

# Introduction

Week 03 - Lecture 2

Processes, Files, Directories

# Team

## Instructors

Giancarlo Succi

Joseph Brown

## Teaching Assistants

Vladimir Ivanov

Stanislav Litvinov

Alexey Reznik

Munir Makhmutov

Hamna Aslam

# Sources

- These slides have been adapted from the original slides of the adopted book:
  - Tanenbaum & Bo, *Modern Operating Systems*: 4th edition, 2013  
Prentice-Hall, Inc.and customised for the needs of this course.
- Additional input for the slides are detailed later

# Processes (1)

- **Process** is a key concept in all OSs
- Definition: a program in execution
- Process is associated with an **address space**, a list of memory locations from 0 to some maximum, which the process can read and write
- Also associated with set of resources (registers, open files, related processes, etc.)
- Process can be thought of as a container
  - Holds all information needed to run program

# Processes (2)

- In a multiprogramming system the OS decides to stop running one process and start running another
- When a process is suspended temporarily, it must later be restarted in exactly the same state it had when it was stopped - all the information about the process must be explicitly saved somewhere during the suspension
- In many OSs it is stored in an OS table called the **process table**, which is an array of structures, one for each process currently in existence

# Processes (3)

- A (suspended) process consists of its address space, called the **core image** and its process table entry
- The key process-management system calls deal with the creation and termination of processes
- Example:
  - a process called the **command interpreter** or shell reads commands from a terminal
  - User wants to compile a program
  - Shell creates a process that will run the compiler
  - After it finishes the compilation, it executes a system call to terminate itself

# Processes (4)

- **Process tree structure:** a process can create one or more other processes (**child processes**) and these processes in turn can create child processes (Fig. 1-13)
- Related processes that are cooperating to get some job done often need to communicate with one another and synchronize their activities
- This communication is called **interprocess communication**
- Other process system calls:
  - request more memory (or release unused memory)
  - wait for a child process to terminate
  - overlay its program with a different one

# Processes (5)

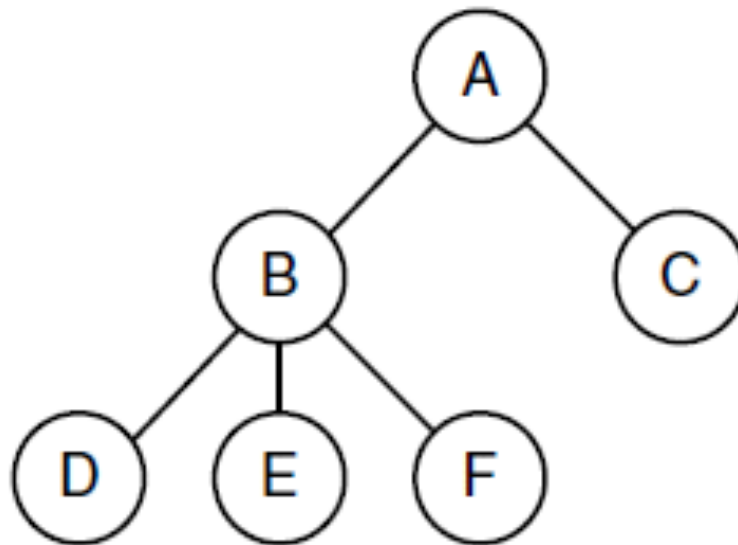


Figure 1-13. A process tree. Process A created two child processes, B and C. Process B created three child processes, D, E, and F.



# Processes (6)

- **Alarm signal:**
  - A process is communicating to another process via the network
  - A message sent to another process or a reply might be lost
  - If no acknowledgment is received the sender should retransmit it somehow
  - It asks its OS to notify it after several number of seconds
  - When the specified time is elapsed, the OS sends the alarm signal to the process which causes it to temporarily suspend, save its registers on the stack, and start running a special signal-handling procedure

# Processes (7)

- Each OS user is assigned a **UID (User IDentification)**
- Every process started by a user has a UID of a user who started it
- A child process has the same UID as its parent
- Users can be members of groups, each of which has a **GID (Group IDentification)**

# Address Space

- Normally, each process has some set of addresses it can use, typically running from 0 up to some maximum
- In the simplest case, the maximum amount of address space a process has is less than the main memory
- Usually, addresses are 32 or 64 bits, giving an address space of  $2^{32}$  or  $2^{64}$  bytes
- If a process has more address space than the computer has main memory, the OS keeps part of the address space in main memory and part on disk and shuttles pieces back and forth between them as needed
- It creates the abstraction of an address space as the set of addresses a process may reference

# System Calls for Process Management (1)

Process management

Call	Description
<code>pid = fork( )</code>	Create a child process identical to the parent
<code>pid = waitpid(pid, &amp;statloc, options)</code>	Wait for a child to terminate
<code>s = execve(name, argv, environp)</code>	Replace a process' core image
<code>exit(status)</code>	Terminate process execution and return status

Figure 1-18. Some of the major POSIX system calls. The return code *s* is -1 if an error has occurred. The return codes are as follows: *pid* is a process id, *fd* is a file descriptor, *n* is a byte count, *position* is an offset within the file, and *seconds* is the elapsed time.

# System Calls for Process Management (2)

- *fork()*:
  - creates an exact duplicate of the original process
  - all the variables have identical values at the time of the fork
  - subsequent changes in one of them do not affect the other one
  - the call returns a value, which is zero in the child and equal to the child's **PID (Process Identifier)** in the parent
  - using the returned PID, the two processes can see which one is the parent and which one is the child

# System Calls for Process Management (3)

- ***waitpid(pid, &statloc, options):***
  - in most cases, after a fork, the child will need to execute different code from the parent
  - to wait for the child to finish, the parent executes a ***waitpid*** system call, which just waits until the child (or any child if more than one exists) terminates
  - it can wait for a specific child, or for any other child by setting the first parameter to -1
  - when ***waitpid*** completes, the address pointed to by the second parameter, ***statloc***, will be set to the child process' exit status (normal or abnormal termination and exit value)
  - various options are also provided, specified by the third parameter. For example, returning immediately if no child has already exited

# System Calls for Process Management (4)

- ***execve(name, argv, environp):***
  - child process executes a command by using the ***execve*** system call, which causes its entire core image to be replaced by the file named in its first parameter
  - in the most general case, ***execve*** has three parameters:
    - the name of the file to be executed
    - a pointer to the argument array
    - a pointer to the environment array
  - various library routines, including ***execl***, ***execv***, ***execle***, and ***execve***, are provided to allow the parameters to be omitted or specified in various ways
  - we will use the name ***exec*** to represent the system call invoked by all of these

# System Calls for Process Management (5)

- ***exit(status)***:
  - processes should use ***exit*** when they are finished executing
  - this system call has one parameter, the exit status (0 to 255)
  - it is returned to the parent via *statloc* in the *waitpid* system call



# System Calls for Process Management (6)

```
#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 */
    }
}
```

Figure 1-19. A stripped-down shell. Throughout the course, *TRUE* is assumed to be defined as 1.

# System Calls for Process Management (7)

- Consider the case of a command used to copy file1 to file2:
  - `cp file1 file2`
- After the shell has forked, the child process locates and executes the file `cp` and passes to it the names of the source and target files
- The main program of most other C programs contains the declaration
  - `main(argc, argv, envp):`
    - *argc* is a count of the number of items on the command line, including the program name
    - *argv* is a pointer to an array  $i_{th}$  element of which is a pointer to the  $i_{th}$  string on the command line
    - *envp* is a pointer to an array of strings containing assignments of the form `name = value` used to pass information such as the terminal type and home directory name to programs, called **the environment**

# System Calls for Process Management (8)

- For the example above:
  - *argc* would be 3
  - *argv[0]* would point to the string “cp”
  - *argv[1]* would point to the string “file1”
  - *argv[2]* would point to the string “file2”
  - no environment is passed to the child, so the third parameter of *execve* would be a zero

# Files (1)

- File system is another key concept supported by almost any OS
- The purpose of file system is to abstract out details of manipulating with disks and other I/O devices
- System calls are needed to create, remove, read and write files
- Before a file can be read, it must be located on the disk and opened, and after being read it should be closed

# Files (2)

- A **directory** is a concept that allows grouping files together
- System calls are needed to create and remove directories, to put an existing file in a directory and to remove a file from a directory
- Directory entries can be both files and another directories. This gives us an hierarchy - a file system (Fig. 1-14)

# Files (3)

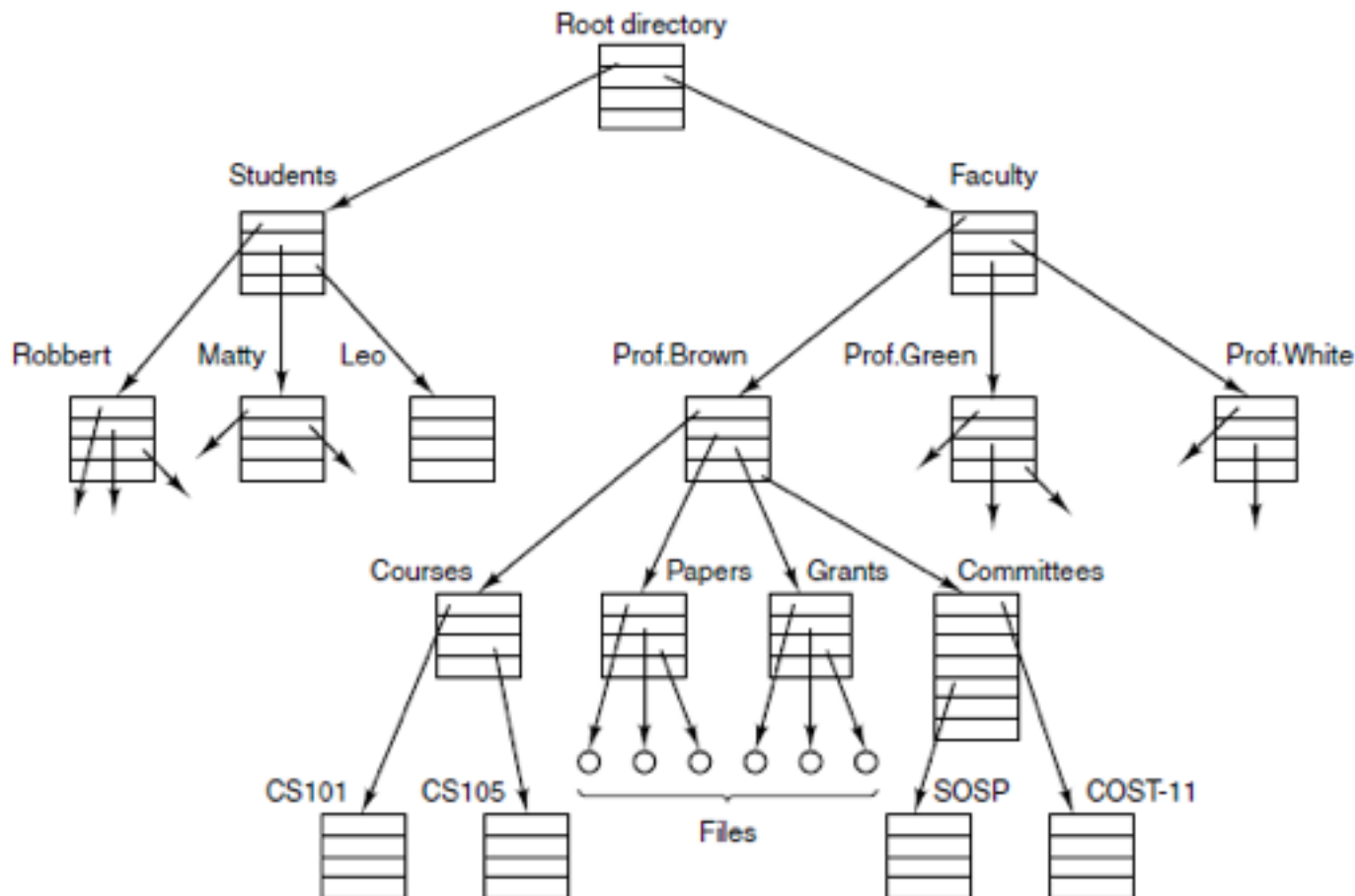


Figure 1-14. A file system for a university department.

# Files (4)

- Top of directory hierarchy is called **root directory**
- Every file within the directory hierarchy can be specified by giving its **path name** from the root directory
- For example, the path for file *CS101* is  
*/Faculty/Prof.Brown/Courses/CS101*
- Each process has a current **working directory**, in which path names not beginning with a root directory are looked for
- Before a program starts reading or writing to a file, the file must be opened, at which time the permissions are checked. The system returns a small integer called a **file descriptor** to use in subsequent operations or an error code if access is prohibited

# Files (5)

- **Mounted file system**

- Used for working with removable media which have own file system
- There is no way in UNIX to prefix path names with drive letter or number - it would cause device dependence
- ***mount*** system call allows to attach file system on removable media to the root file system (Fig. 1-15)
- In the next example the file system on the CD-ROM has been mounted on directory *b*, thus allowing access to files */b/x* and */b/y*



# Files (6)

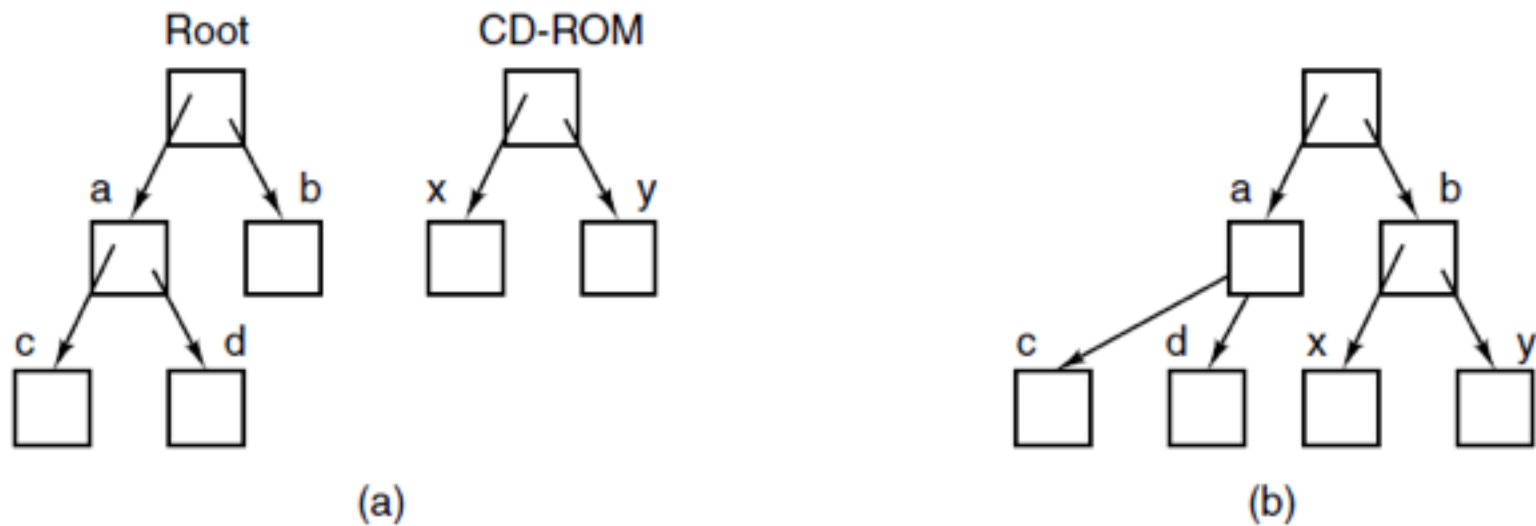


Figure 1-15. (a) Before mounting, the files on the CD-ROM are not accessible. (b) After mounting, they are part of the file hierarchy.

# Files (7)

- **Special files** are provided in order to make I/O devices look like files
- There are two kinds of special files:
  - **block special files** are used to model the devices that consist of a collection of randomly addressable blocks, such as disks. By using it, the program can access blocks on a disk without touching the file system contained on it
  - **character special files** are used to model printers, modems, and other devices that accept or output a character stream

# Files (8)

- **Pipes**

- Used as communication means between the processes (Fig. 1-16)
- When one process (A) needs to send data to another, it writes it on the pipe as it was a regular file
- Another process (B) reads from the pipe as it was an input file
- The only way a process can discover that the output file it is writing on is not really a file, but a pipe, is by making a special system call

# Files (9)

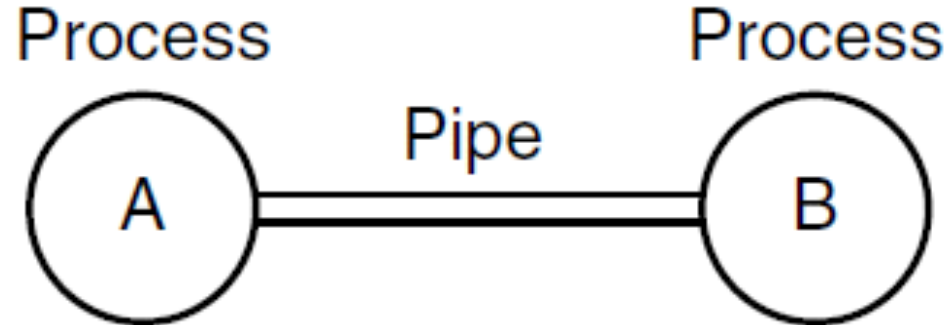


Figure 1-16. Two processes connected by a pipe.

# System Calls for File Management (1)

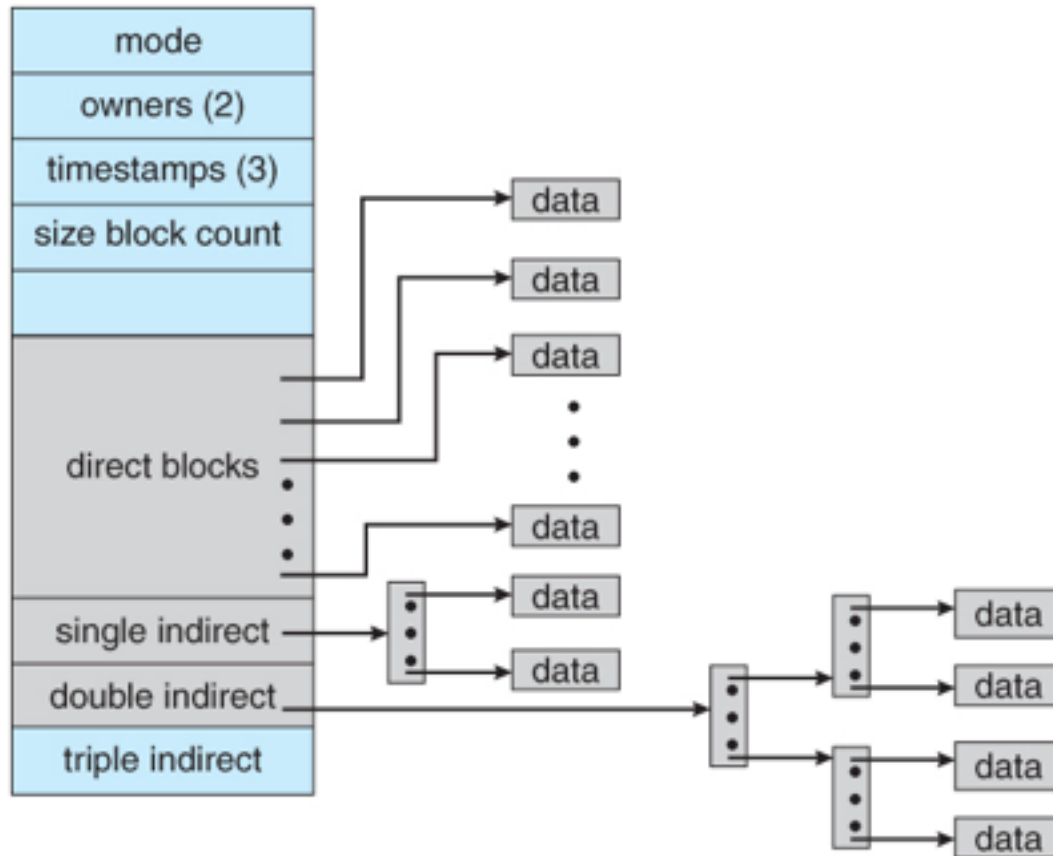
File management	
Call	Description
<code>fd = open(file, how, ...)</code>	Open a file for reading, writing, or both
<code>s = close(fd)</code>	Close an open file
<code>n = read(fd, buffer, nbytes)</code>	Read data from a file into a buffer
<code>n = write(fd, buffer, nbytes)</code>	Write data from a buffer into a file
<code>position = lseek(fd, offset, whence)</code>	Move the file pointer
<code>s = stat(name, &amp;buf)</code>	Get a file's status information

Figure 1-18. Some of the major POSIX system calls. The return code *s* is -1 if an error has occurred. The return codes are as follows: *pid* is a process id, *fd* is a file descriptor, *n* is a byte count, *position* is an offset within the file, and *seconds* is the elapsed time.

# inodes (1)

- An **inode** (or i-node) is a data structure that stores the following information about a file:
  - Size of file
  - Device ID
  - User ID of the file
  - Group ID of the file
  - The file mode information and access privileges
  - File protection flags
  - The timestamps for file creation, modification, etc.
  - Link counter to determine the number of hard links
  - Pointers to the blocks storing file's contents

# inodes (2)



## The UNIX inode structure

# inodes (3)

- In UNIX inodes the first 12 or so data block pointers are stored directly in the inode, and then singly, doubly, and triply indirect pointers provide access to more data blocks as needed
- The advantage of this scheme is that for small files (which many are), the data blocks are readily accessible (up to 48K with 4K block sizes)
- Files up to about 4144K (using 4K blocks) are accessible with only a single indirect block (which can be cached),
- Huge files are still accessible using a relatively small number of disk accesses



# inodes (4)

- When a user tries to access the file or any information related to the file then he/she uses the file name to do so, but internally the filename is first mapped with its inode number stored in a table
- The table containing a set of pairs in format (i-number, ASCII name) is stored in a file which represents a **directory**
- ***i-number*** is a unique number identifying a file in UNIX system

# inodes (5)

- The reason why file name is not stored inside an inode:
  - This is done for maintaining hard-links to files
  - Hard link is a new directory entry with a (possibly new) name, using the i-number of an existing file
  - It allows the same file to appear under two or more names, often in different directories

# inodes (6)

A terminal window with a title bar that reads 'staslitvinov — root@OSC-3: ~/hw2 — -bash — 90x24'. The terminal has a green background. The prompt is 'Stass-MacBook-Pro:~ staslitvinov\$'. The command 'ls -i' has been executed, resulting in a two-column list of files and their corresponding inode numbers. The files listed are: ACTIVstudioError.log, Applications, Applications (Parallels), Desktop, Documents, Downloads, Dropbox, Google Drive, Library, Movies, Music, Pictures, Public, aspectj1.8, eclipse, and whole.avi. The prompt 'Stass-MacBook-Pro:~ staslitvinov\$' is followed by a red cursor block.

```
Stass-MacBook-Pro:~ staslitvinov$ ls -i
11096284 ACTIVstudioError.log          425972 Library
1230658 Applications                   426024 Movies
4897067 Applications (Parallels)       426026 Music
425984 Desktop                        426028 Pictures
425968 Documents                     426030 Public
425970 Downloads                     8060267 aspectj1.8
2224962 Dropbox                      2977412 eclipse
2960325 Google Drive                 10891959 whole.avi
Stass-MacBook-Pro:~ staslitvinov$
```

`ls -i` command. Prints inode number for each file

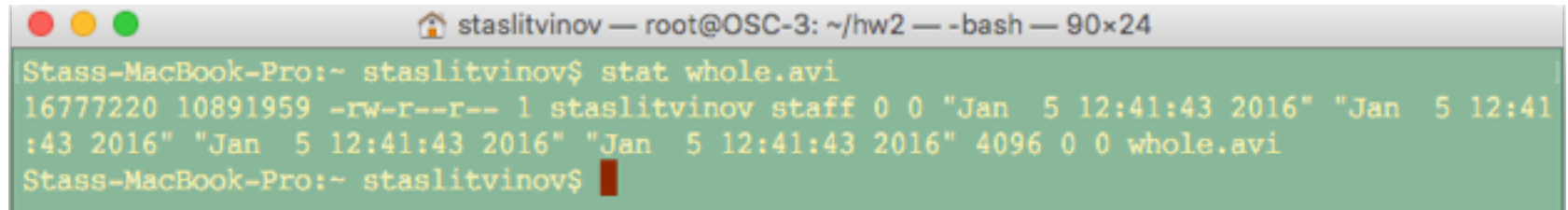
# inodes (7)

```
Stass-MacBook-Pro:~ staslitvinov$ df -i
Filesystem      512-blocks      Used Available Capacity    iused    ifree %iused  Mounted on
/dev/disk1     1462744832 843650280 618582552    58% 105520283 77322819    58% /
devfs           358         358         0    100%      620         0    100% /dev
map -hosts         0           0         0    100%         0         0    100% /net
map auto_home     0           0         0    100%         0         0    100% /home
Stass-MacBook-Pro:~ staslitvinov$
```

df -i command

Displays the inode information of the file system

# inodes (8)



```
staslitvinov — root@OSC-3: ~/hw2 — -bash — 90x24
Stass-MacBook-Pro:~ staslitvinov$ stat whole.avi
16777220 10891959 -rw-r--r-- 1 staslitvinov staff 0 0 "Jan  5 12:41:43 2016" "Jan  5 12:41:43 2016" "Jan  5 12:41:43 2016" "Jan  5 12:41:43 2016" 4096 0 0 whole.avi
Stass-MacBook-Pro:~ staslitvinov$
```

stat command

Displays file statistics that also displays inode number of a file

# System Calls for Directory Management (1)

Directory and file system management

Call	Description
<code>s = mkdir(name, mode)</code>	Create a new directory
<code>s = rmdir(name)</code>	Remove an empty directory
<code>s = link(name1, name2)</code>	Create a new entry, name2, pointing to name1
<code>s = unlink(name)</code>	Remove a directory entry
<code>s = mount(special, name, flag)</code>	Mount a file system
<code>s = umount(special)</code>	Unmount a file system

Figure 1-18. Some of the major POSIX system calls. The return code *s* is -1 if an error has occurred. The return codes are as follows: *pid* is a process id, *fd* is a file descriptor, *n* is a byte count, *position* is an offset within the file, and *seconds* is the elapsed time.

# System Calls for Directory Management (2)

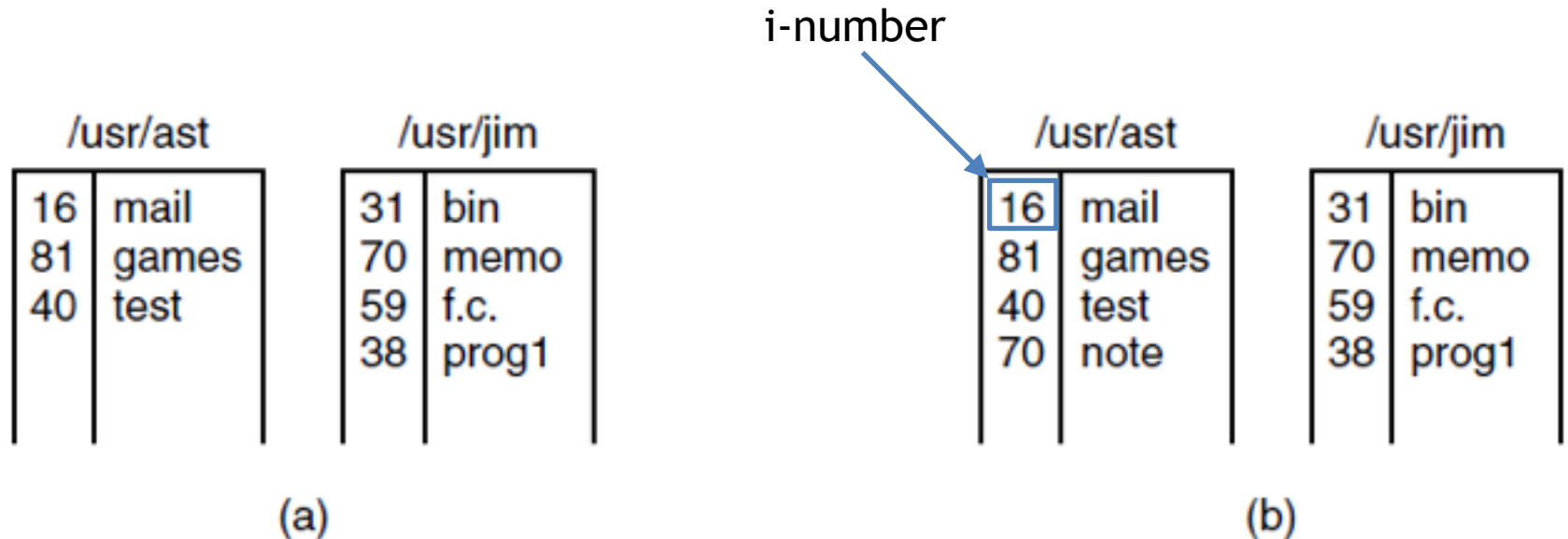


Figure 1-21. (a) Two directories before linking *usr/jim/memo* to *ast*'s directory. (b) The same directories after linking.

# System Calls for Directory Management (3)

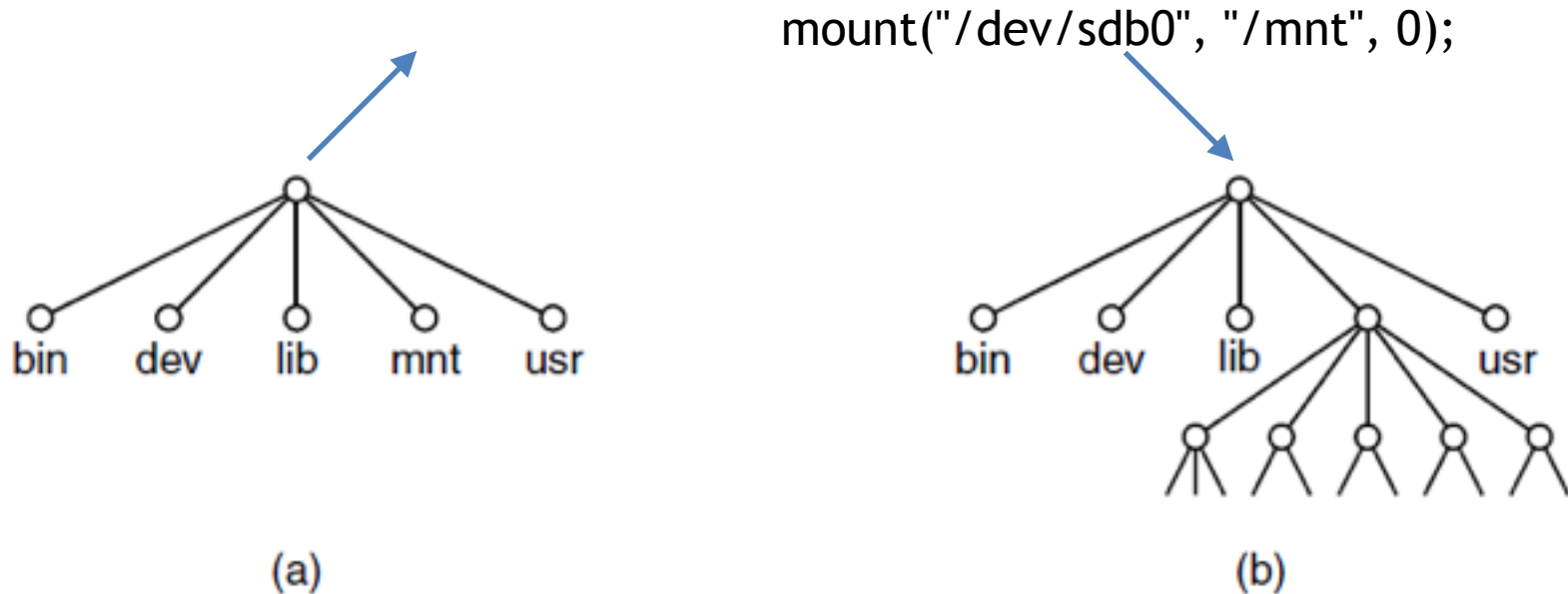


Figure 1-22. (a) File system before the mount.  
(b) File system after the mount.



# Miscellaneous System Calls (1)

Miscellaneous	
Call	Description
<code>s = chdir(dirname)</code>	Change the working directory
<code>s = chmod(name, mode)</code>	Change a file's protection bits
<code>s = kill(pid, signal)</code>	Send a signal to a process
<code>seconds = time(&amp;seconds)</code>	Get the elapsed time since Jan. 1, 1970

Figure 1-18. Some of the major POSIX system calls. The return code *s* is -1 if an error has occurred. The return codes are as follows: *pid* is a process id, *fd* is a file descriptor, *n* is a byte count, *position* is an offset within the file, and *seconds* is the elapsed time.

# Input/Output

- All computers have physical devices for acquiring input and producing output like keyboards, monitors, printers, and so on
- It is up to the OS to manage these devices
- Every OS has an I/O subsystem for managing its I/O devices. Some of the I/O software is device independent, that is, applies to many or all I/O devices equally well
- Other parts of it, such as device drivers, are specific to particular I/O devices

# Protection (1)

- Computers often contain a lot of confidential information that has to be protected
- Each OS manages security so the files are accessible only to authorized users in its own way
- For example, UNIX assigns a 9-bit binary protection code to each file
- The protection code consists of three 3-bit fields that indicate permissions for the following users:
  - owner
  - owner's group
  - any other users

# Protection (2)

- The first bit in the field indicates **read access**, the second bit indicates **write access** and the third one indicates **execute access**
- They are known as **rwX bits**
- Consider the protection code *rwXr-x--x*:
  - the owner can read, write and execute the file
  - other group members can only read or execute the file, but cannot write
  - all the other users can only execute the file
- **x** for the directory indicates search permission

End

Week 03 - Lecture 2

# References

- Tanenbaum & Bo, Modern Operating Systems: 4th edition, 2013  
Prentice-Hall, Inc.
- kill system call: <http://www.linfo.org/kill.html>
- time system call: <http://www.di.uevora.pt/~lmr/syscalls.html>
- [https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/12\\_FileSystemImplementation.html](https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/12_FileSystemImplementation.html)
- <http://www.thegeekstuff.com/2012/01/linux-inodes/>