Operating Systems

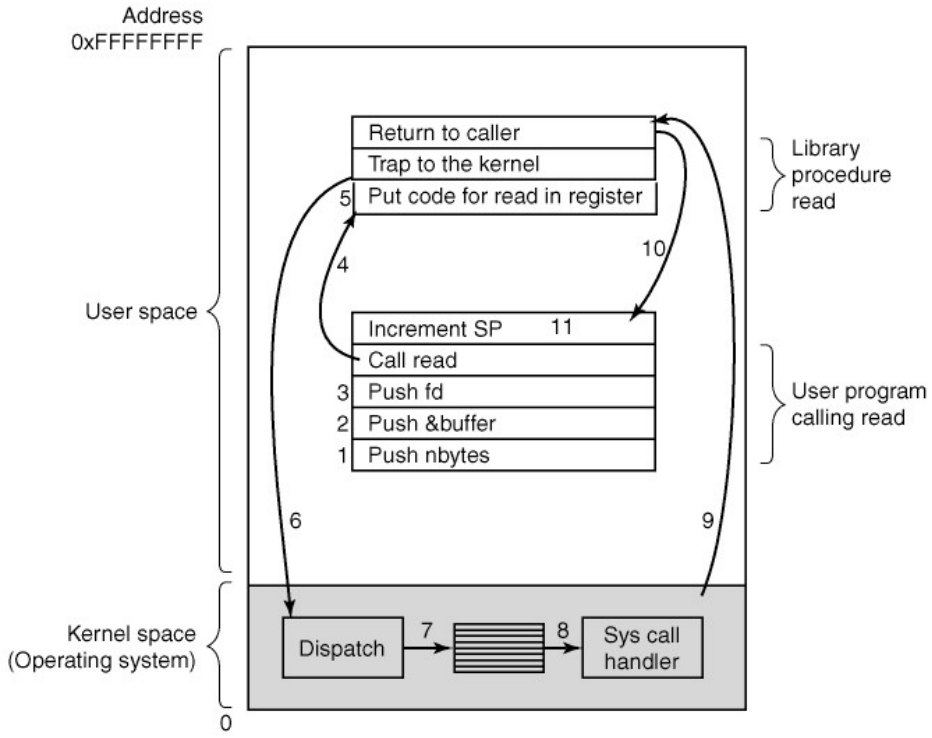
Christopher Kruegel
Department of Computer Science
UC Santa Barbara
http://www.cs.ucsb.edu/~chris/

Overview

- System calls (definition and overview)
- Processes and related system calls
- Signals and related system calls
- Memory-related system calls
- Files and related system calls

System Calls

- System calls are the interface to operating system services - they are how we tell the OS to do something on our behalf
- API to the OS
- Hide OS and hardware complexity from us



System Calls

Example

```
#include <unistd.h>

int main(int argc, char* argv[])
{
  int fd, nread;
  char buf[1024];

  fd = open("my_file",0);      /* Open file for reading */
  nread = read(fd,buf,1024);   /* Read some data */

  /* Presumably we do something with data here */

  close(fd);
}
```

System Calls

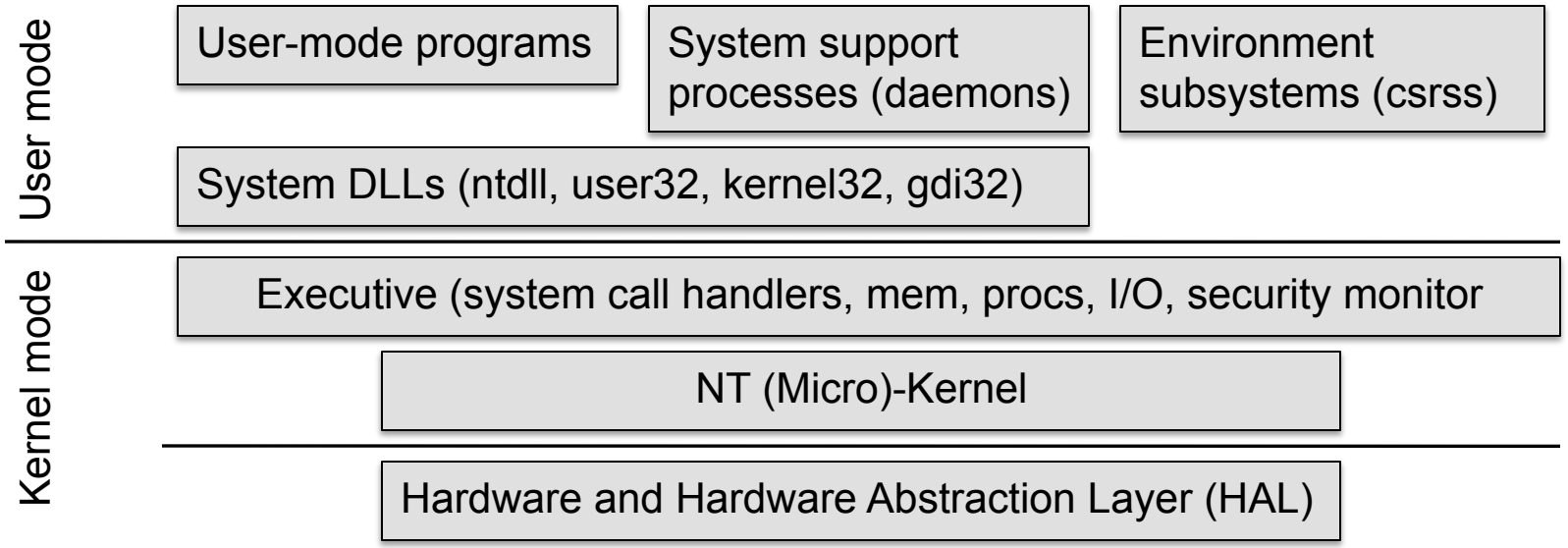
- How the system calls communicate back to us?

- Return value – usually return -1 on error, >= 0 on success
  - library functions set a global variable "errno" based on outcome
    - 0 on success,
    - positive values encode various kinds of errors
  - can use `perror` library function to get a string

- Buffers pointed to by system call arguments
  - e.g., in case of a read system call
  - values need to be copied between user and kernel space

Windows NT

- Competitor to Unix
  - true multi-user
  - emphasis on portability and object-oriented design
  - isolation for applications and resource access control
  - similar to Unix, kernel and user mode

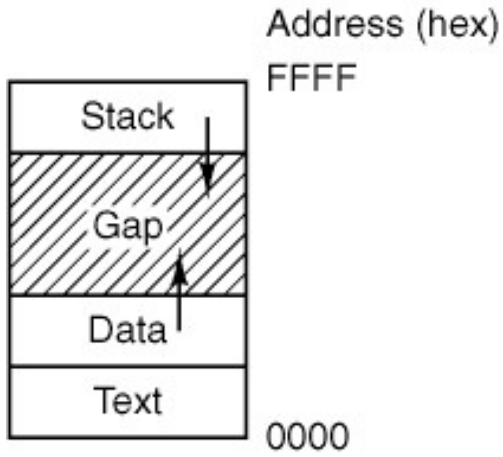


Processes

- Concept
  - processes - program in execution
- Each process has own memory space and process table entry
- Process table entry
  - stores all information associated with a process (except memory)
  - register values, open files, user ID (UID), group ID (GID), ..
- Processes are indexed by the process ID (PID)
  - integer that indexes into process table

Processes

- Memory layout



- OS responsible for changing between multiple processes
- Shakespeare example
  - multiple subplots which get advanced by interleaved scenes
  - actors and props must be taken away and saved while next scene goes on, and they are brought back later

Process System Calls

- fork (create a new process)
- exec (change program in process)
- exit (end process)
- wait (wait for a child process)
- getpid(get process PID)
- getpgrp(get process GID)

fork()

- Get almost identical copy (the child) of the original parent
- File descriptors, arguments, memory, stack … all copied
- Even current program counter
- But not completely identical - why?

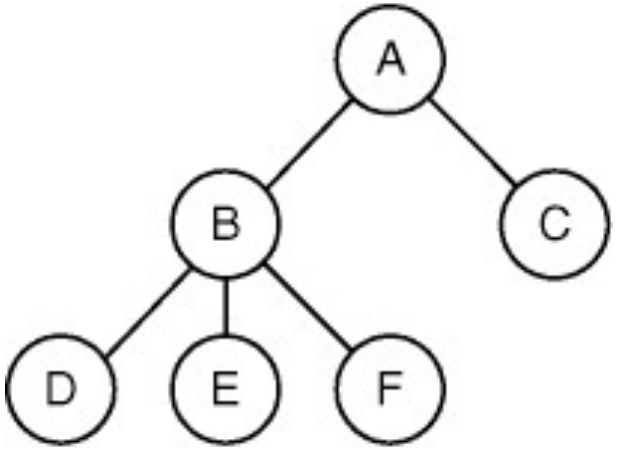
- Syntax: pid = fork();
- Return value from fork call is different is zero in child, but in parent, it is PID of child.

fork() cont.


```
int main(int argc, char* argv[])
{
   int status;
   char* ls_args[2];
   ls_args[0] = ".";
   ls_args[1] = 0;
   if(fork() > 0)
   {
      /* Parent */
      waitpid(-1,&status,0);
      exit(status);
   }
   else
   {
      /* Child */
      execve("/bin/ls", ls_args,0);
   }
```

Process Hierarchy

- Notion of a hierarchy (tree) of processes
- Each process has a single parent - parent has special privileges
- In Unix, all user processes have ’init’ as their ultimate ancestor



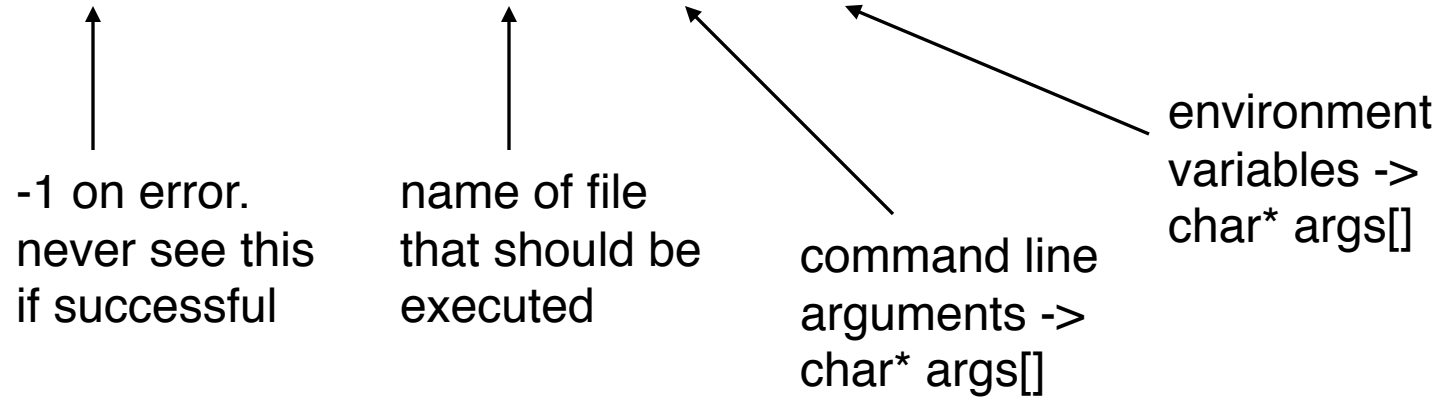
Additional ways to group processes

- Process Groups (job control)
- Sessions (all processes for a user)

exec()

- Change program in process
  - i.e., launch a new program that replaces the current one
- Several different forms with slightly different syntax

```
status = execve(prog, args, env);
```



wait()

- When a process is done it can call exit(status).
- This is the status that "echo $?" can show you in the shell
- A parent can wait for its children (it blocks until they are done)

```
status = waitpid(pid, &statloc, options);
```

-1 on error - otherwise, PID of process that exited

which PID to wait for; -1 means any child

exit code of process that has exited

check man page for details

Shell

- Is an example of a program that makes heavy use of basic process system calls
- Basic cycle:

  prompt, read line, parse line, fork (child execs the command, parent waits)

- Have to handle & (background job)
- Have to handle > | etc, - somehow connecting stdin and stdout of the child to files or other programs

Shell


```
#define TRUE 1

while (TRUE) {                                  /* repeat forever */
      type_prompt( );                           /* display prompt on the screen */
      read_command(command, parameters);        /* read input from terminal */

      if (fork( ) != 0) {                       /* fork off child process */
           /* Parent code. */
           waitpid(–1, &status, 0);             /* wait for child to exit */
      } else {
           /* Child code. */
           execve(command, parameters, 0);      /* execute command */
      }
}
```

Signals

- Report events to processes in asynchronous fashion
  - process stops current execution (saves context)
  - invokes signal handler
  - resumes previous execution
- Examples
  - user interrupts process (terminate process with CTRL-C)
  - timer expires
  - illegal memory access
- Signal handling
  - signals can be ignored
  - signals can be mapped to a signal handler (all except SIGKILL)
  - signals can lead to forced process termination

Signal System Calls

- kill (send signal to process)
- alarm(set a timer)
- pause(suspend until signal is received)
- sigaction(map signal handler to signal)
- sigprocmask(examine or change signal mask)
- sigpending(obtain list of pending signals that are blocked)

Memory System Call

- Quite simple in Unix
  - brk, sbrk – increase size of data segment
    - used internally by user-space memory management routines
  - mmap
    - map a (portion of a) file into process memory

Files

- Conceptually, each file is an array of bytes
- Special files
  - directories
  - block special files (disk)
  - character special files (modem, printer)
- Every running program has a table of open files (file table)
- File descriptors are integers which index into this table
- Returned by open, creat
- Used in read, write, etc. to specify which file we mean

Files

- Initially, every process starts out with a few open file descriptors
  - 0 - stdin
  - 1 - stdout
  - 2 - stderr
- We have a file pointer, which marks where we are currently up to in each file (kept in an OS file table)
- File pointer starts at the beginning of the file, but gets moved around as we read or write (or can move it ourselves with lseek system call)

Inode

- A file just contains its contents
- Information about the file is contained in a separate structure called an inode - one inode per file
- Inode stores
  - permissions, access times, ownership
  - physical location of the file contents on disk (list of blocks)
  - number of links to the file - file is deleted when link counter drops to 0
- Each inode has an index (the I-number) that uniquely identifies it
- The OS keeps a table of all the inodes out on disk
- Inodes do not contain the name of the file - that's directory information

OS File Data Structures

- Where is all the information stored?
  - interaction is complex

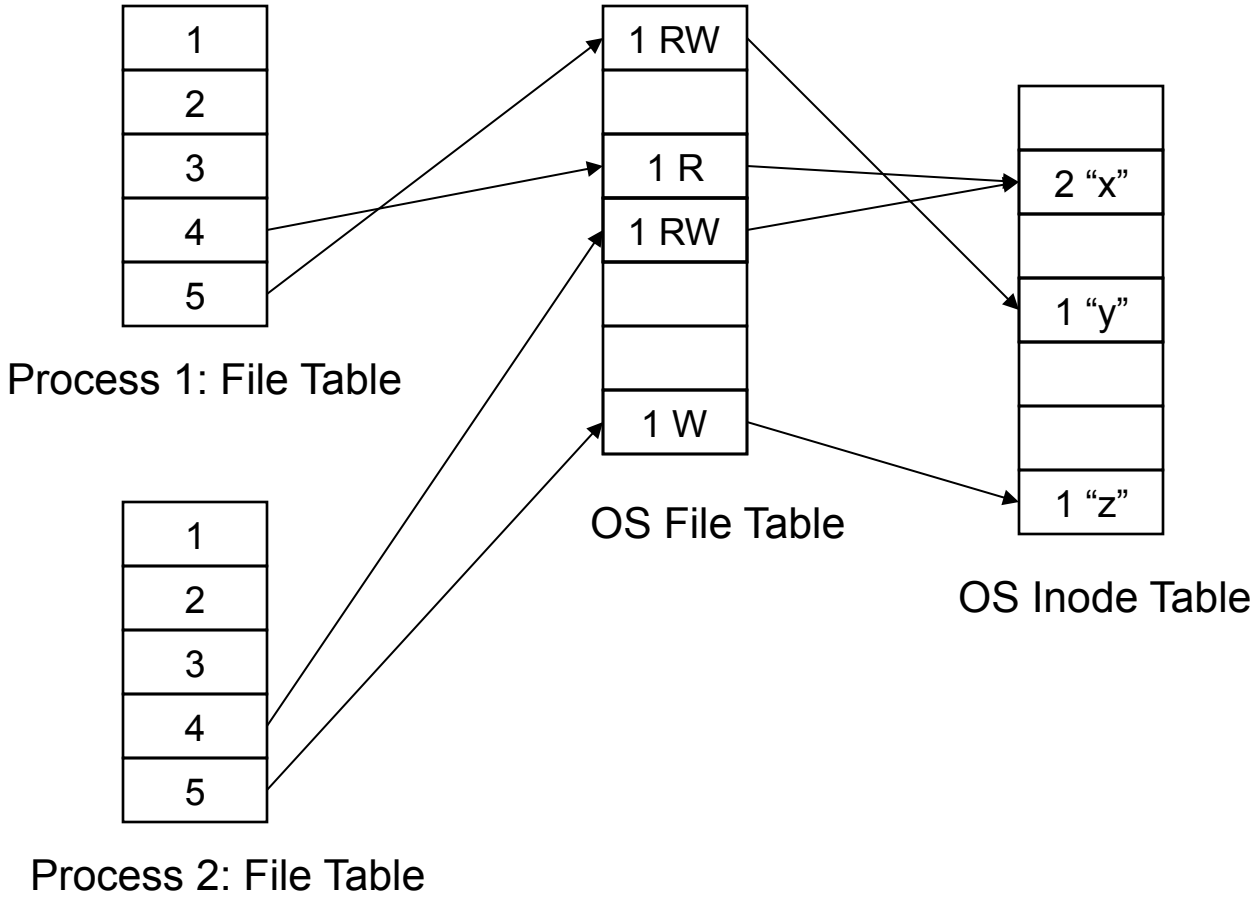
- Example
  - Process 1

```
open("x", O_RDONLY)
open("y", O_RDWR)
```

  - Process 2

```
open("x", O_RDWR)
open(”z", O_WRONLY)
```

OS - File Data Structures



OS File Data Structures

- File pointers live in the OS table, as does the RW information
- Why is it done this way?
- On a fork, the per-process file tables get duplicated in the child
- But child shares file pointer with parent
  - note that counters in OS table would increase to 2 after a fork
- Note that on exec, file tables are not reset
  - process can pass open files to child processes
  - even when the child has no longer the right to actually open this file!

OS File Data Structures

- Counts in the inode are really the link counts
  - processes can have a link into the file just like directories can

- Neat trick - do an open on a filename, then unlink the filename
  - now, there is a file on disk that only you have a link to
  - but no-one else can open (or delete) it

File Permissions

- Users and processes have UIDs and GIDs
  - where is mapping between usernames and UID?
- Every file has a UID and a GID of its owner
- Need a way to control who can access the file
- General schemes:
  - ACL - Access Control Lists (every file lists who can access it)
  - Capabilities (every identity lists what it can access)
- Unix scheme is a cut-down ACL
- There are three sets of bits that control who can do what to a file

File Permissions

- For example, via "ls -l" command

```
-rw-r--r-- 1 chris  chris  1868 Jan 8 22:02 schedule.txt
```

- Usual way to specify the "mode" in a system call is via a single integer using octal notation
  - above example has mode 0644

- UID 0 is the "root" or "superuser" UID
  - is omnipotent

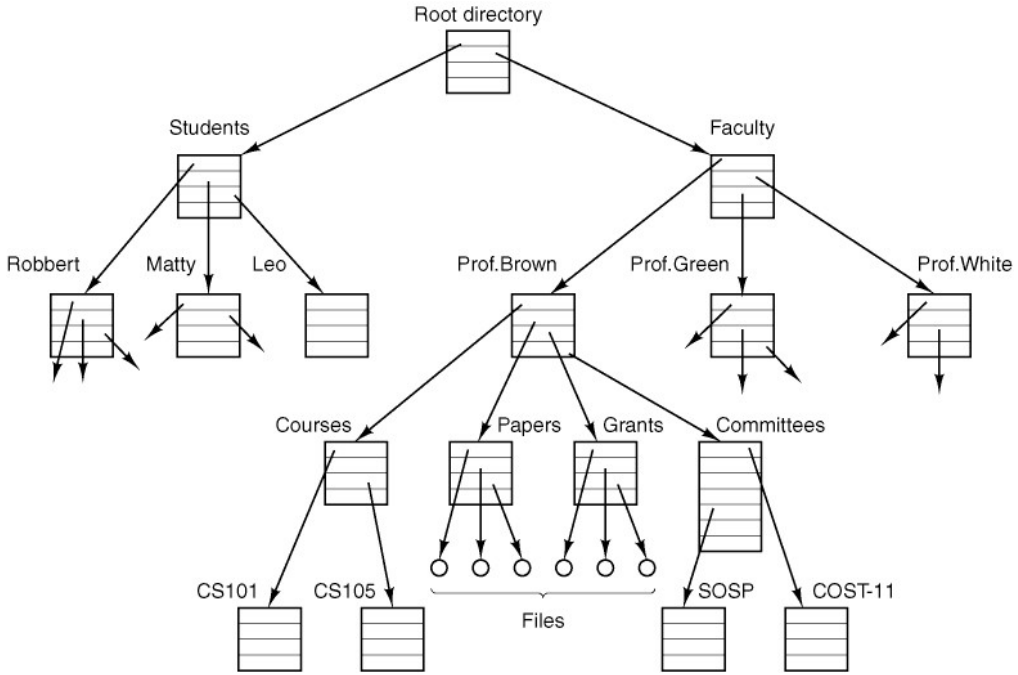
- Ordinary users can only change the mode of their own files, root can change mode or ownership of anybody's files

File System System Calls

- open(open a file)
- close(close a file)
- creat(create a file)
- read(read from file)
- write(write from file)
- chown(change owner)
- chmod(change permission bits)

Directories

- Files are managed in a hierarchical structure (called file system)
- internal nodes in the file system are directories
- leaf nodes are files



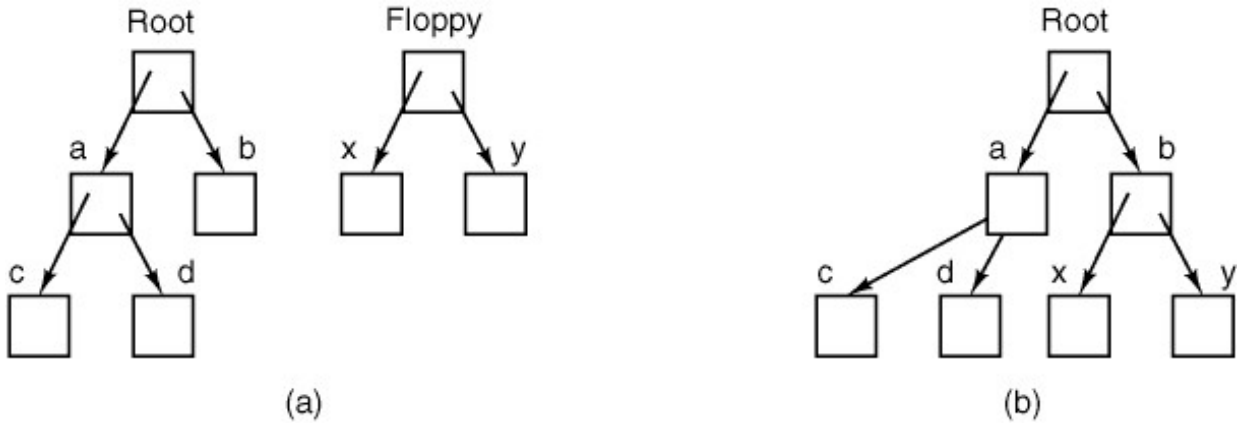
Directories

- Directories are just regular files that happen to contain the names and locations (specifically I-numbers) of other files
- File system is a single name space that starts at the root directory
- Files can be uniquely identified by specifying their absolute path
  - e.g., /home/chris/schedule.txt
- Relative path
  - starts from current working directory (CWD)
  - e.g., chris/schedule.txt, assuming that the current working directory is /home

Multiple File Systems

- How can we include another file system into our root file system?

- mount() system call is used to achieve this!



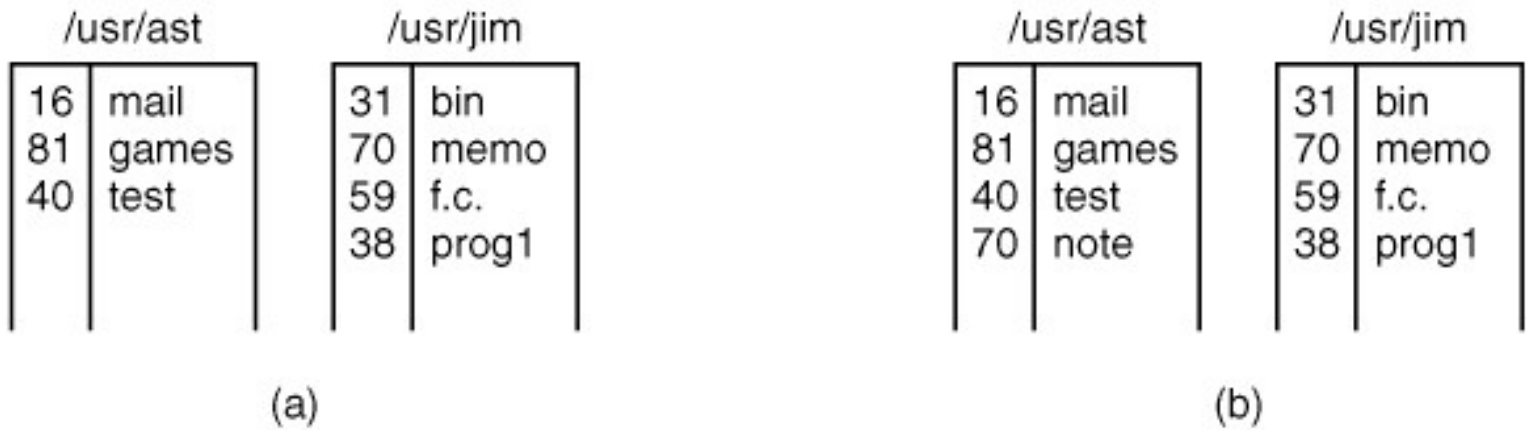
mount("/dev/floppy","/b",0);

Links

- Since the only place that the name of a file appears is in a directory entry, it is possible to have multiple names correspond to the same file
- All it takes is several entries in one or more directories which point to the same I-node (I.e., have the same I-number).
- This is why the directory structure is not really a tree
  - it is really a full directed graph (can even have cycles!)
- This concept refers to hard links. There are also soft links
  - small file that contains the name of the target file

Links

- link() system call establishes a link between two files



link("/usr/jim/memo","/usr/ast/note");

File Deletion

- How to delete files?
  - implicitly done via the unlink() system call
  - when there are no links to a file anymore, it gets removed

Synchronization

- The operating system keeps a lot of stuff in memory about the state of files on disk (e.g., the inodes)
- It does not necessarily store all changes onto disk immediately (for efficiency reasons, things are cached)
- Hence, if the OS dies unexpectedly, the file system can be in an inconsistent state
- sync()
  - tells the OS to write out everything to disk
  - invoked regularly by the `update` process

Pipes

- Common Unix mechanism for processes to communicate with one another
- Pipes are basically special files
- Implemented as circular buffer of fixed size (e.g., 4k)
- Communication through read and write system calls
- Block if reading an empty pipe or writing a full one
- Use at shell level (ls | wc, who | sort | lpr)

Pipe System Call

- Create a pipe:
  need array of size 2
  array[0] is FD for reading, array[1] is FD for writing.

```
int fildes[2];              /* FD's for pipe. */
pipe(fildes);               /* create pipe */
read(fildes[0], ...);       /* read from pipe */
write(fildes[1], ...);      /* write to pipe */
```

- But talking to ourselves is no fun. Need someone else to talk to
- Solution - create pipe, then fork

Inter-process Communication


```
#define STD_INPUT 0                     /* file descriptor for standard input */
#define STD_OUTPUT 1                    /* file descriptor for standard output */

pipeline(process1, process2)
char *process1, *process2;              /* pointers to program names */
{
  int fd[2];

  pipe(&fd[0]);                         /* create a pipe */
  if (fork() != 0) {
        /* The parent process executes these statements. */
        close(fd[0]);                   /* process 1 does not need to read from pipe */
        close(STD_OUTPUT);              /* prepare for new standard output */
        dup(fd[1]);                     /* set standard output to fd[1] */
        close(fd[1]);                   /* this file descriptor not needed any more */
        execl(process1, process1, 0);
  } else {
```

Inter-process Communication


```
        /* The child process executes these statements. */
        close(fd[1]);                   /* process 2 does not need to write to pipe */
        close(STD_INPUT);               /* prepare for new standard input */
        dup(fd[0]);                     /* set standard input to fd[0] */
        close(fd[0]);                   /* this file descriptor not needed any more */
        execl(process2, process2, 0);
    }
}
```