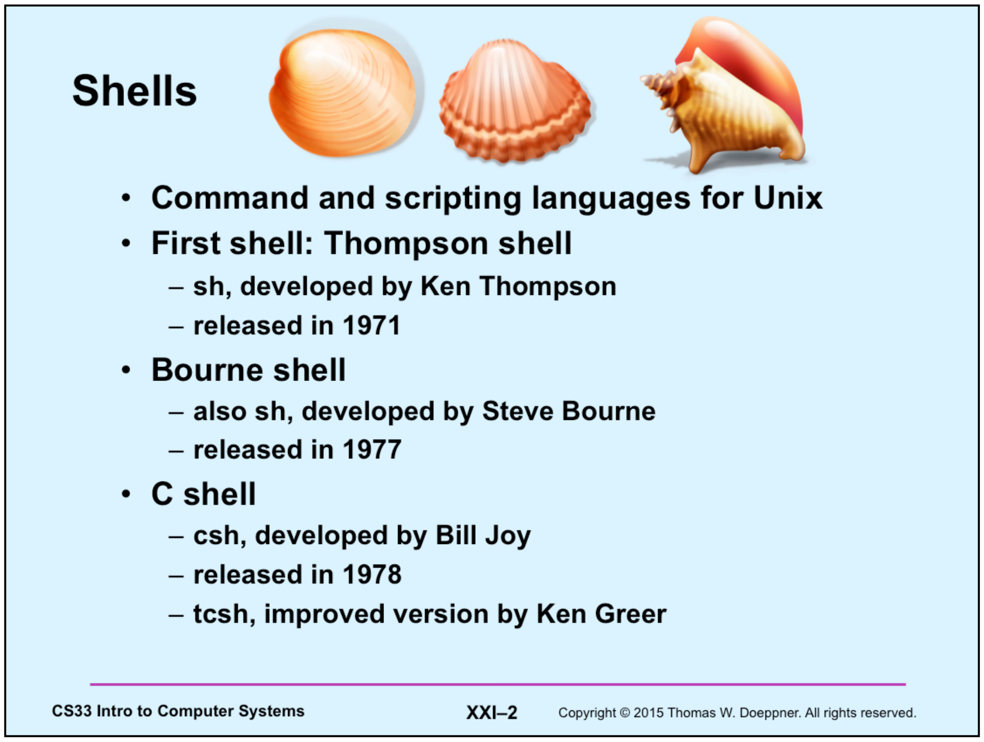
# CS 33

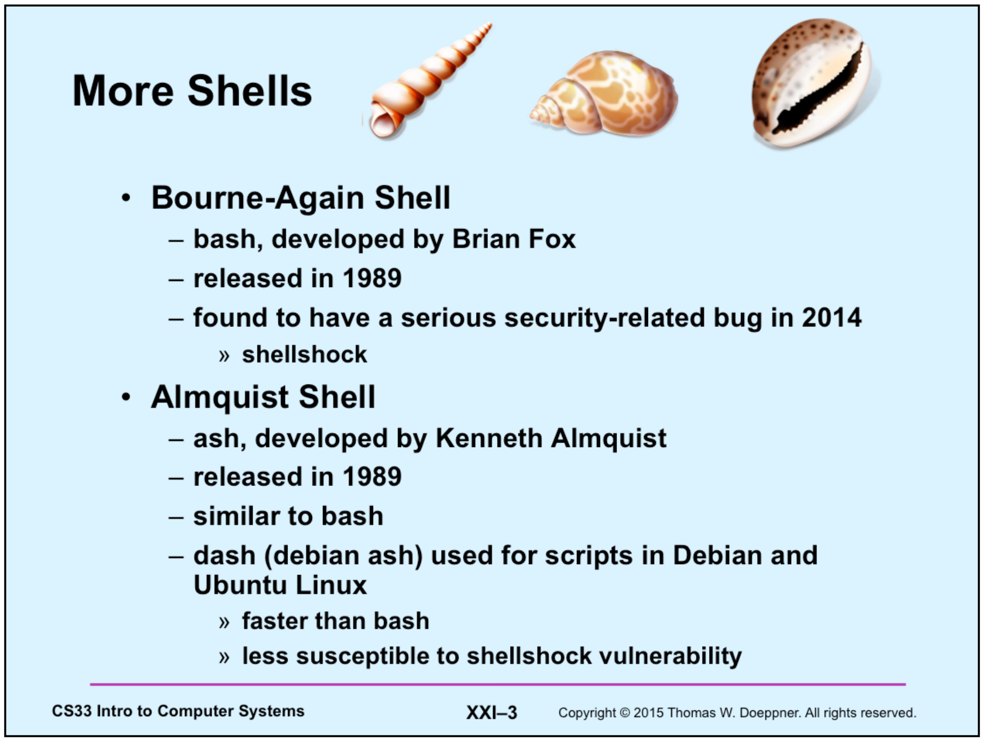
## Shells and Files



## Shells

- Command and scripting languages for Unix
- First shell: Thompson shell
  - sh, developed by Ken Thompson
  - released in 1971
- Bourne shell
  - also sh, developed by Steve Bourne
  - released in 1977
- C shell
  - csh, developed by Bill Joy
  - released in 1978
  - tcsh, improved version by Ken Greer

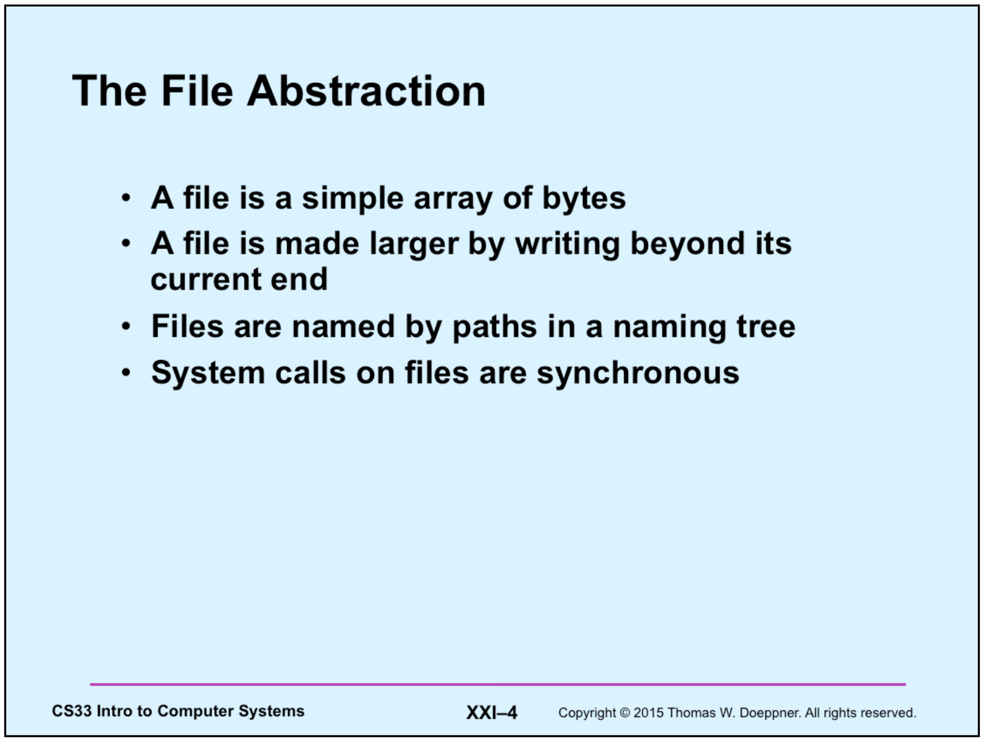
This information is from Wikipedia.



## More Shells

- Bourne-Again Shell
  - bash, developed by Brian Fox
  - released in 1989
  - found to have a serious security-related bug in 2014
    - » shellshock
- Almquist Shell
  - ash, developed by Kenneth Almquist
  - released in 1989
  - similar to bash
  - dash (debian ash) used for scripts in Debian and Ubuntu Linux
    - » faster than bash
    - » less susceptible to shellshock vulnerability

This information is also from Wikipedia.



## The File Abstraction

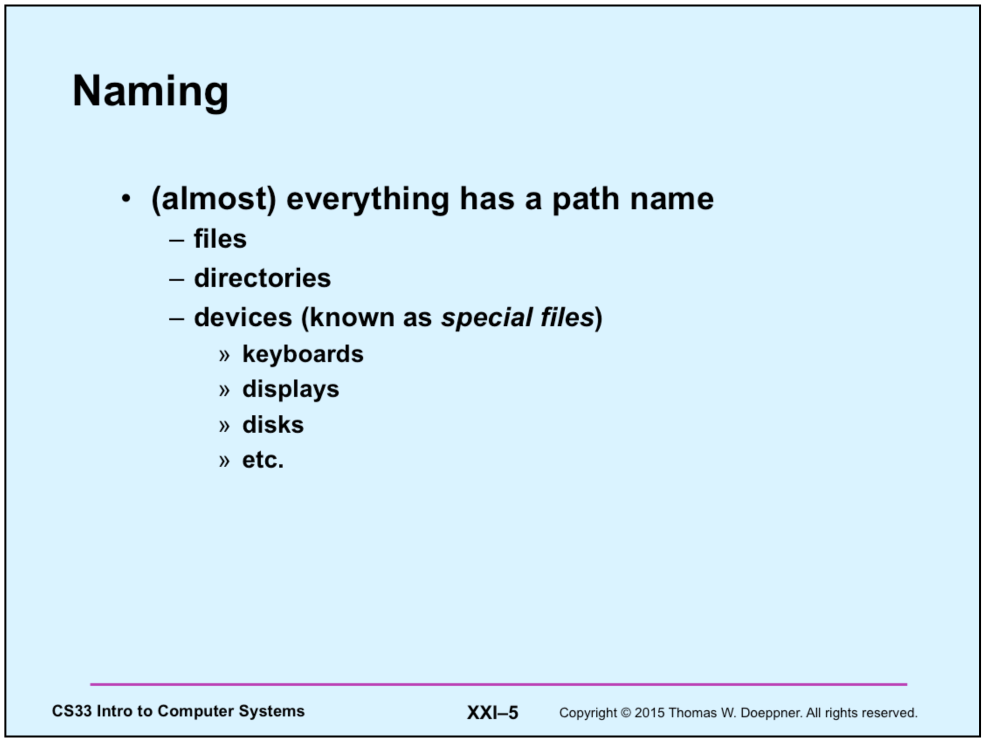
- A file is a simple array of bytes
- A file is made larger by writing beyond its current end
- Files are named by paths in a naming tree
- System calls on files are synchronous

CS33 Intro to Computer Systems XXI–4

Most programs perform file I/O using library code layered on top of system calls. In this section we discuss just the kernel aspects of file I/O, looking at the abstraction and the high-level aspects of how this abstraction is implemented.

The Unix file abstraction is very simple: files are simply arrays of bytes. Some systems have special system calls to make a file larger. In Unix, you simply write where you've never written before, and the file "magically" grows to the new size (within limits). The names of files are equally straightforward — just the names labeling the path that leads to the file within the directory tree. Finally, from the programmer's point of view, all operations on files appear to be synchronous — when an I/O system call returns, as far as the process is concerned, the I/O has completed. (Things are different from the kernel's point of view.)

Note that there are numerous issues in implementing the Unix file abstraction that we do not cover in this course. In particular, we do not discuss what is done to lay out files on disks (both rotating and solid-state) so as to take maximum advantage of their architectures. Nor do we discuss the issues that arise in coping with failures and crashes. What we concentrate on here are those aspects of the file abstraction that are immediately relevant to application programs.



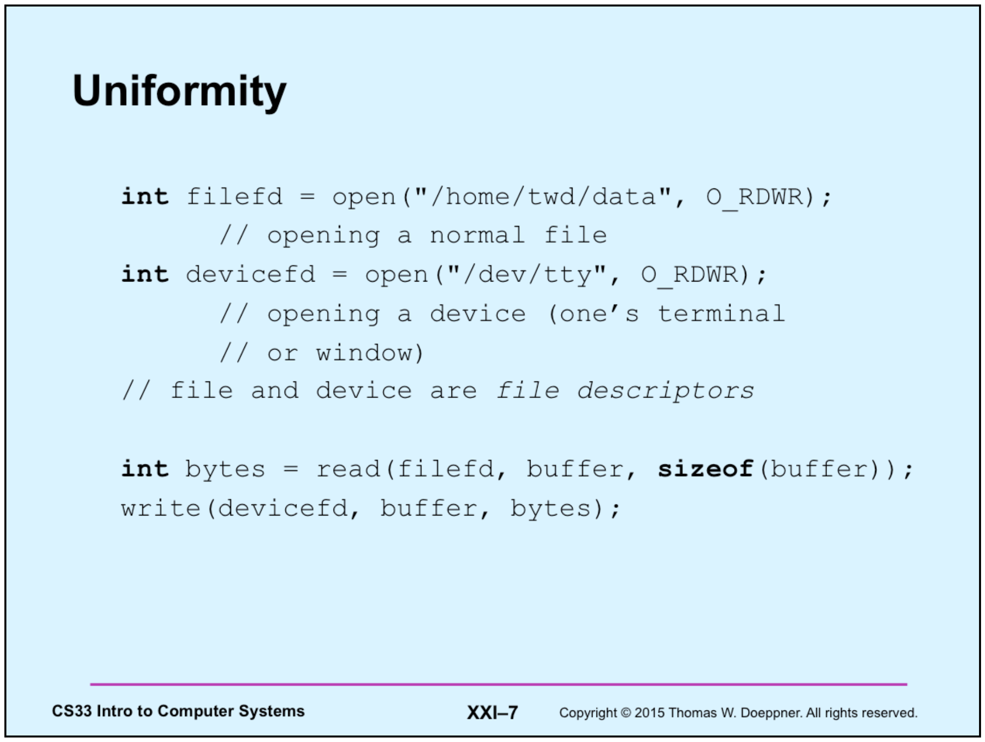
## Naming

- (almost) everything has a path name
  - files
  - directories
  - devices (known as *special files*)
    - » keyboards
    - » displays
    - » disks
    - » etc.

The notion that almost everything in Unix has a path name was a startlingly new concept when Unix was first developed; one that has proved to be important.

## I/O System Calls

- **`int`** `file_descriptor = open(pathname, mode [, permissions])`
- **`int`** `close(file_descriptor)`
- **`int`** `count = read(file_descriptor, buffer_address, buffer_size)`
- **`int`** `count = write(file_descriptor, buffer_address, buffer_size)`



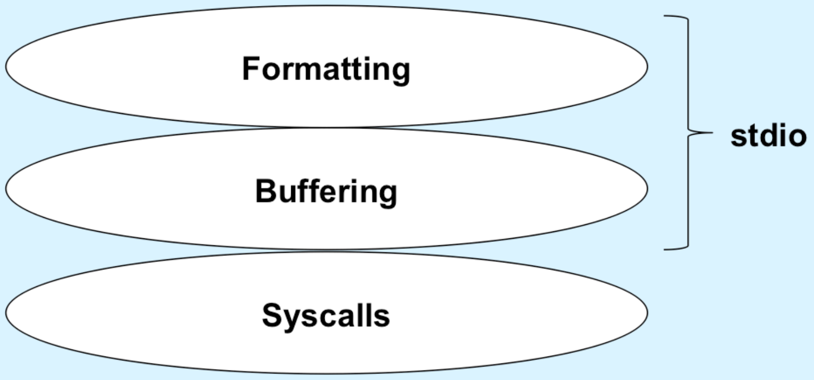
## Uniformity

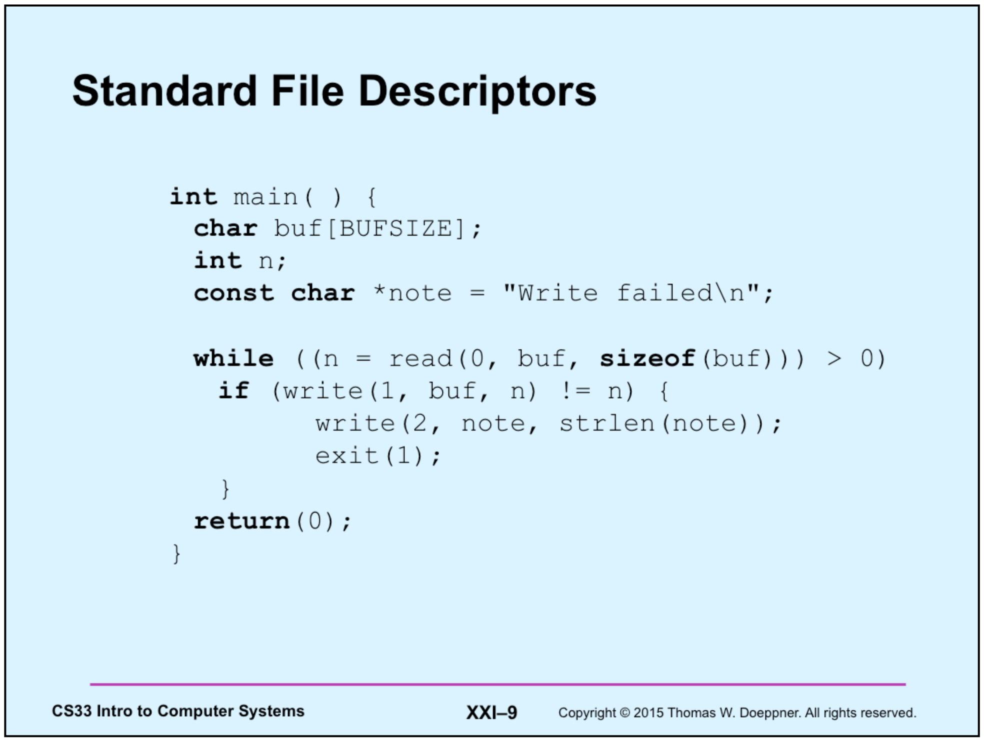
```
int filefd = open("/home/twd/data", O_RDWR);
      // opening a normal file
int devicefd = open("/dev/tty", O_RDWR);
      // opening a device (one’s terminal
      // or window)
// file and device are file descriptors

int bytes = read(filefd, buffer, sizeof(buffer));
write(devicefd, buffer, bytes);
```

This notion that everything has a path name facilitates a uniformity of interface. Reading and writing a normal file involves a different set of internal operations than reading and writing a device, but they are named in the same style and the I/O system calls treat them in the same way. What we have is a form of polymorphism (though the term didn’t really exist when the original Unix developers came up with this way of doing things).

Note that the *open* system call returns an integer called a *file descriptor,* used in subsequent system calls to refer to the file.

## Standard I/O Library





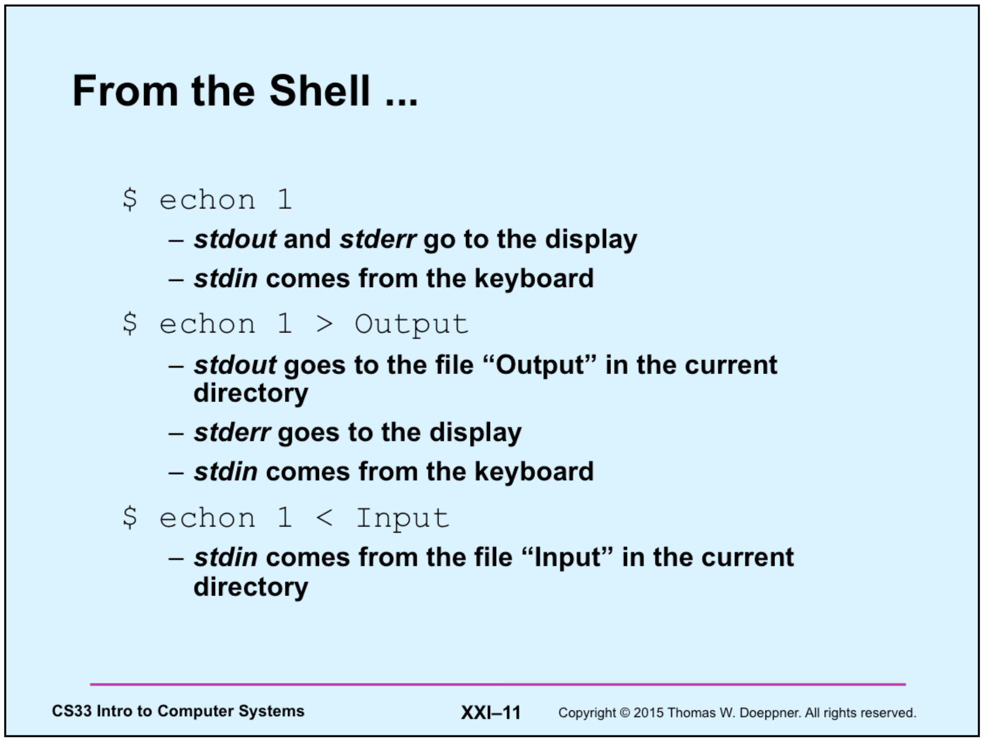
## Standard File Descriptors

```
int main( ) {
  char buf[BUFSIZE];
  int n;
  const char *note = "Write failed\n";

  while ((n = read(0, buf, sizeof(buf))) > 0)
    if (write(1, buf, n) != n) {
            write(2, note, strlen(note));
            exit(1);
    }
  return(0);
}
```

The file descriptors 0, 1, and 2 are opened to access your window when you log in, and are preserved across forks, unless redirected.

## A Program

```
int main(int argc, char *argv[]) {
  if (argc != 2) {
    fprintf(stderr, "Usage: echon reps\n");
    exit(0);
  }
  int reps = atoi(argv[1]);
  if (reps > 2) {
    fprintf(stderr, "reps too large, reduced to 2\n");
    reps = 2;
  }
  char buf[256];
  while (fgets(buf, 256, stdin) != NULL)
    for (int i=0; i<reps; i++)
      fputs(buf, stdout);
  return(0);
}
```



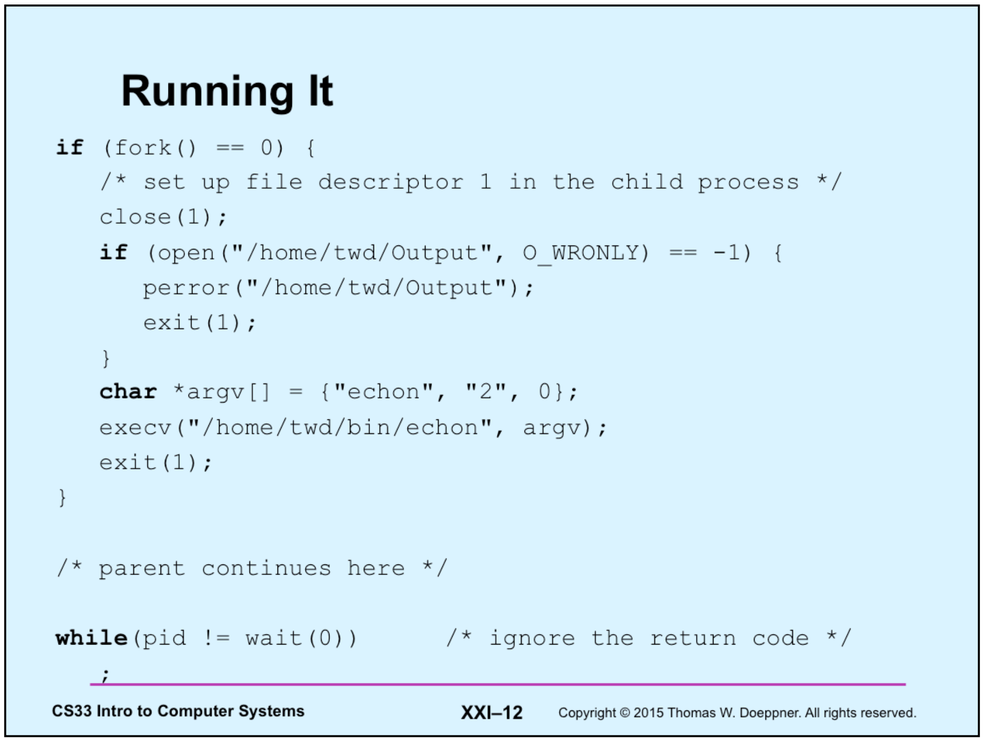
## From the Shell ...

```
$ echon 1
```
- ***stdout*** **and** ***stderr*** **go to the display**
- ***stdin*** **comes from the keyboard**

```
$ echon 1 > Output
```
- ***stdout*** **goes to the file "Output" in the current directory**
- ***stderr*** **goes to the display**
- ***stdin*** **comes from the keyboard**

```
$ echon 1 < Input
```
- ***stdin*** **comes from the file "Input" in the current directory**

Our shell examples are all in bash.



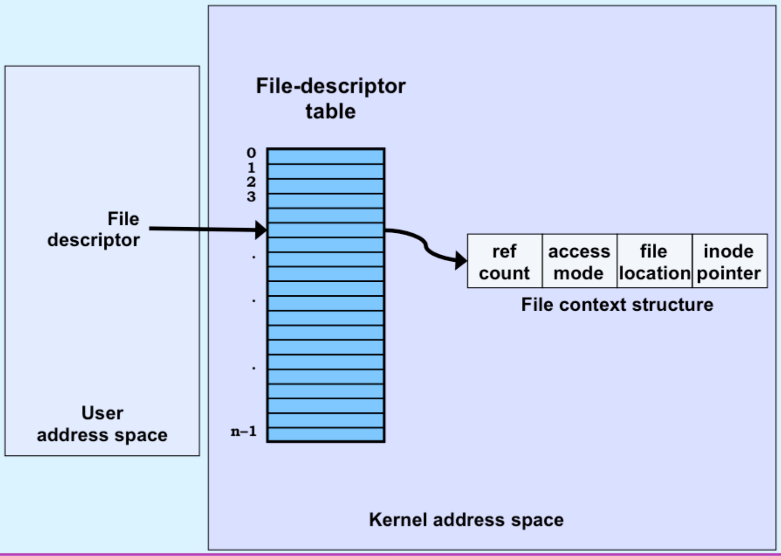
## Running It

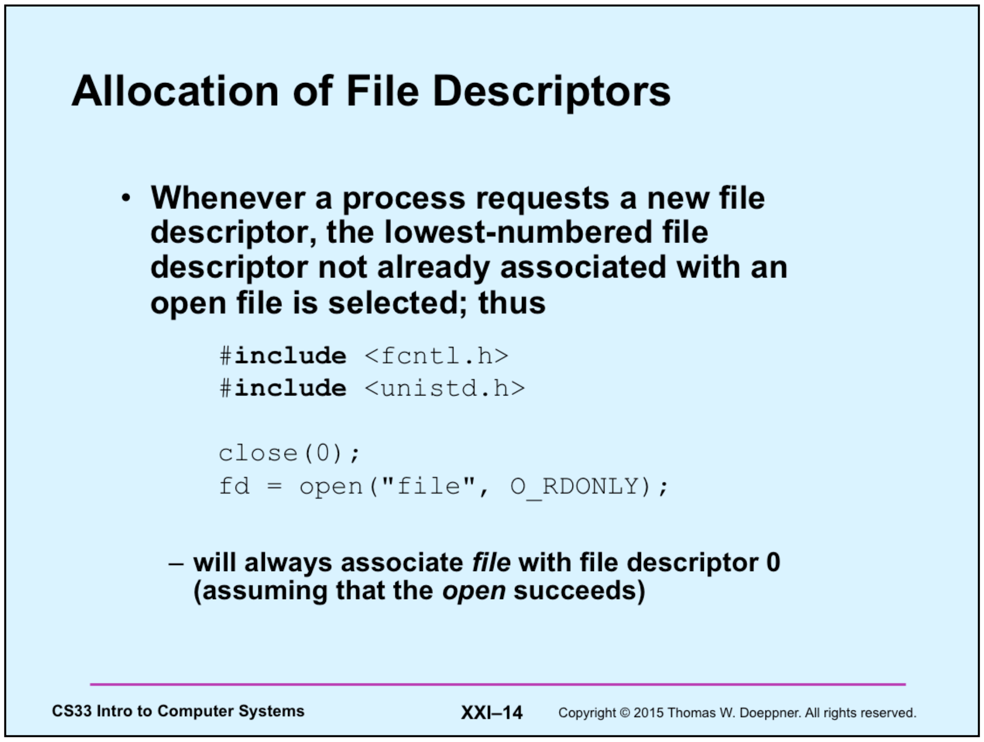
```
if (fork() == 0) {
   /* set up file descriptor 1 in the child process */
   close(1);
   if (open("/home/twd/Output", O_WRONLY) == -1) {
      perror("/home/twd/Output");
      exit(1);
   }
   char *argv[] = {"echon", "2", 0};
   execv("/home/twd/bin/echon", argv);
   exit(1);
}

/* parent continues here */

while(pid != wait(0))      /* ignore the return code */
   ;
```

Here we arrange so that file descriptor 1 (standard output) refers to /home/twd/Output. As we discuss soon, if open succeeds, the file descriptor it assigns is the lowest-numbered one available. Thus if file descriptors 0, 1, and 2 are unavailable (because they correspond to standard input, standard output and standard error), then if file descriptor 1 is closed, it becomes the lowest-numbered available file descriptor. Thus the call to open, if it succeeds, returns 1.

The *wait* system call is similar to *waitpid*, except that it waits for any child process to terminate, not just some particular one. Its argument is the address of where return status is to be stored. In this case, by specifying zero, we're saying that we're not interested — status info should not be stored.

# File-Descriptor Table





## Allocation of File Descriptors

- **Whenever a process requests a new file descriptor, the lowest-numbered file descriptor not already associated with an open file is selected; thus**

```
#include <fcntl.h>
#include <unistd.h>

close(0);
fd = open("file", O_RDONLY);
```

  - **will always associate *file* with file descriptor 0 (assuming that the *open* succeeds)**

One can depend on always getting the lowest available file descriptor.

## Redirecting Output … Twice

```
if (fork() == 0) {
   /* set up file descriptors 1 and 2 in the child process */
   close(1);
   close(2);
   if (open("/home/twd/Output", O_WRONLY) == -1) {
      exit(1);
   }
   if (open("/home/twd/Output", O_WRONLY) == -1) {
      exit(1);
   }
   char *argv[] = {"echon", 2};
   execv("/home/twd/bin/echon", argv);
   exit(1);
}
/* parent continues here */
```

## From the Shell ...

```
$ echon 1 >Output 2>Output
```

- **both stdout and stderr go to Output file**

## Quiz 1

- **Suppose we run**

```
% echon 3 >Output 2>Output
```

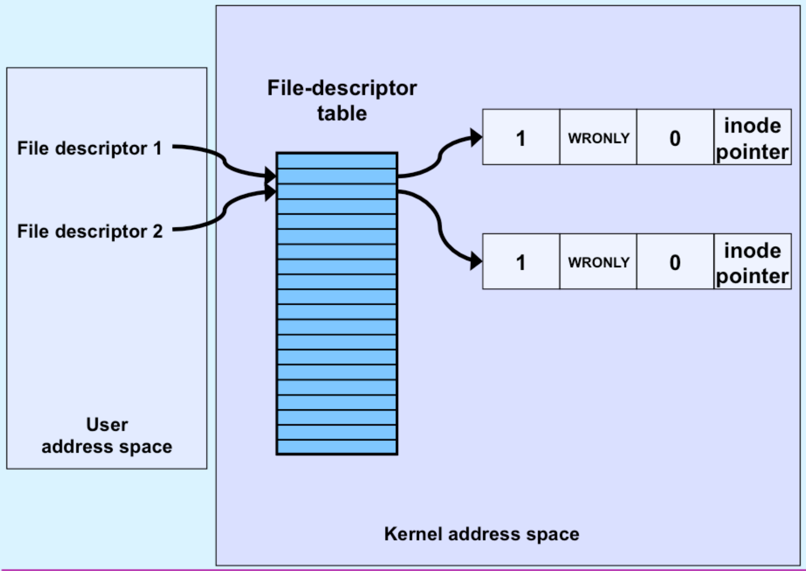
- **The input line is**

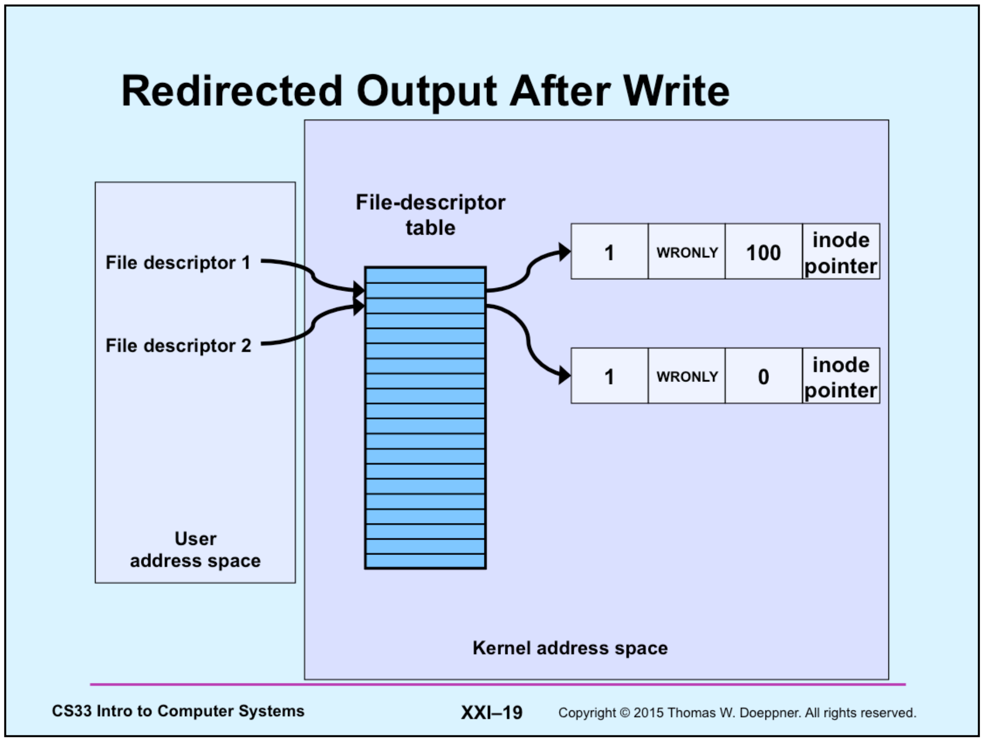
```
X
```

- **What is the final content of Output?**

```
a) reps too large, reduced to 2\nX\nX\n
b) X\nX\nreps too large, reduced to 2\n
c) X\nX\n too large, reduced to 2\n
```

## Redirected Output



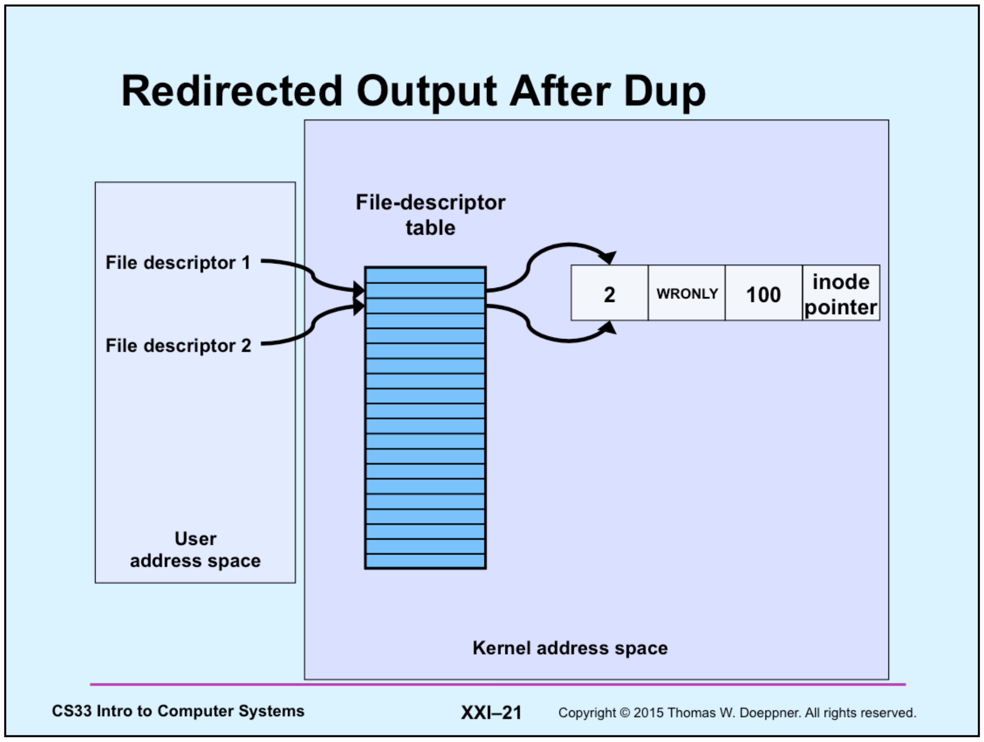


## Redirected Output After Write

The potential problem here is that, since our file (/home/twd/Output) has been opened once for each file descriptor, when a write is done through file descriptor 1, the file location field in its context is incremented by 100, but not that in the other context. Thus a subsequent write via file descriptor 2 would overwrite what was just written via file descriptor 1.

## Sharing Context Information

```
if (fork() == 0) {
   /* set up file descriptors 1 and 2 in the child process */
   close(1);
   close(2);
   if (open("/home/twd/Output", O_WRONLY) == -1) {
      exit(1);
   }
   dup(1); /* set up file descriptor 2 as a duplicate of 1 */
   char *argv[] = {"echon", 2};
   execv("/home/twd/bin/echon", argv);
   exit(1);
}
/* parent continues here */
```



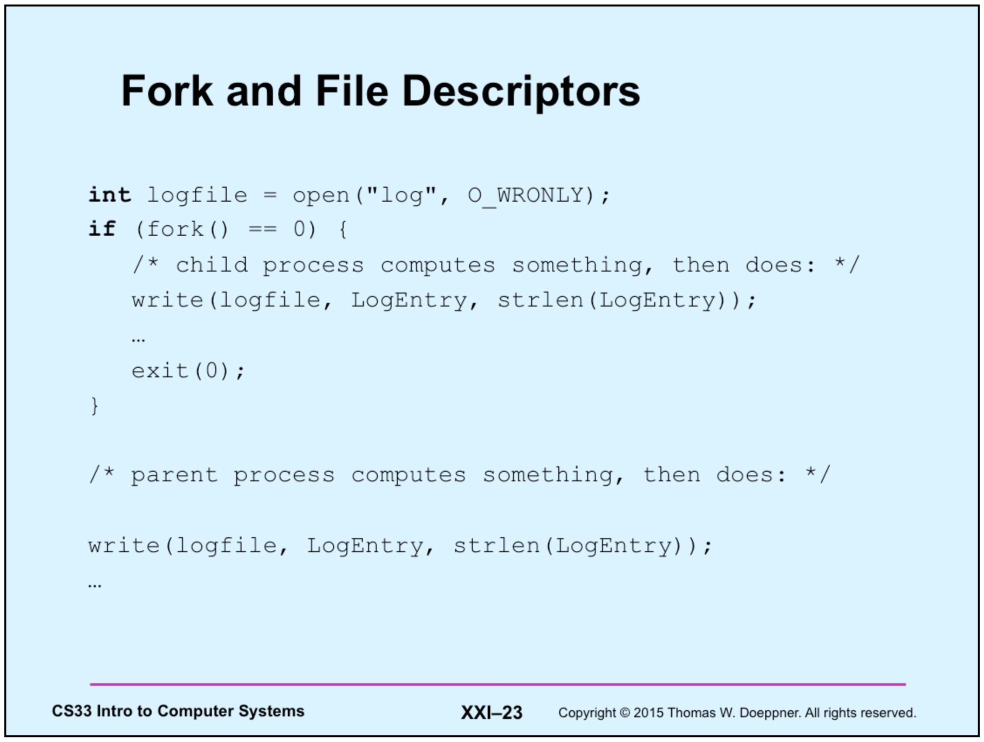
## Redirected Output After Dup

Here we have one file construct structure shared by both file descriptors, so an update to the file location field done via one file descriptor affects the other as well.

## From the Shell ...

```
$ echon 3 >Output 2>&1
```

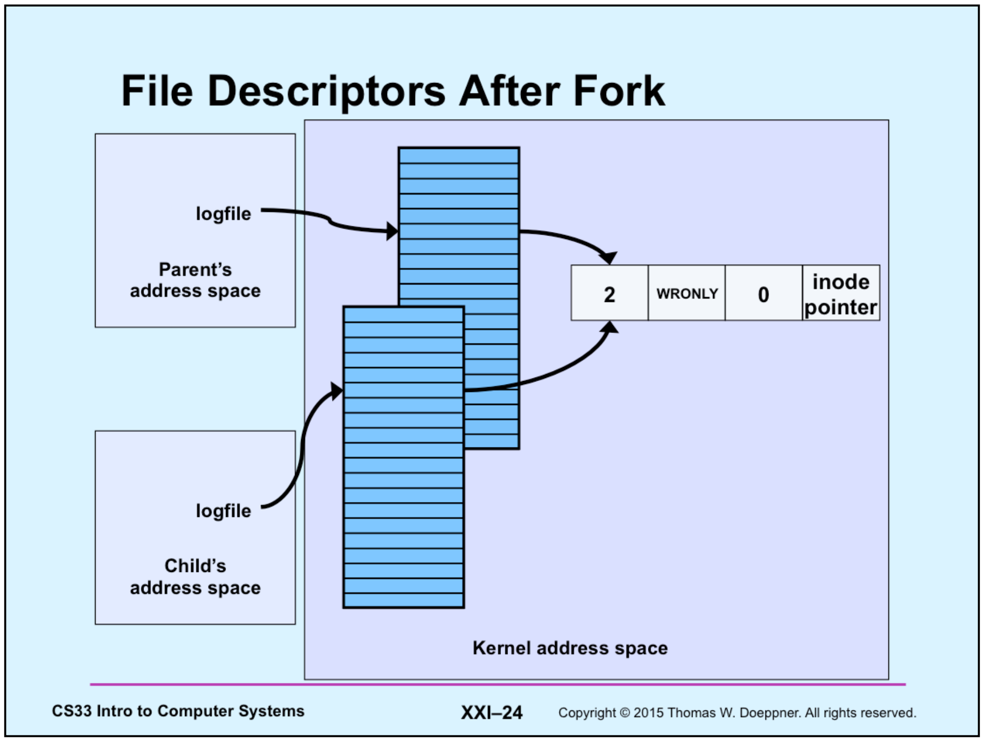
- stdout goes to Output file, stderr is the dup of fd 1



## Fork and File Descriptors

```
int logfile = open("log", O_WRONLY);
if (fork() == 0) {
   /* child process computes something, then does: */
   write(logfile, LogEntry, strlen(LogEntry));
   …
   exit(0);
}

/* parent process computes something, then does: */

write(logfile, LogEntry, strlen(LogEntry));
…
```

Here we have a logfile into which important information should be appended by each of our processes. To make sure that each write goes to the current end of the file, it's desirable that the "logfile" file descriptor in each process refer to the same shared file context structure. As it turns out, this does indeed happen: after a fork, the file descriptors in the child process refer to the same file context structures as they did in the parent.



## File Descriptors After Fork

Note that after a fork, the reference counts in the file context structures are incremented to account for the new references by the child process.

## Quiz 2

```
int main() {
  if (fork() == 0) {
    fprintf(stderr, "Child");
    exit(0);
  }
  printf("Parent");
}
```

**Suppose the program is run as:**

```
% prog >file 2>&1
```

**What is the final content of file?**

- **a) either “ChildParent” or “ParentChild”**
- **b) either “Childt” or “Parent”**
- **c) either “Child” or “Parent”**