

# CS 33

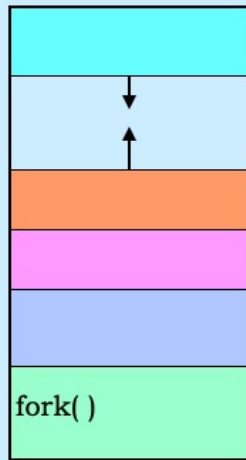
## Architecture and the OS (2)

# Creating Your Own Processes



```
#include <unistd.h>
int main( ) {
    pid_t pid;
    if ((pid = fork()) == 0) {
        /* new process starts
           running here */
    }
    /* old process continues
       here */
}
```

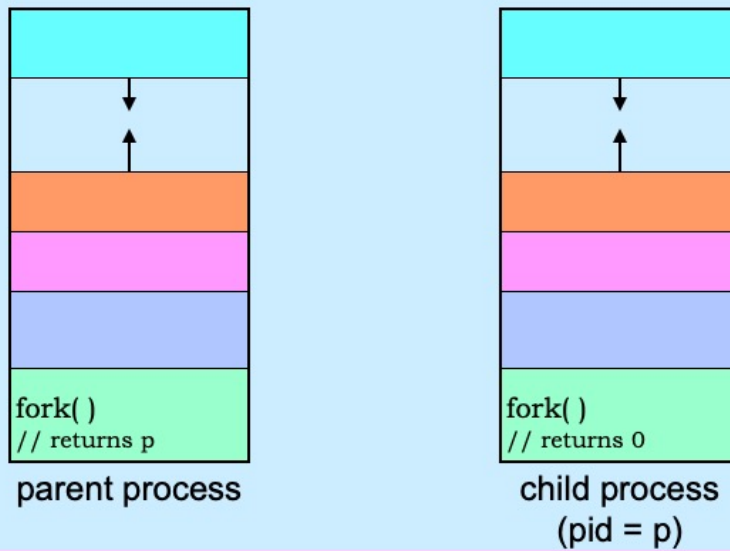
## Creating a Process: Before



parent process

The only way to create a new process is to use the **fork** system call.

## Creating a Process: After



By executing **fork** the parent process creates an almost exact clone of itself that we call the child process. This new process executes the same text as its parent, but contains a copy of the data and a copy of the stack. This copying of the parent to create the child can be very time-consuming if done naively. Some tricks are employed to make it much less so.

Fork is a very unusual system call: one thread of control flows into it but two threads of control flow out of it, each in a separate address space. From the parent's point of view, fork does very little: nothing happens to the parent except that fork returns the process ID (PID — an integer) of the new process. The new process starts off life by returning from fork, which it sees as returning a zero.

## Quiz 1

The following program

- a) runs forever
- b) terminates quickly

```
int flag;  
int main() {  
    while (flag == 0) {  
        if (fork() == 0) {  
            // in child process  
            flag = 1;  
            exit(0); // causes process to terminate  
        }  
    }  
}
```

## Process IDs

```
int main( ) {  
    pid_t pid;  
    pid_t ParentPid = getpid();  
  
    if ((pid = fork()) == 0) {  
        printf("%d, %d, %d\n",  
               pid, ParentPid, getpid());  
        return 0;  
    }  
    printf("%d, %d, %d\n",  
           pid, ParentPid, getpid());  
    return 0;  
}
```

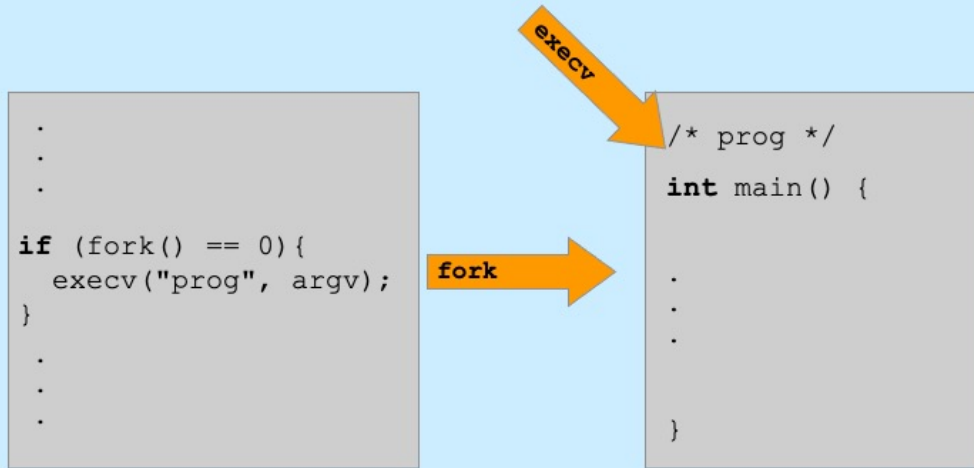
parent prints:  
27355, 27342, 27342

child prints:  
0, 27342, 27355

The **getpid** function returns the caller's process ID.

The parent process executes the second **printf**; the child process executes the first **printf**.

## Putting Programs into Processes



# Exec

- Family of related system functions

- we concentrate on one:

- » `execv(program, argv)`

```
char *argv[] = {"MyProg", "12", (void *)0};  
if (fork() == 0) {  
    execv("/MyProg", argv);  
}
```

First "real"  
argument

End of  
list

Name of the file that  
contains the program

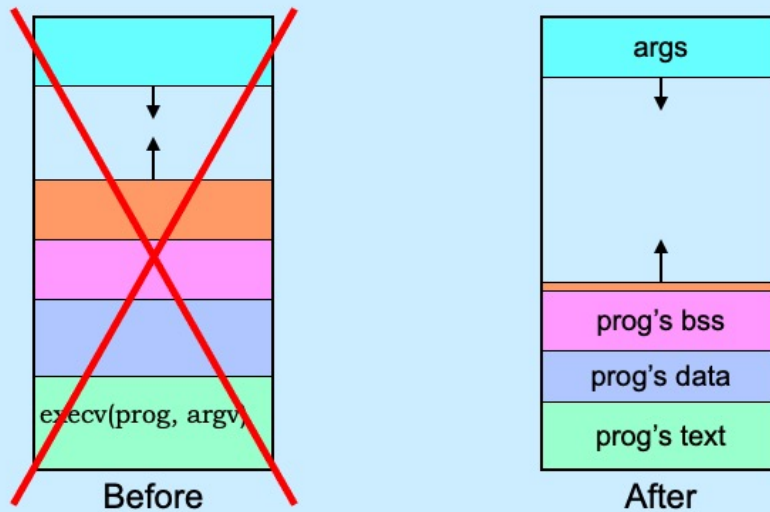
`argv[0]` is the name  
of the program

We will use the convention that the name of the program, as given in `argv[0]` is the last component of the file's pathname.

Note that a null pointer, termed a **sentinel**, is used to indicate the end of the list of arguments.



## Loading a New Image



Most of the time the purpose of creating a new process is to run a new (i.e., different) program. Once a new process has been created, it can use one of the *exec* system calls to load a new program image into itself, replacing the prior contents of the process's address space. Exec is passed the name of a file containing an executable program image. The previous text region of the process is replaced with the text of the program image. The data, BSS and dynamic areas of the process are “thrown away” and replaced with the data and BSS of the program image. The contents of the process's stack are replaced with the arguments that are passed to the main procedure of the program.

## A Random Program ...

```
int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: random count\n");
        exit(1);
    }
    int stop = atoi(argv[1]);
    for (int i = 0; i < stop; i++)
        printf("%d\n", rand());
    return 0;
}
```

The argument **argv** is what was provided to **execv**. The argument **argc** is the number of elements of **argv** (i.e., the number of arguments, including **argv[0]**).

## Passing It Arguments

- **From the shell**

```
$ random 12
```

- **From a C program**

```
if (fork() == 0) {  
    char *argv[] = {"random", "12", (void *)0};  
    execv("./random", argv);  
}
```

## Quiz 2

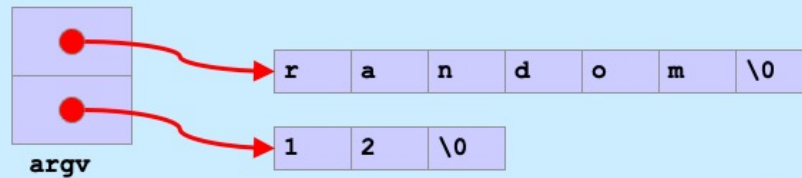
```
if (fork() == 0) {  
    char *argv[] = {"random", "12", (void *)0};  
    execv("./random", argv);  
    printf("random done\n");  
}
```

The *printf* statement will be executed

- a) only if `execv` fails
- b) only if `execv` succeeds
- c) always

## Receiving Arguments

```
int main(int argc, char *argv[]) {  
    if (argc != 2) {  
        fprintf(stderr, "Usage: random count\n");  
        exit(1);  
    }  
    int stop = atoi(argv[1]);  
    for (int i = 0; i < stop; i++)  
        printf("%d\n", rand());  
  
    return 0;  
}
```



Note that `argv[0]` is the name by which the program is invoked. `argv[1]` is the first “real” argument. In this program, `argv[2]` will contain the NULL pointer (0). `argc` is two, indicating two arguments (`argv[0]` and `argv[1]`).

## Not So Fast ...

- How does the shell invoke your program?

```
if (fork() == 0) {  
    char *argv = {"random", "12", (void *)0};  
    execv("./random", argv);  
}  
/* what does the shell do here??? */
```

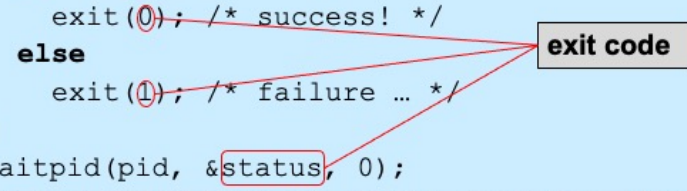
## Wait

```
#include <unistd.h>
#include <sys/wait.h>
...
pid_t pid;
int status;
...
if ((pid = fork()) == 0) {
    char *argv[] = {"random", "12", (void *)0};
    execv("./random", argv);
}
waitpid(pid, &status, 0);
```

There's a variant of **waitpid**, called **wait**, that waits for any child of the current process to terminate.

## Exit

```
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
int main( ) {
    pid_t pid;
    int status;
    if ((pid = fork()) == 0) {
        if (do_work() == 1)
            exit(0); /* success! */
        else
            exit(1); /* failure ... */
    }
    waitpid(pid, &status, 0);
    /* low-order byte of status contains exit code.
       WEXITSTATUS(status) extracts it */
}
```



The exit code is used to indicate problems that might have occurred while running a program. The convention is that an exit code of 0 means success; other values indicate some sort of error. Note that if the main function returns, it returns to code that calls exit; thus, returning from main is equivalent to calling **exit**. The argument passed to **exit** in this case is the value returned by main.



## Shell: To Wait or Not To Wait ...

```
$ who
```

```
    if ((pid = fork()) == 0) {  
        char *argv[] = {"who", 0};  
        execv("who", argv);  
    }  
    waitpid(pid, &status, 0);  
    ...
```

```
$ who &
```

```
    if ((pid = fork()) == 0) {  
        char *argv[] = {"who", 0};  
        execv("who", argv);  
    }  
    ...
```

## System Calls

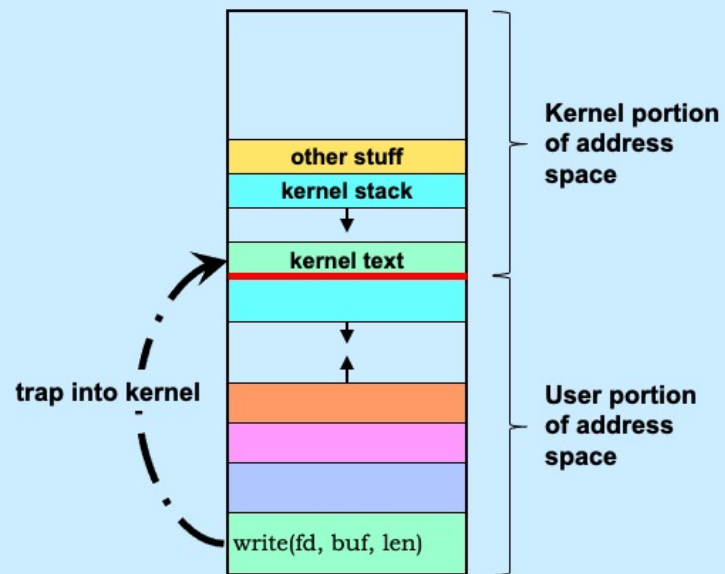
- **Sole direct interface between user and kernel**
- **Implemented as library function that execute *trap* instructions to enter kernel**
- **Errors indicated by returns of  $-1$ ; error code is in global variable *errno***

```
if (write(fd, buffer, bufsize) == -1) {  
    // error!  
    printf("error %d\n", errno);  
    // see perror  
}
```

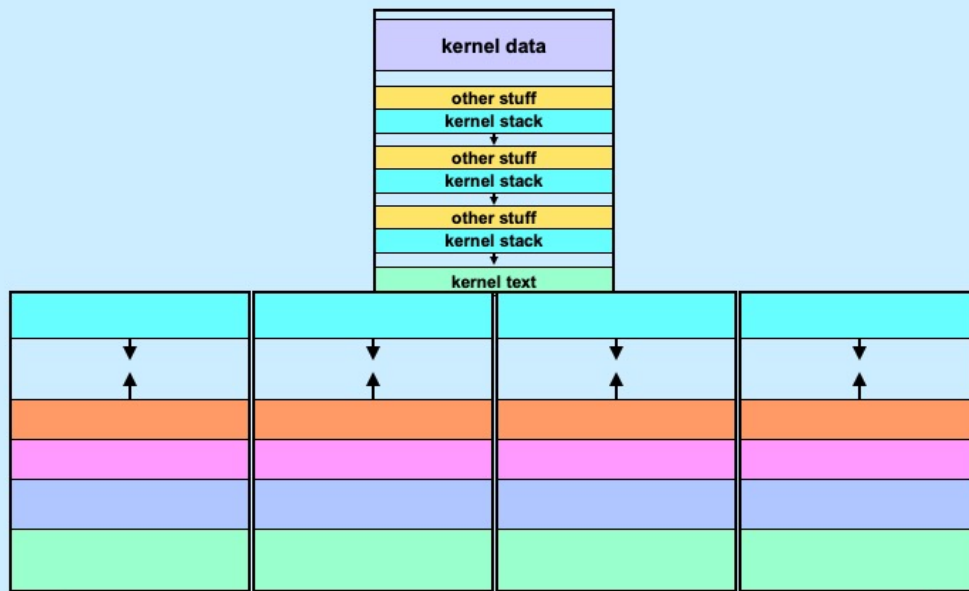
System calls, such as **fork**, **execv**, **read**, **write**, etc., are the only means for application programs to communicate directly with the kernel: they form an API (application program interface) to the kernel. When a program calls such a function, it is actually placing a call to a function in a system library. The body of this function contains a hardware-specific trap instruction that transfers control and some parameters to the kernel. On return to the library function, the kernel provides an indication of whether or not there was an error and what the error was. The error indication is passed back to the original caller via the functional return value of the library function: If there was an error, the function returns  $-1$  and a positive-integer code identifying the error is stored in the global variable **errno**. Rather than simply print this code out, as shown in the slide, one might instead print out an informative error message. This can be done via the **perror** function.

The “hardware-specific trap instruction” is (or used to be) the “int” (interrupt) instruction on the x86. However, this instruction is now considered too expensive for such performance-critical operations as system calls. A new facility, known as “syscall/sysret” was introduced with the Pentium II processors (in 1997) and has been used by operating systems (including Windows and Linux) ever since. Its description is beyond the scope of this course.

# System Calls



## Multiple Processes



Each process has its own user address space, but there's a single kernel address space. It contains context information for each user process, including the stacks used by each process when executing system calls.

# 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.

CS Department computers run Debian Linux (and thus weren't affected by shellshock).

Our examples use bash syntax.

# 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**
- **Files are permanent**

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.) Another important property of files is permanence: they continue to exist until explicitly deleted.

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. We discuss this in more detail in the next lecture.

## 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**).

## 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 before a process starts. File descriptor 0 refers to input (the keyboard, by default). Descriptors 1 and 2 are for output: normal output goes to file descriptor 1, error messages go to file descriptor 2. By default, this output goes to the current window.

We'll soon see a way to print more informative error messages than the one given here.

## Standard I/O Library

Formatting

**printf**

...

**scanf**

Buffering

**stdin**

**stdout**

**stderr**

...

Syscalls

fd 0

fd 1

fd 2

...

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

## Standard I/O

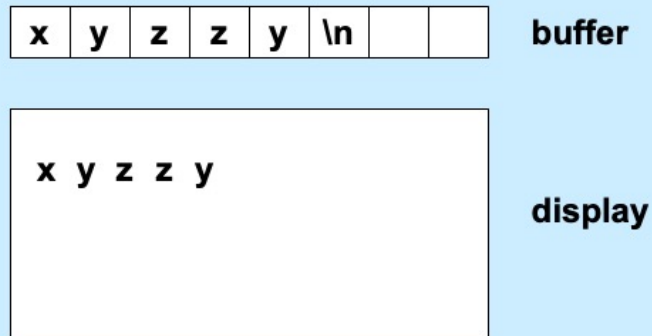
```
FILE *stdin;           // declared in stdio.h
FILE *stdout;          // declared in stdio.h
FILE *stderr;          // declared in stdio.h
```

```
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 data from/to file descriptors 0, 1, and 2, respectively.

## Buffered Output

```
printf("xy");  
printf("zz");  
printf("y\n");
```



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");
```

**x y z z y**

**display**

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. (**fgets** is what programs should use rather than **gets**, as we saw when we discussed buffer-overflow attacks.) The **fputs** function writes its first argument to the file stream given by the second argument.

## From the Shell ...

```
$ echo 1
```

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

```
$ echo 1 > Output
```

- ***stdout*** goes to the file “Output” in the current directory
- ***stderr*** goes to the display
- ***stdin*** comes from the keyboard

```
$ echo 1 < Input
```

- ***stdin*** comes from the file “Input” in the current directory

Our shell examples are all in bash. The slide shows how, via the shell, we can change what ***stdout*** and ***stdin*** are. We'll soon see how we can do so for ***stderr***.

## Redirecting Stdout in C

```
if ((pid = 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 */

waitpid(pid, 0, 0);    // wait for child to terminate
```

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.

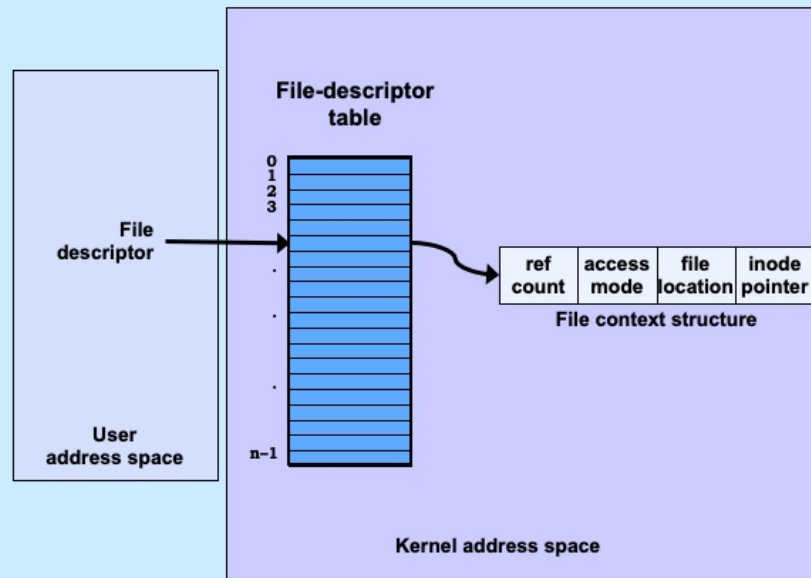
By setting the second argument of **waitpid** to 0, we're ignoring the exit status.

Note the use of **perror**. It's declared in `stdio.h` and is used for printing error messages after a system call fails (returning -1). As we saw in the previous lecture, when a system call fails, in addition to returning -1 it puts the failure code in the global variable **errno**. The function **perror** uses the value in **errno** to index into an array of error messages and prints (to `stderr`) its argument followed by the text of the error message.

Note that it's used only for system calls, such as `open`, `close`, `read`, `write`, `fork`, and `execv`. It doesn't give correct results for functions that aren't system calls, such as `printf`. A function is a system call if its description is in section 2 of the online unix manual. Thus, for system calls, typing, for example, "man 2 open", results in a description of the open system call. Typing "man 2 printf" results in an error message, since `printf` is not a system call, but a function supplied by the `stdio` library.

In many cases typing "man <function\_name>" (without specifying a section number) gives you the correct man page for that function, but some function names are ambiguous. For example, `printf` is both a shell command (which is documented in section 1 of the unix manual) and a function in the `stdio` library (which is documented in section 3). To see the man page for the `stdio` library function `printf`, one should type "man 3 printf".

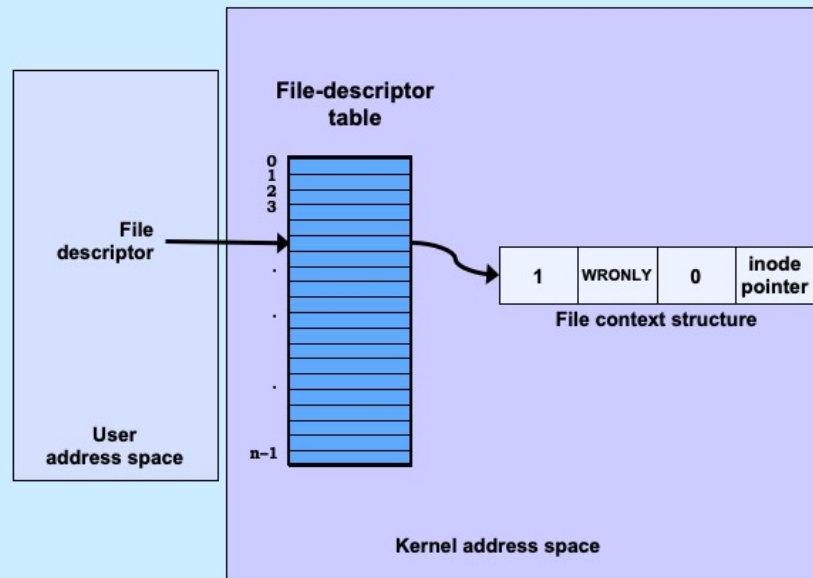
# File-Descriptor Table



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:

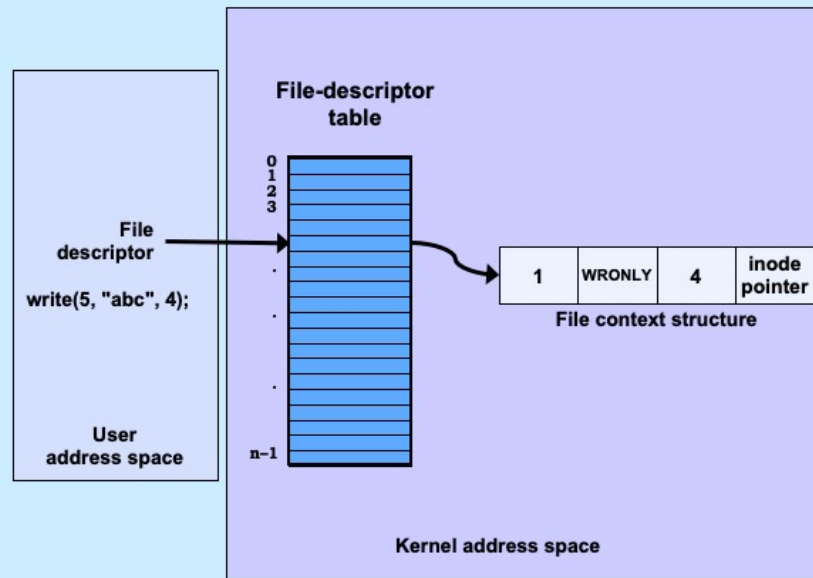
- a **reference count**, whose use we will see shortly
- an **access mode**, which specifies how the file was opened and thus how the process may use the file (e.g., read-only or read-write)
- the **file location**, which is the byte offset into the file where the next operation will take place
- the **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 is brought into kernel memory when needed

# File Location



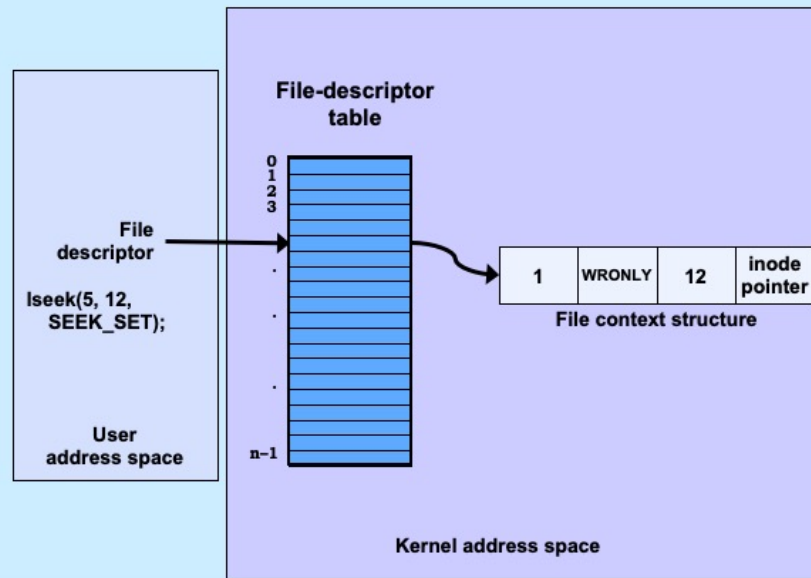
The file-location field in the context structure indicates the offset into the file at which the next read or write operation will take place. It's normally set to 0 by OS when the file is opened (one can also have it set to the offset of the end of the file by setting the `O_APPEND` flag in `open`).

## File Location



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.

## File Location



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 and then writes to the file at that position, 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`.

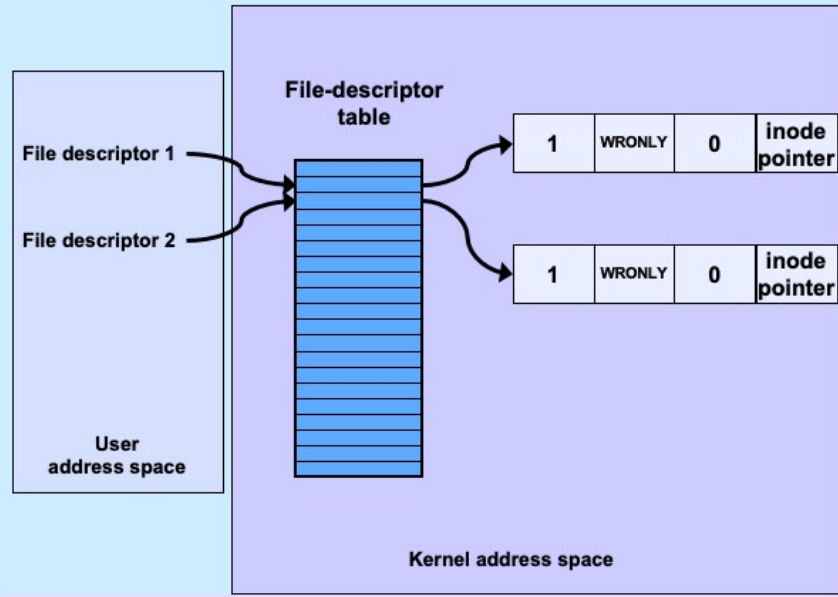
## From the Shell ...

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

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

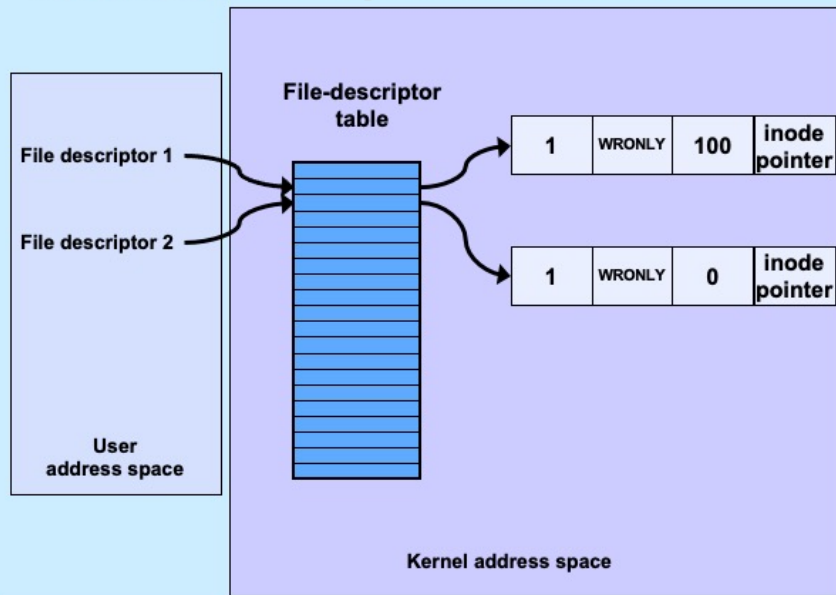
This is the syntax used in `bash` (which is how it was done on the Bourne shell). Other shells have different syntaxes for this.

## Redirected Output



After opening the Output file twice, the file-descriptor table appears as shown in the slide.

## Redirected Output After Write



There is a potential problem here. Since our file (/home/twd/Output) has been opened once for each file descriptor, when a write (in this case of 100 bytes) 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 3

- **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\nnreps too large, reduced to 2\n
- c) X\nX\n too large, reduced to 2\n

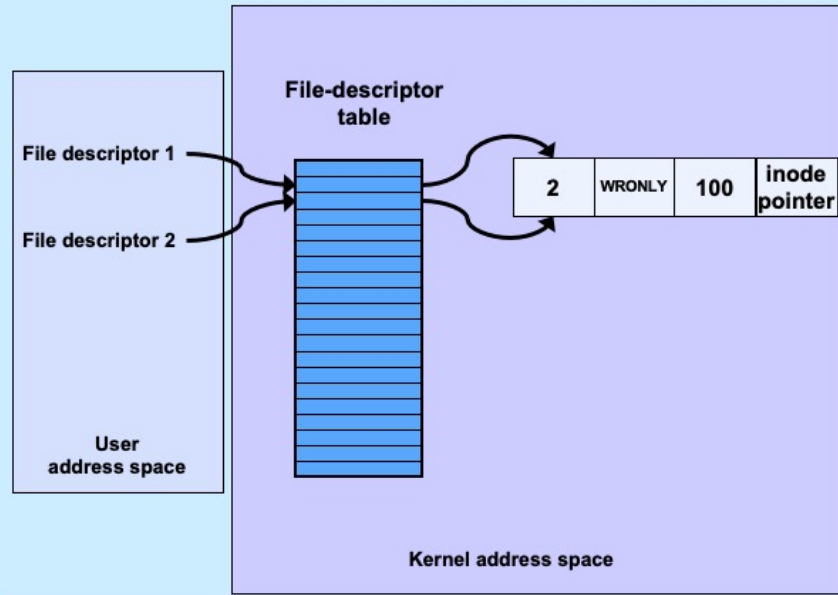
Note that the actual input consists of X followed by a newline character.

Recall that **echon** will first write "reps too large, reduced to 2" to file descriptor 2, then write "x\nx\n" to 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 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
```



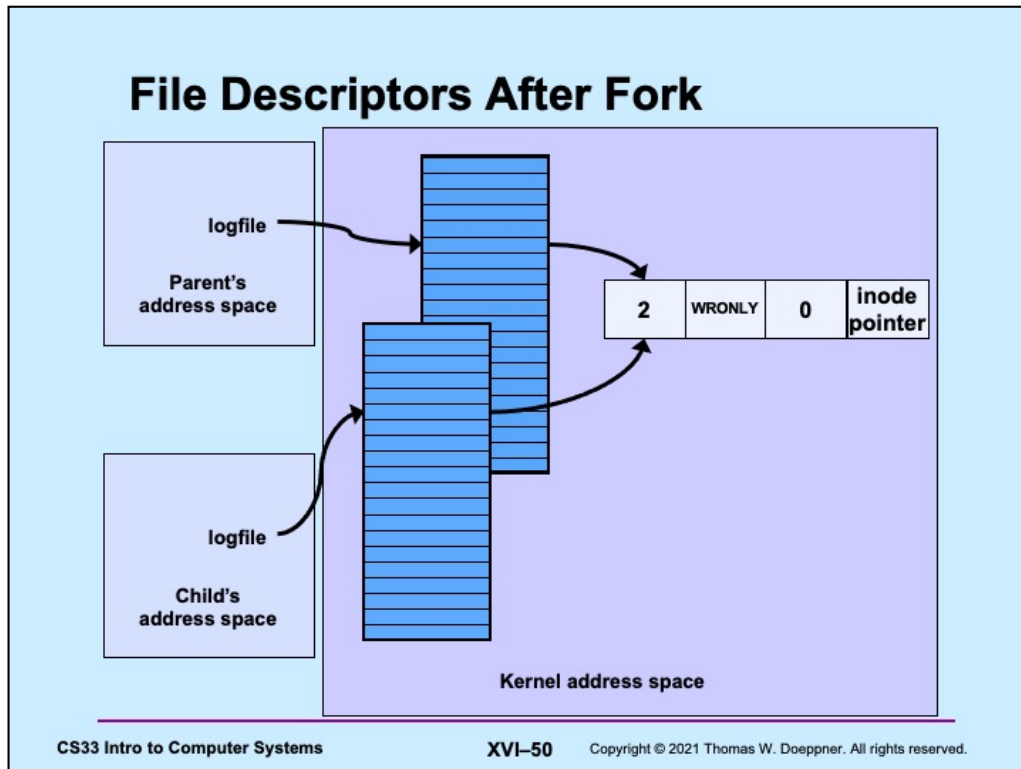
## Fork and File Descriptors

```
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 4

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

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.