

# 1. Evolution of OS

Yong Zhong, Liuyi Wu, Zixu Wang

# **Part 1.**

# **Multics**

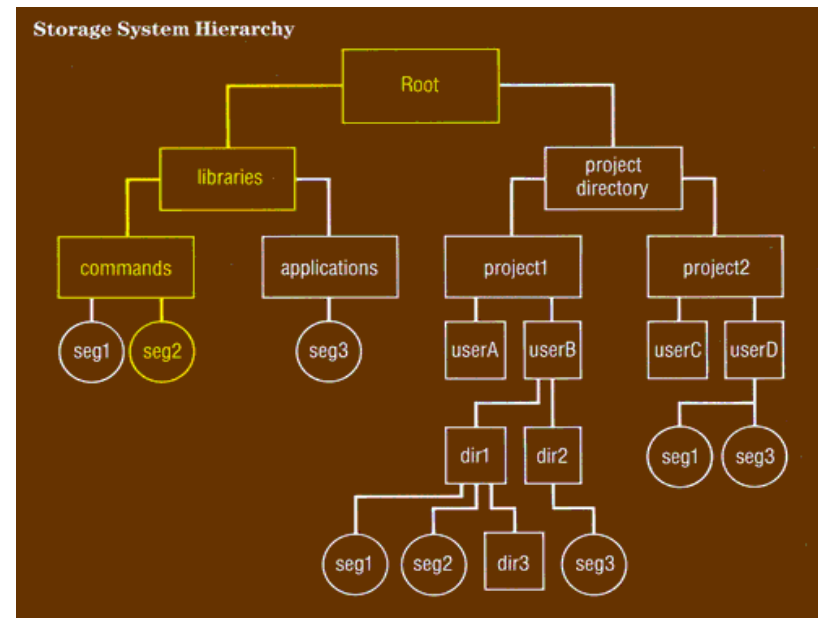
# Multics ---- history

- MAC Project (Multiple Access Computer)
  - ARPR (Advanced Research Projects Agency)
  - MIT, General Electric, Honeywell, Bell's Labs
  - MIT Prof. Robert M. Fano
- 1965 - 2000
- Bell Labs withdraws (4/1969)
  - doesn't mean it failed in 1969
- Development canceled by Honeywell (7/1985)
- Last site shut down (October 30, 2000)



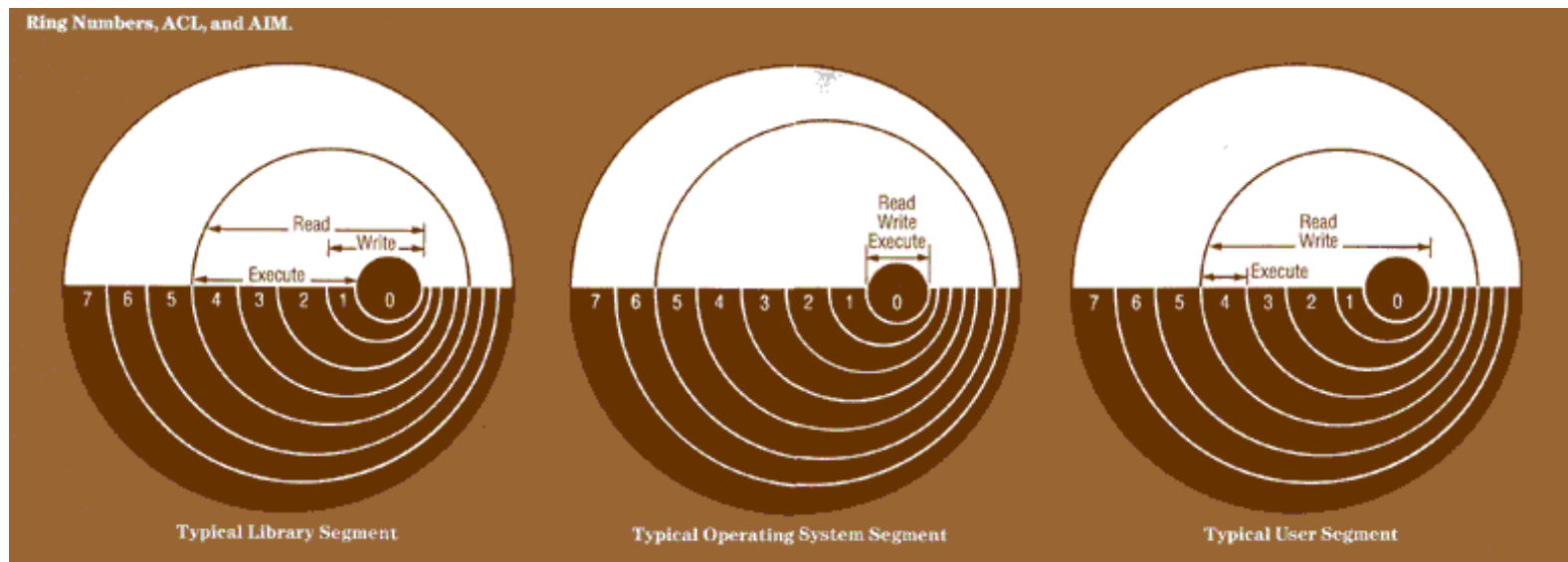
# Multics ---- great ideas

- Hierarchical File system
- Virtual Memory
- Dynamic Linkage



# Multics ---- great ideas

- ACL, Access Control List
- AIM, Access Isolation Mechanism
- Multiple rings of protection
  - recall: HarmonyOS



# Multics ---- myth

- Failure?
- Multics met most of the original goals, or at least, partially met

Goal	Accomplishment
Convenient remote terminal use.	<b>Met.</b> Set the standard.
Continuous operation analogous to <a href="#">power &amp; telephone services</a> .	<b>Partially met.</b> Never as reliable as power and phone.
A wide range of system configurations, changeable without system or user program reorganization.	<b>Partially Met.</b> A single system <a href="#">boot tape</a> worked on any Multics configuration. User programs were insulated from the configuration. It was possible to add and remove CPUs, memory, and I/O controllers without restarting the system.
A high reliability internal <a href="#">file system</a> .	<b>Met.</b> First modern file system. Hardware and software reliability improved over time.
Support for selective information sharing.	<b>Met.</b> Set the standard.
Hierarchical structures of information for system administration and decentralization of user activities.	<b>Met.</b>
Support for a wide range of applications.	<b>Met.</b>
Support for multiple programming environments & human interfaces.	<b>Met.</b>
The ability to evolve the system with changes in technology and in user aspirations.	<b>Partially Met.</b> Continued to evolve for over 20 years.

# Multics ---- a dead end

- The Second System Effect (The Mythical Man-month, Ch5)
  - The temptation to follow a simple, first system with a much more complicated second effort
- The first system: CTSS
  - spare and clean
- The second system: Multics
  - overambitious

How does the architect avoid the second-system effect? Well, obviously he can't skip his second system. But he can be conscious of the peculiar hazards of that system, and exert extra self-discipline to avoid functional ornamentation and to avoid extrapolation of functions that are obviated by changes in assumptions and purposes.

# Get more info from website

- <https://multicians.org/>

**part 2 UNIX**

# The Birth of Unix

- Many ideas from Multics.
- Successfully created at Nokia Bell Labs.

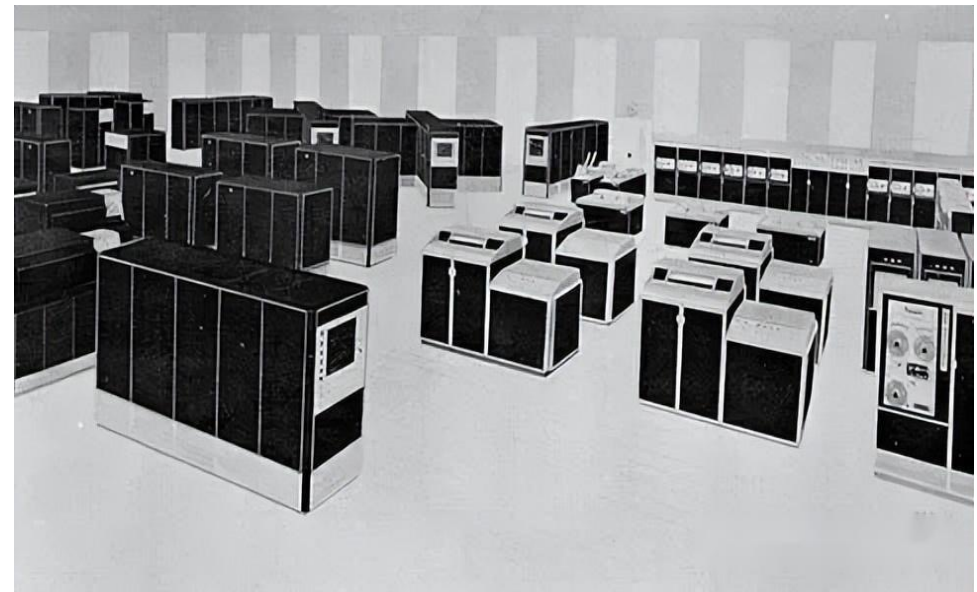


# The Story of Designer

- Ken Thompson



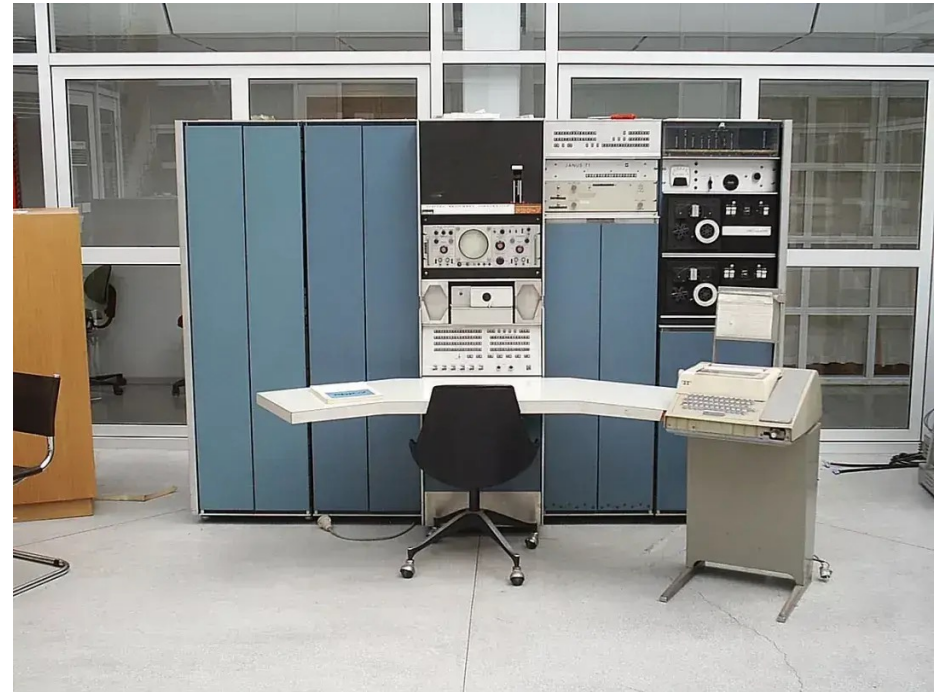
Contribution: Unix, **C Programming**,  
Go., Regular Expressions,  
UTF-8 code



GE-645 Machine

# Creation of UnICS

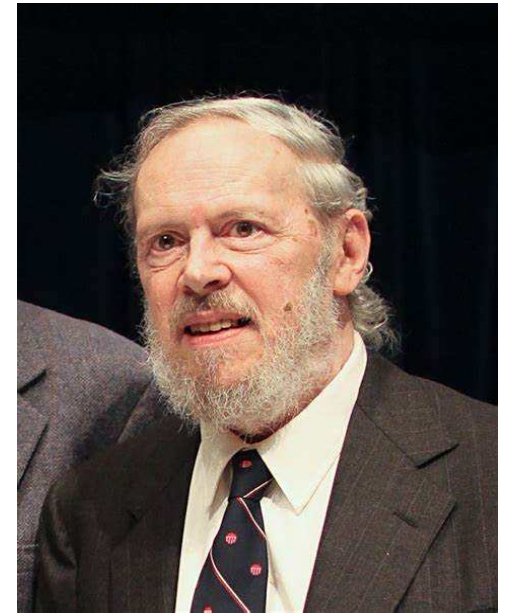
- Rewrote the Space Travel game using **assembly language**.
- To develop a new operating system.
- Created UnICS (Uniplexed Information and Computing System).



PDP-7 Machine

# The Invention of C Language

- UNIX 1.0 was written in PDP - 7 **assembly language**.
- Labs not support **BCPL** language compilers
- the **B language** was jointly invented in 1970
- However, the B language still could not be compatible with processors of **different architectural** designs.
- So he and **Dennis Ritchie** invented the C language.



# The Invention of C Language

```
GET "libhdr"  
LET start() = VALOF  
{ LET n = 0  
  n := readn()  
  writen(n + 1)  
  RESULTIS 0  
}
```

BCPL language

```
main() {  
    auto n;  
    n = getnum();  
    putnum(n + 1);  
    return(0);  
}
```

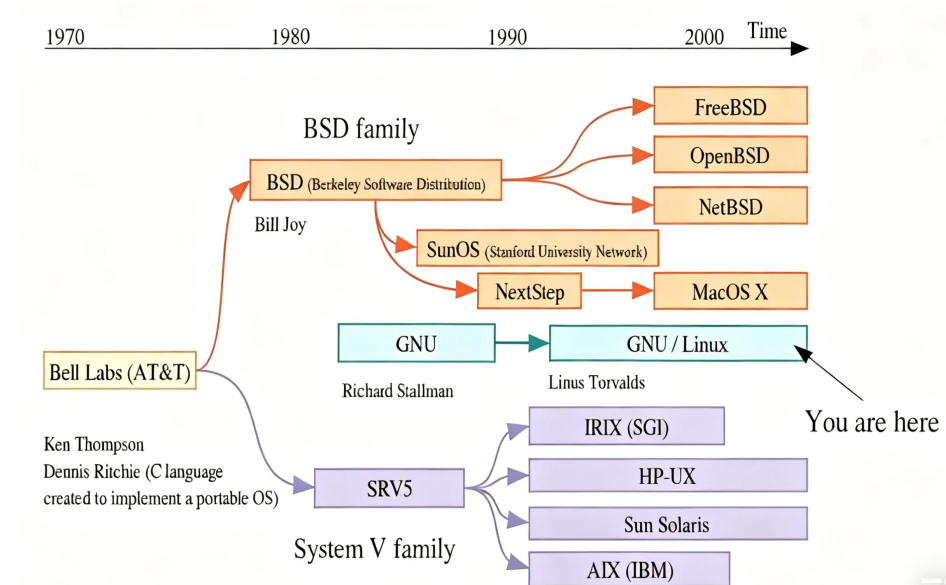
B language

```
#include <stdio.h>  
int main(void) {  
    int n;  
    scanf("%d", &n);  
    printf("%d\n", n + 1);  
    return 0;  
}
```

C language

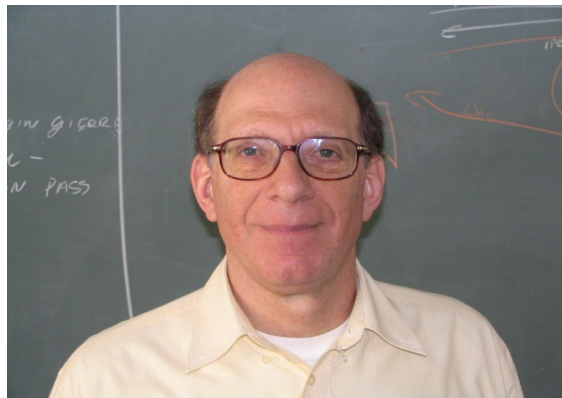
# AT&T's Use of UNIX

- In 1974, Thompson and Ritchie publish an article about UNIX in Communications of the **ACM**.
- In 1975, AT&T freely licensed Unix source code to academia for institutions and teaching.
- Berkeley Software Distribution (BSD)



# Restriction of UNIX and Origin of Linux

- Due to Unix's **commercial value**
- In 1979, the 7th - generation Unix copyright **prohibited** providing source code to academia.
- In 1987, Andrew S. Tanenbaum of the Free University of Amsterdam, Netherlands, **wrote Minix**
- Richard Stallman launched the GNU Project in 1984.
- Clone GCC, Emacs, etc.



# GNU/Linux Appears

- Linus Torvalds (U. Helsinki sophomore) wrote core program.
- Many Linux suite: Ubuntu, Apache, Postfix



# GNU/Linux Impact

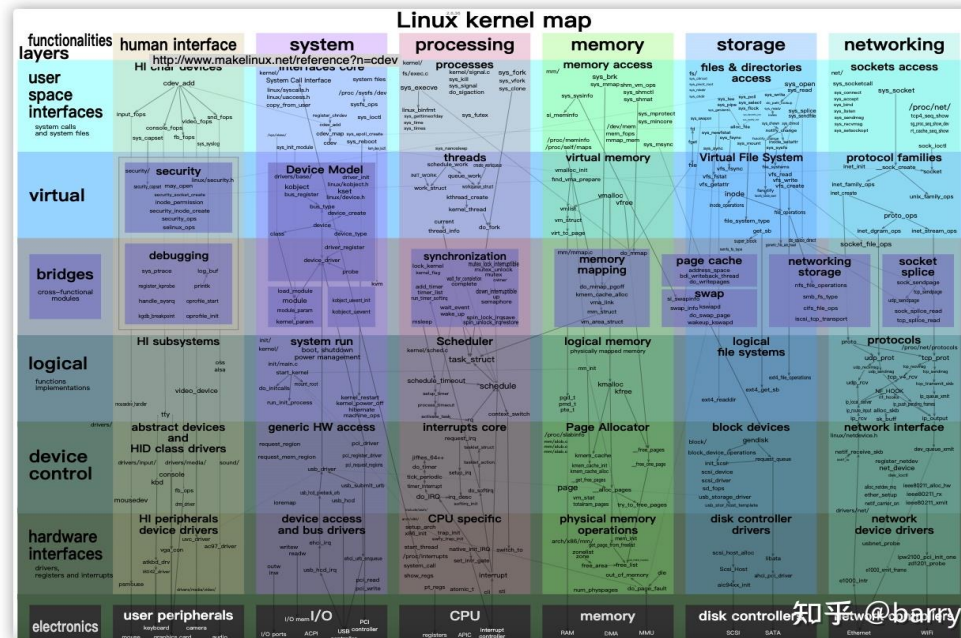
- 100% Free
- 2.5 Billion Devices
- Contributions to the open source community
- More than 90% iCloud Infrastructure



**Part 3**  
**Monolithic Kernel**  
**vs**  
**Microkernel**

# Monolithic Kernel(宏内核)

- Integrate the core functions of the operating system into a **single kernel module**
- Scheduling (调度), File system (文件系统), Networking (网络), Device driver (设备驱动程序), Memory management (存储管理), Paging(存储页面管理)
- All kernel services exist and execute in the kernel space. The kernel can invoke functions directly.
- eg.UNIX, Linux



# features

- Efficiency: Due to all functions being integrated into the kernel, the interaction between functional modules within the kernel is very fast. No need for complex inter-process communication (IPC).
- Complexity: Due to the large kernel containing many different functions and services, the system becomes relatively complex. The strong coupling between different modules increases **the difficulty of development and maintenance**.
- Stability and security issues: As all functions run in kernel mode, **any module failure may cause the entire system to crash or become unstable**. In addition, **high coupling between kernel layers** may lead to security vulnerabilities, as once a module is breached, attackers may be able to gain control of the entire kernel.

# Microkernel(微内核)

- Only **the most basic services** run in **kernel space**, with **other services running in user space**.

The microkernel itself typically includes only the most fundamental services, such as:

- **Inter-process Communication (IPC)**: Mechanisms for processes to communicate and synchronize with each other.
- **Basic Scheduling**: Managing the execution of processes.
- **Minimal Memory Management**: Essential functions for memory allocation and protection.

Servers invoke "services" from each other by sending messages via IPC. Device drivers, file systems, and network protocols, are implemented in **user space** as separate processes.

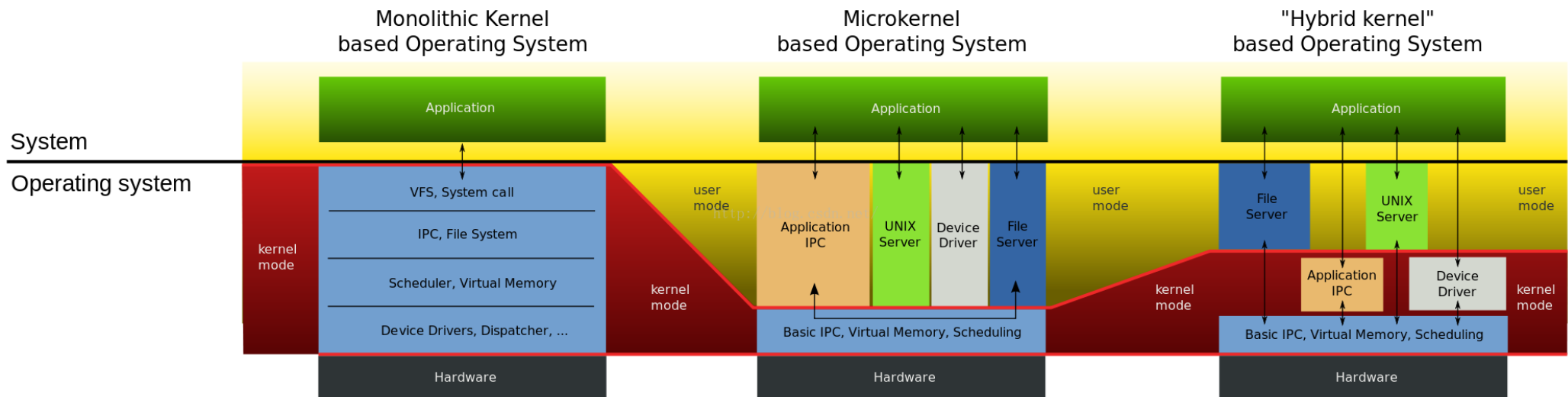
eg: Mac OS

# features

**Modularization and Flexibility:** The coupling between various parts of the operating system is low, making it **easier to maintain and expand**.

**Message passing mechanism:** The various service modules in the microkernel typically **communicate with each other through a message passing mechanism** to ensure independence and isolation between modules. This type of communication is usually achieved through specific interfaces such as IPC.

# Monolithic Kernel VS Microkernel



ref:<https://blog.csdn.net/lxl584685501/article/details/46801037>

# Monolithic Kernel VS Microkernel

	Monolithic Kernel	Microkernel
Kernel structure	contains all core functionalities, such as process management, file system, and device drivers, etc.	only contains basic functionalities; other functionalities run as user-space services.
Performance	Higher performance, because communication between modules takes place within the kernel space, resulting in higher efficiency	Relatively lower performance, because communication between modules requires a message passing mechanism.
Scalability	Poor extensibility, kernel functionalities are relatively fixed; modification and extension require modifying the kernel itself.	Highly extensible; new functionalities can be added as user-space services.
System stability and security	If a module within the kernel crashes, it may lead to the crash of the entire operating system.	more stable, because most functionalities run in user space, so a failure will not affect the entire kernel.
Development Complexity	relatively complex; kernel modules are tightly coupled.	relatively easy, but requiring handling communication between kernel and user-space processes.
Fault Recovery	A fault in the kernel may cause the system to crash, requiring a reboot of the entire operating system.	restart user-space service processes to recover, without needing to reboot the entire system.

# Performance

- Switch between user mode and kernel mode
- Shared page cache(eg. File System and Virtual Memory)

# Tanenbaum–Torvalds debate

- 工程派代表vs典型学院派老学究
- He noted that even though Linux was free, it wouldn't be a viable choice for his students, as they would not be able to afford the expensive hardware required to run it, and that MINIX could be used on "a regular 4.77 MHz PC with no hard disk."
- Microkernel(MINIX, Microkernel): future, scalability, modularize, security
- Monolithic Kernel(Linux, Microkernel): high performance
- Monolithic kernel (Linux) won the general computing market in the late 20th and early 21st centuries
- Microkernel (isolation, modularity) won the "philosophical victories" in the security field through microkernel concept in the distributed application field through microservice architecture

ref: [https://en.wikipedia.org/wiki/Tanenbaum–Torvalds\\_debate](https://en.wikipedia.org/wiki/Tanenbaum–Torvalds_debate)

## From L3 to seL4 What Have We Learnt in 20 Years of L4 Microkernels?

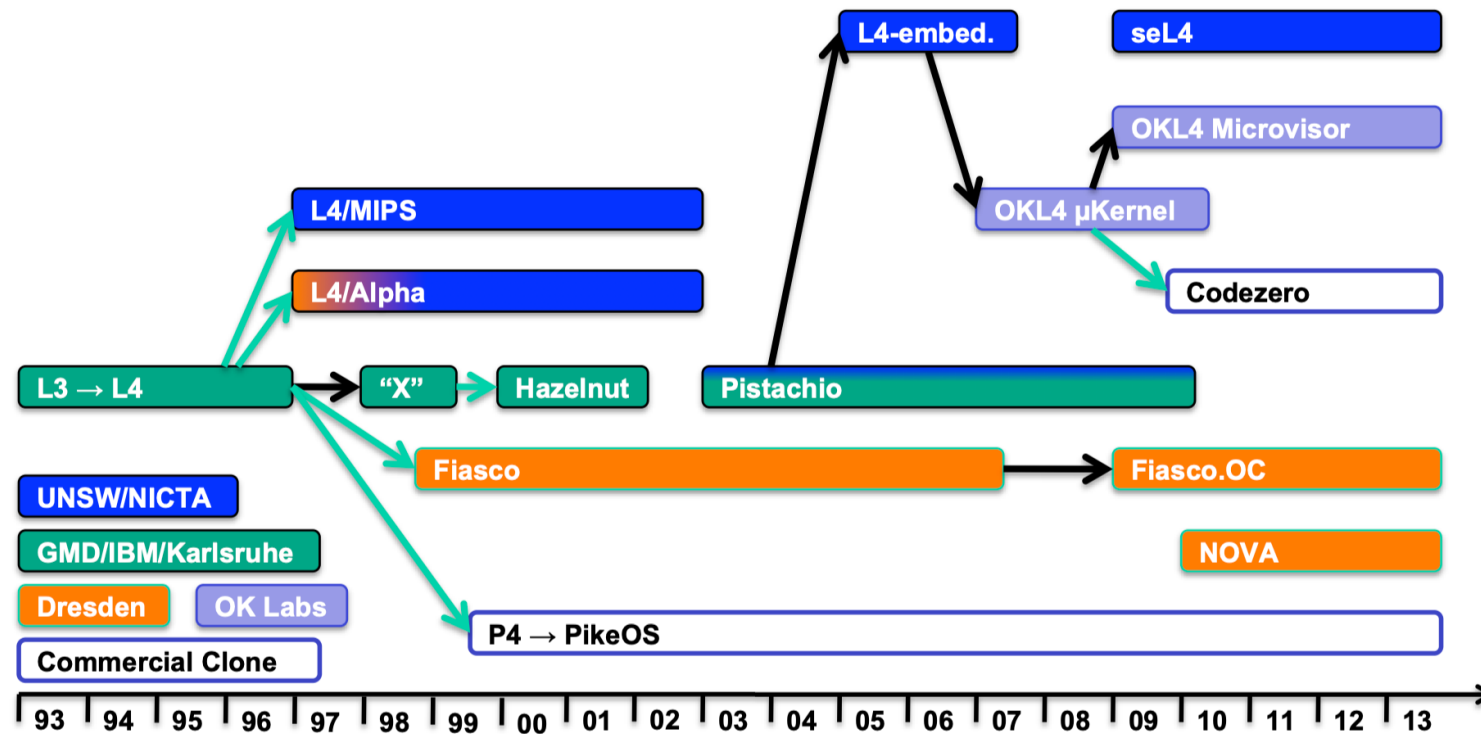


Figure 1: The L4 family tree (simplified). Black arrows indicate code, green arrows ABI inheritance. Box colours indicate origin as per key at the bottom left.

# **L4: Intended to provide high-performance microkernel architecture**

**Design and implementation decisions:**

- **Minimality**
- **IPC**
- **seL4 Formal Verification**

# Minimality

mer helps the latter. Specifically, he formulated the microkernel minimality principle:

A concept is tolerated inside the  $\mu$ -kernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality [Liedtke, 1995].

- (ref:Hermann Hartig, Michael Hohmuth, Jochen Liedtke, Sebastian Schonberg, and Jean Wolter. The performance of  $\mu$ -kernel-based systems. In Proceedings of the 16th ACM Symposium on Operating Systems Principles, pages 66–77, St. Malo, France, October 1997.)

# IPC

- 10–20x faster
- **Register Passing:** Small messages were passed directly via CPU registers, avoiding memory copying and cache pollution.
- **Direct Data Transfer:** For large messages, L4 used techniques like **map/unmap** or **direct transfer** to minimize or eliminate unnecessary data copying between address spaces.
- **Context Switch Optimization:** The kernel minimized the work done during a context switch, which is inherent in every IPC operation.

# Multicore Concurrency

- Minimality: less code, less synchronized and locked code
- Compare to Monolithic kernel: much easier

# seL4 Formal Verification

- **Mathematically prove** that the seL4 kernel's C code implementation **strictly conforms to its abstract specification**, demonstrating its correctness
- seL4 was the first OS kernel to be fully Formally Verified, proving no common implementation bugs (eg. buffer overflows, null pointer dereferences, and incorrect logic).
- Minimality