INF333
Operating Systems
Lecture I

Burak Arslan
ext-inf333@burakarslan.com

Galatasaray Üniversitesi

Lecture I
2025-02-12

Course website

burakarslan.com/inf333

Based On

cs111.stanford.edu

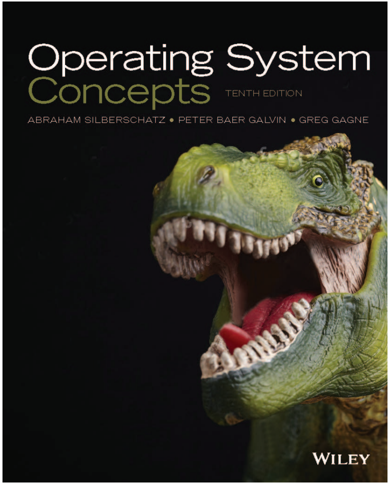
cs212.stanford.edu

OSC-10 Slides

Book

Operating System Concepts

Silberschatz, Galvin, Gagne



Course Structure

This is a project-driven course – you will have to do a lot of hands-on coding work!

- No attendance required
- **%60 of grade**: 3 or 4 homeworks + 1 homework = All of the *travaux pratiques*
- **%40 of grade**: Final exam
- No midterm

Two Courses in One

- This course: Operating Systems
- TP course: Linux Fundamentals

Which type of OS?

- Preemtpive vs Cooperative Multitasking
- Single-user vs multi-user
- Kernel vs unikernel
- Local vs distributed
- Single Process vs Multi process

Course Highlights

- Threads & Processes
- Concurrency & Synchronization
- Scheduling
- Virtual Memory
- I/O
  - File systems
  - Networking[1]

[1] We will have some very rudimentary coverage since we have dedicated networking courses

Course Goals

By the end of the semester, we hope to have taught you about:

- Caching, concurrency, memory management, I/O
- Dealing with complexity, big codebases

And improved on your skills about:

- Being better team players
- Email manners 😆
- $\LaTeX$

Course Goals

Fact:

Knowing about OS internals will make you a more effective software engineer

Homeworks

What kind of Code?

- We will write lots of C
- Mostly kernelspace but userspace as well

Homeworks I

You will work in groups of 2:

- Same team until the end of semester
- Course discussion between groups is encouraged
- However you are supposed to do your work in isolation
- Duplicate homeworks get **0** with no questions asked!
- Non-compiling projects get **0** with no questions asked!
- Don't miss the deadlines!

Homeworks II

You will hand in reports where you detail your solution:

- We will verify that you actually implemented your design
- Happy path coders will lose points – do proper error handling!
- Messy code will also cost you points – we need to understand your code!

Homeworks III

You will submit the git repo that you worked on as a team.

- We need to see **gradual progress** towards the homework goals in the commit log.
- We need to see **real collaboration** by both members of the team.
- We may **deduct points** from the lesser contributor, if we deem it necessary

Homeworks IV

We expect

- **We won't accept a submission with a single commit at the last minute**
- **We won't accept a fresh git repository for each submission** – You are expected to build on your previous work. This is simply a **semester-long project** split into 4 homeworks!

Homeworks V

Do not look at other people's solutions

- We have a big repository of PINTOS submissions
- Do not publish your own solutions
- That means using (public) github is asking trouble

You may read but not copy other OSes

- E.g., Linux, OpenBSD/FreeBSD, etc.

Cite any code that inspired your code:

- As long as you cite what you used, it's not cheating
- In worst case, we deduct points if it undermines the assignment

Operating Systems

A Gentle Introduction

Intro to OS

A fundamental goal of the OS is to elevate the hardware at hand to a **well-defined abstraction level**

Intro to OS

Some examples:

NIC	Ethernet, WiFi, Infiniband, VPN, ...
Connectivity	USB, Bluetooth, PCI, Thunderbolt, Serial, ...
Display	HDMI, USB-C, VGA, DVI, DisplayPort, ...
Input	Keyboard, Mouse, Gamepad, Touchpad, ...
FS	xfs, zfs, ext4, ntfs, apfs, ufs, ...

Intro to OS

In the dark ages of consumer operating systems, ...

Intro to OS

In the dark ages of consumer operating systems, ...

... we used DOS!

DOS apps like video games came with drivers for:

- Different graphics standards (EGA, VGA, Tandy, etc.)
- Different sound cards (SoundBlaster, AdLib, etc.)
- Memory manager (DOS/4GW)

Intro to OS

Modern operating systems have come a long way!

A modern OS provides:

- A proper layer between applications and hardware
- (Almost) Universal interfaces to the outside world
- Some level of protection against threats from (local / external) (malfunctioning / malicious) software.

Intro to OS

It's a mature field:

- We all use a small number of well-known operating systems
- Greenfield OS projects are a rare occurence, hardly have any impact

Intro to OS

However the so-called mature OS are dealing with huge problems;

- Security (hacking is still a thing!)
- Scalability (making use of growing hardware capacity)
- Efficiency (performance per watt)
- Difficulty to retrofit to new types of devices that would require new approaches

Back to basics

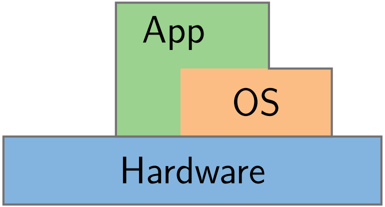
Let's go back to basics and retrace the steps of the modern operating system

Back to basics

Unikernel

A unikernel is a *statically linked* operating system:

- The system runs one program at a time
- No need for **memory protection**
- No precautions against malicious users or programs

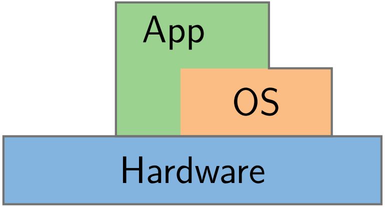


Back to basics

Unikernel

In the general case, this is furthest from optimal as possible:

- HUGE amount of code since it needs to have drivers, scheduler, memory manager, etc. etc.
- Can not run other tasks when idle or eg. CPU is waiting for IO
- Bad use of its users time: It's now up to the user to switch between tasks (computers?) manually

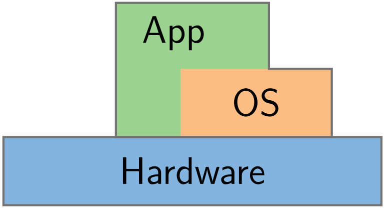


Back to basics

Unikernel

Not safe at all:

- It's up to the app to let other apps run (cooperative multitasking)
- The app can damage other users' data (single-user)
- The app has full access to the hardware (imagine dragons!)

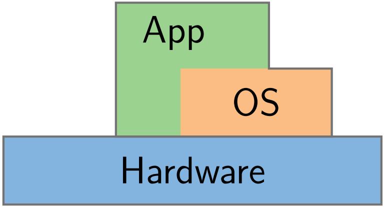


Back to basics

Unikernel

No program needs to be compatible with another:

- May use own file system
- May require specific hardware configuration
- That's why most of the code in early consumer operating systems consisted of the file system

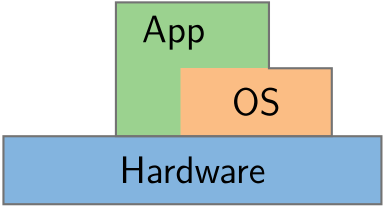


Back to basics

Unikernel

Also not an interesting topic of discussion:

- Today, any of you could write one given all the hardware manuals and enough time
- It's like writing games for very old game consoles!

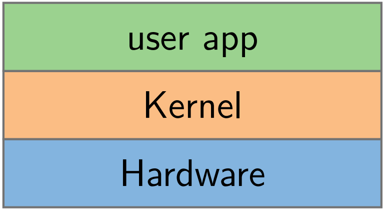


Back to basics

OS comes with Hardware

What if OS was "part of" the hardware instead of the application?

- User programs are now more loosely coupled with the OS
- Talks to the OS instead of the hardware
  - Of course, there may be exceptions...

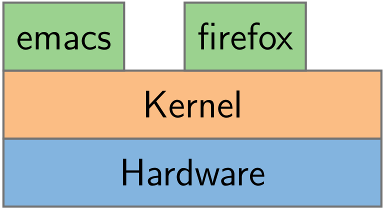


Back to basics

Multitasking

Maybe we could try to run other tasks when possible?

- Now we need a mechanism to protect tasks from each other
- What if a process wants to manipulate other users' data?
- What if a process wants to use all the storage capacity?

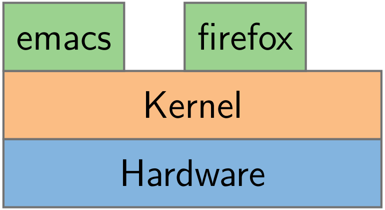


Back to basics

Multitasking

Solutions:

- **Preemptive scheduling**: Stop assuming well-mannered processes
- **Memory protection**: Stop processes from reading other processes' memory

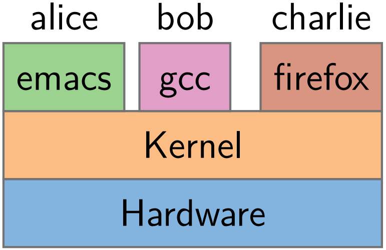


Back to basics

Multiple Users

Maybe we let more than one person use the computer as well?

- Now we need a mechanism to protect users from each other
- What if a process doesn't want to let others do some work?
- What if a process wants to manipulate other processes' memory?

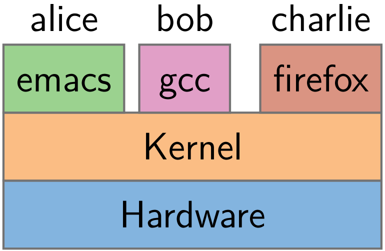


Back to basics

Multiple Users

Solutions:

- User authentication, permissions
- Virtual memory:
  - Allocate memory when actually used
  - Each process gets own address space
  - Swapping other processes' memory when required



A Modern OS

Made of two essential components:

Kernel For privileged operations

Userland For everything else

$\Rightarrow$ Userland is actually optional.

A Modern OS

The part of userland that interacts with its user(s) is called a **shell**.

Examples

- Ba**sh** is a command line shell.
- KDE is a graphical shell (among other subsystems)
- SSH is the **S**ecure **SH**ell
- etc.

A Modern OS

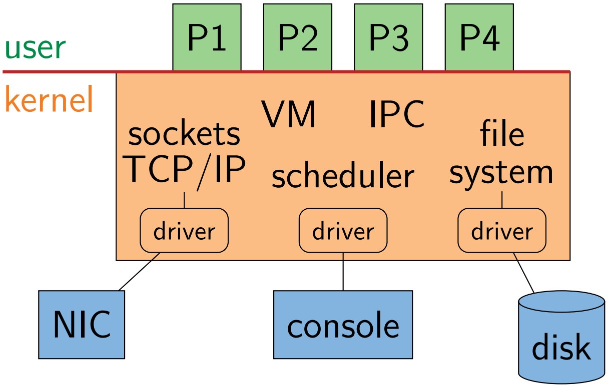
The user processes run in *user mode* where:

- The need for direct access to hardware is obviated by abstractions
- Every resource use attempt is verified

In this design, only the OS components are trusted to do the right thing.

A Modern OS

- Most code runs as user-level processes (P[1-4])
- The kernel runs in *privileged* mode
  - Manages processes
  - Mediates access to hardware



A Modern OS

This course will focus on the following operating systems that are:

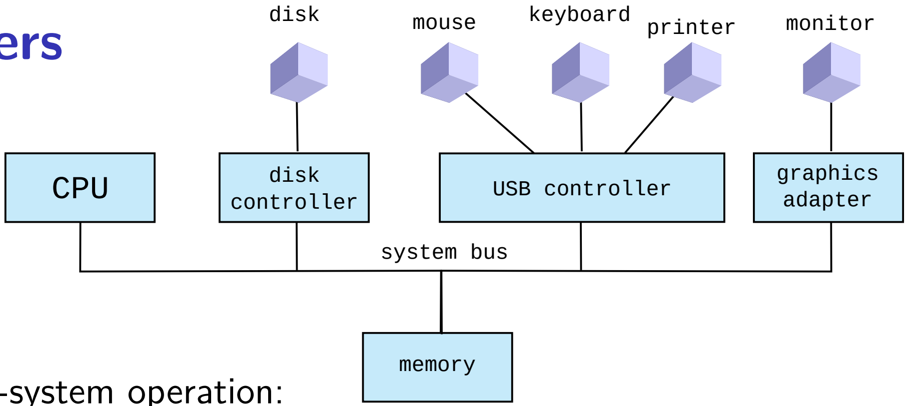
- Preemptive (not cooperative)
- Multi-user (not single user)
- Local (not distributed)
- Multi-address space / distinct kernel (not unikernel)
- Multi-process, with hardware support for memory protection

We will use Linux or similar OS to illustrate these and other concepts.

Computers

What exactly are they?

Computers



Computer-system operation:

- One or more CPUs, device controllers connect through common bus providing access to shared memory
- Concurrent execution of CPUs and devices competing for memory or cycles

Computers

- I/O devices and the CPU can execute concurrently
- Each device controller is in charge of a particular device type
- Each device controller has a local buffer
- Each device controller type has an operating system **device driver** to manage it
- CPU moves data from/to main memory to/from local buffers
- I/O is from the device to local buffer of controller
- Device controller informs CPU that it has finished its operation **by causing an interrupt**

Interrupts

Terminology

We say that:

- The device controller **raises** an interrupt by asserting a signal on the interrupt request line,
- The CPU **catches** the interrupt and **dispatches** it to the interrupt handler, and
- The handler **clears** the interrupt by servicing the device.

Interrupts

- An interrupt transfers control to the interrupt service routine, generally through the **interrupt vector**,
- IV contains the addresses of all the **interrupt service routines**: Callbacks that are invoked to handle the interrupt at hand
- A **trap** or an **exception** is a software-generated interrupt caused either by an error or a user request
- Operating systems are **interrupt driven**

Interrupts

When an interrupt is triggered:

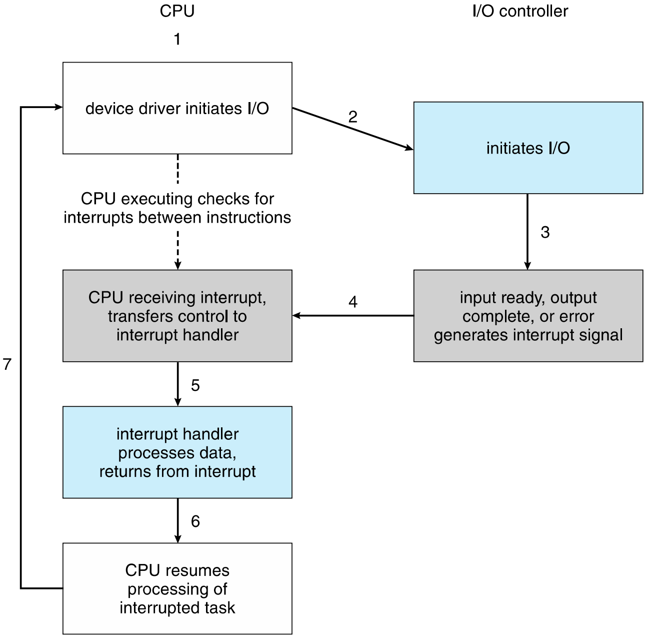
- The operating system preserves the state of the CPU by storing the registers and the program counter
- Handles the interrupt by invoking the ISR
- Restores CPU state and continues whet it left off.

Input/Output

Two methods for handling I/O:

Async After I/O starts, control returns to user program **without waiting for I/O completion**

Sync After I/O starts, control returns to user program **only upon I/O completion**



Input/Output

Sync Case

Sync (blocking) case has to wait;

- ... either by using a wait instruction
  (idles the CPU until the next interrupt)
- ... or by using a wait loop
  (causes contention for memory access).

In this mode, at most one I/O request is outstanding at a time, no simultaneous I/O processing

Input/Output

Async Case

Async case doesn't wait

- **System call** – request to the OS to allow user to wait for I/O completion anyway
- **Device-status** table contains entry for each I/O device indicating its type, address, and state
- OS indexes into I/O device table to determine device status and to modify table entry to include interrupt

Storage Hierarchy

Storage systems are organized in a hierarchy

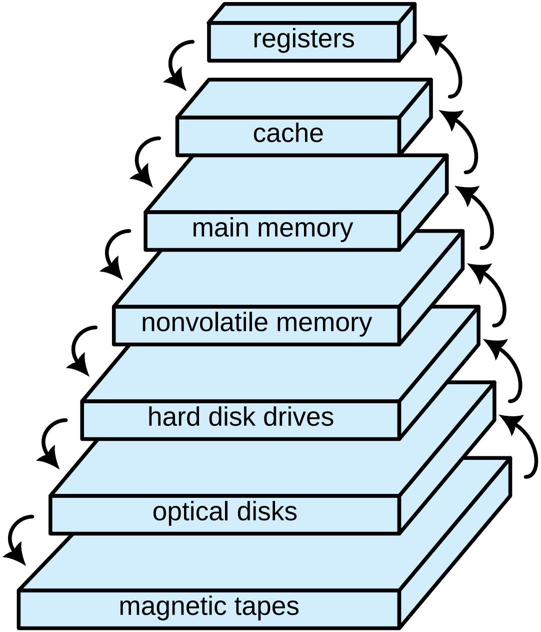
- Speed
- Latency
- Volatility

Storage Hierarchy

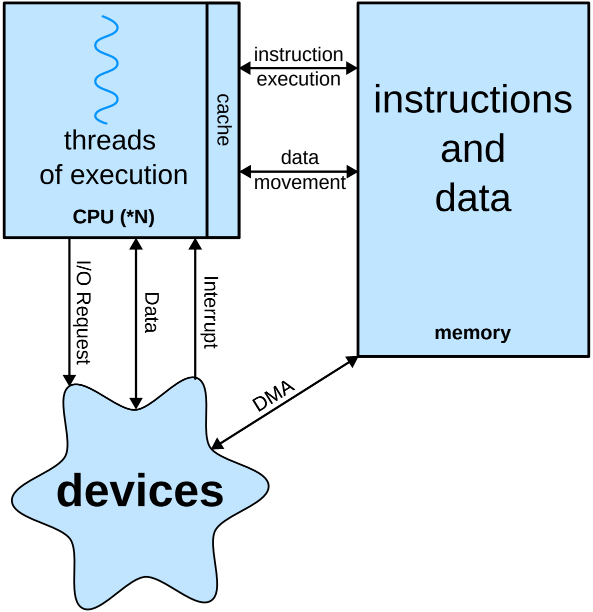
Caching – copying information into faster storage system

Example

- Main memory can be viewed as a cache for secondary storage



Computer Architecture



This is the infamous **von Neumann** computer