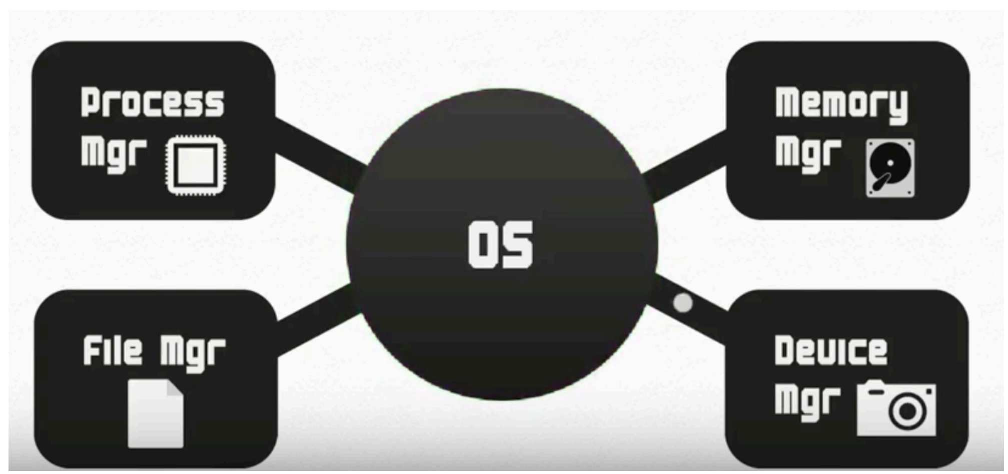
Operating systems

Essential skills 2016-2017

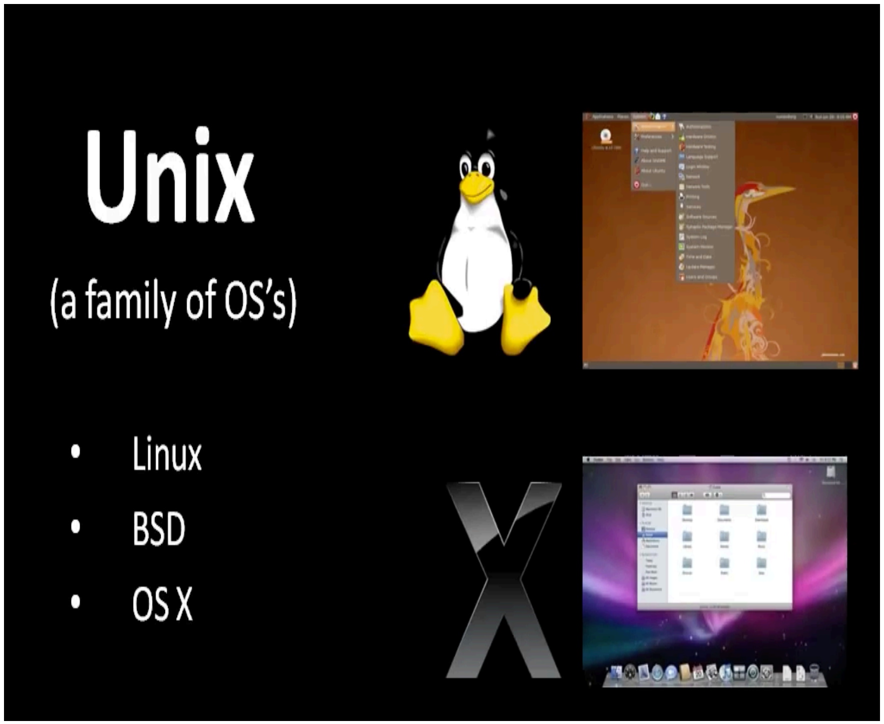
Adam Belloum

content

- Introduction to Operating systems (OS)
- Process management
- Memory management
- File system
- Device management

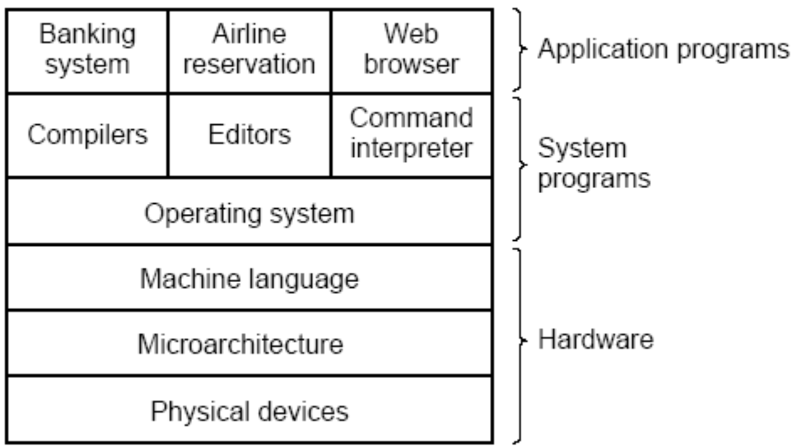


Most known Operating systems



Operating systems

- What do they do:
  - OS as a Virtual Machine
    - **Abstraction →** decouple applications from low level device details
  - OS as a Resource Manager
    - **Arbitration →** Fair sharing of **resources** (scheduling), protect against simultaneous usage of resources



Abstraction or Arbitration?

- Support both Intel and AMD processor
- Switching between Applications
- Separating memory Allocated to different applications
- Enabling video card software to use different camera devices
- Access two different hard disks
- Sending/receiving messages over the network

Hardware resources

- Central Unit processing (CPU)
  - Execute program instructions
  - Multiple CPU cores execute instructions in parallel
- Memory
  - Hierarchy of different memory speeds
  - Fastest memory attached to CPU: registers and cache
  - Random Access memory (RAM) - Slower
  - Persistent memory (disk) – slowest
- Input/output (I/O) devices
  - Keyboard, mouse, Network Interface Card, Screen, printer, …

How to view an OS

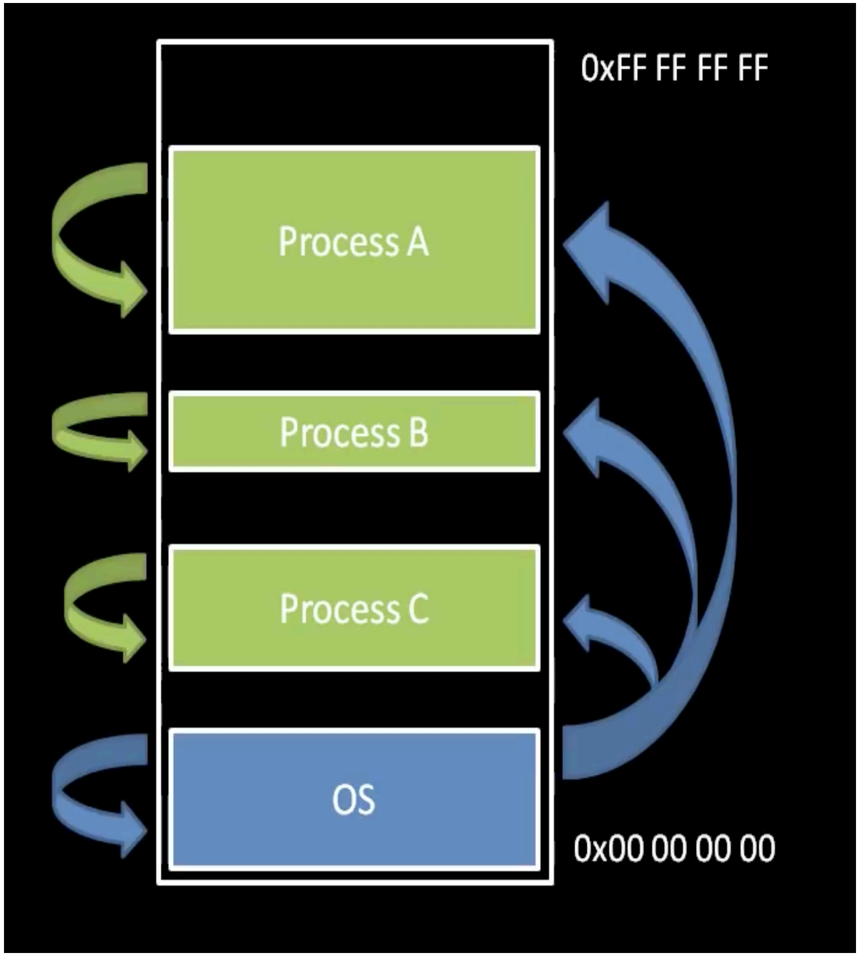
- From Application/user point of view
  - OS appears to the application and users as a **library of functions** ➔ system calls
- Form OS point of view
  - A **process** represent a user or an application and executes a program.
  - Data needed for the processing is retrieved from and stored in **files**. Data in files are persistent over processes

process = program in execution

Process

- Each process has its own memory space

- A process is **not allowed to access** other processes memory space.



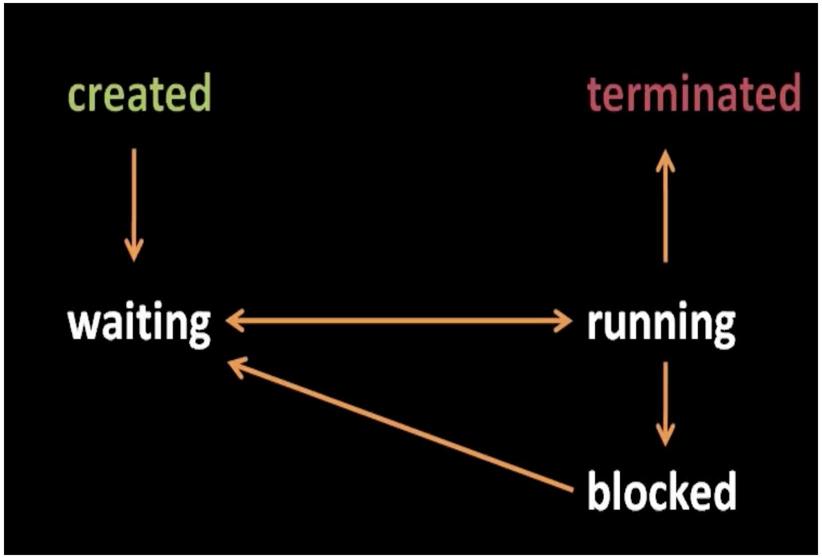
Process lifecycle

1. Process is first created

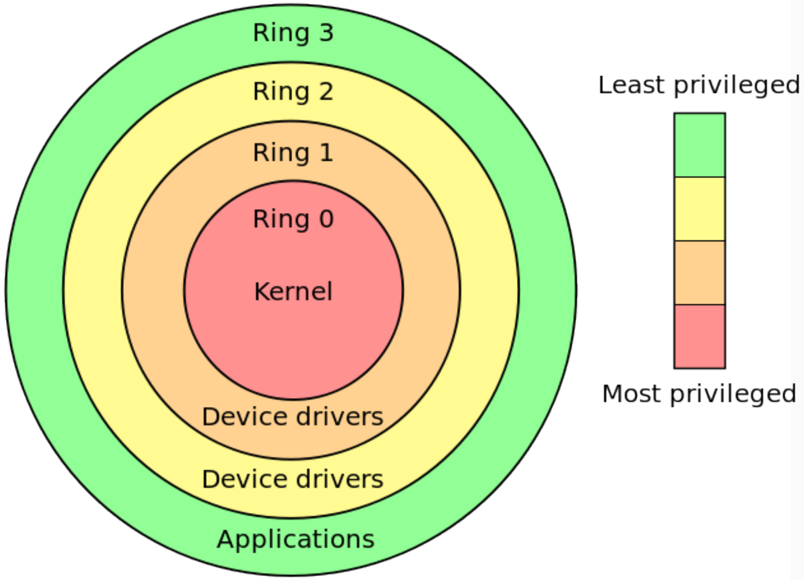
2. Then goes to the **waiting state** → for the OS (scheduler) to select the process

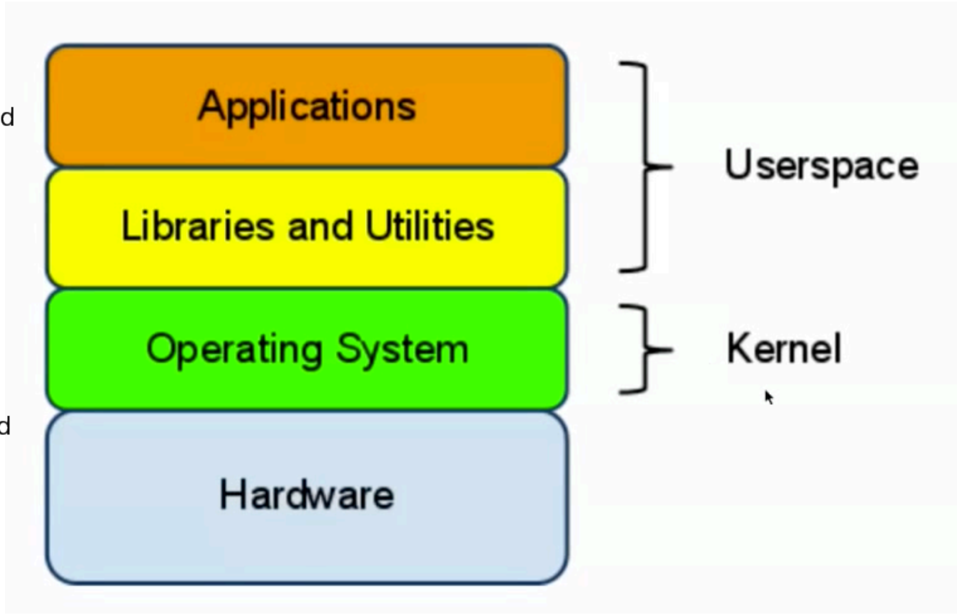
3. Then keep bouncing between two states: **running** and **waiting**
   – Can be interrupted

4. When the process request to resource or data not available it goes to the **blocked state**

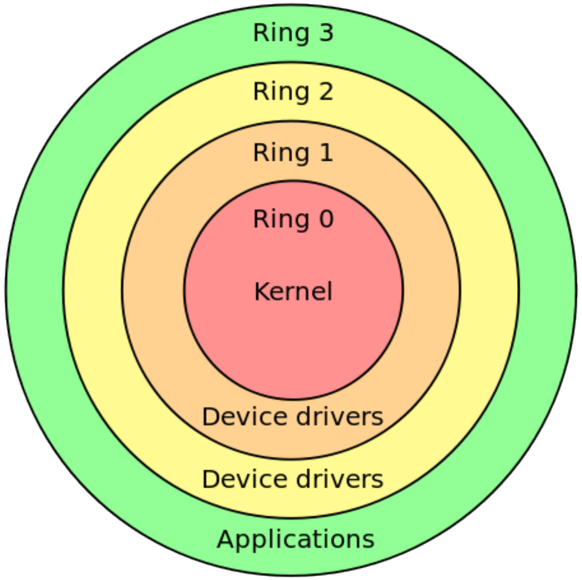


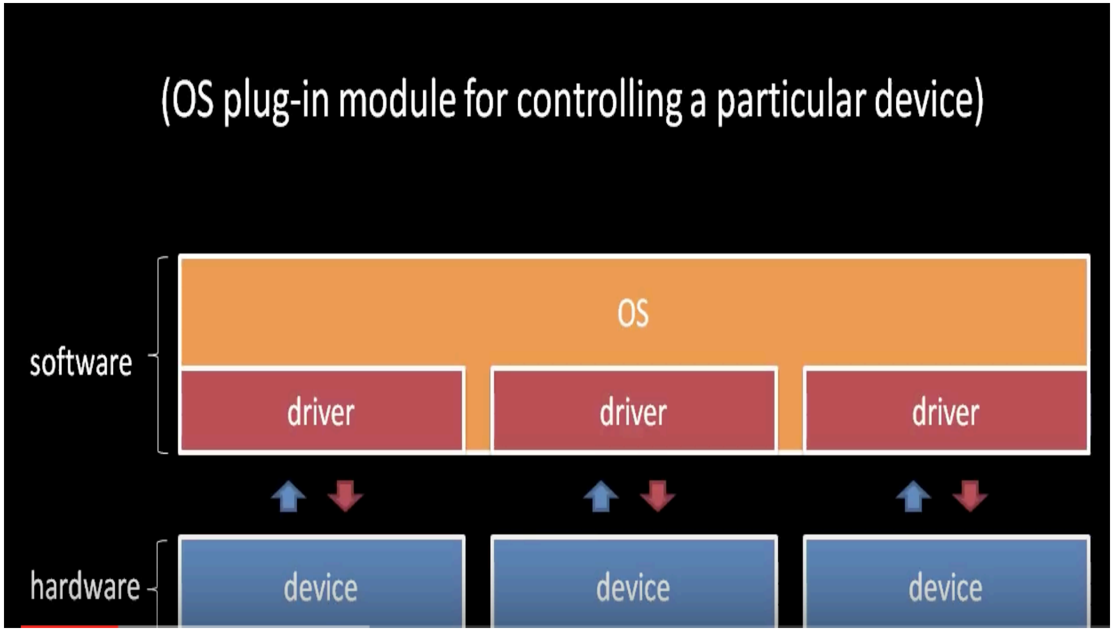
Kernel and user space





Device driver





Communicating with Hardware

OS implement a common mechanism for allowing **applications** to access hardware (abstraction) and the way around

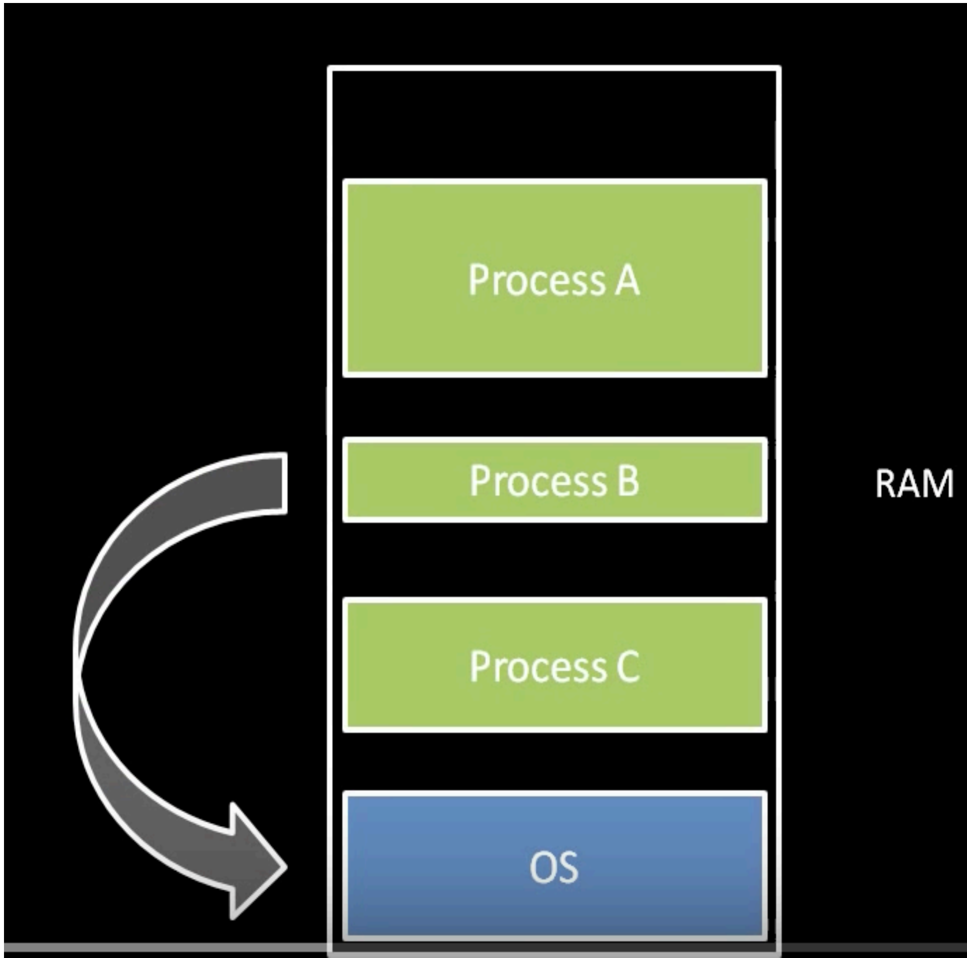
- Application make requests to the OS via **system calls**

- OS alert or terminate Applications via **signals**

System calls

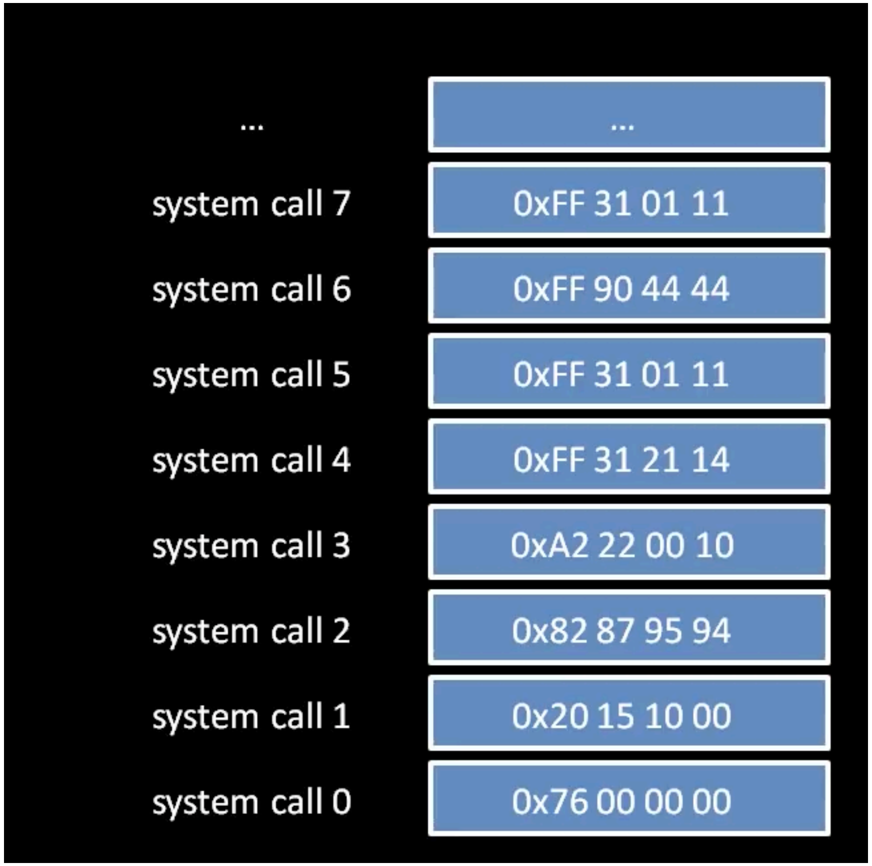
A System call allow a process to access OS space

- Initiate request to the OS to open a file, send/receive data over the network etc.



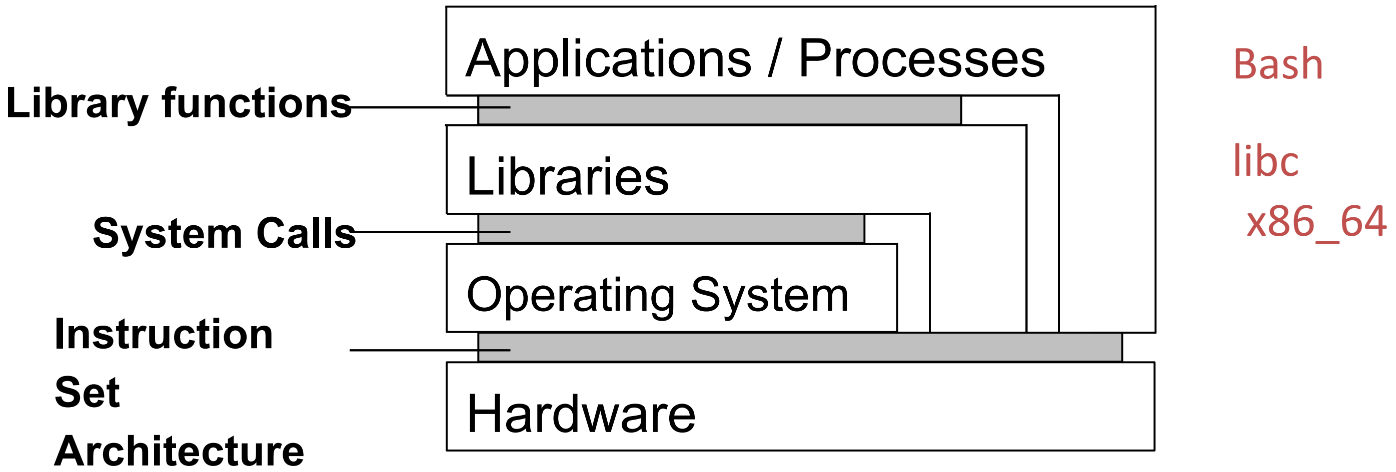
System call table

- In a system call the process specify a **system call number**
- The CPU looks in the **system call table** for the address of the routine to be executed



System calls

- Problem: Mechanics of issuing a system call are highly machine dependent
- Solution: Provide a library to allow system calls from C programs: libc



System calls: process Management

- Consider a very simple command shell
  1. Wait for the user to type in the command.
  2. Start the process that execute the program
     - Load the specified file into memory
     - And execute the program ➔ process
  3. Wait until the child process finishes

System calls: process Management

Needed: (1) process creation, (2) have process execute a file, (3) have a process wait for a child to finish.

```
while(TRUE){
  read_command(command, parameters);
  pid = fork();

  if( pid != 0 ){ /* parent process */
    waitpid( pid, &status, 0 );
  }
  else{ /* child process */
    execve(command, parameters, 0);
    exit(0);
  }
}
```

fork: Create a child process identical to the parent

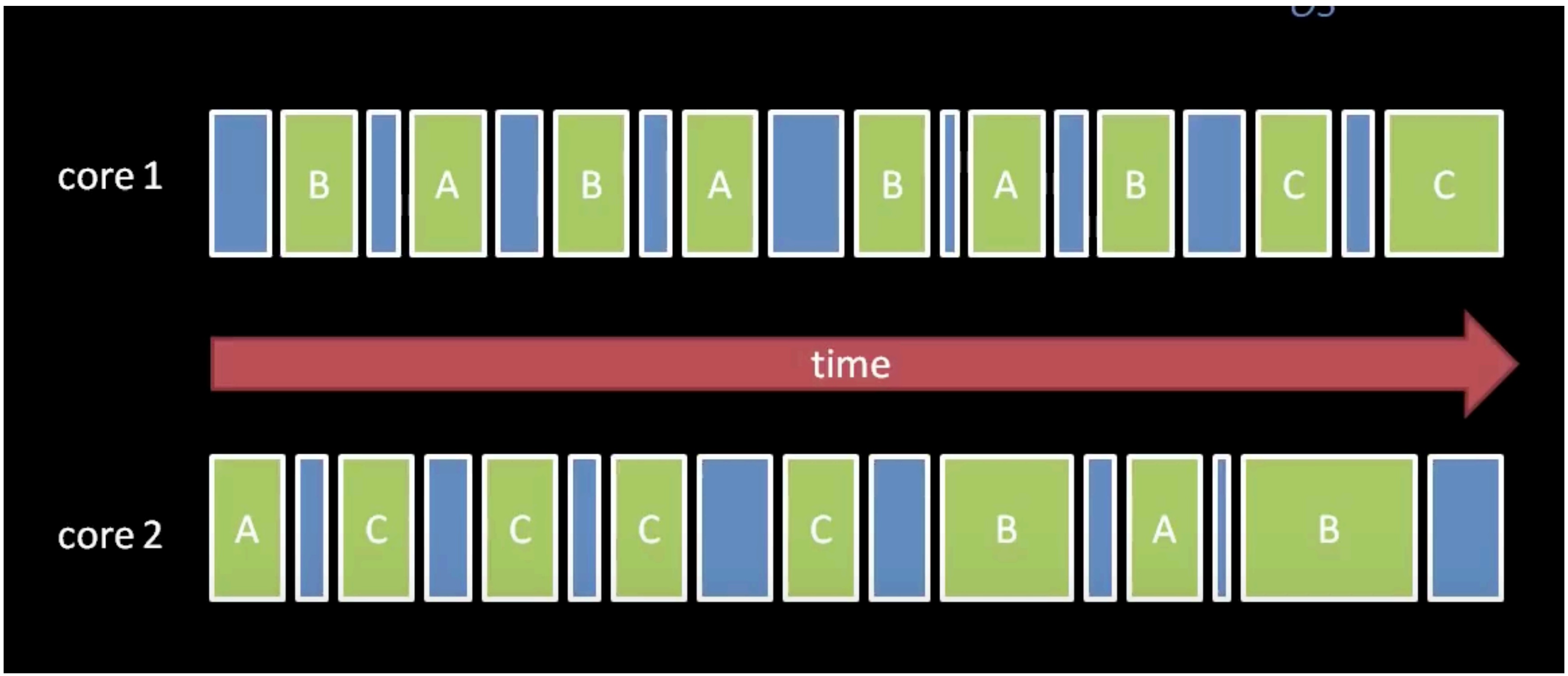
waitpid: Wait for a (specific) child to terminate

execve: Replace a process's core image

exit: Terminate process execution

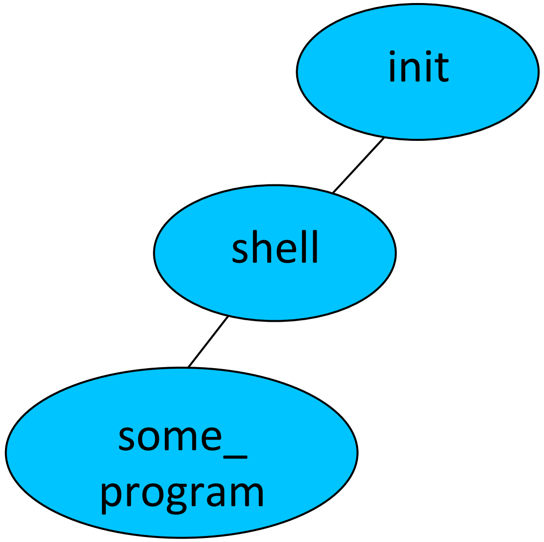
Multiple process

- Modern OS can execute more than one process at the time



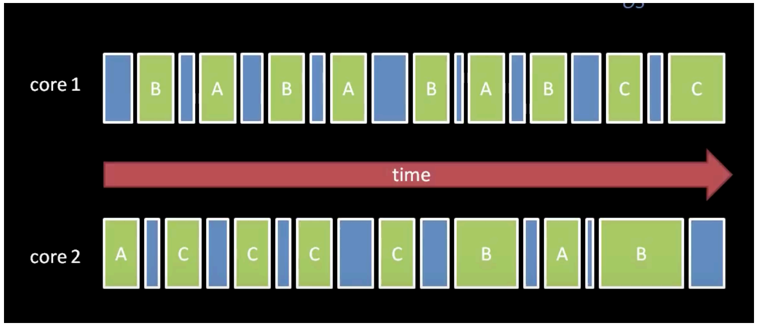
Multiple processes

- How do we get multiple processes in the first place?
  - Process can create other processes
  - Example: the command shell



Multiple processes

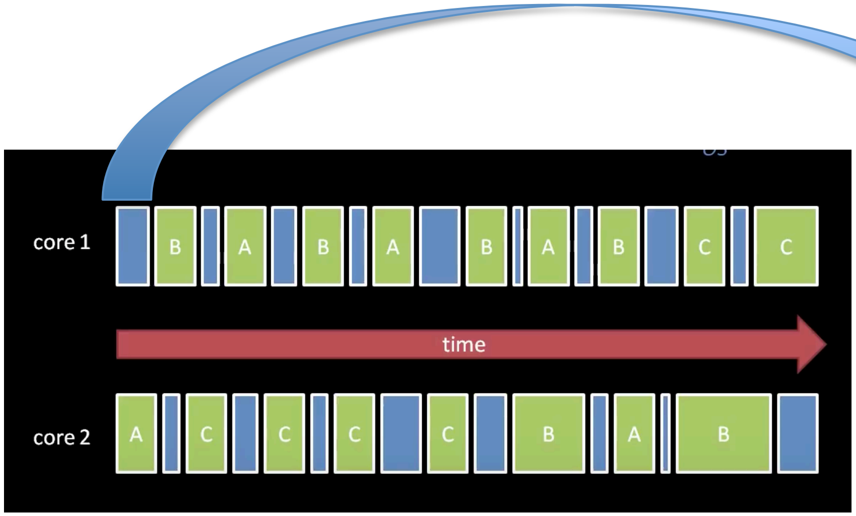
- How the OS executes multiple processes on the same resources (CPU, memory , I/O devices) at the same time?
  - OS has to interrupt the processes while they are running
    1. OS send a **signal** to a process to interrupt it,
    2. Process **catch the signal** by executing a specific handler



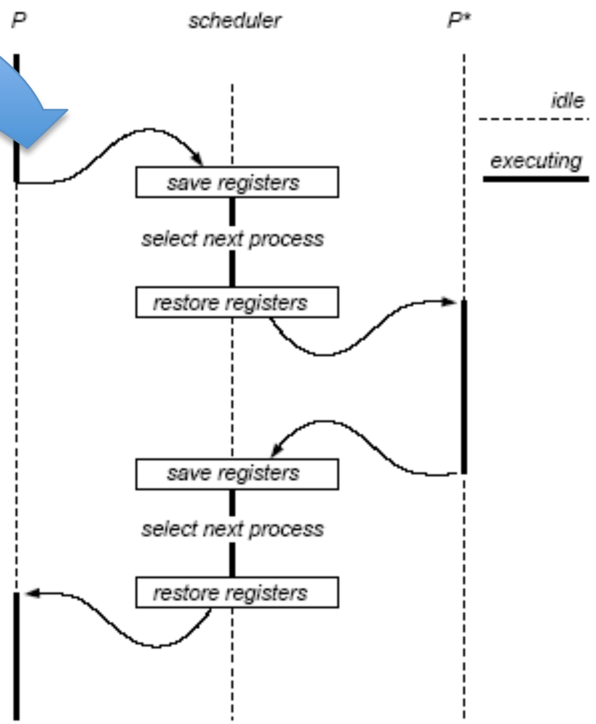
Pre-emptive multitasking

- CPU receives interrupts
- Interrupts stores Program Counter (PC)
- Interrupts invokes handler
- Handler save rest of state the CPU for the process
- Handler does it work
- Handler invokes the scheduler
- Scheduler selects a process to run
- Scheduler restores the state of the CPU that process
- Scheduler jumps execution to that process

Context switch



Problem: We have to change from one process to another. The stuff that is going to be used by another process should be saved $\Rightarrow$ CPU registers.



Threads vs processes

- Thread fall under the regime of a **single process**, and thus reside in the **same address** space
  - Each thread has its **own stack**, processor **context**, and **state**;
  - Threads synchronise through simple primitives (**semaphores** and **monitors**)
  - Each thread may call upon any service provides by the OS, it does it on behalf of the process to which belongs.

Threads –Some Problems

- Does the OS keep an administration of threads (kernel threads), or not (user threads)?
- What happens if a user thread does a blocking system call?
- when process is cloned does the new process get all the threads as well? What about the threads currently blocking on a system all?
- When the OS send a signal, how can you related the signal to a thread shloud you relate it to a theard?

Mutual Exclusion

- Critical Region: is a part of a **multi-process** program (piece of code) that **may not be concurrently** executed by more than one of the program's processes/threads
- Solutions
  - Semaphores
  - Monitors

Semaphores

- Semaphores are **data structure** that provides mutual exclusion to critical sections:
- Semaphores support two operations
  - Wait (Sem): block until semaphore is open
    - P(), after the Dutch word test (proberen)

  - Signal (sem):allow another thread to enter
    - V() after the Dutch word increment (verhogen)

Semaphore types

- Mutex Semaphore
  - Represent a **single access to** a resource
  - Guarantees a mutual exclusion to a CS

- Counting semaphore
  - Represent **a resource with many units available**
  - Multiple threads can pass the semaphore

Semaphore in use

```
struct Semaphore {
    int value;
    Queue q;
} S;
withdraw (account, amount) {
    wait(S);
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    signal(S);
    return balance;
}
```

```
wait(S);
balance = get_balance(account);
balance = balance – amount;
```

```
wait(S);
```

```
wait(S);
```

```
put_balance(account, balance);
signal(S);
```

```
…
signal(S);
```

```
…
signal(S);
```

Threads block

It is undefined which thread runs after a signal

Monitors

- A monitor is a **programming language construct** that controls access to the shared data
  - Synchronise code added by the compiler, enforced at runtime
- A monitor is a module that encapsulates
  - Shared data structures
  - Synchronisation Procedures
- A monitor **protects** its data form **unstructured access**
  - It guarantees that threads accessing its data through its procedures interact only in well defined way

Condition variables

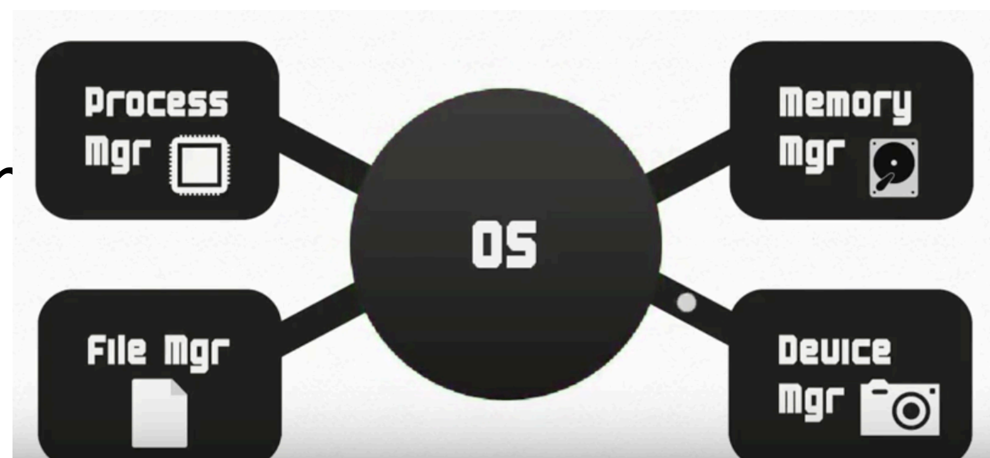
- Condition variables provide a mechanism to **wait for events** (a "rendezvous point")
- Condition variables support three operations:
  - **Wait** – release monitor lock, wait for C/V to be signaled
    - So condition variables have wait queues, too
  - **Signal** – wakeup one waiting thread
  - **Broadcast** – wakeup all waiting threads

Access to the monitor is controlled by a **lock**

Monitor	semaphore
Wait() • **blocks** the **calling thread** on the queue, • and **gives up the lock** To call wait, the thread has to be in the monitor (hence has lock)	Wait() or P() • **blocks calling thread** on the queue
Signal () • causes a **waiting thread to wake up** If there is no waiting thread, **the signal is lost**. → Condition variables have no history	Signal () or V() • **increases the semaphore count**, allowing future entry even if no thread is waiting

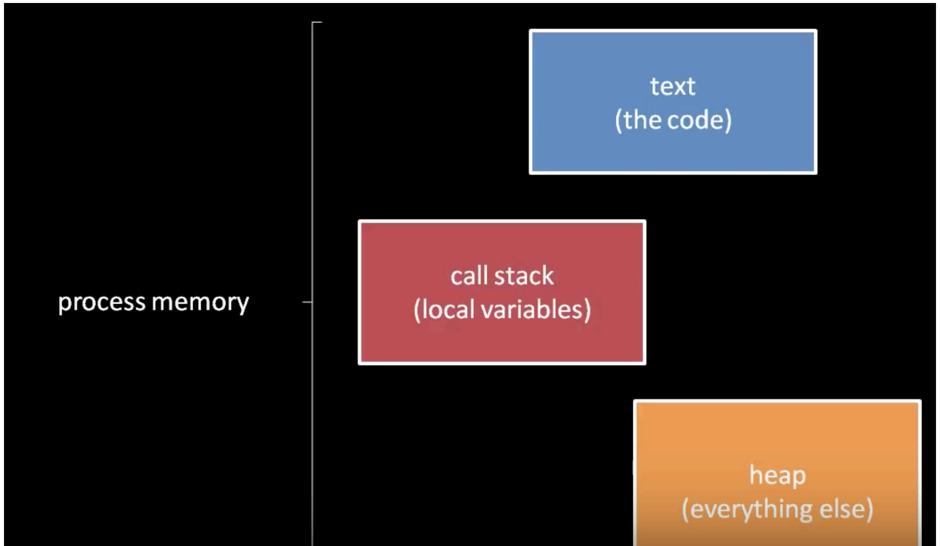
content

- Introduction to Operating systems (OS)
- Process management
  - Single process management
  - Multi process management
- Memory management
- File system
- Device managem



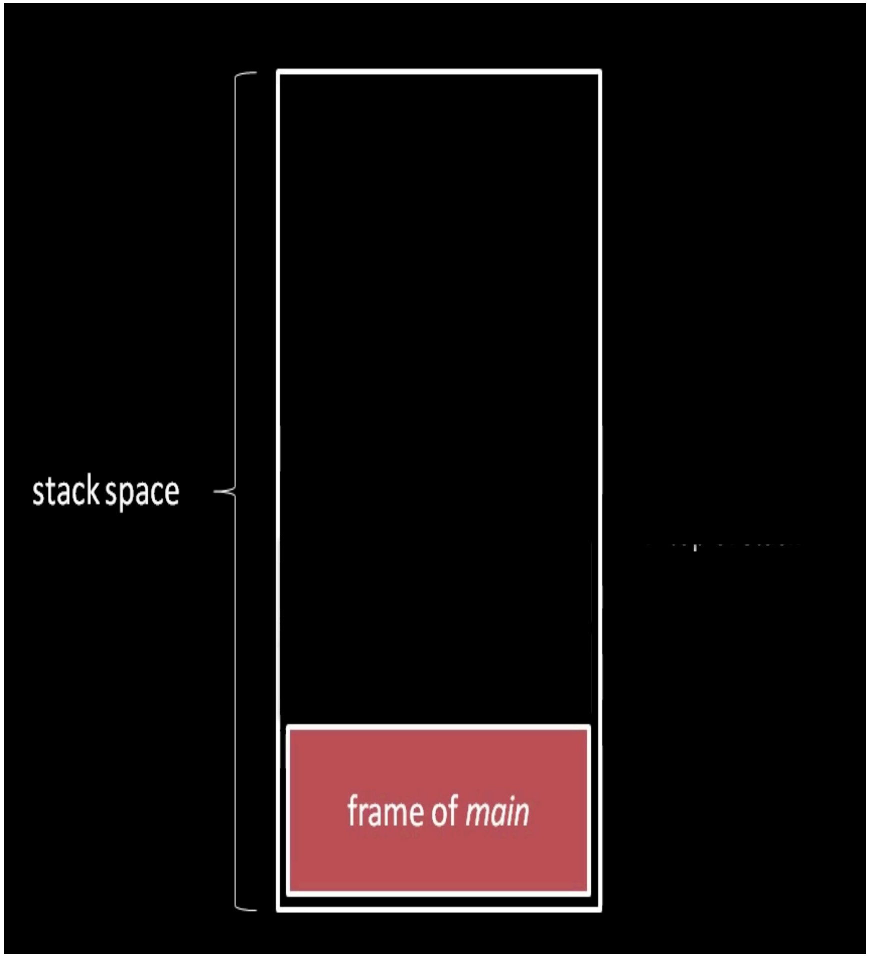
Process Memory organization

- Process code (text)
  - Never modified during execution except of dynamic linking with shared library
- Process **stack** (**local variables**)
- Process **heap** (**for anything else**)



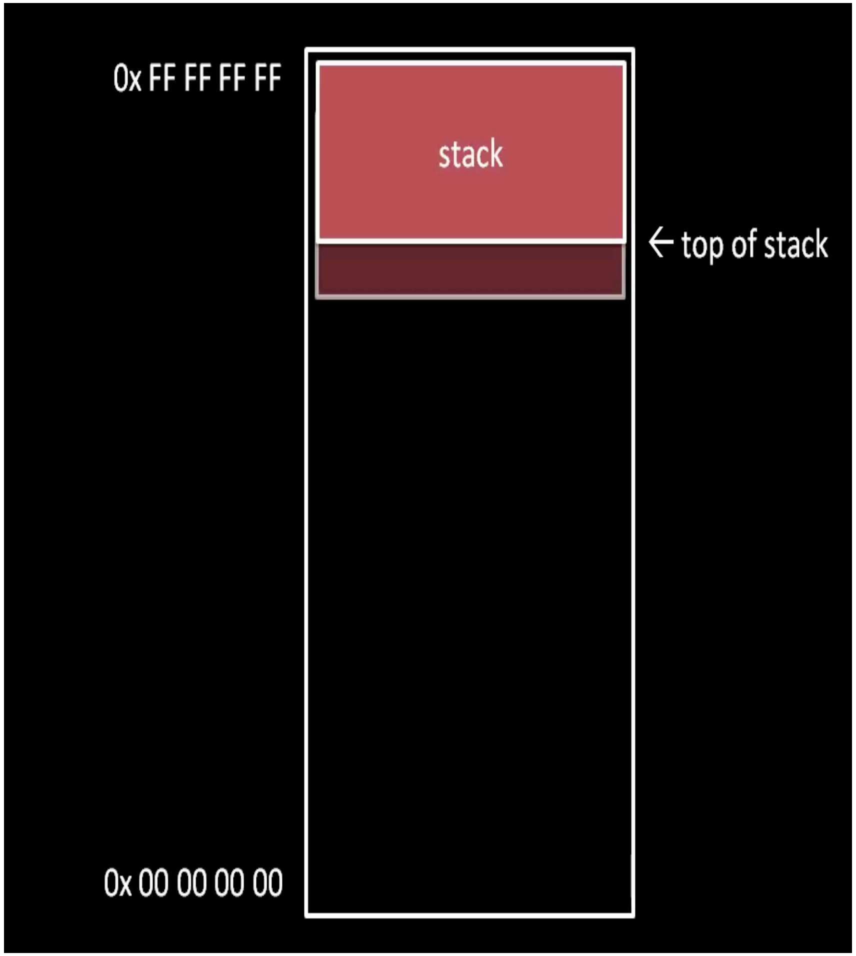
Stack management 1/2

- Start unused **when functions** are called address space is reserved for the function to save:
  - Local variable of the called function
  - Return address
  - Size of the frame
- Top of the stack is keep in a special register called the **stack pointer**



Stack management 2/2

- The **size** of the stack is kept in another register
- When stack pointer reaches this stack boundary → it triggers and exception
  - Increase the size of the stack
  - terminate the process if the stack is too big
- Problem: When a program exceeds in stack space → stack overflow

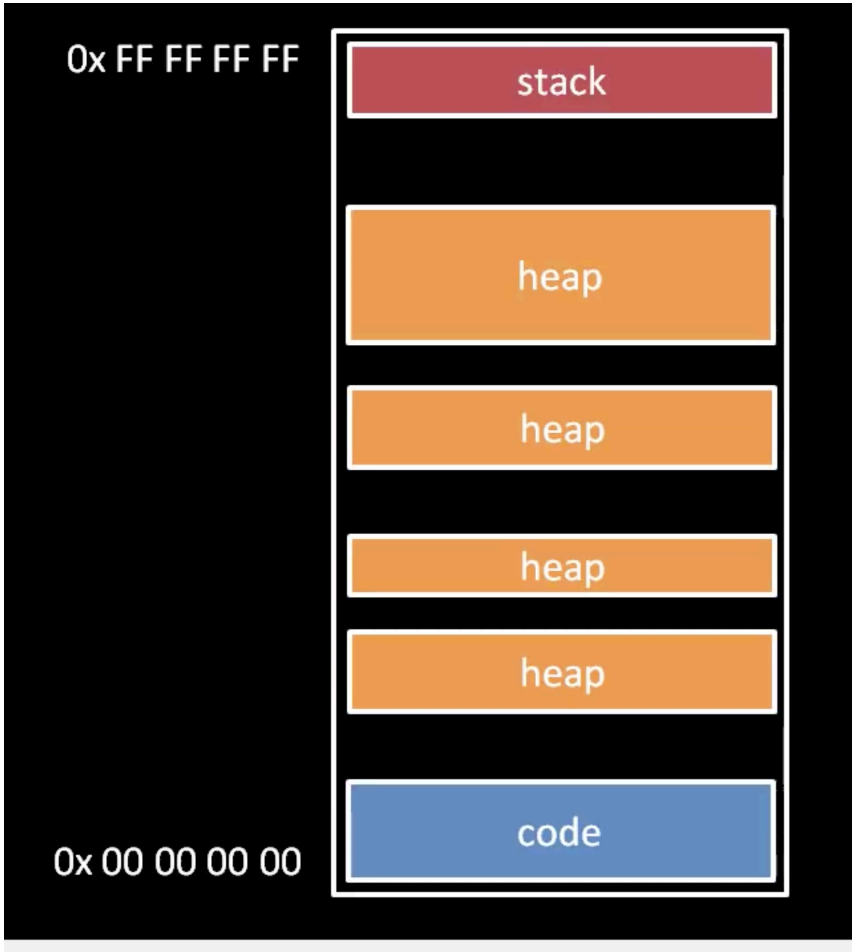


Heap management 1/2

- Memory space between the stack and the code of the process

1. **Process** requests allocation of heap space from the OS
2. **OS** allocates a piece of adjacent space
3. **de-allocation** is the responsibility of the **process**

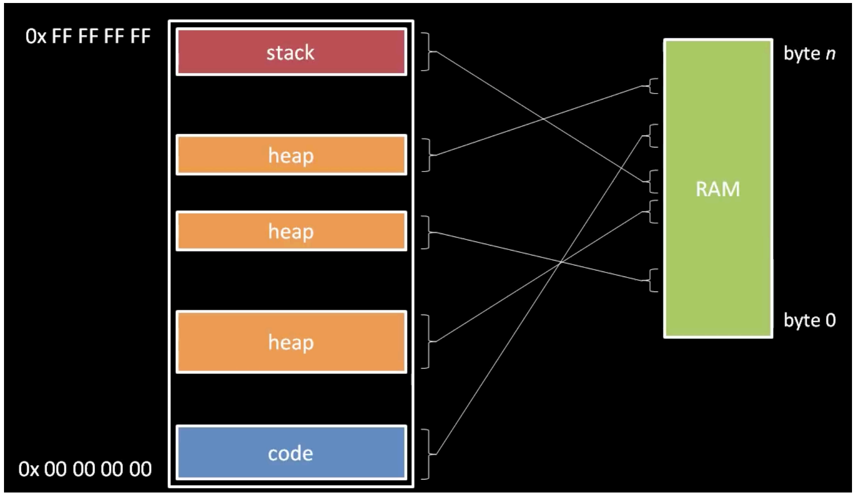
- Problems
  - Run out of heap space
  - Fragmentation of the heap space



Heap management 2/2

- The memory address of a process do not refer to actual byte in the system memory (RAM)

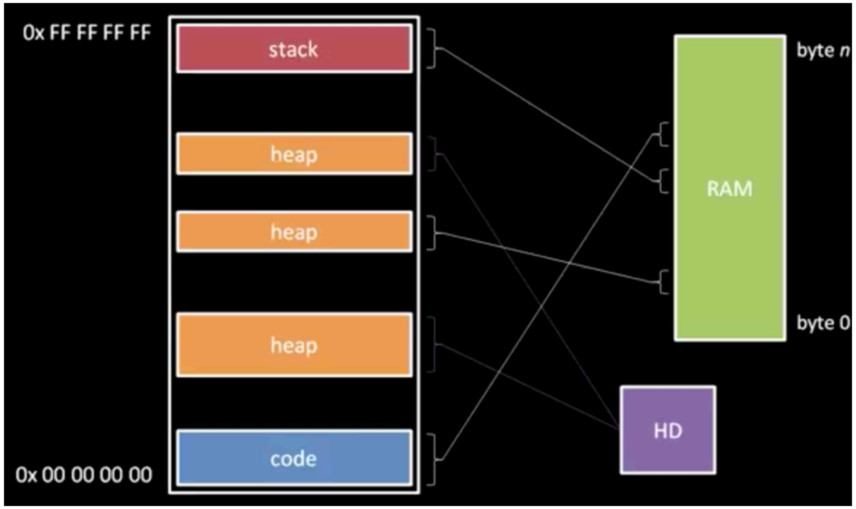
- Instead chunks of the process address space are mapped by the OS to chunks of the byte in the system memory
  - Not contiguous
  - Or in a particular order



Heap management: SWAP

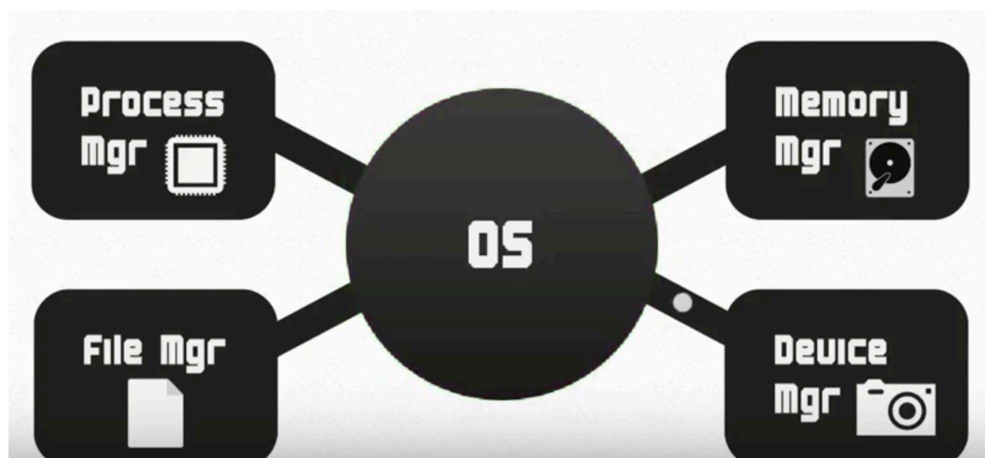
- To free up valuable system memory (RAM), the OS may decide to **swap out pages** of a process to a Hard drive

- Heap data that are not mapped to any part of the RAM and are marked the **process memory table** as **SWAPPED**
- If a process tries to access SWAPPED pages an exception is generated
  1. Copy back data to RAM
  2. Adjust the process mem table



content

- Introduction to Operating systems (OS)
- Process management
  - Single process management
  - Multi process management
- Memory management
- File system

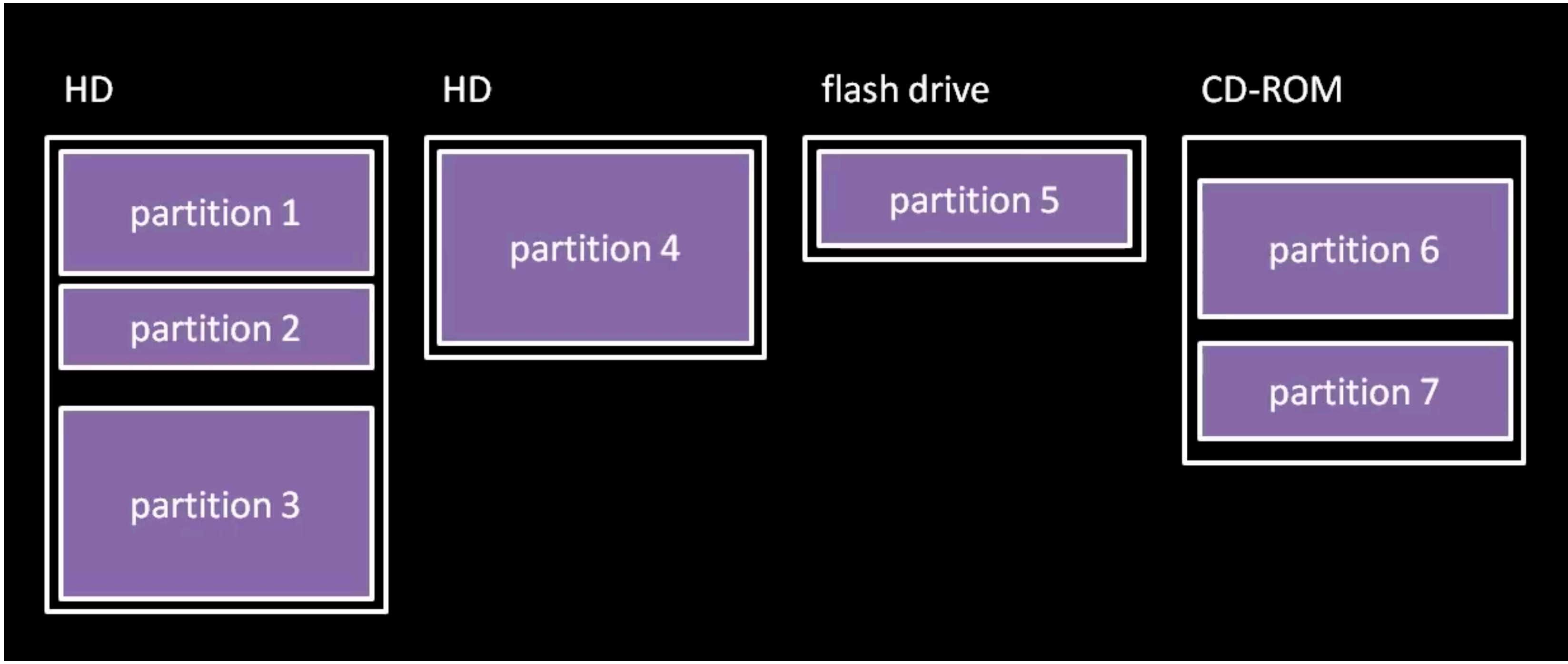


File systems 1/2

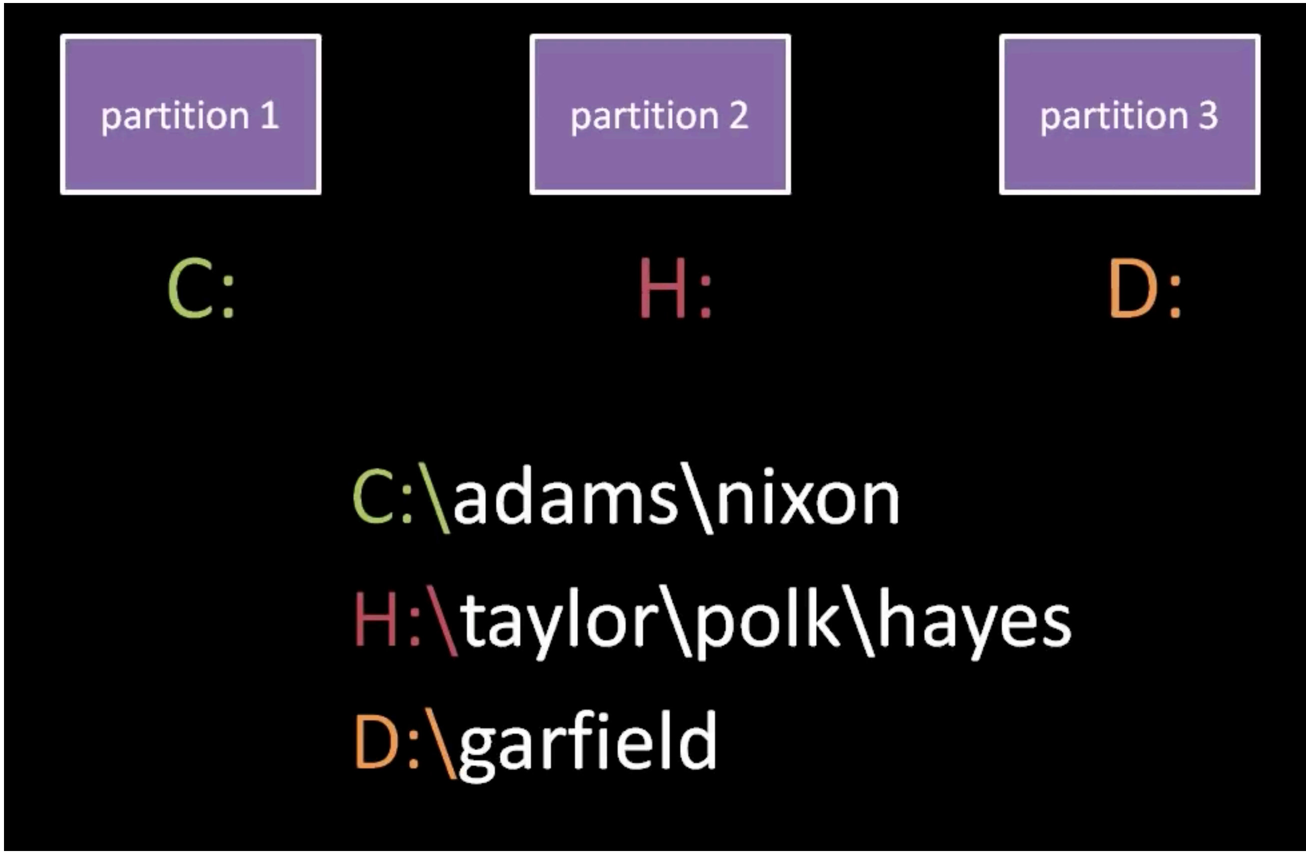
- OS provide an useful **abstraction** of the **storage system** known as file system
- File systems presents storage devices **as hierarchy of directories** and files stored in these directories
- Processes can read/write data from/to **logically contiguous** memory space called file, which can be accessed by a logical name

Partitions

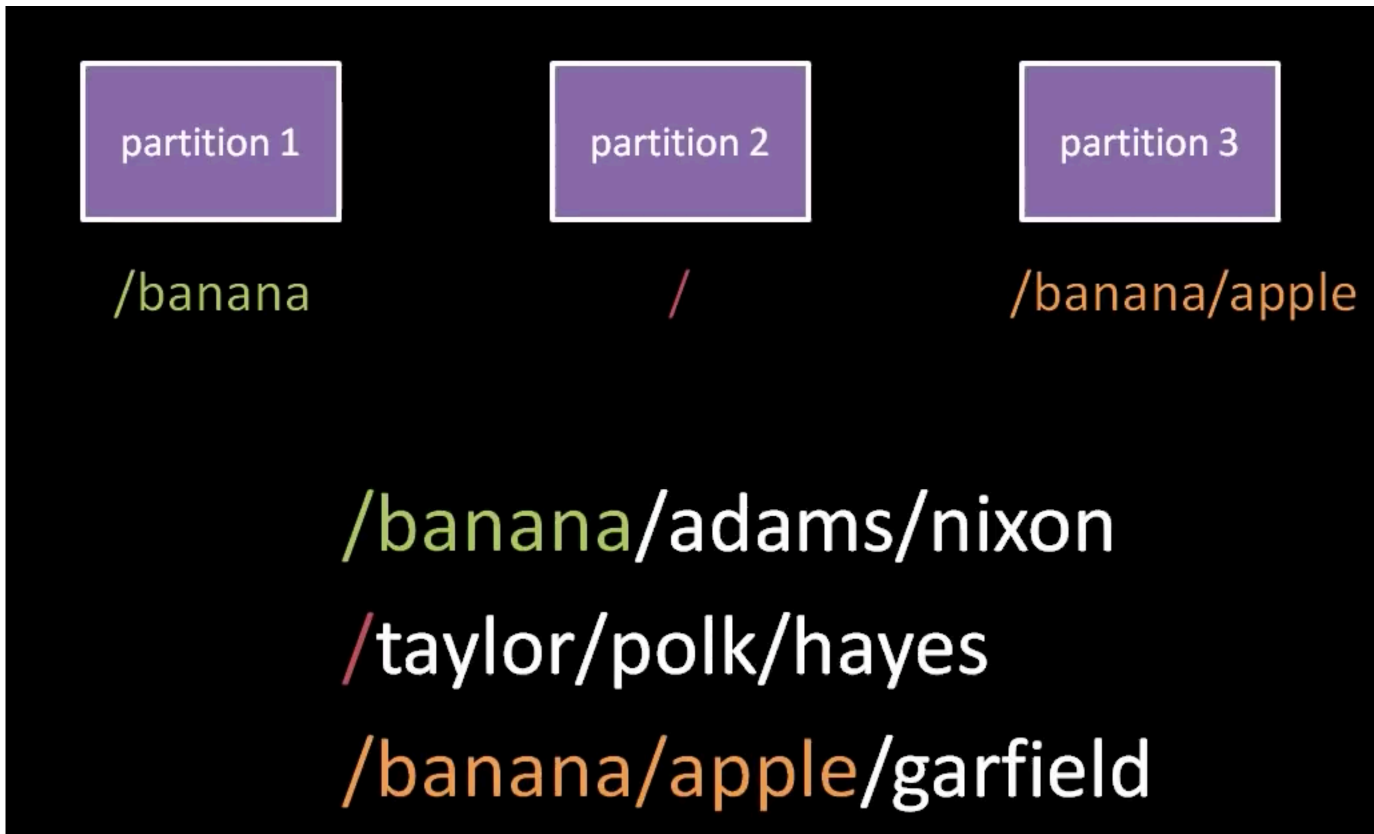
- Storage devices are composed of one or more partitions



Windows Partitions

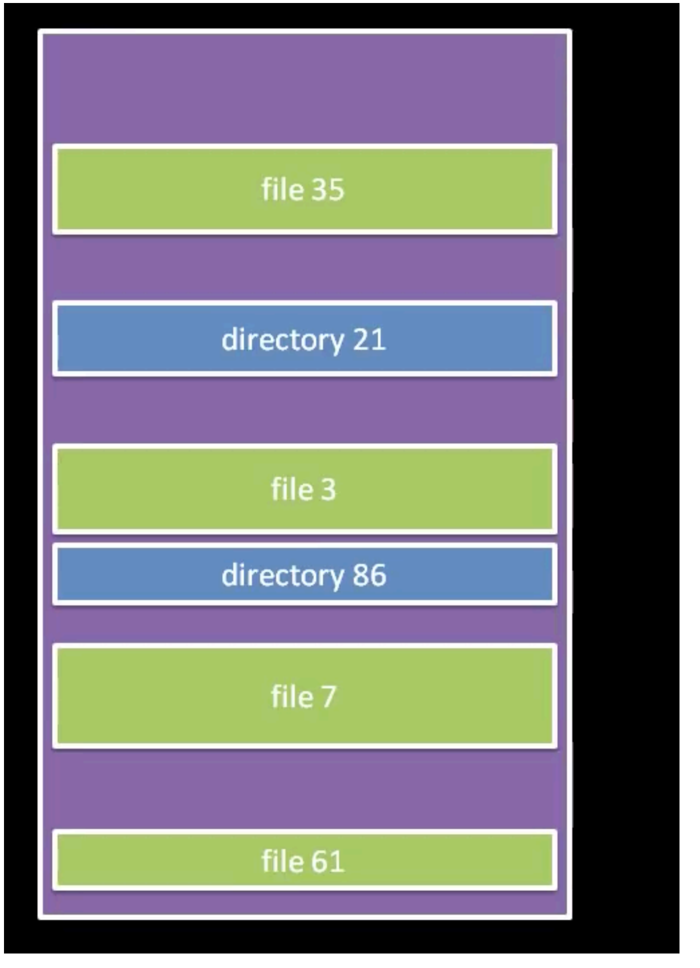


Unix-like Partitions



Partitions

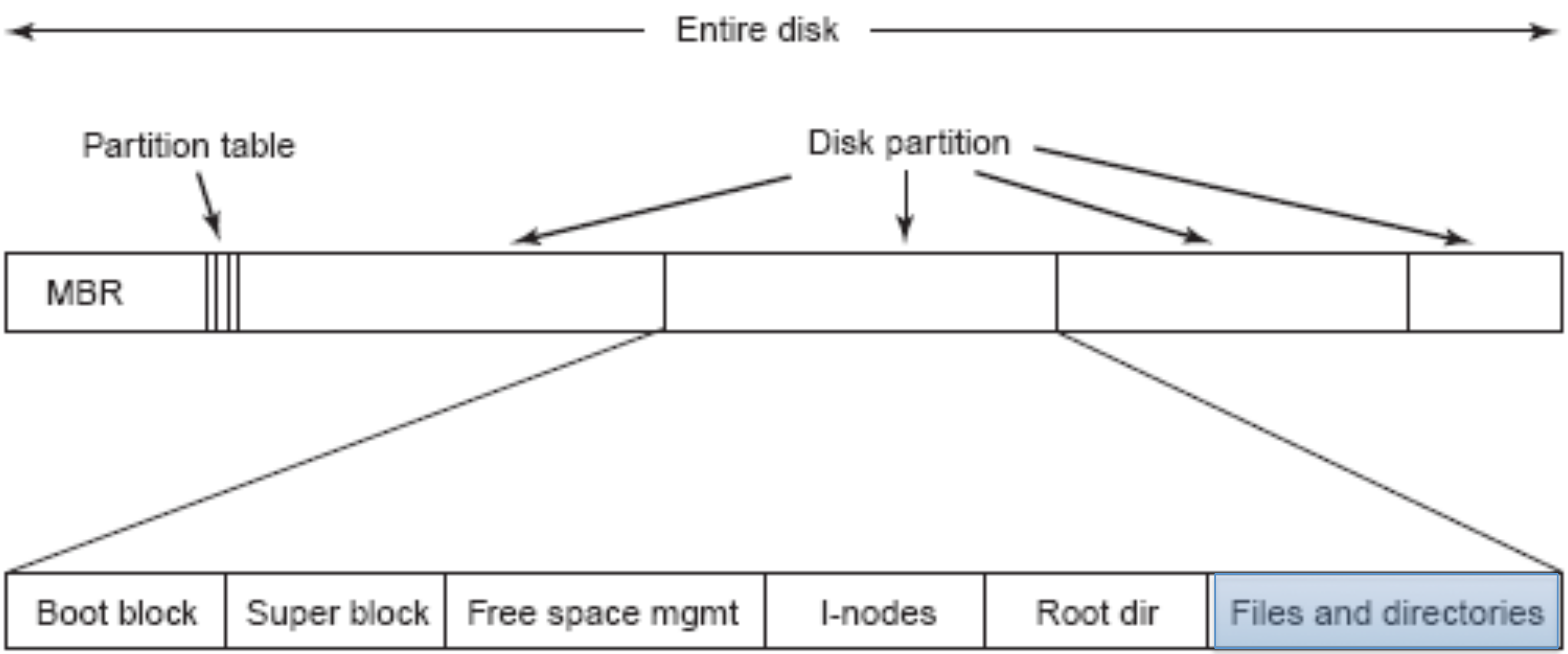
- Each partition is organized as a **hierarchy of** files and directories
- Each file and directory is known by a **unique identifier**
  - File Attributes



File system design

- File storage
- Directory implementation
- Disk space management
- Consistency

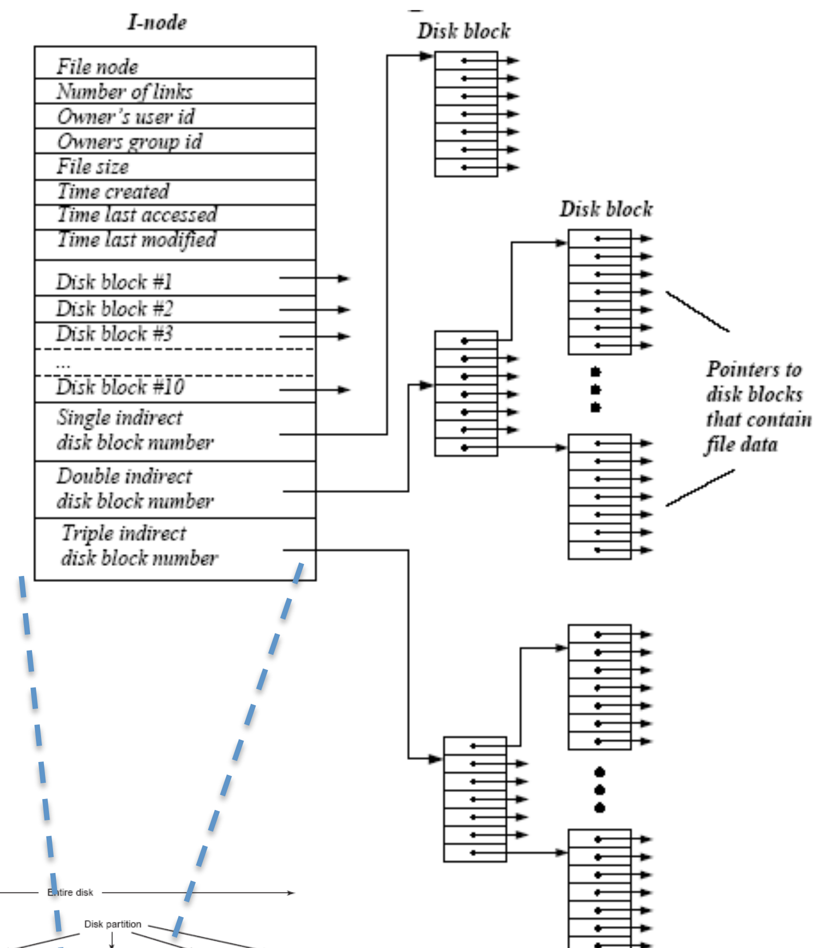
File storage: Disk Layout

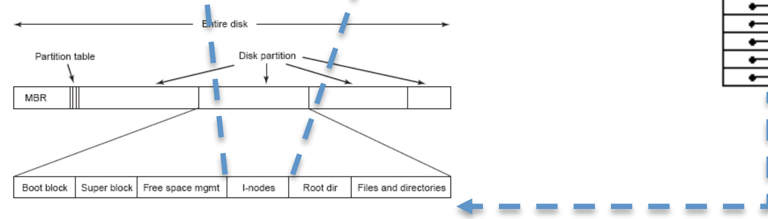


File storage - Inode

inode is a data structure used to represent **a filesystem object**, which can be one of various things including a **file** or a **directory**.

Each inode stores the attributes and disk block location(s) of the filesystem object's data





File storage –name resolution