/bytes) and any alignment. The downside is that because block devices are buffered, the programmer does not know how long it will take before written data is passed from the kernel's buffers to the actual device, or indeed in what order two separate writes will arrive at the physical device

- Most systems create both block and character devices to represent hardware like hard disks. FreeBSD and Linux notably do not; the former has removed support for block devices, while the latter creates only block devices. In Linux, to get a character device for a disk, one must use the "raw" driver, though one can get the same effect as opening a character device by opening the block device with the Linux-specific O_DIRECT flag

https://en.wikipedia.org/wiki/Device_file#Node_creation

Pseudo-devices (more on)

Pseudo-devices

Device nodes on Unix-like systems do not necessarily have to correspond to physical devices

Nodes that lack this correspondence form the group of pseudo-devices. They provide various functions handled by the operating system

Some of the most commonly used (character-based) pseudo-devices include:

- /dev/null – accepts and discards all input written to it; provides an end-of-file indication when read from
- /dev/zero – accepts and discards all input written to it; produces a continuous stream of null characters (zero-value bytes) as output when read from
- /dev/full – produces a continuous stream of null characters (zero-value bytes) as output when read from and generates an ENOSPC ("disk full") error when attempting to write to it
- /dev/random – produces bytes generated by the kernel's cryptographically secure pseudorandom number generator: its exact behavior varies by implementation, and sometimes variants such as /dev/urandom or /dev/arandom are also provided

https://en.wikipedia.org/wiki/Device_file#Node_creation

Pseudo-devices

```
bash-3.2$
bash-3.2$ cat /dev/random
```

It will continue to write random bytes to the terminal until you hit Ctrl-C. Don't do this on a low performing system…

```
chap1-intro — ??=o?z?????[?(?ʒhʒU??^$???x?B?k??ym?.`v?ı???J???}?X%????N?/???H — bash — 10
bash-3.2$
bash-3.2$ head -1 /dev/random
?x??[?|?j?&d?[?]|?V1?C?q^P???:G6X&3
                                    ?3?VeU"?8xv???yD??Iu??v?O      ?.w\u'?wY[??Mr?n@??Q`???D
bash-3.2$
bash-3.2$
bash-3.2$ od -d /dev/urandom | head -1
0000000     61452    40931    51091    41745    29710    10543    19834    21226
bash-3.2$
bash-3.2$
bash-3.2$
bash-3.2$ od -d /dev/urandom | head -2
0000000     30422    63952    20401    53018    48565    20247    65103    64851
0000020     57455    18787    28106    42322     3292    49486     8825    43547
bash-3.2$
```

The first row only

The first row only, decimal

The first two rows only, decimal

man urandom

Some device files under macOS (1/2)

File or directory	Description
bpf[0-3]	Berkeley Packet Filter devices. See *bpf(4)*.
console	The system console. This is owned by whoever is currently logged in. If you write to it, the output will end up in */var/tmp/console.log*, which you can view with the Console application (*/Applications/Utilities*).
cu.modem	Modem device for compatibility with the Unix *cu* (call up) utility.
disk[0-n]	Disk.
disk[0-n]s[0-n]	Disk partition. For example, */dev/disk0s1* is the first partition of */dev/disk0*.
fd/	Devices that correspond to file descriptors. See the *fd* manpage for more details.
klog	Device used by *syslogd* to read kernel messages.
kmem	Image of kernel memory.
mem	Image of the system memory.
null	Bit bucket. You can redirect anything here, and it will disappear.

Also referred to as major and minor numbers (see previous slide)

See also "Everything is a File" at:

http://www.linux-databook.info/?page_id=4777

Some device files under macOS (2/2)

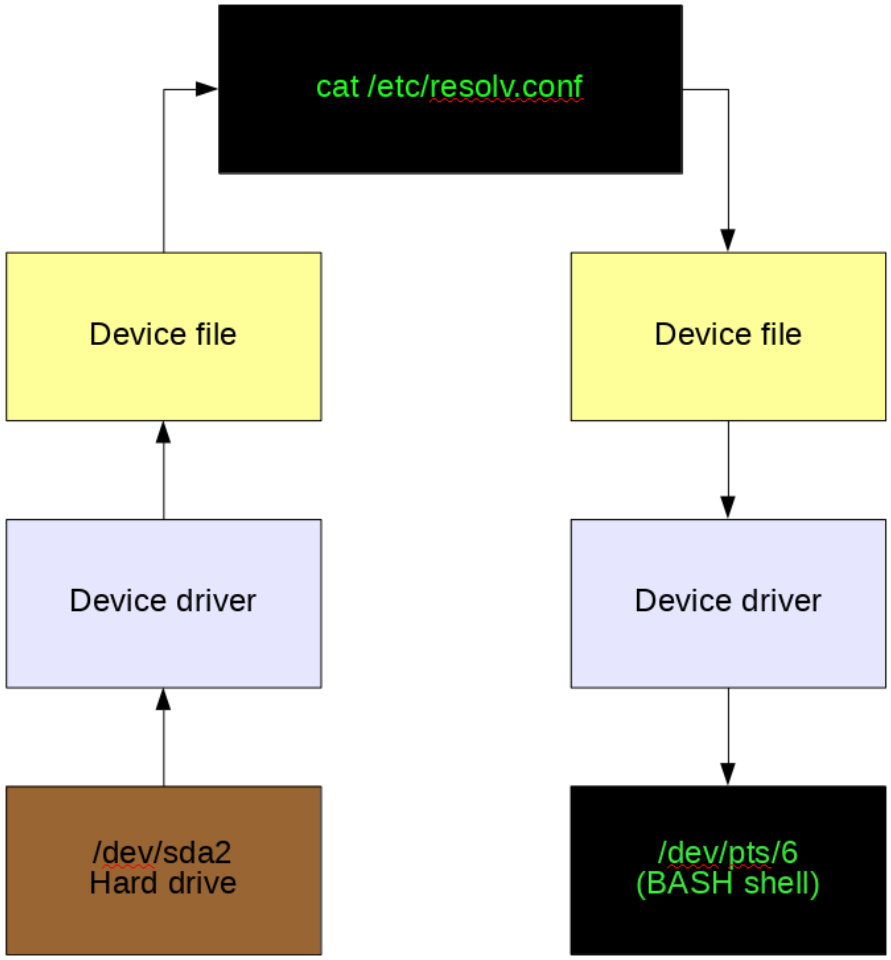
ptyp[0-f]	Master ends of the first sixteen pseudo-*tty*s.
pty[q-w][0-f]	Master ends of the remaining pseudo-*tty*s.
random	Source of pseudorandom data. See *random(4)*.
rdisk[0-n]	Raw disk device.
rdisk[0-n]s[0-n]	Raw disk partition.
stderr	Symbolic link to */dev/fd/2*.
stdin	Symbolic link to */dev/fd/0*.
stdout	Symbolic link to */dev/fd/1*.
tty	Standard output stream of the current Terminal or remote login

Device files @work

Linux-based sample

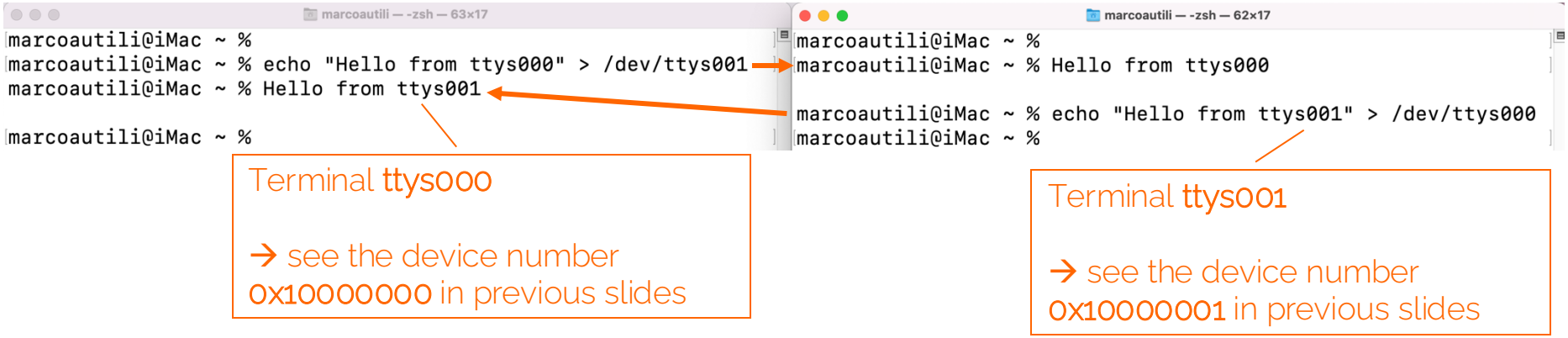
- Data is passed from an application or the operating system to the device file which then passes it to the device driver which then sends it to the physical device

- The reverse data path is also used, from the physical device through the device driver, the device file, and then to an application or another device



http://www.linux-databook.info/?page_id=5108

Playing with device files ☺



Terminal **ttys000**

→ see the device number **0x10000000** in previous slides

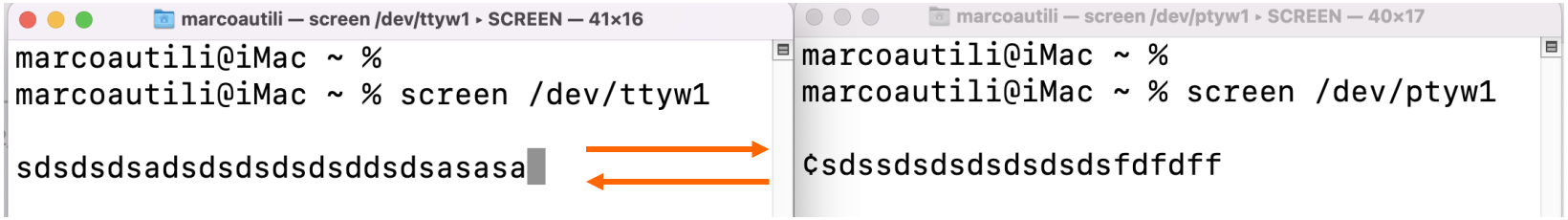
Terminal **ttys001**

→ see the device number **0x10000001** in previous slides

if you type something in one window, it appears in the other (and vice versa)

tty – return user's terminal name

With the new version of my OS, this example "behaves" slightly differently. Ty yourself …



Chapter 5

Standard I/O Library

Standard I/O Library

- The standard I/O functions provide a buffered interface to the unbuffered I/O functions
- Using standard I/O relieves us from having to choose optimal buffer sizes, such as the BUFFSIZE constant
- In fact, the standard I/O library handles such details as buffer allocation and performing I/O in optimal-sized chunks, obviating our need to worry about using the correct block size (as described in next slide)

The goal of the buffering provided by the standard I/O library is to use the minimum number of read and write calls (more on buffering later)

- For example, the standard I/O functions also simplify dealing with lines of input (a common occurrence in UNIX-like applications)
  - e.g., the fgets function reads an entire line; the read function, in contrast, reads a specified number of bytes
- No need to recall that the most common standard I/O function is printf()

One question we haven't answered, however, is how we chose the `BUFFSIZE` value. Before answering that, let's run the program using different values for `BUFFSIZE`. Figure 3.6 shows the results for reading a 516,581,760-byte file, using 20 different buffer sizes.

From Section "3.9 I/O Efficiency" of the textbook

The discussion concerns the previous example "mycat.c"

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

Most file systems support some kind of read-ahead to improve performance

When sequential reads are detected, the system tries to read in more data than an application requests, assuming that the application will read it shortly

The effect of read-ahead can be seen in Figure 3.6, where the elapsed time for buffer sizes as small as 32 bytes is as good as the elapsed time for larger buffer sizes

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

Remember that `st_blksize` is part of the `stat` structure

mycat.c ☺

The file was read using the program shown in Figure 3.5, with standard output redirected to `/dev/null`. The file system used for this test was the Linux `ext4` file system with 4,096-byte blocks. (The `st_blksize` value, which we describe in Section 4.12, is 4,096.) This accounts for the minimum in the system time occurring at the few timing measurements starting around a `BUFFSIZE` of 4,096. Increasing the buffer size beyond this limit has little positive effect.

The null device is a device file that discards all data written to it but reports that the write operation succeeded

Remember: this device is called /dev/null on Unix and Unix-like systems → next slides

File Descriptors versus Streams

- Differently from the unbuffered I/O routines (that focus on File Descriptors), standard I/O library routines focus on Streams
- As we will see, streams are managed by a structure of type FILE
  - In other words,
    - when opening or creating a file with the standard I/O library, we say that we have associated a stream with the file
  - Remember that
    - with the ASCII character set, a single character is represented by a single byte
    - with international character sets, a single character can be represented by more than one byte
- Standard I/O file streams can be used with both
  - single-byte character sets (the stream is said to be byte oriented)
  - multi-byte ("wide") character sets (the stream is said to be wide oriented)
- Only two functions can change the orientation once set (see next slide)

More on Streams orientation

If a multibyte I/O function (see **<wchar.h>**) is used on a stream without orientation, the stream's orientation is set to wide oriented. If a byte I/O function is used on a stream without orientation, the stream's orientation is set to byte oriented. Only two functions can change the orientation once set. The **freopen** function (discussed shortly) will clear a stream's orientation; the **fwide** function can be used to set a stream's orientation.

```
#include <stdio.h>
#include <wchar.h>

int fwide(FILE *fp, int mode);
```

Returns: positive if stream is wide oriented,
negative if stream is byte oriented,
or 0 if stream has no orientation

The **fwide** function performs different tasks, depending on the value of the *mode* argument.

- If the *mode* argument is negative, **fwide** will try to make the specified stream byte oriented.
- If the *mode* argument is positive, **fwide** will try to make the specified stream wide oriented.
- If the *mode* argument is zero, **fwide** will not try to set the orientation, but will still return a value identifying the stream's orientation.

Note that **fwide** will not change the orientation of a stream that is already oriented.

The wchar.h header file declares types and functions for wide-character types, e.g., wprintf, wscanf, fgetwc, fputwc and fwide as well

freopen()
later on this slides

We will only deal with byte-oriented streams

Streams: FILE objects

- When we open a stream, the standard I/O function fopen (Section 5.5) returns a pointer to a FILE object

- This object is normally a structure that contains all the information required by the standard I/O library to manage the stream, e.g.:
  - the file descriptor used for actual I/O
  - a pointer to a buffer for the stream and the size of the buffer
  - a count of the number of characters currently in the buffer
  - an error flag

- Applications should never need to examine a FILE object inside
  - To reference the stream, we pass its FILE pointer as an argument to each standard I/O function
  - We will refer to a pointer to a FILE object as a *file pointer*, i.e., its type is type FILE *

Streams: FILE objects

- Three streams are predefined and automatically available to a process
  - standard input, standard output, and standard error, which are referenced through the predefined file pointers stdin, stdout and stderr, respectively
  - these streams refer to the same files as the file descriptors STDIN_FILENO, STDOUT_FILENO and STDERR_FILENO
- We can obtain the file descriptor associated with a stream by calling the following function

```
#include <stdio.h>
int fileno(FILE *fp);
```

Returns: the file descriptor associated with the stream

Remember `FILE *fp = fdopen(fd, "r+");` from previous slides

Buffering

- As already said, the goal of the buffering provided by the standard I/O library is to use the minimum number of read and write calls
- The following three types of buffering are provided, and the different implementations have their own default types
- If we do not like the provided defaults for any given stream, we can change the buffering by calling either the setbuf or setvbuf function

1. Fully buffered. In this case, actual I/O takes place when the standard I/O buffer is filled. Files residing on disk are normally fully buffered by the standard I/O library. The buffer used is usually obtained by one of the standard I/O functions calling `malloc` (Section 7.8) the first time I/O is performed on a stream.

   The term *flush* describes the writing of a standard I/O buffer. A buffer can be flushed automatically by the standard I/O routines, such as when a buffer fills, or we can call the function `fflush` to flush a stream. Unfortunately, in the UNIX environment, *flush* means two different things. In terms of the standard I/O library, it means writing out the contents of a buffer, which may be partially filled. In terms of the terminal driver, such as the `tcflush` function in Chapter 18, it means to discard the data that's already stored in a buffer.

Buffering

2. Line buffered. In this case, the standard I/O library performs I/O when a newline character is encountered on input or output. This allows us to output a single character at a time (with the standard I/O `fputc` function), knowing that actual I/O will take place only when we finish writing each line. Line buffering is typically used on a stream when it refers to a terminal—standard input and standard output, for example.

   Line buffering comes with two caveats. First, the size of the buffer that the standard I/O library uses to collect each line is fixed, so I/O might take place if we fill this buffer before writing a newline. Second, whenever input is requested through the standard I/O library from either (a) an unbuffered stream or (b) a line-buffered stream (that requires data to be requested from the kernel), *all* line-buffered output streams are flushed. The reason for the qualifier on (b) is that the requested data may already be in the buffer, which doesn't require data to be read from the kernel. Obviously, any input from an unbuffered stream, item (a), requires data to be obtained from the kernel.

3. Unbuffered. The standard I/O library does not buffer the characters. If we write 15 characters with the standard I/O `fputs` function, for example, we expect these 15 characters to be output as soon as possible, probably with the `write` function from Section 3.8.

   The standard error stream, for example, is normally unbuffered so that any error messages are displayed as quickly as possible, regardless of whether they contain a newline.

Opening a Stream

```
#include <stdio.h>

FILE *fopen(const char *restrict pathname, const char *restrict type);

FILE *freopen(const char *restrict pathname, const char *restrict type,
              FILE *restrict fp);

FILE *fdopen(int fd, const char *type);
```

next slide

From file descriptors (int fd)
To streams (* FILE)

All three return: file pointer if OK, NULL on error

The differences in these three functions are as follows:

1. The `fopen` function opens a specified file.
2. The `freopen` function opens a specified file on a specified stream, closing the stream first if it is already open. If the stream previously had an orientation, `freopen` clears it. This function is typically used to open a specified file as one of the predefined streams: standard input, standard output, or standard error.
3. The `fdopen` function takes an existing file descriptor, which we could obtain from the `open`, `dup`, `dup2`, `fcntl`, `pipe`, `socket`, `socketpair`, or `accept` functions, and associates a standard I/O stream with the descriptor. This function is often used with descriptors that are returned by the functions that create pipes and network communication channels. Because these special types of files cannot be opened with the standard I/O `fopen` function, we have to call the device-specific function to obtain a file descriptor, and then associate this descriptor with a standard I/O stream using `fdopen`.

type argument for fopening files

type	Description	open(2) Flags
`r` or `rb`	open for reading	`O_RDONLY`
`w` or `wb`	truncate to 0 length or create for writing	`O_WRONLY|O_CREAT|O_TRUNC`
`a` or `ab`	append; open for writing at end of file, or create for writing	`O_WRONLY|O_CREAT|O_APPEND`
`r+` or `r+b` or `rb+`	open for reading and writing	`O_RDWR`
`w+` or `w+b` or `wb+`	truncate to 0 length or create for reading and writing	`O_RDWR|O_CREAT|O_TRUNC`
`a+` or `a+b` or `ab+`	open or create for reading and writing at end of file	`O_RDWR|O_CREAT|O_APPEND`

Figure 5.2 The *type* argument for opening a standard I/O stream

Restriction	`r`	`w`	`a`	`r+`	`w+`	`a+`
file must already exist	•			•		
previous contents of file discarded		•			•	
stream can be read	•			•	•	•
stream can be written		•	•	•	•	•
stream can be written only at end			•			•

Figure 5.3 Six ways to open a standard I/O stream

The character "b" (as part of the *type*) allows the standard I/O system to differentiate between a text file and a binary file

(see Section 5.5)

stdin, stdout, and stderr as predefined into <stdio.h>

stdio.h

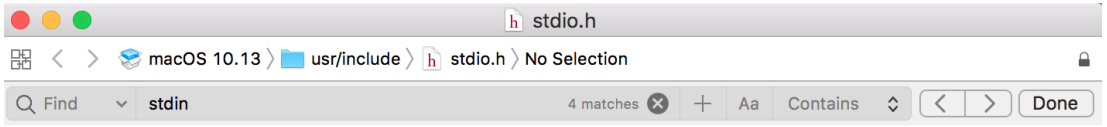
macOS 10.13 〉 usr/include 〉 stdio.h 〉 No Selection

Find stdin 4 matches Aa Contains Done

```
154
155 __BEGIN_DECLS
156 extern FILE *__stdinp;
157 extern FILE *__stdoutp;
158 extern FILE *__stderrp;
159 __END_DECLS
```

When "extern" is used with a variable, it is only declared not defined

The compiler does not need to allocate memory for it since it is allocated elsewhere




```
212 #ifndef SEEK_END
213 #define SEEK_END    2   /* set file offset to EOF plus
        offset */
214 #endif
215
216 #define stdin   __stdinp
217 #define stdout  __stdoutp
218 #define stderr  __stderrp
```

The standard just states that
"FILE is an object type capable of recording all the information needed to control a stream"
(it's up to the implementation)

The only portable way to declare FILE is with #include <stdio.h> (or <cstdio> in C++).

Reading and Writing a Stream

Once we open a stream, we can perform three types of unformatted I/O

1. Character-at-a-time I/O

   We can read or write one character at a time, with the standard I/O functions handling all the buffering, if the stream is buffered (getc, putc, fgetc, fputc, getchar, putchar). The functions getchar and putchar(c) are equivalent to getc(stdin) and putc(c, stdout), respect

2. Line-at-a-time I/O

   If we want to read or write a line at a time, we use gets, fgets and puts, fputs. Each line is terminated with a newline character, and we have to specify the maximum line length that we can handle when we call fgets (see Section 5.7 where these functions are explained)

3. Direct I/O (aka, Binary I/O, object-at-a-time I/O, record-oriented I/O or structure-oriented I/O)

   This type of I/O is supported by the fread and fwrite functions. For each I/O operation, we read or write some number of objects, where each object is of a specified size. These two functions are often used for binary files where we read or write a structure with each operation (see Section 5.9 where these functions are explained)

Formatted I/O can be performed by using the functions, e.g., printf, fprintf, scanf and fscanf

Input and Output (intro)

Unformatted I/O sample

getcputc.c

```c
#include "apue.h"

int
main(void)
{
    int     c;

    while ((c = getc(stdin)) != EOF)
        if (putc(c, stdout) == EOF)
            err_sys("output error");

    if (ferror(stdin))
        err_sys("input error");

    exit(0);
}
```

Line 1, Column 1 Tab Size:

Note that there is no need to define the file pointers stdin and stdout (they are predefined in <stdio.h>)
SEE PREVIOUS SLIDE

See also the equivalent version with fgets and fputs (fgetsfputs.c)

getc reads one character at a time, and this character is written by putc

infile.txt

```
1
2  This is a test file!
3  This is a test file!This is a test file!
4  This is a test file!This is a test file!This is a test file!
5
```

chap1-intro — -bash — 66×9

```
desktop-jisqlks:chap1-intro marcoautili$
desktop-jisqlks:chap1-intro marcoautili$
desktop-jisqlks:chap1-intro marcoautili$ ./getcputc < infile.txt

This is a test file!
This is a test file!This is a test file!
This is a test file!This is a test file!This is a test file!

desktop-jisqlks:chap1-intro marcoautili$
```

getc()/putc() VS fgetc()/fputc()

The difference between getc and fgetc is that getc can be implemented as a macro, whereas fgetc cannot be implemented as a macro.

This means three things:

- if getc is implemented as a macro, it may evaluate stream more than once, so the corresponding argument should never be an expression with side effects

- Since fgetc is guaranteed to be a function, we can take its address. This allows us to pass the address of fgetc as an argument to another function

- Calls to fgetc probably take longer than calls to getc, as it usually takes more time to call a function

https://stackoverflow.com/questions/18480982/getc-vs-fgetc-what-are-the-major-differences
https://en.cppreference.com/w/c/io/fgetc

Print buffering for various standard I/O streams

```
chap5-stdio — -zsh — 91×33
marcoautili@iMac chap5-stdio %
marcoautili@iMac chap5-stdio % ./buf
enter any character
a
one line to standard error
stream = stdin, line buffered, buffer size = 4096
stream = stdout, line buffered, buffer size = 4096
stream = stderr, unbuffered, buffer size = 1
stream = /etc/passwd, fully buffered, buffer size = 4096
marcoautili@iMac chap5-stdio %
marcoautili@iMac chap5-stdio %
marcoautili@iMac chap5-stdio %
marcoautili@iMac chap5-stdio % ./buf < fgetsfputs.c > bufstdout.txt 2> bufstderr.txt
marcoautili@iMac chap5-stdio %
marcoautili@iMac chap5-stdio %
marcoautili@iMac chap5-stdio %
marcoautili@iMac chap5-stdio % cat bufstdout.txt
enter any character
stream = stdin, fully buffered, buffer size = 4096
stream = stdout, fully buffered, buffer size = 4096
stream = stderr, unbuffered, buffer size = 1
stream = /etc/passwd, fully buffered, buffer size = 4096
marcoautili@iMac chap5-stdio %
marcoautili@iMac chap5-stdio %
marcoautili@iMac chap5-stdio %
marcoautili@iMac chap5-stdio %
marcoautili@iMac chap5-stdio % cat bufstderr.txt
one line to standard error
marcoautili@iMac chap5-stdio %
```

See the example chap5-stdio/buf.c

Study the code and then have a look at your stdio.h (see next slide)

regular file defaults to fully buffered

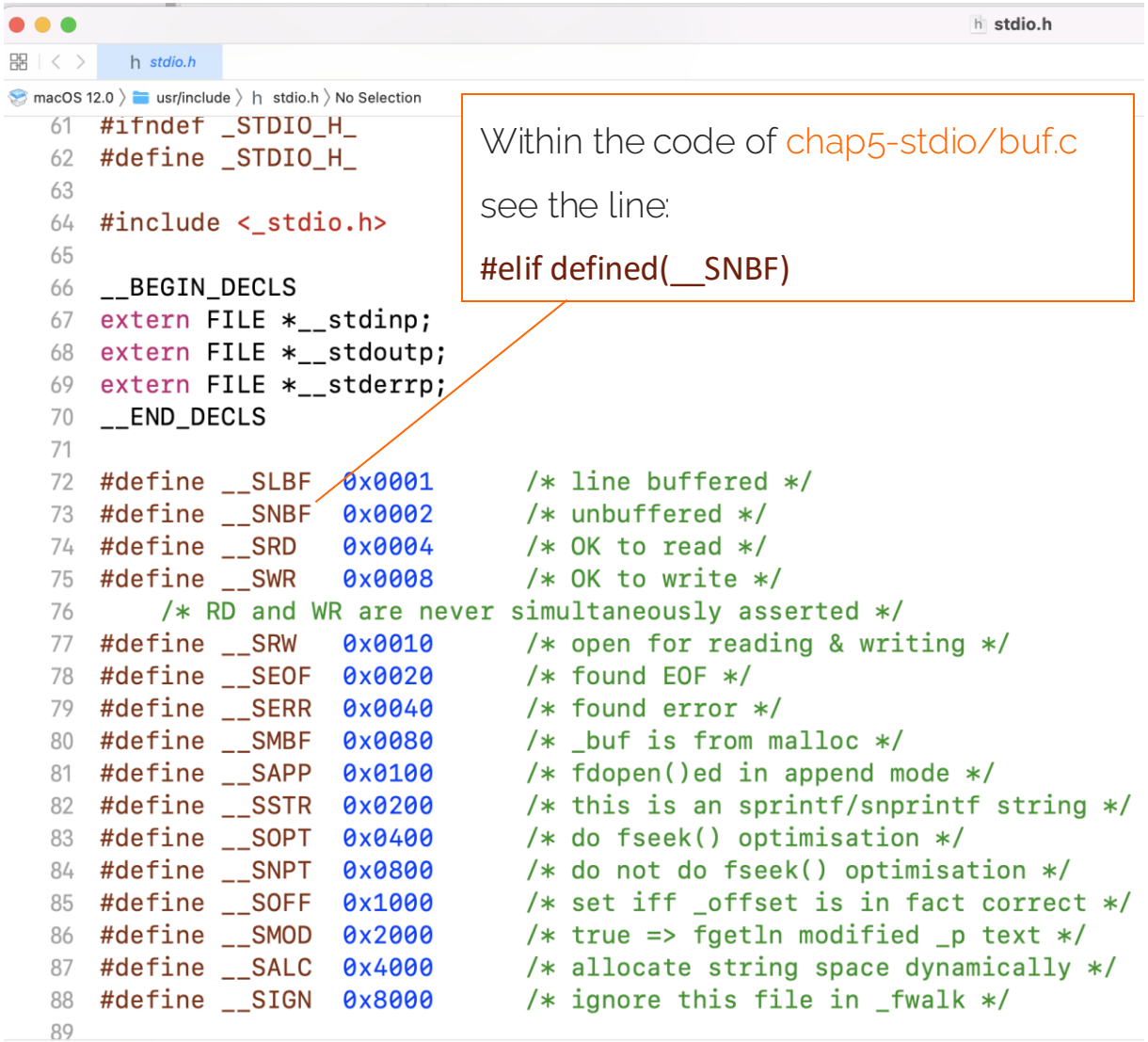
standard error is always unbuffered

Compare this output with that of the textbook (→ next slide)

Print buffering for various standard I/O streams

```
$ ./a.out                                   stdin, stdout, and stderr connected to terminal
enter any character
                                            we type a newline
one line to standard error
stream = stdin, line buffered, buffer size = 1024
stream = stdout, line buffered, buffer size = 1024
stream = stderr, unbuffered, buffer size = 1
stream = /etc/passwd, fully buffered, buffer size = 4096
$ ./a.out < /etc/group > std.out 2> std.err
                                            run it again with all three streams redirected
$ cat std.err
one line to standard error
$ cat std.out
enter any character
stream = stdin, fully buffered, buffer size = 4096
stream = stdout, fully buffered, buffer size = 4096
stream = stderr, unbuffered, buffer size = 1
stream = /etc/passwd, fully buffered, buffer size = 4096
```

stdio.h under my macOS



Within the code of chap5-stdio/buf.c see the line:

#elif defined(__SNBF)

Remember that the internals of a FILE struct are implementation-dependent

For example, in my macOS system, <stdio.h> defines the values on the left for its own _flags field of the FILE struct (i.e., fp->_flags field in the code of chap5-stdio/buf.c)

The _flags field is generally a bitmask which the stdio package uses to keep track of various options which have been set (read, write, line-buffered, etc.), and various other bits of state concerning the stream, such as whether it's hit EOF

stdio.h under a GNU/Linux Debian distribution

stdio.h

h stdio.h

h stdio.h 〉 No Selection

Find BUFSIZ

```
76  # ifndef __ssize_t_defined
77  typedef __ssize_t ssize_t;
78  # define __ssize_t_defined
79  # endif
80  #endif
81
82  /* The type of the second argument to `fgetpos' and `fsetpos'.  */
83  #ifndef __USE_FILE_OFFSET64
84  typedef __fpos_t fpos_t;
85  #else
86  typedef __fpos64_t fpos_t;
87  #endif
88  #ifdef __USE_LARGEFILE64
89  typedef __fpos64_t fpos64_t;
90  #endif
91
92  /* The possibilities for the third argument to `setvbuf'.  */
93  #define _IOFBF 0        /* Fully buffered.  */
94  #define _IOLBF 1        /* Line buffered.  */
95  #define _IONBF 2        /* No buffering.  */
96
97
98  /* Default buffer size.  */
99  #define BUFSIZ 8192
100
101
102 /* The value returned by fgetc and similar functions to indicate the
103    end of the file.  */
104 #define EOF (-1)
```

Within the code of chap5-stdio/buf.c

see the line:

#elif defined(_IONBF)

Within the code of chap5-stdio/buf.c see the line:

return(BUFSIZ); /* just a guess */

stdio.h under a Solaris distribution (more on …)

```
36  #ifdef  _LP64
37
38  #ifndef _FILE64_H
39
40  struct __FILE_TAG {
41          long    __pad[16];
42  };
43
44  #endif  /* _FILE64_H */
45
46  #else
47
48  struct __FILE_TAG       /* needs to be binary-compatible with old versions */
49  {
50  #ifdef _STDIO_REVERSE
51          unsigned char   *_ptr;  /* next character from/to here in buffer */
52          int             _cnt;   /* number of available characters in buffer */
53  #else
54          int             _cnt;   /* number of available characters in buffer */
55          unsigned char   *_ptr;  /* next character from/to here in buffer */
56  #endif
57          unsigned char   *_base; /* the buffer */
58          unsigned char   _flag;  /* the state of the stream */
59          unsigned char   _magic; /* Old home of the file descriptor */
60                                  /* Only fileno(3C) can retrieve the value now */
61          unsigned        __orientation:2; /* the orientation of the stream */
62          unsigned        __ionolock:1;   /* turn off implicit locking */
63          unsigned        __seekable:1;   /* is file seekable? */
64          unsigned        __extendedfd:1; /* enable extended FILE */
65          unsigned        __xf_nocheck:1; /* no extended FILE runtime check */
66          unsigned        __filler:10;
67  };
```

Within the code of chap5-stdio/buf.c see the line:

```
#ifdef _LP64
#define _flag __pad[4]
#define _ptr __pad[1]
#define _base __pad[2]
#endif
```

Some function prototypes

From appendix A of the text book (pag. 845)

```
int        fputc(int c, FILE *fp);
                  <stdio.h>                                          p. 152
                  Returns: c if OK, EOF on error

int        fputs(const char *restrict str, FILE *restrict fp);
                  <stdio.h>                                          p. 153
                  Returns: non-negative value if OK, EOF on error

int        fgetc(FILE *fp);
                  <stdio.h>                                          p. 150
                  Returns: next character if OK, EOF on end of file or error

char      *fgets(char *restrict buf, int n, FILE *restrict fp);
                  <stdio.h>                                          p. 152
                  Returns: buf if OK, NULL on end of file or error
```

Some function prototypes

From appendix A of the reference book (pag. 845)

```
size_t    fread(void *restrict ptr, size_t size, size_t nobj,
                FILE *restrict fp);
                      <stdio.h>                                    p. 156
                      Returns: number of objects read

size_t    fwrite(const void *restrict ptr, size_t size, size_t nobj,
                 FILE *restrict fp);
                      <stdio.h>                                    p. 156
                      Returns: number of objects written

int       fscanf(FILE *restrict fp, const char *restrict format, ...);
                      <stdio.h>                                    p. 162
                      Returns: number of input items assigned, EOF if input
                               error or end of file before any conversion

int       fprintf(FILE *restrict fp, const char *restrict format, ...);
                      <stdio.h>                                    p. 159
                      Returns: number of characters output if OK, negative value
                               if output error
```

Some function prototypes

```
int     feof(FILE *fp);
```
<stdio.h> p. 151

Returns: nonzero (true) if end of file on stream,
0 (false) otherwise

```
int     fseek(FILE *fp, long offset, int whence);
```
<stdio.h> p. 158

whence: SEEK_SET, SEEK_CUR, SEEK_END

Returns: 0 if OK, –1 on error

For a binary file, a file's position indicator is measured in bytes from the beginning of the file

```
int     fseeko(FILE *fp, off_t offset, int whence);
```
<stdio.h> p. 158

whence: SEEK_SET, SEEK_CUR, SEEK_END

Returns: 0 if OK, –1 on error

```
long    ftell(FILE *fp);
```
<stdio.h> p. 158

Returns: current file position indicator if OK, –1L on error

```
off_t   ftello(FILE *fp);
```
<stdio.h> p. 158

Returns: current file position indicator if OK,
(off_t)–1 on error

fread() and fwrite() common uses

A fundamental problem with binary I/O is that it can be used to read only data that has been written on the same system

We cannot write data on one system and process it on another… Most probably, it will not work properly ☹

1. Read or write a binary array. For example, to write elements 2 through 5 of a floating-point array, we could write

```
float   data[10];

if (fwrite(&data[2], sizeof(float), 4, fp) != 4)
    err_sys("fwrite error");
```

It will write 4 elements from data[2] to data[5]

Here, we specify *size* as the size of each element of the array and *nobj* as the number of elements.

2. Read or write a structure. For example, we could write

```
struct {
  short  count;
  long   total;
  char   name[NAMESIZE];
} item;

if (fwrite(&item, sizeof(item), 1, fp) != 1)
    err_sys("fwrite error");
```

Back to data alignment, note that we actually write aligned data (hence, data + padding)

Here, we specify *size* as the size of structure and *nobj* as 1 (the number of objects to write).

Play with the examples in the folder chap5-stdio