
THE UNIVERSITY *of* EDINBURGH
informatics

# Operating Systems (INFR10079)

2020/2021 Semester 2

## Structure

(Syscalls)

abarbala@inf.ed.ac.uk

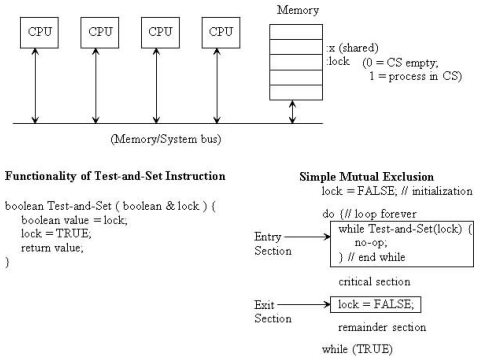
Chapter 2.1, 2.2, 2.3

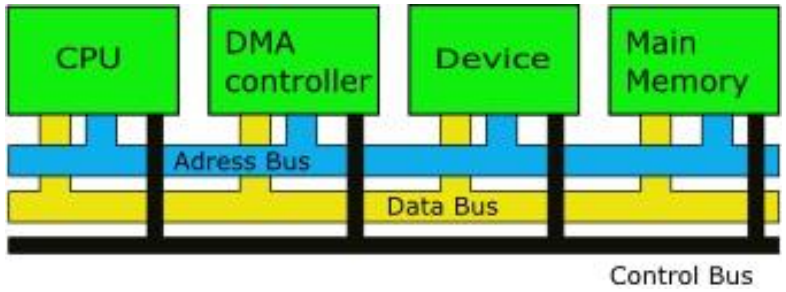
# Overview

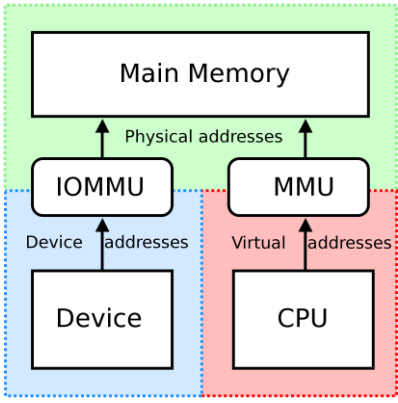
- **Architecture impact**
- **Application-Operating System interaction**
- *Operating System structure*

# Hardware Architecture
# Affects (is Affected by) the OS

- Operating system supports **sharing and protection** of HW
  - multiple applications can run concurrently, sharing HW resources
  - a buggy or malicious application should not disrupt other applications or the system
- The HW architecture determines **what is viable** (reasonably efficient, or even possible)
  - includes instruction set (synchronization, I/O, …)
  - also hardware components like MMU, DMA controllers, etc.

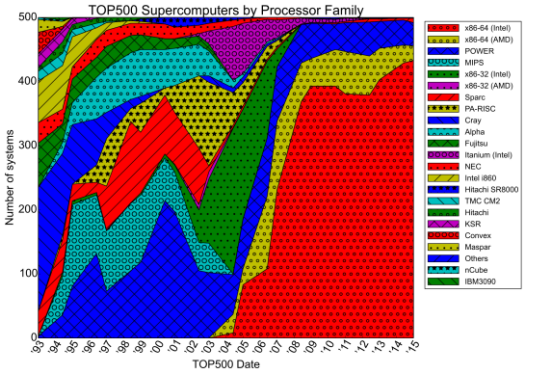


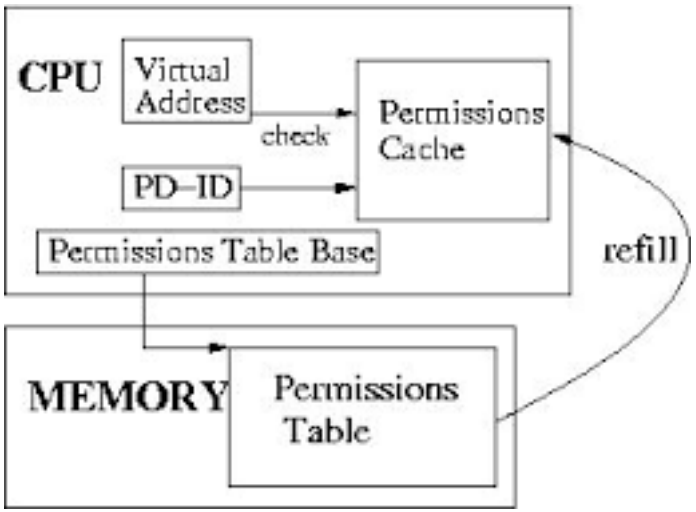
# Hardware Architecture Support for the OS

- Architectural support can simplify OS tasks
  - Example #1
    - Early PC OSes (DOS, MacOS) lacked support for virtual memory
    - At that time PCs lacked necessary hardware support (MMU)
  - Example #2
    - Until recently, Intel-based PCs didn't support for 64bit addresses
    - 64bit addressing has been available for decades on other HW architectures (MIPS, Alpha, IBM, etc.)
    - Changed driven by AMD's 64-bit architecture

# Hardware Architectural Features Affecting OS

- At the very beginning **hardware/software co-design**
  - Not anymore
- Features built primarily to **support OS**
  - timer (clock) operation
  - memory protection
  - I/O control operations
  - interrupts
  - **protected mode(s) of execution**
    - **kernel vs user mode**
    - **privileged instructions**
    - **system calls**
  - virtualization

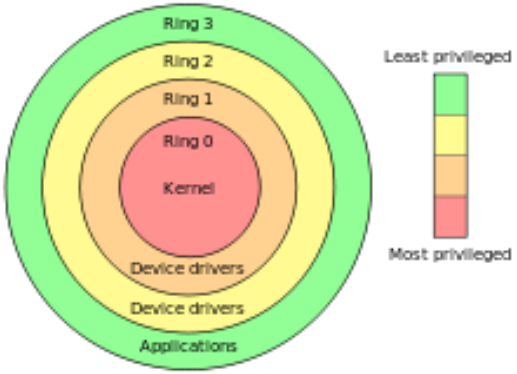


# Privileged Instructions

- Some instructions are restricted to the OS
  - Known as **privileged** instructions
- Only the OS can
  - Directly access some classes of I/O devices
  - Manipulate memory state management
    - Page table pointers, TLB loads, etc.
  - Manipulate special ‘mode bits’
    - Interrupt priority level
- **Restrictions provide safety and security**

# OS Protection

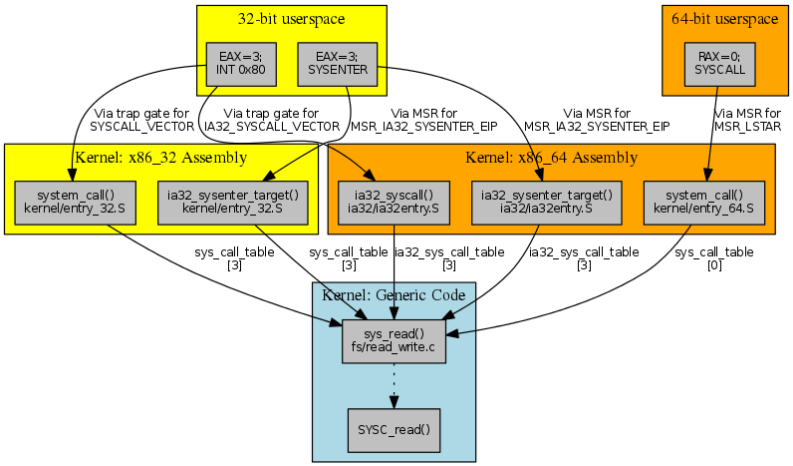
- How does the processor know if a privileged instruction can be executed?
  - Architecture must support **at least two modes** of operation
    - **kernel** mode
    - **user** mode
  - x86 supports 4 protection modes (rings)




  - **Mode** is set by status bit in a protected processor register
    - User programs execute in user mode
    - OS kernel executes in kernel (privileged, supervisor) mode
- Privileged instructions can only be executed in kernel mode
  - When code running in **user mode** attempts to execute a privileged instruction the “Privileged Instruction” exception is triggered

# Crossing Protection Boundaries

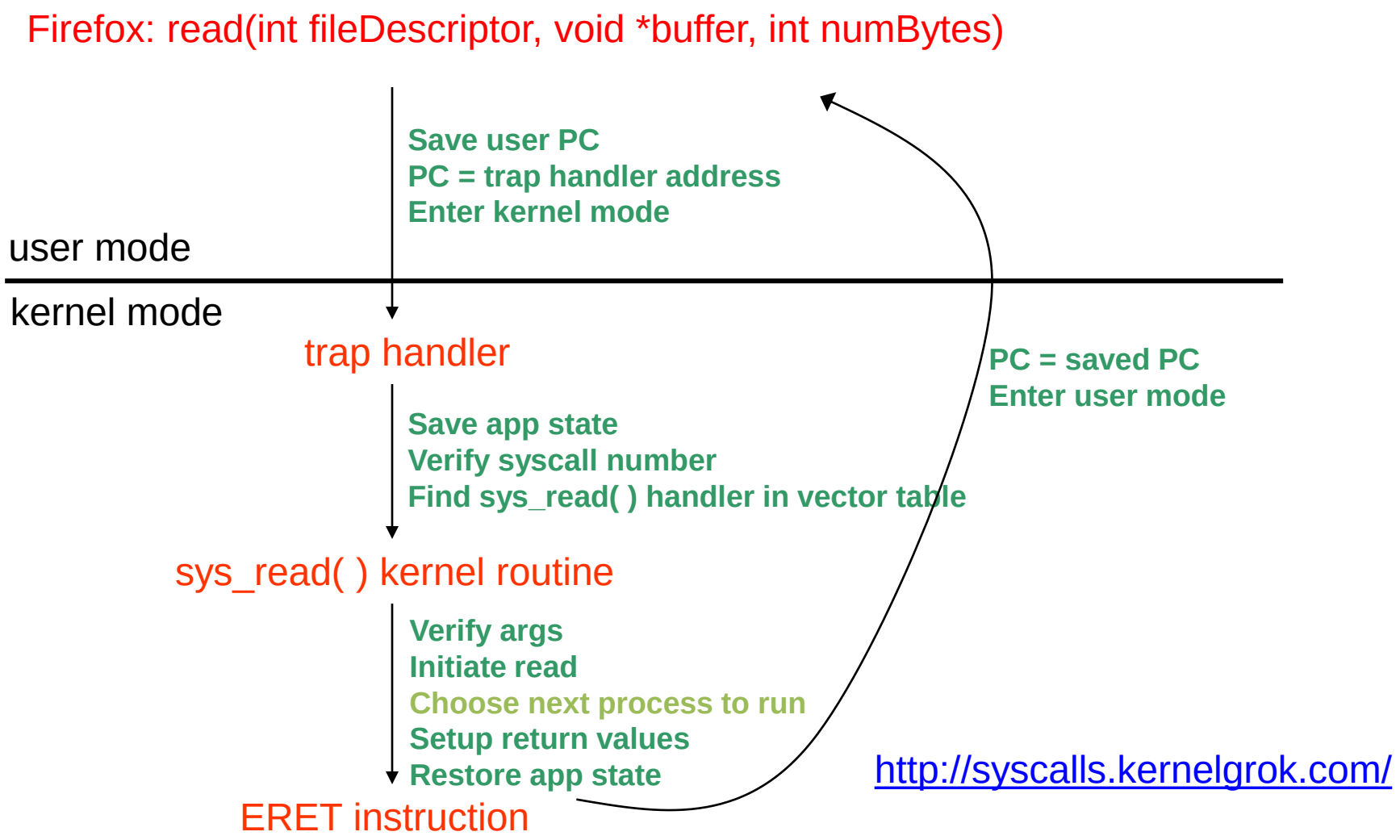
- So how do user programs do something privileged?
  - e.g., how can you write to a disk if you can't execute an I/O instructions?
- User programs must call an OS procedure – i.e., ask the OS to do that for them
  - OS defines a set of system calls
  - User-mode program executes system call instruction
- **Syscall instruction**
  - "protected procedure call"



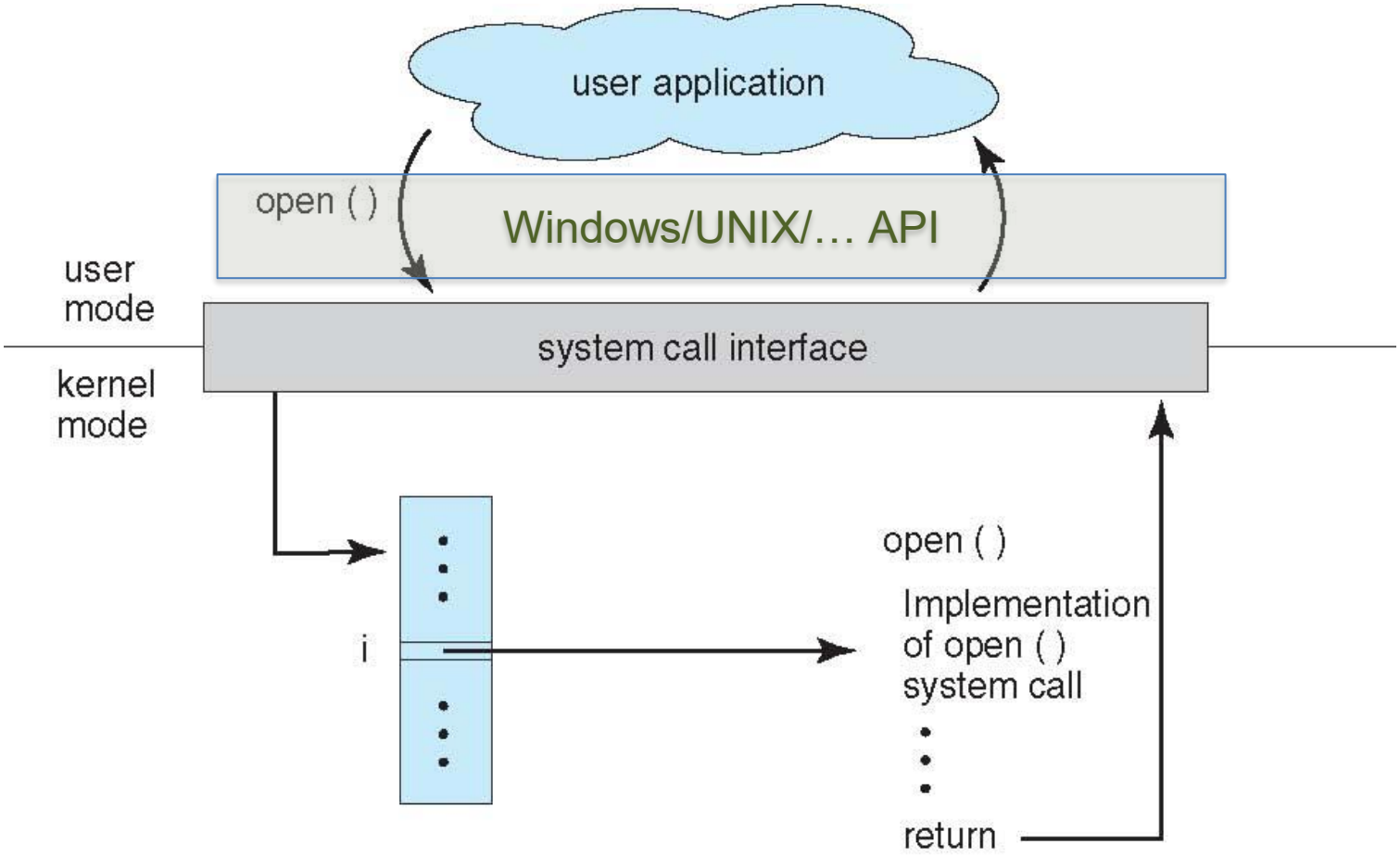
# Syscall

- The **syscall** instruction, **atomically,** on a single CPU/core
  - Saves the current PC
  - Sets the execution mode to privileged
  - Sets the PC to a handler address
- Similar to a **procedure call**
  - Caller puts arguments in a place callee expects (usually, registers)
    - One arg is a **syscall number**, indicating what OS function to call
    - Callee (OS) saves caller's state (registers, other control state) so it can use the CPU
  - OS **function** code runs
    - OS must verify caller's arguments (e.g., pointers)
  - OS **returns** using a special instruction
    - Automatically sets PC to return address and sets execution back to user mode

# Kernel Crossing Illustrated



# What Syscall to Run?

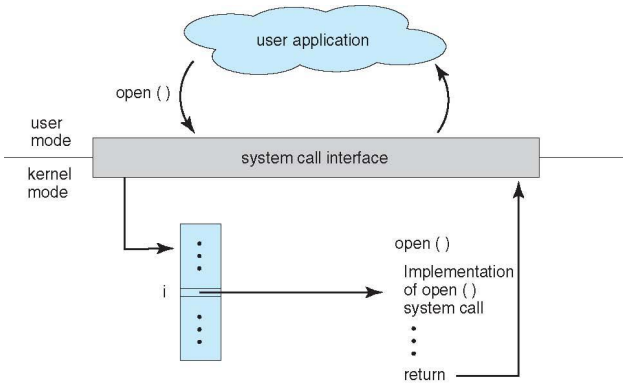


# Examples of Windows and Unix System Calls

| | Windows | Unix |
|---|---|---|
| Process Control | CreateProcess()<br>ExitProcess()<br>WaitForSingleObject() | fork()<br>exit()<br>wait() |
| File Manipulation | CreateFile()<br>ReadFile()<br>WriteFile()<br>CloseHandle() | open()<br>read()<br>write()<br>close() |
| Device Manipulation | SetConsoleMode()<br>ReadConsole()<br>WriteConsole() | ioctl()<br>read()<br>write() |
| Information Maintenance | GetCurrentProcessID()<br>SetTimer()<br>Sleep() | getpid()<br>alarm()<br>sleep() |
| Communication | CreatePipe()<br>CreateFileMapping()<br>MapViewOfFile() | pipe()<br>shmget()<br>mmap() |
| Protection | SetFileSecurity()<br>InitlializeSecurityDescriptor()<br>SetSecurityDescriptorGroup() | chmod()<br>umask()<br>chown() |

# System Call vs Subroutine Call

- Syscall **is not  subroutine call**, with the caller specifying the next PC
  - Caller knows where the subroutines are located in memory
    - Syscall is an ID
  - Subroutines trust each other
  - All subroutines share memory
- The kernel saves state
  - Prevents overwriting  of values
- The kernel verify arguments
  - Prevents buggy code crashing system
- Referring to kernel objects as arguments
  - Data  copied between user buffer and kernel buffer



# OS Services

- All entries to the OS occur via the mechanism just shown
  - Acquiring privileged mode and branching to the trap handler are **inseparable**
- **Terminology**
  - **Exception**: synchronous; unexpected problem with code
  - **Syscall**: synchronous; intended transition to OS
  - **Interrupt**: asynchronous; caused by an external device
- Privileged instructions and resources sharing are the basis for almost everything OS-related
  - memory protection, protected I/O, limiting user resource consumption, etc.