# System-Level I/O

**Computer Systems**

Troels Henriksen

**Based on slides by:**

Randal E. Bryant and David R. O'Hallaron

# Today

- **Unix I/O**
- Metadata, sharing, and redirection
- RIO (robust I/O) package
- Standard I/O
- Closing remarks

# Unix I/O Overview

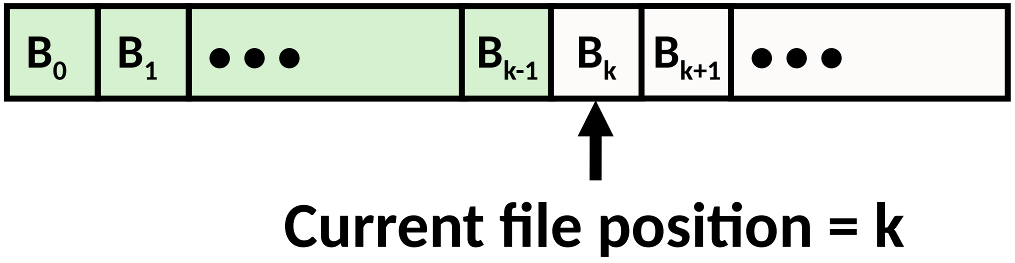
- **A Linux *file* is a sequence of *m* bytes:**
  - $B_0, B_1, .... , B_k , .... , B_{m-1}$

- **Cool fact: All I/O devices are represented as files:**
  - **`/dev/tty`** (the current terminal)
  - **`/dev/sda2`** (a disk partition)
  - **`/dev/tty2`** (some other terminal)

- **Even the kernel is represented as a file:**
  - **`/boot/vmlinuz-3.13.0-55-generic`** (kernel image)
  - **`/proc`** (process information)
  - **`/sys`** (kernel data structures)

# Unix I/O Overview

- **Elegant mapping of files to devices allows kernel to export simple interface called *Unix I/O*:**
  - Opening and closing files
    - **`open()`** and **`close()`**
  - Reading and writing a file
    - **`read()`** and **`write()`**
  - Changing the ***current file position*** (seek)
    - indicates next offset into file to read or write
    - **`lseek()`**
    - Not all files support seeking (e.g. pipes, sockets)



# File Types

- **Each file has a *type* indicating its role in the system**
  - *Regular file:* Contains arbitrary data
  - *Directory*:  Index for a related group of files
  - *Socket*: For communicating with a process on another machine

- **Other file types beyond our core scope**
  - *Named pipes (FIFOs)*
  - *Symbolic links*
  - *Character and block devices*

# Regular Files

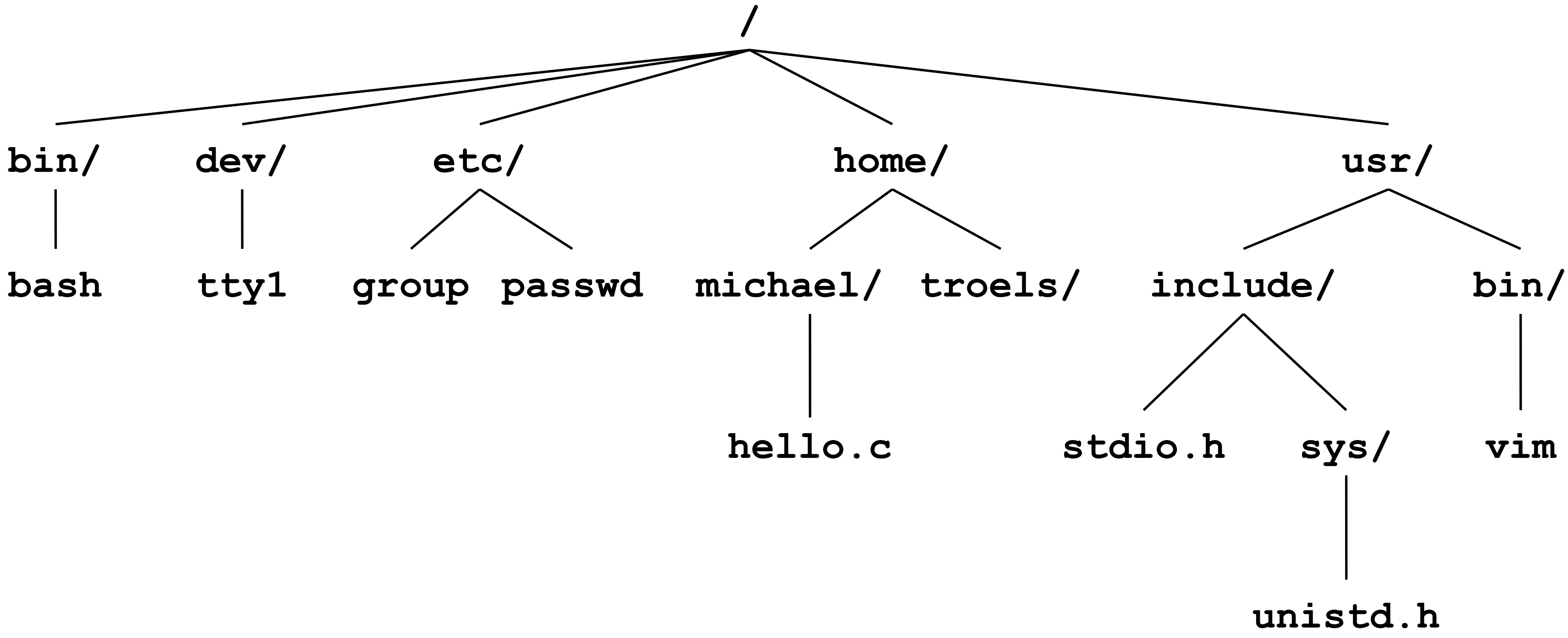
- **A regular file contains arbitrary data**
- **Applications often distinguish between *text files* and *binary files***
  - Text files are regular files with only ASCII or Unicode characters
  - Binary files are everything else
    - e.g., object files, JPEG images, PDFs
  - Kernel doesn't know the difference!
  - Question of *interpretation* of the byte contents
- **Text file is sequence of *text lines***
  - Text line is sequence of chars terminated by *newline char* (`\n`)
    - Newline is `0xA`, same as ASCII line feed character (LF)
- **End of line (EOL) indicators in other systems**
  - Linux and macOS: `\n` (`0xA`)
  - Windows and Internet protocols: `\r\n` (`0xD 0xA`)
    - Carriage return (CR) followed by line feed (LF)
  - C64, Pre-OS X Mac OS, Lisp Machines: `\r`

# Directories

- **Directory consists of an array of *links***
  - Each link maps a *filename* to an *inode*
  - An inode contains metadata and disk location of file.
  - Multiple links may point to same file!
- **Each directory contains at least two entries**
  - `.` (dot) is  a link to itself
  - `..` (dot dot) is a link to *the parent directory* in the *directory hierarchy* (next slide)
- **Commands for manipulating directories**
  - `mkdir`: create empty directory
  - `ls`: view directory contents
  - `rmdir`: delete empty directory

# Directory Hierarchy

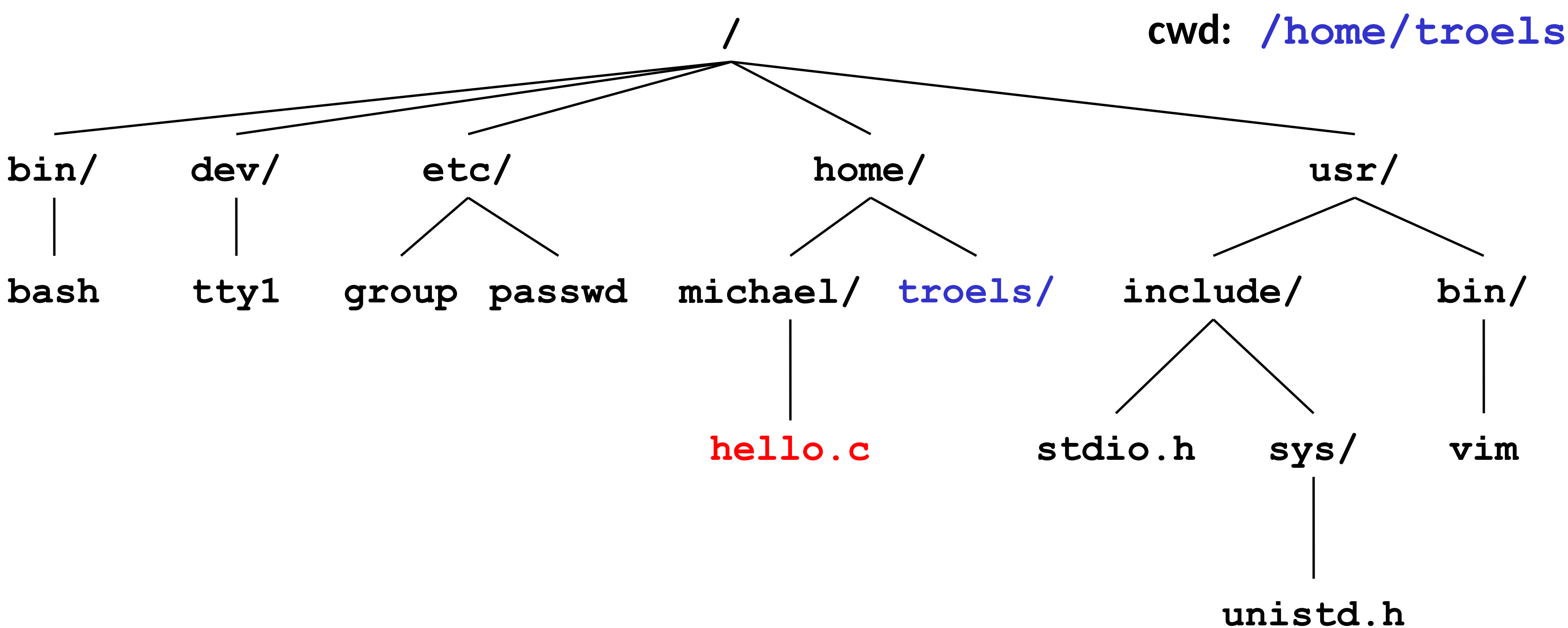
- **All files are organized as a hierarchy anchored by root directory named / (slash)**




- **Kernel maintains *current working directory (cwd)* for each process**
  - Modified using the `cd` command or `chdir()` syscall
  - **Question:** is `cd` a normal program run by the shell with `fork`+`exec`?

# Pathnames

- **Locations of files in the hierarchy denoted by *pathnames***
  - *Absolute pathname* starts with ‘/’ and denotes path from root
    - `/home/troels/hello.c`
  - *Relative pathname* denotes path from current working directory
    - `../home/michael/hello.c`



# 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 Linux shell begins life with three open files associated with a terminal:**
  - 0: standard input (stdin)
  - 1: standard output (stdout)
  - 2: standard error (stderr)

# 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);
}
```

- Closing an already closed file is a recipe for disaster in threaded programs, because the file descriptor number may have been re-used
- Always check return codes, even for seemingly benign functions such as `close()`

# Reading 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 at least 1 byte and
   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`**
  - Return type **`ssize_t`** is signed integer
  - **`nbytes < 0`** indicates that an error occurred
  - *Short counts* (**`nbytes < sizeof(buf)`** ) are possible and are not errors!

# Writing 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!

# Simple Unix I/O example

- Copying stdin to stdout, one byte at a time
- Slow (examples later)

```
#include "csapp.h"

int main(void)
{
    char c;

    while(Read(STDIN_FILENO, &c, 1) != 0)
        Write(STDOUT_FILENO, &c, 1);
    exit(0);
}
```

# On Short Counts

- **Short counts often occurs in these situations:**
  - Encountering (end-of-file) EOF on reads
  - Reading text lines from a terminal
  - Reading and writing network sockets

- **Short counts rarely occurs in these situations:**
  - Reading from disk files (except for EOF)
    - …but may happen for huge reads, depending on file system.
  - Writing to disk files
    - …similarly.

- **Best practice is to always allow for short counts.**
- **Distinctly Unix: https://www.dreamsongs.com/RiseOfWorseIsBetter.html**

# Today

- Unix I/O
- **Metadata, sharing, and redirection**
- RIO (robust I/O) package
- Standard I/O
- Closing remarks

# File Metadata

- *Metadata* is data about data, in this case file data
- Per-file metadata 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 */
};
```

# Example of Accessing File Metadata

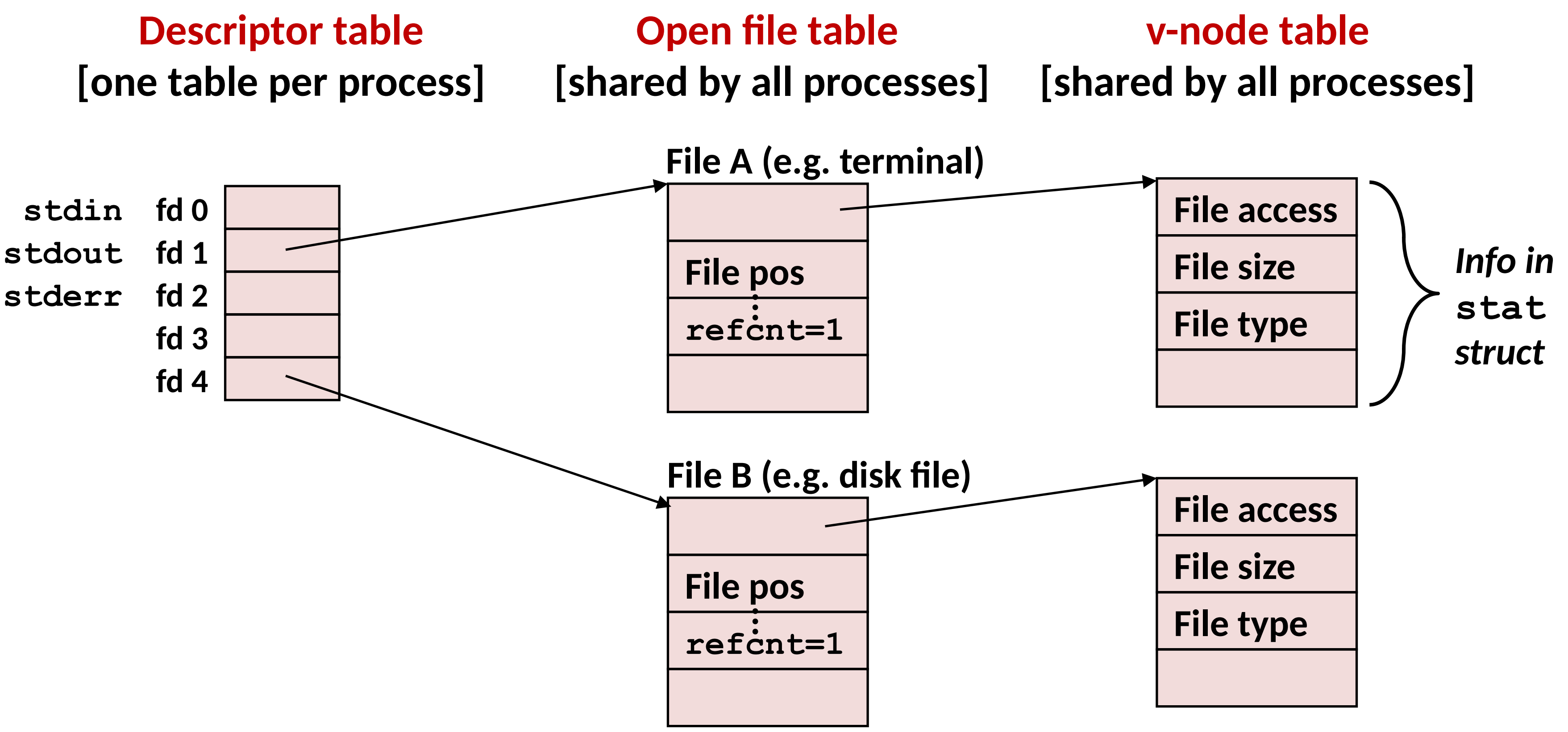
```
int main (int argc, char **argv)
{
    struct stat stat;
    char *type, *readok;

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

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

statcheck.c

```
$ ./statcheck statcheck.c
type: regular, read: yes
$ chmod 000 statcheck.c
$ ./statcheck statcheck.c
type: regular, read: no
$ ./statcheck ..
type: directory, read: yes
```

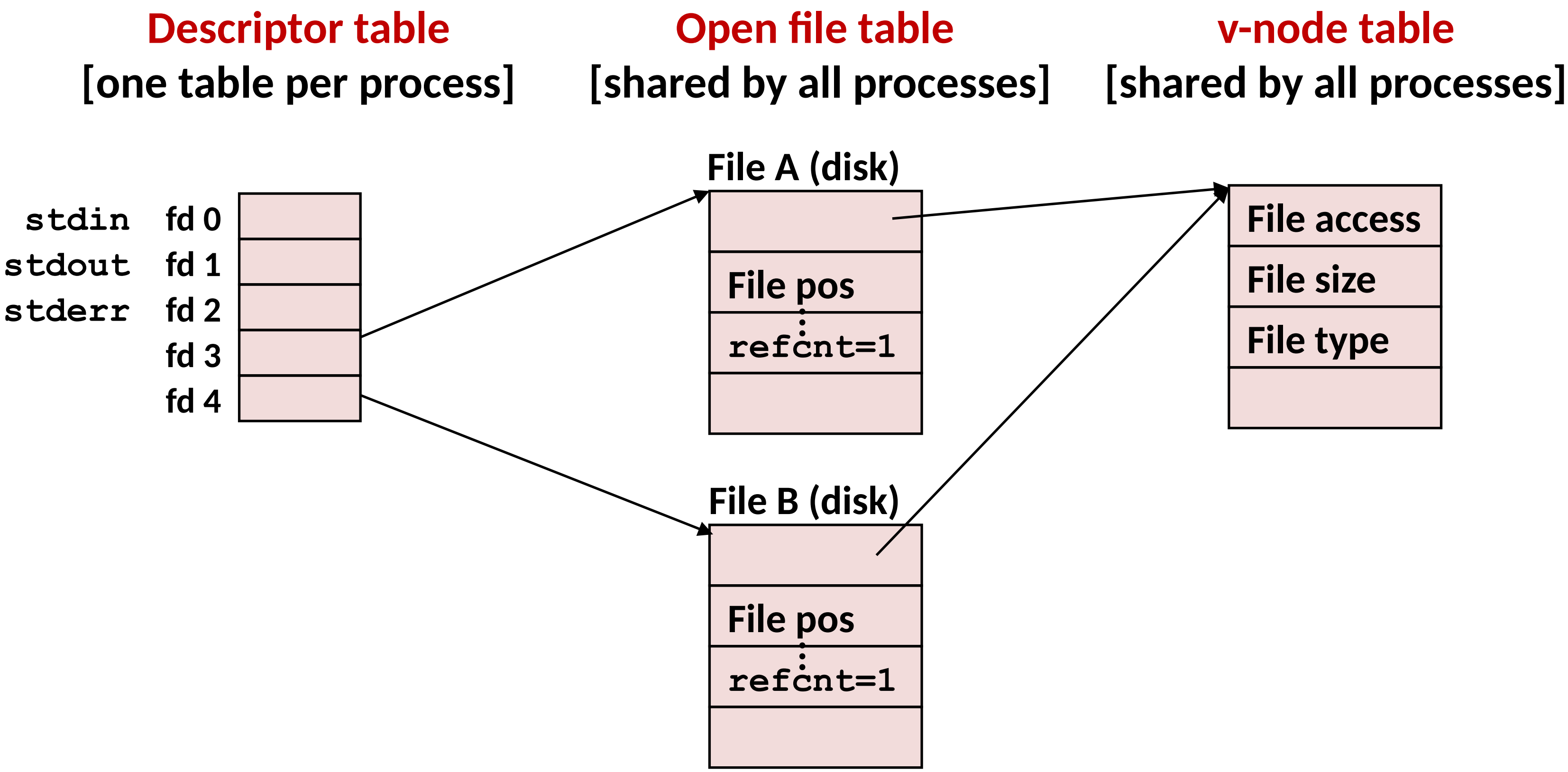
# How the Unix Kernel Represents Open Files

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



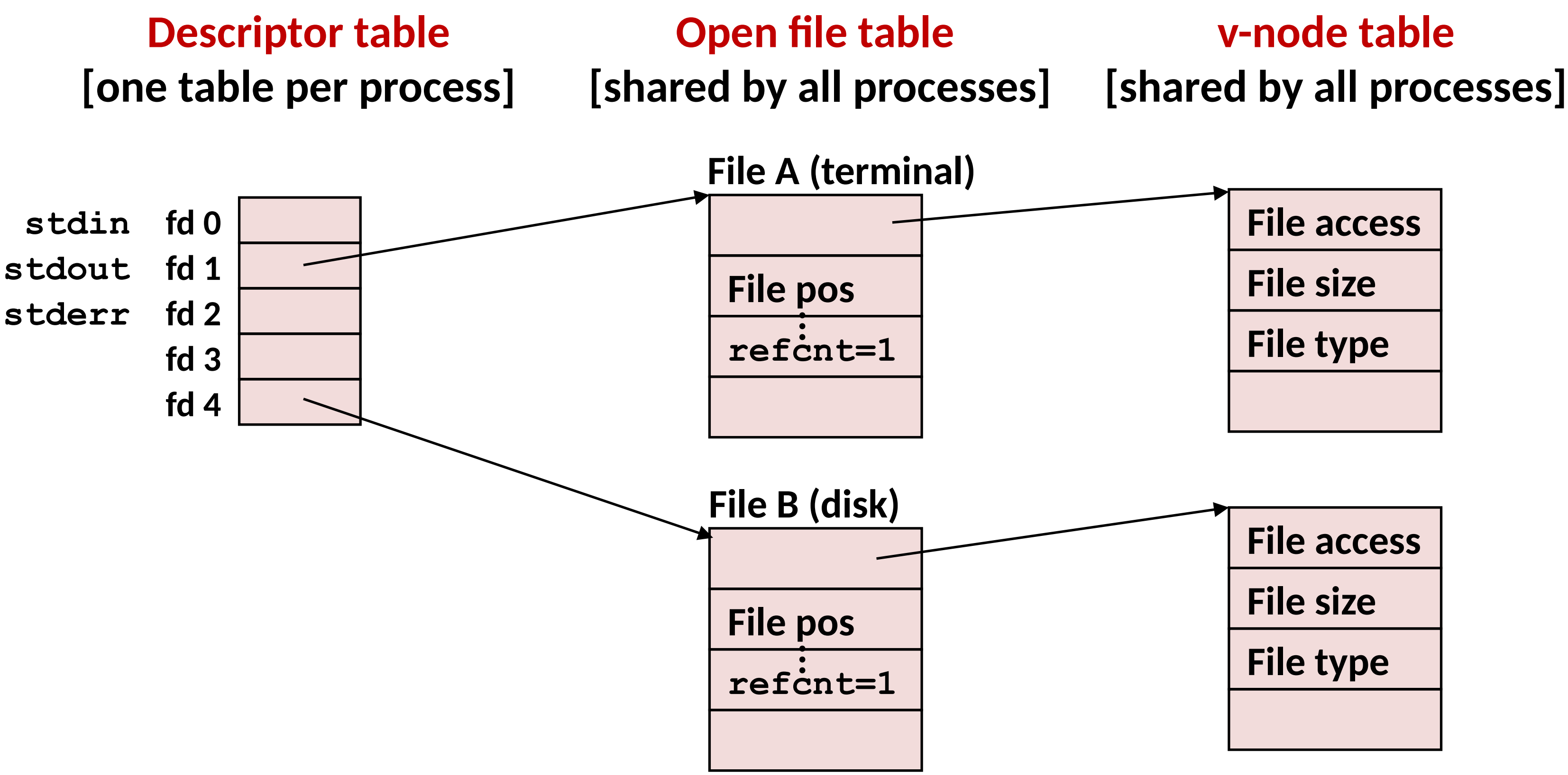
# File Sharing

- **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
  - Not necessarily a good idea - but the kernel won't stop you



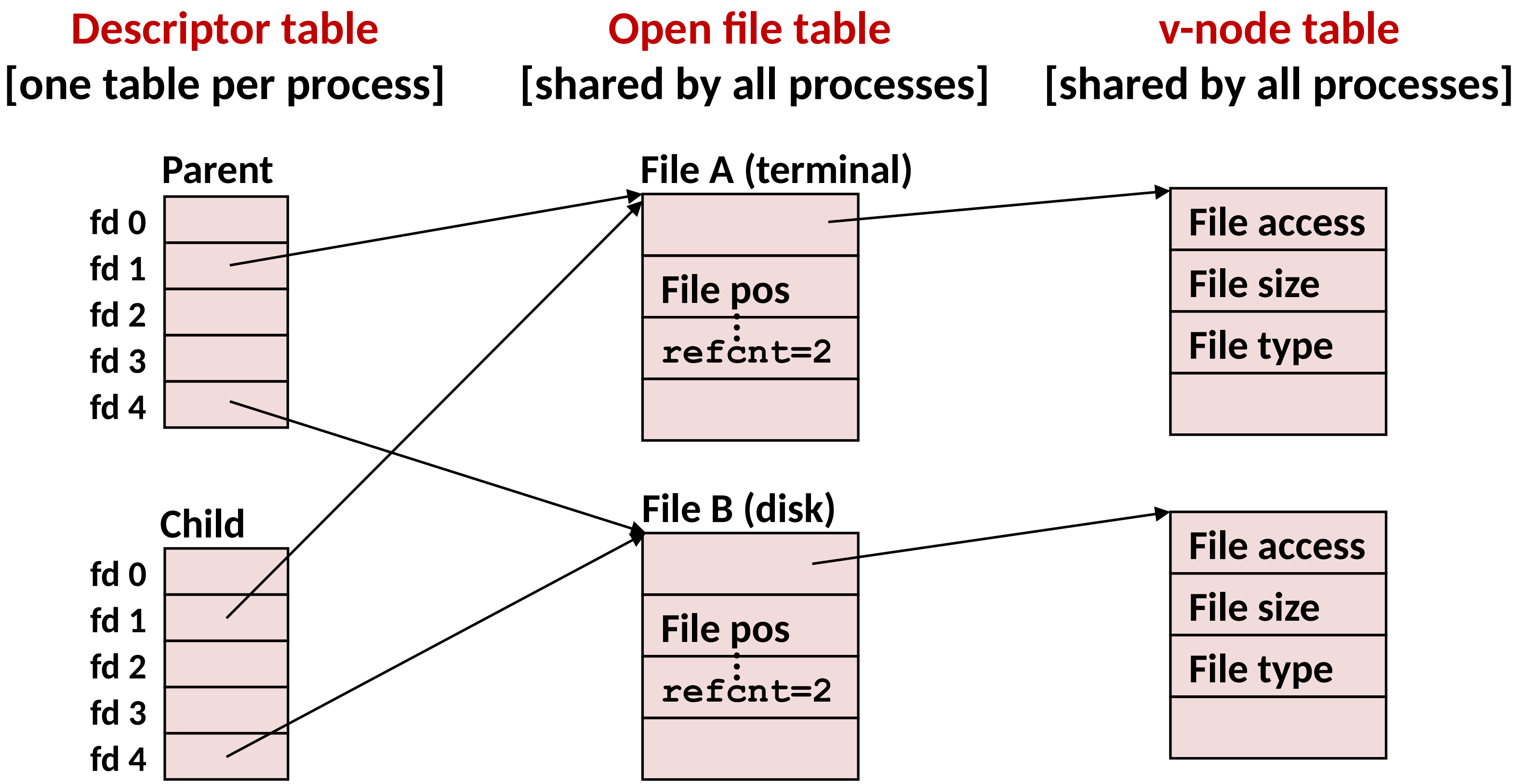
# How Processes Share Files: `fork`

- A child process inherits its parent's open files
  - Note: situation unchanged by **`exec`** functions (use **`fcntl`** to change)
- *Before* **`fork`** call:



# How Processes Share Files: `fork`

- **A child process inherits its parent's open files**
- ***After* `fork`:**
  - Child's table same as parent's, and +1 to each refcnt
  - If one process does a read, the read position will change for both!



# I/O Redirection

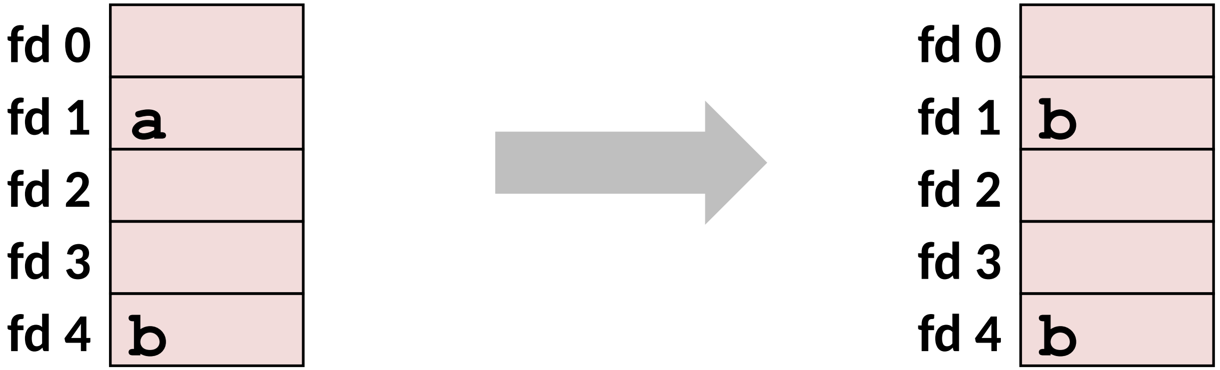
- **Question: How does a shell implement I/O redirection?**

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

- **Answer: By calling the `dup2(oldfd, newfd)` function**
  - Copies (per-process) descriptor table entry **`oldfd`** to entry **`newfd`**

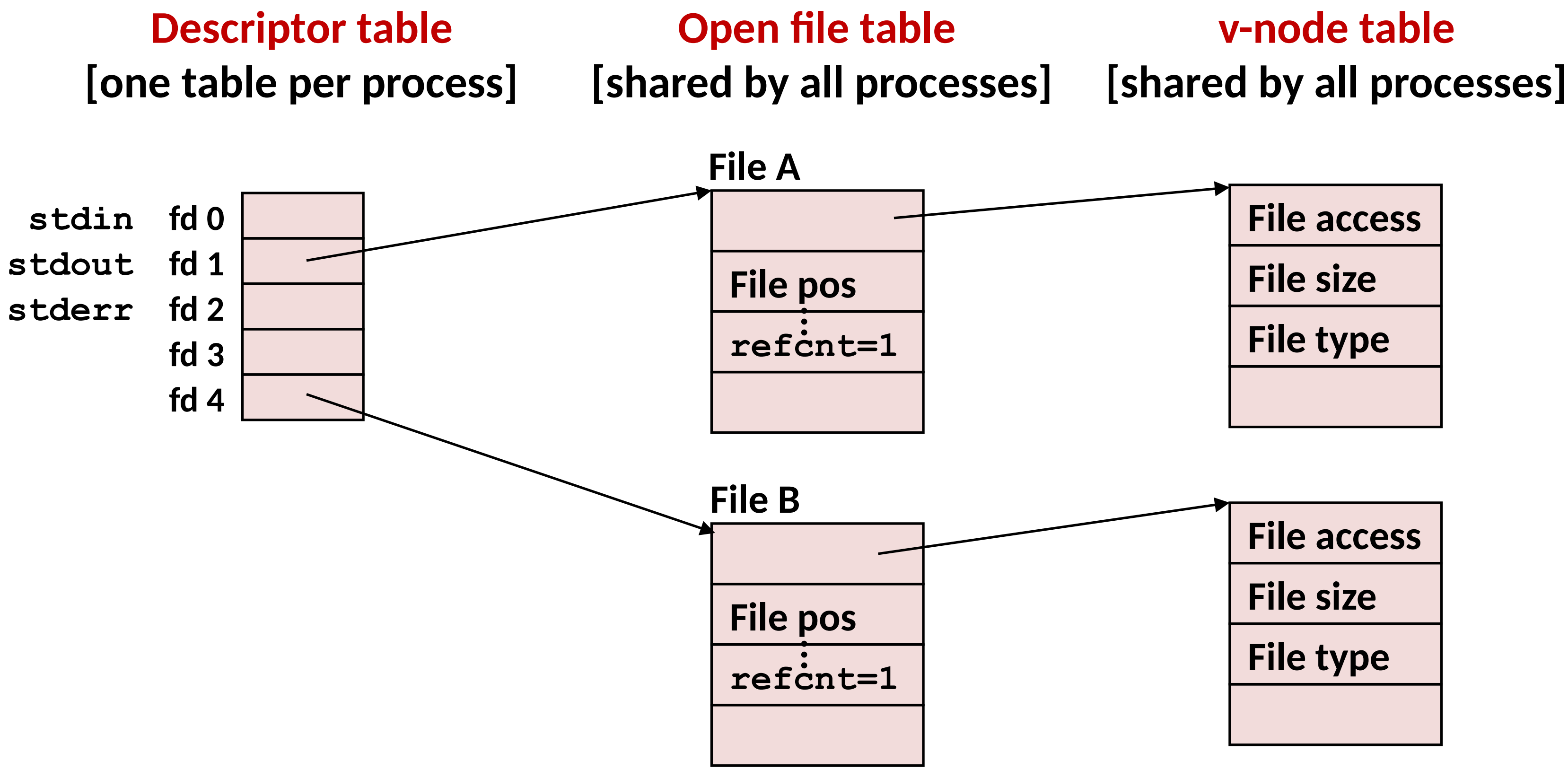
**Descriptor table**
***before*** **`dup2(4,1)`**

**Descriptor table**
***after*** **`dup2(4,1)`**



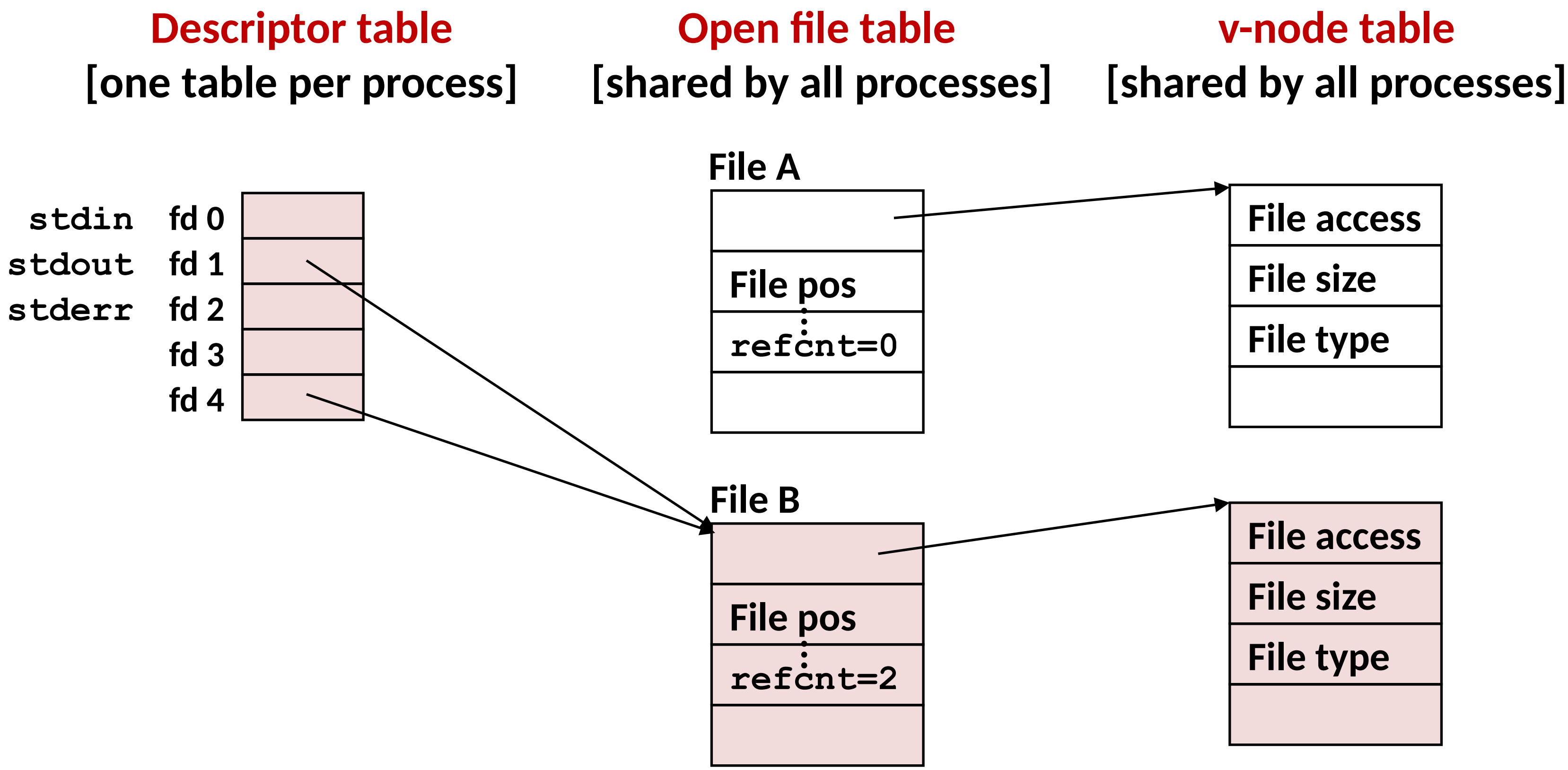
# I/O Redirection Example

- **Step #1: open file to which stdout should be redirected**
  - Happens in child executing shell code, before **`exec`**



# I/O Redirection Example (cont.)

- **Step #2: call `dup2(4,1)`**
  - cause fd=1 (stdout) to refer to disk file pointed at by fd=4



# Today

- Unix I/O
- Metadata, sharing, and redirection
- **RIO (robust I/O) package**
- Standard I/O
- Closing remarks

# The RIO Package

- **RIO is a set of wrappers that provide efficient and robust I/O in apps, 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 text lines and binary data
    - **`rio_readlineb`** and **`rio_readnb`**
    - Buffered RIO routines are thread-safe and can be interleaved arbitrarily on the same descriptor

- Part of **`csapp.c/csapp.h`**

# 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
  - Only use it when you know how many bytes to read
- **`rio_writen`** never returns a short count
- Calls to **`rio_readn`** and **`rio_writen`** can be interleaved arbitrarily on the same descriptor

# 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 */
}
```

csapp.c

# 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
- Stopping conditions
  - **`maxlen`** bytes read
  - EOF encountered
  - Newline (‘**`\n`**’) encountered

# Buffered RIO Input Functions (cont)

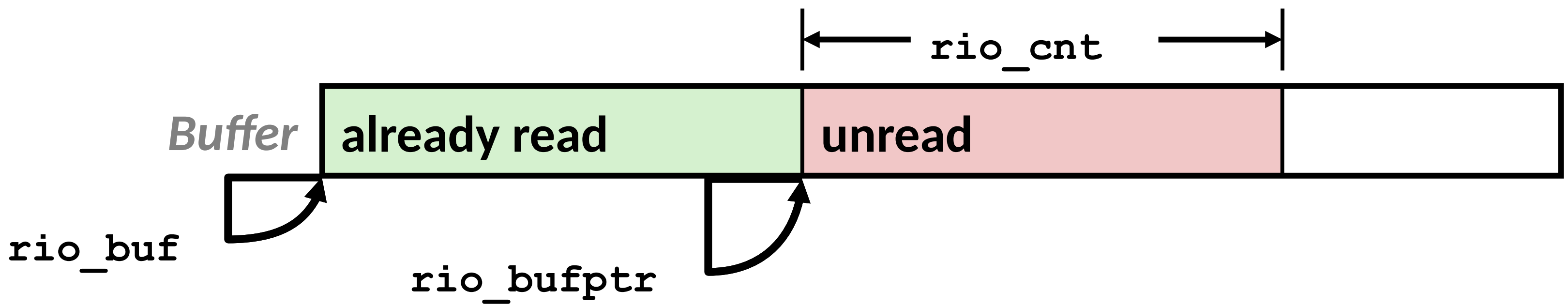
```
#include "csapp.h"

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**

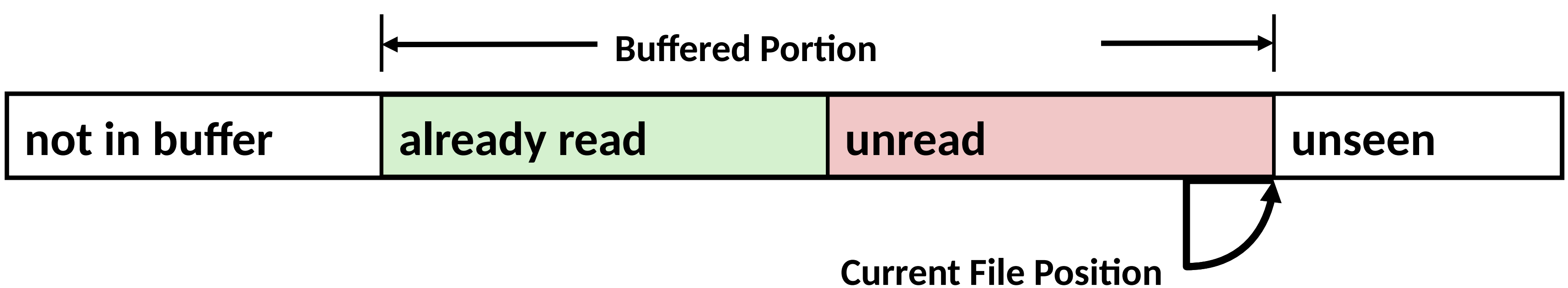
- **`rio_readnb`** reads up to **`n`** bytes from file **`fd`**
- Stopping conditions
  - **`maxlen`** bytes read
  - EOF encountered
- Calls to **`rio_readlineb`** and **`rio_readnb`** can be interleaved arbitrarily on the same descriptor
  - Warning: Don't interleave with calls to **`rio_readn`**

# Buffered I/O: Implementation

- For reading from file
- File has associated buffer to hold bytes that have been read from file but not yet read by user code

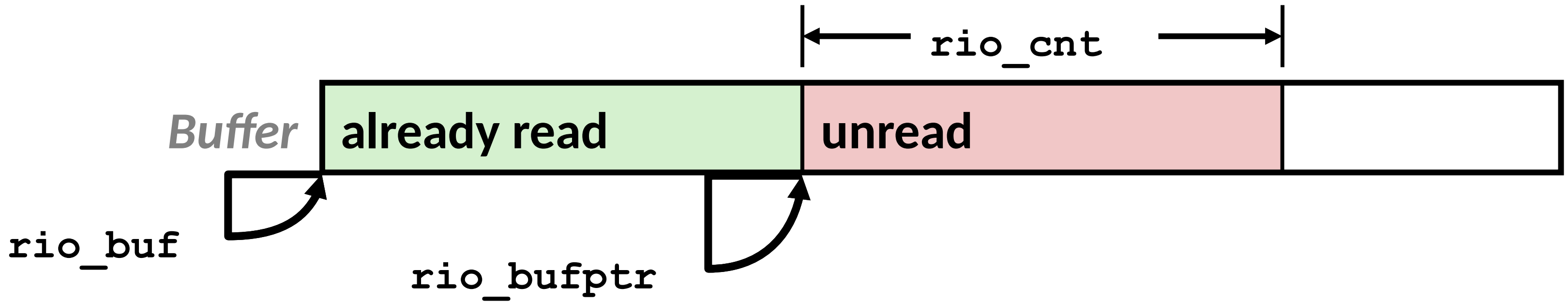



- Layered on Unix file:



# Buffered I/O: Declaration

- All information contained in `struct`




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

# Buffered I/O: Read some bytes

```
static ssize_t rio_read(rio_t *rp, char *usrbuf, size_t n)
{
    int cnt;

    while (rp->rio_cnt <= 0) {  /* Refill if buf is empty */
        rp->rio_cnt = read(rp->rio_fd, rp->rio_buf,
                           sizeof(rp->rio_buf));
        if (rp->rio_cnt < 0) {
            if (errno != EINTR)   /* If not interrupted */
                return -1;        /* Then it must be a real error */
        }
        else if (rp->rio_cnt == 0)  /* EOF */
            return 0;
        else
            rp->rio_bufptr = rp->rio_buf; /* Reset buffer ptr */
    }

    /* Copy min(n, rp->rio_cnt) bytes from internal buf to user buf */
    cnt = n;
    if (rp->rio_cnt < n)
        cnt = rp->rio_cnt;
    memcpy(usrbuf, rp->rio_bufptr, cnt);
    rp->rio_bufptr += cnt;
    rp->rio_cnt -= cnt;
    return cnt;
}
```

csapp.c

# Buffered I/O: Read *n* bytes robustly

```
ssize_t rio_readnb(rio_t *rp, void *usrbuf, size_t n)
{
    size_t nleft = n;
    ssize_t nread;
    char *bufp = usrbuf;

    while (nleft > 0) {
        if ((nread = rio_read(rp, bufp, nleft)) < 0)
            return -1;          /* errno set by read() */
        else if (nread == 0)
            break;              /* EOF */
        nleft -= nread;
        bufp += nread;
    }
    return (n - nleft);         /* return >= 0 */
}
```

csapp.c

# 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);
}
```

cpfile.c

# Today

- Unix I/O
- Metadata, sharing, and redirection
- RIO (robust I/O) package
- **Standard I/O**
- Closing remarks

# Standard I/O Functions

- **The C standard library (`libc.so`) contains a collection of higher-level *standard I/O* functions**
  - Documented in 8.2 of your C book
  - Or read the manpages!

- **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`**)
  - Formatted reading and writing (**`fscanf`** and **`fprintf`**)

# Standard I/O Streams

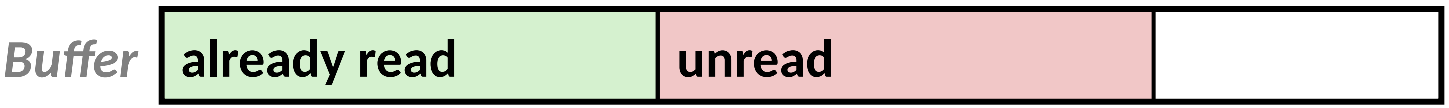
- **Standard I/O models open files as *streams***
  - Abstraction for a file descriptor and a buffer in memory

- **C programs begin life with three open streams (defined in `stdio.h`)**
  - `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");
}
```

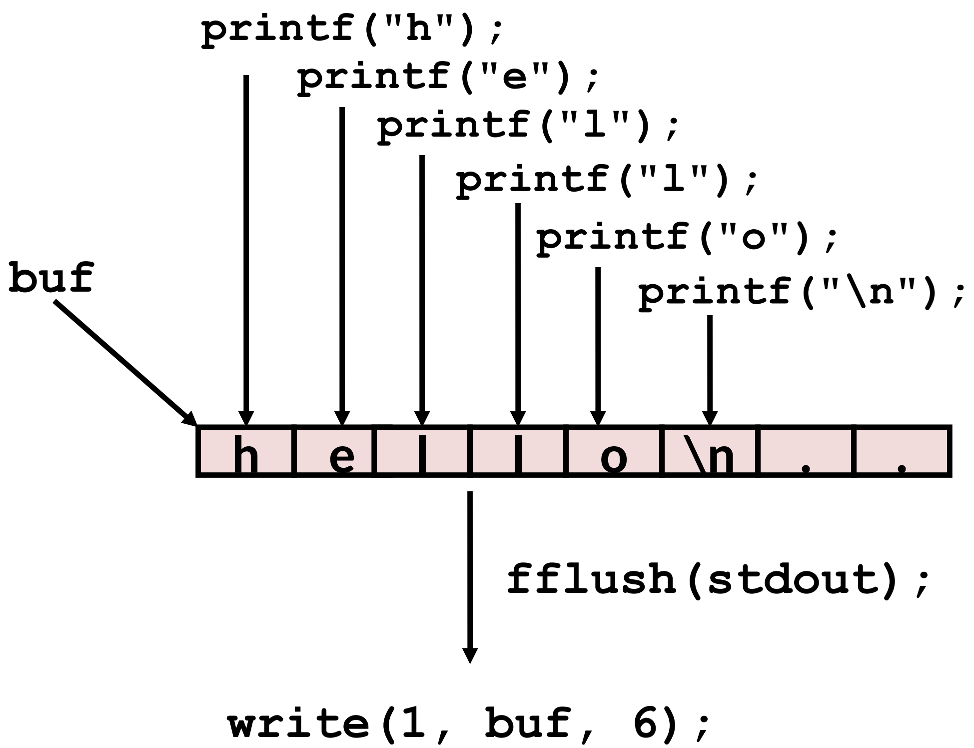
# Buffered I/O: Motivation

- **Applications often read/write one character at a time**
  - `getc, putc, ungetc`
  - `gets, fgets`
    - Read line of text one character at a time, stopping at newline
- **Implementing as Unix I/O calls expensive**
  - `read` and `write` require Unix kernel calls
    - > 10,000 clock cycles
- **Solution: Buffered read *and write***
  - Use Unix `read` to grab block of bytes
  - User input functions take one byte at a time from buffer
    - Refill buffer when empty



# Buffering in Standard I/O

- Standard I/O functions use buffered I/O




- Buffer flushed to output fd on “\n”, call to `fflush` or `exit`, or return from `main`.

# Standard I/O Buffering in Action

- You can see this buffering in action for yourself, using the always fascinating Linux `strace` program:

```
#include <stdio.h>

int main()
{
    printf("h");
    printf("e");
    printf("l");
    printf("l");
    printf("o");
    printf("\n");
    fflush(stdout);
    exit(0);
}
```

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

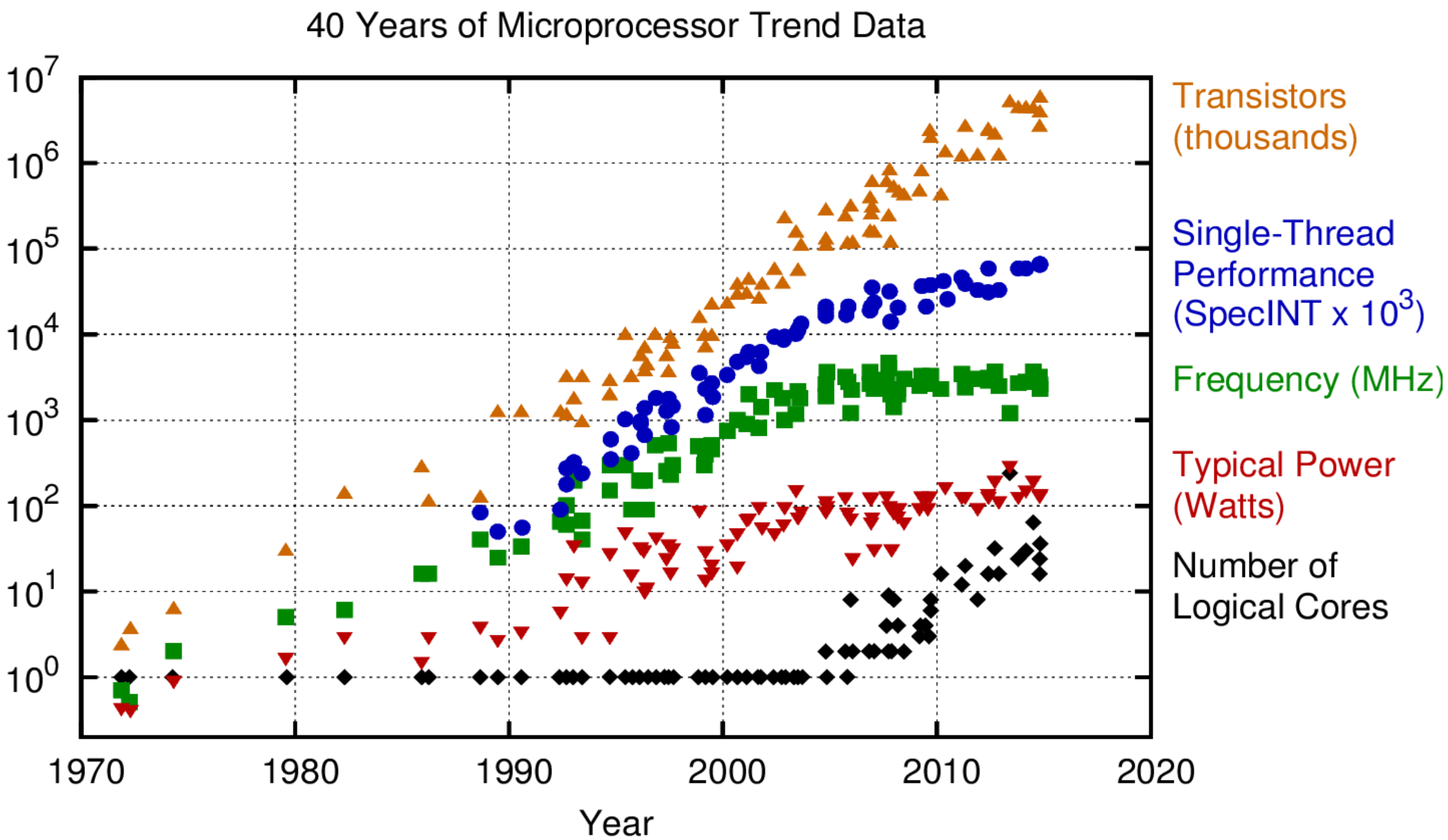
# Today

- Unix I/O
- Metadata, sharing, and redirection
- RIO (robust I/O) package
- Standard I/O
- **Closing remarks**

# Motivation for Performance

- **The *only purpose* of a computing machine is to be faster than a human.**
- **All novel programs are the result of a good idea combined with a *performance surplus*.**
  - Surplus can be generated by new/more/better machines.
  - … or by clever programming.
- **We can no longer (only) depend on engineers solving our problems by building better machines.**

# Our Situation



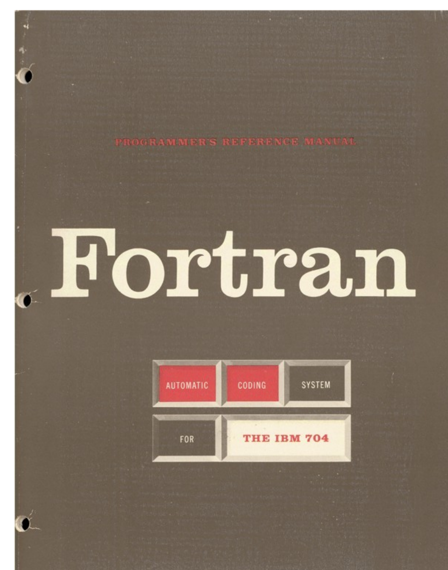

Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

# Inspirational quote

"The most amazing achievement of the computer software industry is its continuing cancellation of the steady and staggering gains made by the computer hardware industry." - Henry Petroski

# Example: high-level languages

- **The performance surplus:** computers of the 50s got faster and faster (1000s of statements per second!).
- **The good idea:** a high-level language (FORTRAN) could improve productivity, in most cases offsetting the lower performance compared to hand-coding.
- **The edge:** an *optimizing* compiler (particularly CSE) was used to narrow the gap (see *THE FORTRAN AUTOMATIC CODING SYSTEM* from 1957).

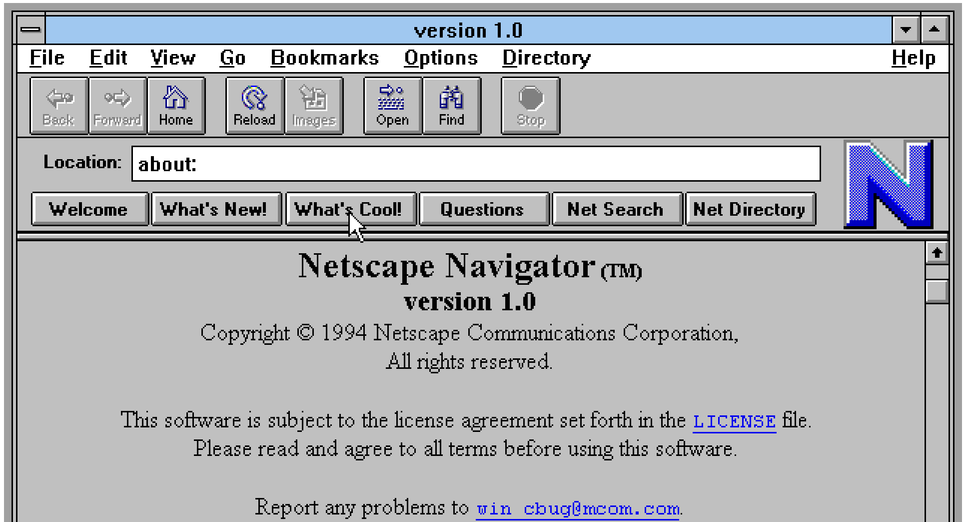


# Example: operating systems

- **The performance surplus:** increasing transistor budgets afforded non-computational circuits like MMUs.
- **The good idea:** impose a virtualisation layer that permitted running multiple applications simultaneously, safely.
- **The edge:** a clean distinction between API and implementation, making a single program runnable on machines of vastly different capabilities (IBM System/360).

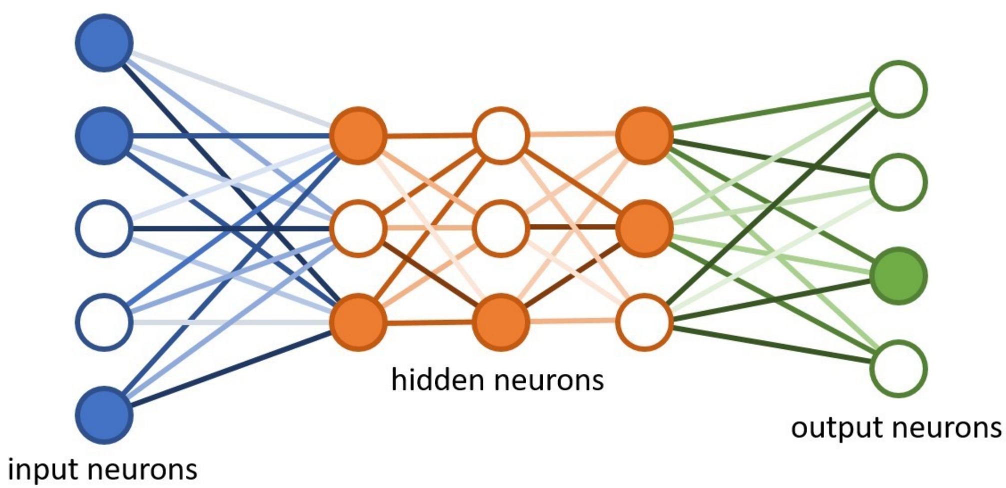
# Example: Visual web browsers

- **The performance surplus:** personal computers of the early 90s got fast enough to run sophisticated GUIs.
- **The good idea:** accessible hypertext along with *really clever programming* in the browsers – particularly to handle concurrent network requests alongside rendering.
  - **Later:** JIT compilation of Javascript (particularly Chrome's V8 in 2008) sped up web applications to create a new *software-based* performance surplus.



# Example: Deep Learning

- **The good (and old) idea:** deep sequences of simple layers of *neurons* can be trained to perform input classification, if given sufficient (huge) numbers of examples.
- **The performance surplus:** cheap *massive parallelism* in the form of generally programmable graphics processors (GPUs), funded by millions of gamers in the 90s.
- **The edge:** programming tools and techniques that made GPUs accessible to more than just graphics.




# Goodbye for now

- **This was my last lecture this year. Final advice:**
  - Check your return codes.
  - Use `assert()` liberally.
  - Use a better language than C if at all possible.
  - If such a language does not exist, *invent it*.
  - No, C++ is not that language.

- **If you like this kind of stuff, take these courses:**
  - *Programming Massively Parallel Hardware* (PMPH), in block 1.
  - *Data Parallel Programming* in block 2.
  - (Nominally master's courses, but don't let that stop you.)
  - Also, check out my research: https://futhark-lang.org

# Extra Slides

# Fun with File Descriptors (1)

```
#include "csapp.h"
int main(int argc, char *argv[])
{
    int fd1, fd2, fd3;
    char c1, c2, c3;
    char *fname = argv[1];
    fd1 = Open(fname, O_RDONLY, 0);
    fd2 = Open(fname, O_RDONLY, 0);
    fd3 = Open(fname, O_RDONLY, 0);
    Dup2(fd2, fd3);
    Read(fd1, &c1, 1);
    Read(fd2, &c2, 1);
    Read(fd3, &c3, 1);
    printf("c1 = %c, c2 = %c, c3 = %c\n", c1, c2, c3);
    return 0;
}
```

ffiles1.c

- **What would this program print for file containing “abcde”?**

# Fun with File Descriptors (2)

```
#include "csapp.h"
int main(int argc, char *argv[])
{
    int fd1;
    int s = getpid() & 0x1;
    char c1, c2;
    char *fname = argv[1];
    fd1 = Open(fname, O_RDONLY, 0);
    Read(fd1, &c1, 1);
    if (fork()) { /* Parent */
        sleep(s);
        Read(fd1, &c2, 1);
        printf("Parent: c1 = %c, c2 = %c\n", c1, c2);
    } else { /* Child */
        sleep(1-s);
        Read(fd1, &c2, 1);
        printf("Child: c1 = %c, c2 = %c\n", c1, c2);
    }
    return 0;
}
```

ffiles2.c

- **What would this program print for file containing “abcde”?**

# Fun with File Descriptors (3)

```
#include "csapp.h"
int main(int argc, char *argv[])
{
    int fd1, fd2, fd3;
    char *fname = argv[1];
    fd1 = Open(fname, O_CREAT|O_TRUNC|O_RDWR, S_IRUSR|S_IWUSR);
    Write(fd1, "pqrs", 4);
    fd3 = Open(fname, O_APPEND|O_WRONLY, 0);
    Write(fd3, "jklmn", 5);
    fd2 = dup(fd1);  /* Allocates descriptor */
    Write(fd2, "wxyz", 4);
    Write(fd3, "ef", 2);
    return 0;
}
```

ffiles3.c

- **What would be the contents of the resulting file?**

# Accessing Directories

- **Only recommended operation on a directory: read its entries**
  - `dirent` structure contains information about a directory entry
  - DIR structure contains information about directory while stepping through its entries

```
#include <sys/types.h>
#include <dirent.h>

{
  DIR *directory;
  struct dirent *de;
  ...
  if (!(directory = opendir(dir_name)))
      error("Failed to open directory");
  ...
  while (0 != (de = readdir(directory))) {
      printf("Found file: %s\n", de->d_name);
  }
  ...
  closedir(directory);
}
```