CLASSIC SYSTEMS: UNIX AND THE

CS6410 | Hakim Weatherspoon

The UNIX Time-Sharing System
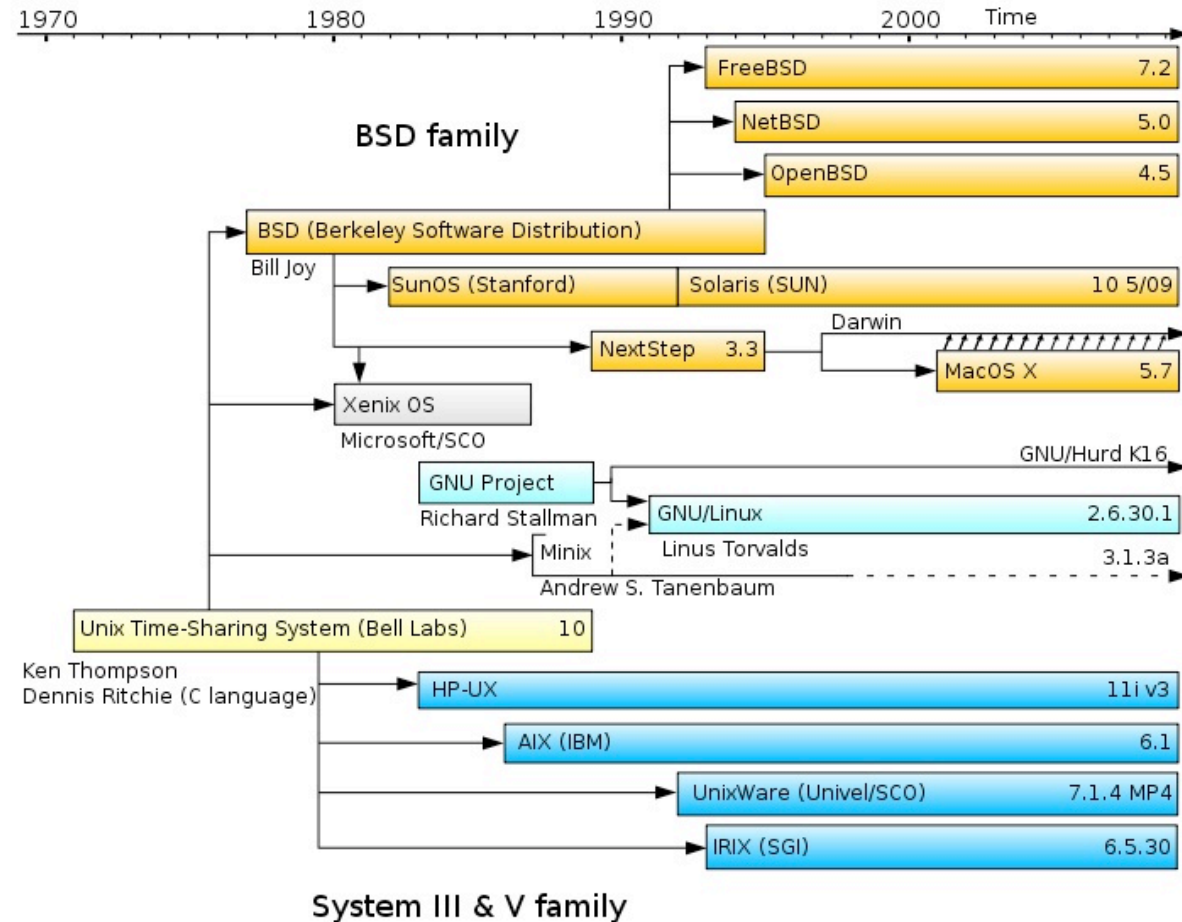# Dennis Ritchie and Ken Thompson

☐ **Background of authors at Bell Labs**

　☐ Both won Turing Awards in 1983

☐ **Dennis Ritchie**

　☐ Key developer of The C Programming Lanuage, Unix, and Multics

☐ **Ken Thompson**

　☐ Key developer of the B programming lanuage, Unix, Multics, and Plan 9

　☐ Also QED, ed, UTF-8
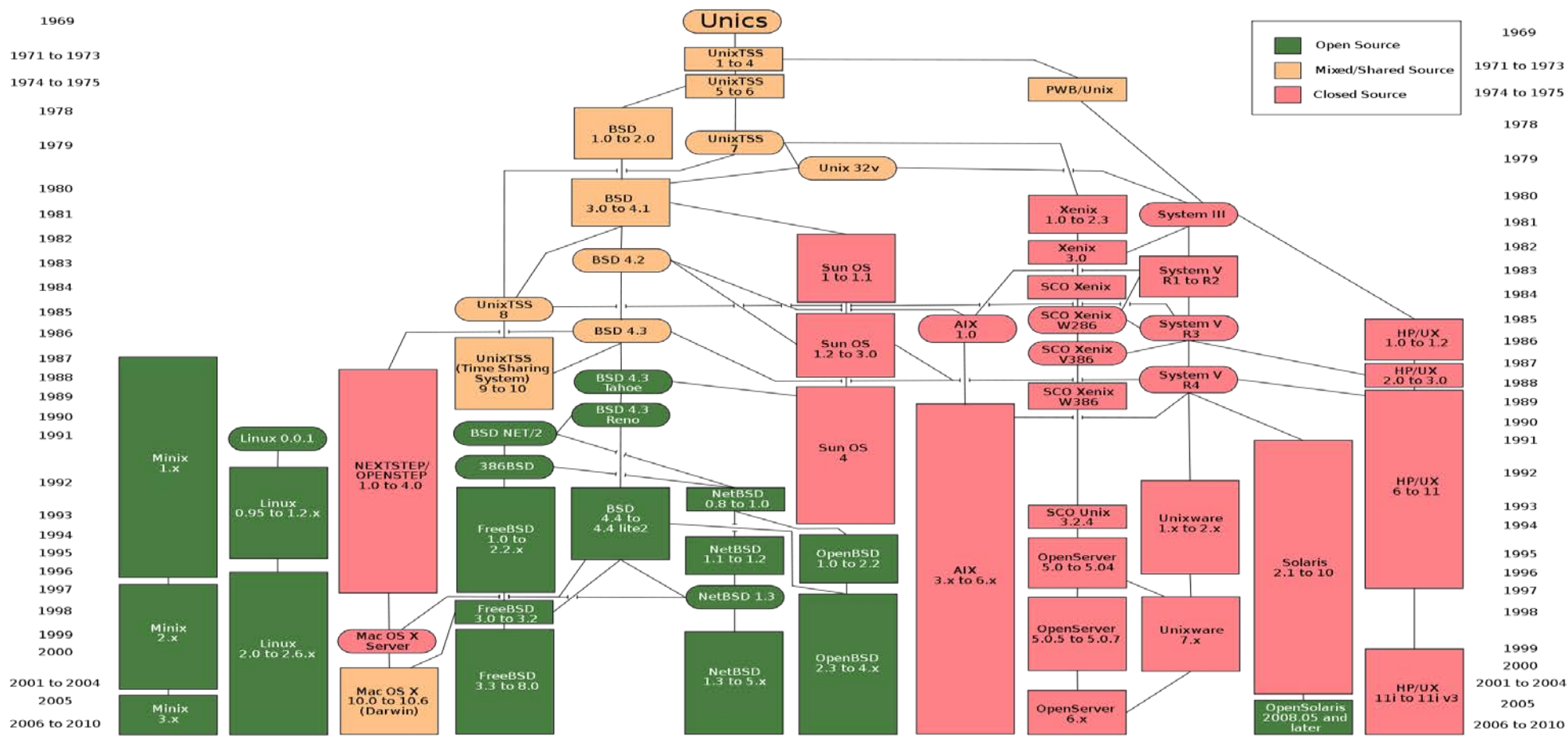
The UNIX Time-Sharing System
Dennis Ritchie and Ken Thompson

The UNIX Time-Sharing System

Dennis Ritchie and Ken Thompson

The UNIX Time-Sharing System
Dennis Ritchie and Ken Thompson

- Classic system and paper
  - described almost entirely in 10 pages

- Key idea
  - elegant combination: a few concepts that fit together well
  - Instead of a perfect specialized API for each kind of device or abstraction, the API is deliberately small

System features

- □ Time-sharing system

- □ Hierarchical file system

- □ Device-independent I/O

- □ Shell-based, tty user interface

- □ Filter-based, record-less processing paradigm

- □ Major early innovations: "fork" system call for process creation, file I/O via a single subsystem, pipes, I/O redirection to support chains

Version 3 Unix

- 1969: Version 1 ran PDP-7

- 1971: Version 3 Ran on PDP-11's
  - Costing as little as $40k!

- < 50 KB

- 2 man-years
  to write

- Written in C

PDP-7

PDP-11

File System

- Ordinary files (uninterpreted)
- Directories (protected ordinary files)
- Special files (I/O)

Uniform I/O Model

- open, close, read, write, seek
  - Uniform calls eliminates differences between devices
  - Two categories of files: character (or byte) stream and block I/O, typically 512 bytes per block
- other system calls
  - close, status, chmod, mkdir, ln
- One way to "talk to the device" more directly
  - ioctl, a grab-bag of special functionality
- lowest level data type is raw bytes, not "records"

Directories

- □ root directory

- □ path names

- □ rooted tree

- □ current working directory

- □ back link to parent

- □ multiple links to ordinary files

Special Files

- Uniform I/O model
  - Each device associated with at least one file
  - But read or write of file results in activation of device

- Advantage: Uniform naming and protection model
  - File and device I/O are as similar as possible
  - File and device names have the same syntax and meaning, can pass as arguments to programs
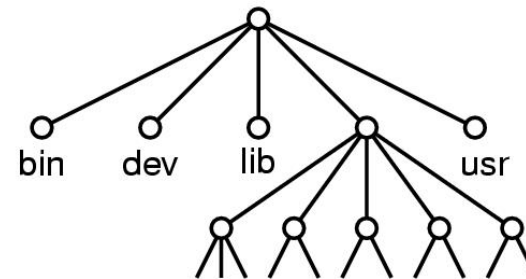 - Same protection mechanism as regular files

Removable File System

- Tree-structured

- *Mount*'ed on an ordinary file
  - Mount replaces a leaf of the hierarchy tree (the ordinary file) by a whole new subtree (the hierarchy stored on the removable volume)
  - After mount, virtually no distinction between files on permanent media or removable media

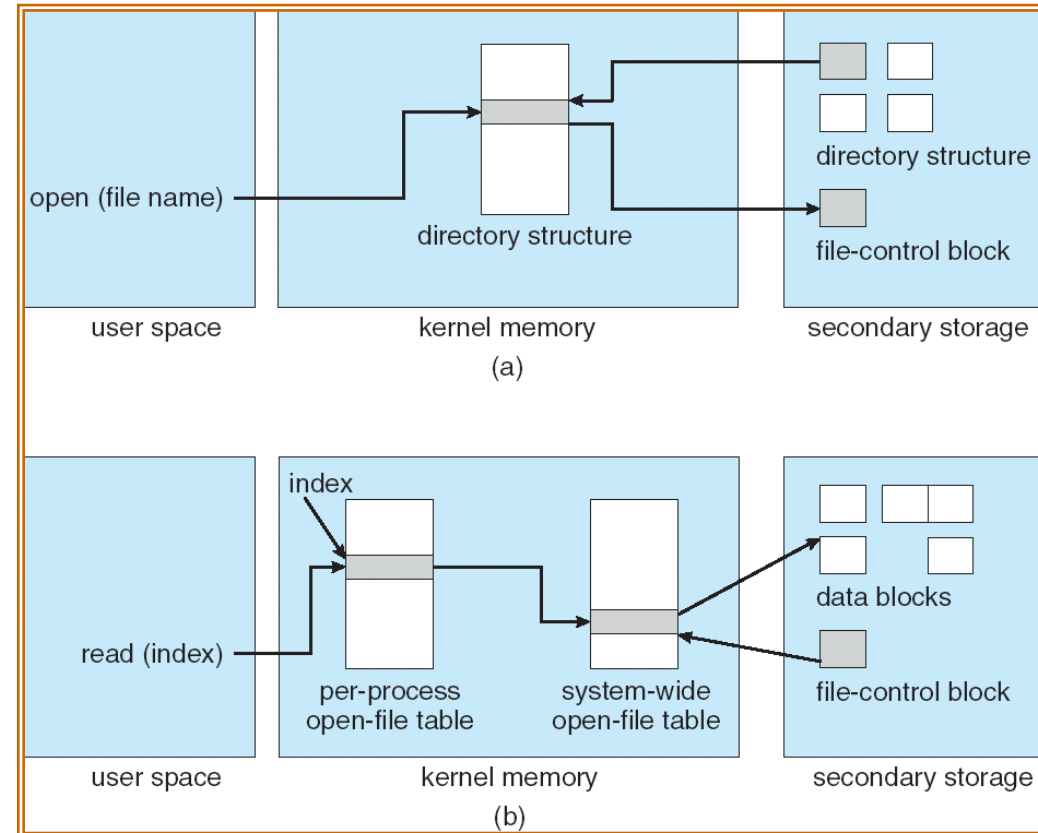(a)   (b)
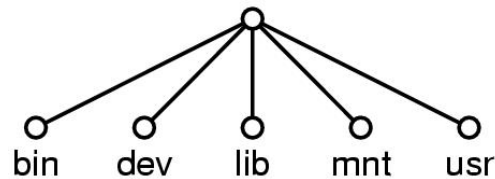
Protection

- □ User-world, RWX bits

- □ set-user-id bit

- □ super user is just special user id
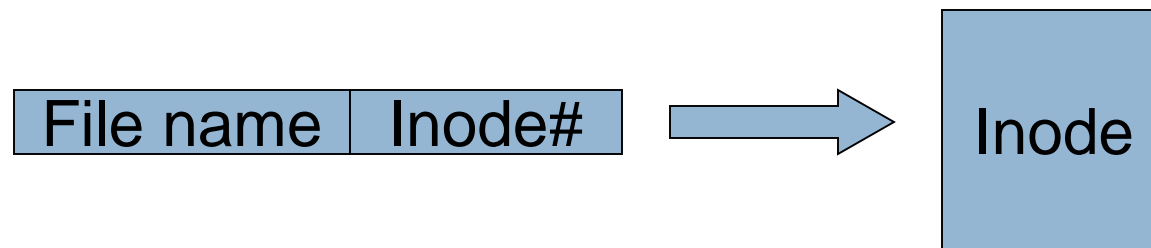
File System Implementation

- System table of i-numbers (i-list)

- i-nodes

- path names (directory is just a special file!)

- mount table

- buffered data

- write-behind

I-node Table

☐ short, unique name that points at file info.

☐ allows simple & efficient fsck

☐ cannot handle accounting issues

File name	Inode#

→ **Inode**

Many devices fit the block model

- Disks
- Drums
- Tape drives
- USB storage

- Early version of the ethernet interface was presented as a kind of block device (seek disabled)

- But many devices used IOCTL operations heavily

Processes and images

- text, data & stack segments
- process swapping
- pid = fork()
- pipes
- exec(file, arg1, ..., argn)
- pid = wait()
- exit(status)

Easy to create pipelines

- A "pipe" is a process-to-process data stream, could be implemented via bounded buffers, TCP, etc

- One process can write on a connection that another reads, allowing chains of commands

% cat *.txt | grep foo | wc

- In combination with an easily programmable shell scripting model, very powerful!

The Shell

- cmd arg1 ... argn
- stdio & I/O redirection
- filters & pipes
- multi-tasking from a single shell
- shell is just a program

- Trivial to implement in shell
  - Redirection, background processes, cmd files, etc

Traps

- Hardware interrupts

- Software signals

- Trap to system routine

Perspective

- Not designed to meet predefined objective
- Goal: create a comfortable environment to explore machine and operating system
- Other goals
  - Programmer convenience
  - Elegance of design
  - Self-maintaining

Perspective

- But had many problems too. Here are a few:
  - Weak, rather permissive security model
  - File names too short and file system damaged on crash
  - Didn't plan for threads and never supported them well
  - "Select" system call and handling of "signals" was ugly and out of character w.r.t. other features
  - Hard to add dynamic libraries (poor handling of processes with lots of "segments")
  - Shared memory and mapped files fit model poorly
- …in effect, the initial simplicity was at least partly because of some serious limitations!

Even so, Unix has staying power!

□ Today's Linux systems are far more comprehensive yet the core simplicity of Unix API remains a very powerful force

□ Struggle to keep things simple has helped keep O/S developers from making the system specialized in every way, hard to understand

□ Even with modern extensions, Unix has a simplicity that contrasts with Windows .NET API… Win32 is really designed as an internal layer that libraries invoke, but that normal users never encounter.

"THE"-Multiprogramming System
Edsger W. Dijkstra
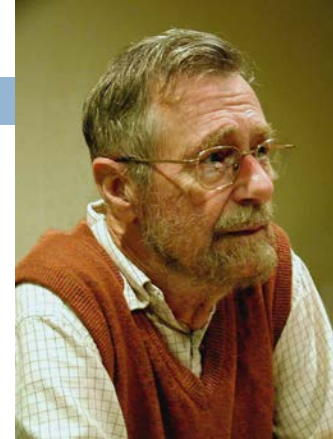
□ Received Turing Award in 1972

□ Contributions
  ▪ Shortest Path Algorithm, Reverse Polish Notation, Bankers algorithm, semaphore's, self-stabilization

□ Known for disliking 'goto' statements and using computers!

"THE"-Multiprogramming System
Edsger W. Dijkstra

- Never named "THE" system; instead, abbreviation for "Technische Hogeschool Eindhoven"

- Batch system (no human intervention) that supported multitasking (processes share CPU)
  - THE was *not* multiuser

- Introduced
  - software-based memory segmentation
  - Cooperating sequential processes
  - semaphores

Design

- Layered structure
  - Later Multics has layered structure, ring segmentation
- Layer 0 – the scheduler
  - Allocated CPU to processes, accounted for blocked proc's
- Layer 1 – the pager
- Layer 2 – communication between OS and console
- Layer 3 – managed I/O
- Layer 4 – user programs
- Layer 5 – the user
  - "Not implemented by us"!

Perspective

- Layered approach
  - Design small, well defined layers
  - Higher layers dependent on lower ones
    - Helps prove correctness
    - Helps with debugging

- Sequential process and Semaphores

Next Time

- Read and write review:
  - *Required* The Design and Implementation of a Log-Structured File System, Mendel Rosenblum and Ousterhout. Proceedings of the thirteenth *ACM symposium on Operating systems principles*, October 1991, pages 1--15.On the duality of operating system structures, H. C. Lauer and R. M. Needham. ACM SIGOPS Operating Systems Review Volume 12, Issue 2 (April 1979), pages 3--19.
  - **Optional**: A Fast File System for UNIX. Marshall K. McKusick, William N. Joy, Samuel J. Leffler, Robert S. Fabry. *ACM TOCS* 2(3), Aug 1984, pages 181--197.

Next Time

- Read and write review:

- MP0 – available later today and by Friday

- Project Proposal due end of next week
  - talk to me and other faculty and email and talk to me

- Check website for updated schedule