Lecture 10

Operating System

Overview

Operating System
- Motivation
- Responsibilities

Three Main Functionalities:
- Running Process
- Handling Memory
- Providing File / Folder Abstractions

Wizard of os

Let me show you magic……

Modern OS: Overview

PC

Windows 10

macOS 10.13

Ubuntu 16.04 LTS

Mobile

iOS

android

Real-Time

freeRTOS

Embedded

Raspbian

Operating System: **Resource Drain**?

Operating System is a **program**

➔ Take up resources on a computer system

Example: **Windows 10**

- **Memory**: 1GB for 32bit; 2GB for 64bit
- **Hard disk**:16GB for 32bit; 20GB for 64bit
- **CPU**: Take away CPU from other program when it is running

Motivation for OS: **Abstraction**

Large variation in hardware configurations

Example (**Hard disk**):

- Different capacity
- Different capabilities

However, hardware in the same category has **well defined and common functionality**

Motivation for OS: Abstraction

Operating System serves as an abstraction:

– Hide the different low level details
– Present the common high level functionality to user

The user can then perform essential tasks through operating system

– no need to concern with low level details

Provides:

– Efficiency and portability

Motivation for OS: Resource Allocator

Program execution requires many resources:

- CPU, memory, I/O devices etc

Multiple programs should be allowed to execute simultaneously

OS is a resource allocator

- Manages all resources
- Arbitrate potentially conflicting requests
  - for efficient and fair resource use

Motivation for OS: Control Program

Program can misuse the computer:
- Accidentally: due to coding bugs
- Maliciously: virus, malware etc

Multiple users can share the computer:
- Tricky to ensure separate user space

OS is a control program
- Controls execution of programs
  - Prevent errors and improper use of the computer
  - Provides security and protection

Motivation for OS: **Summary**

Abstraction

- Hide low level details
- Provide simple interface

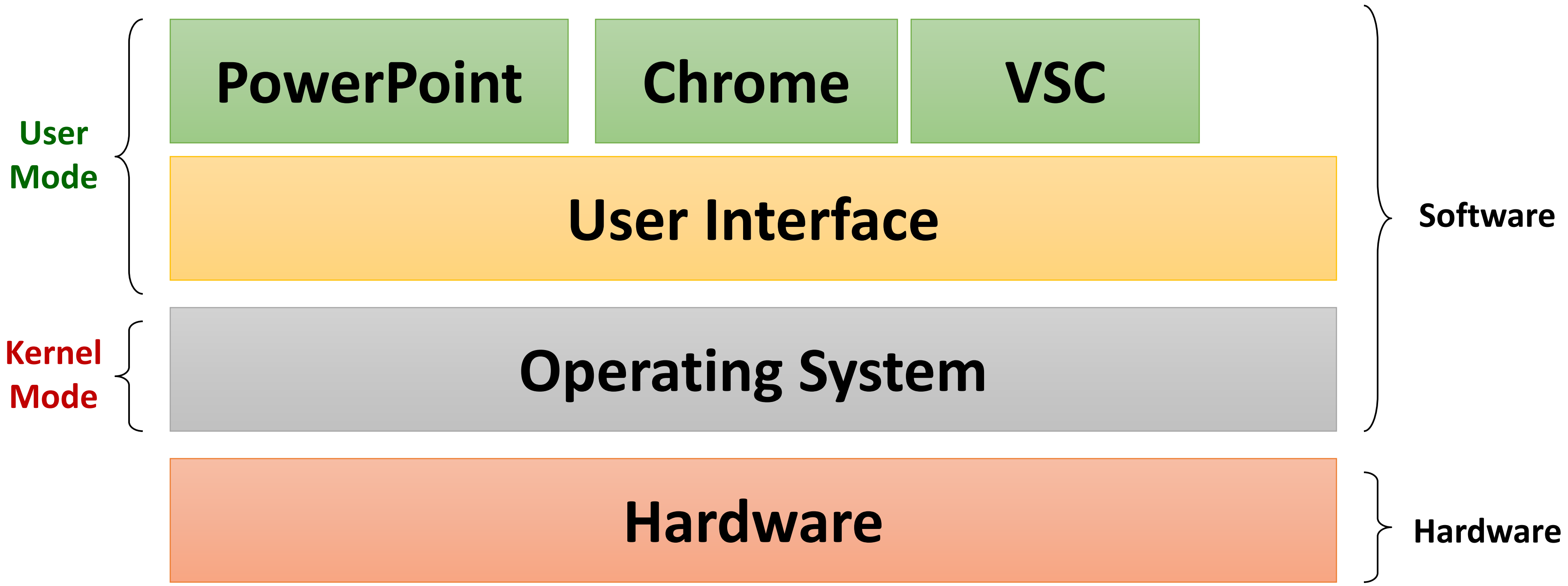
Resource Allocation

- Manage hardware resources
- Arbitrate conflicting requests

Control

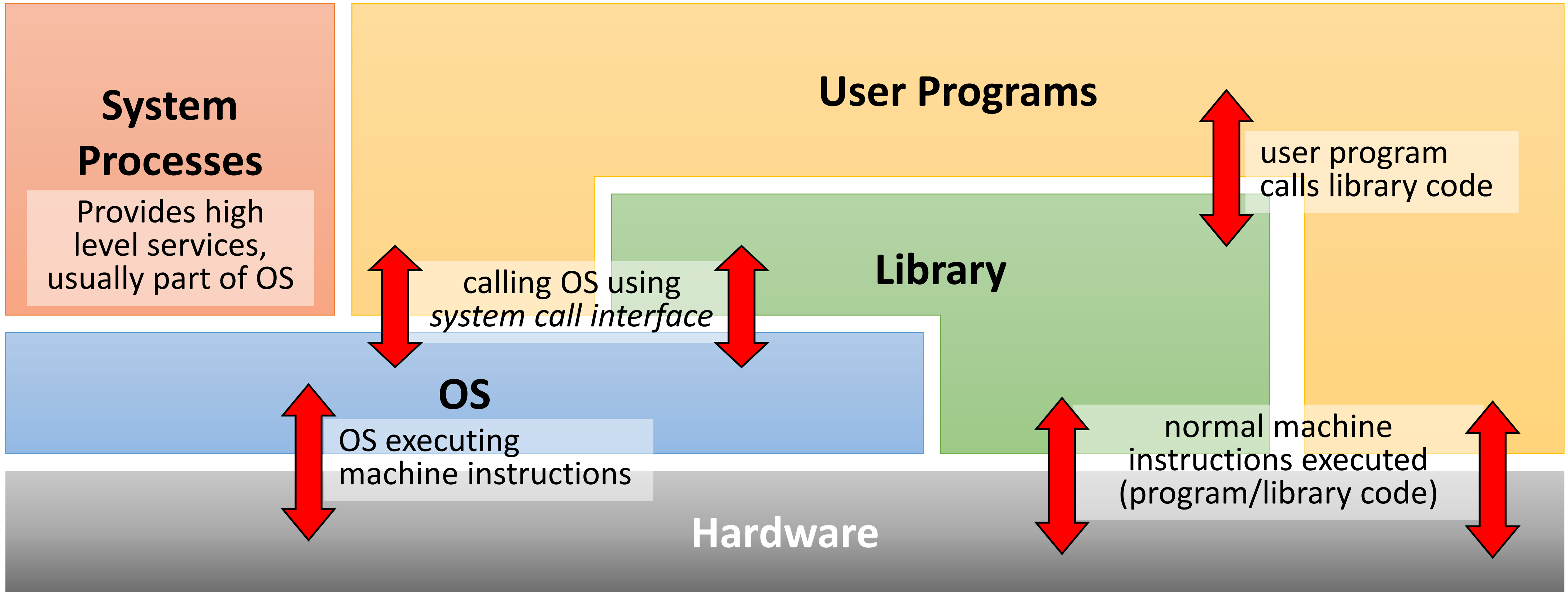
- Prevent errors and improper use
- Security and protection

High Level View of OS



- Operating System is essentially a software
  - Runs in kernel mode: Have complete access to all hardware resources
- Other software executes in user mode
  - With limited (or controlled) access to hardware resources

Interaction between Components



System Calls

Application Program Interface (API) to OS

- Provides way of calling facilities/services in kernel
- **NOT** the same as normal function call
  - have to change from user mode to kernel mode

Different OS have different APIs:

- Unix Variants:
  - Most follows POSIX standards
  - Small number of calls: ~100
- Windows Family:
  - Uses Win API across different Windows versions
  - New version of windows usually adds more calls
  - Huge number of calls:~1000

Unix System Calls in C/C++ program

In C/C++ program, system call can be invoked *almost directly*

– Majority of the system calls have a library version with the **same name** and the same parameters
  - The library version act as a function wrapper

– Other than that, a few library functions present a more user friendly version to the programmer
  - E.g. lesser number of parameters, more flexible parameter values etc
  - The library version acts as a function adapter

System Calls: Example

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
        printf("Hello Again!\n");

        exit(0); //same effect
                 // as "return 0;" in main()
}
```

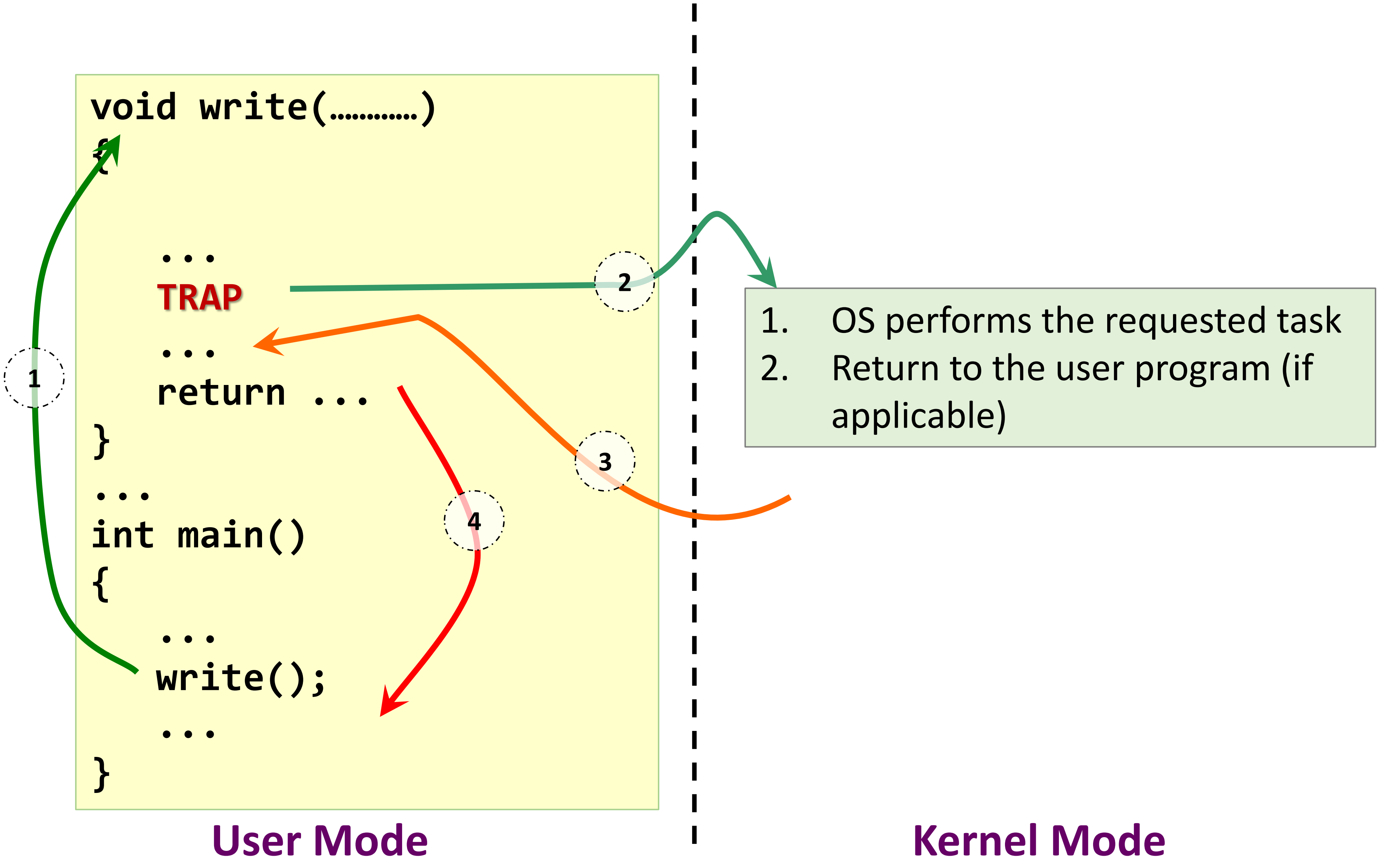
Library call that make a **system call**

Library call that has the same name as a **system call**

System Calls invoked in this example:

- **`write()`** – made by **`printf()`** library call
- **`exit()`**

System Calls: **What happen?**



Simplified System Call Mechanism

1. [User Mode] User program invokes the library call
2. [User Mode] Library call executes a special instruction to switch from user mode to kernel mode
   - That instruction is commonly known as TRAP
3. [Kernel Mode] OS now takes over:
   - Carry out the actual request
   - Switch from kernel mode to user mode
4. [User Mode] Library call return to the user program

3 “Magic tricks”

3 Main Functionalities of OS

Three Major Functionalities of OS

Process Management

- Allow multiple programs to run "together"

Memory Management

- Allow memory to be shared
- Allow hard disk to be used as extension of RAM

File Management

- Provide Files / Folders
- Allow hard disk to be used efficiently

Process Management: **Problem**

Process:

- A running executable AND
- Its “environment”

Process uses:

- CPU
- Memory
- I/O devices

There are **many** processes “running” on the system at the same time

- How to “share” the Processor?

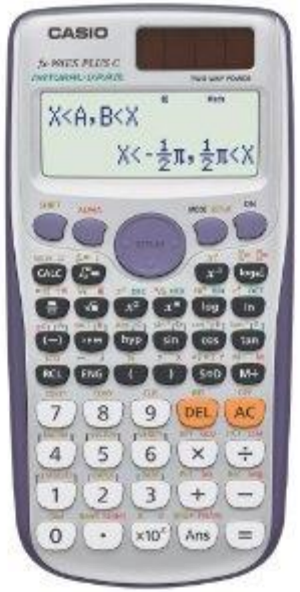
Process Problem: Analogy

Imagine each of you have a list of expressions to calculate, but there is only one **calculator**

Need to Calculate

a. 1 + 2 + 3 + …+100
b. 2 * 4 * 8 …. *128
c. $5^2 + 7^2 + \ldots.$
d. cos(49.5) + sin(72.8)*pi + …
e. ………….

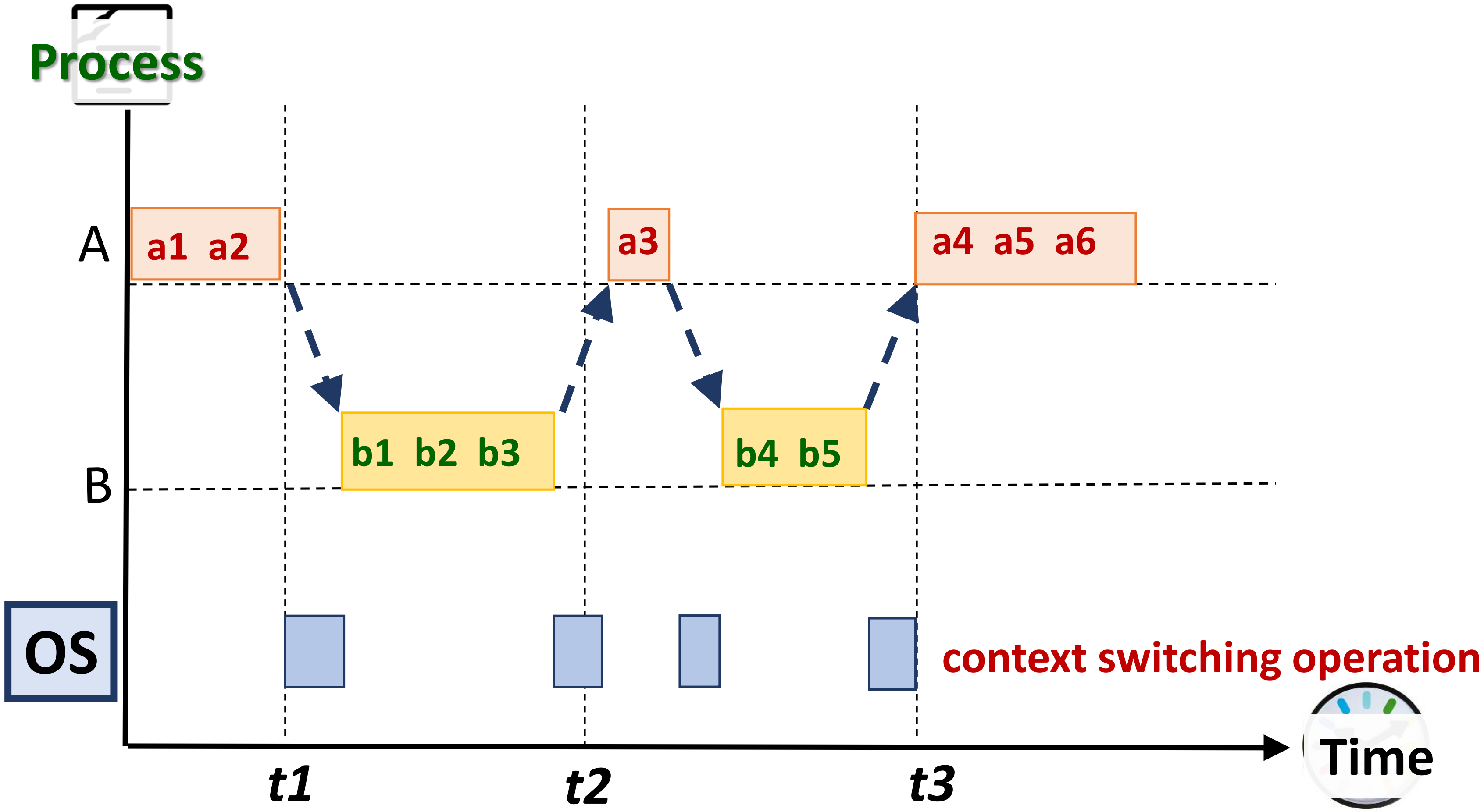
Program



CPU

I/O Device

Interleaved Execution: **Solution**



Multitasking needs to change context between A and B:

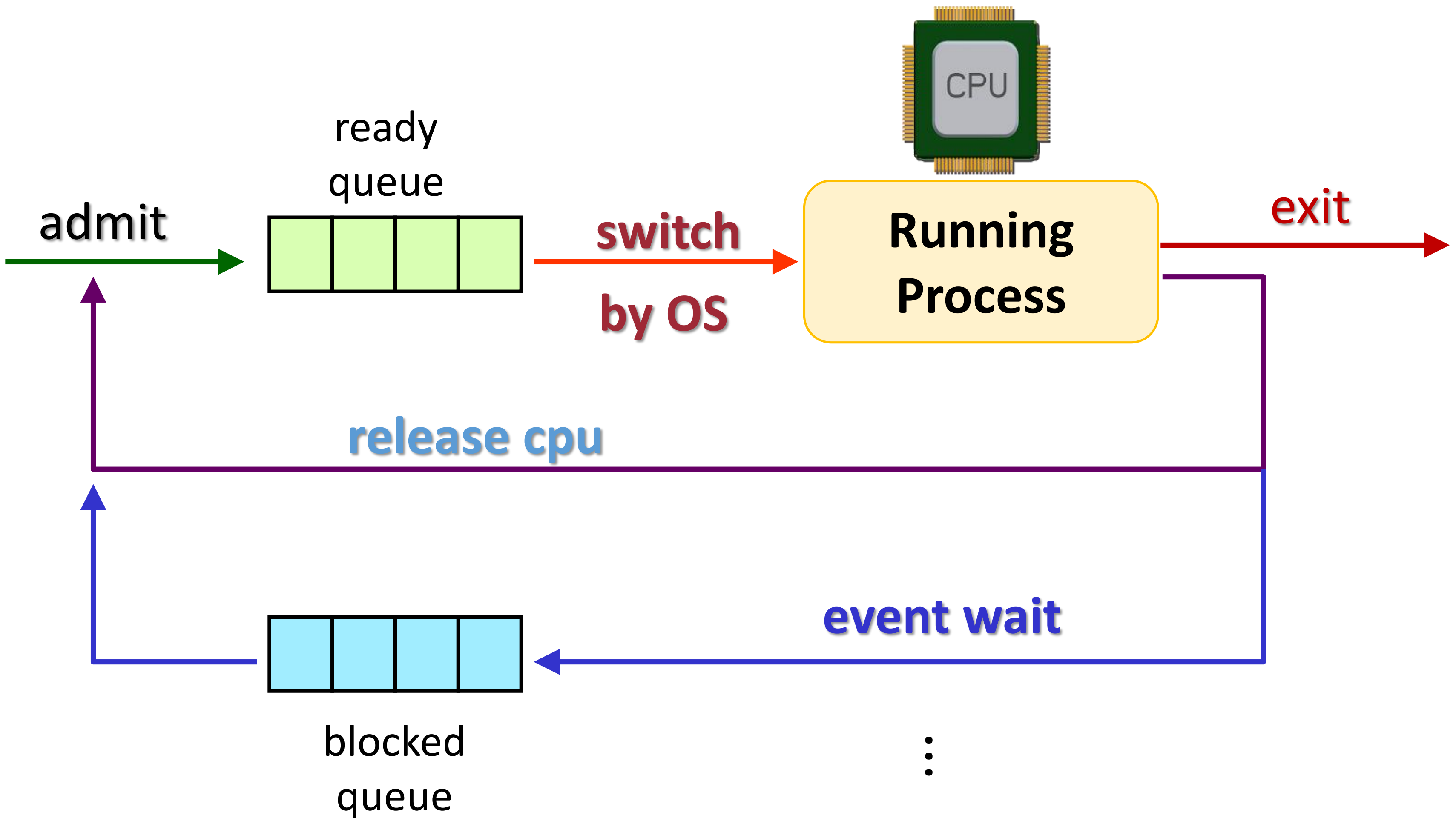
– OS incurs overhead in switching processes

Process Management

All ready processes are in a **Ready Queue**

1. [OS] pick one of the process **P**
2. [OS] setup the environment for process **P**
3. [OS] pass the processor for **P** to run and give a time limit **T**
4. [P] runs for the time limit **T**
5. [OS] pack up the environment of P
6. [OS] go to Step 1 and choose another process

[OOS] Process Management: OS View



[OOS] For your exploration

How to choose the process?

- Round Robin
- Priority
- Shortest job first
- Many others

How does the OS "wakes up"?

- Timer Interrupts
- Time Quantum
- Scheduler execution

Memory Management

Memory Management: **Problem**

A process thinks it "owns" the whole memory space

Simple questions:

- If process P has an item stored at memory location 1024, does other processes have memory location 1024 too?
- What if we run process P twice and let both copy of P running at the same time? Does both Ps have memory location 1024?

Paging Scheme: Basic Idea

The **physical memory** is split into regions of fixed size (decided by hardware)

- known as **physical frame**

The **logical memory** of a process is similarly split into regions of ***same size***

- known as **logical page**

At execution time, the pages of a process are loaded into **any available** memory frame

➔Logical memory space remain contiguous

➔Occupied physical memory region can be disjoined!

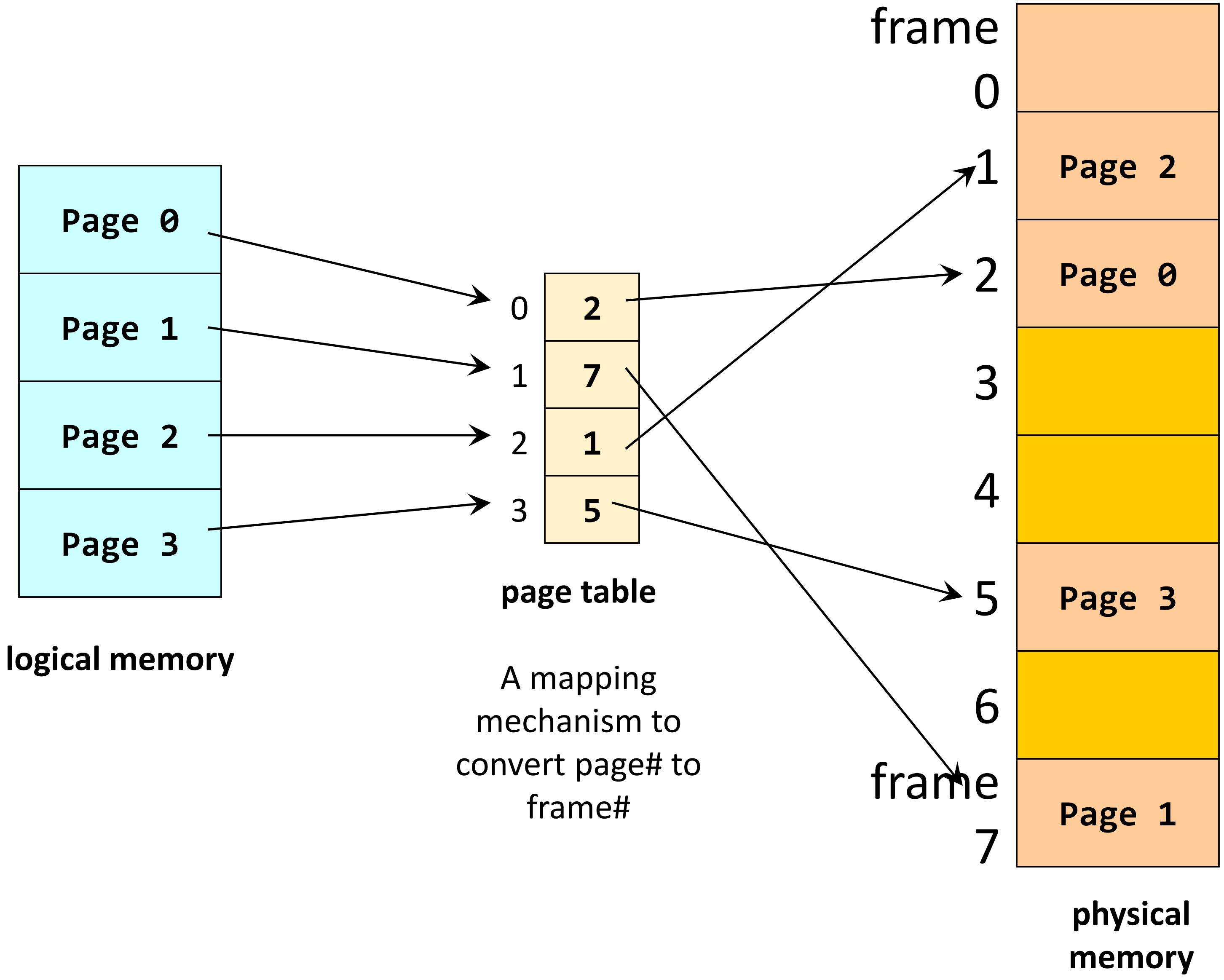
Page Table: Lookup Mechanism

Under paging scheme:

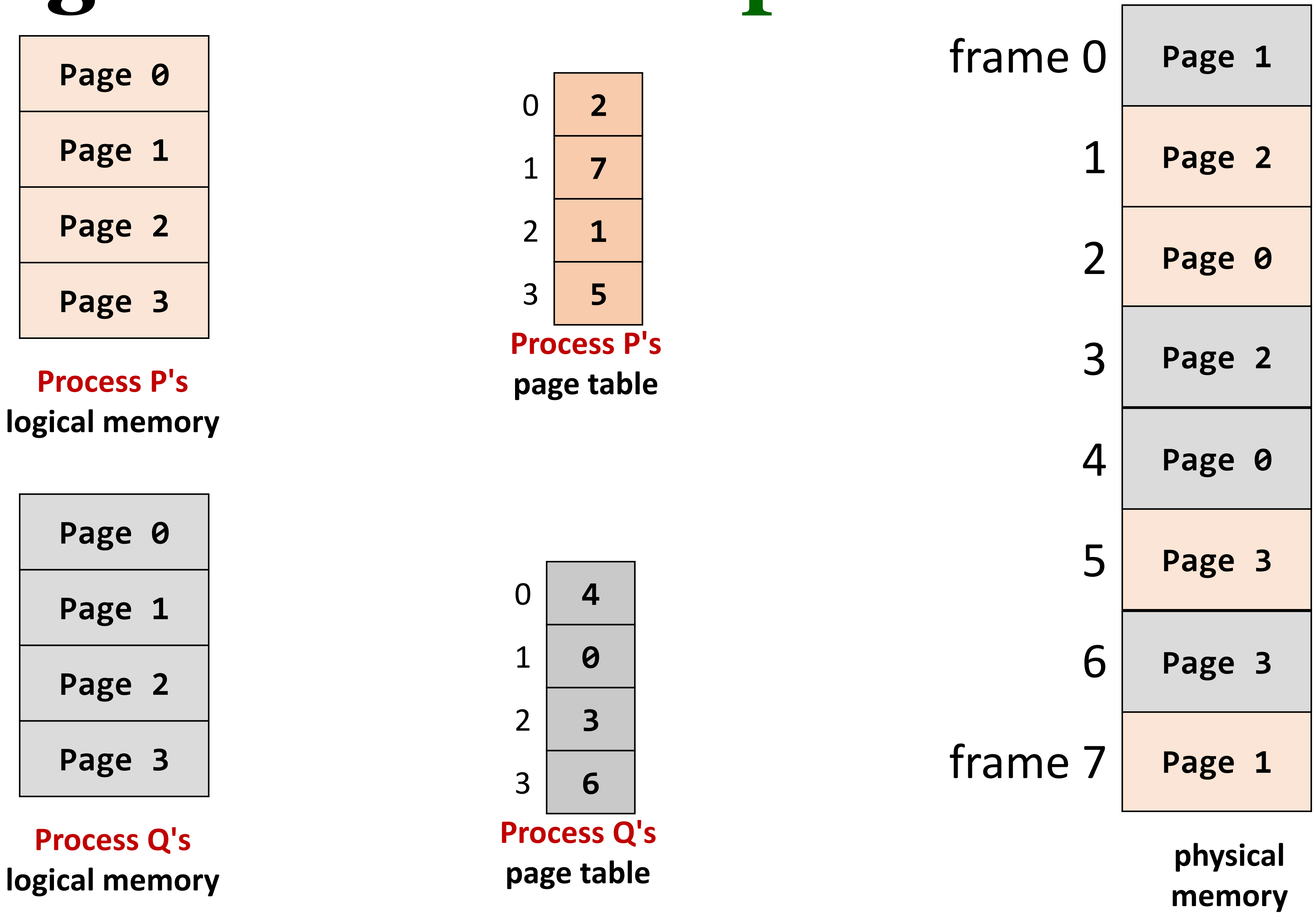
- Logical page ←→ Physical frame mapping is not straightforward
- Need a lookup table to provide the translation
- This structure is known as a Page Table

During execution, Page Table is consulted to find out the "real" (physical memory address) of an item

Paging: Illustration



Paging Scheme: Multiple Processes



Virtual Memory: Motivation

Consider the following facts:

1. There are a large number of processes running at any point in time (typical 100+ on Windows)

2. Total memory usage for these processes is huge (100 processes x 100Mb each = 10,000Mb = 10Gb)

3. A process only needs a small portion of its memory space at any point in time (locality)

Virtual Memory: Basic Idea

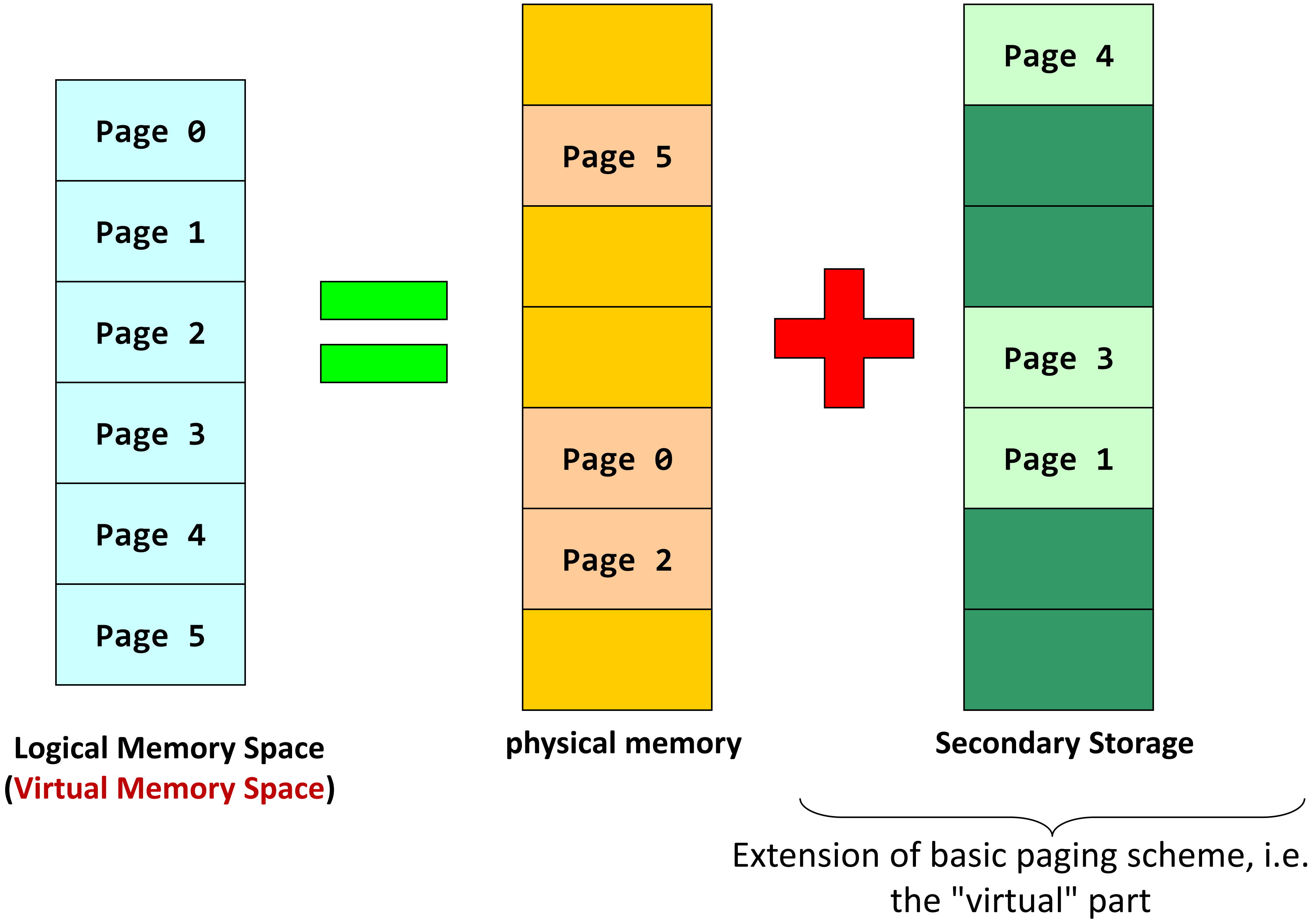
Observation:

- Secondary storage has much larger capacity compared to physical memory

Basic Idea:

- Extension of the paging scheme:
  - Logical memory space split into fixed size page
  - Some pages may be in physical memory
  - Other pages can be stored in secondary storage

Virtual Memory: Paging Illustration



Extended Paging Scheme

Basic idea remains unchanged:

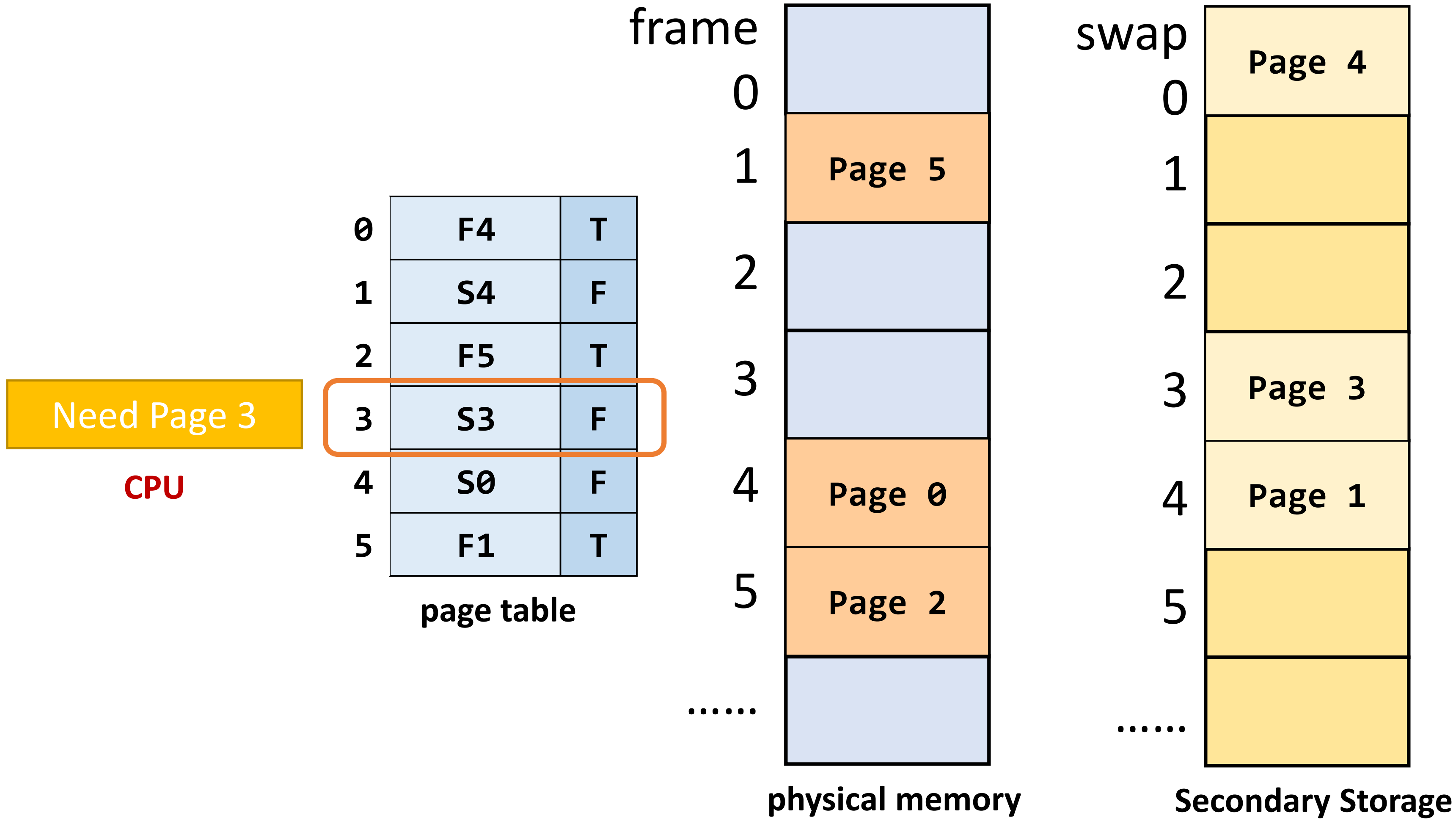
- Use page table to translate **virtual** address to physical address

New addition:

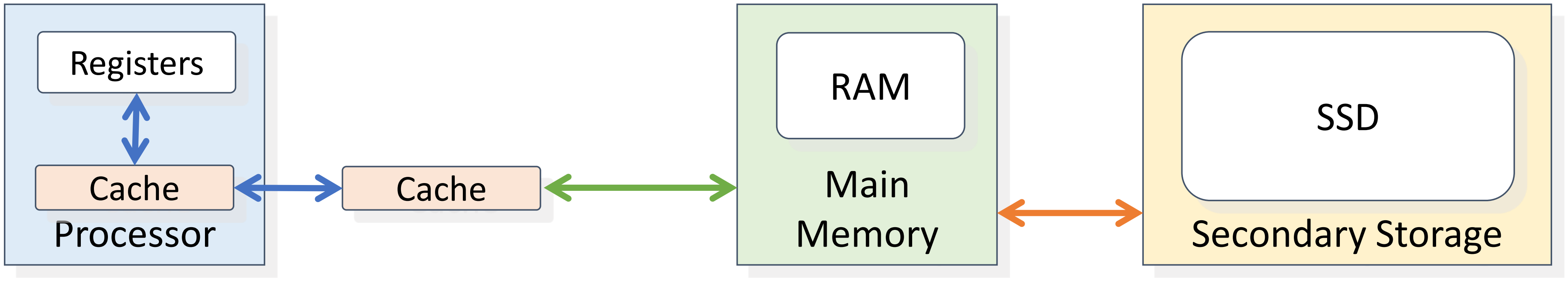
- To distinguish between two page types
  - **memory resident** (pages in physical memory)
  - **non-memory resident** (pages in secondary storage)

- CPU can only access memory resident pages:
  - **Page Fault**: When CPU tries to access non-memory resident page
  - **OS** need to bring a non-memory resident page into physical memory

Virtual Memory in Action: Illustration



Recap and Summary: Memory Hierarchy



make SLOW main memory appear **faster**?

- **Cache**: a small but fast SRAM near the CPU
- Hardware managed: Transparent to programmer

make SMALL main memory appear **bigger**?

- **Virtual memory**
- OS managed: Transparent to programmer

FILE Management

File Management: **Problems**

1. What are the major abstraction provided?
   - Files and Folders

2. How can we support files / folders?
   - Created often
     - Some files are HUGE, some are tiny
   - Modified often
     - Size changes (can shrink or grow)
   - Deleted often
     - E.g. backup files created when opening words, excels etc

File: Basic Description

Represent a logical unit of **persistent** information

An ***abstraction***

- A set of common operations "wraps" around a set of data
- Various possible implementations

Contains:

- Data: Information structured in some ways
- Metadata: Additional information associated with the file
  - Also known as file attributes

Common File Metadata

Name:	A human readable reference to the file
Identifier:	A unique id for the file used internally by FS
Type:	Indicate different type of files E.g. executable, text file, object file, directory etc
Size:	Current size of file (in bytes, words or blocks)
Protection:	Access permissions, can be classified as reading, writing and execution rights
Time, date and owner information:	Creation, last modification time, owner id etc
Table of content:	Information for the FS to determine how to access the file

File Protection: How?

Most common approach:

- Restrict access base on the user identity
- Usually implemented as **permission bits**

Most general scheme:

- Access Control List
  - A list of user identity and the allowed access types
  - Pros: Very customizable
  - Cons: Additional information associated with file

File Protection: **Permission Bits**

Classified the users into three classes:

1. **Owner**: The user who created the file
2. **Group**: A set of users who need similar access to a file
3. **Universe**: All other users in the system

Example (Linux)

– Define permission of three access types (**R**ead/**W**rite/E**x**ecute) for the 3 classes of users

```
rwxr--r--   somefile.txt
```

Owner **Group** **Universe**

Folder: Basics

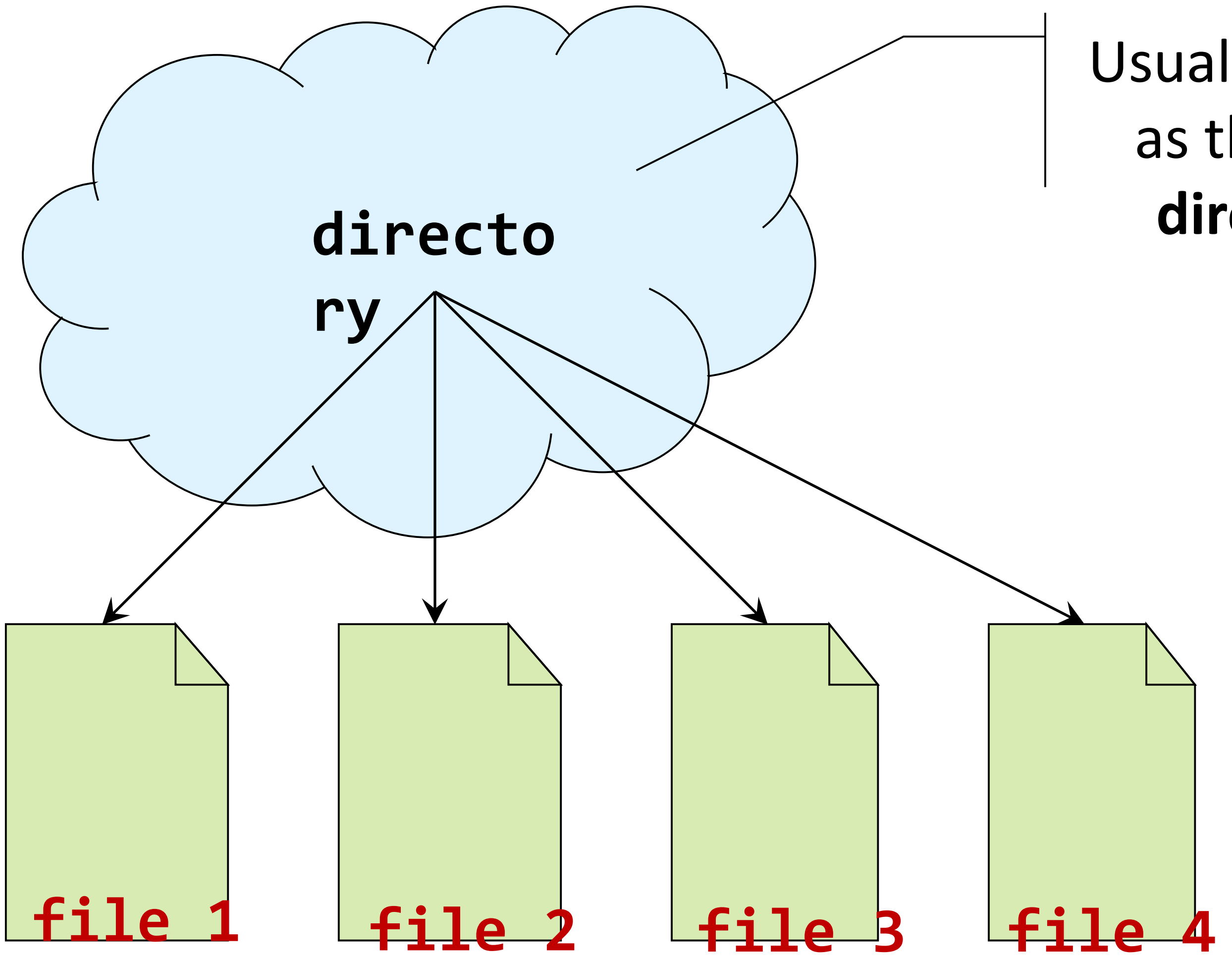
Folder (directory) is used to:

1. Provide a logical grouping of files
    - The user view of folder
2. Keep track of files
    - The actual system usage of folder

Several ways to structure folder:

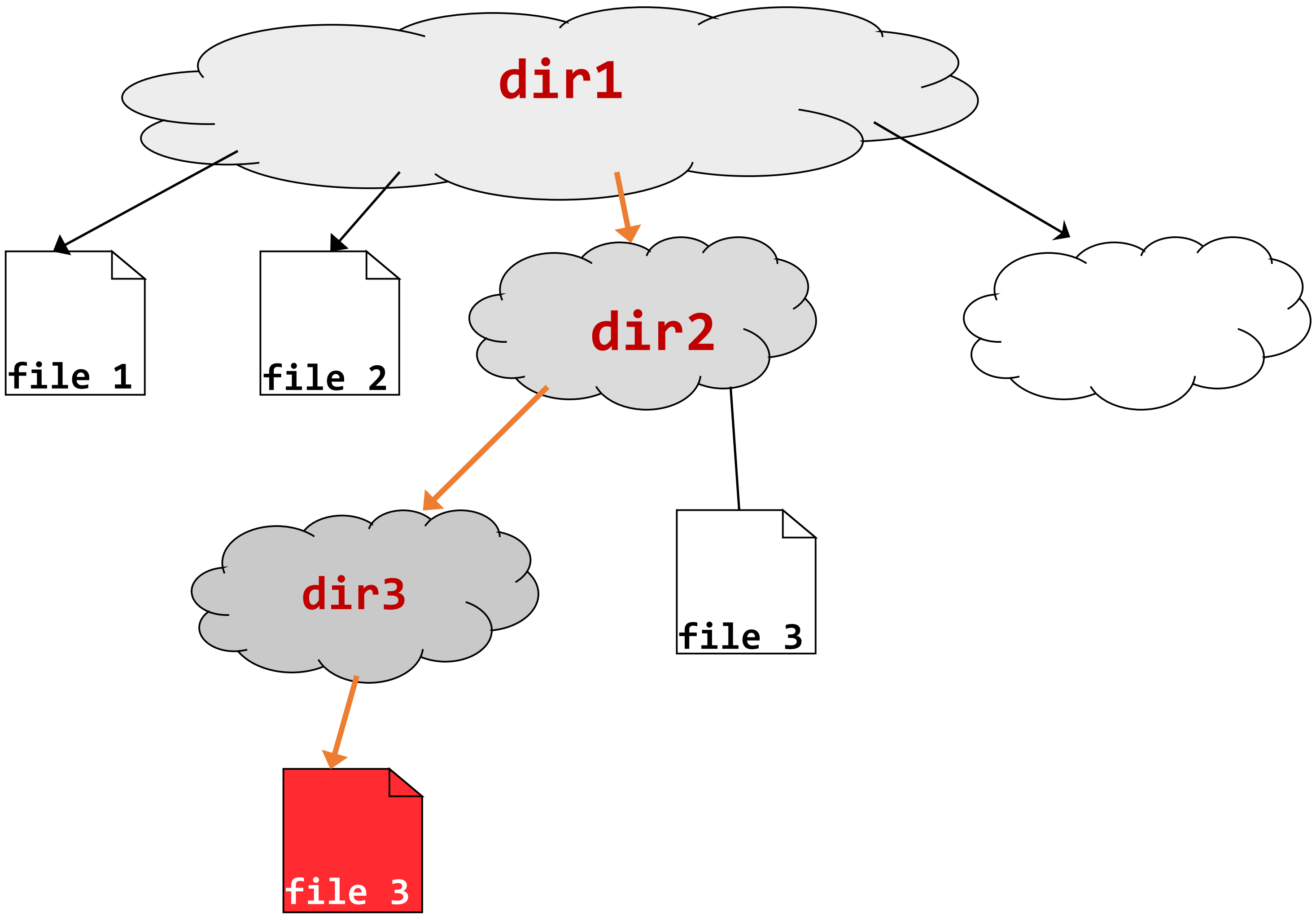
- Single-Level
- Tree-Structure
- Directed Acyclic Graph (DAG)
- General Graph

Directory Structure: **Single-Level**

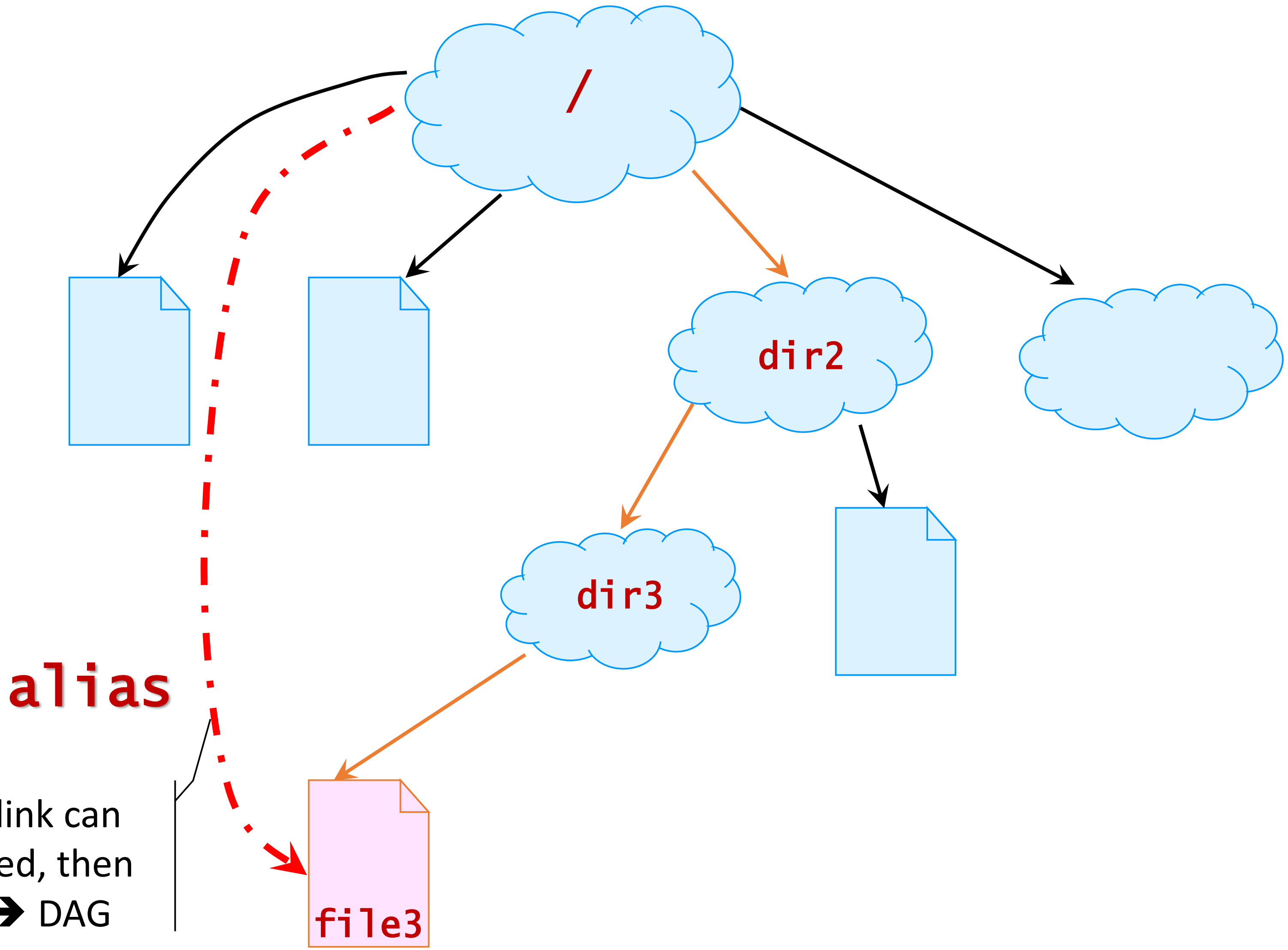


Usually known as the **root directory**

Directory Structure: **Tree-Structured**



Directory Structure: DAG



If this link can be added, then Tree ➔ DAG

Directory Structure: DAG

If a file *can be shared:*

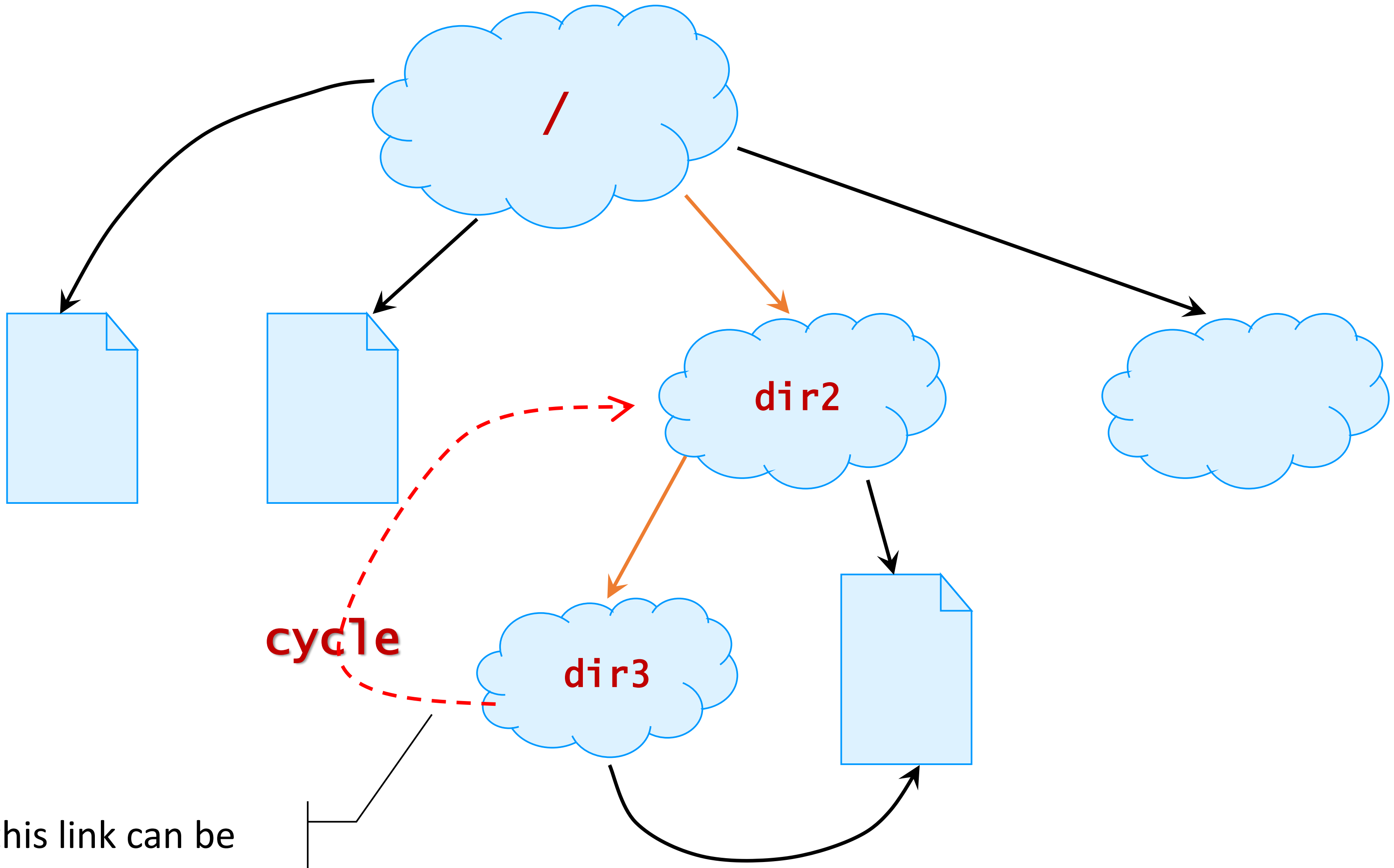
- Only one copy of actual content
- "Appears" in multiple directories
  - With different path names

Then tree structure → DAG

Possible in Windows / Unix:

- Symbolic Link
  - Can be file or directory
  - This has an "interesting" effect….

Directory Structure: **General Graph**



If this link can be added, then Tree ➔ General Graph

Directory Structure: General Graph

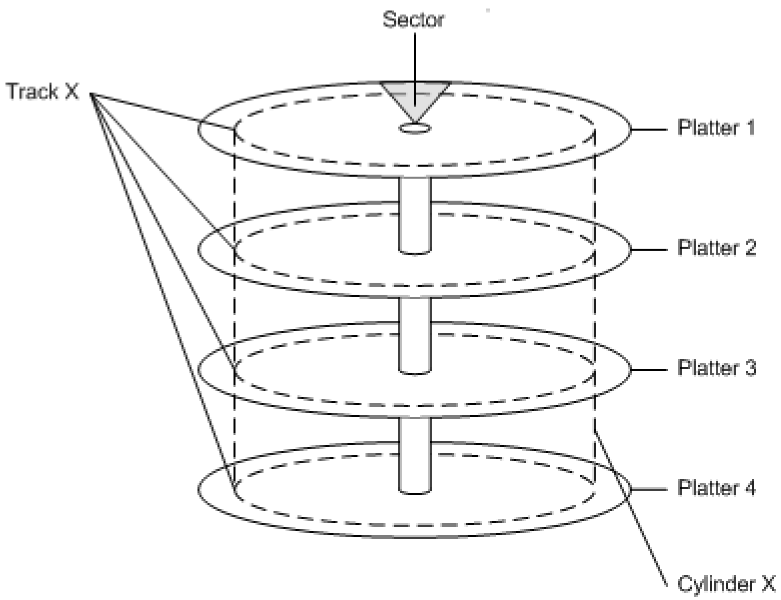
General Graph Directory Structure is ***not desirable:***

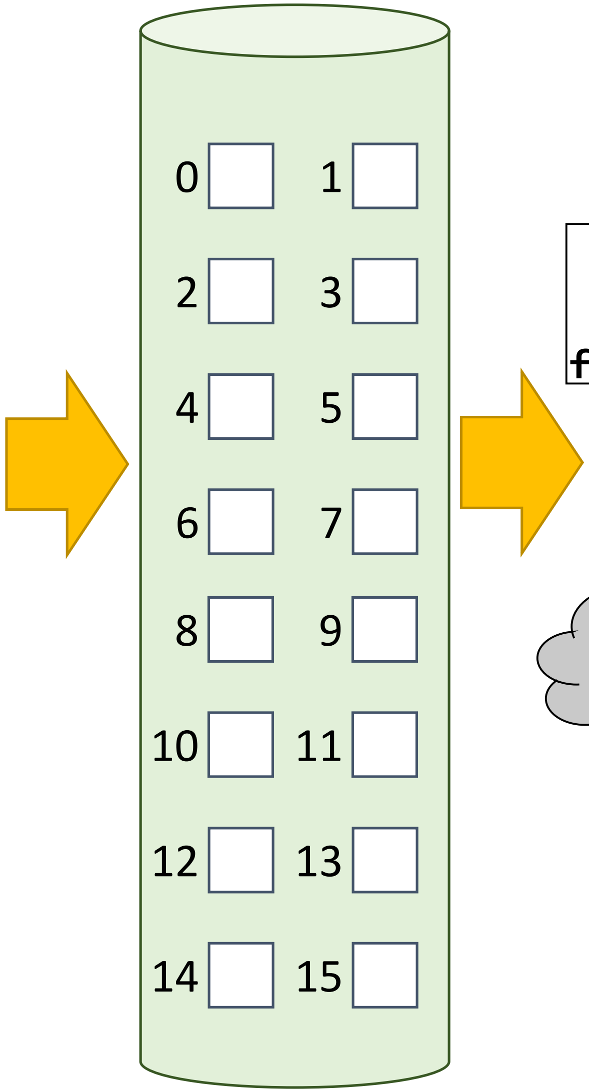
- Hard to traverse
  - Need to prevent infinite looping
- Hard to determine when to remove a file/directory

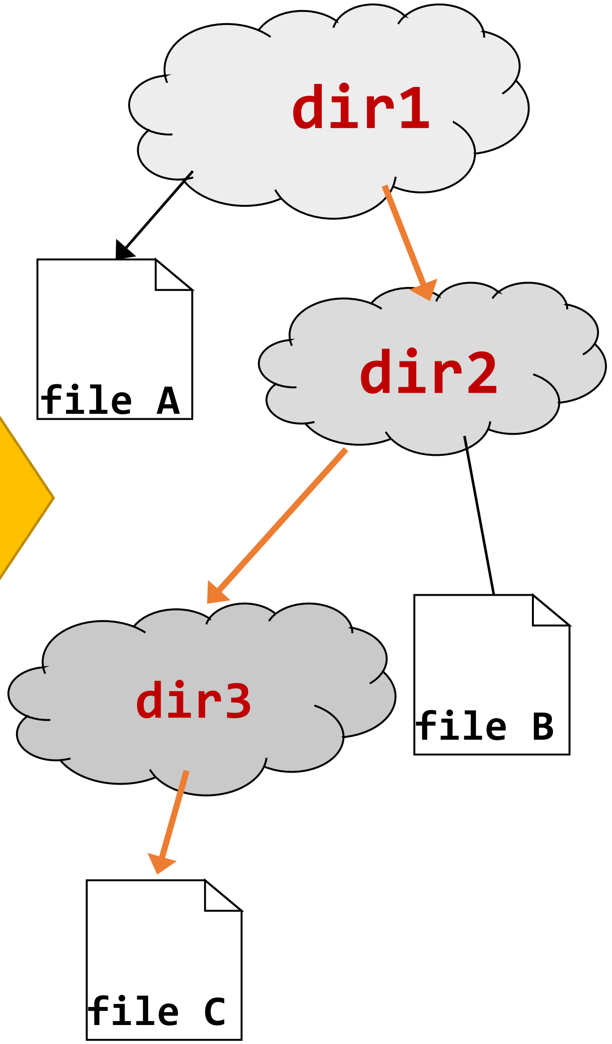
In Windows / Unix:

- Symbolic link is allowed to link to directory
  - General Graph **can be created**

Hardware←→OS ←→ User View







END