COSC 315: Introduction to Operating Systems

OS Basics and Architecture

1

Overview

- Why Architecture Matters to the Operating System
  - An operating system is not just software—it is software tightly coupled to hardware support
  - Hardware mechanisms define what the OS can safely and efficiently do
- This topic explores how architectural features enable OS abstractions
  - Virtualization: Making limited physical resources appear abundant
  - Protection & Isolation: Preventing faults and enforcing boundaries
  - Controlled Privilege Transitions: User → kernel via traps and syscalls
  - Concurrency Support: Timers, interrupts, and atomic instructions
  - Process Abstraction: Execution state, memory layout, and protection

The OS and hardware form a **co-designed system**: hardware enforces rules, the OS defines policy

2

What is an OS?

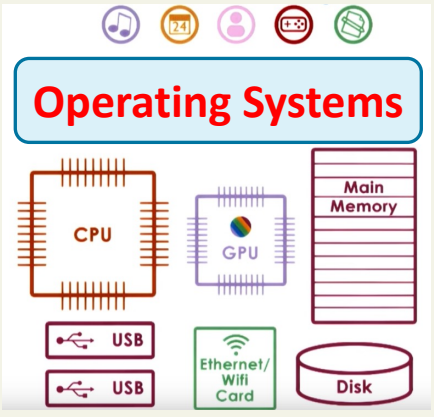
Consider all of the BIG issues:

- Software that
  - Abstracts
  - Arbitrates
  - Schedules
  - Manages

….the use of a computer system

- Directs operational resources
  - Controls use of CPU, memory, and peripheral devices ...
- Enforces working policies
  - Fairness to resources usage and access..
- Mitigates difficulty of complex tasks
  - Abstracts hardware details (system calls)

What is an OS?



An operating system is a layer of system software that:

- Directly has **privileged access** to the underlying hardware,
- **Hides** the hardware complexity (**Abstraction**)
- **Manages hardware** on behalf of one or more applications according to some predefined rules and policies
- In addition, it **ensures** that applications are **isolated and protected** from each other

It offers:

- **Virtualization**: presenting simplified views of complex hardware.
- **Concurrency**: managing multiple things happening at once.
- **Persistence**: managing data stored long-term.

Key Things:

1. Virtualization (not a VM)

- Virtualization means an OS makes physical resources look like:
  - More abundant than they really are (e.g., many programs think they have the CPU).
  - Cleaner abstraction boundaries between programs and hardware.
- Examples:
  - **CPU sharing** (time-slicing processes).
  - **Private address space** abstractions (memory).

5

Key Things:

2. Concurrency and Multiprogramming:

- OS must coordinate multiple programs that *appear* to run at once.
- Early systems introduced **multiprogramming** to improve utilization.
- Concurrency raises issues such as interleaved execution and correctness.

3. Persistence: (more later)

- Managing data that outlives a single CPU execution
  - File systems
  - Reliable storage

6

Modern Operating System Functionality

OS Service	Hardware Support
Protection	Kernel/user mode, protected instructions, base/limit registers
Interrupts	Interrupt vectors
System calls	Trap instructions and trap vectors
I/O	Interrupts and memory mapping
Scheduling, error recovery, accounting	Timer
Synchronization	Atomic instructions
Virtual memory	Translation look-aside buffers

- System calls cross a *privilege boundary* from user mode to kernel mode
  - Exposes OS services safely (programs can't directly manipulate hardware)

Protection

Kernel mode vs. User mode: To protect the system from aberrant users and processors, some instructions are restricted to use only by the OS.

Users may not:

- Address I/O directly
- Use instructions that manipulate the state of memory (page table pointers, TLB load, etc.)
- Set the mode bits that determine user or kernel mode
- Disable and enable system interrupts
- Halt the machine

In kernel mode, the OS can do all these things.

Protection: Rings

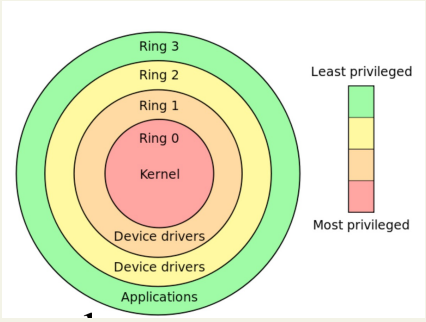
The hardware must support at least kernel and user mode:

- A status bit in a protected processor register indicates the mode.
- Protected instructions can only be executed in kernel mode

Modern mainstream OSes (Linux, Windows, macOS) usually use **only two rings**:

Ring 0 → kernel

Ring 3 → user programs



Inner rings -> trusted code (can do dangerous things)

Outer rings -> untrusted code (restricted, safer)

9

Protection: Rings

Ring model enforces three core goals:

Protection

- A buggy user program can't crash the whole system

Isolation

- Programs can't read or write each other's memory

Controlled access

- Hardware is accessed only through the kernel

How users Talking to the kernel

- User programs cannot jump directly into Ring 0 but must use a controlled gateway
- System calls (**syscalls**) which are special CPU instructions that:
  - Switch privilege level (Ring 3 → Ring 0)
  - Jump to a validated kernel entry point
  - Return safely back to Ring 3
- Key difference in calls :
  - Procedure call → same privilege level
  - System call → crosses rings

10

Traps

A **Trap** is a **synchronous**, CPU-detected exception detected by the architecture because of an issue with the current instruction, such as:

- Intentional: systems call
- Faults: page fault, write to a read-only page, overflow,

- On detecting a trap, the hardware
  - Saves the state of the process (PC, stack, etc.)
  - Transfers control to appropriate trap handler (OS routine) in a controlled and defined way:
    1. CPU indexes the memory-mapped trap vector with the trap number
    2. Jumps to the address given in the vector
    3. Starts to execute at that address.
  - On completion, the OS resumes execution of the process

A **trap** is how the hardware says: ***Stop! The OS needs to handle this.***

11

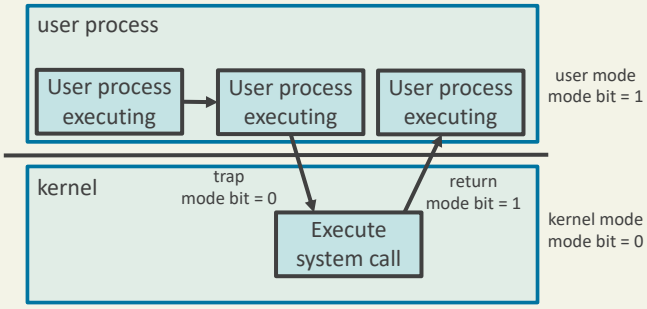
System Calls (syscall): Crossing Protection Boundaries

- OS procedure that executes privileged instructions
- (e.g., I/O) ; also API exported by the kernel
  - Causes a trap, which vectors (jumps) to the trap handler in the OS kernel
  - The trap handler uses the parameter from the system call to jump to the appropriate handler (I/O, Terminal, etc.)
  - The handler saves caller's state (PC, mode bit) so it can restore control to the user process
  - The architecture must permit the OS to verify the caller's parameters
  - The architecture must also provide a way to return to user mode when finished.

A system call is not just a function call; it is a change in privilege

12

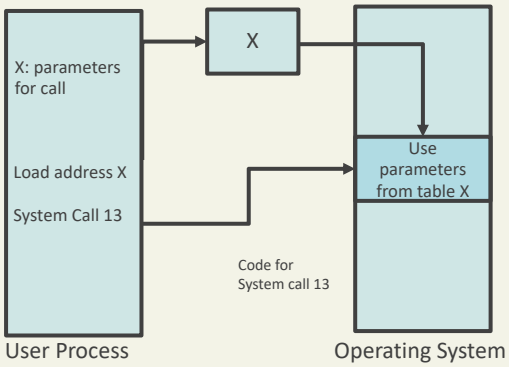
Using System Calls from User Space



Synchronous Mode: Wait until the system call completes

To make a system call an application must:

- write arguments
- save relevant data at well-defined locations
- make the system call



13

Important System Calls

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

14

Linux vs Windows System Calls

- Code at System Call level is not portable across operating systems. However, high level codes are portable

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

Interrupt Based Asynchronous I/O

Traps are synchronous; interrupts are asynchronous.

- Device controller has its own small processor which executes asynchronously with the main CPU
- Device puts an interrupt signal on the bus when it is finished
- CPU takes an interrupt.
  1. Save critical CPU state (hardware state),
  2. Disable interrupts,
  3. Save state that interrupt handler will modify (software state)
  4. Invoke interrupt handler using the *in-memory Interrupt Vector*
  5. Restore software state
  6. Enable interrupts
  7. Restore hardware state, and continue execution of interrupted process

Traps vs Interrupts

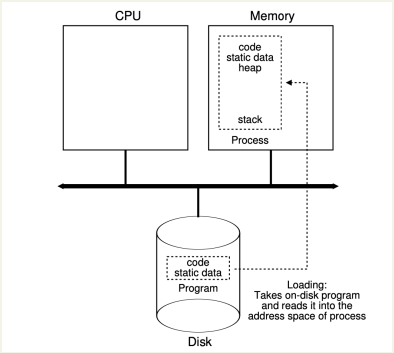
Feature	Trap	Interrupt
Caused by	**Current instruction** (Exception from a user process (overflow, error, sys call, etc)	**External event** (Generated by the hardware (HDD, GPU, I/O, etc)
Timing	**Synchronous** (user code is suspended and continues after)	**Asynchronous** (can happen unpredictably from a user's perspective)
Example	System call, page fault	Timer tick, keyboard
Used for syscalls?	Yes	No

Traps and interrupts **both** run with **elevated privileges**

17

Process

- What is a Process?
  - A process is the abstraction of a running program (not just code on disk)
  - Processes are the unit of **execution** and **resource ownership**.
- They allow:
  - **Multiprogramming**: switching between tasks on the CPU.
  - **Isolation**: one process can't normally read another's memory.
- It consists of:
  - Address space (code, data, heap, stack)
  - Execution state (registers, program counter)
  - OS metadata (open file descriptors, scheduling info)
- Tracked with pid (process id)

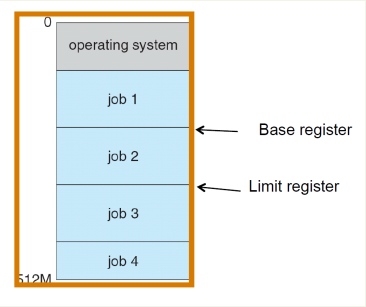


More on this later….

18

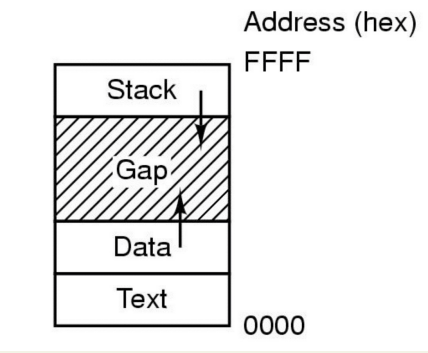
Memory Protection

- Architecture must provide support so that the OS can
  - protect user programs from each other
  - protect the OS from user programs
- The simplest technique is to use base and limit registers
  - **Base register** → *where the process's memory starts*
  - **Limit register** → *how big that memory region is*
- Base and limit registers are loaded by the OS before starting a program
- The CPU checks each user reference (instruction and data addresses), ensuring it falls between the base and limit register values
  - If **inside range** → access allowed
  - If **outside range** → **trap to the OS**



19

Process Layout in Memory



What about the heap?
It is part of the process's writable address space (allocation policy) (sits just above data)

Processes have three segments:

text
- program's **executable instructions**

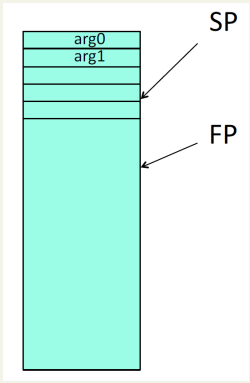
data
- Global variables
- Static variables
- Program state that persists for the lifetime of the process

stack
- Function call frames (activation records)
- Local variables
- Return addresses
- Saved registers

20

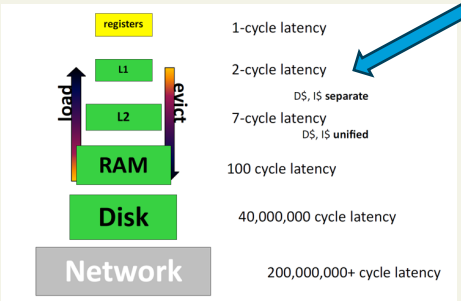
Registers

- Register = dedicated name for one word of memory managed by CPU
  - General-purpose:
    - "AX", "BX", "CX" on x86,
    - "$sX", "$tX", on MIPS
  - Special-purpose:
    - "SP" = stack pointer
    - "FP" = frame pointer
    - "PC" = program counter
- Change processes:
  - save current registers & load saved registers = context switch



Memory Hierarchy

- Higher = small, fast, more $, lower latency
- Lower = large, slow, less $, higher latency



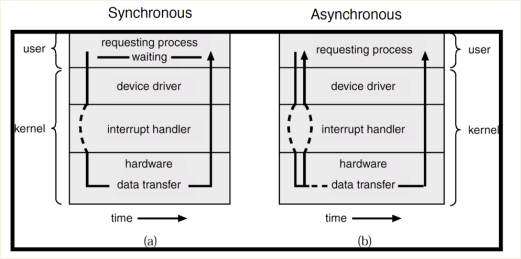
Caches

- Access to main memory: "expensive"
  ~ 100 cycles (slow, but relatively cheap ($))
- Caches: small, fast, expensive memory
  Hold recently-accessed data (D$) or instructions (I$)
  Different sizes & locations
    Level 1 (L1) – on-chip, smallish
    Level 2 (L2) – on or next to chip, larger
    Level 3 (L3) – pretty large, on bus
  Manages lines of memory (32-128 bytes)

Caches are managed by hardware (no explicit OS management)

I/O Control

- Each I/O device has a little processor inside it that enables it to run autonomously.
- CPU issues commands to I/O devices, and continues
- When the I/O device completes the command, it issues an interrupt
- CPU stops whatever it was doing and the OS processes the I/O device's interrupt
- Three models of I/O control: Synchronous, asynchronous, Memory-mapped



Memory-mapped

- Enables direct access to I/O controller (vs. being required to move the I/O code and data into memory)
- PCs (no virtual memory), reserve a part of the memory and put the device manager in that memory
- Access to the device then becomes almost as fast and convenient as writing the data directly into memory

Timer & Atomic Instructions

- A **hardware timer** generates periodic interrupts (e.g., every 100 µs)
- Timer interrupts allow the OS to:
  - regain control from running processes
  - enforce CPU time limits (prevent CPU hogging)
  - support pre-emptive multitasking
- On a timer interrupt, the OS may:
  - continue running the current process
  - switch to another ready process
  - perform accounting or timekeeping

Timer interrupts ***enable*** scheduling decisions but do not force a context switch

The timer interrupt gives the OS back control of the CPU; what the OS does next is a policy decision.

Atomic Instructions

- An atomic instruction is a CPU instruction that executes as an indivisible unit:
  - It either completes entirely, or it does not happen at all
  - No interleaving is possible.
- No timer interrupt, no context switch, no other core can observe it **half-done**
- Atomic instructions are the **foundation of correctness** in a concurrent system
  - Protect shared data
  - Protect critical sections of code
  - Build locks and synchronization primitives
  - Safely coordinate between processes, threads, etc
- Without atomic instructions:
  - Timers would cause corruption
  - syscalls could race
  - kernels could not be preemptive

Synchronization

- Interrupts interfere with executing processes
- OS must be able to synchronize cooperating, concurrent processes
- Architecture must provide a guarantee that short sequences of instructions (e.g., read-modify write) execute atomically
- Two solutions:
  - Architecture mechanism to disable interrupts before sequence, execute sequence, enable interrupts again.
  - A special instruction that executes atomically (e.g., test&set)

Virtual Memory

- Virtual memory allows users to run programs without loading the entire program in memory at once
- Instead, pieces of the program are loaded as they are needed
  - Each process has its own private **virtual address space**
  - **Addresses between applications may look the same but THEY ARE NOT!**
- The OS must keep track of which pieces are in which parts of physical memory and which pieces are on disk
- In order for pieces of the program to be located and loaded without causing a major disruption to the program, the hardware provides a translation lookaside buffer to speed the lookup

Key Takeaways

- OS provides an interface to the architecture, but also requires some additional functionality from the architecture.

  The OS and hardware combine to provide many useful and important features

- The OS provides **powerful illusions**:
  - Many CPUs (time sharing)
  - Private memory (address spaces)
  - Exclusive device access
  - Hardware ensures these illusions are **safe and enforceable**

Summary

- Key Architectural Supports
  - Privilege levels → kernel vs. user mode
  - Traps & interrupts → controlled entry into the kernel
  - Timers → preemption and fairness
  - Atomic instructions → correctness under concurrency
  - Memory protection → isolation and fault containment

Critical Takeaway

Without architectural support, modern operating systems would be impossible