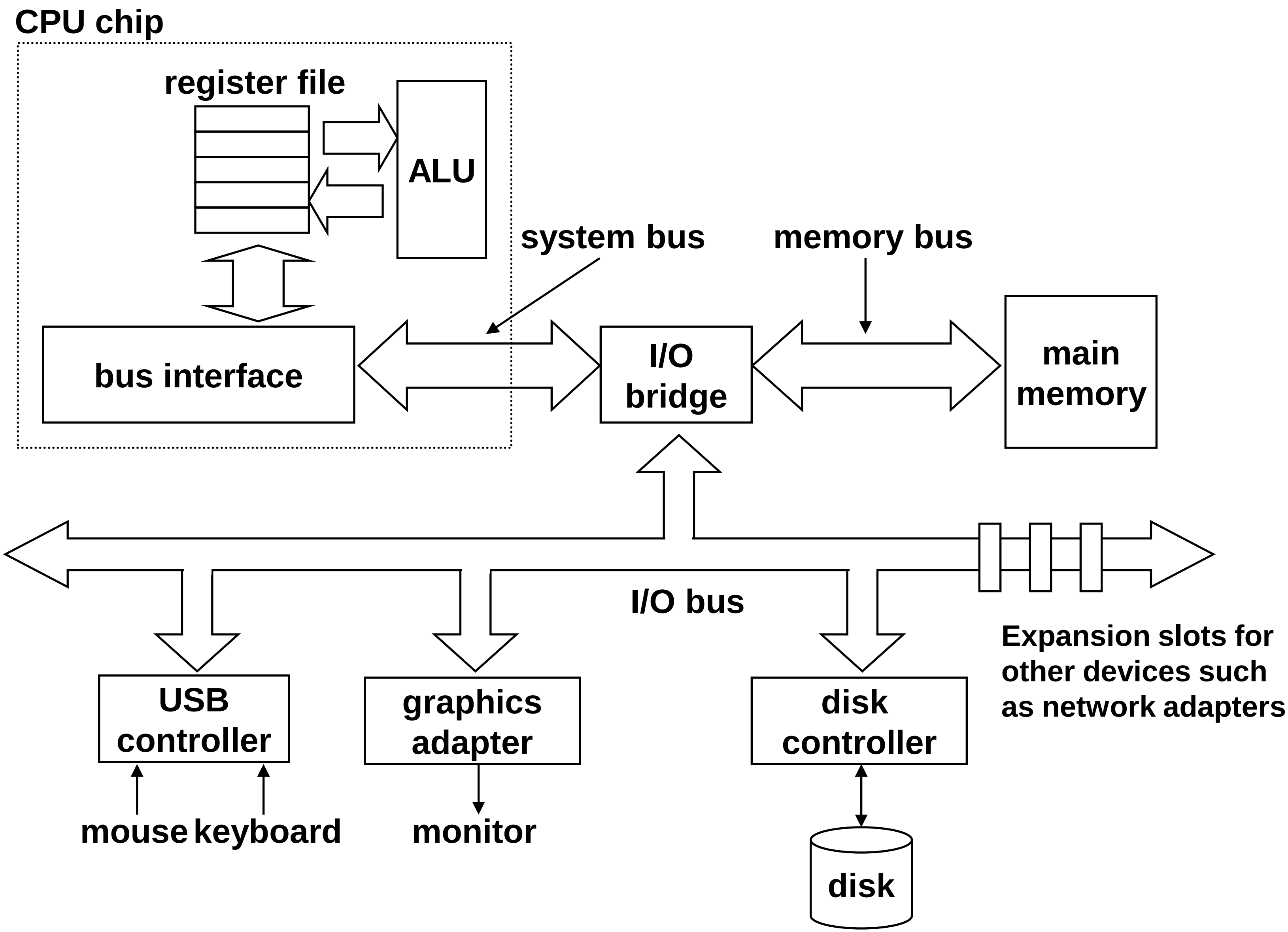
System-Level I/O

Today

- Working with Unix files
- Standard I/O
- Conclusions

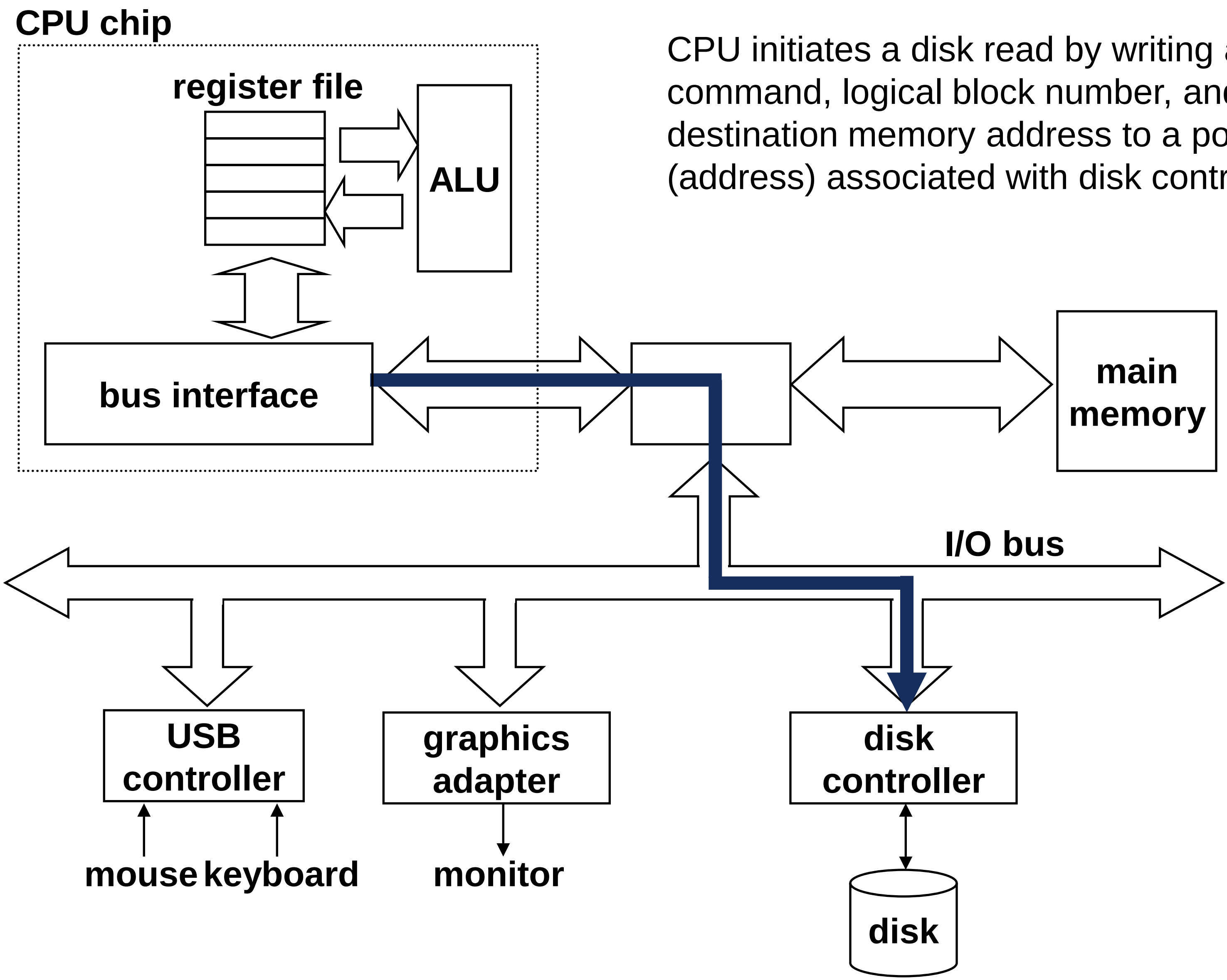
Fabián E. Bustamante, Spring 2010

A typical hardware system

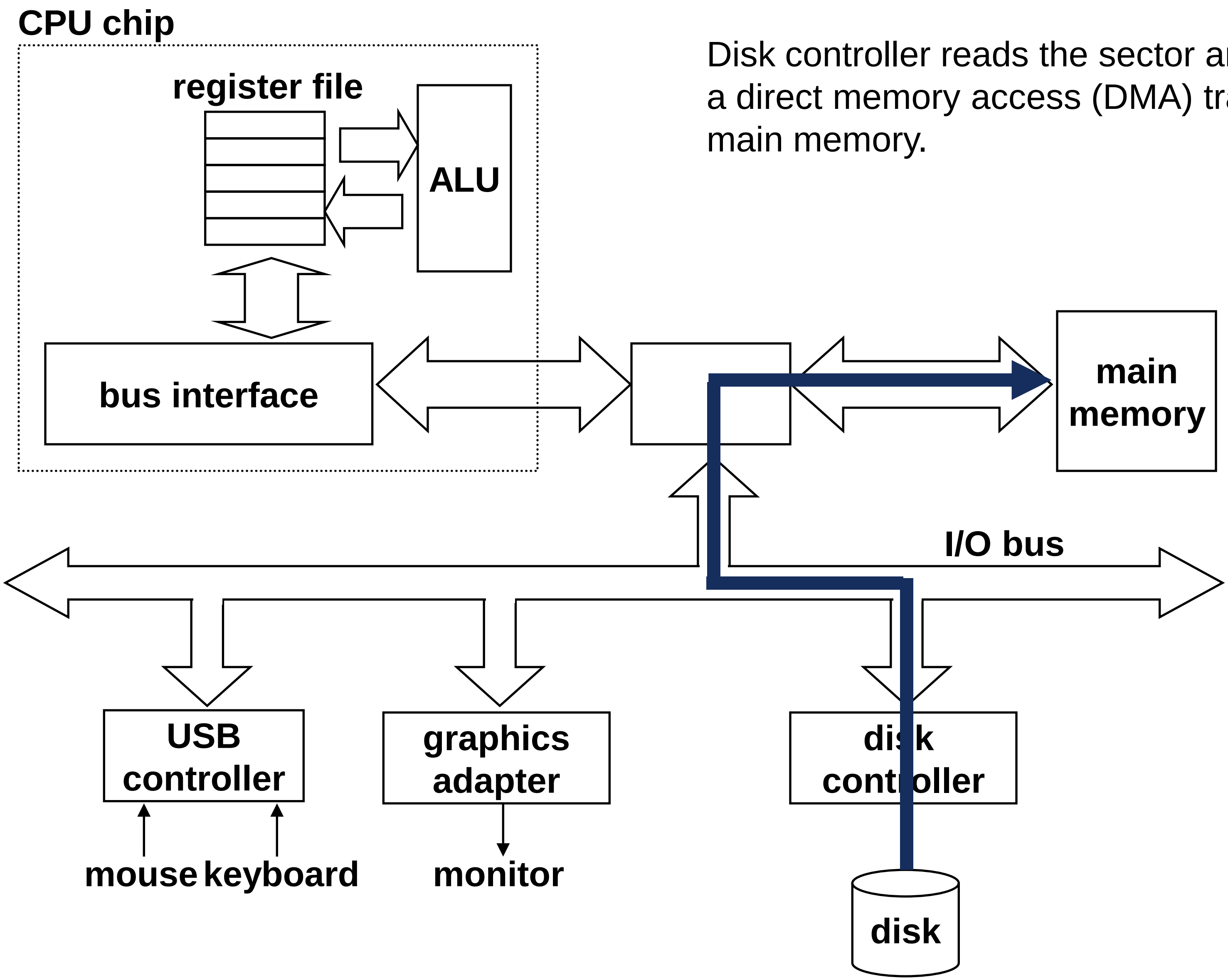


Reading a disk sector: Step 1

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



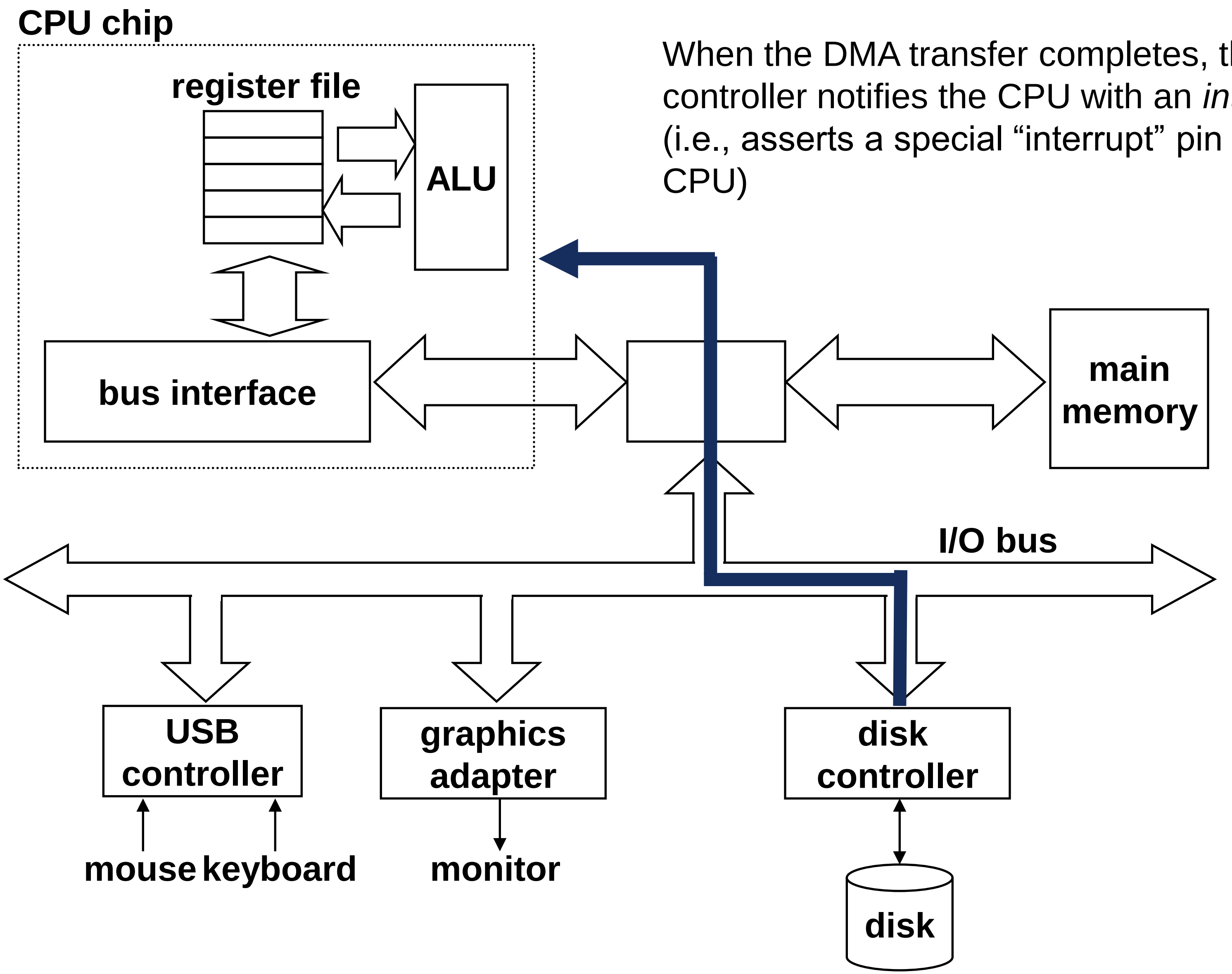
Reading a disk sector: Step 2



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/sda2` (`/usr` disk partition)
  - `/dev/tty2` (terminal)
- Even the kernel is represented as a file:
  - `/dev/kmem` (kernel memory image)
  - `/proc` (kernel data structures)

Unix I/O

- Key features
  - Elegant mapping of files to devices allows kernel to export simple interface
  - Key Unix idea: All input and output is handled in a consistent and uniform way
- Why do we care?
  - Understanding I/O helps you understand other system concepts
  - Sometimes you have no chance but to use Unix I/O functions
- 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()`

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
- Other flags: `O_WRONLY, O_RDWR`
- Each process created by a Unix shell begins life with three open files associated with a terminal:
  - 0: standard input; 1: standard output; 2: standard error

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 (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 */
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`
  - Return type `ssize_t` is signed integer
  - `nbytes == -1` 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 == -1` 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 <stdlib.h>
#include <unistd.h>

int main(void)
{
   char c;

   while((len = read(0 /* stdin */, &c, 1)) == 1) {
      if (write(1 /* stdout */, &c, 1) != 1)
         exit(20);

      if (len == -1) {
         perror("read from stdin failed");
         exit(10);
      }
    }
    exit(0);
}
```

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

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 */
};
```

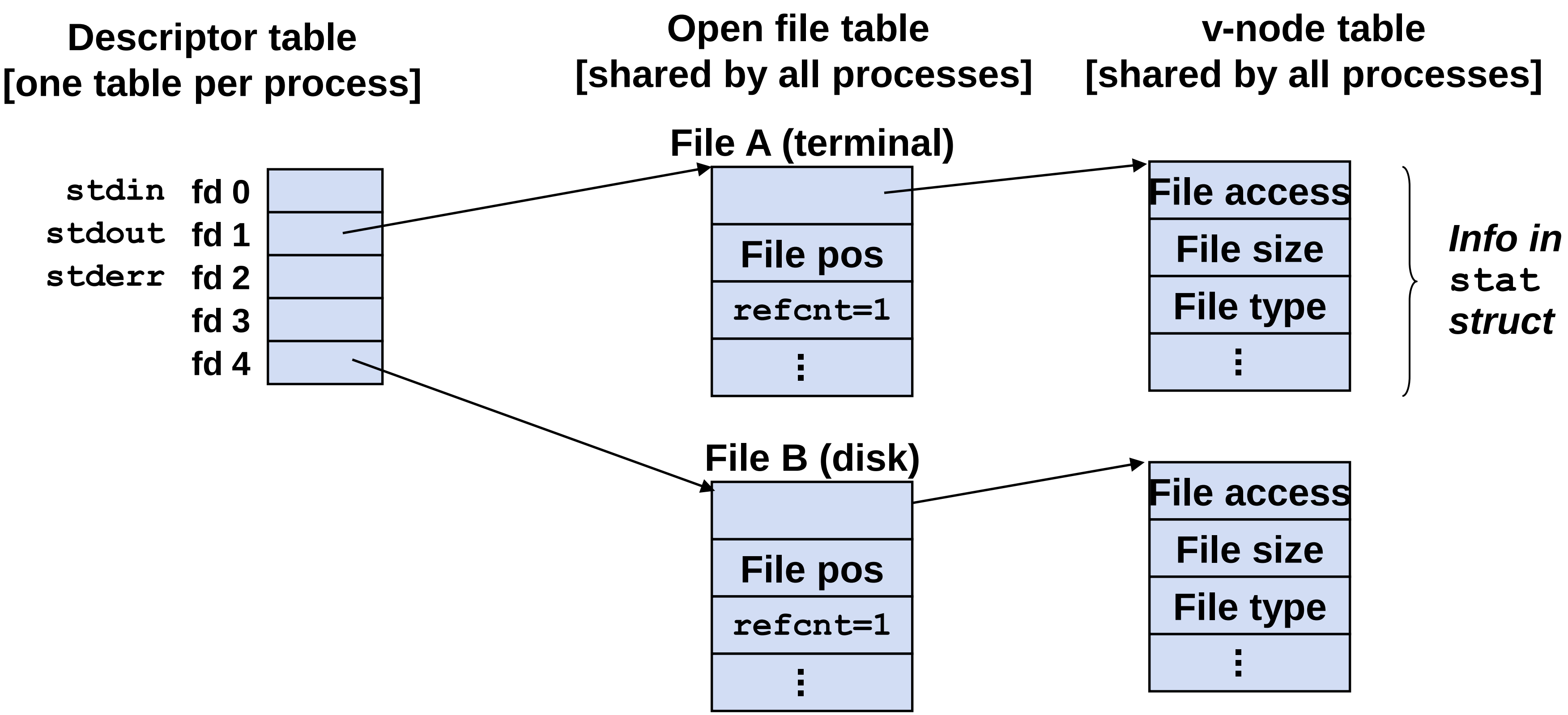
Example of accessing file metadata

```
/* statcheck.c - Querying and manipulating a file's meta data */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.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);
}
```

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

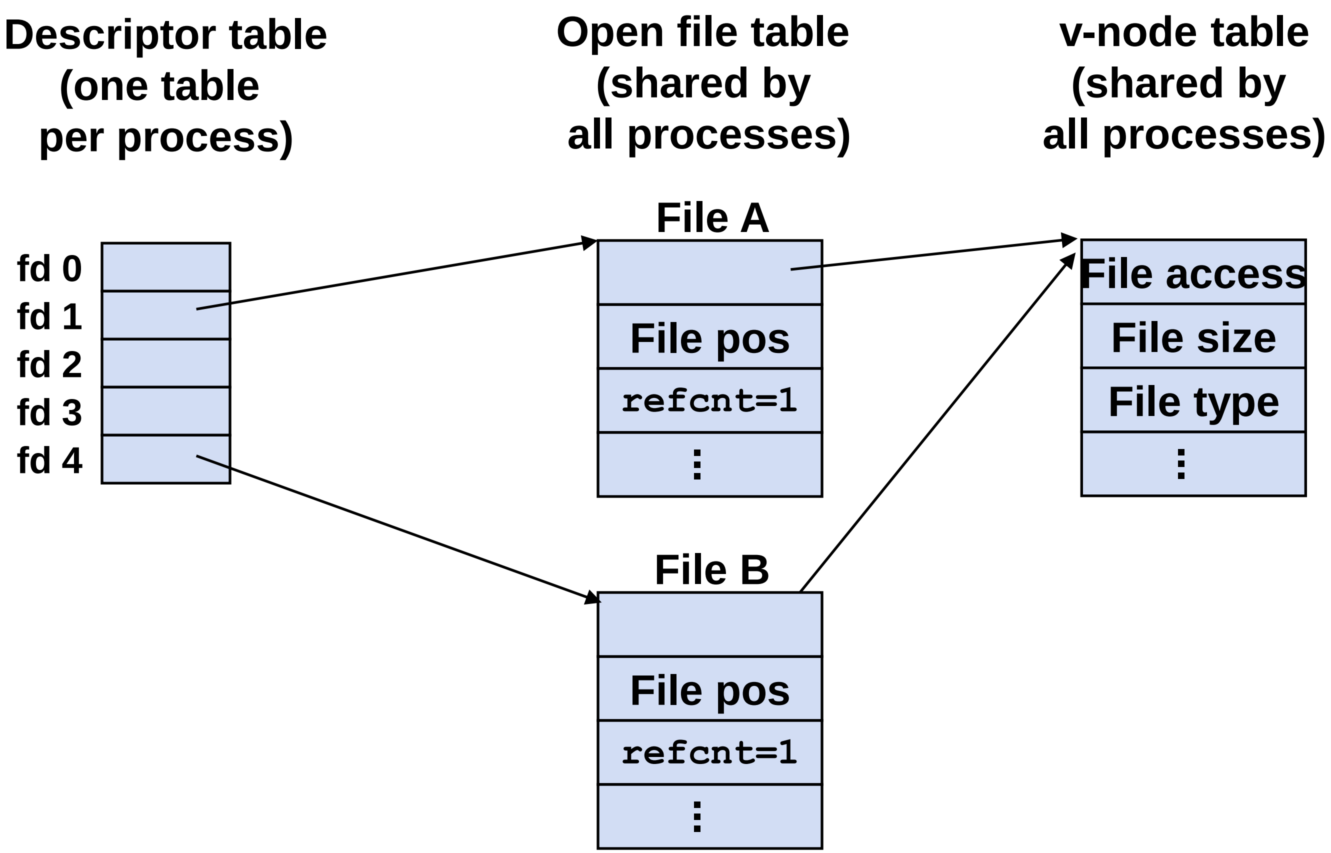
How the kernel represents open files

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



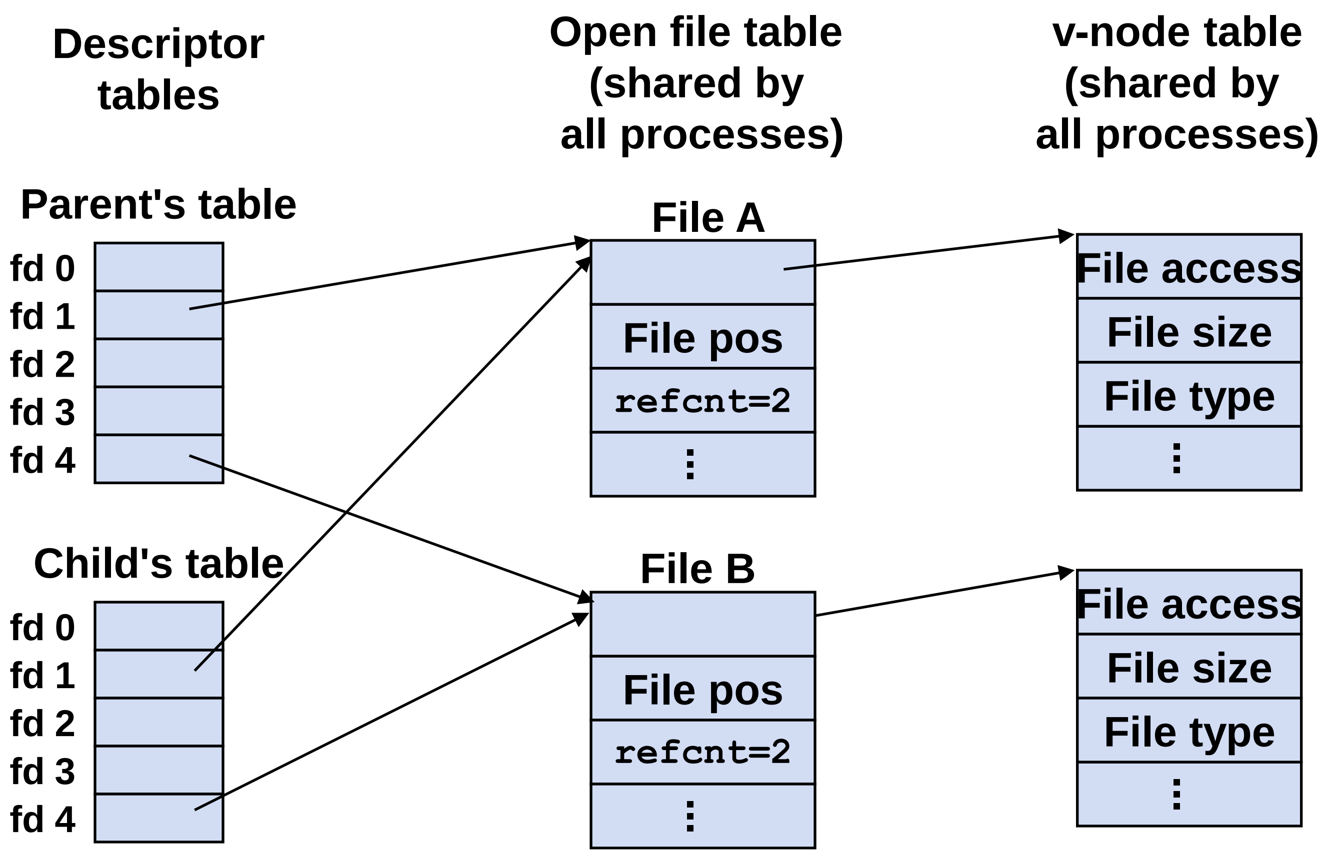
File sharing

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



How processes share files

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

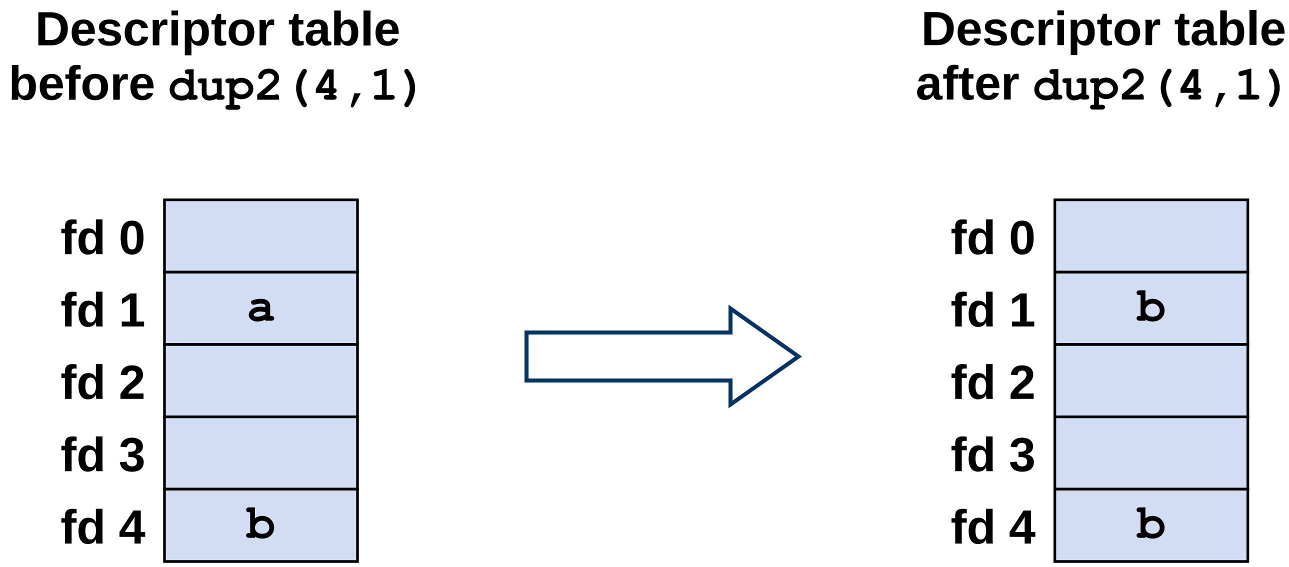


I/O Redirection

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

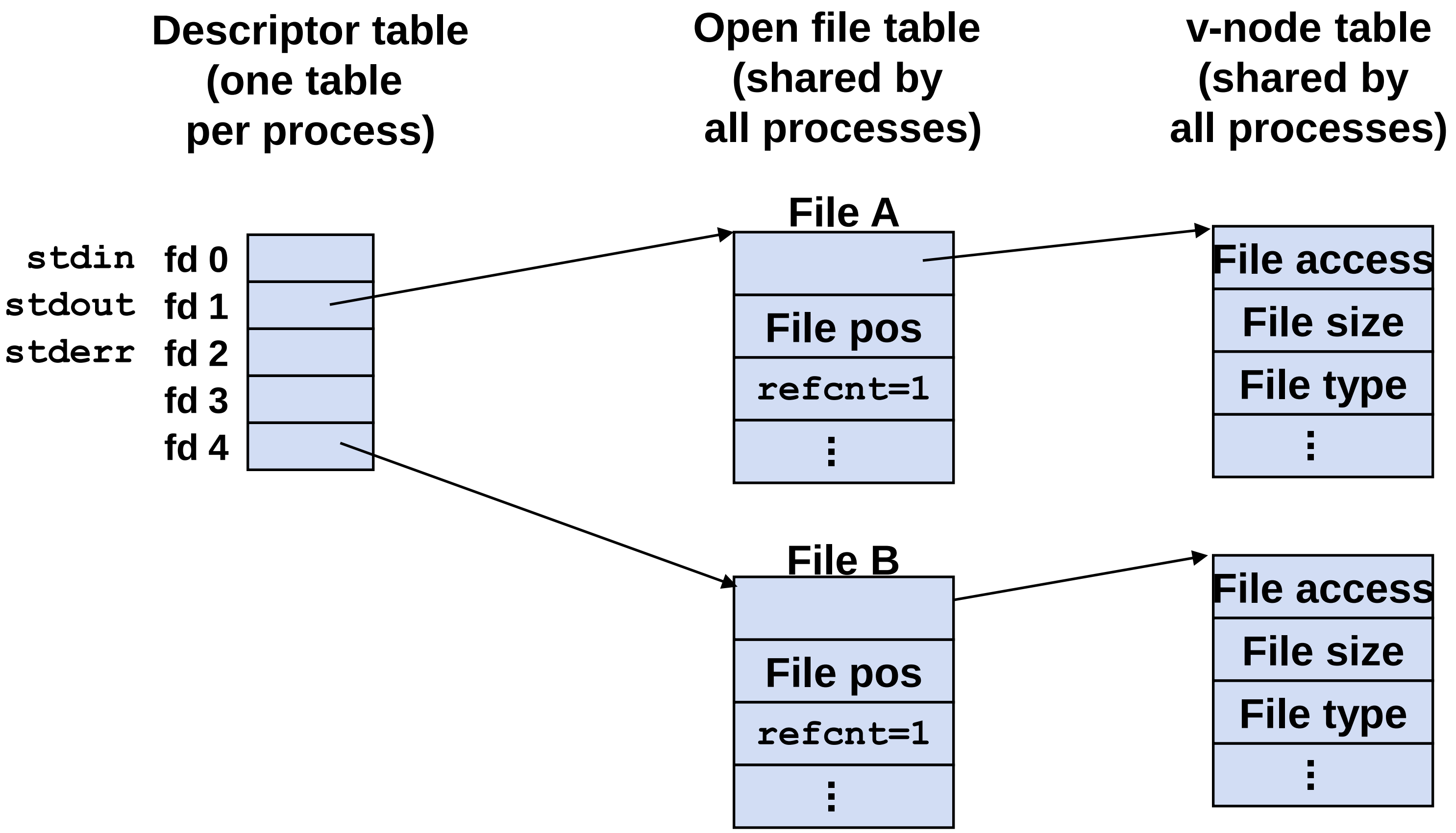
  `unix> ls > foo.txt`

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



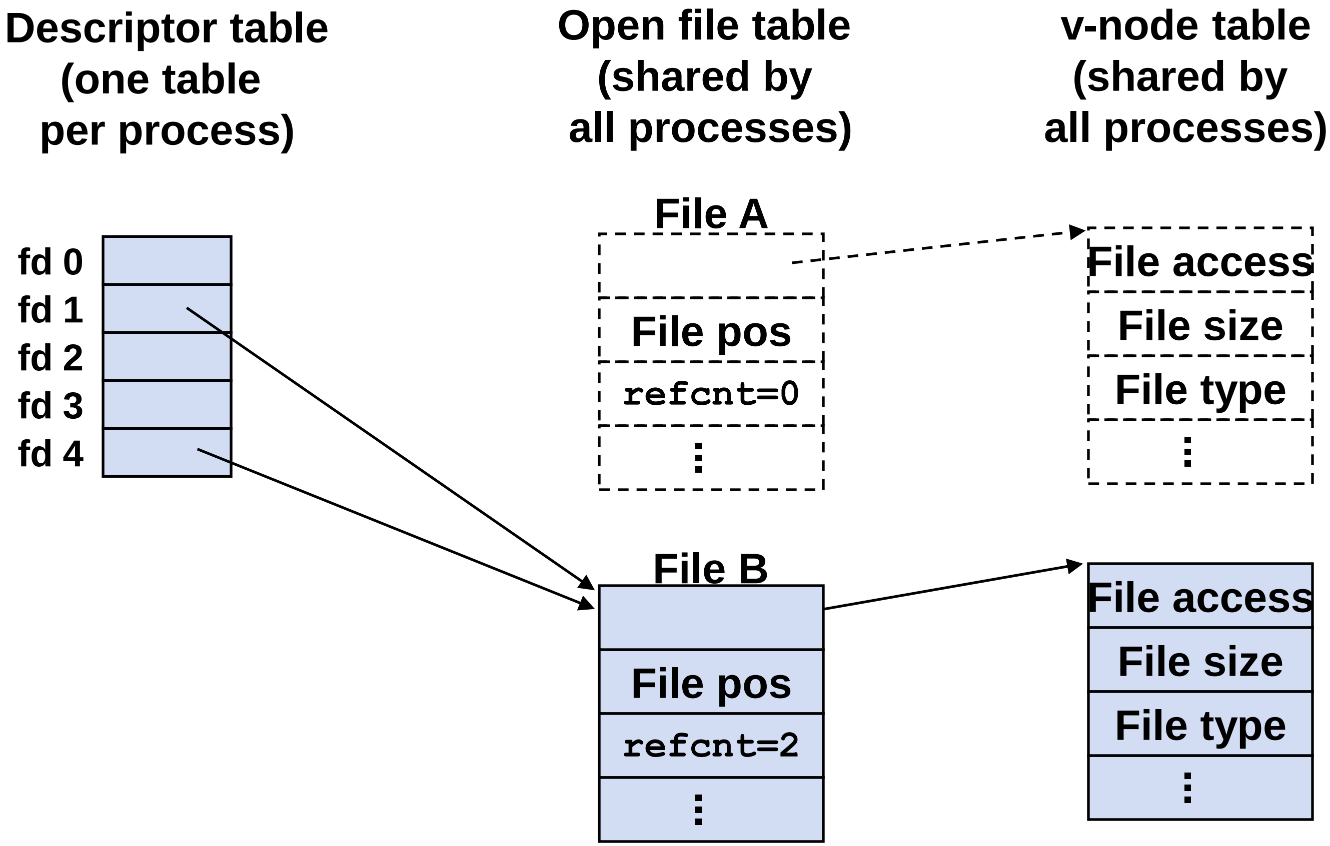
I/O Redirection example

- 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.



Standard I/O functions

- The C standard library (`libc.a`) contains a collection of higher-level standard I/O functions
  - Documented in Appendix B of K&R.
- 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");
}
```

Standard I/O buffering in action

- You can see this buffering in action, using `strace`

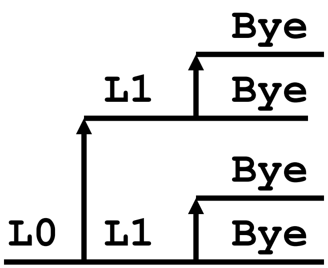
```
#include <stdio.h>

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

```
linux> strace ./bufStdio
execve("./bufStdio", ["./bufStdio"], [/* 24 vars */]) = 0
...
write(1, "hello\n", 6hello ...)                   = 6
exit_group(0)                                 = ?
```

Fork example #2 (earlier lecture)

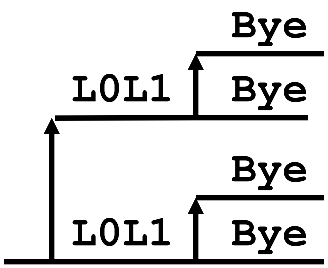
- Both parent and child can continue forking

```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



- Removed the "\n" from the first `printf`
  - "L0" gets printed twice; fork duplicated stream buffer

```
void fork2()
{
    printf("L0");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



Having fun with file descriptors

- What would this program print given a file containing ‘abcde’?

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.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);
  exit(0);
}
```

Having fun with file descriptors

- What would this program print given a file containing ‘abcde’?

```
#include <sys/types.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 {
    sleep(1-s);
    read(fd1, &c2, 1);
    printf("Child: c1 = %c, c2 = %c\n", c1, c2);
  }
  exit(0);
}
```

Having fun with file descriptors

- What would be the content of the resulting file?

```
#include <sys/types.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(fd1, "jklmn", 5);
  fd2 = dup(fd1);
  write(fd2, "wxyz", 4);
  write(fd3, "ef", 2);
  exit(0);
}
```

Pros/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
- 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

Pros/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

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.

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.

Summary

- System level I/O from the programmer perspective
  - For the underlying details – EECS 343

- Next time
  - There is no next time ☹