[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
- IO
  - Implementation, The Buffer Cache
- Processes
  - Unix Process Dynamics, Start of Day, Scheduling and States
- The Shell
  - Examples, Standard IO
- Summary

INTRODUCTION

- **Introduction**
- Design Principles
- Filesystem
- IO
- Processes
- The Shell
- 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

6th edition ("V6") was widely available (1976), including source meaning people could write new tools and nice features of other OSes promptly rolled in

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)

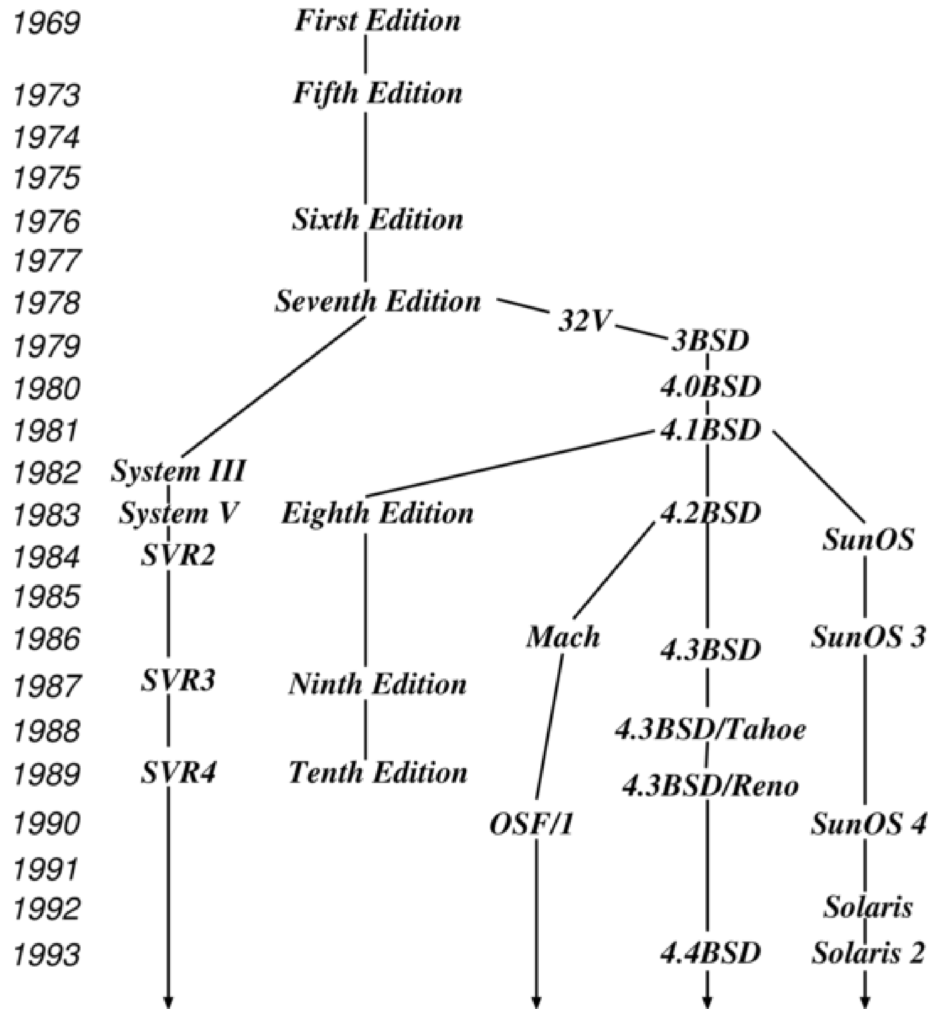
4BSD development supported by DARPA who wanted (among other things) OS support for TCP/IP

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



Linux arises (from Minix?) around 1991 (version 0.01), or more realistically, 1994 (version 1.0). Linux version 2.0 out 1996. Version 2.2 was out in 1998/ early 1999?)

You're not expected to memorise this

DESIGN PRINCIPLES

- Introduction
- **Design Principles**
    - **Structural, Files, Directory Hierarchy**
- Filesystem
- IO
- Processes
- The Shell
- 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

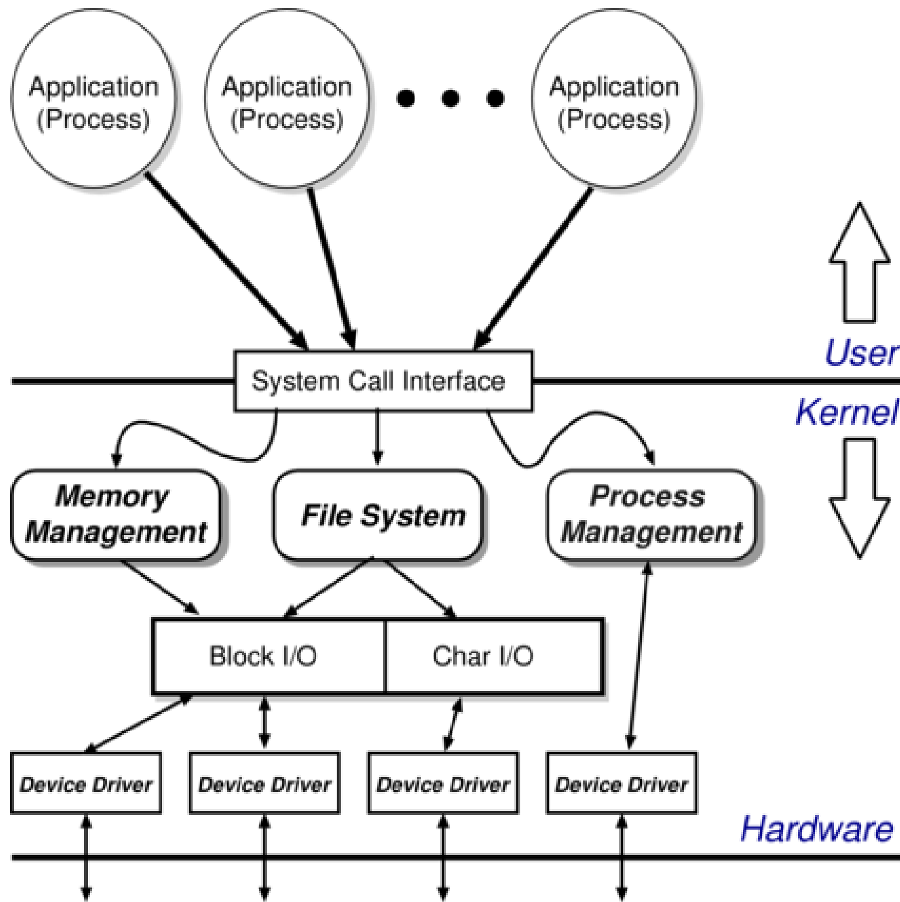
Completely novel at the time: prior to this, everything was "inside" the OS. In Unix separation between essential things (kernel) and everything else

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**
- IO
- Processes
- The Shell
- 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`) this is just an opaque identifier — a good technique for ensuring protection between user and kernel

FILE OPERATIONS

Operations on files are:

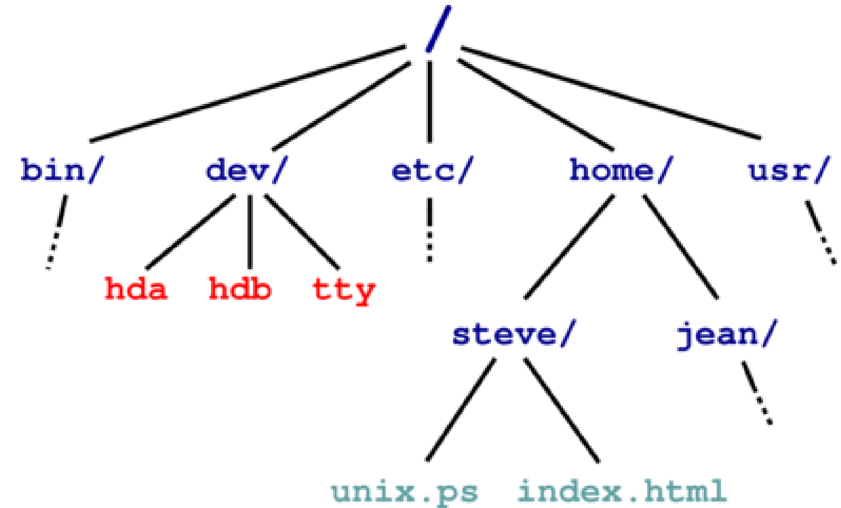
- `fd = open(pathname, mode)`
- `fd = creat(pathname, mode)`
- `bytes = read(fd, buffer, nbytes)`
- `count = write(fd, buffer, nbytes)`
- `reply = seek(fd, offset, whence)`
- `reply = close(fd)`

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

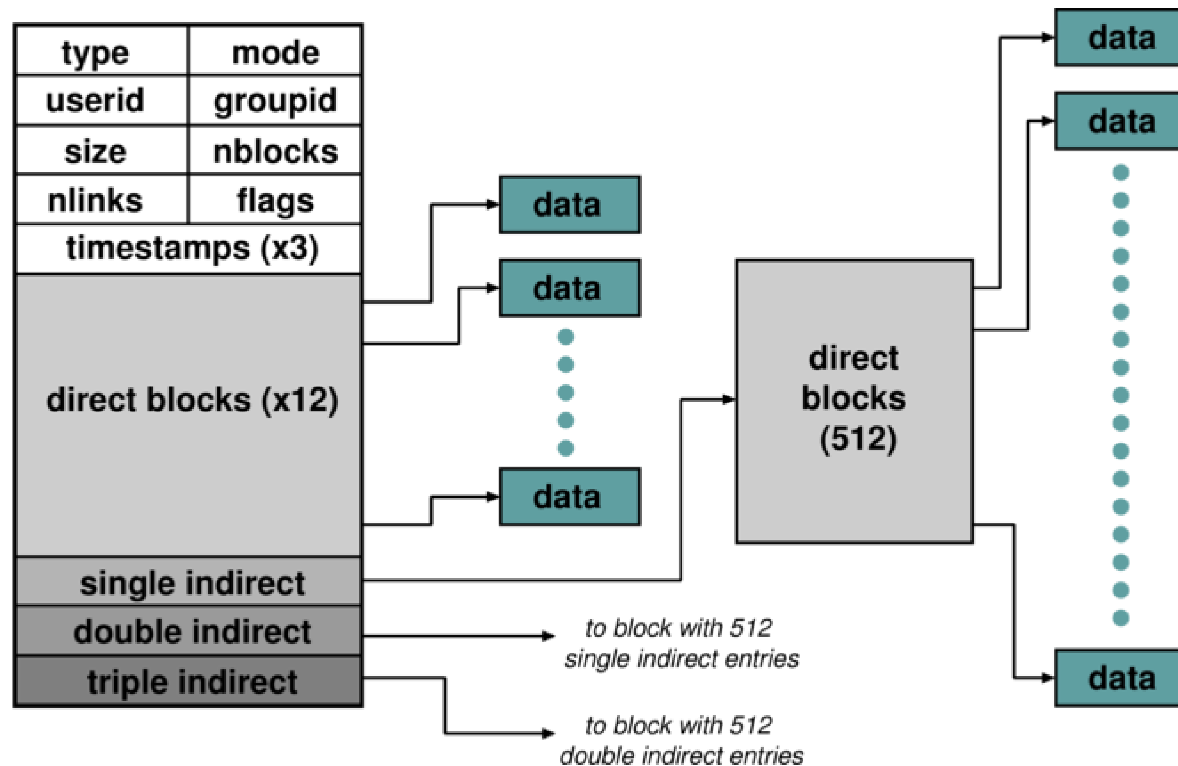
Every directory has `.` and `..` entries: refer to self and parent respectively. Also have shortcut of **current working directory** (cwd) which allows relative path names; and the shell provides access to home directory as ~username (e.g. `~mort/`). Note that kernel knows about former but not latter

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



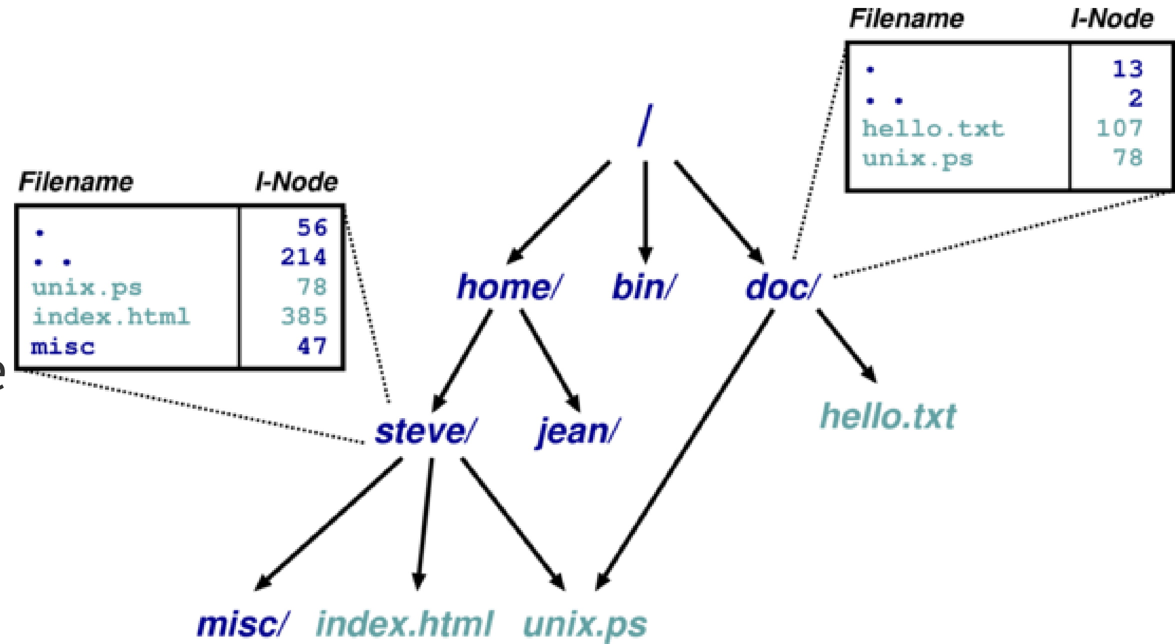
Inside the kernel, a file is represented by a data structure called an **index-node** or **i-node** which hold file meta-data: owner, permissions, reference count, etc. and location on disk of actual data (file contents)

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

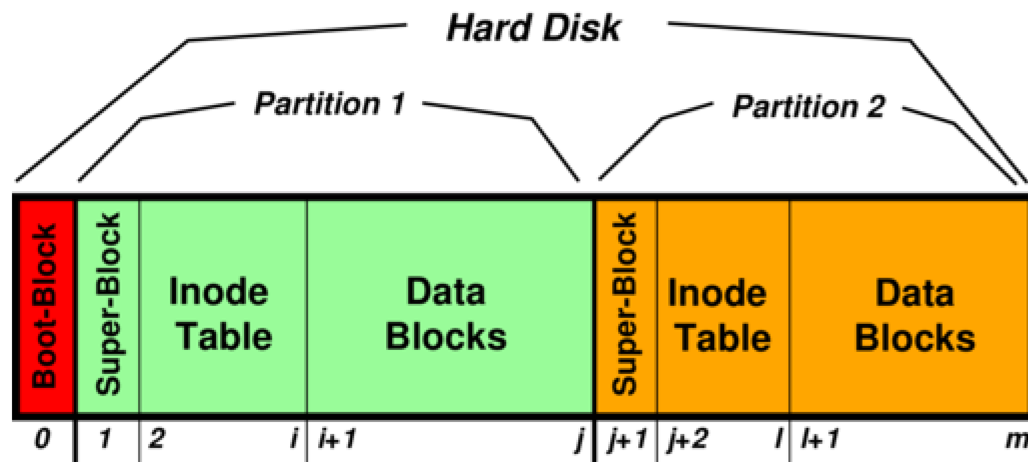
Directory is (just) a file which maps filenames to i-nodes — that is, it has its own i-node pointing to its contents



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 |inode table| ≫ |superblock|; |data blocks| ≫ |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

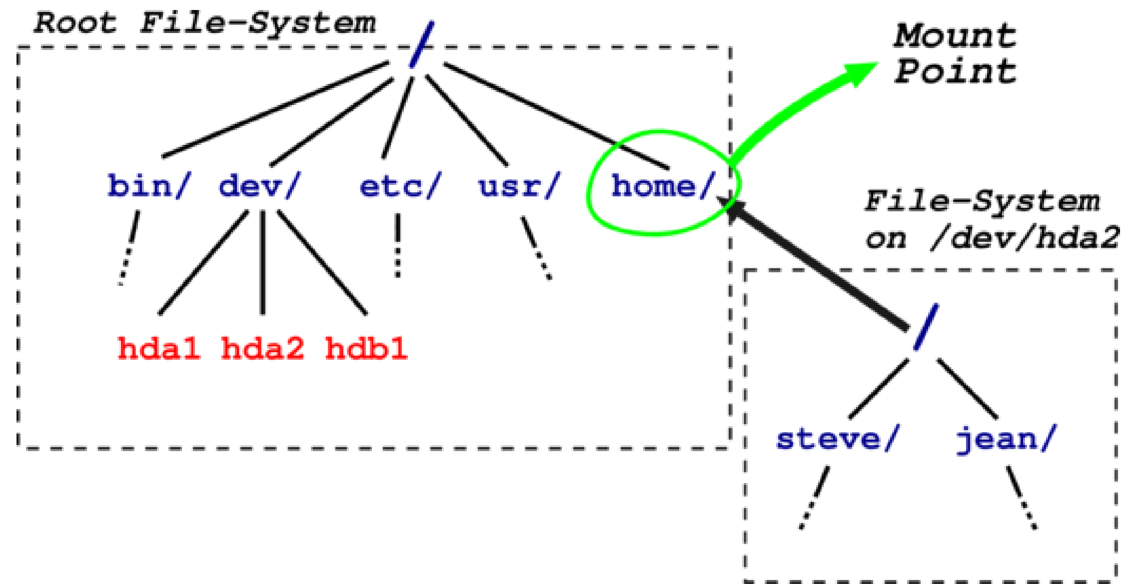
On-disk have a chain of tables (with head in superblock) for each of these. Unfortunately this leaves superblock and inode-table vulnerable to head crashes so we must replicate in practice. In fact, now a wide range of Unix filesystems that are completely different; e.g., log-structure

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?

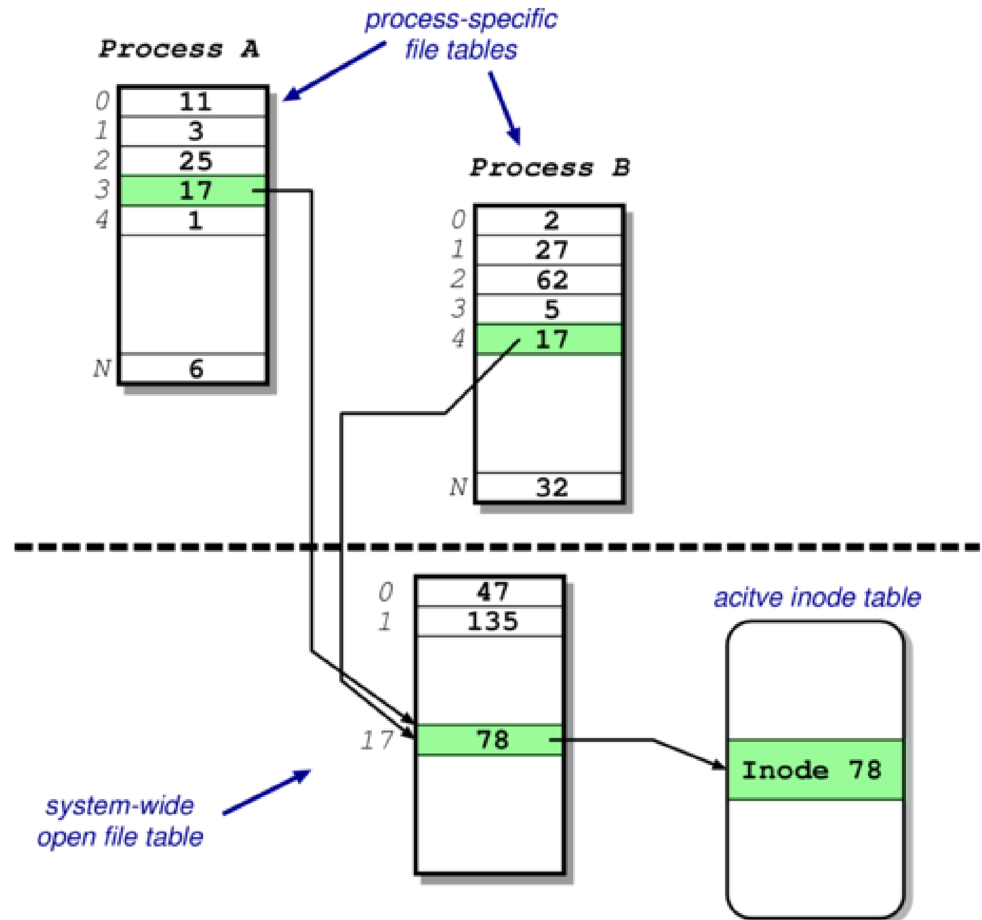
IN-MEMORY TABLES

Recall process sees files as file descriptors

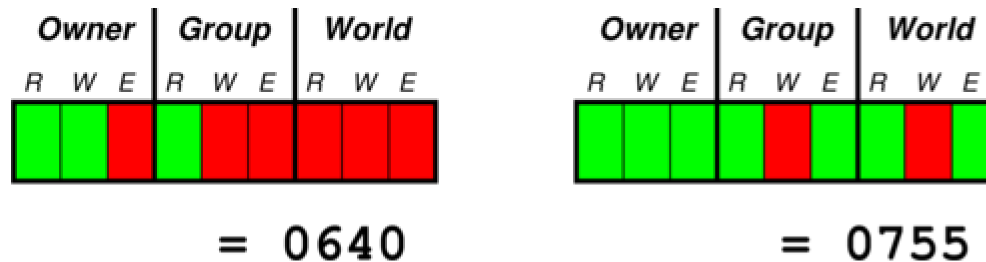
In implementation these are just indices into process-specific open file table

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

These in turn point to (in memory) inode table



ACCESS CONTROL



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 su cient permissions on the file (must have write access)
- Check if user has su cient 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

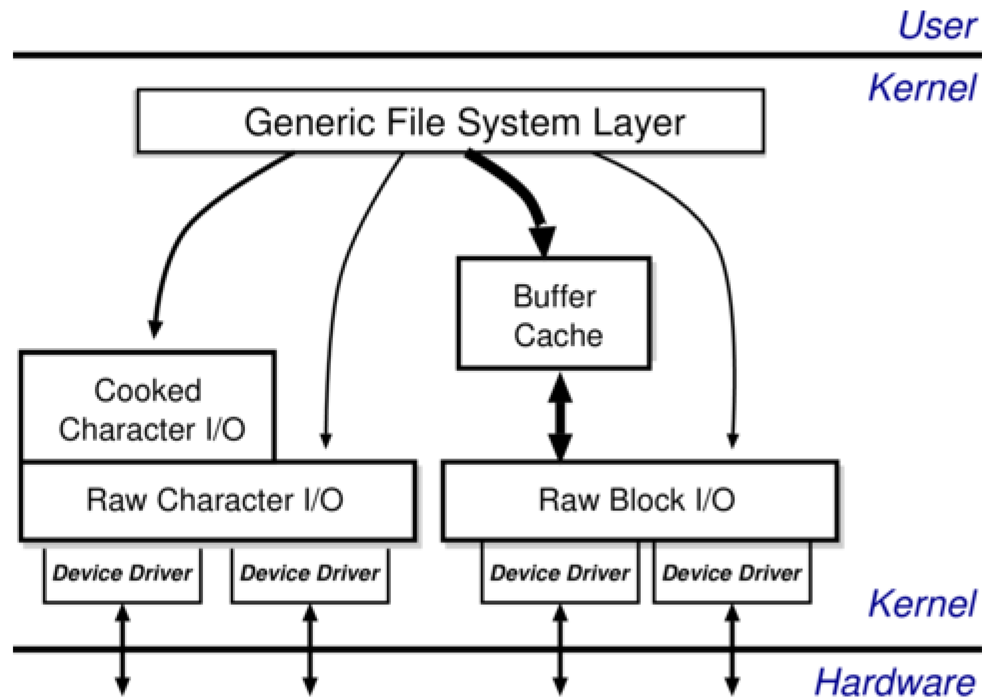
- Files are unstructured byte streams
- Everything is a file: "normal" files, directories, symbolic links, special files
- Hierarchy built from root (/)
- Unified name-space (multiple filesystems may be mounted on any leaf directory)
- Low-level implementation based around inodes
- Disk contains list 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

IO

- Introduction
- Design Principles
- Filesystem
- **IO**
  - **Implementation, The Buffer Cache**
- Processes
- The Shell
- Summary

IO IMPLEMENTATION

- Everything accessed via the file system
- Two broad categories: block and character; ignoring low-level gore:
  - Character IO low rate but complex — most functionality is in the "cooked" interface
  - Block IO simpler but performance matters — emphasis on the buffer cache



THE BUFFER CACHE

Basic idea: keep copy of some parts of disk in memory for speed

On read do:

- Locate relevant blocks (from inode)
- Check if in buffer cache
- If not, read from disk into memory
- Return data from buffer cache

On write do same first three, and then update version in cache, not on disk

- "Typically" prevents 85% of implied disk transfers
- But when does data actually hit disk?

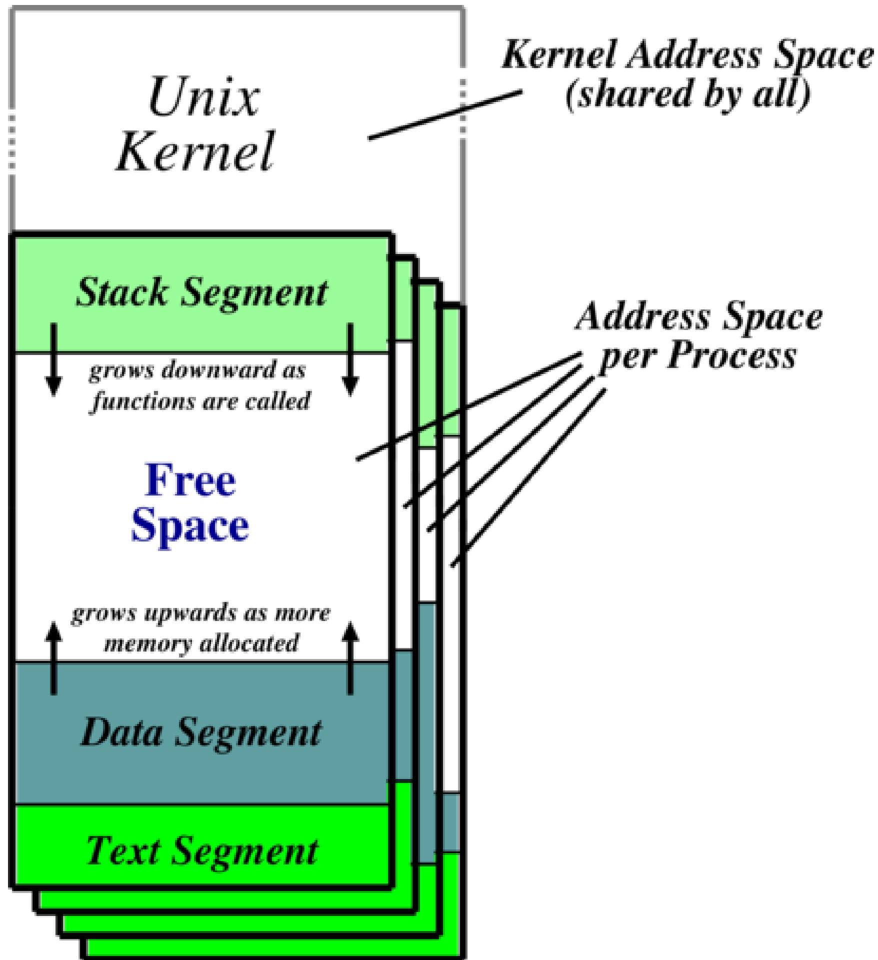
- Call `sync` every 30 seconds to flush dirty buffers to disk

- Can cache metadata too — what problems can that cause?

PROCESSES

- Introduction
- Design Principles
- Filesystem
- IO
- **Processes**
  - **Unix Process Dynamics, Start of Day, Scheduling and States**
- The Shell
- Summary

UNIX PROCESSES



Recall: a process is a program in execution

Processes have three segments: text, data and stack. Unix processes are heavyweight

Text: holds the machine instructions for the program

Data: contains variables and their values

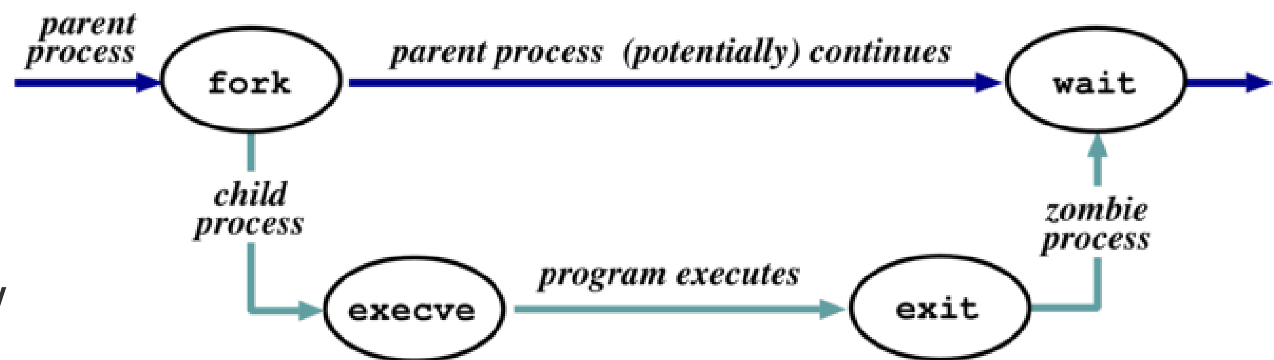
Stack: used for activation records (i.e. storing local variables, parameters, etc.)

UNIX PROCESS DYNAMICS

Process is represented by an opaque process id (`pid`), organised hierarchically with parents creating children. Four basic operations:

- `pid = `**`fork`**`()`
- `reply = `**`execve`**`(pathname, argv, envp)`
- **`exit`**`(status)`
- `pid = `**`wait`**`(status)`

`fork()` nearly always followed by `exec()` leading to `vfork()` and/or copy-on-write (COW). Also makes a copy of entire address space which is not terribly efficient



START OF DAY

Kernel (`/vmunix`) loaded from disk (how — where's the filesystem?) and execution starts. Mounts root filesystem. Process 1 (`/etc/init`) starts hand-crafted

`init` reads file `/etc/inittab` and for each entry:

- Opens terminal special file (e.g. `/dev/tty0`)
- Duplicates the resulting fd twice.
- Forks an `/etc/tty` process.

Each tty process next: initialises the terminal; outputs the string `login:` & waits for input; `execve()`'s `/bin/login`

login then: outputs "password:" & waits for input; encrypts password and checks it against `/etc/passwd`; if ok, sets uid & gid, and `execve()` shell

Patriarch init resurrects `/etc/tty` on exit

UNIX PROCESS SCHEDULING (I)

- Priorities 0-127; user processes $\geq$ PUSER = 50. Round robin within priorities, quantum 100ms.
- Priorities are based on usage and `nice`, i.e.

$$P_j(i) = \text{Base}_j + \frac{\text{CPU}_j(i-1)}{4} + 2 \times \text{nice}_j$$

  gives the priority of process *j* at the beginning of interval *i* where:

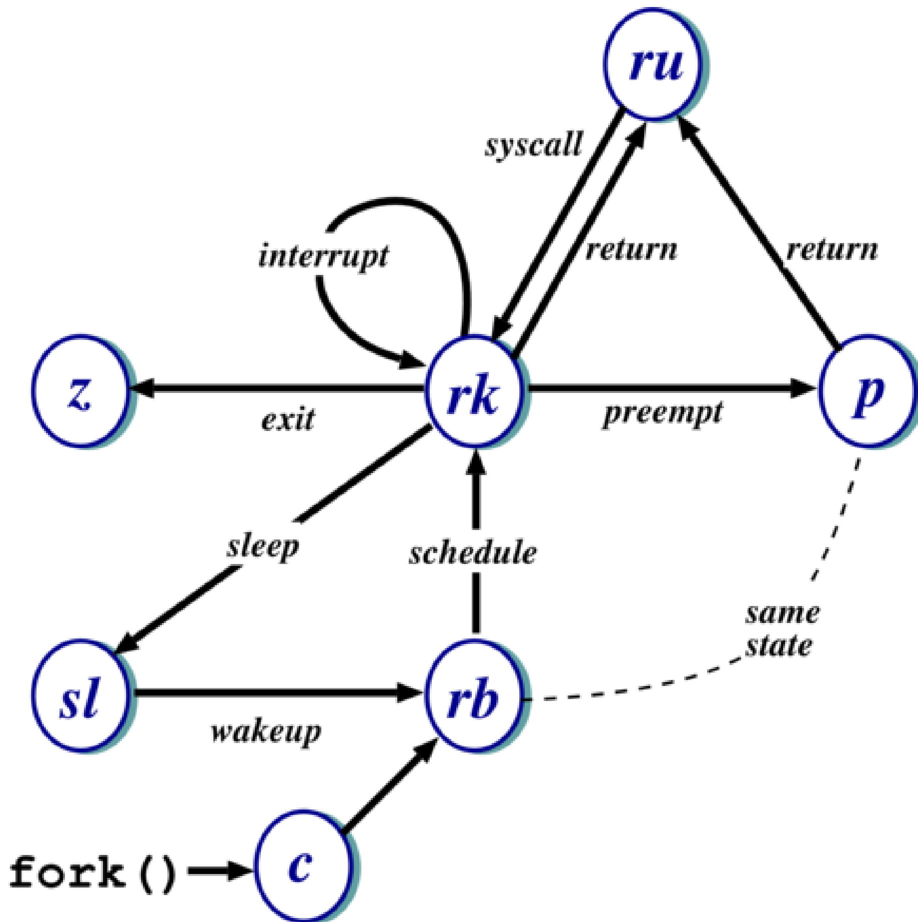
$$\text{CPU}_j(i) = \frac{2 \times \text{load}_j}{(2 \times \text{load}_j) + 1}\text{CPU}_j(i-1) + \text{nice}_j$$

  and nice_j is a (partially) user controllable adjustment parameter in the range $[-20, 20]$
- load_j is the sampled average length of the run queue in which process *j* resides, over the last minute of operation

UNIX PROCESS SCHEDULING (II)

- Thus if e.g. load is 1 this means that roughly 90% of 1s CPU usage is "forgotten" within 5s
- Base priority divides processes into bands; CPU and nice components prevent processes moving out of their bands. The bands are:
  - Swapper; Block IO device control; File manipulation; Character IO device control; User processes
  - Within the user process band the execution history tends to penalize CPU bound processes at the expense of IO bound processes

UNIX PROCESS STATES



ru	=	running (user-mode)	rk	=	running (kernel-mode)
z	=	zombie	p	=	pre-empted
sl	=	sleeping	rb	=	runnable
c	=	created			

NB. This is simplified — see *Concurrent Systems* section 23.14 for detailed descriptions of all states/transitions

THE SHELL

- Introduction
- Design Principles
- Filesystem
- IO
- Processes
- **The Shell**
  - **Examples, Standard IO**
- Summary

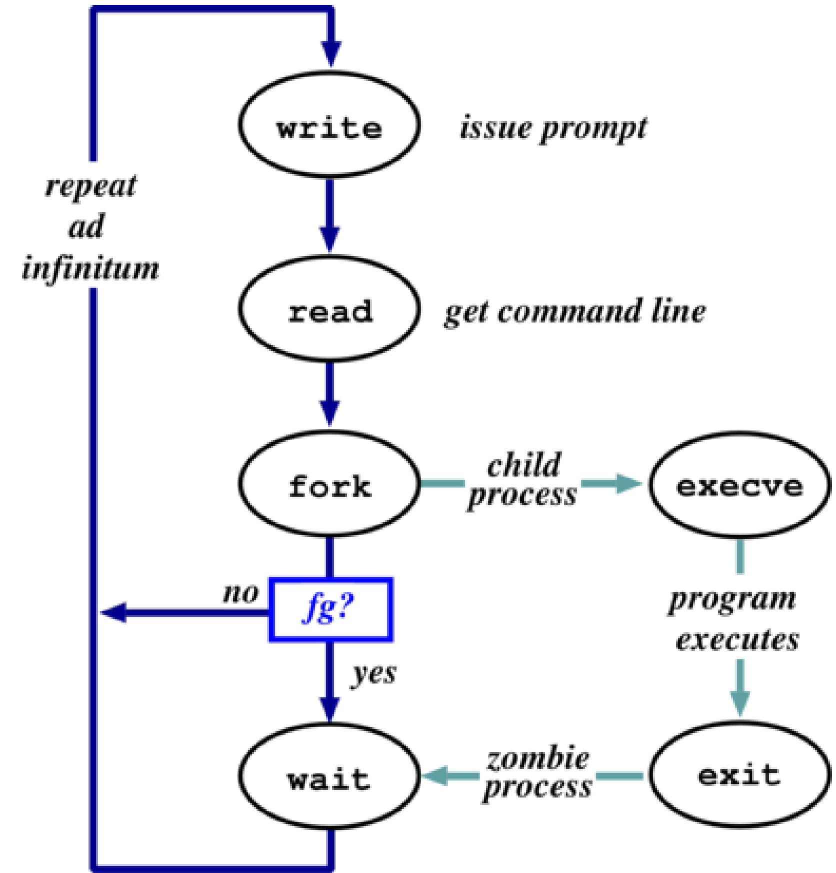
THE SHELL

Shell just a process like everything else. Needn't understand commands, just files

Uses path for convenience, to avoid needing fully qualified pathnames

Conventionally & specifies background

Parsing stage (omitted) can do lots: wildcard expansion ("globbing"), "tilde" processing



SHELL EXAMPLES

```
$ pwd
/Users/mort/src
$ ls -F
awk-scripts/    karaka/         ocamllint/          sh-scripts/
backup-scripts/ mrt.0/          opensharingtoolkit/ sockman/
bib2x-0.9.1/    ocal/           pandoc-templates/   tex/
c-utils/        ocaml/          pttcp/              tmp/
dtrace/         ocaml-libs/     pyrt/               uon/
exapraxia-gae/  ocaml-mrt/      python-scripts/         vbox-bridge/
external/       ocaml-pst/      r/
junk/           ocaml.org/      scrapers/
$ cd python-scripts/
/Users/mort/src/python-scripts
$ ls -lF
total 224
-rw-r--r--   1 mort  staff  17987  2 Jan  2010 LICENSE
-rw-rw-r--   1 mort  staff   1692  5 Jan 09:18 README.md
-rwxr-xr-x   1 mort  staff   6206  2 Dec  2013 bberry.py*
-rwxr-xr-x   1 mort  staff   7286 14 Jul  2015 bib2json.py*
-rwxr-xr-x   1 mort  staff   7205  2 Dec  2013 cal.py*
-rw-r--r--   1 mort  staff   1860  2 Dec  2013 cc4unifdef.py
-rwxr-xr-x   1 mort  staff   1153  2 Dec  2013 filebomb.py*
-rwxr-xr-x   1 mort  staff   1059  2 Jan  2010 forkbomb.py*
```

Prompt is `$`. Use `man` to find out about commands. User friendly?

STANDARD IO

Every process has three fds on creation:

- `stdin`: where to read input from
- `stdout`: where to send output
- `stderr`: where to send diagnostics

Normally inherited from parent, but shell allows redirection to/from a file, e.g.,

- `ls >listing.txt`
- `ls >&listing.txt`
- `sh <commands.sh`

Consider: `ls >temp.txt; wc <temp.txt >results`

- Pipeline is better (e.g. `ls | wc >results`)
- Unix commands are often filters, used to build very complex command lines
- Redirection can cause some buffering subtleties

SUMMARY

- Introduction
- Design Principles
- Filesystem
- IO
- Processes
- The Shell
- **Summary**

MAIN UNIX FEATURES

- File abstraction
  - A file is an unstructured sequence of bytes
  - (Not really true for device and directory files)
- Hierarchical namespace
  - Directed acyclic graph (if exclude soft links)
  - Thus can recursively mount filesystems
- Heavy-weight processes
- IO: block and character
- Dynamic priority scheduling
  - Base priority level for all processes
  - Priority is lowered if process gets to run
  - Over time, the past is forgotten
- But V7 had inflexible IPC, ine cient memory management, and poor kernel concurrency
- Later versions address these issues.

SUMMARY

- Introduction
- Design Principles
    - Structural, Files, Directory Hierarchy
- Filesystem
    - Files, Directories, Links, On-Disk Structures
    - Mounting Filesystems, In-Memory Tables, Consistency
- IO
    - Implementation, The Buffer Cache
- Processes
    - Unix Process Dynamics, Start of Day, Scheduling and States
- The Shell
    - Examples, Standard IO
- Summary