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 Linux
    - » faster than bash
    - » less susceptible to shellshock vulnerability

This information is also from Wikipedia.

Roadmap

- We explore the file abstraction
  - what are files
  - how do you use them
  - how does the OS represent them
- We explore the shell
  - how does it launch programs
  - how does it connect programs with files
  - how does it control running programs

shell 1

shell 2

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

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)
- **ssize_t** count = read(file_descriptor, buffer_address, buffer_size)
- **ssize_t** count = write(file_descriptor, buffer_address, buffer_size)
- **off_t** position lseek(file_descriptor, offset, whence)

Given the name of a file, one uses *open* to get a file descriptor that will refer to that file when performing operations on it. One calls close to tell the system one is no longer using that file descriptor. The *read* and *write* system calls perform the indicated operation on the file, using a buffer described by their second two arguments. By default, *read* and *write* operations go through a file from beginning to end sequentially. The *lseek* system call is used to specify where in a file the next read or write will take place.

ssize_t ("signed size") is a typedef for *long* and represents the number of bytes that were transferred. It's signed so as to allow -1 as a return value, which indicates an error. *off_t* is also a typedef for *long* and represents an offset from some position in the file (the starting position is given by the *whence* argument to *lseek*).

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)
// filefd and devicefd 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).

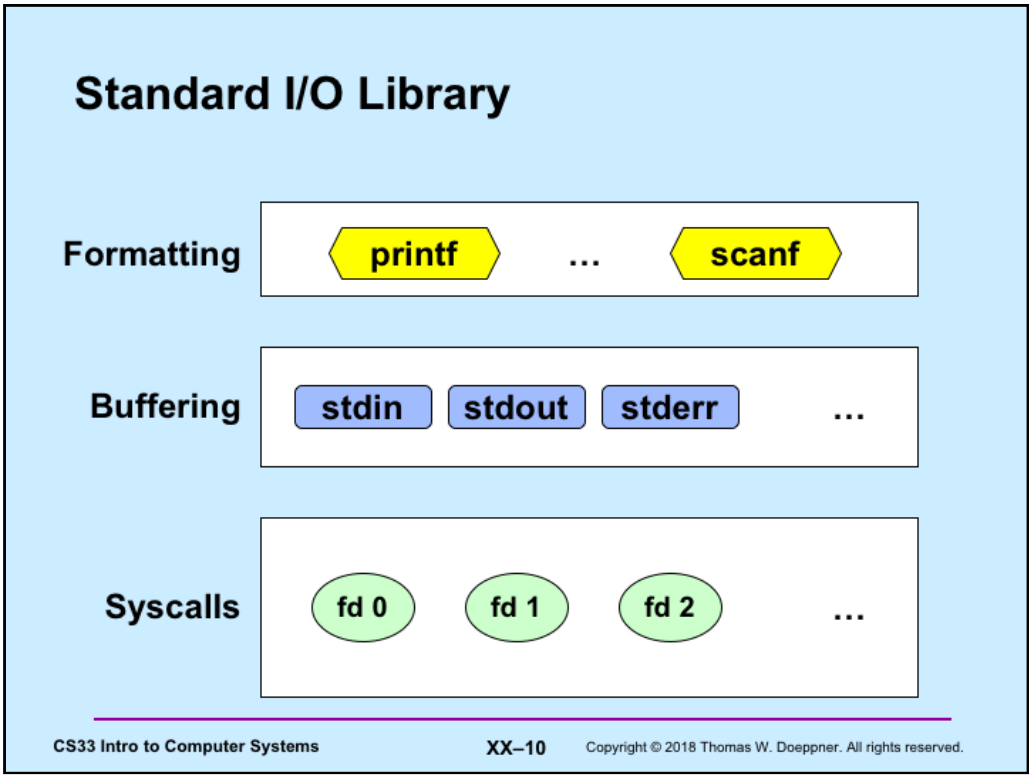
Note that the *open* system call returns an integer called a *file descriptor,* used in subsequent system calls to refer to the file.

Standard 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 set up by default to refer to the current window.



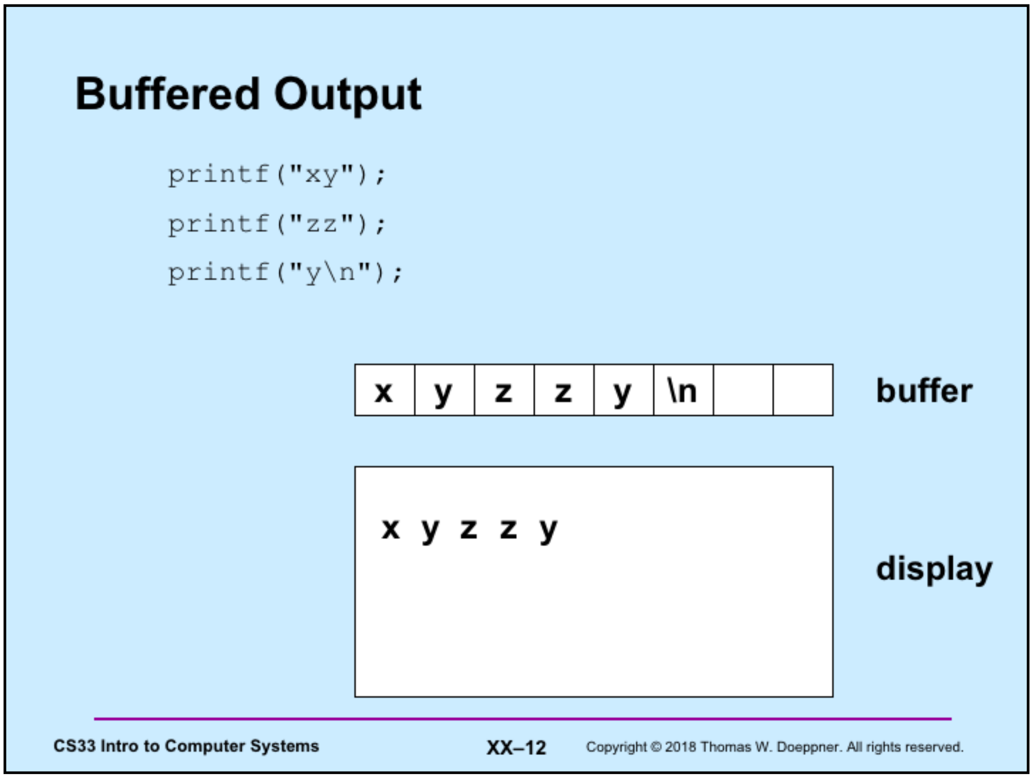
C programs often do I/O via the standard I/O library (known as stdio), which provides both buffering and formatting.

Standard I/O

```
FILE *stdin;          // declared in stdio.h
FILE *stdout;         // declared in stdio.h
FILE *stderr;         // declared in stdio.h

scanf("%d", &in);     // read via f.d. 0
printf("%d\n", in);   // write via f.d. 1
fprintf(stderr, "there was an error\n");
     // write via f.d. 2
```


The *streams* stdin, stdout, and stderr are automatically set up to refer to buffer data from/to file descriptors 0, 1, and 2, respectively.



The *stdout* stream is buffered. This means that characters written to *stdout* are copied into a buffer. Only when either a newline is output or the capacity of the buffer is reached are the characters actually written to the display (via a call to *write*). The reason for doing things this way is to reduce the number of (relatively expensive) calls to write.

Unbuffered Output

```
fprintf(stderr, "xy");
fprintf(stderr, "zz");
fprintf(stderr, "y\n");
```

The *stderr* stream is not buffered. Thus characters output to it are immediately written to the display.

A Program

```
int main(int argc, char *argv[]) {
  if (argc != 2) {
    fprintf(stderr, "Usage: echon reps\n");
    exit(1);
  }
  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);
}
```


The *fgets* function reads from the file stream given by its third argument and puts the data read into the buffer pointed to by its first argument. It stops reading data immediately after reading in a '\n' or after reading the number of bytes given as its second argument, whichever comes first. Note that the '\n' is copied into the buffer.

From the Shell ...

```
$ echon 1
```

- ***stdout*** **(fd 1) and *stderr* (fd 2) go to the display**
- ***stdin*** **(fd 0) 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

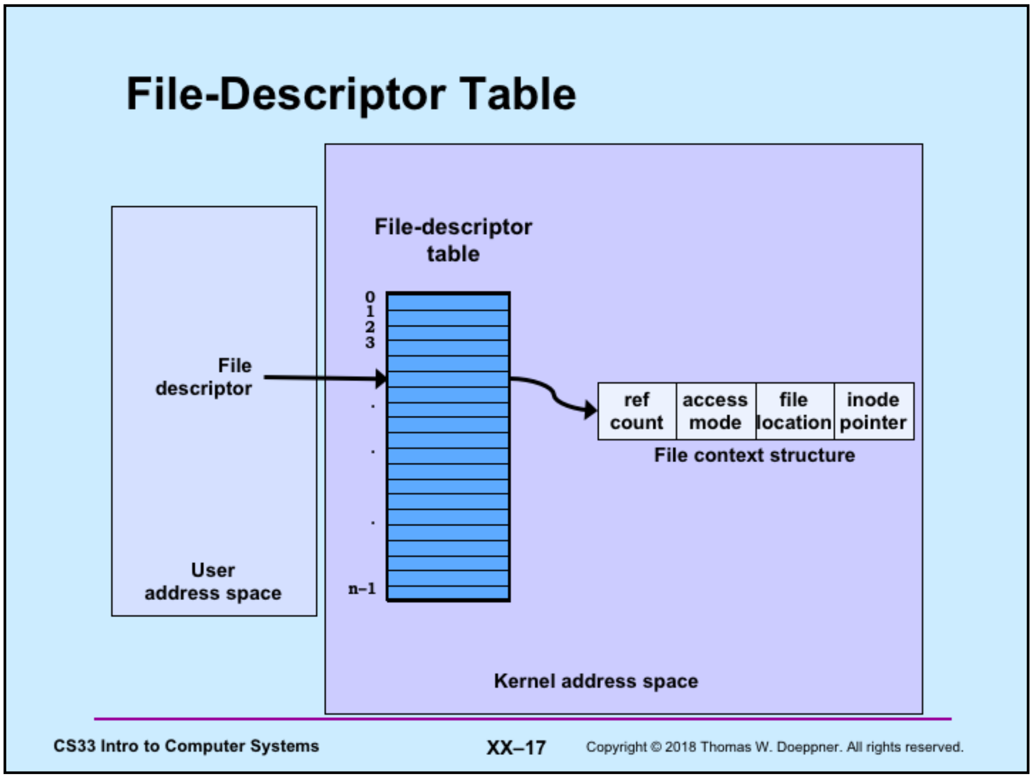
```
if (fork() == 0) {
   /* set up file descriptor 1 in the child process */
   close(1);
   if (open("/home/twd/Output", O_WRONLY) == -1) {
      perror("/home/twd/Output");
      exit(1);
   }
   char *argv[] = {"echon", "2", NULL};
   execv("/home/twd/bin/echon", argv);
   exit(1);
}

/* parent continues here */

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

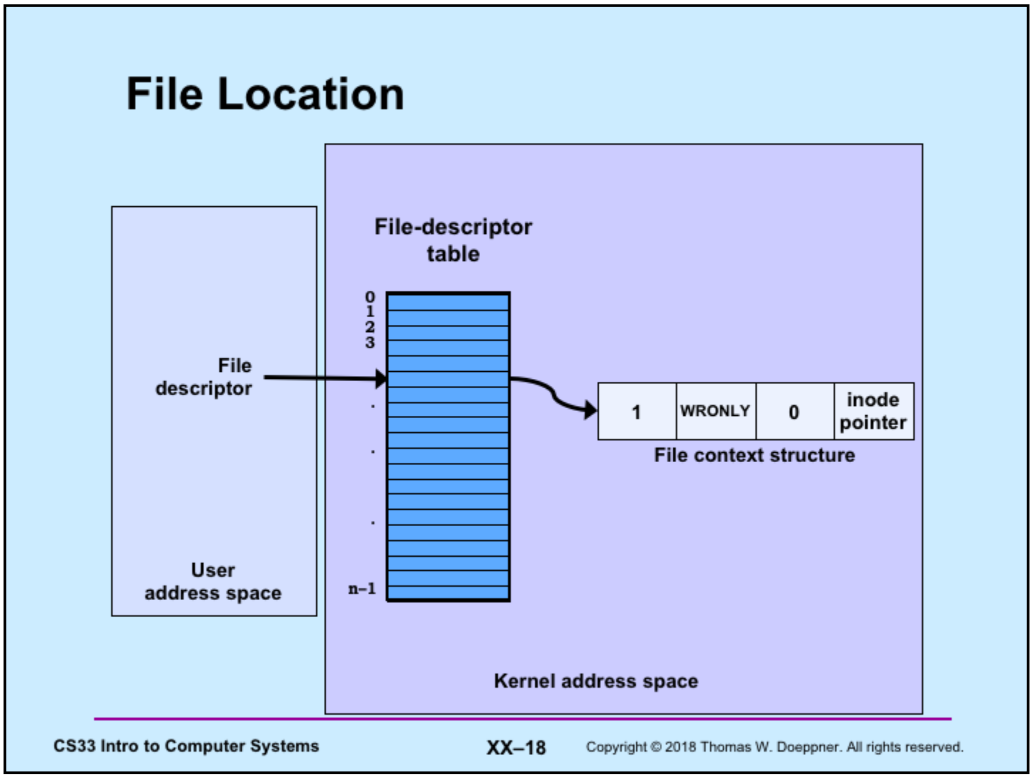

Here we arrange so that file descriptor 1 (standard output) refers to /home/twd/Output. As we discuss soon, if open succeeds, the file descriptor it assigns is the lowest-numbered one available. Thus if file descriptors 0, 1, and 2 are unavailable (because they correspond to standard input, standard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

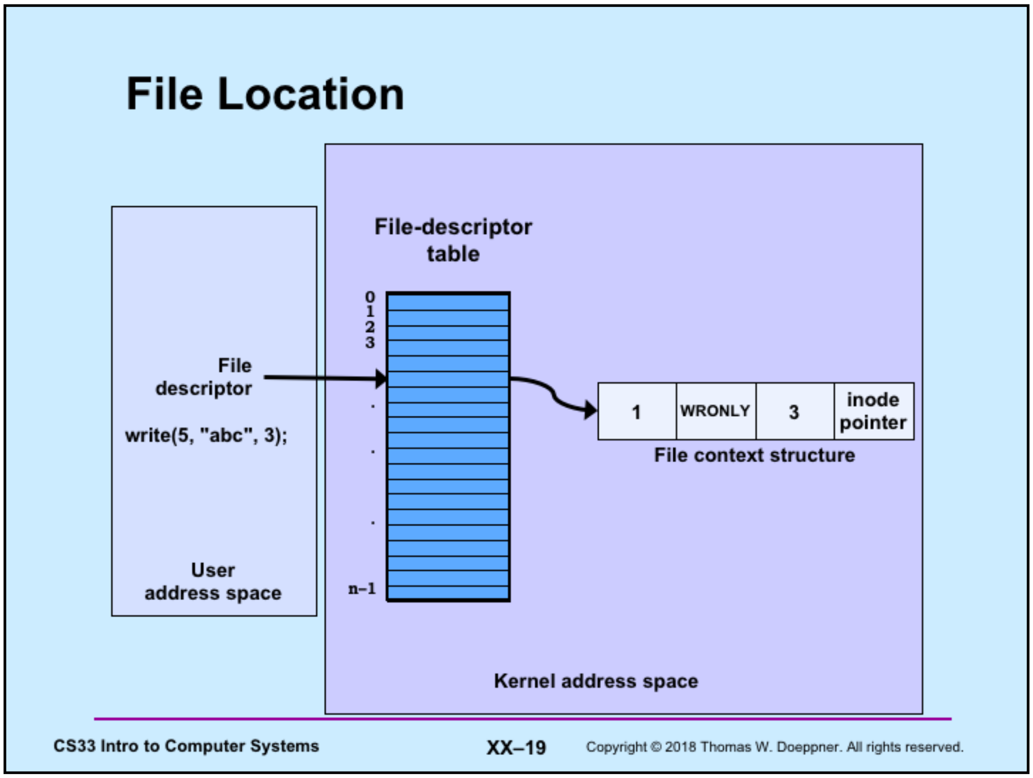


The *file-descriptor table* resides in the operating-system kernel; there's one for each process. Its entries are indexed by file descriptors; thus file descriptor 0 refers to the first entry, file descriptor 1 refers to the second entry, etc. Each entry in the table refers to a *file context structure*, as shown in the slide. This contains:

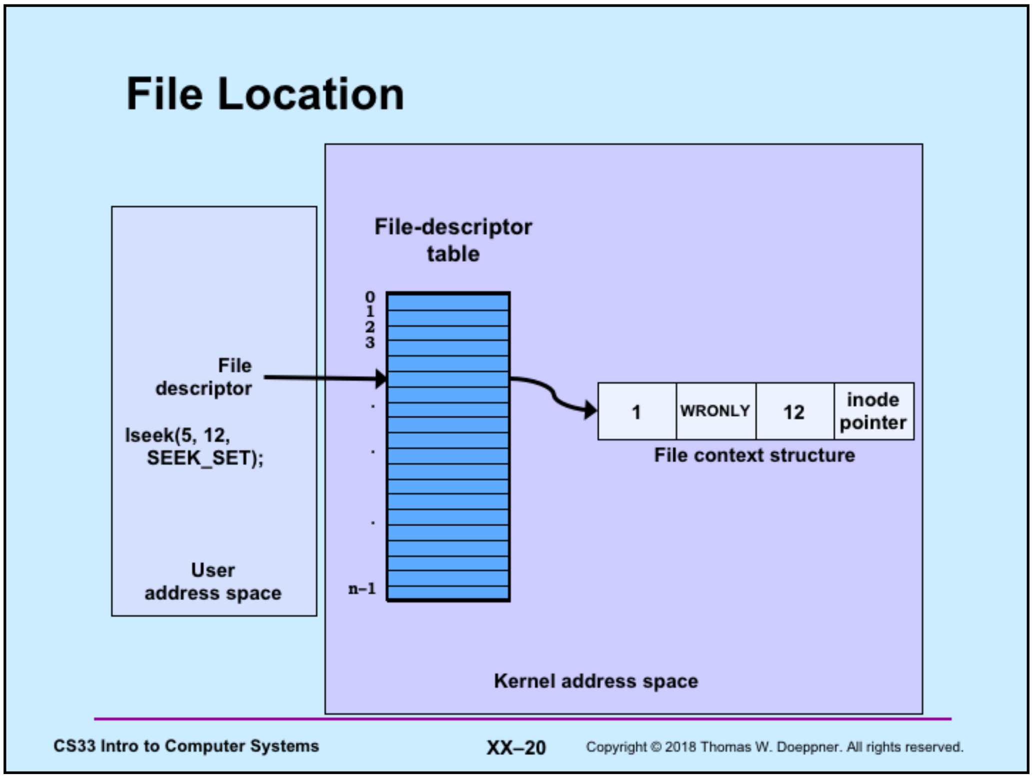
- a *reference count*, whose use we will see shortly
- an *access mode*, which specifies how the file was opened and thus how the process is using the file (e.g., read-only or read-write)
- the *file location*, which is the byte offset into the file where the next operation will take place
- the *inode pointer*, which is a data structure the OS provides for each file providing detailed information about the file, including where it is on disk. It normally resides on disk, but his brought into kernel memory when needed



The file-location field in the context structure indicates the offset into the file at which the next read or write operation will take place. It's normally set to 0 by OS when the file is opened (one can also have it set to the offset of the end of the file).



After reading or writing n bytes to a file, its file-location value is incremented by n. Thus, by default, I/O to files is sequential.



One can set the file location by using the lseek system call. Setting it will affect where the next read or write takes place. If the third argument is SEEK_SET, the offset given in the second argument is treated as an offset from the beginning of the file. If it's SEEK_CUR, it's treated as an offset from the current position in the file. If it's SEEK_END, it's treated as an offset from the end of the file.

If one sets the offset to well beyond the end of the file, leaving a "gap", this gap, when read, is treated as if it contains zeroes.

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 *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, NULL};
   execv("/home/twd/bin/echon", argv);
   exit(1);
}
/* parent continues here */
```

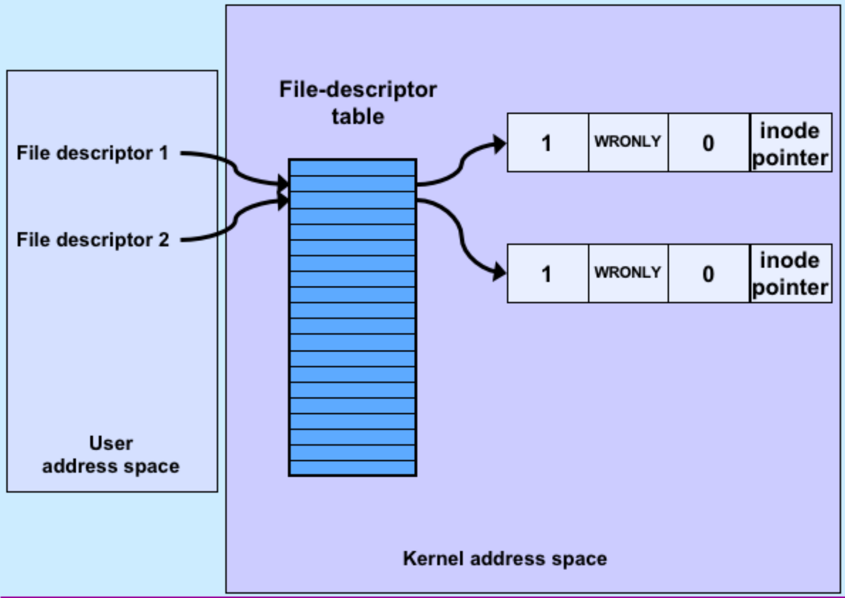

This redirects both standard output and standard error to be the file /home/twd/Output.

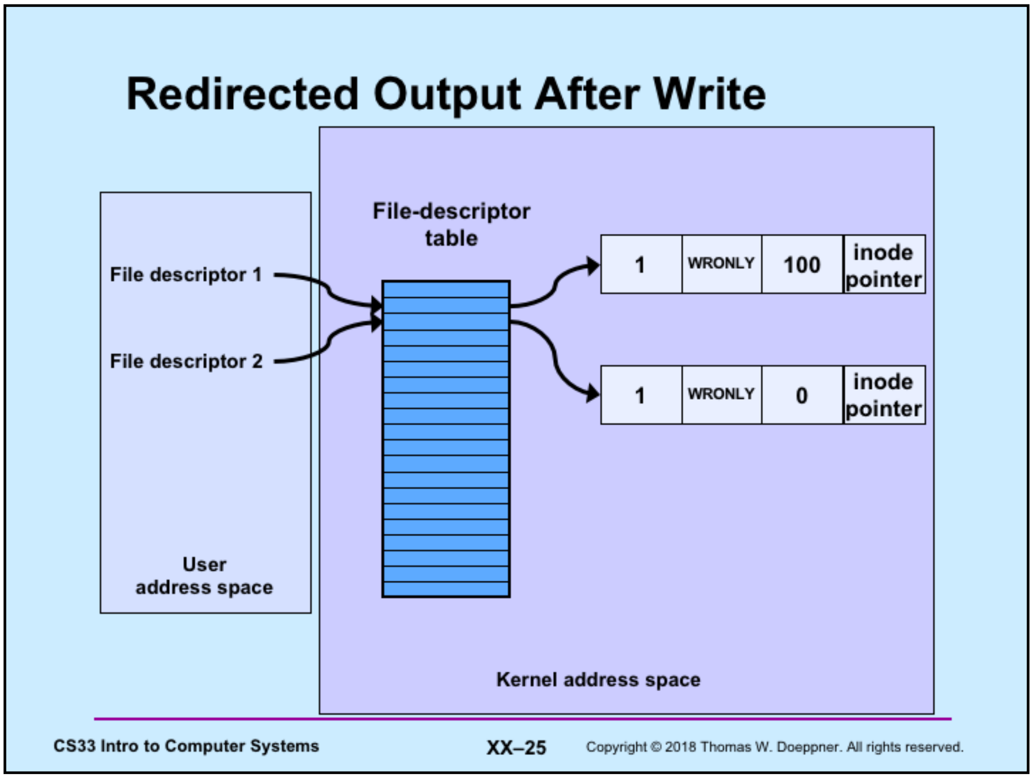
From the Shell ...

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

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

Redirected Output





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.

Quiz 1

- **Suppose we run**

  `% echon 3 >Output 2>Output`

- **The input line is**

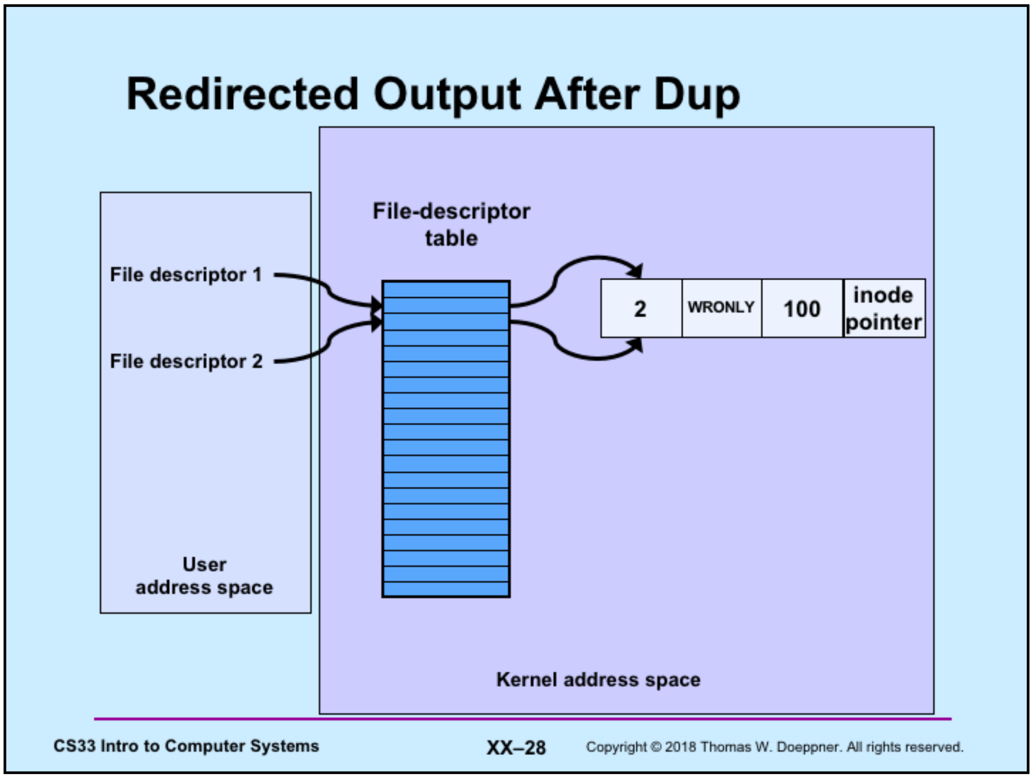
  `X`

- **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
```

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 */
```



Here we have one file context 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**

– **with input “X\n” it now produces in Output:**

```
reps too large, reduced to 2\nX\nX\n
```

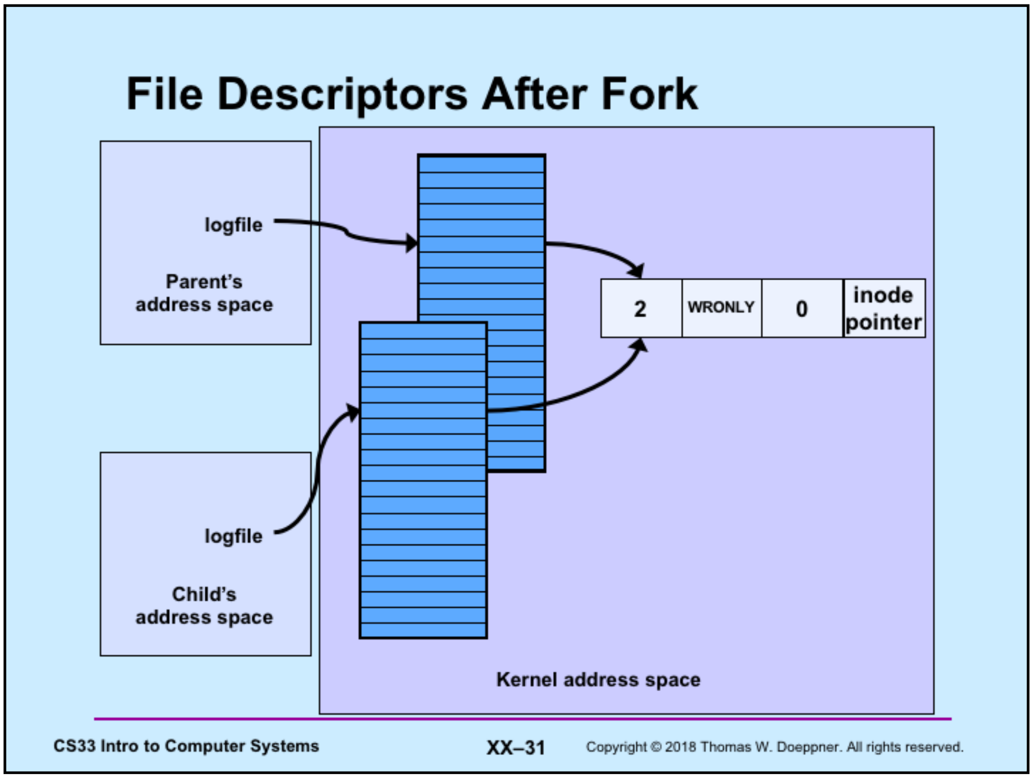

Fork and File Descriptors

```
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 log 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.



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);
  }
  fprintf(stderr, "Parent");
}
```

Suppose the program is run as:

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

What is the final content of file? (Assume writes are “atomic”.)

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

Unix guarantees that writes are *atomic*, which means they effectively happen instantaneously. Thus if two occur at about the same time, the effect is as if one completes before the other starts.