

Coming up...

- Exam 2 handed back next Tuesday
- Malloc lab due 1 week from today
 - Considerable tweaking needed to get most of the points
 - Do not delay starting
- One more lab after that
- Just 4 lectures left!
- Final exam
 - 25% material from first third of class
 - 25% from second third
 - 50% from final third

1

Last Time: More Heap Allocation

- Explicitly linked free lists
 - Doubly linked
 - Advantages? Disadvantages?
- Segregated free lists
- Garbage collection
- Memory-related perils and pitfalls

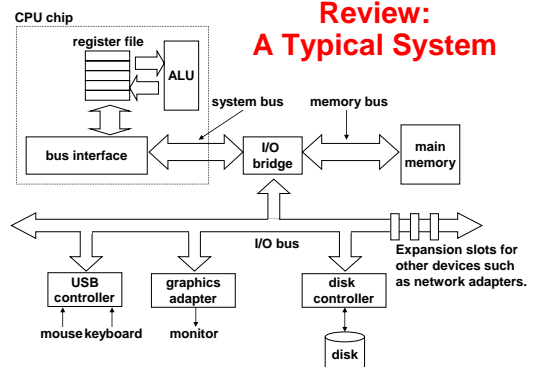
2

Today: System-Level I/O

- Unix I/O
- Robust reading and writing
- Reading file metadata
- Sharing files
- I/O redirection
- Standard I/O

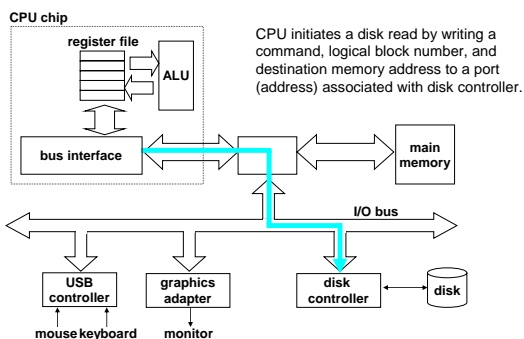
3

Review: A Typical System



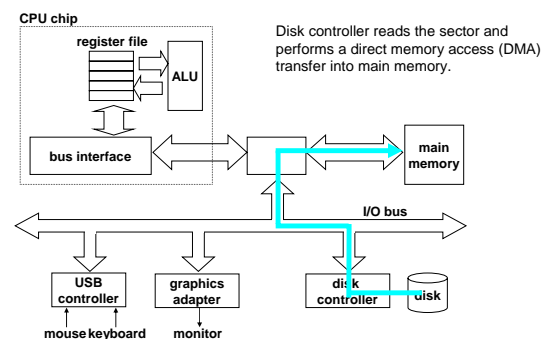
4

Reading a Disk Sector: Step 1



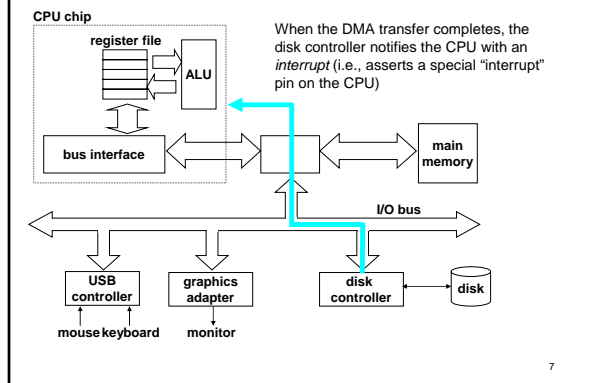
5

Reading a Disk Sector: Step 2



6

Reading a Disk Sector: Step 3



The File Abstraction in UNIX

- A Unix *file* is a sequence of m bytes:
 - B_0, B_1, \dots, B_{m-1}
 - Physical and virtual I/O devices are represented as files:
 - `/dev/sda2` (third partition on the first SCSI disk)
 - `/dev/tty2` (third virtual terminal)
 - `/dev/mouse`
 - `/dev/cdrom`
 - Even the kernel is accessed through files
 - `/dev/kmem` (kernel memory image)
 - `/proc` (user-friendly view of kernel data structures)
 - Pushing the file abstraction is arguably UNIX's most interesting design feature
- 8

Unix File Types

- Regular file – sequence of bytes on disk
 - Executable, text file, data file, etc.
 - Unix (usually) does not enforce file types (c.f. Mac OS)
 - Directory file
 - Special file that contains the names and locations of other files.
 - Character and block device files
 - Terminals (character) and disks (block)
 - FIFO (named pipe)
 - A file type used for interprocess communication
 - » e.g. between processes on the same host
 - Socket descriptor
 - Used for network communication between processes
 - » Potentially on different hosts
 - Often used like a file descriptor but not always
- 9

Unix I/O

- Mapping of files to devices
 - allows kernel to export a simple interface called Unix I/O
 - Important Unix idea: Most input and output is handled in a consistent and uniform way
 - Little difference between a keyboard device and a file on a disk device
 - Basic Unix I/O operations (system calls):
 - Opening and closing files
 - » `open()` and `close()`
 - Changing the *current file position* (seek)
 - » `lseek` (not discussed)
 - Reading and writing a file
 - » `read()` and `write()`
 - Everything else
 - » `ioctl()`
 - » Allows a lot of stuff to be swept under the rug!
- 10

Non-UNIX I/O

- MS Windows
 - File abstraction used only for actual files
 - Additional interfaces were designed to support other kinds of I/O
 - More interfaces means there's more to learn and resulting code is less generic
 - Plan 9
 - Pushes the file abstraction even farther than UNIX does
 - Does a much better job of eliminating the distinction between local and remote objects
 - Mostly a research OS, not commonly used
- 11

Opening a File

- ```
int open (char *filename,
 int flags,
 mode_t mode)
```

    - Filename is something like `/etc/fstab`
    - Flags: controls what to happens on reads or writes
    - Mode: controls access permissions of new files
- 12

## More on Flags

- Control by ORing constants defined in `fcntl.h`
  - 1 of these 3 must be specified
    - » `O_RDONLY`, `O_WRONLY`, `O_RDWR`
  - optional
    - » `O_APPEND` → append to the end of the file on a write
    - » `O_CREAT` → create the file if it doesn't exist
    - » `O_EXCL` → if the file exists and `O_CREAT` is set then generate an error
    - » `O_TRUNC` → if the file exists, set its length to 0
    - » `O_NONBLOCK` → non-blocking open and I/O if a FIFO, block or char special
    - » `O_SYNC` → each write must wait for physical I/O to complete
    - » Etc...

13

## More on Modes

- `umask`
  - each process has a variable called the `umask`
    - » `>` `umask` returns the existing `umask`
    - » `>` `umask 022` sets the `umask` to `022`
    - » Digits are octal
- 3 types of accessors
  - `user`, `group`, `other` (a.k.a. `world`)
  - » each specified by a `rwX` 3-bit field
- system defaults
  - regular files are default created with `mode=666`
  - directories are created with `default=777`
    - » the default applies when a specific creation mode is not supplied
- Permissions for a new file: `mode & ~umask`

14

## Opening Files

- Opening a file informs the kernel that you are getting ready to access that file.

```
int fd; /* file descriptor */
if ((fd = open("/etc/hosts", O_RDONLY)) < 0) {
 perror("open");
 exit(1);
}
```

- Returns a small identifying integer *file descriptor*
  - `fd == -1` indicates that an error occurred
- Each process created by a Unix shell begins life with three open files mapped by default to the terminal device
  - 0: standard input
  - 1: standard output
  - 2: standard error

15

## Closing Files

- Closing a file informs the kernel that you are finished accessing that file.

```
int fd; /* file descriptor */
int retval; /* return value */
if ((retval = close(fd)) < 0) {
 perror("close");
 exit(1);
}
```

- Always check return codes, even for seemingly benign functions such as `close()`

16

## Reading from Files

- Reading a file copies bytes from the current file position to memory, and then updates file position.

```
char buf[512];
int fd; /* file descriptor */
int nbytes; /* number of bytes read */

/* Open file fd ... */
/* Then read up to 512 bytes from file fd */
if ((nbytes = read(fd, buf, sizeof(buf))) < 0) {
 perror("read");
 exit(1);
}
```

- Returns number of bytes read from file `fd` into `buf`
  - `nbytes < 0` indicates that an error occurred.
  - *short counts* (`nbytes < sizeof(buf)`) are possible and are not errors!
  - What does this mean for the programmer?

17

## Writing to Files

- Writing a file copies bytes from memory to the current file position, and then updates current file position.

```
char buf[512];
int fd; /* file descriptor */
int nbytes; /* number of bytes read */

/* Open the file fd ... */
/* Then write up to 512 bytes from buf to file fd */
if ((nbytes = write(fd, buf, sizeof(buf))) < 0) {
 perror("write");
 exit(1);
}
```

- Returns number of bytes written from `buf` to file `fd`
  - `nbytes < 0` indicates that an error occurred.
  - As with reads, short counts are possible and are not errors!
- Transfers up to 512 bytes from address `buf` to file `fd`

18

## File Metadata

- **Metadata** is data about data
- Metadata about files maintained by kernel, accessed by users with the `stat` and `fstat` functions

```
/* Metadata returned by the stat and fstat functions */
struct stat {
 dev_t st_dev; /* device */
 ino_t st_ino; /* inode */
 mode_t st_mode; /* protection and file type */
 nlink_t st_nlink; /* number of hard links */
 uid_t st_uid; /* user ID of owner */
 gid_t st_gid; /* group ID of owner */
 dev_t st_rdev; /* device type (if inode device) */
 off_t st_size; /* total size, in bytes */
 unsigned long st_blksize; /* blocksize for filesystem I/O */
 unsigned long st_blocks; /* number of blocks allocated */
 time_t st_atime; /* time of last access */
 time_t st_mtime; /* time of last modification */
 time_t st_ctime; /* time of last change */
};
```

19

## Metadata and Macros

- Macros from `<sys/stat.h>` help us determine **file type** from the `st_mode` bits
  - `S_ISREG` → regular file
  - `S_ISDIR` → directory file
  - `S_ISSOCK` → network socket
  - `S_ISCHR` → character special
  - `S_ISBLK` → block special
  - `S_ISFIFO` → pipe
  - `S_ISLNK` → symbolic link

20

## Example of Accessing File Metadata

```
/* statcheck.c - Querying and manipulating a file's meta data */
#include "csapp.h"

int main (int argc, char **argv)
{
 struct stat stat;
 char *type, *readok;

 Stat(argv[1], &stat);
 if (S_ISREG(stat.st_mode)) /* file type*/
 type = "regular";
 else if (S_ISDIR(stat.st_mode))
 type = "directory";
 else
 type = "other";
 if ((stat.st_mode & S_IRUSR)) /* OK to read?*/
 readok = "yes";
 else
 readok = "no";

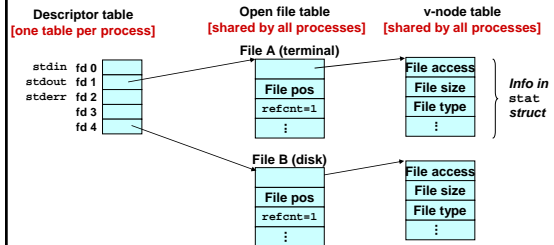
 printf("type: %s, read: %s\n", type, readok);
 exit(0);
}
```

```
sinner> ./statcheck statcheck.c
type: regular, read: yes
sinner> chmod 000 statcheck.c
sinner> ./statcheck statcheck.c
type: regular, read: no
```

21

## How the Unix Kernel Represents Open Files

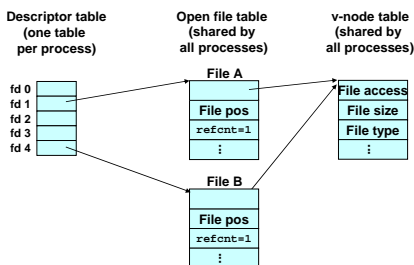
- Two descriptors referencing two distinct open files. Descriptor 1 (stdout) points to terminal, descriptor 4 points to open disk file.



22

## File Sharing

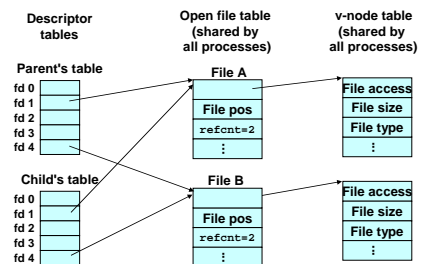
- Two distinct descriptors sharing the same disk file through two distinct open file table entries
  - E.g., Calling `open` twice with the same filename argument



23

## How Processes Share Files

- A child process inherits its parent's open files. Here is the situation immediately after a `fork`



24

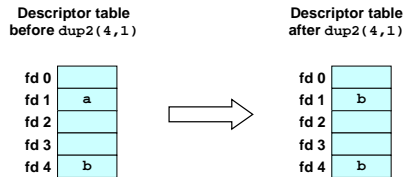
## I/O Redirection

- **Question: How does a shell implement redirection?**

```
$ ls > foo.txt
```

- **Answer:**

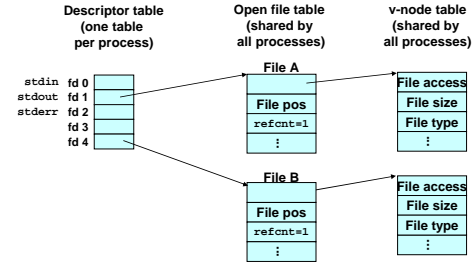
1. Parent: call fork
2. Child: open foo.txt for writing
3. Child: call the `dup2(oldfd,newfd)` function
4. Child: exec "ls"



25

## I/O Redirection Example

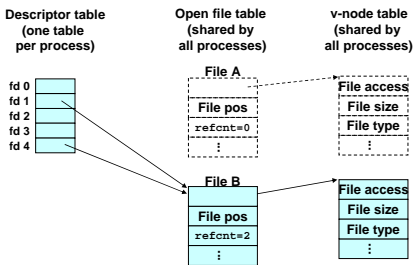
- Before calling `dup2(4,1)`, `stdout` (descriptor 1) points to a terminal and descriptor 4 points to an open disk file.



26

## I/O Redirection Example (cont)

- After calling `dup2(4,1)`, `stdout` is now redirected to the disk file pointed at by descriptor 4.



27

## Back To Short Counts

- **Short counts can occur in these situations:**
  - Encountering (end-of-file) EOF on reads
  - Reading text lines from a terminal
    - » read's from a character special file terminate on end of line
      - `\n` or `0x0A` in Linux
  - Reading and writing network sockets or Unix pipes
    - » this will be very important in the next couple of weeks
- **Short counts never occur in these situations:**
  - Reading from disk files (except for EOF)
  - Writing to disk files
- **As a programmer, how should you deal with short counts?**
  - Well, they are painful and uninteresting
  - So hide them from high-level code!

28

## Higher-Level I/O: Standard I/O

- The C standard library (`libc.a`) contains a collection of higher-level standard I/O functions
  - Documented in Appendix B of K&R 2e
  - Fundamental data type is the `FILE *` (pronounce "file pointer")
    - » Not to be confused with file descriptors
- **Examples of standard I/O functions:**
  - Opening and closing files (`fopen` and `fclose`)
  - Reading and writing bytes (`fread` and `fwrite`)
  - Reading and writing text lines (`fgets` and `fputs`)
    - » `gets` reads next line including `\n`
    - » `puts` writes a string to a file (newline may not be present)
  - Formatted reading and writing (`fscanf` and `fprintf`)
    - » same as `scanf` and `printf`
      - I/O format guided by a format string
    - » except the first argument is a file pointer

29

## Standard I/O Streams

- Standard I/O models open files as **streams**
  - Abstraction for a file descriptor and a buffer in memory
- Most UNIX processes begin life with three open file descriptors
- For C programs these are available as the `FILE *s`:
  - `stdin` (standard input)
  - `stdout` (standard output)
  - `stderr` (standard error)

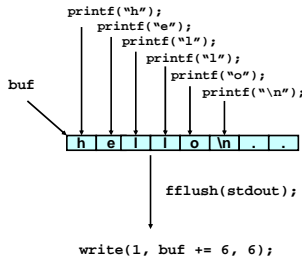
```
#include <stdio.h>
extern FILE *stdin; /* standard input (descriptor 0) */
extern FILE *stdout; /* standard output (descriptor 1) */
extern FILE *stderr; /* standard error (descriptor 2) */

int main() {
 fprintf(stdout, "Hello, world\n");
}
```

30

## Buffering in Standard I/O

- Standard I/O functions use buffered I/O



31

## Standard I/O Buffering in Action

- You can see this buffering in action for yourself, using the always fascinating Unix `strace` program:
  - system utility that traces system calls and signals (man for details)

```
#include <stdio.h>
int main()
{
 printf("h");
 printf("e");
 printf("l");
 printf("l");
 printf("o");
 printf("\n");
 fflush(stdout);
 exit(0);
}
```

```
linux> strace ./hello
execve("./hello", ["hello"], [/* ... */]).
...
write(1, "hello\n", 6...) = 6
...
_exit(0) = ?
```

32

## A Lower-Level Alternative to STDIO: The RIO Package

- RIO is a set of wrappers that provide efficient and robust I/O in applications such as network programs that are subject to short counts
- RIO provides two different kinds of functions
  - Unbuffered input and output of binary data
    - » `rio_readn` and `rio_writen`
  - Buffered input of binary data and text lines
    - » `rio_readlineb` and `rio_readnb`
  - The buffered RIO routines are *thread-safe* and can be interleaved arbitrarily on the same descriptor
- Download from CS:APP web site

33

## Unbuffered RIO Input and Output

- Same interface as Unix `read` and `write`
- Especially useful for transferring data on network sockets

```
#include "csapp.h"
ssize_t rio_readn(int fd, void *usrbuf, size_t n);
ssize_t rio_writen(int fd, void *usrbuf, size_t n);

Return: num. bytes transferred if OK, 0 on EOF (rio_readn only), -1 on error
```

- `rio_readn` returns short count only if it encounters EOF.
- `rio_writen` never returns a short count.
- Calls to `rio_readn` and `rio_writen` can be interleaved arbitrarily on the same descriptor.

34

## What RIO Does

- The problem
  - system calls may fail
  - they may get interrupted by a signal handler return
  - in this case a short count may result
  - this difference may or may not be a pain
  - if it is use Rio
- The RIO solution
  - very simple
  - maintain separate pointer and counts
  - detect whether the system call was interrupted or not
  - if yes:
    - » retry the read or write
  - if not interrupted:
    - » update the real file pointer and counts

35

## Implementation of `rio_readn`

```
/*
 * rio_readn - robustly read n bytes (unbuffered)
 */
ssize_t rio_readn(int fd, void *usrbuf, size_t n)
{
 size_t nleft = n;
 ssize_t nread;
 char *bufp = usrbuf;

 while (nleft > 0) {
 if ((nread = read(fd, bufp, nleft)) < 0) {
 if (errno == EINTR) /* interrupted by sig
 * handler return */
 nread = 0; /* and call read() again */
 else
 return -1; /* errno set by read() */
 }
 else if (nread == 0)
 break; /* EOF */
 nleft -= nread;
 bufp += nread;
 }
 return (n - nleft); /* return >= 0 */
}
```

36

## Buffered RIO Input Functions

- Efficiently read text lines and binary data from a file partially cached in an internal memory buffer

```
#include "csapp.h"
void rio_readinitb(rio_t *rp, int fd);
ssize_t rio_readlineb(rio_t *rp, void *usrbuf, size_t maxlen);
ssize_t rio_readnb(rio_t *rp, void *usrbuf, size_t n);
Return: num. bytes read if OK, 0 on EOF, -1 on error
```

- `rio_readlineb` reads a text line of up to `maxlen` bytes from file `fd` and stores the line in `usrbuf`.
  - Especially useful for reading text lines from network sockets.
- `rio_readnb` reads up to `n` bytes from file `fd`.
- Calls to `rio_readlineb` and `rio_readnb` can be interleaved arbitrarily on the same descriptor.
  - Warning: Don't interleave with calls to `rio_readn`

37

## rio\_t (in csapp.h)

- Simple accounting structure

```
#define RIO_BUFSIZE 8192
typedef struct {
 int rio_fd; /* descriptor for this internal buffer */
 int rio_cnt; /* unread bytes in the internal buffer */
 char *rio_bufptr; /* next unread byte in internal buffer */
 char rio_buf[RIO_BUFSIZE]; /* the internal buffer */
} rio_t;
```

38

## RIO Example

- Copying the lines of a text file from standard input to standard output.

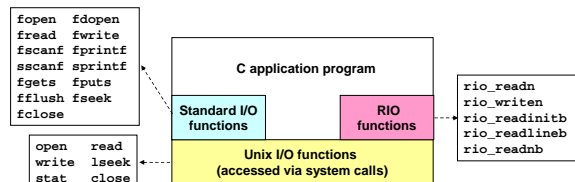
```
#include "csapp.h"
int main(int argc, char **argv)
{
 int n;
 rio_t rio;
 char buf[MAXLINE];

 Rio_readinitb(&rio, STDIN_FILENO);
 while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0)
 Rio_writen(STDOUT_FILENO, buf, n);
 exit(0);
}
```

39

## Unix I/O vs. Standard I/O vs. RIO

- Standard I/O and RIO are implemented using low-level Unix I/O.



- Which ones should you use in your programs?

40

## Pros and Cons of Unix I/O

- Pros**
  - Unix I/O is the most general and lowest overhead form of I/O
    - All other I/O packages are implemented using Unix I/O functions
  - Unix I/O provides functions for accessing file metadata
  - Sometimes the ability to use `dup` and `dup2` is indispensable
    - E.g. communicating with child processes
- Cons**
  - Dealing with short counts is tricky and error prone
  - Efficient reading of text lines requires some form of buffering, also tricky and error prone

41

## Pros and Cons of Standard I/O

- Pros:**
  - Buffering increases efficiency by decreasing the number of `read` and `write` system calls
  - Short counts are handled automatically
- Cons:**
  - Provides no function for accessing file metadata
  - Standard I/O is not appropriate for input and output on network sockets
  - There are poorly documented restrictions on streams that interact badly with restrictions on sockets

42

## Pros and Cons of Standard I/O (cont)

- **Restrictions on streams:**
  - Restriction 1: input function cannot follow output function without intervening call to `fflush`, `fseek`, `fsetpos`, or `rewind`
    - » Latter three functions all use `lseek` to change file position.
  - Restriction 2: output function cannot follow an input function with intervening call to `fseek`, `fsetpos`, or `rewind`
- **Restriction on sockets:**
  - You are not allowed to change the file position of a socket

43

## Pros and Cons of Standard I/O (cont)

- **Workaround for restriction 1:**
  - Flush stream after every output
- **Workaround for restriction 2:**
  - Open two streams on the same descriptor, one for reading and one for writing:

```
FILE *fpin, *fpout;
fpin = fdopen(sockfd, "r");
fpout = fdopen(sockfd, "w");
```

- However, this requires you to close the same descriptor twice:

```
fclose(fpin);
fclose(fpout);
```

44

## Choosing I/O Functions

- **General rule: Use the highest-level I/O functions you can**
  - Many C programmers are able to do all of their work using the standard I/O functions
- **When to use standard I/O?**
  - When working with disk or terminal files
  - When using `fork/dup` to interact with child processes
- **When to use raw Unix I/O**
  - When you need to fetch file metadata
  - In rare cases when you need absolute highest performance
- **When to use RIO?**
  - When you are reading and writing network sockets or pipes
  - Never use standard I/O or raw Unix I/O on sockets or pipes
    - » this will be very important for the proxy lab

45

## For Further Information

- W. Richard Stevens, *Advanced Programming in the Unix Environment*, Addison Wesley, 1993
  - Somewhat dated, but still useful
- Kernighan and Pike, *The UNIX Programming Environment*
- Bovet and Cesati, *Understanding the Linux Kernel*

46