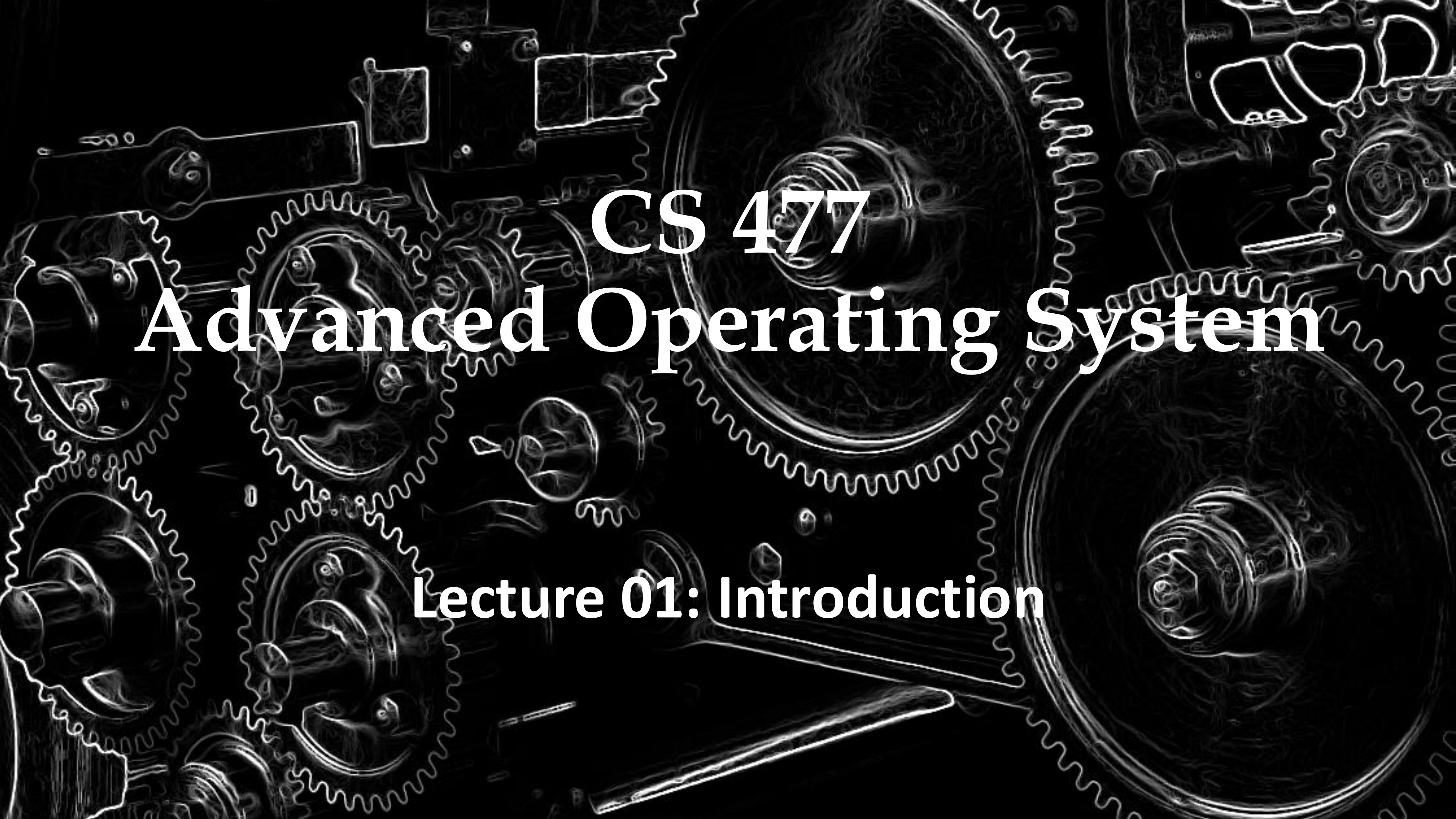

CS 477
Advanced Operating System
Lecture 01: Introduction

Today's lecture

1. **Course and instructor**
2. Foundations: What is an OS
3. Linux: From hobby to hegemony
4. OS design considerations
5. Boundary and cost
6. How will we learn the kernel
7. Course logistics

About your instructor

- Sanidhya Kashyap
- Assistant Professor
- Email: sanidhya.kashyap@epfl.ch
- Office: INN 240
- Homepage: https://sanidhya.github.io

- Research group: https://rs3lab.github.io
- Research interest: operating system, storage system, distributed system, system security

What makes this course “Advanced”?

- CS 477: Advanced OS
  - Focus on the Linux kernel and programming

- Goals:
  - **Understand** the **core subsystems** of the modern OS
  - **Design, implement, and modify** OS kernel code
  - **Test, debug, and evaluate the performance** of systems software in kernel

Three pillars of Advanced OS

Pillar I: Theory

- **Fundamental concepts**
  - Concurrency theory
  - Memory consistency
  - Scheduling algorithms
  - Security principles
  - IO models
  - Performance analysis
  - Formal verification

Pillar II: Implementation

- **Linux kernel mastery**
  - 40M+ lines of production code
  - Real-world constraints
  - Performance at scale
  - Backward compatibility
  - Hardware quirks

Pillar III: Research

- **Open problems**
  - OS for heterogeneous computing
  - Verified systems software
  - Kernel bypass vs. offloaded
  - Energy-efficient computing
  - Post-Moorer’s law OS design

Prerequisite and expectation

- Graduate students:
  - C programming (strict)
  - Linux command line (strict)
  - Computer architecture and basics of OS (recommended)

Course timeline

W1 W2 W3 W4 W5 W6 W7 W8 W9 W10 W11 W12 W13 W14 W15

Foundations	Core subsystems	Storage & IO	Advanced
Kernel basics | System calls | Data structures	Process Mgmt | Scheduling | Memory Mgmt | Intr.	VFS | File System | Block Layer	Virtualization

Lab 1: Syscalls	Lab 2: Kernel DS	Lab 3: CPU Sched. & Profiler	Lab 4: TinyFS

Proj. handout: Sep 23	Proj. proposal: Oct 17	Proj. mid report: Nov 10	Proj. report and presentation: Dec 16; Dec 19

Labs: 40% **Project: 45%** **Exam: 15%**

Today's lecture

1. Course and instructor
2. **Foundations: What is an OS**
3. Linux: From hobby to hegemony
4. OS design considerations
5. Boundary and cost
6. How will we learn the kernel
7. Course logistics

An operating system …

- A software layer that **interfaces** between diverse **hardware resources** and one or many **applications** running on the machine
- Implements **virtual machine** that is easier to program than raw hardware

Easier to use, simpler to code, more reliable, more secure …

App App App

OS interface

Operating system

Physical machine interface

HW

Processor

Memory

Storage

IO connection

OS's role #1: Abstraction

- Provide a standard library for resources
  - CPU: process and/or thread
  - Memory: address space
  - Disk: files
  - Network: socket

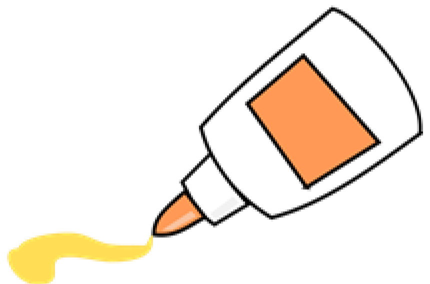
- Advantages?
  1. Allow applications to reuse common facilities
  2. Make different devices look the same
  3. Provide higher-level or more useful functionality

OS's role #2: Resource management

- Share resources well!

- Advantages of an OS providing resource management?
  - Protect applications from one another
  - Provide efficient access to resources (cost, time, energy)
  - Provide fair access to resources

Three hats of an OS

Referee	Illusionist	Glue
		
Manages protection, isolation, and sharing of resources	Provides clean, easy-to-use abstractions of physical resources	Provides a set of common services

Three perspectives from an OS designer point of view

- 👷 Engineer:
  - “How do we build it?” → kernel module, syscalls, debug, optimize performance
- 🔬 Scientist:
  - “Why does it work?” → analyze algos, measure performance, prove correctness, understand tradeoff
- 🤔 Philosopher:
  - “Should it work this way?” → question assumption, consider other alternatives, debate design choices, envison the future

This course: all three perspectives, not just mere implementation

Today’s lecture

1. Course and instructor
2. Foundations: What is an OS
3. **Linux: From hobby to hegemony**
4. OS design considerations
5. Boundary and cost
6. How will we learn the kernel
7. Course logistics

Linux is eating the world!

- 85% of smartphones and tablets run Linux
  - 70% for Android and 15% for others
  - Android powers 1.5B smartphones annually
  - IOS: 15%
- 96.3% of the top 1 million web servers run Linux
- 92% of top 500 fastest supercomputers run Linux
- Enterprise Linux reaches 14.4B by 2025
- SpaceX: From Earth to orbit with Linux and SpaceX

https://www.enterpriseappstoday.com/stats/linux-statistics.html

It is good for your career opportunities

- Linux salary prospects (2025):
  - US: ~$172K/year
  - UK: ~112K/year
  - CH: 95K—135K
  - Around 279 positions opened in Switzerland!

Beginning of Linux: “Just a hobby”

From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds) Newsgroups: comp.os.minix
Subject: What would you like to see most in minix?

Summary: small poll for my new operating system Message-ID:
<1991Aug25.205708.9541@klaava.Helsinki.FI> Date: 25 Aug 91 20:57:08 GMT Organization:
University of Helsinki

Hello everybody out there using minix -

I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).

I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and Id like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them

Linus (torvalds@kruuna.helsinki.fi)

PS. Yes - it's free of any minix code, and it has a multi-threaded fs. It is NOT protable (uses 386 task switching etc), and it probably never will support anything other than AT-harddisks, as that's all I have :-(.

From hobby to hegemony

- 1991: First application, author: Linus Torvalds
- 1992: GPL license, first Linux distribution
- 1993: v1.0 - Single CPU for i386, then ported to Alpha, Sparc, MIPS
- 1996: v2.0 - Symmetric multiprocessing (SMP) support
- 1999: v2.2 - Big Kernel Lock removed
- 2001: v2.4 - USB, RAID, Bluetooth, etc.
- 2003: v2.6 - Physical Address Expansion (PAE), new architectures, etc.
- 2011: v3.0 - Incremental release of v2.6

The open source model

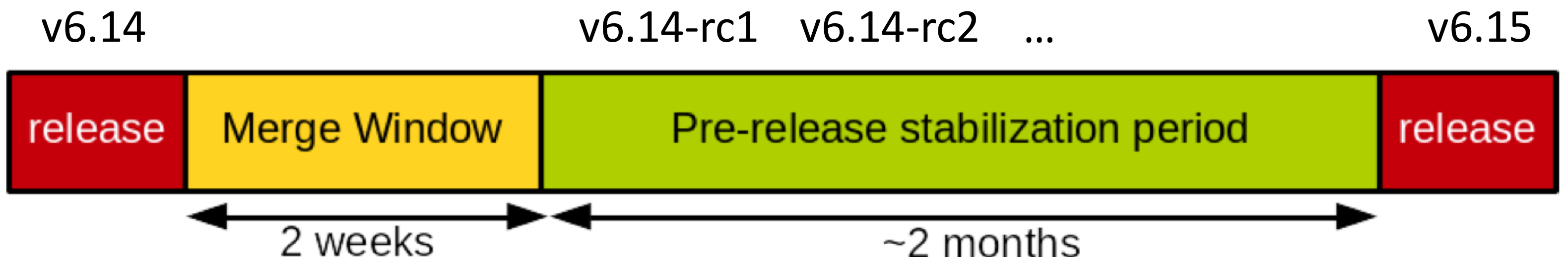
- Linux is licensed under **GPLv2**

```
“You may copy, distribute and modify the software as long as you
track changes/dates in source files. Any modifications to or
software including (via compiler) GPL-licensed code must also be
made available under the GPL along with build & install
instructions.”
```

- Source code is freely available: https://www.kernel.org/
- Ref: tl;drLegal

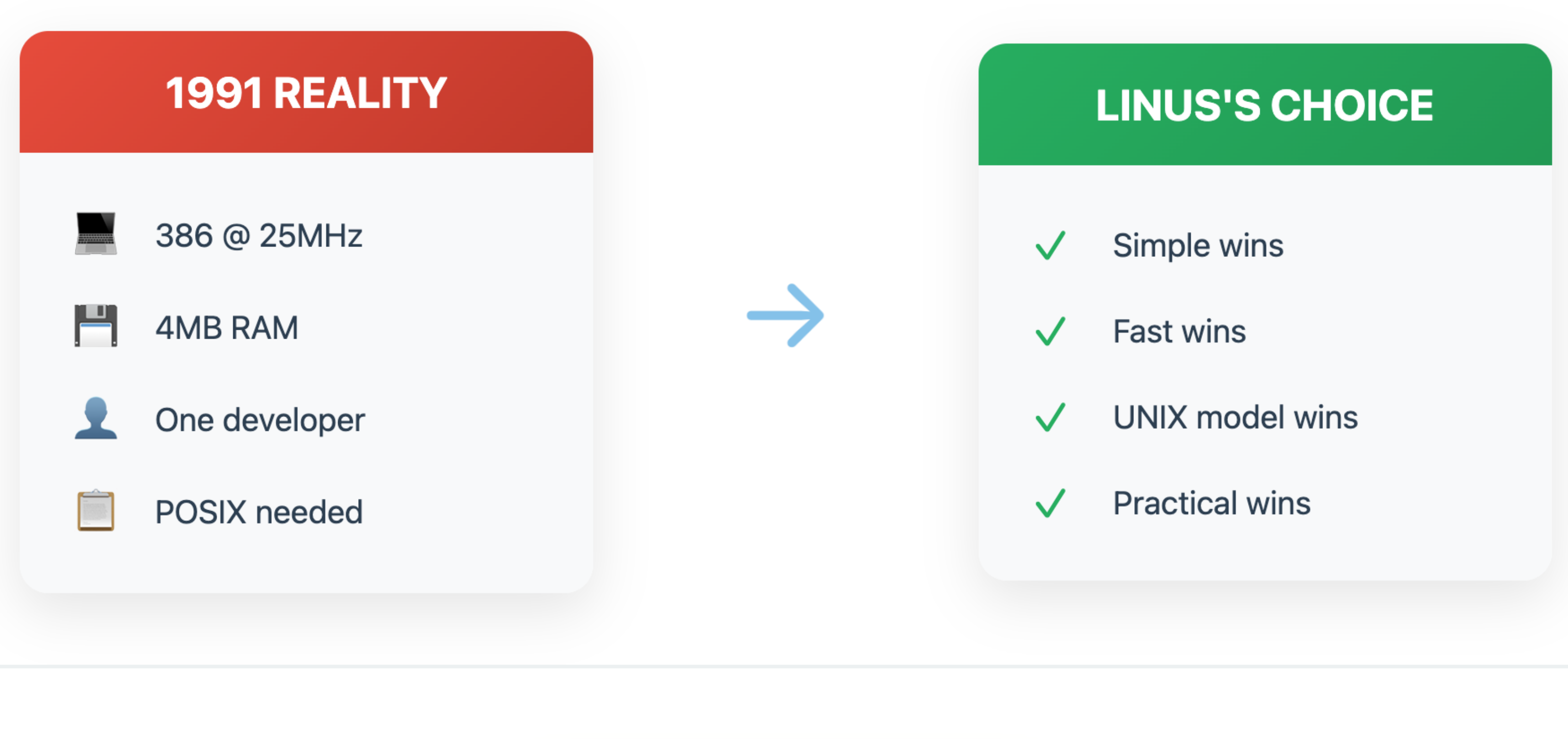
The open source model and its release cycle

- (major).(minor).(stable) → E.g., 6.15.0



- Prepatch or “RC” kernel release → for testing before the mainline release
- Mainline release → maintained by Linus with all new features
- Stable release → additional bug fixes after the mainline release
- Long-term support (**LTS**) for a subset of release → E.g., 6.12.32

Linux 1991 design choices



Why NOT Microkernel?

❌ Too complex for solo dev

❌ Too slow on 386

Result: Linux dominance 30 years later

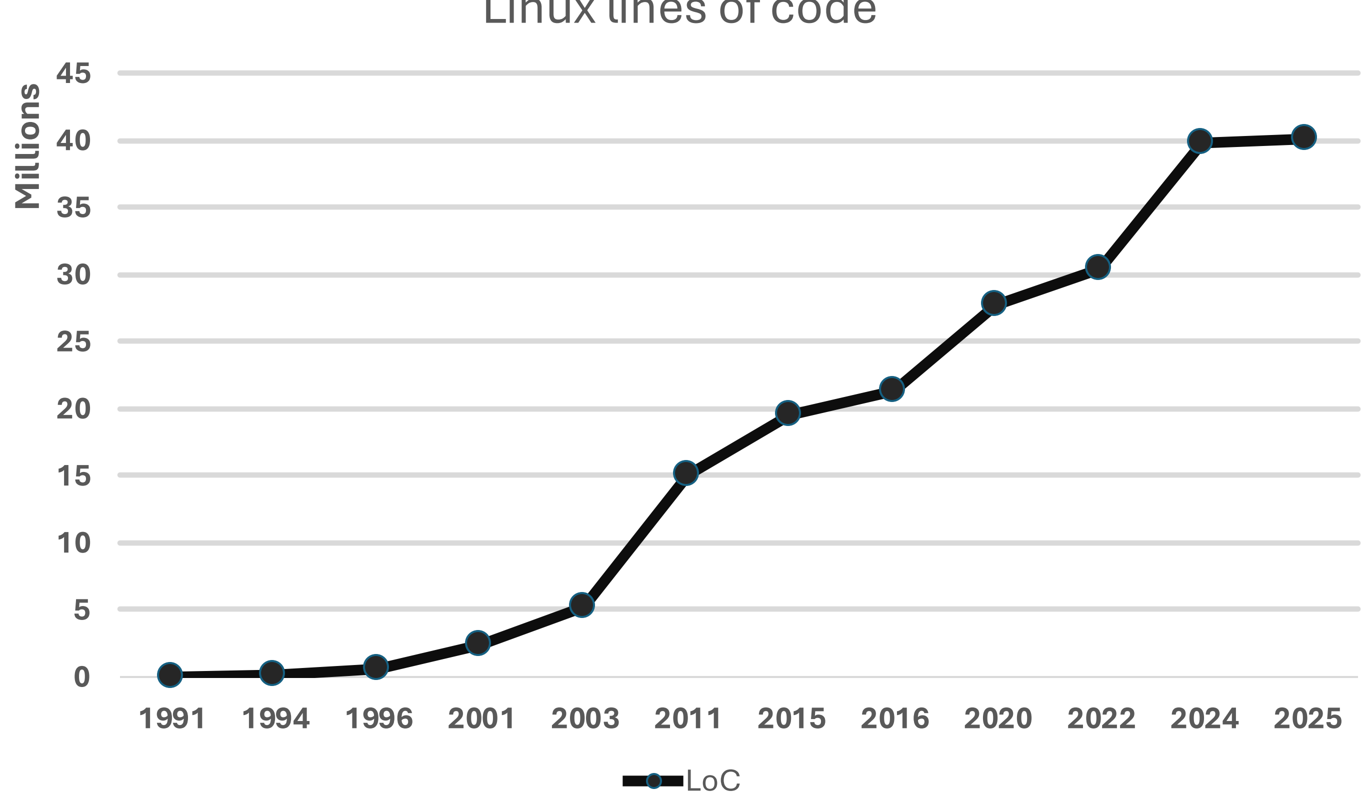
Why Monolithic?

✅ Simpler to build

✅ Better performance

Why play with Linux?

- Huge open-source software project
  - More than 40 million lines of code
  - 7,500+ lines of code added every day



Why play with Linux?

- Very fast development cycle
  - Release about every 70 days
  - 13,000 patches/release
  - 273 companies/release (or, 1,600 developers per release)

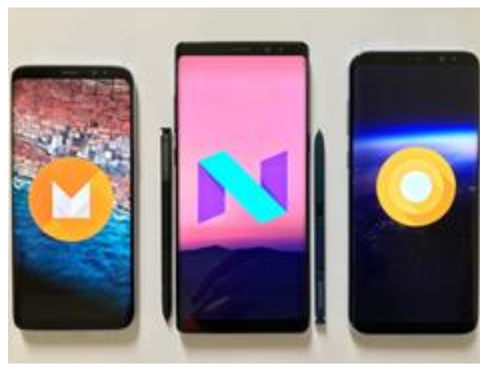
- One of the most well-written/designed/maintained C code
- More here …
  - Linux Foundation Kernel Report 2017
  - Linux Foundation Kernel Report 2023

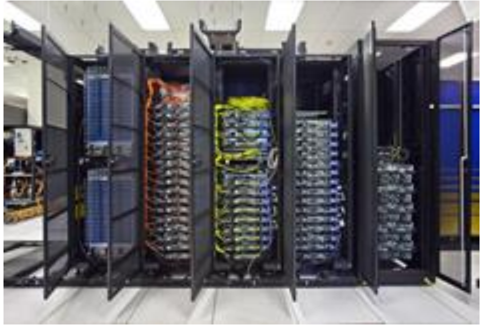
Today's lecture

1. Course and instructor
2. Foundations: What is an OS
3. Linux: From hobby to hegemony
4. **OS design considerations**
5. Boundary and cost
6. How will we learn the kernel
7. Course logistics

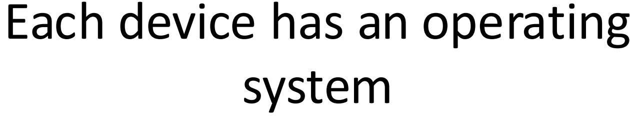
Why is designing an OS difficult?

1) So many different devices!





2) Communicate across devices around the world!



Each device has an operating system

Communicate over the Internet

Interface across huge sets of devices!

3) Bell's law

Mainframe

Mini

Workstation

PC

Laptop

Cell

Computers per person

$1:10^6$

$1:10^3$

1:1

$10^3:1$

years

Number crunching, data storage, ML

Productivity, interactivity

Streaming from/to physical world

A new device class every 10 years!

4) Computer performance trends

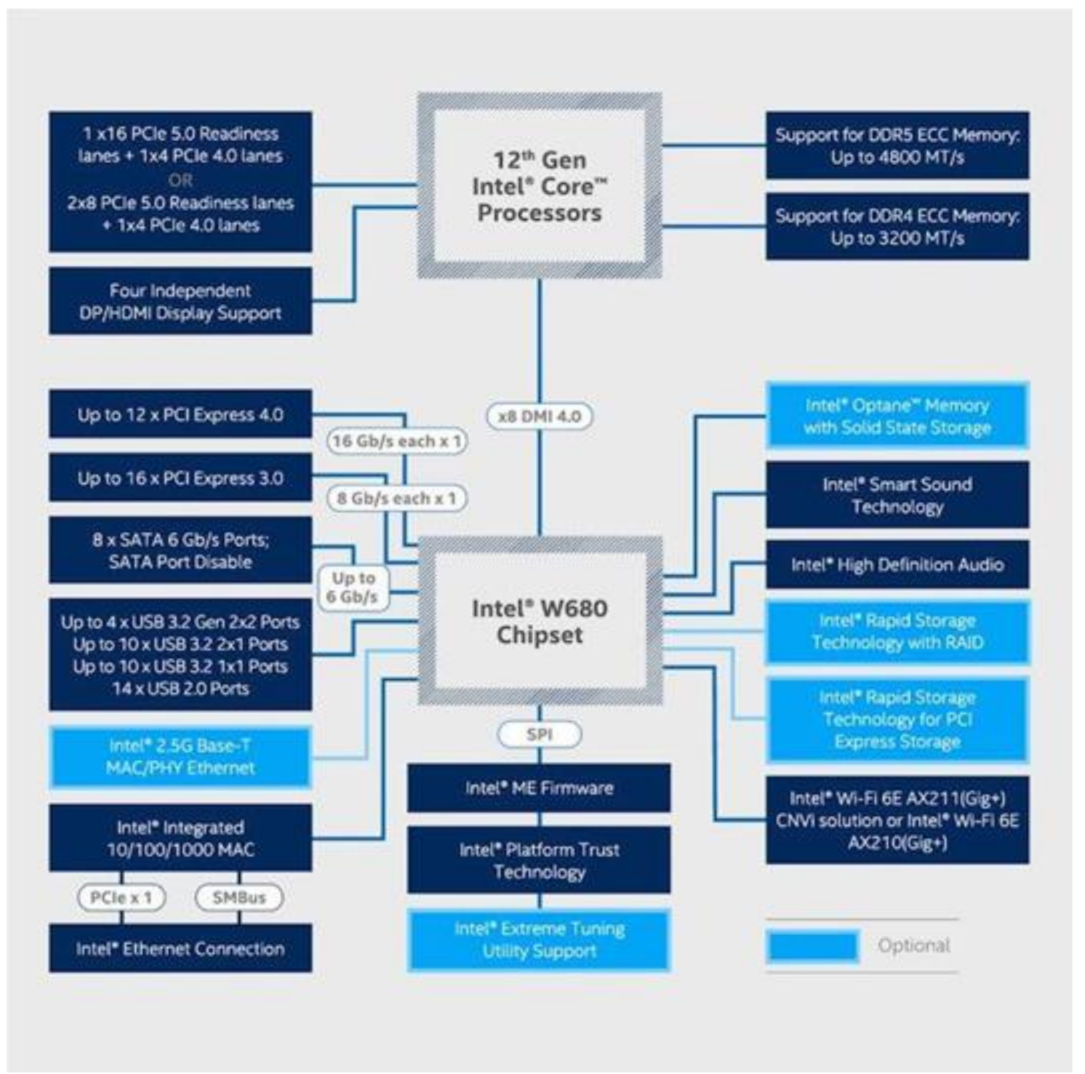
Latency Numbers Everyone Should Know

Operation	Time in ns	Time in ms (1ms = 1,000,000 ns)
L1 cache reference	1	
Branch misprediction	3	
L2 cache reference	4	
Mutex lock/unlock	17	
Main memory reference	100	
Compress 1 kB with Zippy	2,000	0.002
Read 1 MB sequentially from memory	10,000	0.010
Send 2 kB over 10 Gbps network	1,600	0.0016
SSD 4kB Random Read	20,000	0.020
Read 1 MB sequentially from SSD	1,000,000	1
Round trip within same datacenter	500,000	0.5
Read 1 MB sequentially from disk	5,000,000	5
Read 1 MB sequentially from 1Gbps network	10,000,000	10
Disk seek	10,000,000	10
TCP packet round trip between continents	150,000,000	150

New timescales keep coming up with devices

5) Hardware is getting complicated over time

- Hardware is becoming smarter
- Better reliability and security
- Better performance (more efficient code, more parallel computation)
- Better energy efficiency



OS design spectrum becomes complicated

- Handle all different devices
- Communicate across various devices
- Evolve to handle new devices coming on the market every decade
- Even handle new devices with varying characteristics
  - Hardware has its own complexity

→ OS needs to handle such evolving resources while maintaining the abstraction of the underlying hardware with efficiency

OS design spectrum: It's not binary

← MINIMAL KERNEL | MAXIMAL KERNEL →

Exokernel
(MIT JOS, Xok)

"Get out of the way"

KERNEL DOES:
- Hardware multiplexing
- Basic protection
- Nothing else

✓ Flexibility
✓ Custom opt.
✕ Complexity
✕ No sharing

Microkernel
(L4, seL4, QNX)

"Minimize privilege"

KERNEL DOES:
- IPC mechanism
- Memory management
- CPU scheduling

✓ Verification
✓ Isolation
✕ IPC overhead
✕ Complexity

Hybrid
(Windows NT, XNU)

"Pragmatic balance"

KERNEL DOES:
- Core + servers
- Modular design
- Some userspace

✓ Compatibility
✓ Flexible
✕ Complexity
✕ Unclear lines

Monolithic
(Linux, FreeBSD)

"Keep it together"

KERNEL DOES:
- Everything OS
- Shared memory
- Fast paths

✓ Performance
✓ Simplicity
✕ Fault spread
✕ Hard verify

Unikernel
(MirageOS, OSv)

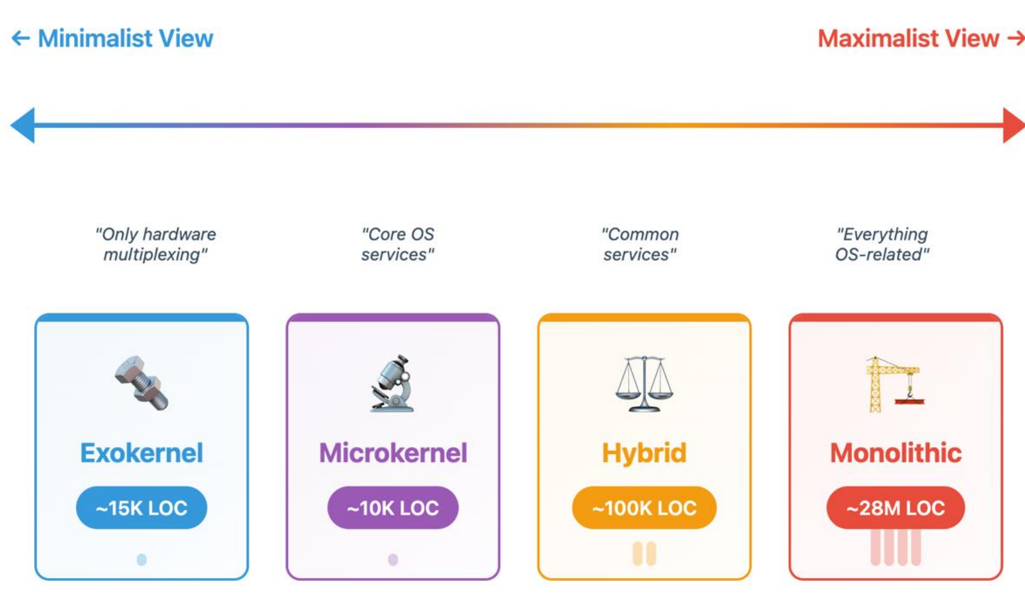
"No separation at all"

EVERYTHING IS:
- One address space
- Compile-time linked
- Single application

✓ Optimization
✓ Tiny size
✕ No protection
✕ Single app

The fundamental question: OS minimal responsibility

- **Mechanism vs. policy**
  - Mechanism: “How” to do something (kernel)
  - Policy: “What/when” to do it (possibly user space)
  - Example: Memory management
    - Mechanism: Page table update (must be the kernel)
    - Policy: page replacement algorithm (could be user space)



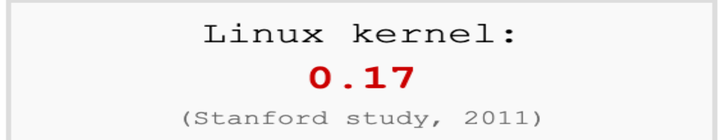
Where you draw the line defines your OS architecture!

Design philosophy: performance vs. principles

Monolithic (Linux)	Microkernel (seL4)
"Practicality beats purity"	"Correctness by design"
• Shared address space • Function calls • 28M LoC in kernel	• Isolated servers • Message passing • 10K LoC in kernel

Performance metrics	Linux	L4
getpid()	0.1µs	2µs
Context switch	2µs	0.5µs
IPC round-trip	N/A	0.5µs
Create process	100µs	50µs

BUG RATES (per KLOC):



seL4:
0.00
(Formally verified)

The famous debate: Tanenbaum vs. Torvalds

- Tanenbaum-Torvalds debate
- Most real-world kernels are mixed: Linux, OS X, Windows
  - Huawei's new OS is a microkernel
  - Linux, being deployed, is still monolithic

- Tanenbaum (1992): "Linux is obsolete."
- Torvalds: "Practice trumps theory"
- 30 years later: Who was right?
  - Plot twist: Both were (different goals)

Today's lecture

1. Course and instructor
2. Foundations: What is an OS
3. Linux: From hobby to hegemony
4. OS design considerations
5. **Boundary and cost**
6. How will we learn the kernel
7. Course logistics

User space vs. kernel space: The great divide

- A CPU executes either in **user space** or in **kernel space**
- Only the kernel is allowed to perform **privileged operations:** *Control CPU and IO devices*
  - E.g., protection rings in the x86 architecture
  - **Ring 3**: user space applications
  - **Ring 0**: OS kernel
- A user-space application talks to the kernel through the **system call interface**
  - `open(), read(), write(), close()`

Demo: Observing how programs use system calls (strace)

System call: crossing the boundary

User space

Kernel space

CPUs, Memory, I/O devices

Example: simplified path in the kernel for reading data into a file on the disk

myfile.readlines()

Python code

Cpython interpreter

C library

read()

...

...

...

System call: read()

System call processing

Virtual File System

File System

Block layer

HDD driver

sys_read()

vfs_read()

ext2_readpage()

submit_bio()

User-kernel interactions in monolithic vs. microkernels

Monolithic vs. Microkernel: Same op, different path

Monolithic

Application

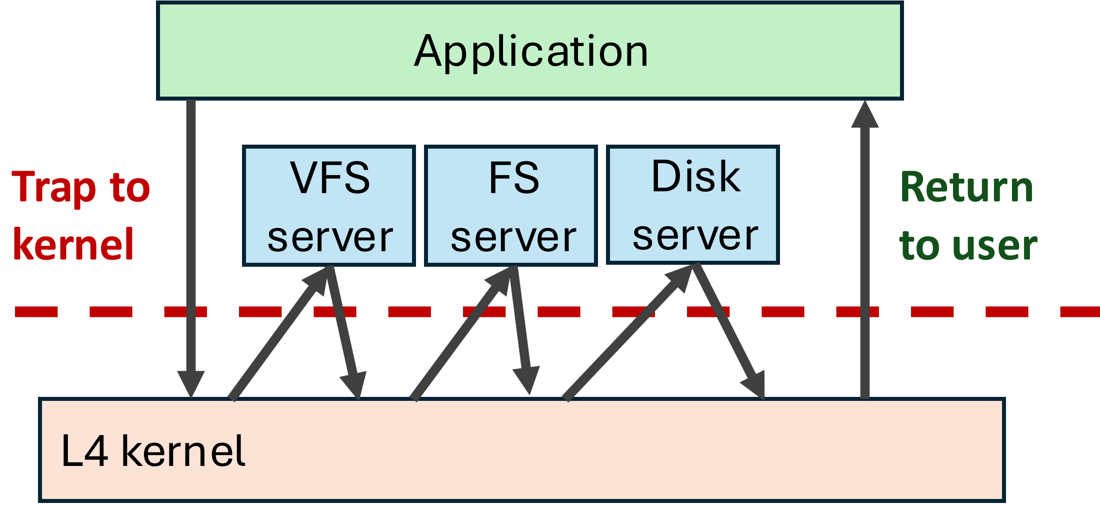
Trap to kernel

Return to user

Kernel space
→ System call handler
→ VFS layer
→ File system (ext4)
→ Block layer
→ Device driver

Time: ~10us
Context switch: 2

Microkernel



Time: ~40us
Context switch: 8

The cost of abstraction

Measurement metric	Monolithic	Microkernel	Difference
Latency	Lower	Higher	2-10x
Throughput	Higher	Lower	20-50%
Complexity	Higher	Lower	100x LoC
Bug impact	System-wide	Contained	Critical vs. local
Verification	Almost impossible	Possible	seL4 proved

Real numbers (2024)	Monolithic	Microkernel	Overhead
Simple syscall	100ns	500ns	5x
Page fault	1+us	5+us	2.5x
Fork	50us	30us	0.6x
IPC roundtrip	NA	500ns	-

Some ops can be faster in Linux, but composition overhead dominates

Current take on abstractions (old ideas, new forms)

- Containers: Microkernel-style isolation in monolithic
  - Namespaces, cgroups, capabilities
- eBPF: Safe in-kernel programming
  - Extending kernel safely with sandboxing and verified programs
- io_uring: Kernel bypass … inside the kernel
  - Shared memory rings for batching syscalls
  - Asynchronous model
- Rust in kernel: memory safety
  - Use language-level compile-time guarantees

Today's lecture

1. Course and instructor
2. Foundations: What is an OS
3. Linux: From hobby to hegemony
4. OS design considerations
5. Boundary and cost
6. **How will we learn the kernel**
7. Course logistics

Peeling the kernel onion: How we’ll learn

- Start simple (week 1) [Core: CPU + memory]
- Add interfaces (week 2) [+ system calls]
- Add management (week 5) [+ process control]
- Add optimization (week 9) [+ virtual memory]
- Add persistence (week 11) [+ file system]
- Add abstraction (week 13) [+ virtualization]

View: Minimal kernel

User space

Kernel space

Processing

Memory Management

Hardware

CPU

Main Memory

View: Adding system calls

User space

Kernel space

System Call Interface

Processing

Memory Management

Human Interface & Various Devices

Storage

Networking

Interrupt Management

Devices drivers

Hardware

CPU

Main Memory

Mouse, Kbd., etc.

HDD / SSD

Network Interface

View: Process management & scheduling

User space

Kernel space

System Call Interface

Processing

Processes & Threads

Scheduling

Time Mgt.

Interrupt Management

Memory Management

Human Interface & Various Devices

Storage

Networking

Devices drivers

Hardware

CPU

Main Memory

Mouse, Kbd., etc.

HDD / SSD

Network Interface

View: Memory management

User space

Kernel space

System Call Interface

Processing
- Processes & Threads
- Scheduling
- Time Mgt.

Interrupts Mgt.
- SoftIrq
- Tasklet
- Work queues
- Interrupt handling

Memory Management
- Physical / Virtual Memory Management
- Process Address Space
- Memory Allocation
- Page Cache

Human Interface & Various Devices

Storage

Networking

Devices drivers

Hardware
- CPU
- Main Memory
- Mouse, Kbd., etc.
- HDD / SSD
- Network Interface

View: IO

User space

Kernel space

System Call Interface

Processing
- Processes & Threads
- Scheduling
- Time Mgt.

Interrupts Mgt.
- SoftIrq
- Tasklet
- Work queues
- Interrupt handling

Memory Management
- Physical / Virtual Memory Management
- Process Address Space
- Memory Allocation
- Page Cache

Human Interface & Various Devices

Storage
- Virtual File System
- File System
- Block layer

Networking
- Sockets
- TCP/UDP
- IP
- Ethernet

Devices drivers
- Char.
- Block
- Network

Hardware
- CPU
- Main Memory
- Mouse, Kbd., etc.
- HDD / SSD
- Network Interface

View: Basic kernel building blocks

User space

Kernel space

Data structures

Synchro-nization

System Call Interface

Processing

Processes & Threads

Scheduling

Time Mgt.

Interrupts Mgt.

SoftIrq

Tasklet

Work queues

Interrupt handling

Memory Management

Physical / Virtual Memory Management

Process Address Space

Memory Allocation

Page Cache

Human Interface & Various Devices

Storage

Virtual File System

File System

Block layer

Networking

Sockets

TCP/UDP

IP

Ethernet

Char.

Block

Network

Devices drivers

Hardware

CPU

Main Memory

Mouse, Kbd., etc.

HDD / SSD

Network Interface

View: Putting it all together

User space
Kernel debugging
Development tools
Static code exploration
Performance eval.

Kernel space
Data structures
Synchro-nization
Debugging
Tracing
Perf. Evaluation

System Call Interface

Processing
Processes & Threads
Scheduling
Time Mgt.

Interrupts Mgt.
SoftIrq
Tasklet
Work queues
Interrupt handling

Memory Management
Physical / Virtual Memory Management
Process Address Space
Memory Allocation
Page Cache

Human Interface & Various Devices

Storage
Virtual File System
File System
Block layer

Networking
Sockets
TCP/UDP
IP
Ethernet

Devices drivers
Char.
Block
Network

Hardware
CPU
Main Memory
Mouse, Kbd., etc.
HDD / SSD
Network Interface

Your learning path

[Phase 1: Foundations] **(Weeks 1–4)**

Milestone: I can modify the kernel!

[Phase 2: Core Systems] **(Weeks 5–8)**

Milestone: I understand how Linux schedules!

[Phase 3: Memory/Storage] **(Weeks 9–12)**

Milestone: I can implement core subsystems!

[Phase 4: Advanced/Project] **(Weeks 13–15)**

Milestone: I can improve on Linux!

Labs: From theory to code

- Four labs
  - Lab 1 (10%): **System calls** (2 weeks)
  - Lab 2 (25%): **Kernel data structures** (2 weeks)
  - Lab 3 (30%): **CPU profiling** (~4 weeks)
  - Lab 4 (45%): **Simple concurrent file system** (~5 weeks)
- We will provide a shim template and guidelines; you need to add code to finish the lab
- We will use Moodle for submitting reports and code patches

Demo: Hello World kernel module

Today's lecture

1. Course and instructor
2. Foundations: What is an OS
3. Linux: From hobby to hegemony
4. OS design considerations
5. Boundary and cost
6. How will we learn the kernel
7. **Course logistics**

Grading policy

- Labs: 40%
- Project: 45%
- Exams: 15%
  - Finals at the end of the exam session
  - Written exam
  - Includes questions from both lectures and labs

Lab policy

- TA will be present to help you with labs
- Each student will work individually

- **Late days:**
  - **Seven** days of a grace period for all labs
  - Don’t have to inform us
  - We deduct 10% points for each late day after utilizing the 7-day grace period

Academic honesty

You are allowed to collaborate on everything in this course, **BUT** you must adhere to the academic honesty policies of the university:

- You **must not share or text about labs** with others
- You **must participate in collaboration**
- You **must not use someone else's code/text** in your solution
- You **cannot force anyone** to provide you with the answer

Beware

Cheating and academic integrity violations will be reported. Each lab will be checked for plagiarism. Please check the academic integrity policy of EPFL:

https://bit.ly/3BmfHJU

You will learn a great deal in this course, but you will have to work diligently from the start

Communication

- **Lecture**
  - Location: GRA330
  - No recorded lecture
  - Lecture slides will be available before the lecture
  - You are expected to attend every lecture
  - Q&A protocol
    - Ask questions verbally if you attend in person
- **Tutorial/exercise**
  - Tutorials on some essential topics relevant to labs
  - Students are expected to attend the session to ask questions about the lab and project

Communication

- **Moodle**
  - **Moodle link**
  - Syllabus, lecture, slides, schedule, notes, etc.
  - For lab and project submissions
- **ED**: Communicate with the course staff and other students

Next actions

- Finish setting up the course environment
  - Bring your laptop

Open research problems

1. Can we build a verified Linux subsystem?
2. OS for heterogeneous computing
3. Learned OS components
4. Evolving Linux components
5. Energy-proportional kernel