System-level I/O

Alan L. Cox
alc@rice.edu

Some slides adapted from CMU 15.213 slides

Objectives

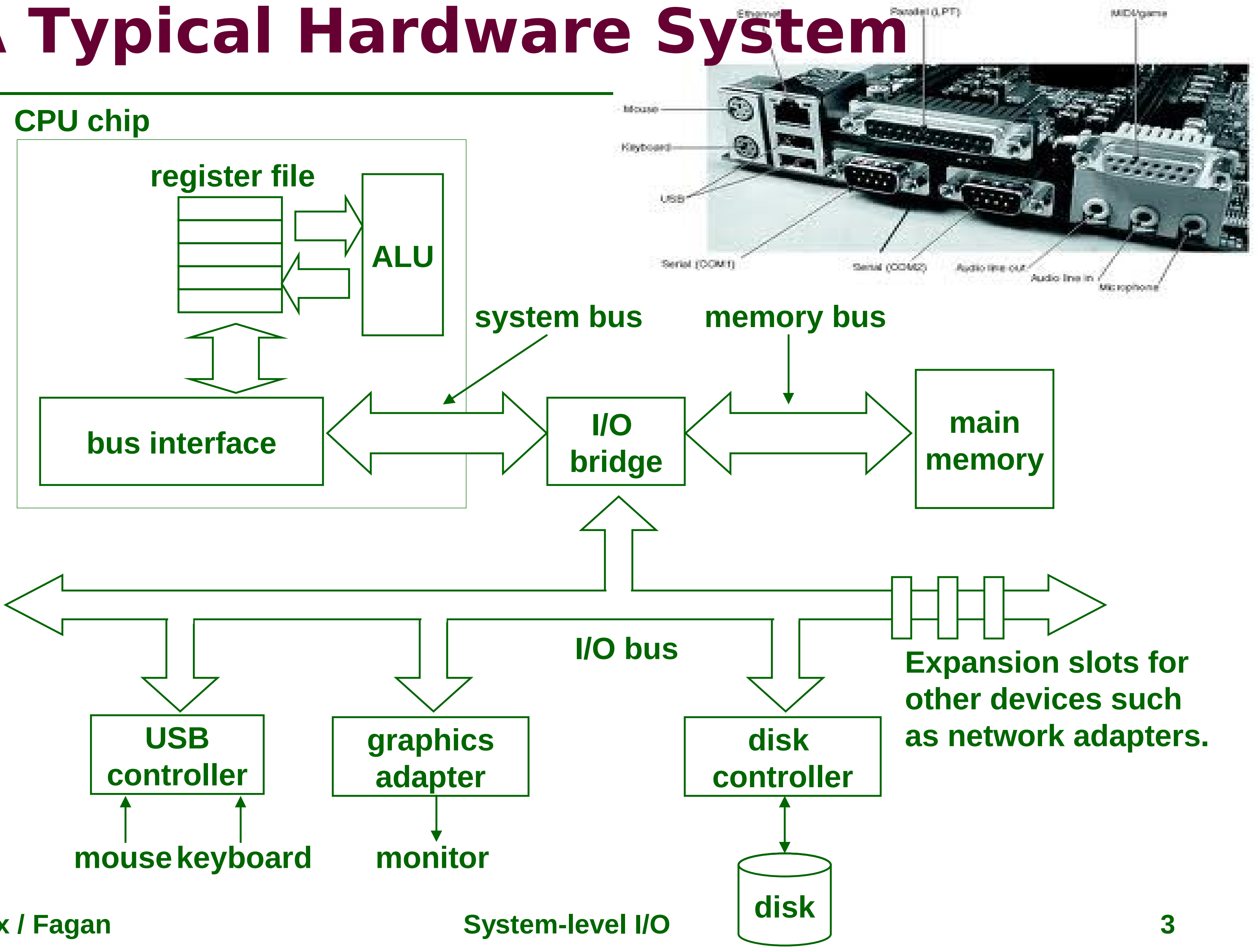
Appreciate the ingenuity of UNIX I/O model

Be able to choose the right I/O interfaces for the task

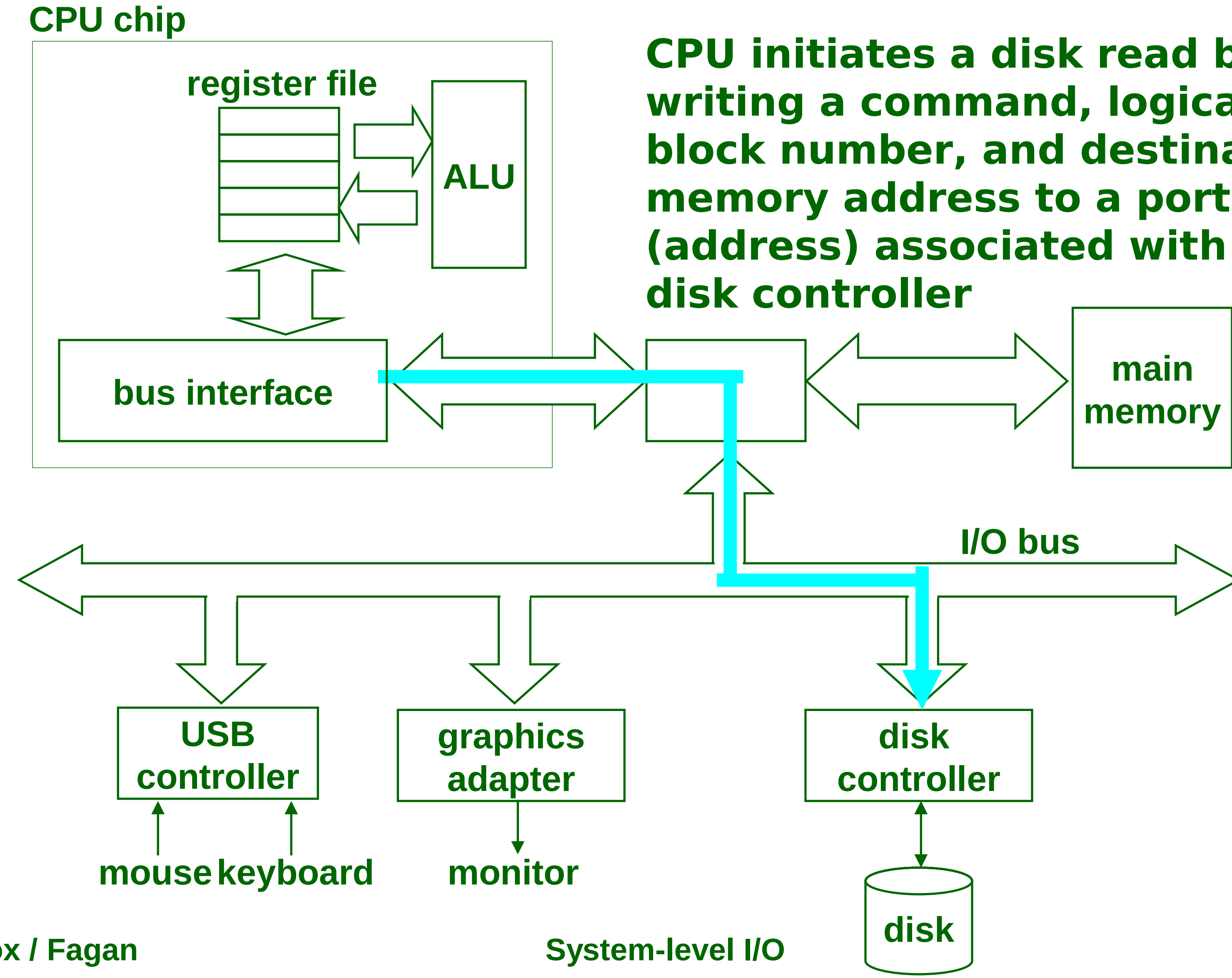
Be able to use I/O interfaces robustly and efficiently

Be able to perform I/O redirection and file sharing between processes

A Typical Hardware System



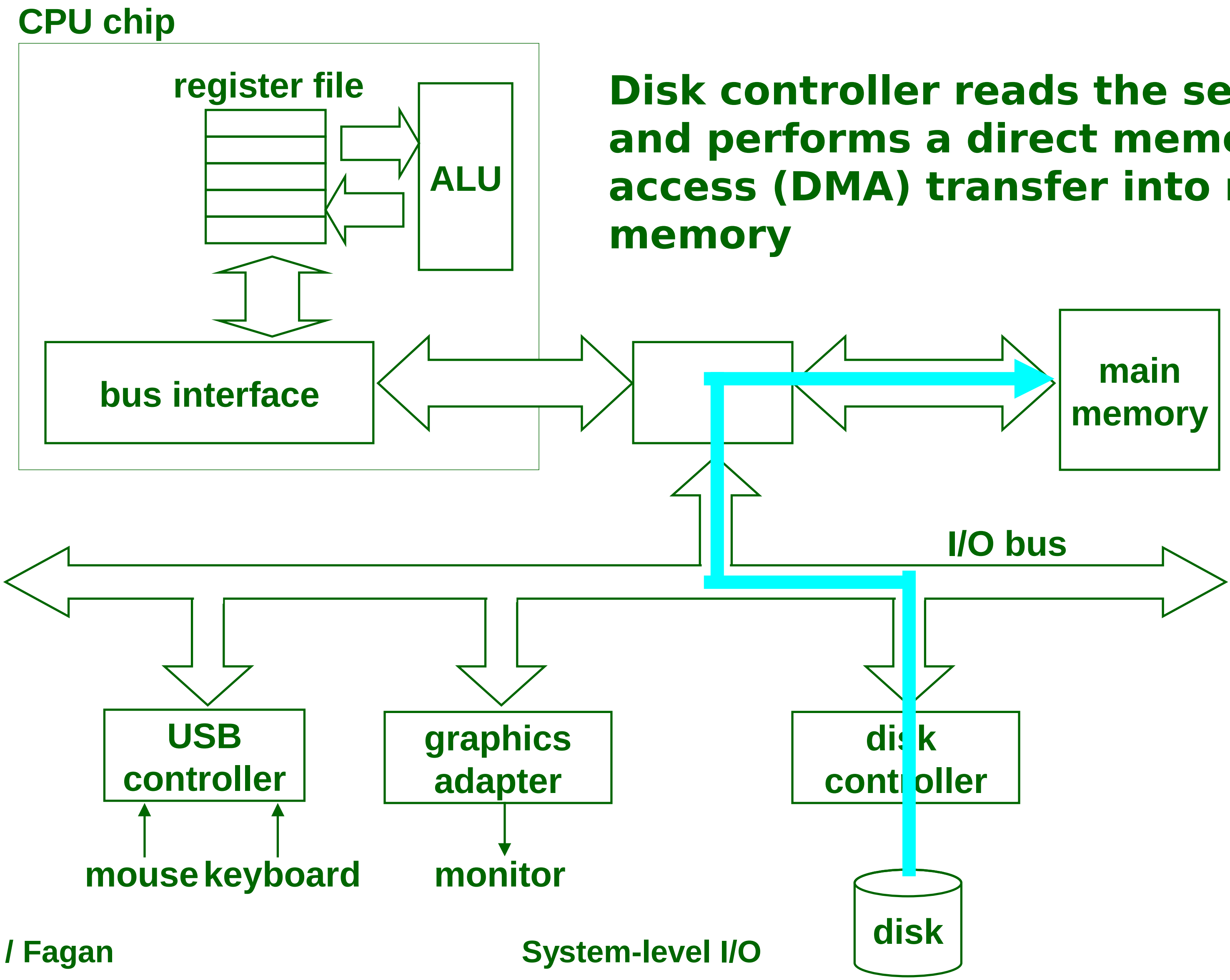
Reading a Disk Sector: Step 1



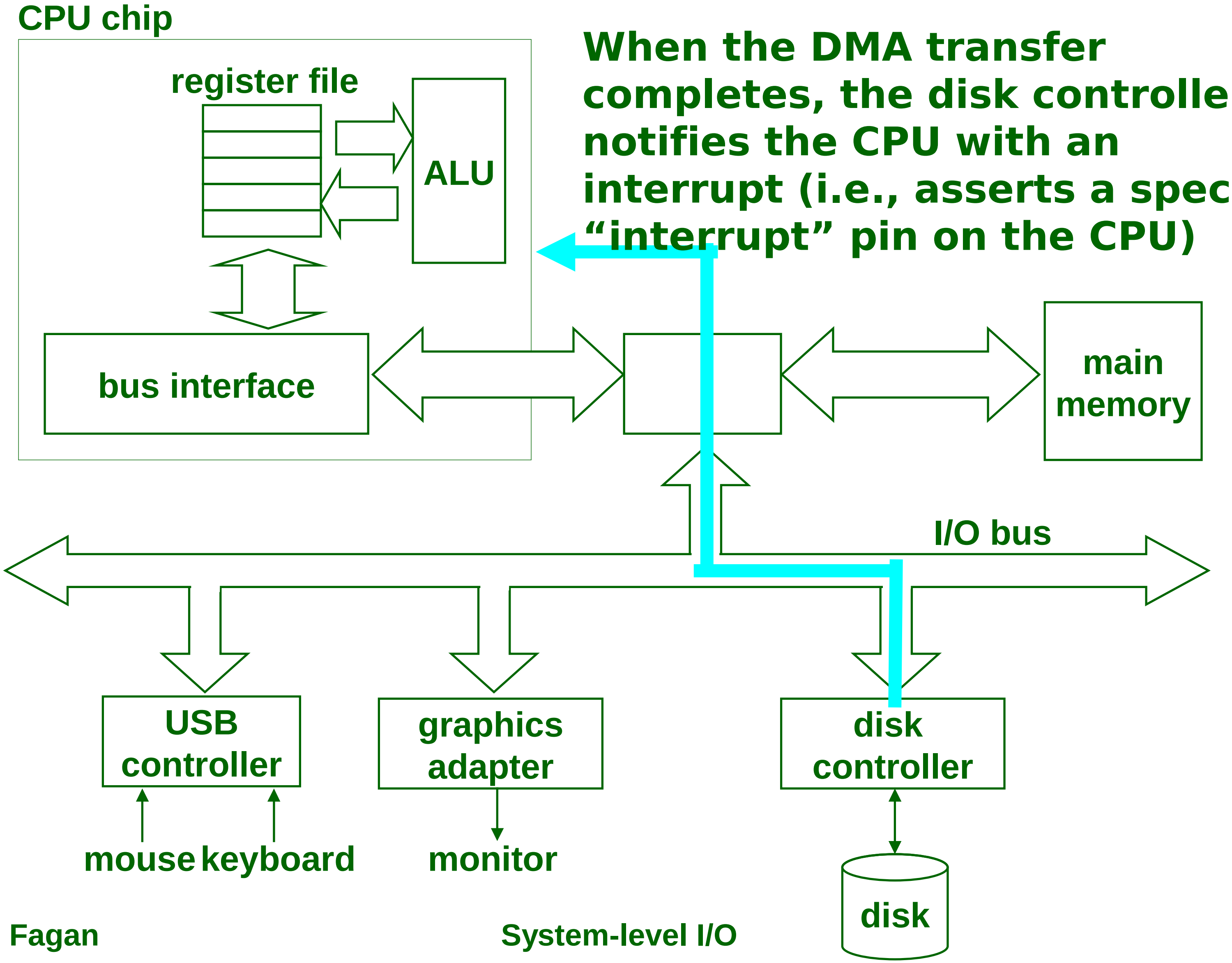
CPU initiates a disk read by writing a command, logical block number, and destination memory address to a port (address) associated with the disk controller

Reading a Disk Sector: Step 2

Disk controller reads the sector and performs a direct memory access (DMA) transfer into main memory



Reading a Disk Sector: Step 3



When the DMA transfer completes, the disk controller notifies the CPU with an interrupt (i.e., asserts a special "interrupt" pin on the CPU)

Unix Files

A Unix file is a sequence of m bytes:

- $B_0, B_1, \ldots, B_k, \ldots, B_{m-1}$

All I/O devices are represented as files:

- /dev/fd/0 (stdin)
- /dev/sd1a (disk)
- /dev/tty (terminal)

Even the kernel is represented as a file:

- /dev/kmem (kernel memory image)
- /proc (kernel data structures)

Unix File Types

Regular file

- Binary or text file
- Unix does not know the difference!

Directory file

- A file that contains the names and locations of other files

Character special and block special files

- Terminals (character special) and disks (block special)

FIFO (named pipe)

- A file type used for interprocess communication

Socket

- A file type used for network communication between processes

Unix I/O

- **The mapping of files to devices allows kernel to export simple interface called Unix I/O**
- **Key Unix idea: All input and output is handled in a consistent and uniform way**
- **Basic Unix I/O operations (system calls):**
  - **Opening and closing files**
    - `open()` **and** `close()`
  - **Changing the current file position**
    - `lseek()`
  - **Reading and writing a file**
    - `read()` **and** `write()`

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 (-1 on error)

Each process created by a Unix shell begins with three open files:

- 0: standard input
- 1: standard output
- 2: standard error

Must specify mode:

- O_RDONLY
- O_WRONLY
- O_RDWR

Writable files need additional information (O_CREAT, O_TRUNC, O_APPEND) and must also specify file permissions

Closing Files

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

```
int fd;     /* file descriptor */
if (close(fd) < 0) {
   perror(“close”);
   exit(1);
}
```

- **Returns 0 on success, -1 on failure**
- **Closing an already closed file is a recipe for disaster in threaded programs (more on this later)**
  - **Moral: 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 */
ssize_t 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!**

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 */
ssize_t 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!**

Unix I/O Example

Copying standard input to standard output one byte at a time

```
#include "csapp.h"

int main(void)
{
    char c;

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

Note the use of error handling wrappers for read and write

Dealing with Short Counts

Short counts can occur in these situations:

- Encountering (end-of-file) EOF on reads
- Reading text lines from a terminal
- Reading and writing network sockets or Unix pipes

Short counts never occur in these situations:

- Reading from disk files (except for EOF)
- Writing to disk files

How should you deal with short counts in your code?

- Use the RIO (Robust I/O) package from `csapp.c`

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**

Available at:

`/clear/www/htdocs/comp321/src/csapp.c`

`/clear/www/htdocs/comp321/include/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: number of bytes transferred if OK, 0 on EOF (`rio_readn` only), -1 on error

- `rio_readn` **returns short count only 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**

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;      /* so 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 */
}
```

Implementation of `rio_writen`

```
/* rio_writen - robustly write n bytes (unbuffered) */

ssize_t rio_writen(int fd, void *usrbuf, size_t n)
{
    size_t nleft = n;
    ssize_t nwritten;
    char *bufp = usrbuf;

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

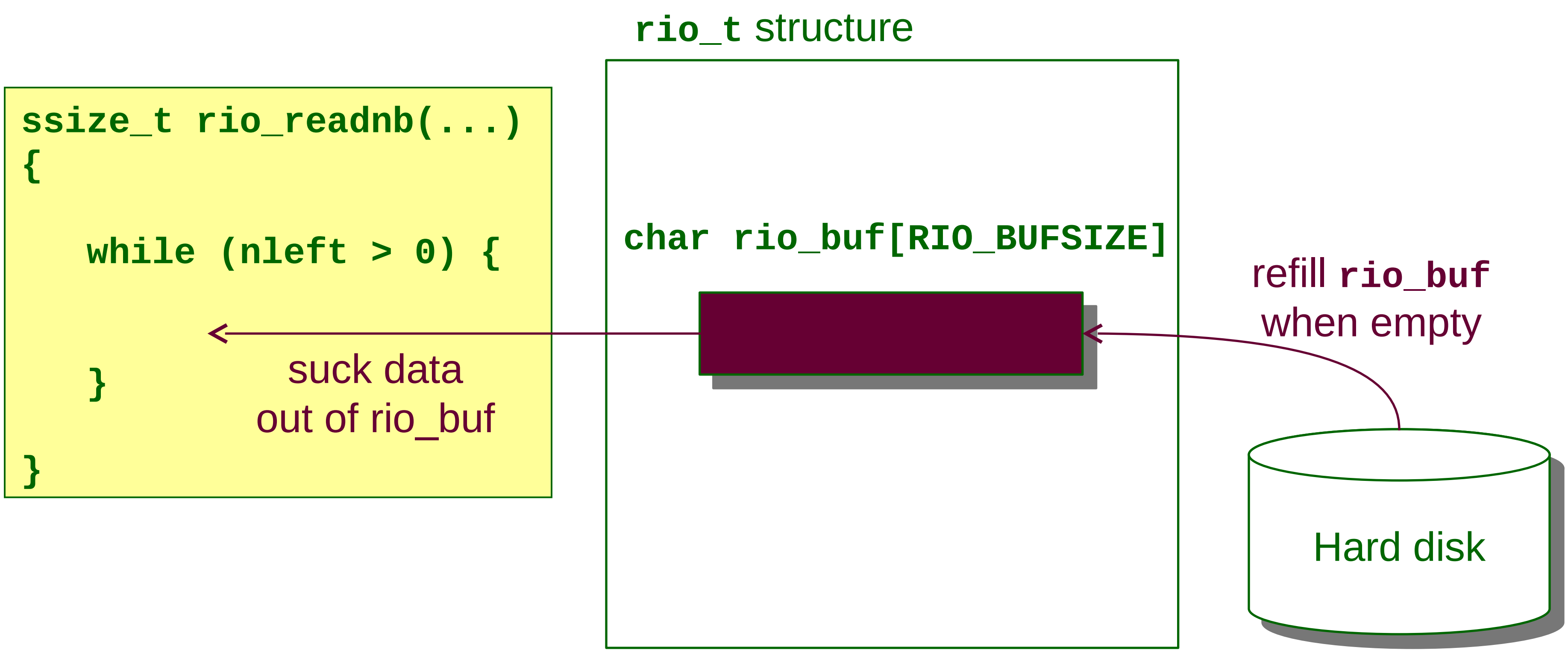
Buffered RIO Input Functions

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: number of 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`

Why is it more efficient?



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);
}
```

File Metadata

- **Metadata is data about data, in this case file data**
- **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

```
/* 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);
}
```

```
unix% ./statcheck statcheck.c
type: regular, read: yes
unix% chmod 000 statcheck.c
unix% ./statcheck statcheck.c
type: regular, read: no
```

Accessing Directories

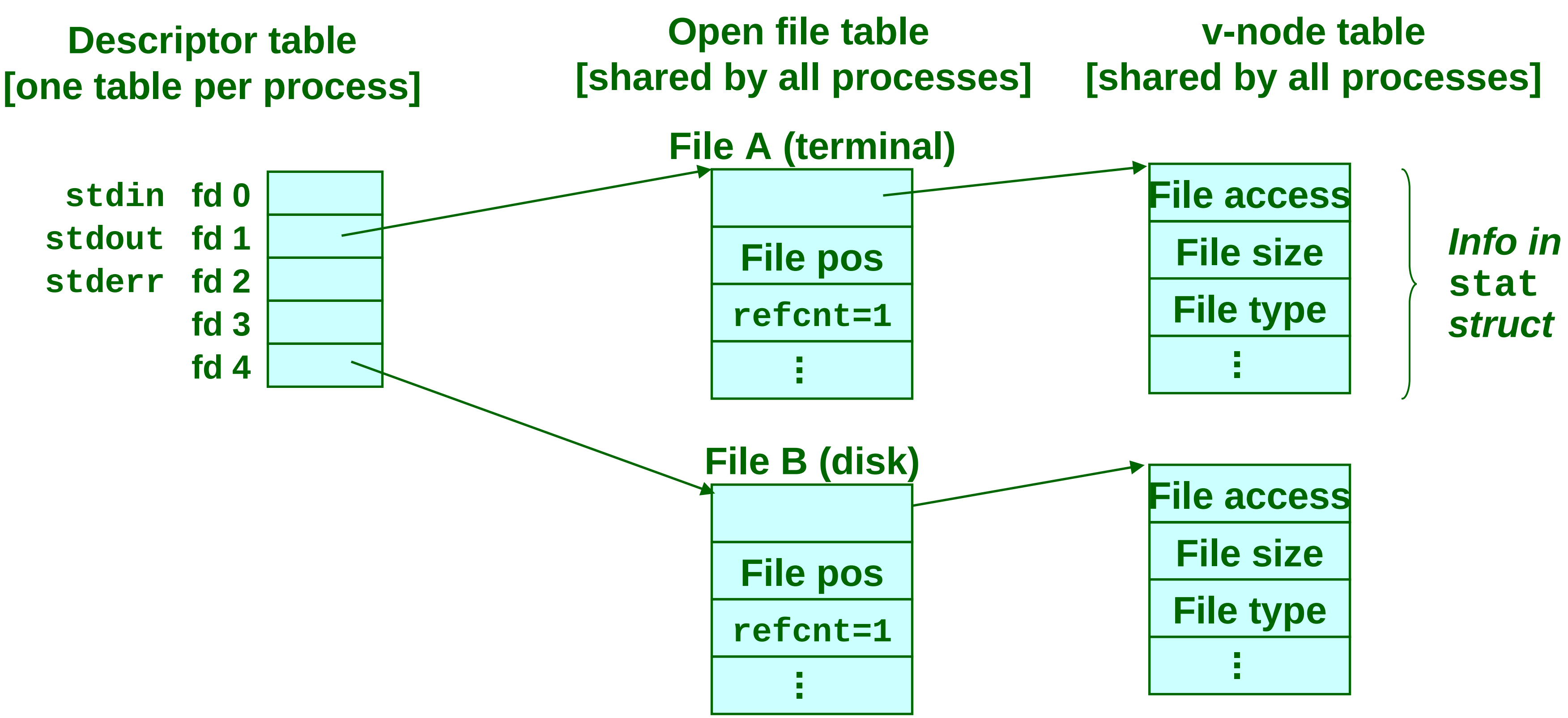
The only recommended operation on directories is to read its entries

```
#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);
}
```

How the Unix Kernel Represents Open Files

Two descriptors referencing two distinct open disk files

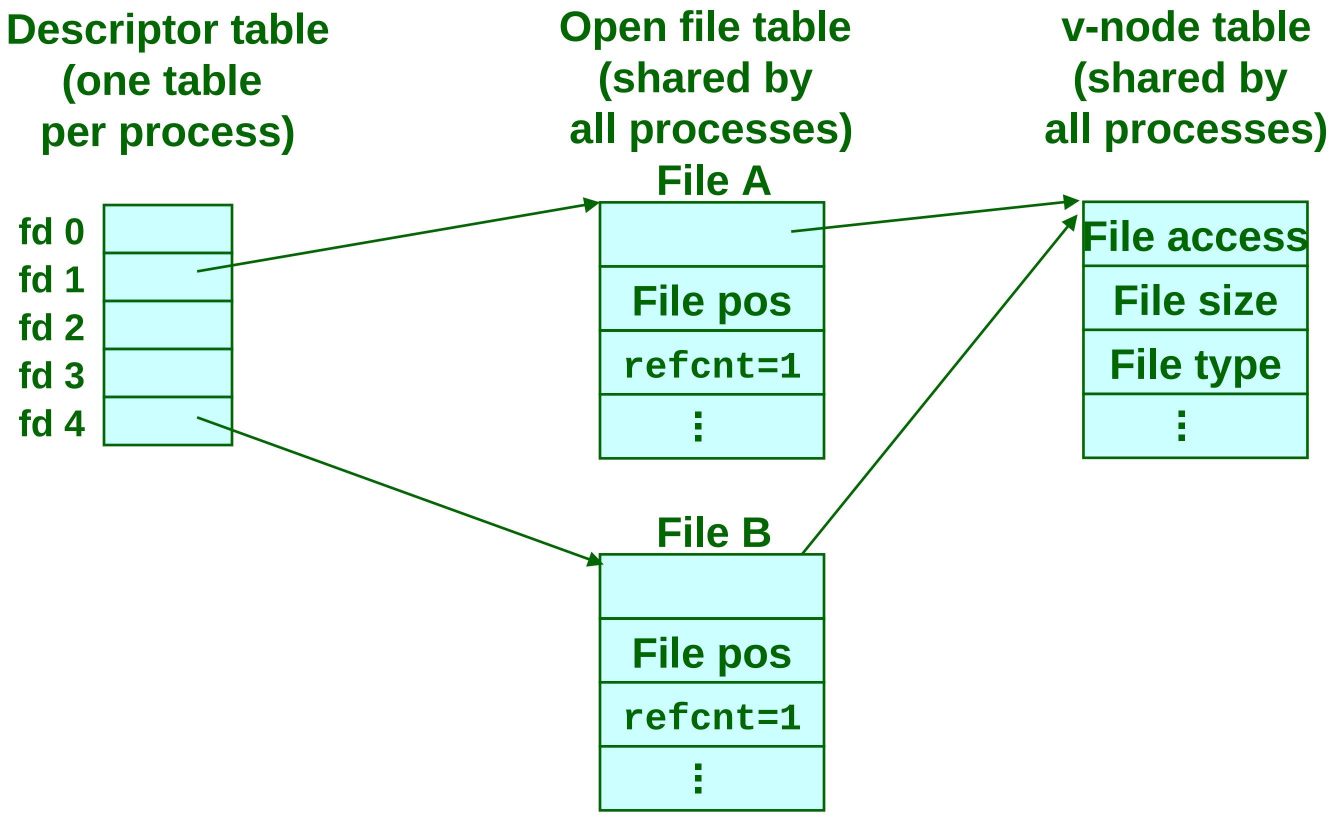
Descriptor 1 (stdout) points to terminal, and descriptor 4 points to open disk file



File Sharing

Two distinct descriptors sharing the same disk file through two distinct open file table entries

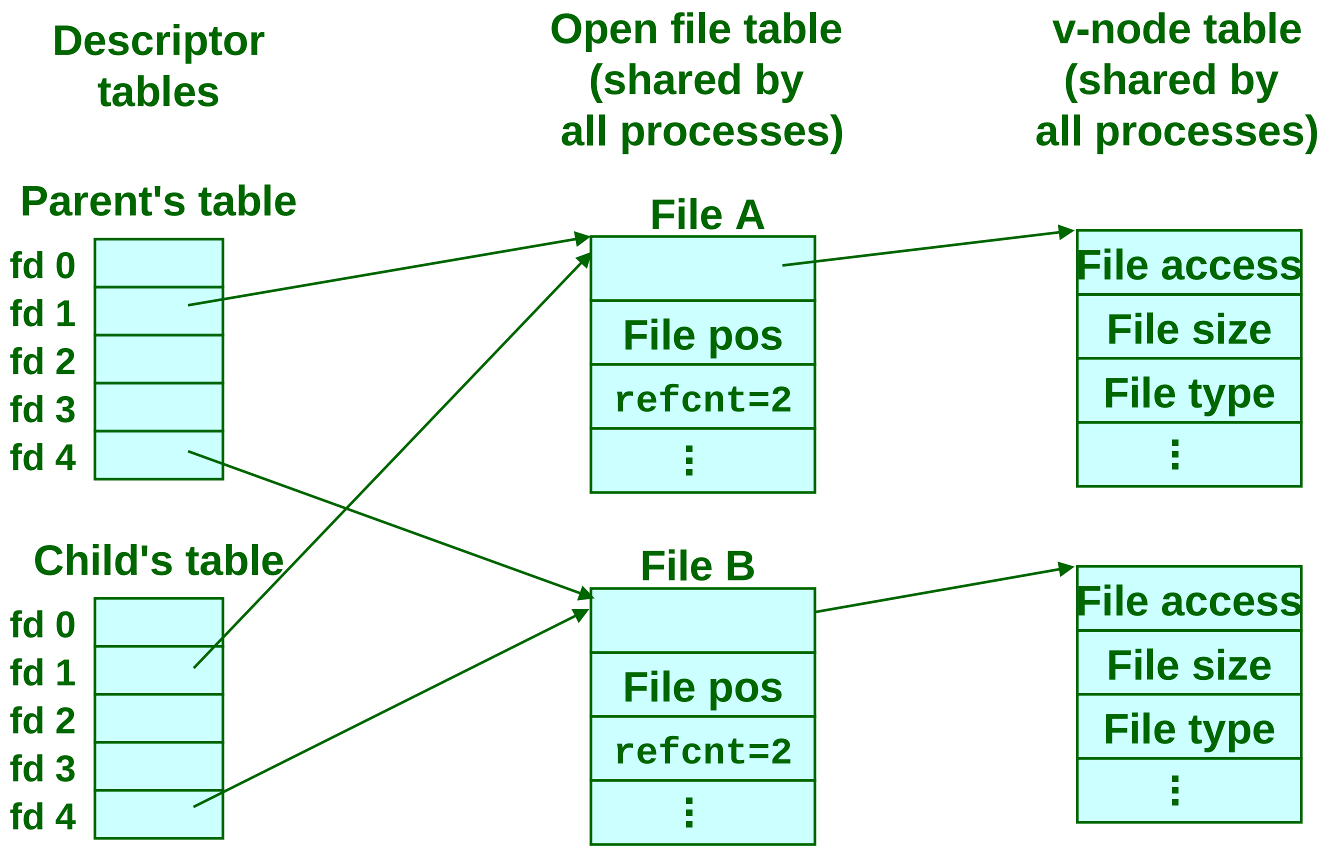
- **E.g., calling open twice with the same filename**



How Processes Share Files

A child process inherits its parent’s open files

- **Here is the situation immediately after a `fork`:**



I/O Redirection

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

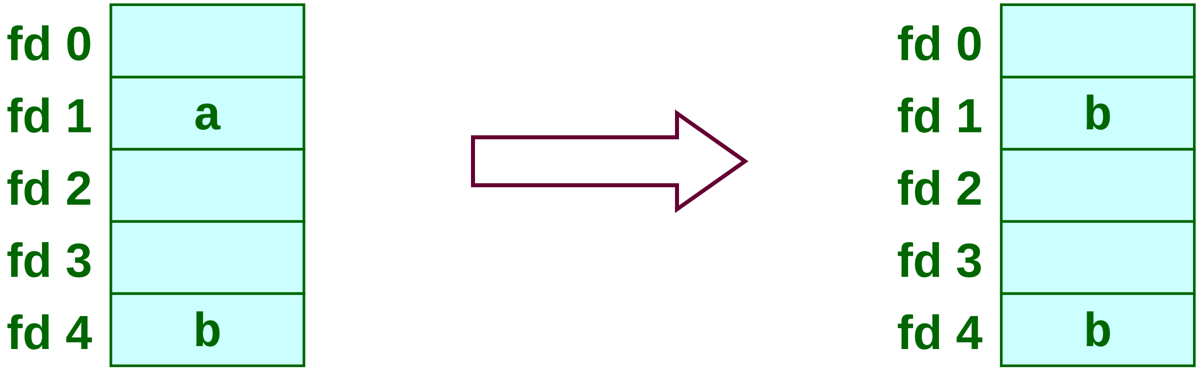
- `unix% ls > foo.txt`

Answer: By calling the `dup2(oldfd, newfd)` function

- **Copies (per-process) descriptor table entry oldfd to entry newfd**

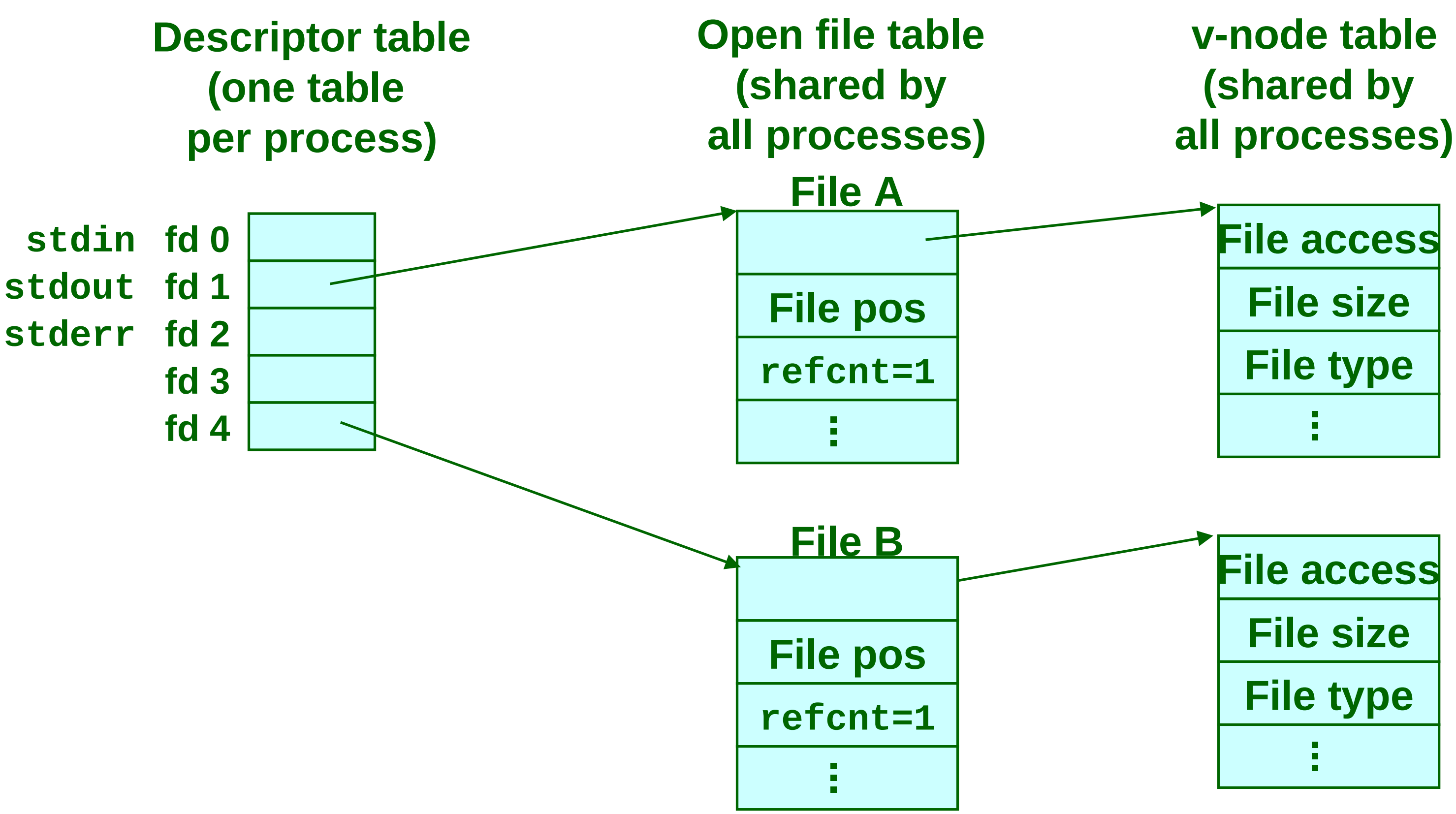
Descriptor table before `dup2(4,1)`

Descriptor table after `dup2(4,1)`



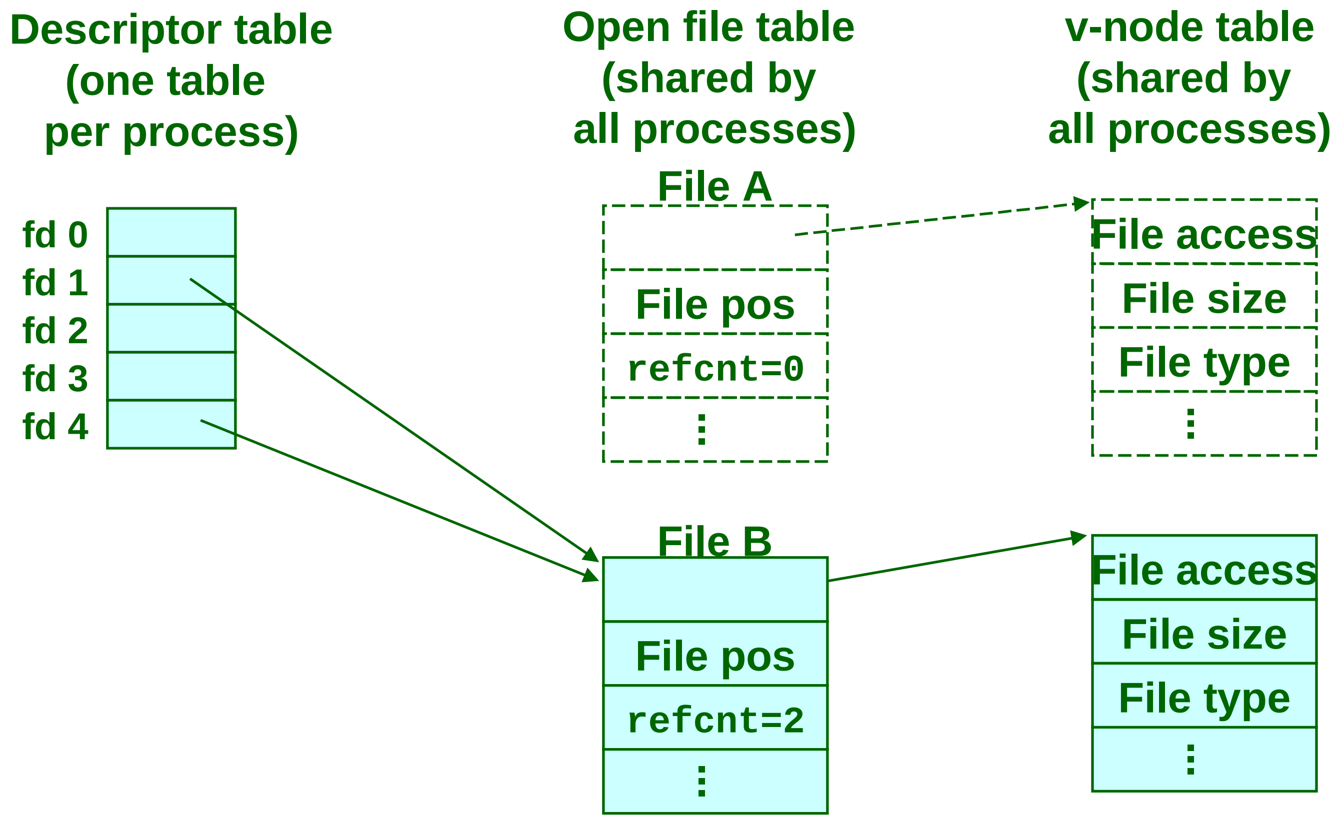
I/O Redirection Example

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



I/O Redirection Example (cont)

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



I/O Redirection

```
/* tsh.c – tiny shell */

void eval(char *cmdline)
{
  <…snip…>
  /* Create a child process */
  pid = Fork();

  if (pid == 0) { /* Child Process */
    <…snip…>

    input_fd = Open(infile, O_RDONLY);
    output_fd = Open(outfile, O_WRONLY|O_CREAT, S_IRUSR|S_IRGRP|S_IROTH);
    dup2(input_fd, 0);
    dup2(output_fd, 1);

    /* Now load and run the program in the new job */
    if (execve(argv[0], argv, environ) < 0) {
        printf("%s: Command not found\n", argv[0]);
        exit(0);
    }
  }
  <…snip…>
}
```

Accessing the Terminal

Only one process group can be in control of the terminal

- Foreground process has access
- All background processes do not
- Background processes receive `SIGTTIN`/`SIGTTOU` signal if they attempt to read/write terminal

Must explicitly place child in the foreground to allow it access to the terminal

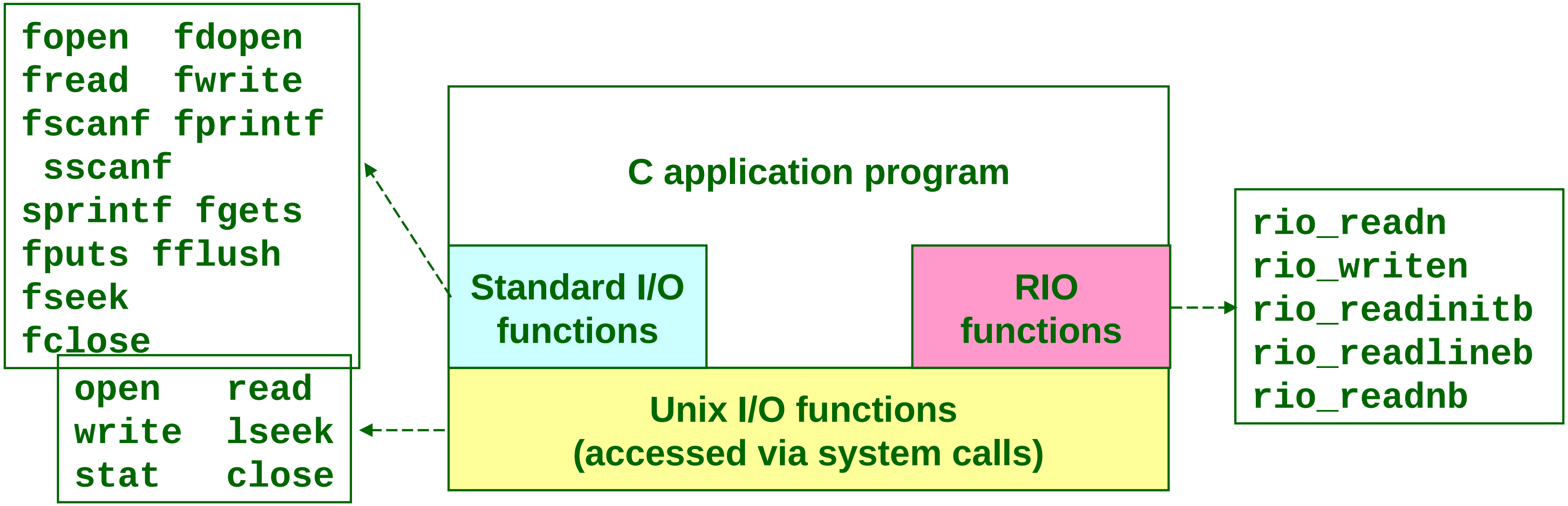
- `int tcsetpgrp(int filedes, pid_t pgid_id)`
- Must block or ignore `SIGTTOU` while setting foreground group (several other intricacies in getting this right)

Standard I/O Functions

- **The C standard library (`libc.a`) contains a collection of higher-level standard I/O functions**
  - **Presented earlier in the lab**
- **Recall the 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`)**

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?

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
- No intermediate buffering (eliminates a copy)

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
- Both of these issues are addressed by the standard I/O and RIO packages

Pros and Cons of Standard I/O

Pros:

- Buffering increases efficiency of small operations, e.g., fgetc(), by decreasing the number of `read` and `write` system calls
- Short counts are handled automatically

Cons:

- Provides no functions for accessing file metadata
- Standard I/O is not appropriate for input and output on network sockets
- There are poorly documented restrictions on stdio streams that interact badly with restrictions on network sockets

Standard I/O Restrictions

Restrictions on streams:

- Restriction 1: input function cannot follow output function without an 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 without an intervening call to `fseek`, `fsetpos`, or `rewind`

Restriction on sockets:

- You are not allowed to change the file position of a socket

Standard I/O Workarounds

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);
```

- Creates a deadly race in concurrent threaded programs!

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 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, but RIO spins in a loop until operation completes, thread does nothing else in the meantime
- Never use standard I/O or raw Unix I/O on sockets or pipes, there are specialized functions for sockets to be discussed

Concurrent I/O

How to deal with multiple I/O operations concurrently?

- For example: wait for a keyboard input, a mouse click and input from a network connection

Select system call

```
int select(int n, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

Poll system call (same idea, different implementation)

```
int poll(struct pollfd *ufds, unsigned int nfds, int timeout);

struct pollfd { int fd;           /* file descriptor */
                short events;     /* requested events */
                short revents;    /* returned events */
              };
```

Other mechanisms are also available

- /dev/poll (Solaris), /dev/epoll (Linux)
- kqueue (FreeBSD)
- Posix real-time signals + sigtimedwait()
- Native Posix Threads Library (NPTL)

Asynchronous I/O

POSIX P1003.4 Asynchronous I/O interface functions:
(available in Solaris, AIX, Tru64 Unix, Linux 2.6,…)

- **aio_cancel**
  - cancel asynchronous read and/or write requests
- **aio_error**
  - retrieve Asynchronous I/O error status
- **aio_fsync**
  - asynchronously force I/O completion, and sets errno to ENOSYS
- **aio_read**
  - begin asynchronous read
- **aio_return**
  - retrieve return status of Asynchronous I/O operation
- **aio_suspend**
  - suspend until Asynchronous I/O Completes
- **aio_write**
  - begin asynchronous write
- **lio_listio**
  - issue list of I/O requests

For Further Information

- W. Richard Stevens, Advanced Programming in the Unix Environment, Addison Wesley, 1993
  - Somewhat dated, but still useful
- W. Richard Stevens, Unix Network Programming : Networking Apis: Sockets and Xti (Volume 1), 1998

Next Time

Networking