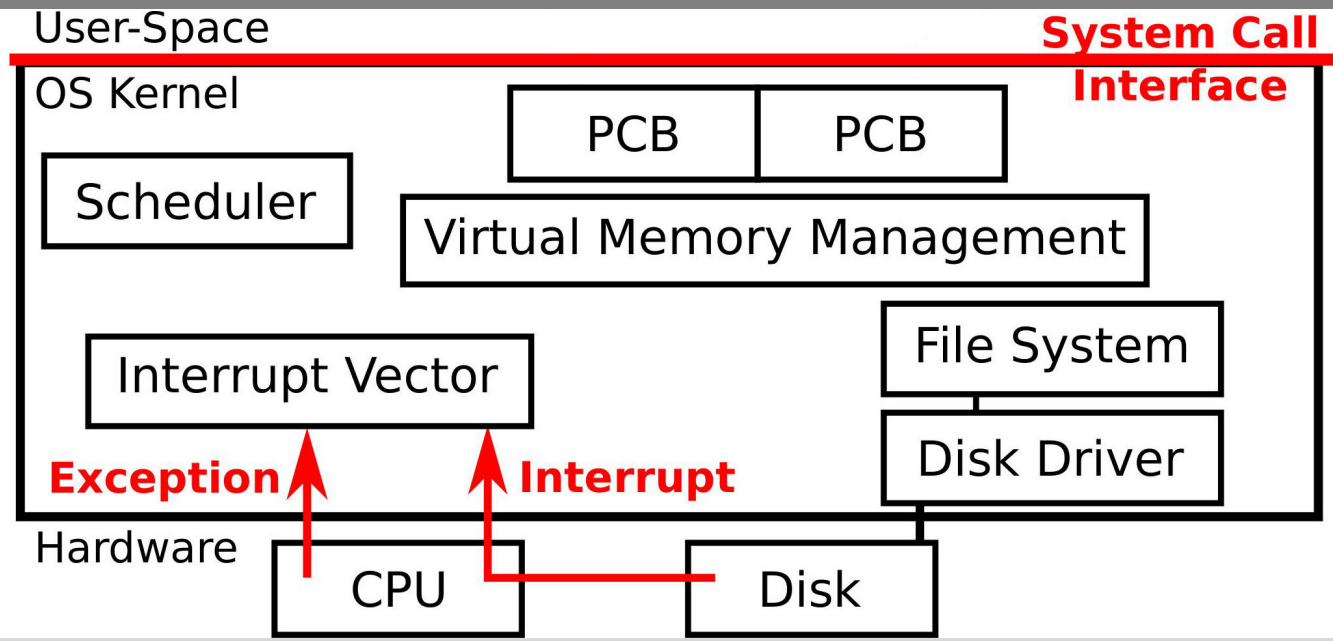
Karlsruhe Institute of Technology

Betriebssysteme

02. OS Concepts

Prof. Dr.-Ing. Frank Bellosa | WT 2016/2017

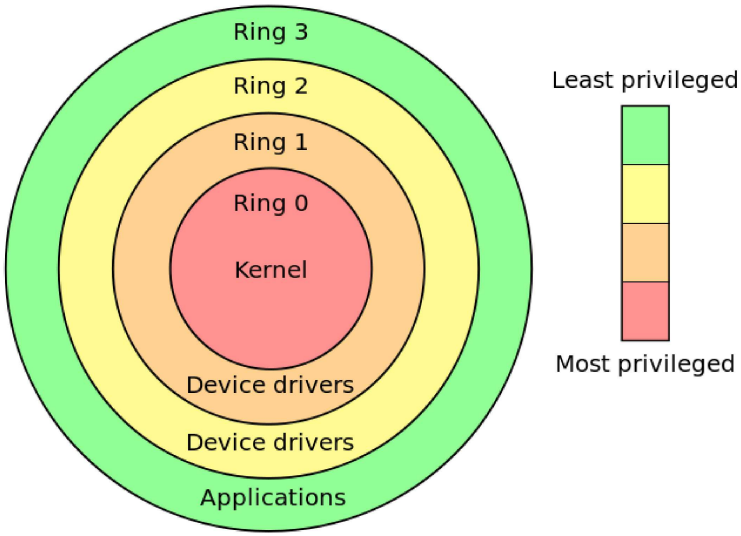


Where we ended last lecture

- OS makes hardware useful to the programmer
- Provides **abstractions** for applications
  - Manages and hides details of hardware
  - Multiplexes hardware to multiple programs
- Provides **protection**
  - E.g., from users/processes using up all resources
  - E.g., from processes writing into other processes' memory (address spaces)
- Protection requires hardware support!
  - Applications unprivileged (user-mode)
  - OS privileged (kernel-mode)

Central Processing Unit (CPU) - Modes of Execution

- User Mode (x86: “Ring 3” or CPL3)
  - Only non-privileged instructions may be executed
  - Cannot manage hardware in this mode → protection!
- Kernel Mode (x86: “Ring 0” or CPL0)
  - All instructions allowed: Can manage hardware with *privileged instructions*



OS Invocation

Invoking the Operating System

- The Operating System Kernel does **not** always run in the background
  - Not even if there are multiple cores/CPUs!

- Three occasions invoke the Kernel and switch to kernel-mode

  System calls User-Mode processes requires higher privileges

  Interrupts CPU-external device sends a signal

  Exceptions The CPU signals an unexpected condition

System Call Motivation

- Problem: Want to protect processes from one another
- Idea: Restrict processes by running them in CPU user-mode

- Problem: Now processes cannot manage hardware and other protected resources
  - Who can switch between processes?
  - Who decides if the process may open a certain file?

- Idea: The operating system provides services to applications (e.g., hardware management)
  - Application calls the system if service needed (System Call, syscall)
  - OS can check if application is allowed to perform the action that it asks for
  - If application may perform that action and has not exceeded its quota yet, the OS performs the action in kernel mode, on behalf of application

Examples of Linux System Calls

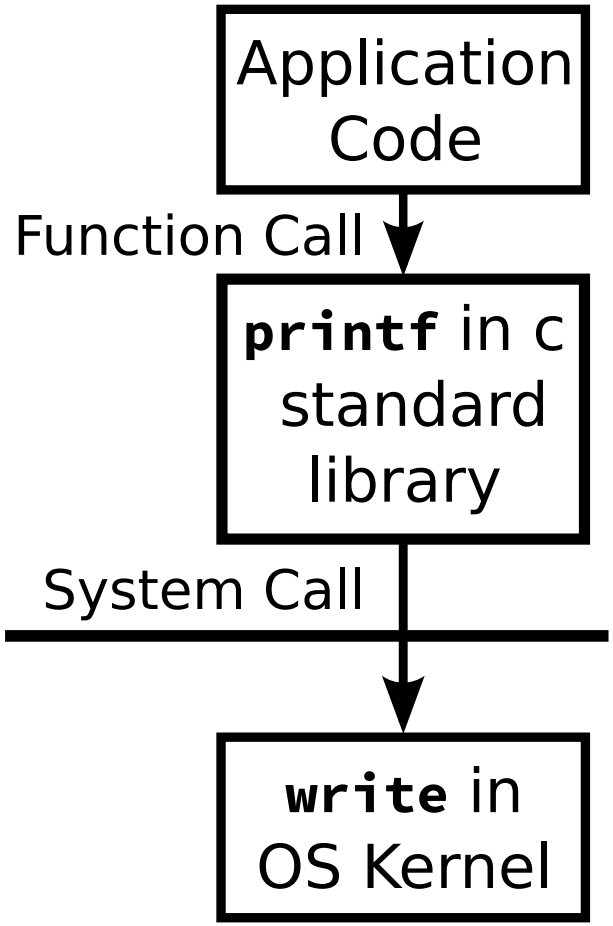
File management	
Call	**Description**
fd = open(file, how, ...)	Open a file for reading, writing, or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file

- In Linux system calls are documented in manual section 2
  - e.g., **`man 2 write`**
- An overview of all syscall is given in **`man 2 syscalls`**
- If you like colored keywords, consider replacing your default **`$PAGER`**
  - **`sudo apt-get install most`**, then **`export PAGER=most`** in **`.bashrc`**
  - **`vimpager`** is great if you like vim

System Calls vs. APIs

- The syscall interface between applications and OS services provides a limited number of well-defined entry points to the kernel

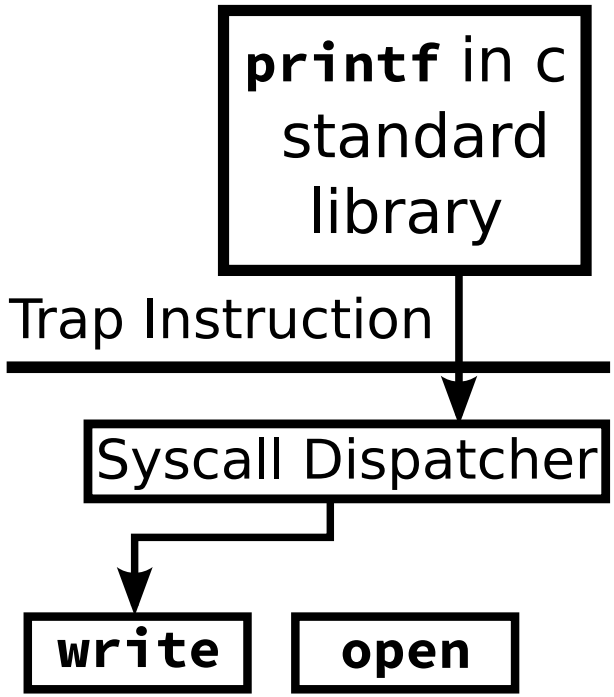
- Programmers often use syscalls via Application Program Interfaces (APIs)
  - In this example the **`printf`** library call uses the **`write`** system call to output text to the console.



- Most common APIs are
  - Win32 API for Windows
  - POSIX API for virtually all versions of UNIX, Linux, and Mac OS X
  - C API man pages can be found in **`man`** section **`3`** (e.g., **`man 3 printf`**)

System Call Implementation

- Although there are many different system calls, there is only one system call interface (entry point) into the kernel
- The trap instruction is that single entry point
  - The trap instruction switches the CPU to kernel mode and enters the kernel in the same, predefined way for every syscall (e.g., Intel: **`sysenter`**, AMD: **`syscall`**)
  - The system call dispatcher in the kernel then acts as a multiplexer for all syscalls



- Syscalls are identified by a number which is passed as a parameter
  - The system call table maps system call numbers to kernel functions
  - The dispatcher decides where to jump based on the number and table
  - Programs (e.g., stdlib) have the system call number compiled in! For compatibility: never reuse old numbers in future versions of kernel.

Interrupts

- Devices use interrupts to signal predefined conditions to the OS
  - Recall: The device has an “interrupt line to CPU”
  - e.g., device controller informs CPU that it has finished an operation
- The Programmable Interrupt Controller manages interrupts (e.g., x86 APIC)
  - Interrupts can be masked (ignored for now)
  - Masked interrupts are queued and delivered when the interrupt is unmasked
  - The queue has finite length → interrupts can get lost
- Notable examples for interrupts are
  - e.g., *Timer-Interrupt* periodically interrupts processes and switches to kernel → Can then switch to different process to enforce fairness between processes
  - e.g., *Network Interface Card* interrupts CPU when a packet was received → Can deliver the packet to process and free the NIC buffer

Interrupts

- When interrupted, the CPU
  - looks-up the interrupt vector, a table that is pinned in memory and contains the addresses of all service routines (set up by the OS)
  - transfers control to the respective interrupt service routine in the OS that handles the interrupt

- The interrupt service routine must first save the state of the interrupted process
  - Instruction pointer
  - Stack pointer
  - Status word

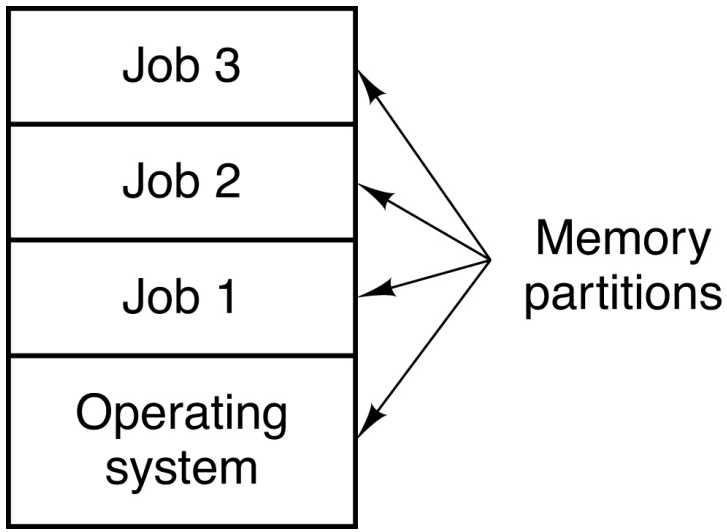
Exceptions

- Sometimes, an unusual condition makes it impossible for the CPU to continue processing
  - What should happen if a program calls `div` with a zero denominator?
  - What should happen if a program tries to write a read-only memory area?
  - What if the program jumps to an invalid opcode?
- On such occasions, an exception is generated within the CPU
  - The CPU interrupts the program and gives the kernel control
  - The kernel can determine the reason for the exception
  - If the kernel can resolve the problem it does so and continues the faulting instruction
  - Otherwise it kills the process
- Note that, in addition to the source, there is another distinction between interrupts and exceptions
  - Interrupts can happen in any context
  - Exceptions always occur synchronous to and in the context of a process

OS Concepts

Physical Memory

- Up to the early 60's programs were loaded into and run directly into physical memory
- If the program was too large, the programmer partitioned his program manually into overlays
- The OS could then swap overlays between disk and memory
- Different jobs could observe and modify each others memory contents



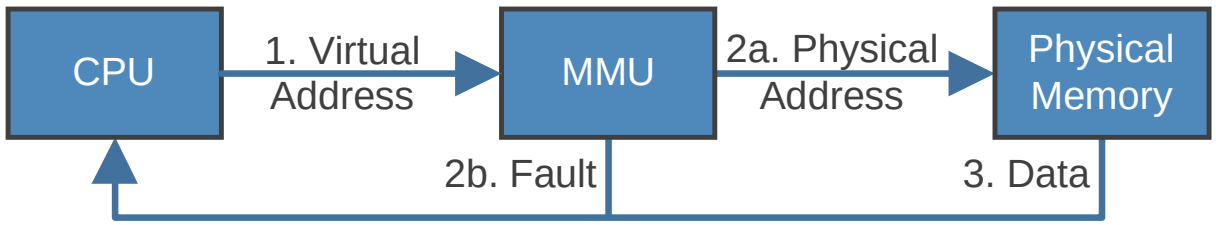
Address Spaces

- Need to isolate bad programs and people. Otherwise:
  - Buggy program could trash other programs
  - Malicious user could steal other users passwords
  - Selfish user could use up all memory for himself

- Idea: Give every job the illusion of having all memory to itself
  - Every job has its own address space and cannot name addresses of other jobs
  - Jobs always and only use virtual addresses

Virtual Memory: Indirect Addressing

- Today, every CPU has a memory management unit (MMU) built-in
- The MMU translates virtual addresses to physical addresses at every load and store operation
- Address translation protects one program from another



- Definitions
  - Virtual address address in process' address space
  - Physical address address of real memory

Memory Protection

- MMU allows kernel-only virtual addresses
  - Kernel typically part of all address spaces
  - Ensures that applications can't touch kernel memory
- MMU can enforce read-only virtual addresses
  - Makes safe sharing of memory between applications possible
- MMU can enforce execute disable
  - Makes code injection attacks harder

Page Faults

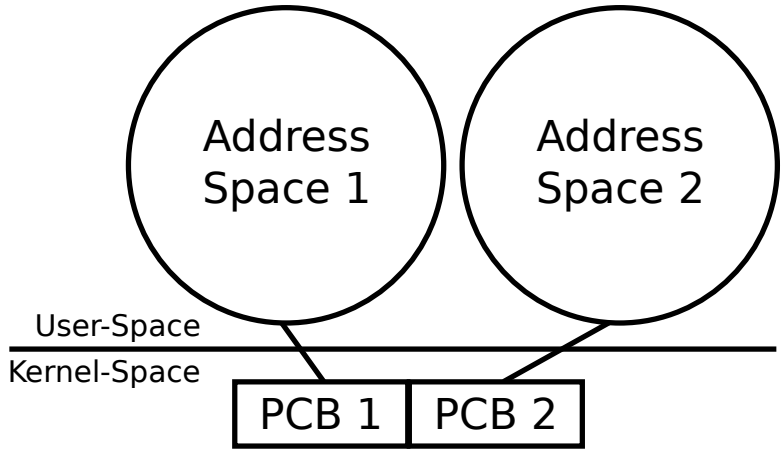
- Not all addresses need to be mapped at all times
  - MMU issues a page fault exception when an accessed virtual address is not mapped
  - The OS handles page faults by loading the faulting addresses and then continuing the program

  → Memory can be over-committed: More memory than physically available can be allocated to application

- Page faults are also issued by the MMU on illegal memory accesses e.g., if an application tries to
  - access kernel memory
  - write read-only memory
  - set the instruction pointer to executable disable memory

Processes

- A process is a program in execution – an “instance” of a program
- Each process is associated with a process control block (PCB)
  - Information about allocated resources, e.g., open files with seek pointer



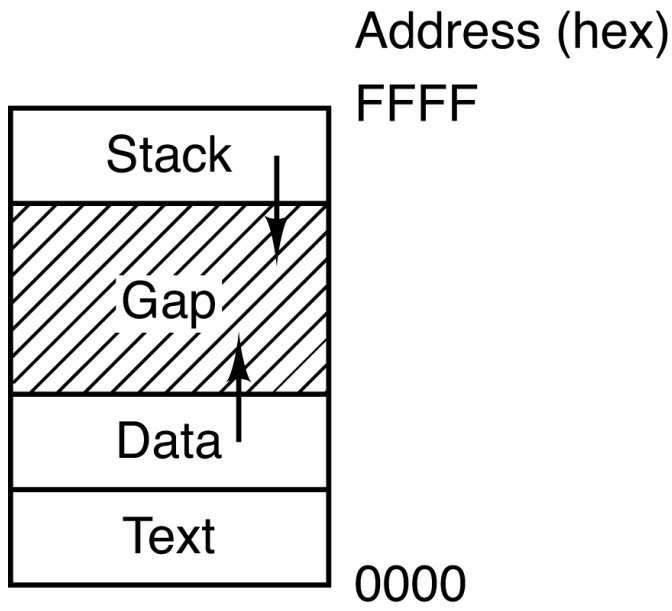
- Each process is associated with an virtual address space (AS)
  - All (virtual) memory locations a program can name
  - Starts at 0 and runs up to a maximum
  - Address 123 in AS1 is generally not the same as address 123 in AS2
  - Indirect addressing makes it possible to give different processes different AS

→ Protection between processes: If you can’t name it, you can’t touch it

Address Space Layout

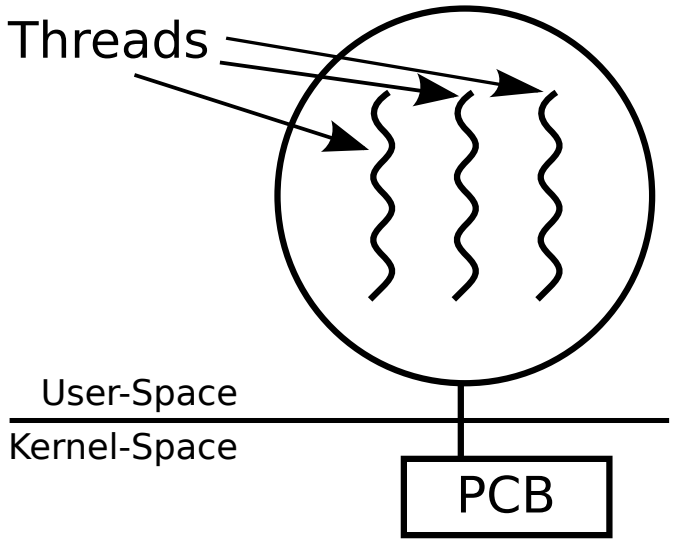
- Typically address spaces are laid-out in different sections
  - Memory addresses between sections are illegal
  - Using such illegal addresses (also) leads to a page fault
  - Such page faults are more specifically called segmentation fault
  - The OS generally handles segmentation faults by killing the faulting process

Stack	Function history and local variables
Data	Constants, static variables, global variables, strings
Text	Program code



Threads

- Each process consists of ≥ 1 threads representing execution states
  - Instructions pointer register (IP) stores the currently executed instruction
    - An address in the text section
  - Stack pointer (SP) register stores the address of the top of the stack
    - With > 1 threads, there are also multiple stacks!
  - Program status word (PSW) contains flags about the execution history
    - e.g., last calculation was zero → used in following jump instruction
  - And more, e.g., general purpose registers, floating point registers, ...



Policies vs. Mechanisms

When designing an OS it is useful to separate policies from mechanisms

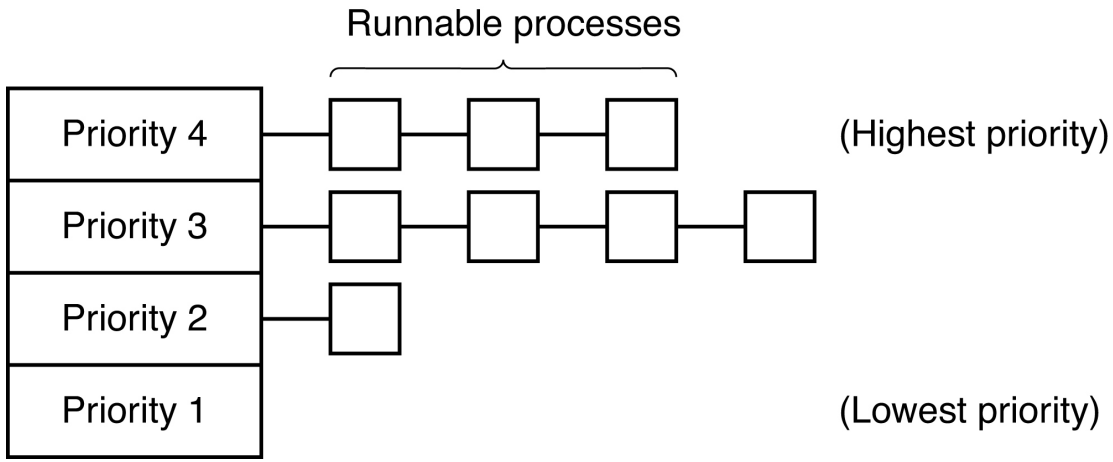
Mechanism Implementation of what is done
(e.g., the commands to put a HDD into standby mode)

Policy The rules which decide when what is done and how much
(e.g., how often, how many resources are used, . . .)

Mechanisms can be reused even when the policy changes

Scheduling

- With multiple processes and threads available, the OS needs to switch between processes to provide multi-tasking
- The scheduler decides which job to run next (policy)
- The dispatcher performs the actual task-switching (mechanism)
- Schedulers try to
  - provide fairness
  - while meeting goals
  - and adhering priorities

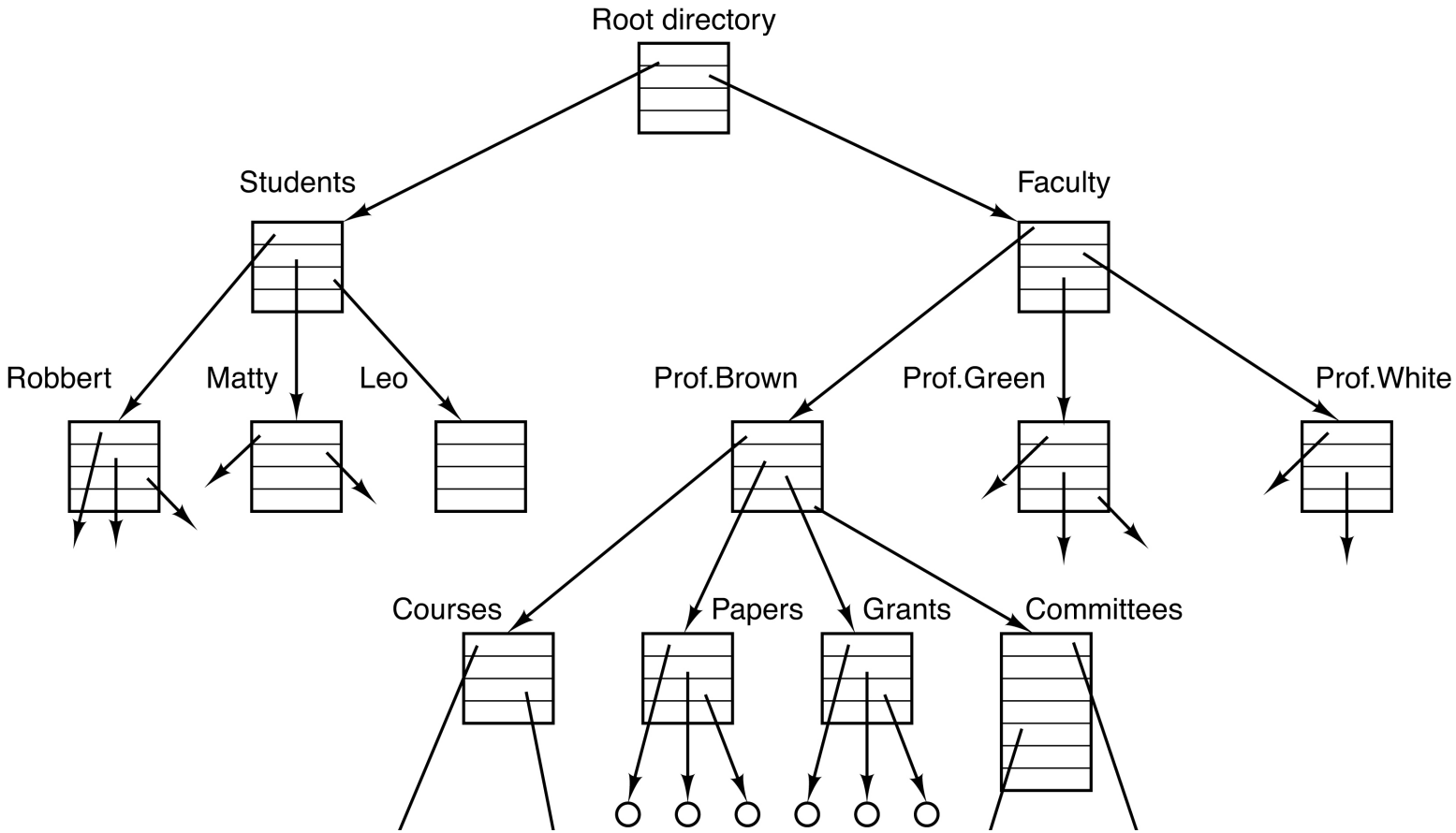


Files

- The OS hides peculiarities of disks and other I/O devices
- Programmer uses device-independent files and directories for persistent storage
  - Files associate file name and offset with bytes
  - Directories associate directory names either with other directory names or with file names
- The file system is an ordered collection of blocks
  - Main task: translate (directory name + file name + offset) to block
  - Programmer uses file system operations to operate on files
    - `open`, `read`, `seek`
- Processes can communicate directly through a special named pipe file
  - Pipes are used with the same operation as any other file

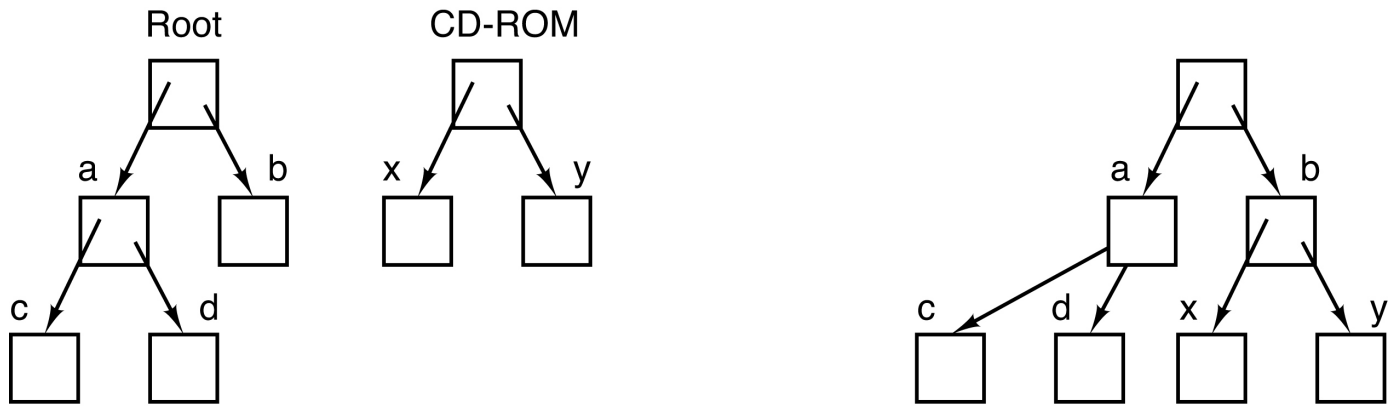
Directory tree

- Directories form a directory tree/file hierarchy → structure data
- The root directory is the topmost directory in the directory tree
- Files are specified by providing the path name to the file



Mounting File Systems

- In UNIX like systems, it is common to orchestrate multiple file systems in a single file hierarchy
  - File systems can be mounted on a directory



- In Windows, users manage multiple directory hierarchies named with drive letters
  - E.g.: `C:\Users`

Storage Management

- OS provides uniform, logical view of information storage to file systems
  - Drivers hide specific hardware devices → hides peculiarities of devices
  - General interface abstracts physical properties to logical units → block

- OS increases the performance of I/O devices

Buffering	Store data temporarily while it is being transferred
Caching	Store parts of data in faster storage for performance
Spooling	Overlap of output of one job with input of other jobs

Summary

- The OS provides abstractions for and protection between application
- The kernel does not always run: Certain events invoke the kernel
  - System call: Process asks the kernel for a service (e.g., open a file)
  - Interrupt: Device sends signal that the OS has to handle (e.g., I/O finished)
  - Exceptions: CPU encounters unusual situation (e.g., divide by 0)
- Processes encapsulate resources needed to run a program in the OS
  - Threads: represent different execution states of a process
  - Address space: all memory the process can name
  - Allocated resources, e.g., open files
- The scheduler decides which process to run next when multi-tasking
- Virtual memory implements address spaces and provides protection between processes
- The I/O drivers and file system abstract the background store
  - Simple interface: Files and directories

Further Reading

- Tanenbaum/Bos, “Modern Operating Systems”, 4th Edition: Pages 38–50
- Stallings, “Operating Systems – Internals and Design Principles”, 6th Edition: Pages 50–104
- Silberschatz, Galvin, Gagne, “Operating System Concepts”, 8th Edition: Pages 23–54