

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

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.

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 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) {
            (void)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.

Back to Primes ...

```
int nprimes;
int *prime;
int main(int argc, char *argv[]) {
    ...
    for (i=1; i<nprimes; i++) {
        ...
    }
    if (write(1, prime, nprimes*sizeof(int)) == -1) {
        perror("primes output");
        exit(1);
    }
    return(0);
}
```


Human-Readable Output

```
int nprimes;
int *prime;
int main(int argc, char *argv[]) {
    ...
    for (i=1; i<nprimes; i++) {
        ...
    }
    for (i=0; i<nprimes; i++) {
        printf("%d\n", prime[i]);
    }
    return(0);
}
```

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[] = {"primes", "300", 0};
    execv("/home/twd/bin/primes", 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.

From the Shell ...

```
$ primes
```

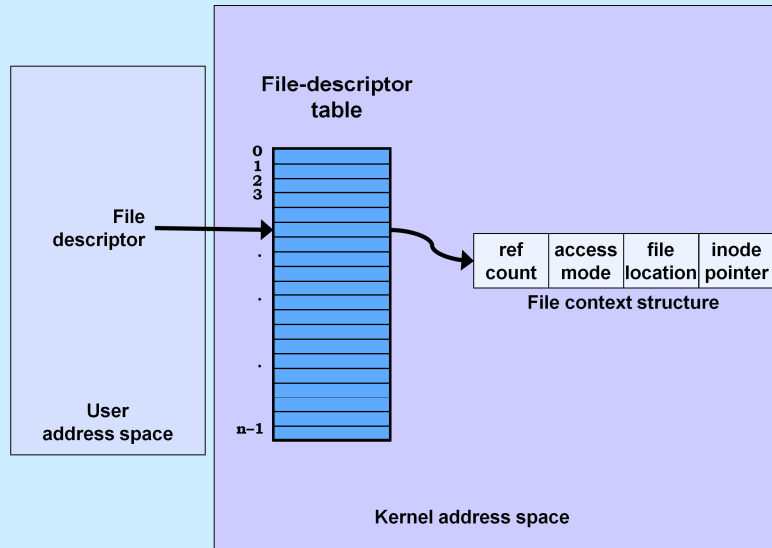
– **output goes to the display**

```
$ primes > Output
```

– **output goes to the file “Output” in the current directory**

Our shell examples are all in bash.

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

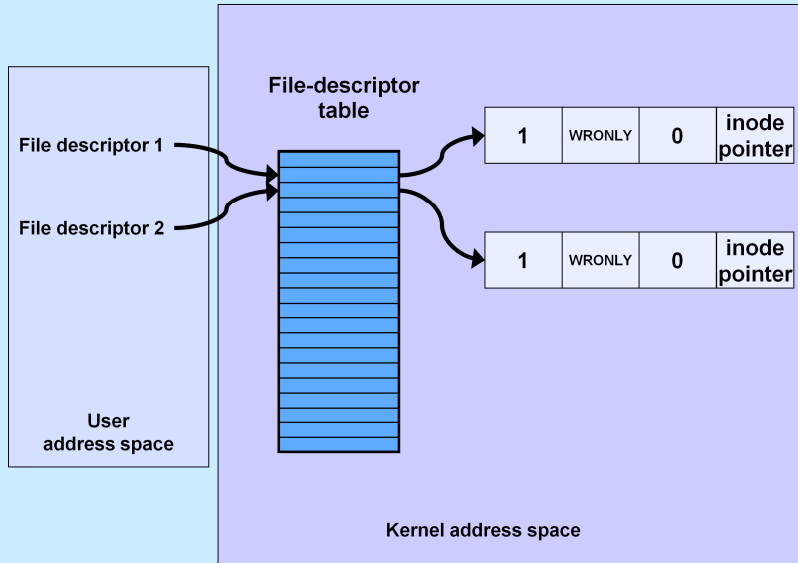
```
if (fork() == 0) {
    /* set up file descriptors 1 and 2 in the child process */
    close(1);
    close(2);
    if (open("/home/twd/Output", O_WRONLY) == -1) {
        exit(1);
    }
    if (open("/home/twd/Output", O_WRONLY) == -1) {
        exit(1);
    }
    char *argv[] = {"program", 0};
 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

From the Shell ...

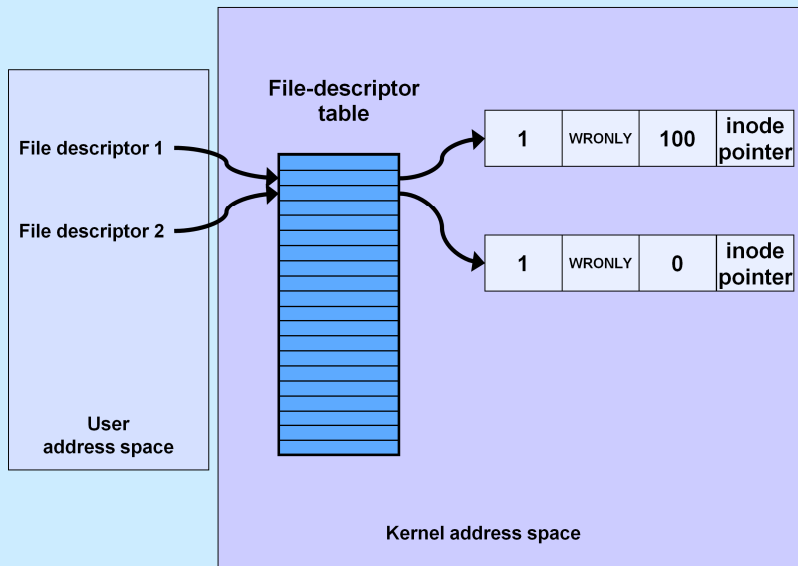
```
$ primes >Output 2>Output
```

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

Redirected Output



Redirected Output After Write

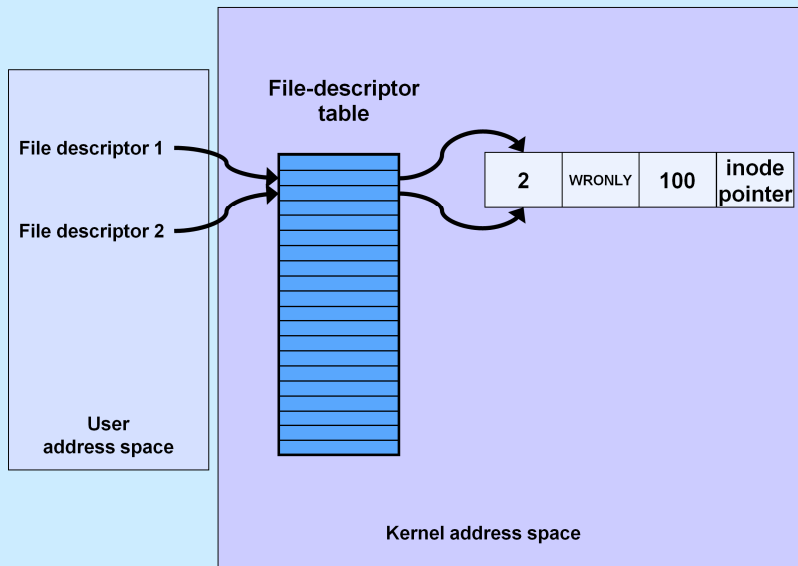


The potential problem here is that, since our file (/home/twd/Output) has been opened once for each file descriptor, when a write is done through file descriptor 1, the file location field in its context is incremented by 100, but not that in the other context. Thus a subsequent write via file descriptor 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[] = {"program", 0};
    execv("/home/twd/bin/program", 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 ...

```
$ primes >Output 2>&1
```

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

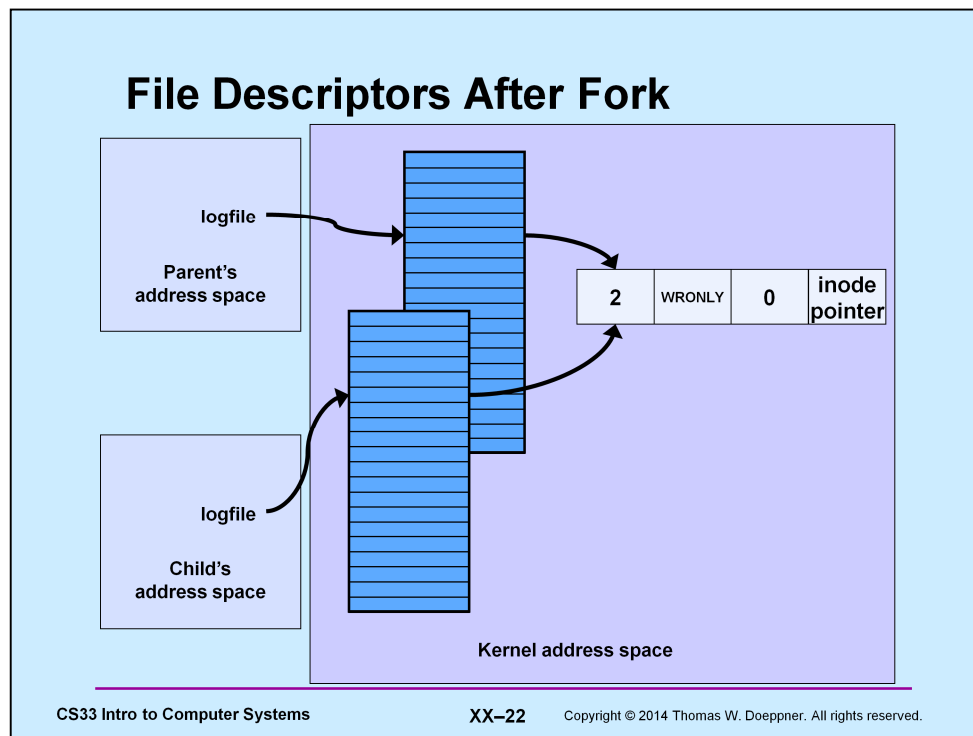
Fork and File Descriptors

```
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.

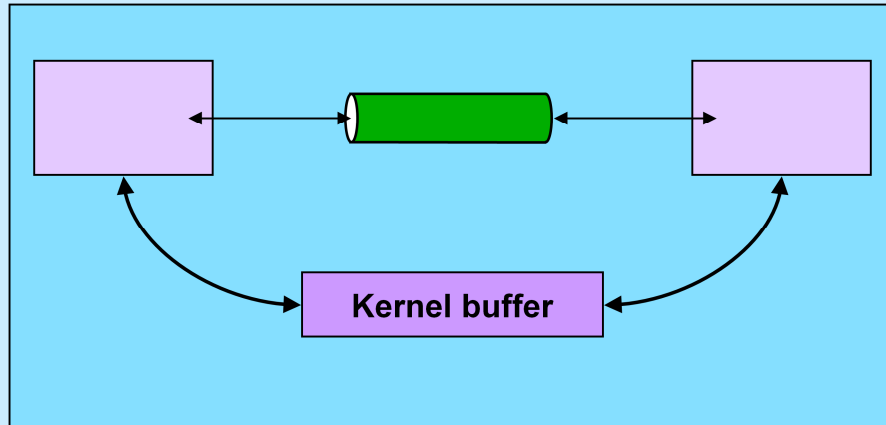


Note that after a fork, the reference counts in the file context structures are incremented to account for the new references by the child process.

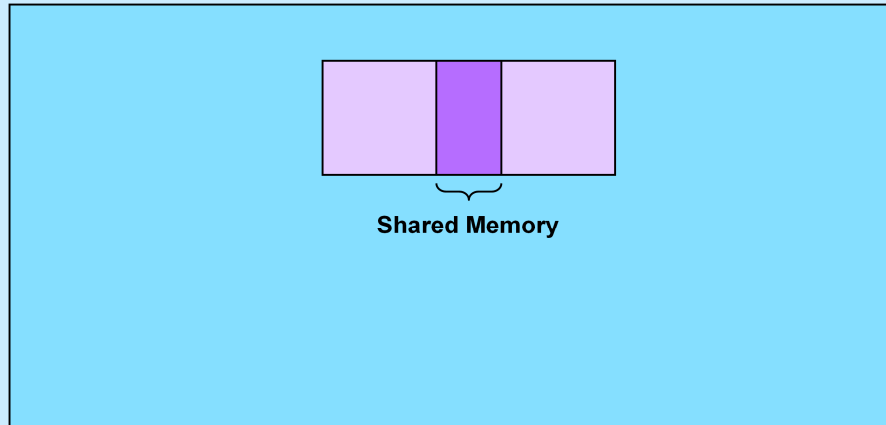
Interprocess Communication (IPC)



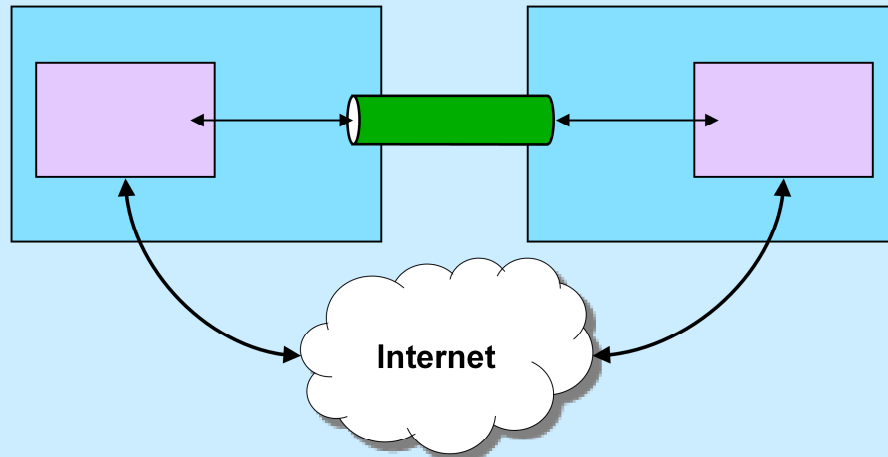
Interprocess Communication: Same Machine I



Interprocess Communication: Same Machine II

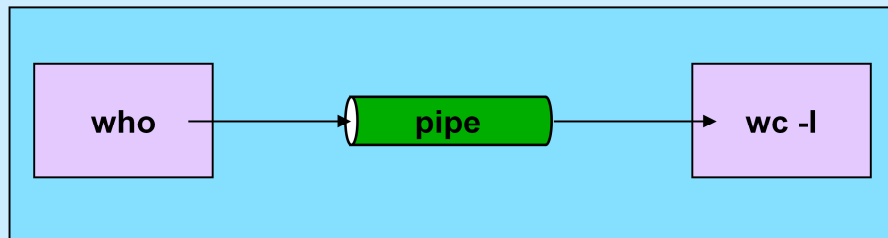


Interprocess Communication: Different Machines



Intramachine IPC

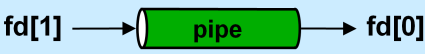
```
$cs1ab2e who | wc -l
```



Intramachine IPC

```
$cslab2e who | wc -l
```

```
int fd[2];
pipe(fd);
if (fork() == 0) {
    close(fd[0]);
    close(1);
    dup(fd[1]); close(fd[1]);
    execlp("who", "who", 0); // who sends output to pipe
}
if (fork() == 0) {
    close(fd[1]);
    close(0);
    dup(fd[0]); close(fd[0]);
    execlp("wc", "wc", "-l", 0); // wc gets input from pipe
}
close(fd[1]); close(fd[0]);
// ...
```



```
graph LR
    fd1[fd[1]] --> pipe((pipe))
    pipe --> fd0[fd[0]]
```

The *pipe* system call creates a “pipe” in the kernel and sets up two file descriptors. One, in `fd[1]`, is for writing to the pipe; the other, in `fd[0]`, is for reading from the pipe.

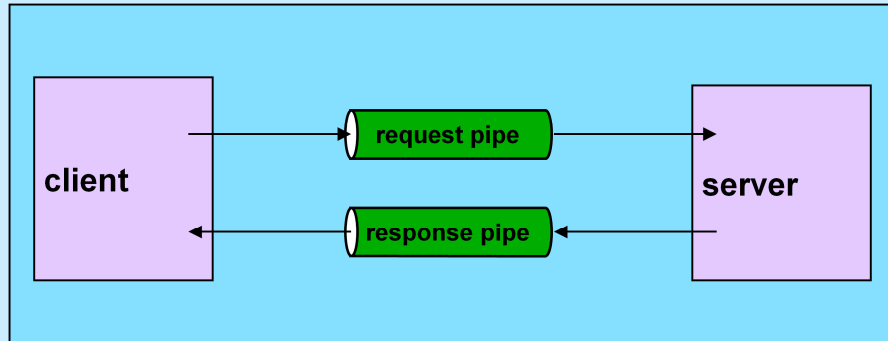
Pipes

- **Pro**
 - really easy to use
 - anonymous: no names to worry about
- **Con**
 - anonymous: can't give them names
 - » communicating processes must be related

Named Pipes

```
mkfifo("/u/twd/service", 0622);  
    // creates a named pipe (FIFO) that  
    // anyone may write to but only whose  
    // owner may read from  
  
int wfd = open("/u/twd/service", O_WRONLY);  
write(wfd, request, sizeof(request));  
    // send request in one process  
  
int rfd = open("/u/twd/service", O_RDONLY);  
read(rfd, request, sizeof(request));  
    // receive request in another process
```

Client/Server



Intermachine Communication

- Can pipes and named pipes be made to work across multiple machines?

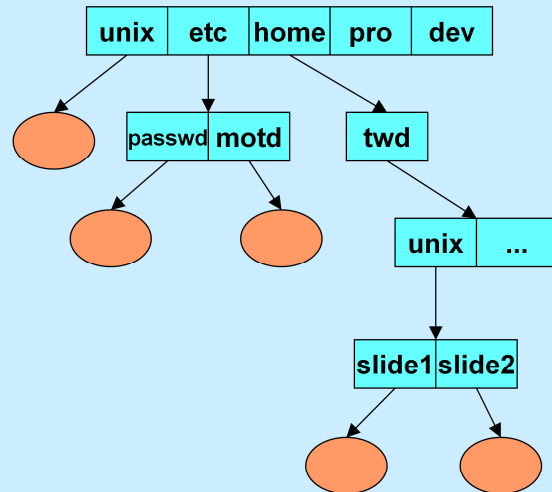
– covered soon ...

» what happens when you type

```
who | ssh cs1ab3a wc -l
```

?

Directories



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

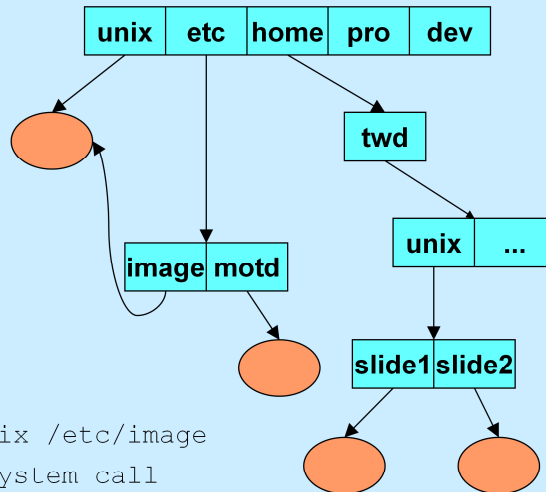
Component Name	Inode Number
directory entry	

.	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



```
$ 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

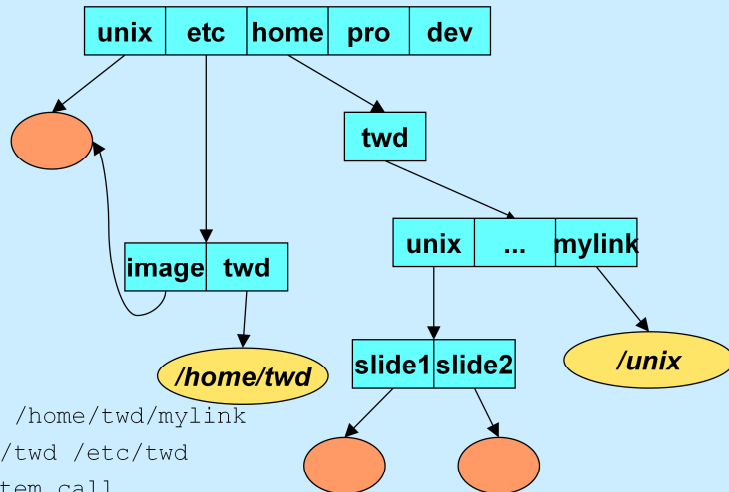
Here are the (abbreviated) contents of both the *root* (/) and */etc* directories, showing how */unix* and */etc/image* are the same file. Note that if the directory entry */unix* is deleted (via the shell's "rm" command), the file (represented by inode 117) continues to exist, since there is still a 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



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
 - displayed (via shell) using “pwd”
 - » how is this done?

The *working directory* is maintained (as the inode number (explained subsequently) of the directory) 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
- » O_NONBLOCK don't wait if I/O can't be done immediately

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`, shortly.

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

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

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

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

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 *tom* and *trina* are members of the *adm* group; *andy* is not.

- May *andy* list the contents of directory *A*?
- May *andy* read *A/x*?
- May *trina* list the contents of directory *B*?
- May *trina* modify *B/y*?
- May *tom* modify *B/x*?
- May *tom* 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 = $0777 \& \sim(027) = 0750$
 - » editors: permissions = $0666 \& \sim(027) = 0640$
 - 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.

Creating a File

- Use either *open* or *creat*
 - `open(const char *pathname, int flags, mode_t mode)`
 - » flags must include `O_CREAT`
 - `creat(const char *pathname, mode_t mode)`
 - » *open* is preferred
- The *mode* parameter helps specify the permissions of the newly created file
 - permissions = mode & ~umask

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 next slide).