CS 33

Files Part 2

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");
```

| x | y | z | z | y | \n | | | **buffer** |

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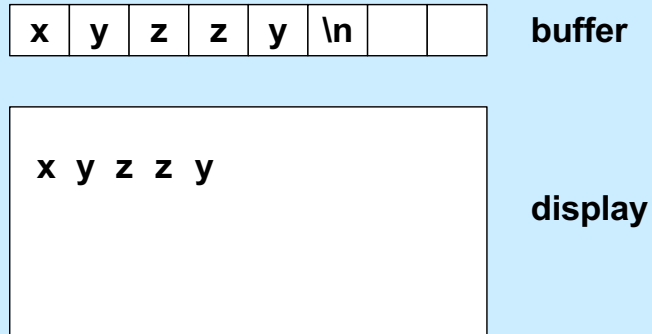
The **stdout** stream is buffered. This means that characters written to **stdout** are copied into a buffer. Only when either a newline is output or the capacity of the buffer is reached are the characters actually written to the display (via a call to **write**). The reason for doing things this way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

```c
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);
}
```

This is the code for the program **echon**, which we'll be using as an example in the upcoming slides.

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 ...

```
$ 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. 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

```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.

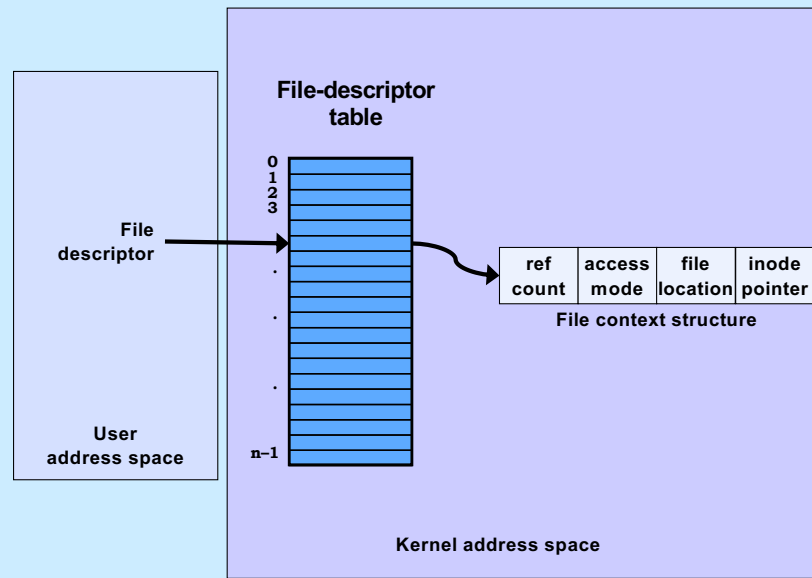By setting the second argument of **waitpid** to 0, we're ignoring the exit status.

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".
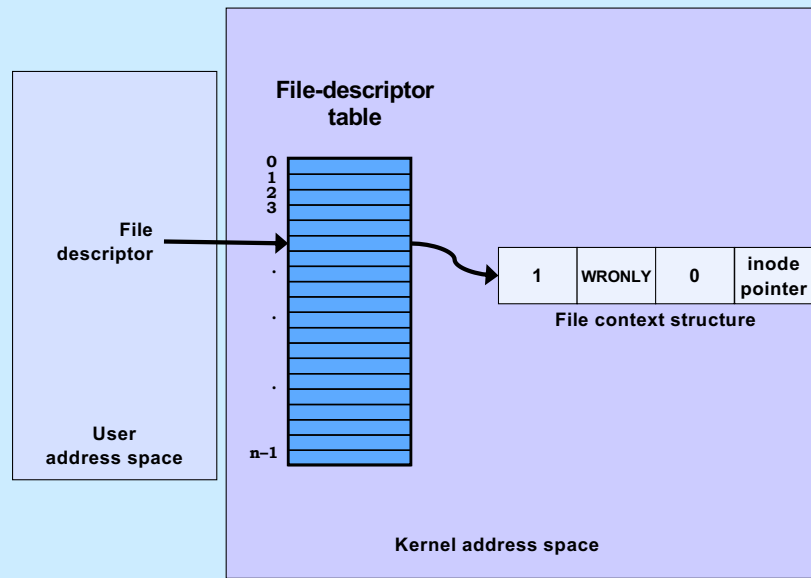
File-Descriptor Table

File-descriptor
table

0
1
2
3

File
descriptor

.

.

.

n−1

ref count	access mode	file location	inode pointer

File context structure

User
address space

Kernel address space

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:
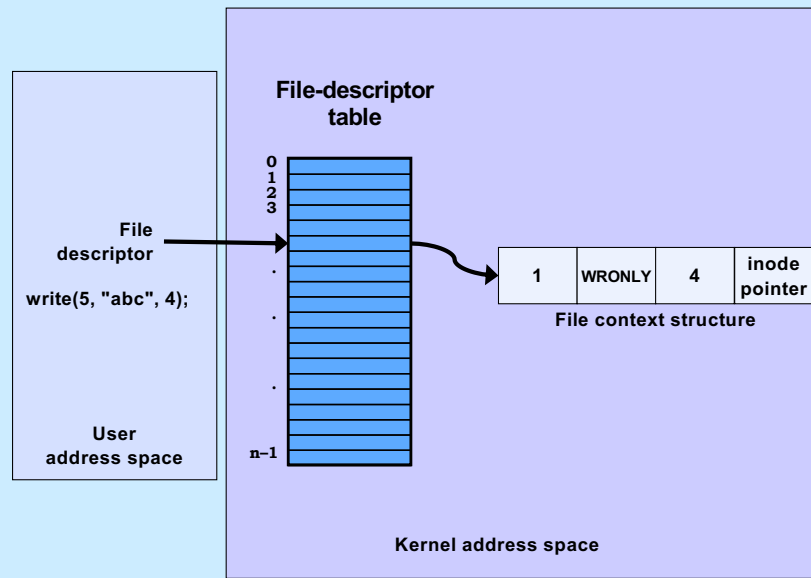
- a **reference count**, whose use we will see shortly

- 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 his brought into kernel memory when needed

File Location

File-descriptor table

```
0
1
2
3
.
.
.
n−1
```

File descriptor →

File context structure

1	WRONLY	0	inode pointer

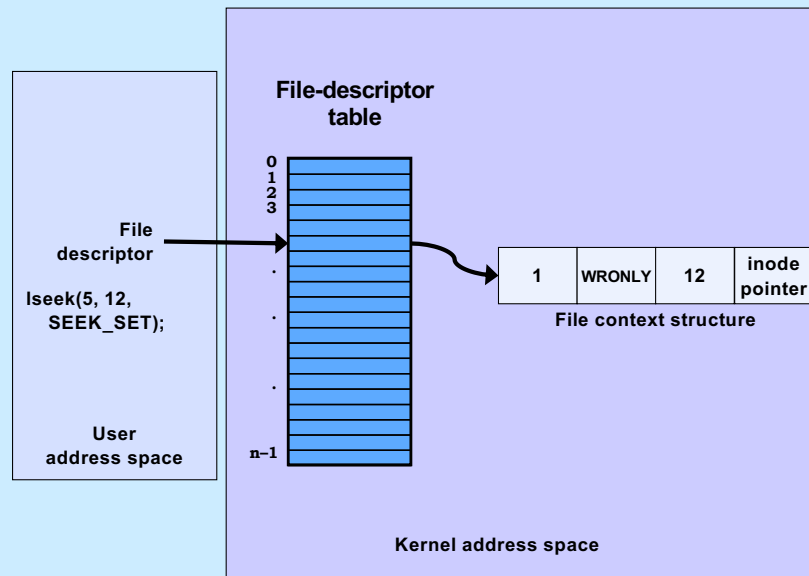**User address space**

Kernel address space

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

File-descriptor table

```
0
1
2
3
.
.
.
n–1
```

File
descriptor

write(5, "abc", 4);

1	WRONLY	4	inode pointer

File context structure

User address space

Kernel address space

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

File-descriptor table

0
1
2
3

File descriptor

lseek(5, 12, SEEK_SET);

User address space

n–1

| 1 | WRONLY | 12 | inode pointer |

File context structure

Kernel address space

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 */
```

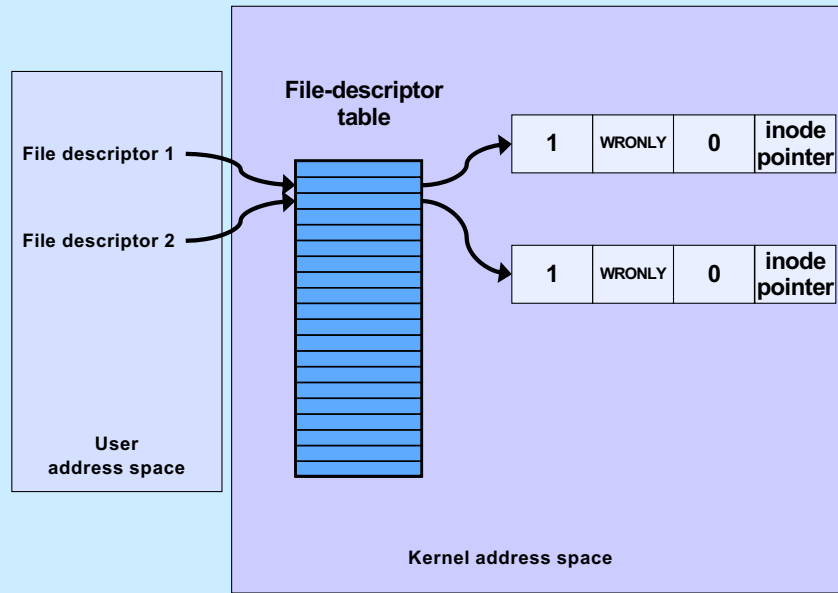This redirects both standard output and standard error to be the file /home/twd/Output.

From the Shell ...

```
$ echon 1 >Output 2>Output
```
   – **both stdout and stderr go to Output file**

CS33 Intro to Computer Systems   **XVII–15**

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

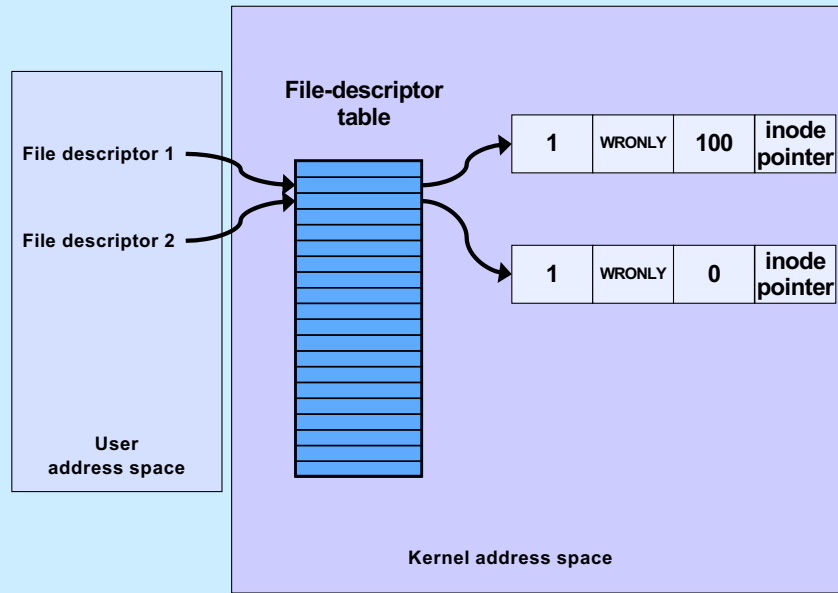After opening the Output file twice, the file-descriptor table appears as shown in the slide.

Redirected Output After Write

File-descriptor table

File descriptor 1

File descriptor 2

1	WRONLY	100	inode pointer

1	WRONLY	0	inode pointer

User address space

Kernel address space

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 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
  ```

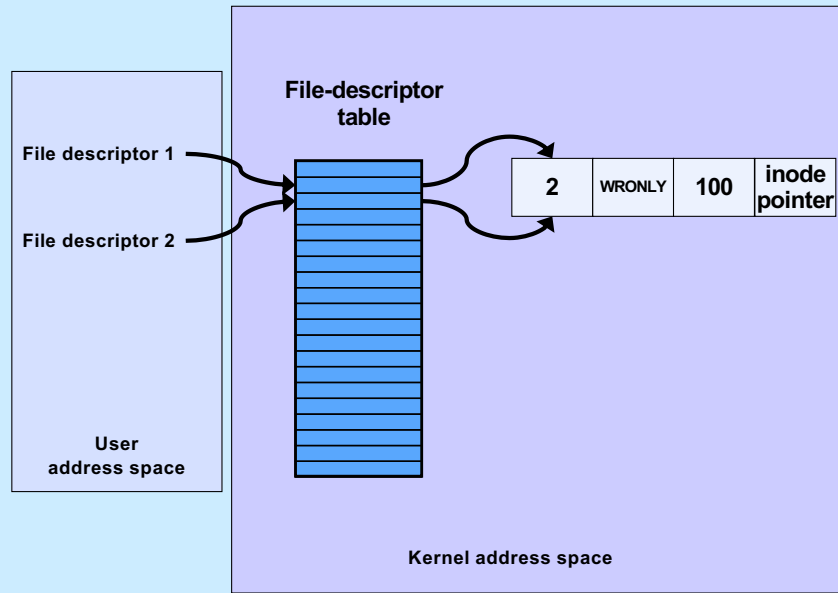Note that the actual input consists of X followed by a newline character.

Recall that **echon** first writes "reps too large, reduced to 2" to file descriptor 2, then writes "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 */
```

The **dup** system call returns a newly allocated file descriptor that refers to the same file context structure as does the file descriptor of its argument.

Redirected Output After Dup

File-descriptor table

File descriptor 1

File descriptor 2

| 2 | WRONLY | 100 | inode pointer |

User address space

Kernel address space

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
```
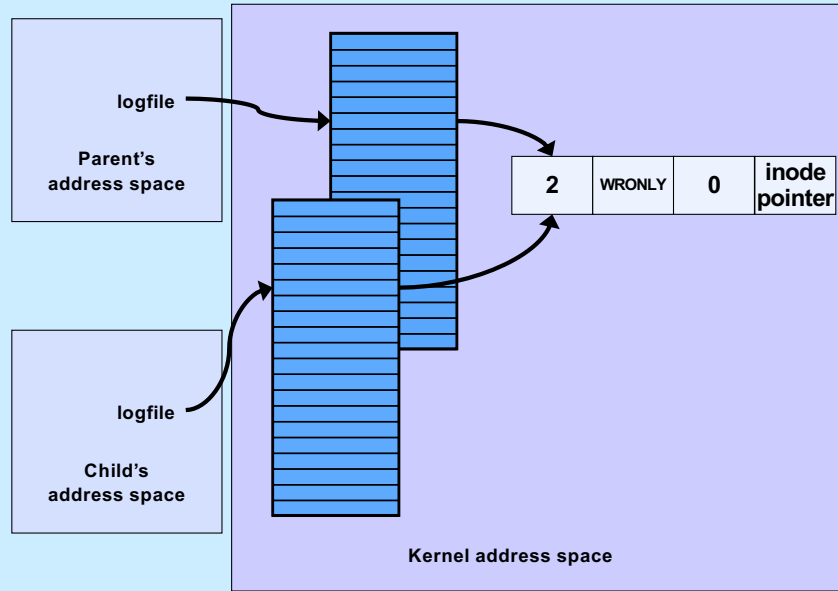
Fork and File Descriptors

```
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.

File Descriptors After Fork

logfile

Parent's address space

logfile

Child's address space

2	WRONLY	0	inode pointer

Kernel address space

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 "Childt" or "Parent"
   b) either "Child" or "Parent"
   c) either "ChildParent" or "ParentChild"

**XVII–24**

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.

**Directories**

unix | etc | home | pro | dev

passwd | motd

twd

unix | ...

slide1 | slide2
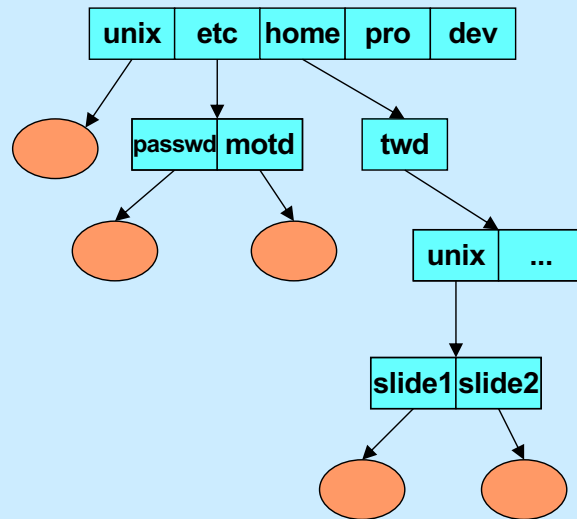
Here is a portion of a Unix directory tree. The ovals represent files, the rectangles represent directories (which are really just special cases of files).

# Directory Representation

| Component Name | Inode Number |
|:--------------:|:------------:|

**directory entry**

| | |
|:--:|:--:|
| . | 1 |
| .. | 1 |
| unix | 117 |
| etc | 4 |
| home | 18 |
| pro | 36 |
| dev | 93 |

A simple implementation of a directory consists of an array of pairs of **component name** and **inode number**, where the latter identifies the target file's *inode* to the operating system (an inode is data structure maintained by the operating system that represents a file). Note that every directory contains two special entries, "." and "..". The former refers to the directory itself, the latter to the directory's parent (in the case of the slide, the directory is the root directory and has no parent, thus its ".." entry is a special case that refers to the directory itself).

While this implementation of a directory was used in early file systems for Unix, it suffers from a number of practical problems (for example, it doesn't scale well for large directories). It provides a good model for the semantics of directory operations, but directory implementations on modern systems are more complicated than this (and are beyond the scope of this course).

## Hard Links

| unix | etc | home | pro | dev |

| twd |

| unix | ... |

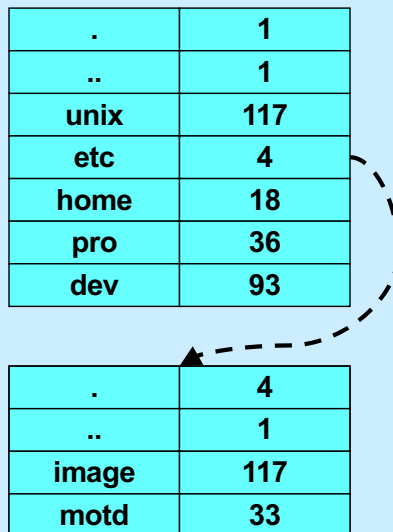| image | motd |

| slide1 | slide2 |
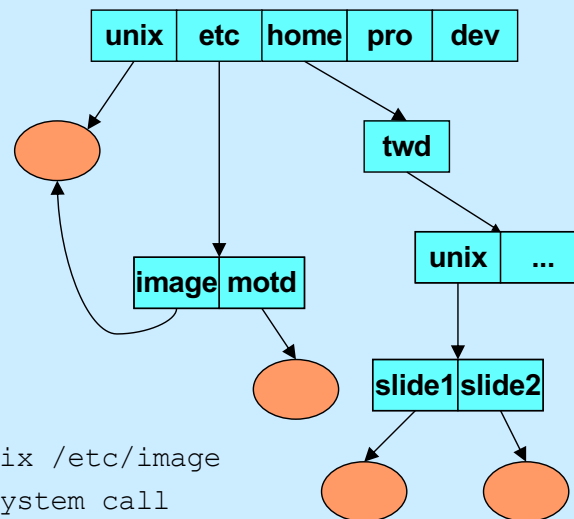
```
$ ln /unix /etc/image
# link system call
```

Here are two directory entries referring to the same file. This is done, via the shell, through the **ln** command which creates a (hard) link to its first argument, giving it the name specified by its second argument.

The shell's "ln" command is implemented using the **link** system call.

Directory Representation

.	1
..	1
unix	117
etc	4
home	18
pro	36
dev	93

| .	4
..	1
image	117
motd	33

Here are the (abbreviated) contents of both the **root** (/) and **/etc** directories, showing how **/unix** and **/etc/image** are the same file. Note that if the directory entry **/unix** is deleted (via the shell's "rm" command), the file (represented by inode 117) continues to exist, since there is still a directory entry referring to it. However, if **/etc/image** is also deleted, then the file has no more links and is removed. To implement this, the file's inode contains a link count, indicating the total number of directory entries that refer to it. A file is actually deleted only when its inode's link count reaches zero.

Note: suppose a file is open, i.e. is being used by some process, when its link count becomes zero. Rather than delete the file while the process is using it, the file will continue to exist until no process has it open. Thus the inode also contains a reference count indicating how many times it is open: in particular, how many system file table entries point to it. A file is deleted when and only when both the link count and this reference count become zero.

The shell's "rm" command is implemented using the **unlink** system call.

Note that **/etc/..** refers to the root directory.

Symbolic Links

```
% ln –s /unix /home/twd/mylink
% ln –s /home/twd /etc/twd
# symlink system call
```
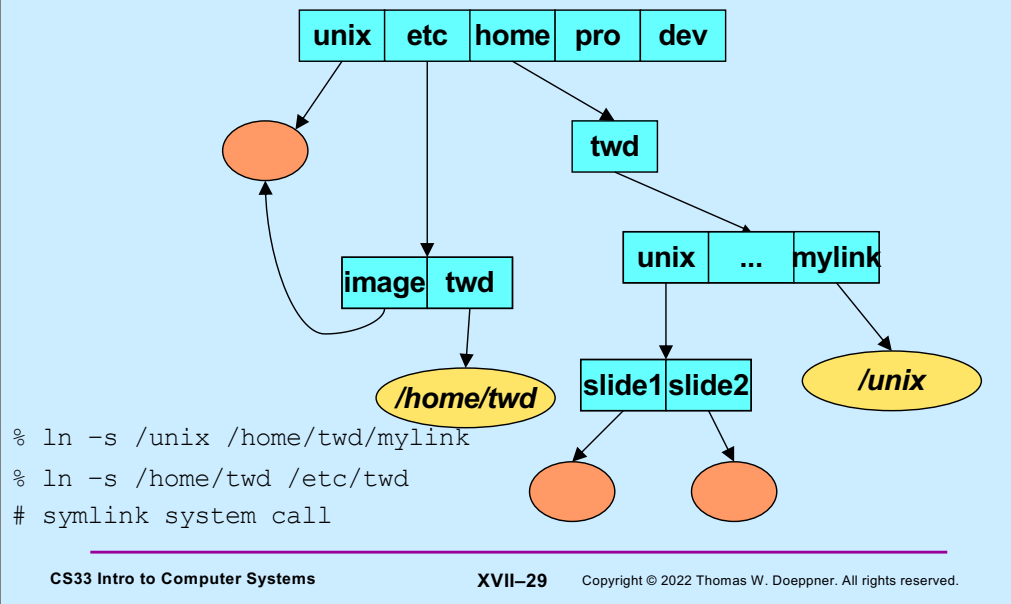
Differing from a hard link, a **symbolic link** (often called soft link) is a special kind of file containing the name of another file. When the kernel processes such a file, rather than simply retrieving its contents, it makes use of the contents by replacing the portion of the directory path that it has already followed with the contents of the soft-link file and then following the resulting path. Thus referencing **/home/twd/mylink** results in the same file as referencing **/unix**. Referencing **/etc/twd/unix/slide1** results in the same file as referencing **/home/twd/unix/slide1**.

The shell's "ln" command with the "-s" flag is implemented using the **symlink** system call.

Working Directory

- **Maintained in kernel for each process**
  - **paths not starting from "/" start with the working directory**
  - **changed by use of the *chdir* system call**
    - » ***cd* shell command**
  - **displayed (via shell) using "pwd"**
    - » **how is this done?**

The **working directory** is maintained in the kernel for each process. Whenever a process attempts to follow a path that doesn't start with "/", it starts at its working directory (rather than at "/").

Open

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *path, int options [, mode_t mode])
```

– **options**
- » **O_RDONLY**   **open for reading only**
- » **O_WRONLY**   **open for writing only**
- » **O_RDWR**   **open for reading and writing**
- » **O_APPEND**   **set the file offset to** *end of file* **prior to each** *write*
- » **O_CREAT**   **if the file does not exist, then create it, setting its mode to** *mode* **adjusted by** *umask*
- » **O_EXCL**   **if O_EXCL and O_CREAT are set, then** *open* **fails if the file exists**
- » **O_TRUNC**   **delete any previous contents of the file**

Here's a partial list of the options available as the second argument to open. (Further options are often available, but they depend on the version of Unix.) Note that the first three options are mutually exclusive: one, and only one, must be supplied. We discuss the third argument to *open,* mode, in the next few slides.

Appending Data to a File (1)

```
int fd = open("file", O_WRONLY);
lseek(fd, 0, SEEK_END);
    // sets the file location to the end
write(fd, buffer, bsize);
    // does this always write to the
    // end of the file?
```

We'd like to write data to the end of a file. One approach, shown here, is to use the **lseek** system call to set the file location in the file context structure to the end of the file. Once this is done, then when we write to the file, we're writing to its end and thus are appending data to the file.

However, this assumes that no other program is writing data to the end of the file at the same time. If another program were doing this, then the file could grow between our calls to **lseek** and **write**. If this happens, then the write would no longer be to the end of the file but would overwrite the data written by the other program.

Appending Data to a File (2)

```
int fd = open("file", O_WRONLY | O_APPEND);
write(fd, buffer, bsize);
    // this is guaranteed to write to the
    // end of the file
```

By using the O_APPEND option of open, we make certain that writes on this file descriptor are always to the end of file. If another program is doing this at the same time, the operating system makes certain that one doesn't start until after the other ends.

In the Shell ...

% program >> file

The ">>" operator tells the shell to open file with the O_APPEND flag so that writes are always to the end of the file.

File Access Permissions

- **Who's allowed to do what?**
  - **who**
    - » **user (owner)**
    - » **group**
    - » **others (rest of the world)**
  - **what**
    - » **read**
    - » **write**
    - » **execute**

Each file has associated with it a set of access permissions indicating, for each of three classes of principals, what sorts of operations on the file are allowed. The three classes are the owner of the file, known as **user**, the group owner of the file, known simply as **group**, and everyone else, known as **others**. The operations are grouped into the classes **read**, **write**, and **execute**, with their obvious meanings. The access permissions apply to directories as well as to ordinary files, though the meaning of execute for directories is not quite so obvious: one must have **execute** permission for a directory file in order to follow a path through it.

The system, when checking permissions, first determines the smallest class of principals the requester belongs to: user (smallest), group, or others (largest). It then, within the chosen class, checks for appropriate permissions.

Permissions Example

adm group:
joe, angie

```
$ ls -lR
.:
total 2
drwxr-x--x  2 joe    adm      1024 Dec 17 13:34 A
drwxr-----  2 joe    adm      1024 Dec 17 13:34 B

./A:
total 1
-rw-rw-rw-  1 joe    adm       593 Dec 17 13:34 x

./B:
total 2
-r--rw-rw-  1 joe    adm       446 Dec 17 13:34 x
-rw----rw-  1 angie  adm       446 Dec 17 13:45 y
```

The `ls -lR` command lists the contents of the current directory, its subdirectories, their subdirectories, etc. in long format (the `l` causes the latter, the `R` the former).

In the current directory are two subdirectories, **A** and **B**, with access permissions as shown in the slide. Note that the permissions are given as a string of characters: the first character indicates whether or not the file is a directory, the next three characters are the permissions for the owner of the file, the next three are the permissions for the members of the file's group's members, and the last three are the permissions for the rest of the world.

Quiz: the users **joe** and **angie** are members of the **adm** group; **leo** is not.

- May **leo** list the contents of directory *A*?

- May **leo** read *A/x*?

- May **angie** list the contents of directory *B*?

- May **angie** modify *B/y*?

- May **joe** modify *B/x*?

- May **joe** read *B/y*?

Setting File Permissions

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char *path, mode_t mode)
```

- **sets the file permissions of the given file to those specified in *mode***
- **only the owner of a file and the superuser may change its permissions**
- **nine combinable possibilities for *mode***
  (*read/write/execute* for *user*, *group*, and *others*)
  - » **S_IRUSR (0400), S_IWUSR (0200), S_IXUSR (0100)**
  - » **S_IRGRP (040), S_IWGRP (020), S_IXGRP (010)**
  - » **S_IROTH (04), S_IWOTH (02), S_IXOTH (01)**

The **chmod** system call (and the similar **chmod** shell command) is used to change the permissions of a file. Note that the symbolic names for the permissions are rather cumbersome; what is often done is to use their numerical equivalents instead. Thus the combination of read/write/execute permission for the user (0700), read/execute permission for the group (050), and execute-only permission for others (01) can be specified simply as 0751.

Umask

- **Standard programs create files with "maximum needed permissions" as mode**
  - compilers: 0777
  - editors: 0666
- **Per-process parameter, *umask*, used to turn off undesired permission bits**
  - e.g., turn off all permissions for others, write permission for group: set umask to 027
    - » compilers: permissions = 0777 & ~(027) = 0750
    - » editors: permissions = 0666 & ~(027) = 0640
  - set with *umask* system call or (usually) shell command

The **umask** (often called the "creation mask") allows programs to have wired into them a standard set of maximum needed permissions as their file-creation modes. Users then have, as part of their environment (via a per-process parameter that is inherited by child processes from their parents), a limit on the permissions given to each of the classes of security principals. This limit (the **umask**) looks like the 9-bit permissions vector associated with each file, but each one-bit indicates that the corresponding permission is not to be granted. Thus, if **umask** is set to 022, then, whenever a file is created, regardless of the settings of the mode bits in the *open* or *creat* call, write permission for *group* and *others* is not to be included with the file's access permissions.

You can determine the current setting of **umask** by executing the **umask** shell command without any arguments.

(Recall that numbers written with a leading 0 are in octal (base-8) notation.)

Originally in Unix one created a file only by using the **creat** system call. A separate O_CREAT flag was later given to **open** so that it, too, can be used to create files. The **creat** system call fails if the file already exists. For **open**, what happens if the file already exists depends upon the use of the flags O_EXCL and O_TRUNC. If O_EXCL is included with the flags (e.g., **open("newfile", O_CREAT|O_EXCL, 0777)),** then, as with **creat**, the call fails if the file exists. Otherwise, the call succeeds and the (existing) file is opened. If O_TRUNC is included in the flags, then, if the file exists, its previous contents are eliminated and the file (whose size is now zero) is opened.

When a file is created by either **open** or **creat**, the file's initial access permissions are the bitwise AND of the mode parameter and the complement of the process's **umask** (explained in the previous slide).

Link and Reference Counts

1	RDONLY	0	inode pointer

n1	dir1	dir2

link count == 2
reference count == 1

n2	f1

```
int fd = open("n1", O_RDONLY);
    // n1's reference count is
    // incremented by 1
```
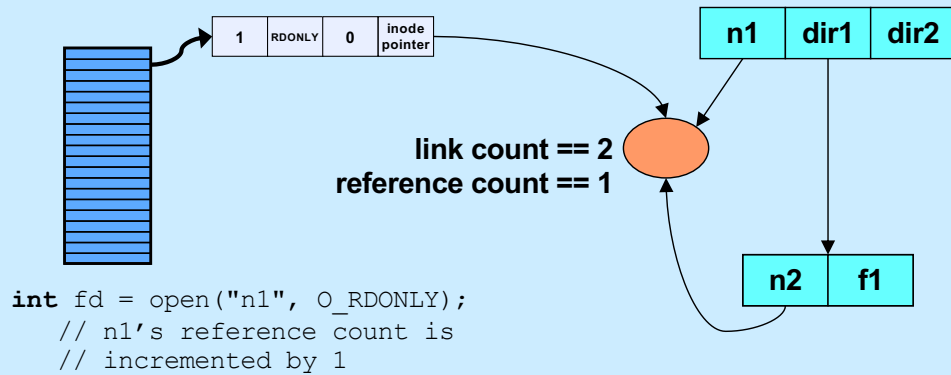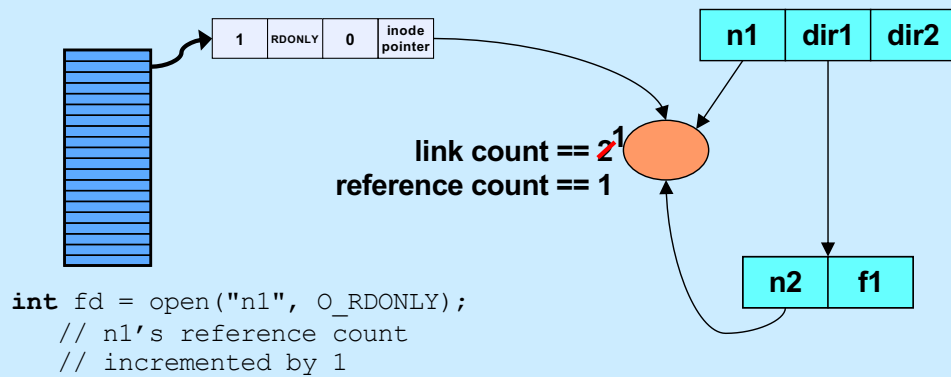
A file's link count is the number of directory entries that refer to it. There's a separate reference count that's the number of file context structures that refer to it (via the inode pointer – see slide XVII-9). These counts are maintained in the file's inode, which contains all information used by the operating system to refer to the file (on disk).

Link and Reference Counts


```
int fd = open("n1", O_RDONLY);
    // n1's reference count
    // incremented by 1

unlink("n1");
    // link count decremented by 1
    // same effect in shell via "rm n1"
```
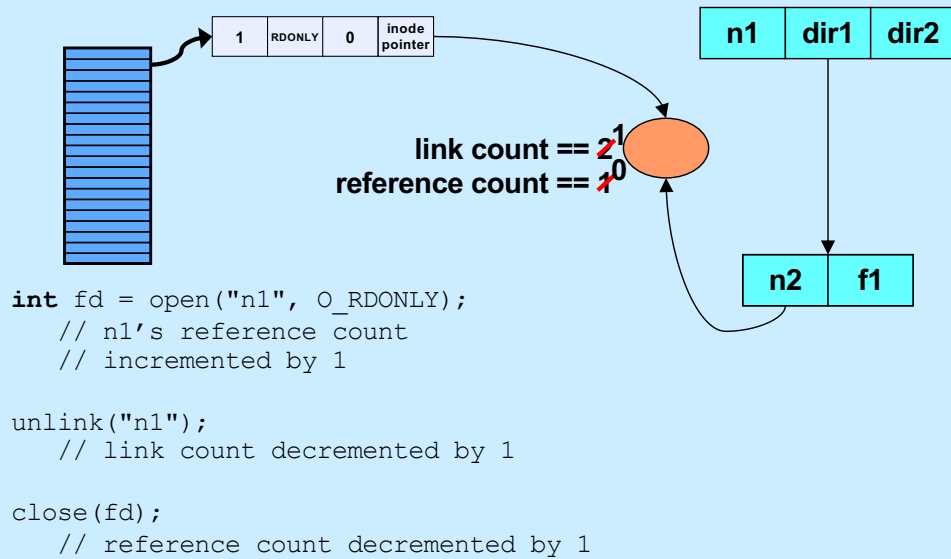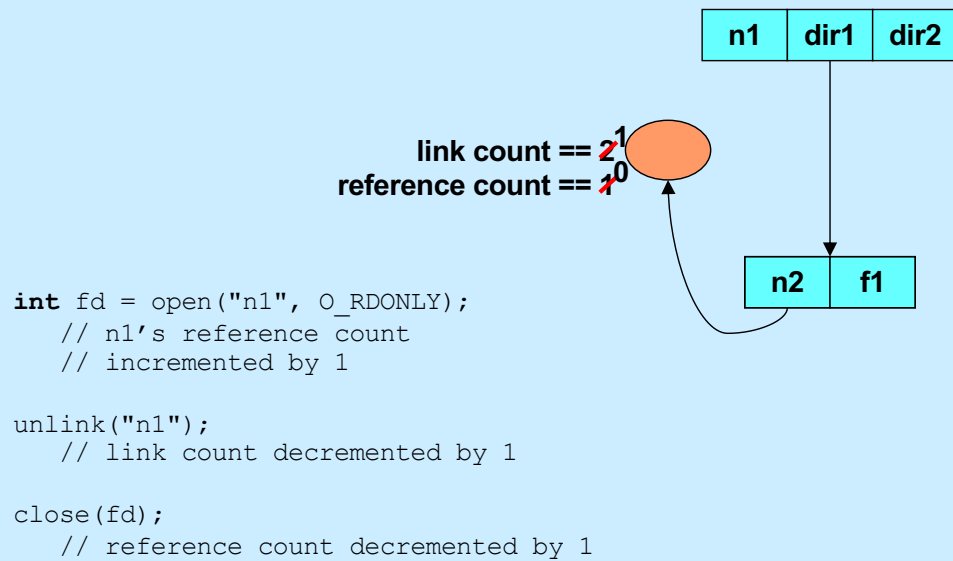
Note that the shell's rm command is implemented using unlink; it simply removes the directory entry, reducing the file's link count by 1.

Link and Reference Counts

1	RDONLY	0	inode pointer

n1	dir1	dir2

link count == $\overset{1}{2}$
reference count == $\overset{0}{1}$

n2	f1

```
int fd = open("n1", O_RDONLY);
   // n1's reference count
   // incremented by 1

unlink("n1");
   // link count decremented by 1

close(fd);
   // reference count decremented by 1
```

Link and Reference Counts

n1	dir1	dir2

link count == 2^1
reference count == 1^0

n2	f1

```
int fd = open("n1", O_RDONLY);
    // n1's reference count
    // incremented by 1

unlink("n1");
    // link count decremented by 1

close(fd);
    // reference count decremented by 1
```

A file is deleted if and only if both its link and reference counts are zero.

Link and Reference Counts

n1	dir1	dir2

link count == ~~1~~ 0
reference count == 0

n2	f1

```
unlink("dir1/n2");
    // link count decremented by 1
```
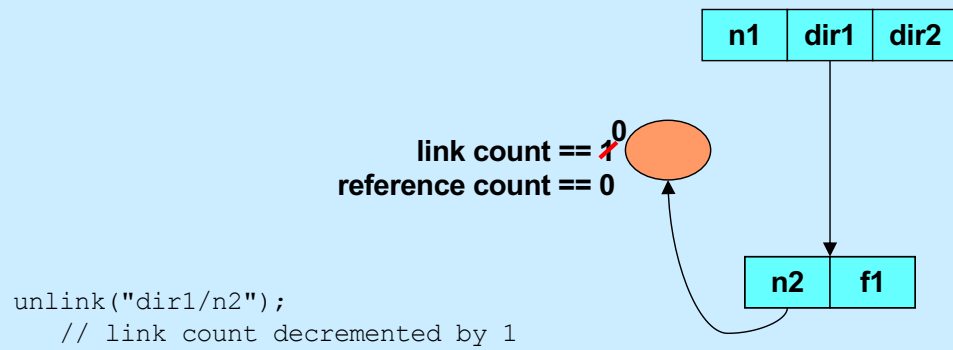
A file is deleted if and only if both its link and reference counts are zero.

Quiz 3

```
int main() {
  int fd = open("file", O_RDWR|O_CREAT, 0666);
  unlink("file");
  PutStuffInFile(fd);
  GetStuffFromFile(fd);
  return 0;
}
```

Assume that *PutStuffInFile* writes to the given file, and *GetStuffFromFile* reads from the file.
a) This program is doomed to failure, since the file is deleted before it's used
b) Because the file is used after the unlink call, it won't be deleted
c) The file will be deleted when the program terminates

Note that when a process terminates, all its open files are automatically closed.