

# [11] CASE STUDY: UNIX

# OUTLINE

- Introduction
- Design Principles
  - Structural, Files, Directory Hierarchy
- Filesystem
  - Files, Directories, Links, On-Disk Structures
  - Mounting Filesystems, In-Memory Tables, Consistency
- Summary

# INTRODUCTION

- **Introduction**
- Design Principles
- Filesystem
- Summary

# HISTORY (I)

First developed in 1969 at Bell Labs (Thompson & Ritchie) as reaction to bloated Multics. Originally written in PDP-7 asm, but then (1973) rewritten in the "new" high-level language C so it was easy to port, alter, read, etc. Unusual due to need for performance

V6 was mainly used by universities who could afford a minicomputer, but not necessarily all the software required. The first really portable OS as same source could be built for three different machines (with minor asm changes)

Bell Labs continued with V8, V9 and V10 (1989), but never really widely available because V7 pushed to Unix Support Group (USG) within AT&T

AT&T did System III first (1982), and in 1983 (after US government split Bells), System V. There was no System IV

## HISTORY (II)

By 1978, V7 available (for both the 16-bit PDP-11 and the new 32-bit VAX-11). Subsequently, two main families: AT&T "System V", currently SVR4, and Berkeley: "BSD", currently 4.4BSD

Later standardisation efforts (e.g. POSIX, X/OPEN) to homogenise

USDL did SVR2 in 1984; SVR3 released in 1987; SVR4 in 1989 which supported the POSIX.1 standard

In parallel with AT&T story, people at University of California at Berkeley (UCB) added virtual memory support to "32V" [32-bit V7 for VAX] and created 3BSD

# HISTORY (III)

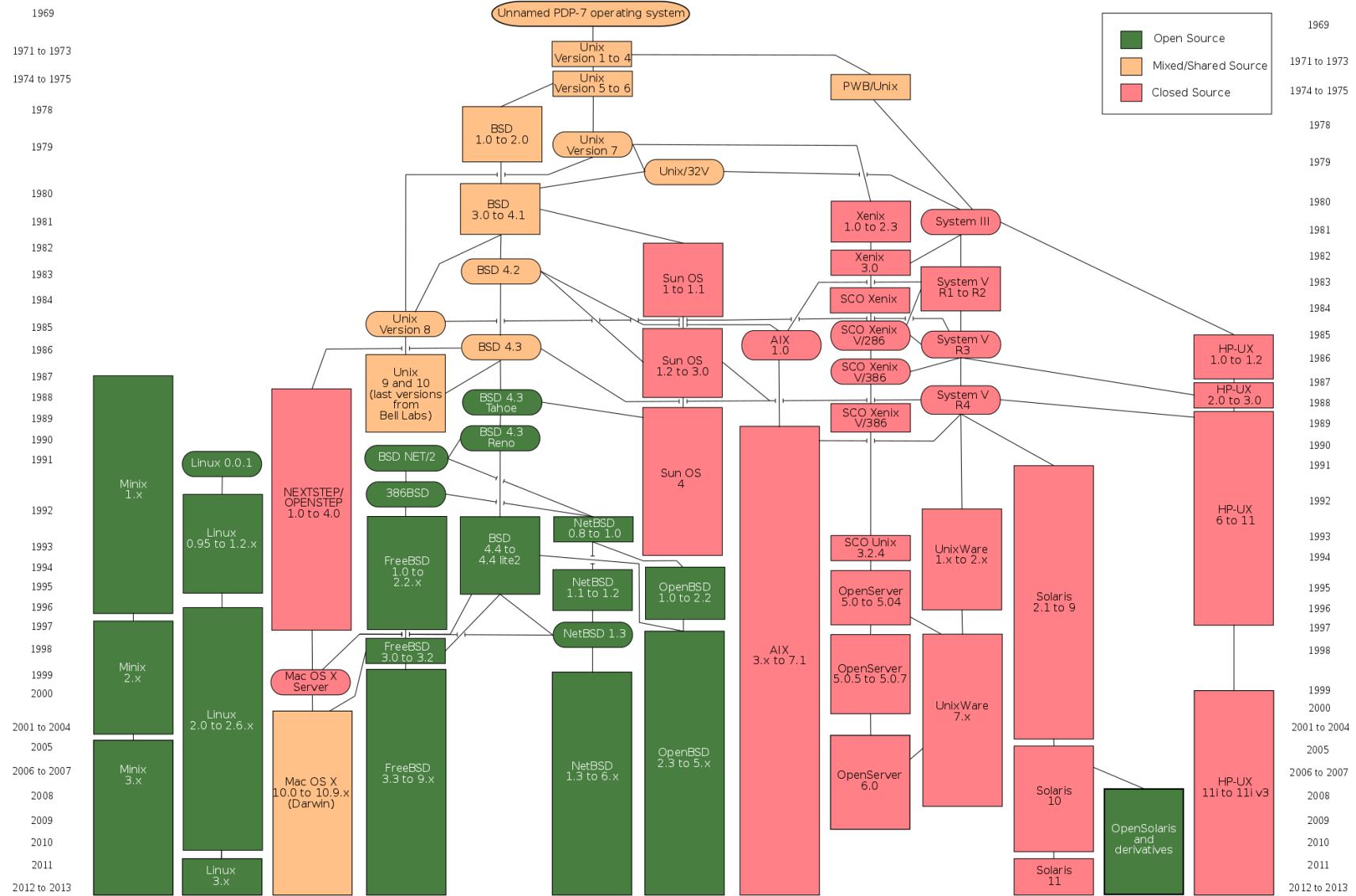
4BSD development supported by DARPA who wanted (among other things) OS support for TCP/IP

By 1983, 4.2BSD released at end of original DARPA project

1986 saw 4.3BSD released – very similar to 4.2BSD, but lots of minor tweaks. 1988 had 4.3BSD Tahoe (sometimes 4.3.1) which included improved TCP/IP congestion control. 19xx saw 4.3BSD Reno (sometimes 4.3.2) with further improved congestion control. Large rewrite gave 4.4BSD in 1993; very different structure, includes LFS, Mach VM stuff, stackable FS, NFS, etc.

Best known Unix today is probably Linux, but also get FreeBSD, NetBSD, and (commercially) Solaris, OSF/1, IRIX, and Tru64

# SIMPLIFIED UNIX FAMILY TREE (NON-EXAMINABLE)



# DESIGN PRINCIPLES

- Introduction
- Design Principles
  - Structural, Files, Directory Hierarchy
- Filesystem
- Summary

# DESIGN FEATURES

Ritchie & Thompson (CACM, July 74), identified the (new) features of Unix:

- A hierarchical file system incorporating demountable volumes
- Compatible file, device and inter-process IO (naming schemes, access control)
- Ability to initiate asynchronous processes (i.e., address-spaces = heavyweight)
- System command language selectable on a per-user basis

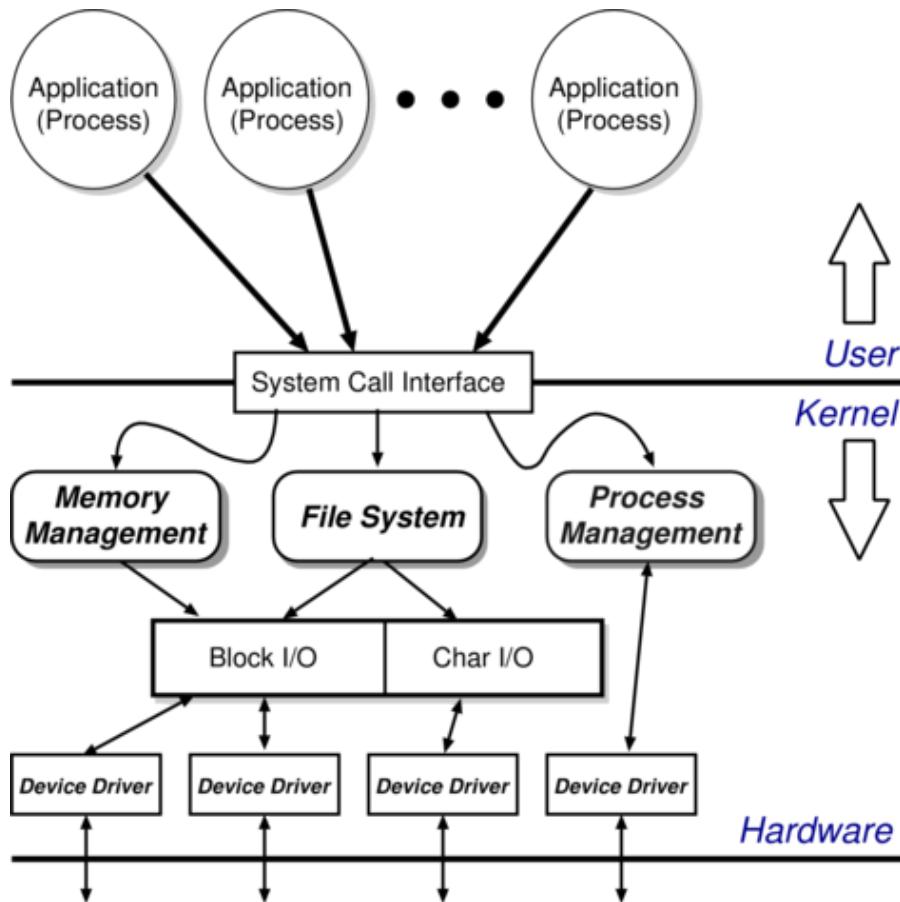
Completely novel at the time: prior to this, everything was "inside" the OS. In Unix separation between essential things (kernel) and everything else

Among other things: allows user wider choice without increasing size of core OS; allows easy replacement of functionality – resulted in over 100 subsystems including a dozen languages

Highly portable due to use of high-level language

Features which were not included: real time, multiprocessor support

# STRUCTURAL OVERVIEW



Clear separation between user and kernel portions was the big difference between Unix and contemporary systems – only the **essential** features *inside* OS, not the editors, command interpreters, compilers, etc.

Processes are unit of scheduling and protection: the command interpreter ("shell") just a process

No concurrency within kernel

All IO looks like operations on files: in Unix, everything is a file

# FILESYSTEM

- Introduction
- Design Principles
- **Filesystem**
  - **Files, Directories, Links, On-Disk Structures**
  - **Mounting Filesystems, In-Memory Tables, Consistency**
- Summary

# FILE ABSTRACTION

File as an unstructured sequence of bytes which was relatively unusual at the time: most systems lent towards files being composed of records

- Cons: don't get nice type information; programmer must worry about format of things inside file
- Pros: less stuff to worry about in the kernel; and programmer has flexibility to choose format within file!

Represented in user-space by a file descriptor (`fd`), simply an opaque identifier – a good technique for ensuring protection between user and kernel

# FILE OPERATIONS

Operations on files are:

- `fd = open(pathname, mode)`
- `fd = creat(pathname, mode)`
- `bytes = read(fd, buffer, nbytes)`
- `count = write(fd, buffer, nbytes)`
- `reply = seek(fd, offset, whence)`
- `reply = close(fd)`

The kernel keeps track of the current position within the file

Devices are represented by special files:

- Support above operations, although perhaps with bizarre semantics
- Also have `ioctl` for access to device-specific functionality

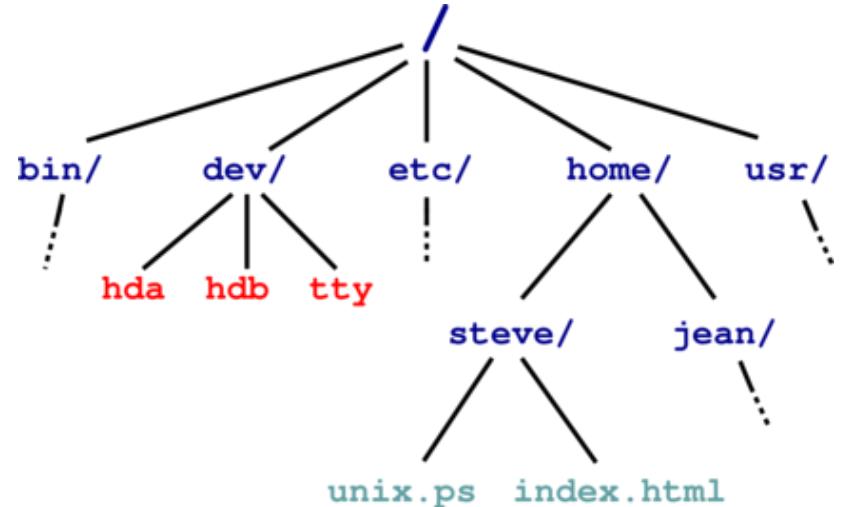
# DIRECTORY HIERARCHY

Directories map names to files (and directories) starting from distinguished root directory called /

Fully qualified pathnames mean performing traversal from root

Every directory has . and .. entries: refer to self and parent respectively. Also have shortcut of **current working directory** (cwd) which allows relative path names; and the shell provides access to home directory as ~username (e.g. ~mort/). Note that kernel knows about former but not latter

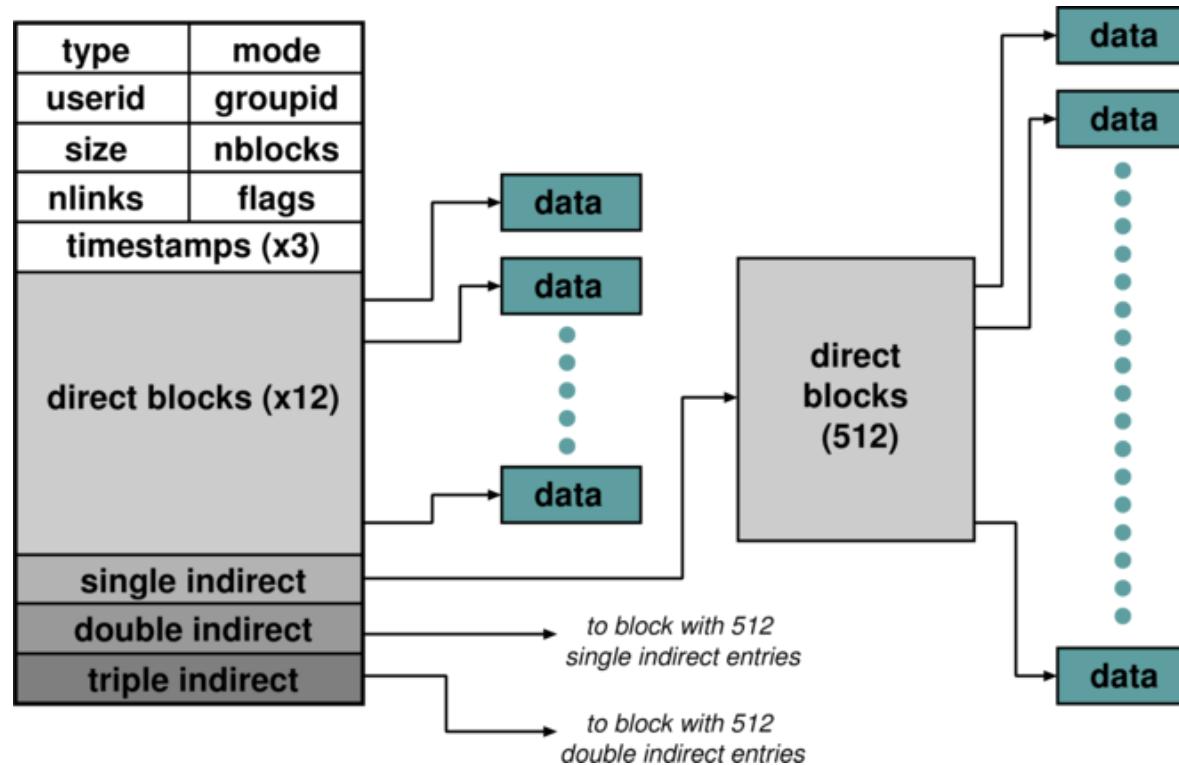
Structure is a tree in general though this is slightly relaxed



# ASIDE: PASSWORD FILE

- /etc/passwd holds list of password entries of the form `user-name:encrypted-passwd:home-directory:shell`
- Also contains user-id, group-id (default), and friendly name.
- Use one-way function to encrypt passwords i.e. a function which is easy to compute in one direction, but has a hard to compute inverse. To login:
  - Get user name
  - Get password
  - Encrypt password
  - Check against version in /etc/password
  - If ok, instantiate login shell
  - Otherwise delay and retry, with upper bound on retries
- Publicly readable since lots of useful info there but permits offline attack
- Solution: shadow passwords (/etc/shadow)

# FILE SYSTEM IMPLEMENTATION



Inside the kernel, a file is represented by a data structure called an **index-node** or **i-node** which hold file meta-data: owner, permissions, reference count, etc. and location on disk of actual data (file contents)

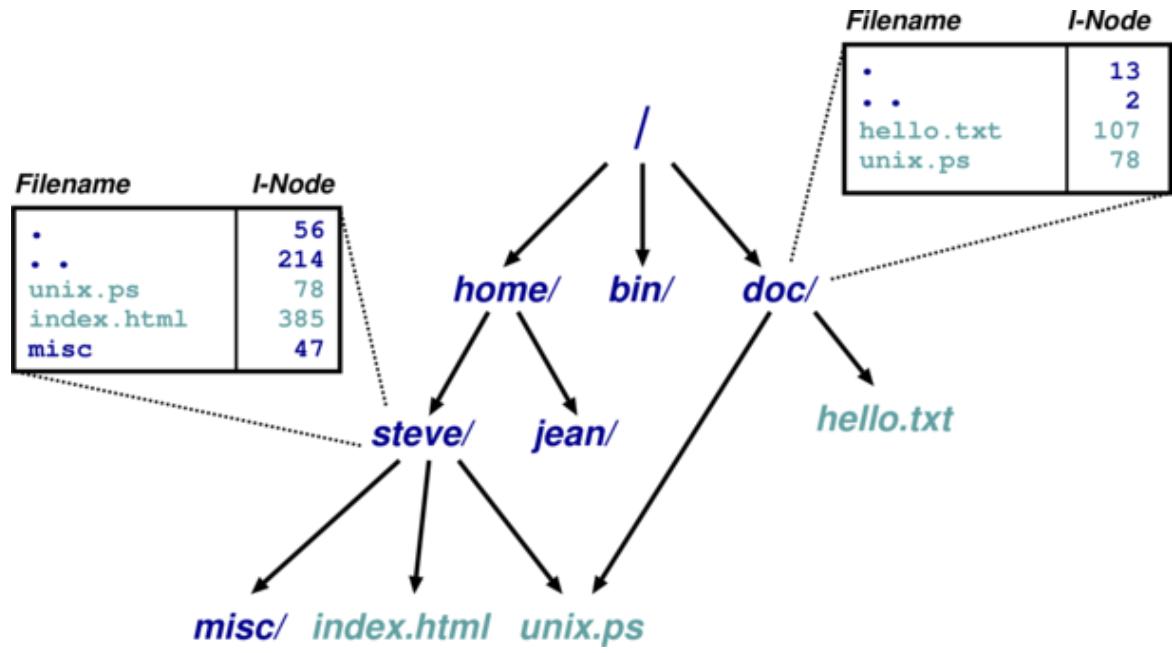
# I-NODES

- Why don't we have all blocks in a simple table?
- Why have first few in inode at all?
- How many references to access blocks at different places in the file?
- If block can hold 512 block-addresses (e.g. blocks are 4kB, block addresses are 8 bytes), what is max size of file (in blocks)?
- Where is the filename kept?

# DIRECTORIES AND LINKS

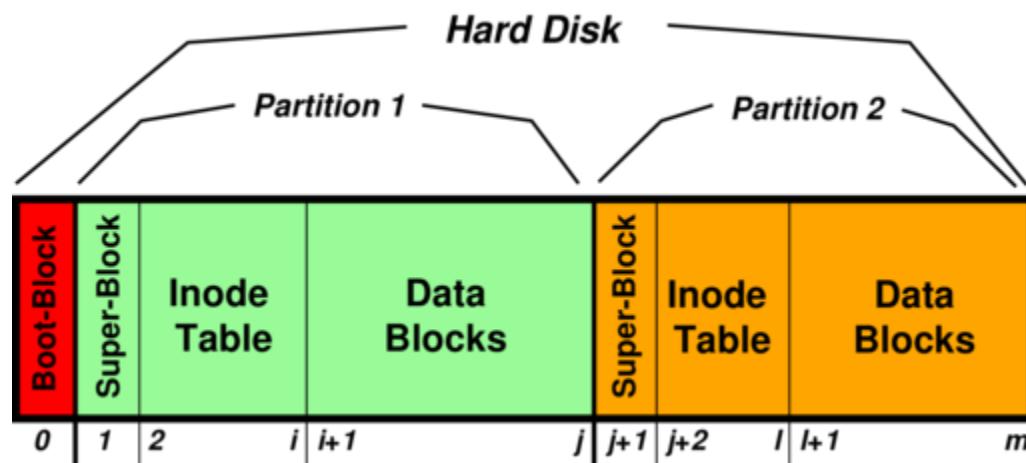
Directory is (just) a file which maps filenames to i-nodes – that is, it has its own i-node pointing to its contents

An instance of a file in a directory is a (hard) link hence the reference count in the i-node. Directories can have at most 1 (real) link. Why?



Also get soft- or symbolic-links: a 'normal' file which contains a filename

# ON-DISK STRUCTURES



A disk consists of a boot block followed by one or more partitions. Very old disks would have just a single partition. Nowadays have a **boot block** containing a partition table allowing OS to determine where the filesystems are

Figure shows two completely independent filesystems; this is not replication for redundancy. Also note  $|\text{inode table}| \gg |\text{superblock}|$ ;  $|\text{data blocks}| \gg |\text{inode table}|$

# ON-DISK STRUCTURES

A partition is just a contiguous range of  $N$  fixed-size blocks of size  $k$  for some  $N$  and  $k$ , and a Unix filesystem resides within a partition

Common block sizes: 512B, 1kB, 2kB, 4kB, 8kB

Superblock contains info such as:

- Number of blocks and free blocks in filesystem
- Start of the free-block and free-inode list
- Various bookkeeping information

Free blocks and inodes intermingle with allocated ones

On-disk have a chain of tables (with head in superblock) for each of these. Unfortunately this leaves superblock and inode-table vulnerable to head crashes so we must replicate in practice. In fact, now a wide range of Unix filesystems that are completely different; e.g., log-structure

# MOUNTING FILESYSTEMS

Entire filesystems can be mounted on an existing directory in an already mounted filesystem

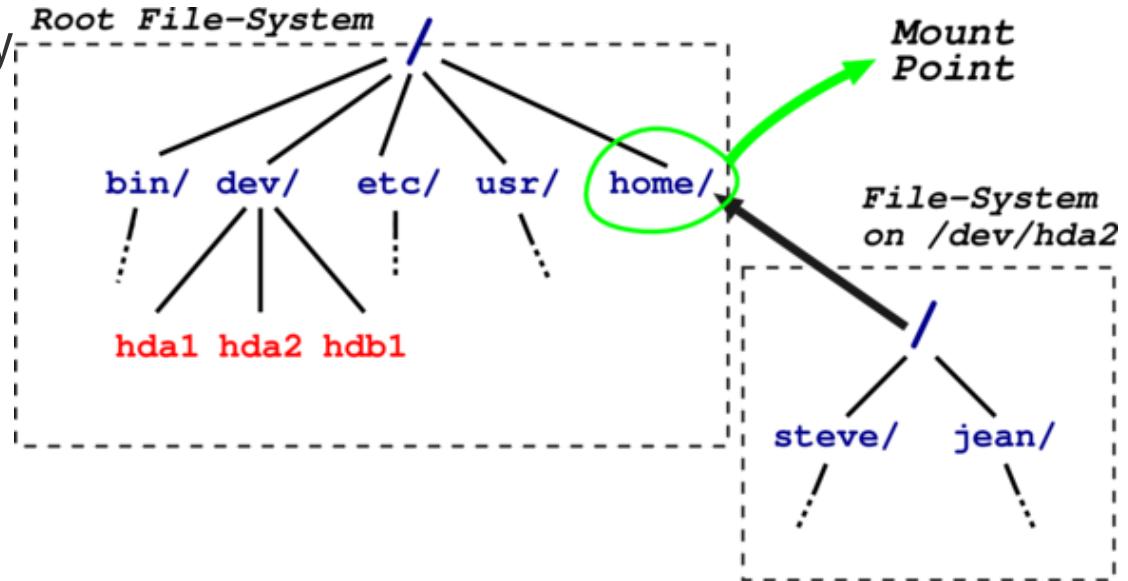
At very start, only / exists so must mount a root filesystem

Subsequently can mount other filesystems, e.g.

```
mount( "/dev/hda2" ,  
" /home" , options )
```

Provides a unified name-space: e.g. access /home/mort/ directly (contrast with Windows9x or NT)

Cannot have hard links across mount points: why? What about soft links?



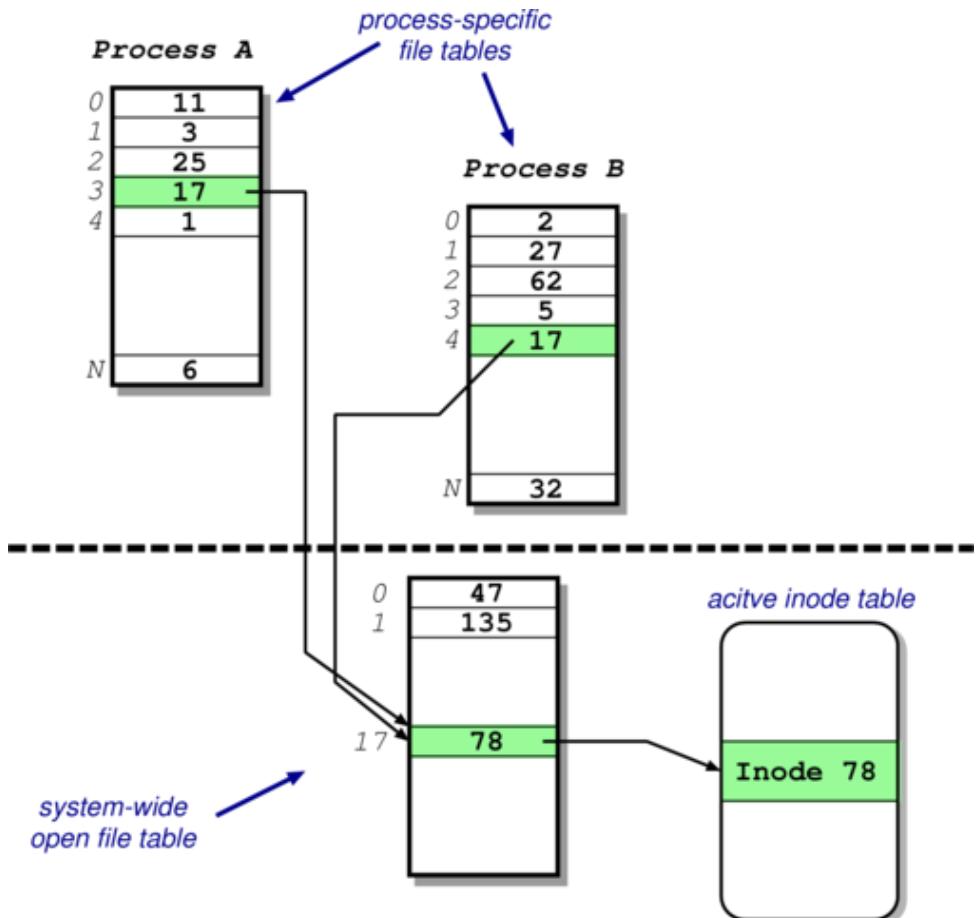
# IN-MEMORY TABLES

Recall process sees files as file descriptors

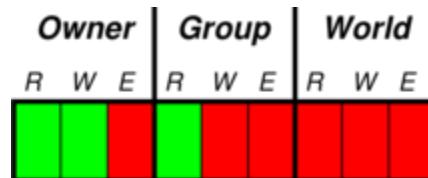
In implementation these are just indices into process-specific open file table

Entries point to system-wide open file table. Why?

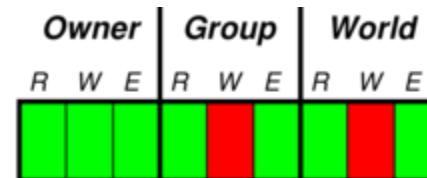
These in turn point to (in memory) inode table



# ACCESS CONTROL



= 0640



= 0755

Access control information held in each inode

- Three bits for each of owner, group and world: read, write and execute
- What do these mean for directories? Read entry, write entry, traverse directory

In addition have setuid and setgid bits:

- Normally processes inherit permissions of invoking user
- Setuid/setgid allow user to "become" someone else when running a given program
- E.g. prof owns both executable test (0711 and setuid), and score file (0600)

# CONSISTENCY ISSUES

To delete a file, use the `unlink` system call – from the shell, this is `rm <filename>`

Procedure is:

- Check if user has sufficient permissions on the file (must have write access)
- Check if user has sufficient permissions on the directory (must have write access)
- If ok, remove entry from directory
- Decrement reference count on inode
- If now zero: free data blocks and free inode

If crash: must check entire filesystem for any block unreferenced and any block double referenced

Crash detected as OS knows if crashed because root fs not unmounted cleanly

# UNIX FILESYSTEM: SUMMARY

- Files are unstructured byte streams
- Everything is a file: "normal" files, directories, symbolic links, special files
- Hierarchy built from root (/)
- Unified name-space (multiple filesystems may be mounted on any leaf directory)
- Low-level implementation based around inodes
- Disk contains list of inodes (along with, of course, actual data blocks)
- Processes see file descriptors: small integers which map to system file table
- Permissions for owner, group and everyone else
- Setuid/setgid allow for more flexible control
- Care needed to ensure consistency

# SUMMARY

- Introduction
- Design Principles
  - Structural, Files, Directory Hierarchy
- Filesystem
  - Files, Directories, Links, On-Disk Structures
  - Mounting Filesystems, In-Memory Tables, Consistency
- Summary