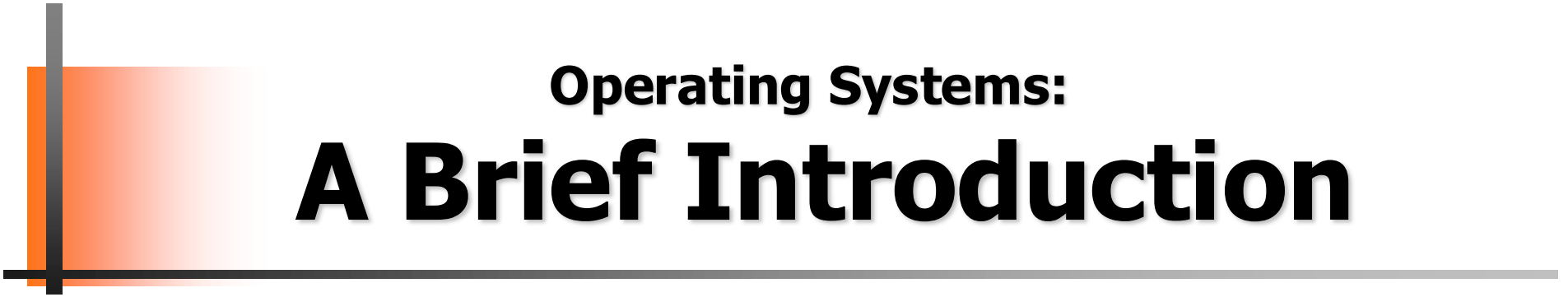
IN1020 – Introduksjon til datateknologi



Operating Systems:

A Brief Introduction

Håkon Kvale Stensland & Pål Halvorsen

Can we do without Operating Systems?

- **Yes, but cumbersome**



- From the history books -
  A Nord-12 machine from 1972:
  - Load a punched tape with program to edit code.
  - Load a punch tape with my program.
  - Edit the program.
  - Print a punch tape with the updated application.
  - Load a punch tape with the compiler.
  - Load the punch tape with my program.
  - The compiler writes a punched tape with finished machine code.
  - Load the punch tape with the code and run the program



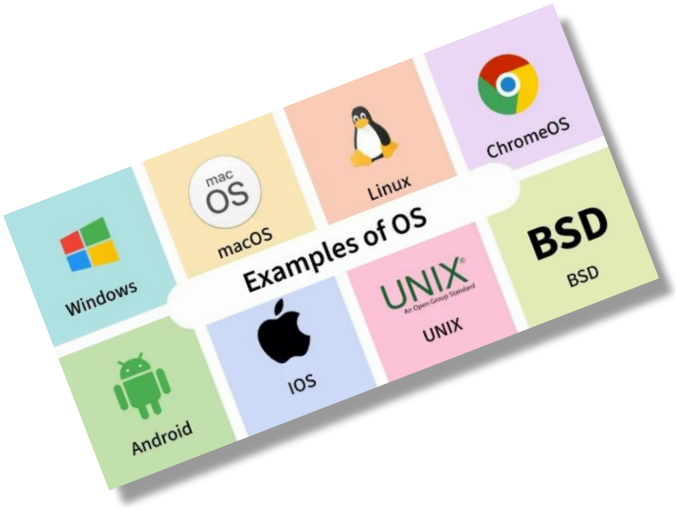
→ **Make the smallest programs takes forever…**
→ **More complex world today in every aspect!!**

Overview

- Basic execution environment

  (why do you need an operating system (OS), or need to know anything about it)

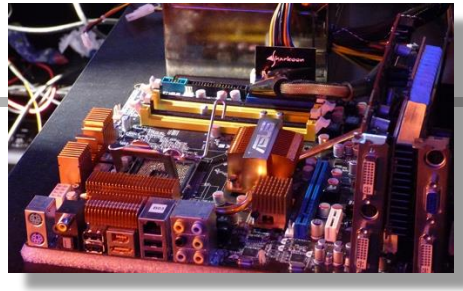
- What is an OS?



- OS components and services

  (greatly extended in later courses)

Hardware



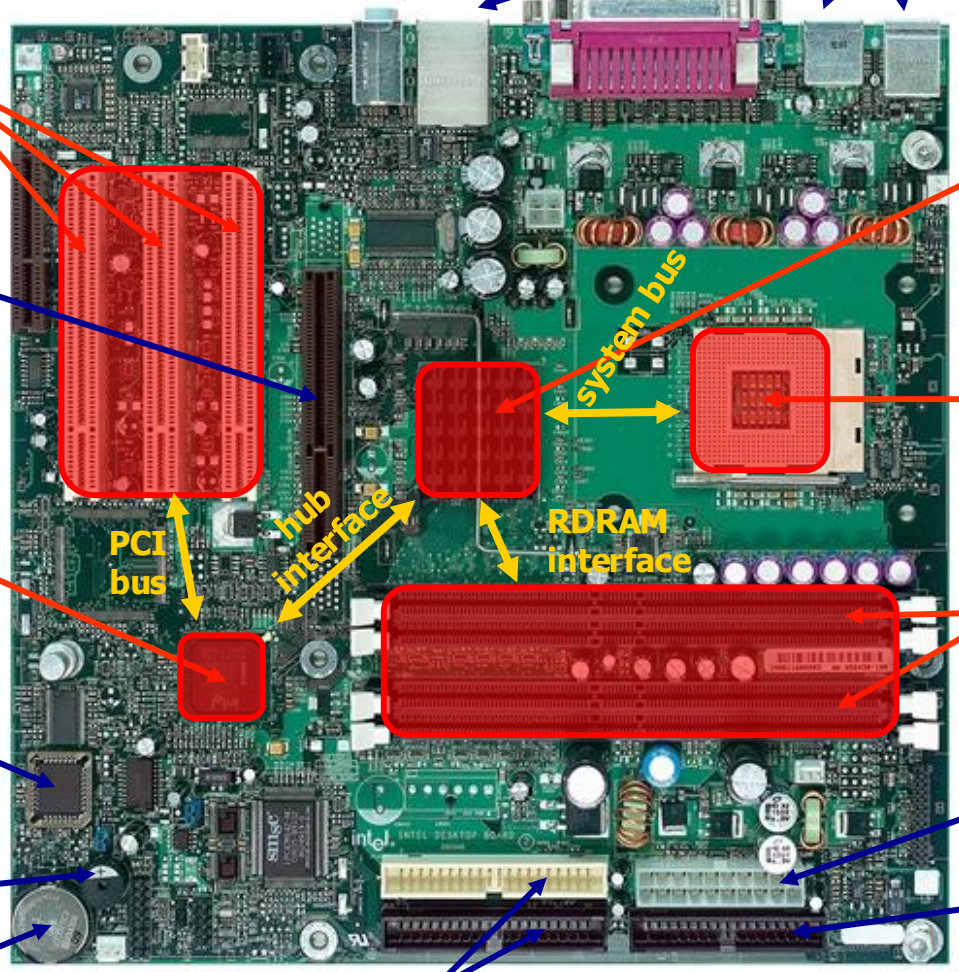
- **Central Processing Units**
  (CPUs, GPUs, …)
- **Memory**
  (cache(s), RAM, ROM, Flash, …)
- **I/O Devices**
  (network cards, disks, CD, keyboard, mouse, …)
- **Links**
  (interconnects, busses, …)

An easy, **old** example:

Intel Hub Architecture

Intel D850MD Motherboard:

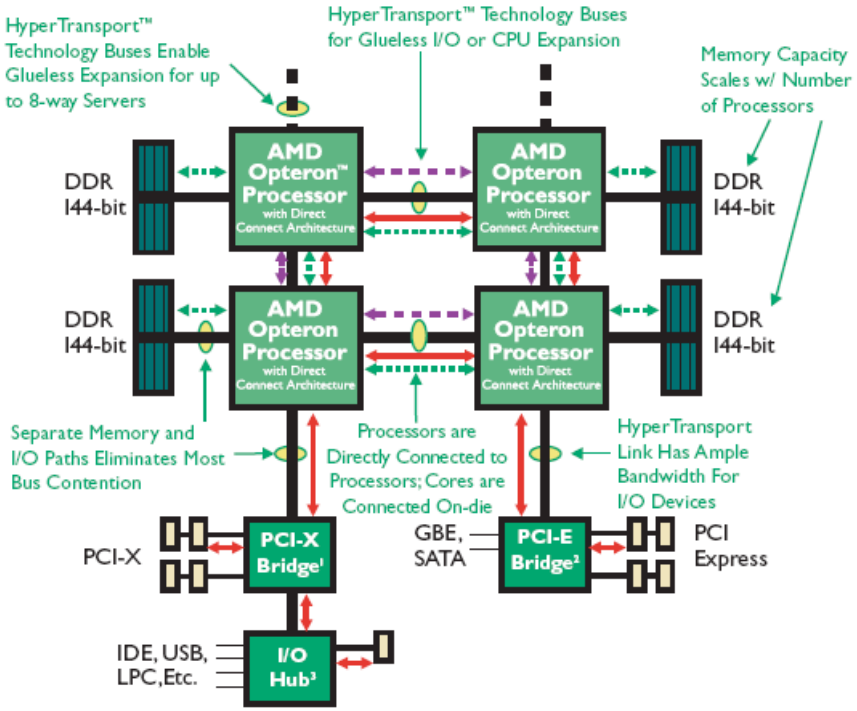
Source: Intel® Desktop Board D850MD/D850MV Technical Product Specification



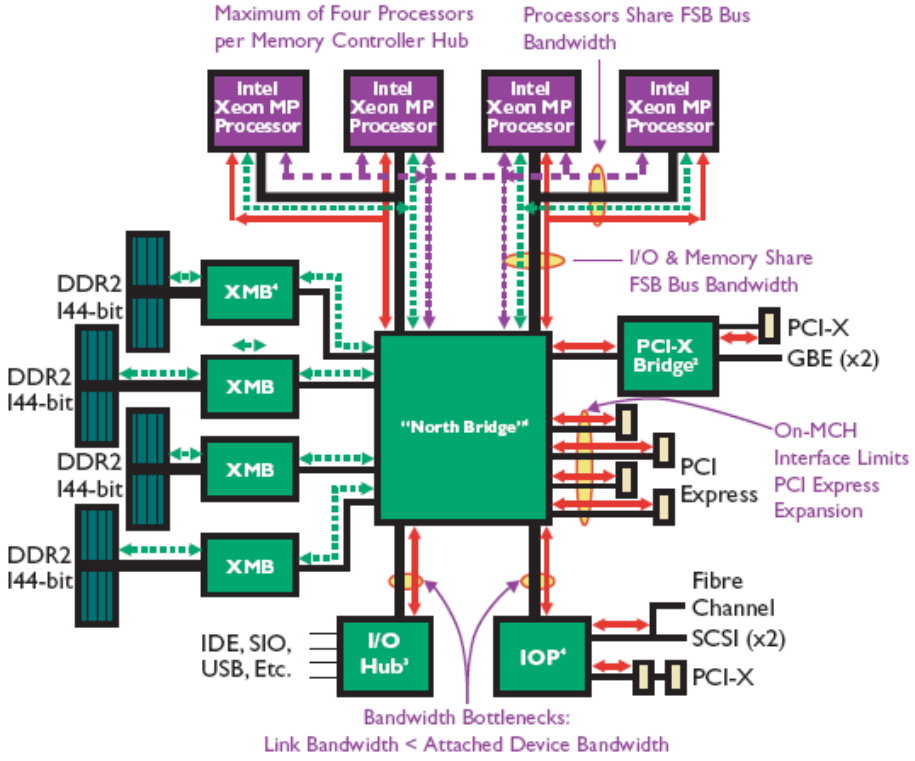
Other examples:

AMD Opteron & Intel Xeon

AMD Opteron™ Processor-based 4P Server

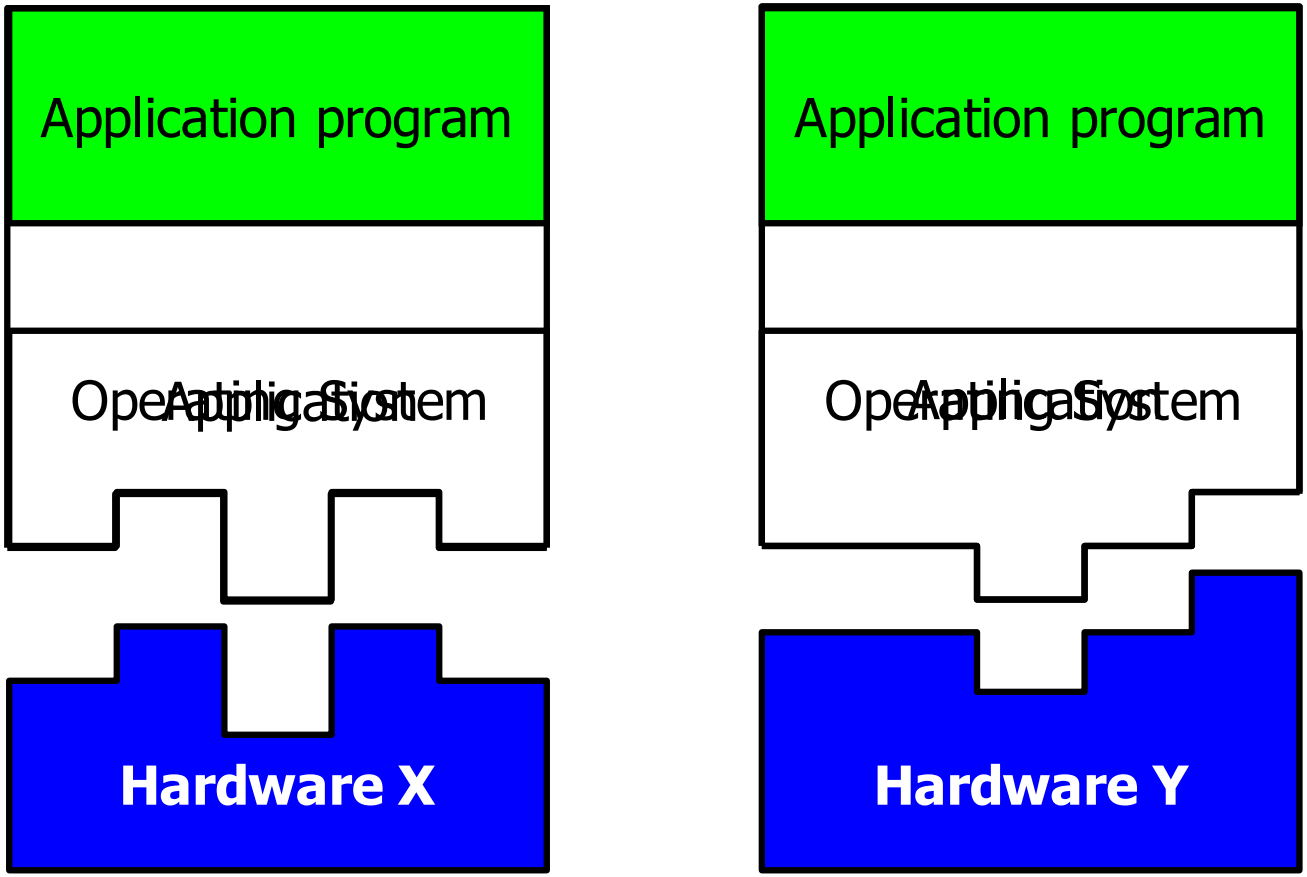


Intel Xeon MP Processor-based 4P



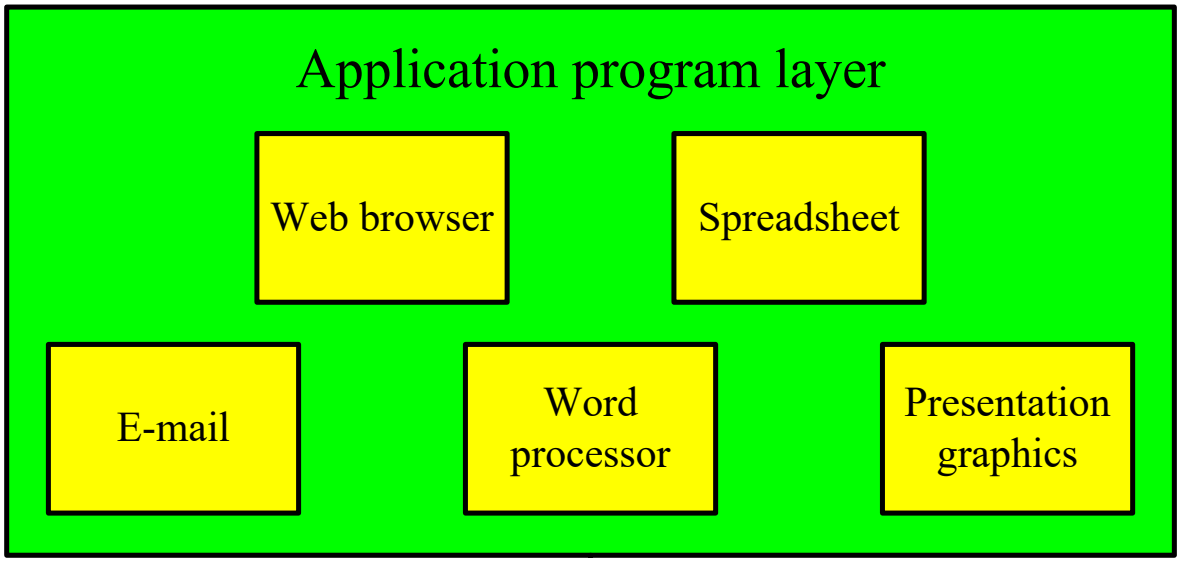
☞ Different hardware may have different bottlenecks
==> nice to have an **operating system** to control the HW?

Different Hardware



Many Concurrent Tasks

- Better use & utilization
  - many concurrent processes
    - performing different tasks
    - using different parts of the machine



  - many concurrent users



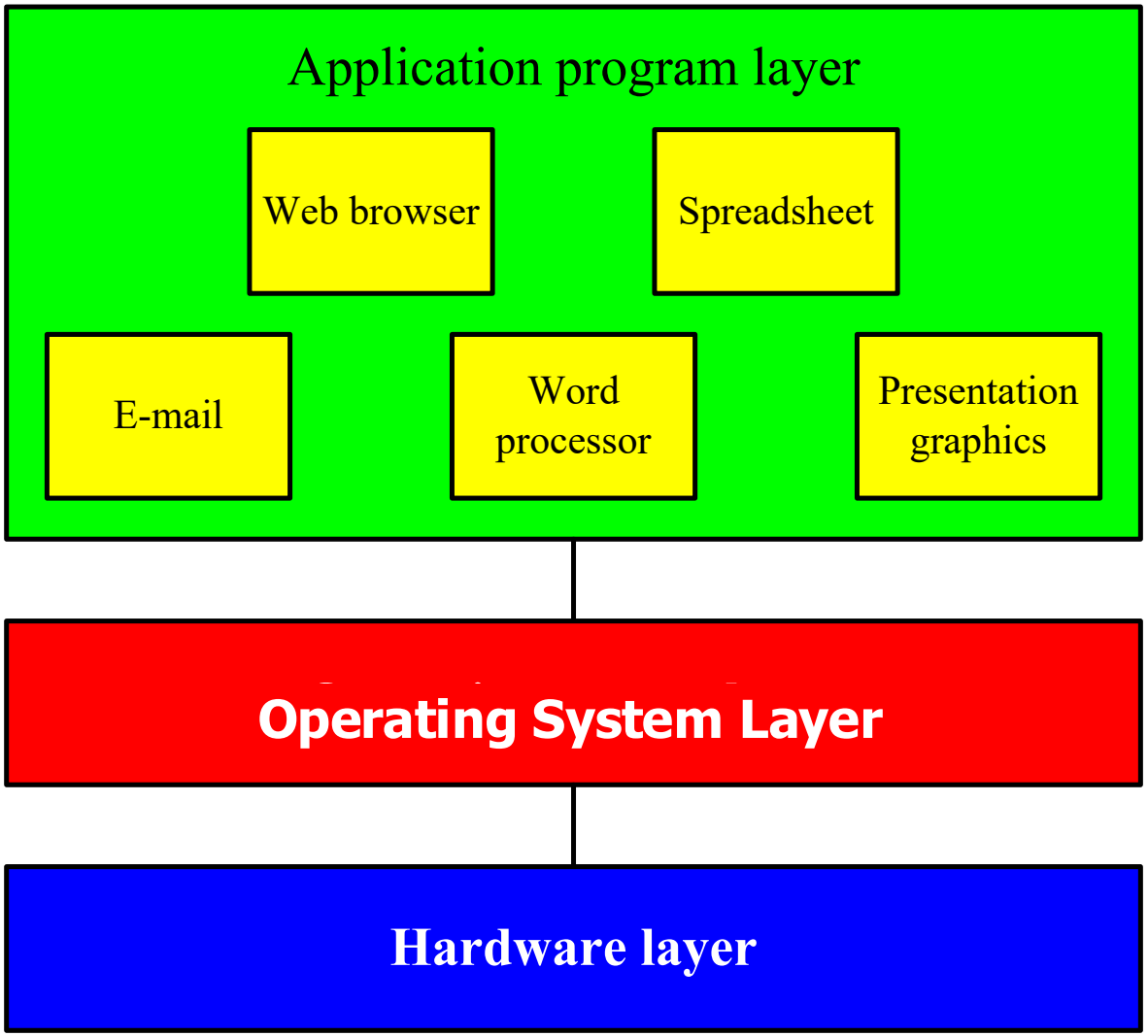


- Challenges
  - "concurrent" access
  - protection/security
  - fairness
  - ...

Many Concurrent Tasks

extra copy for handout

- Better use & utilization
  - many concurrent processes
    - performing different tasks
    - using different parts of the machine
  - many concurrent users
- Challenges
  - "concurrent" access
  - protection/security
  - fairness
  - ...

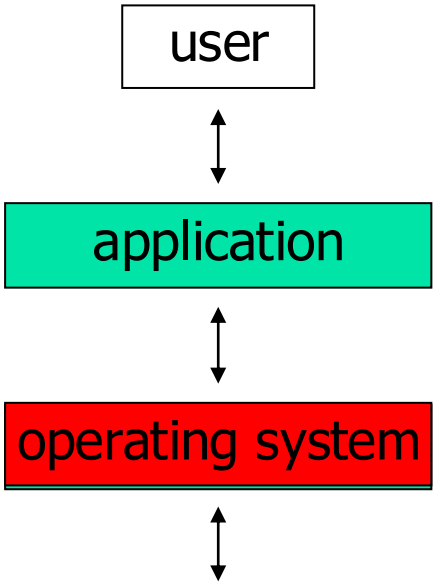


What is an Operating System (OS)?

- *"An operating system (OS) is a collection of programs that acts as an intermediary between the hardware and its user(s), providing a high-level interface to low level hardware resources, such as the CPU, memory, and I/O devices. The OS provides various facilities and services that make the use of the hardware convenient, efficient and safe"*

Lazowska, E. D.: Contemporary Issues in Operating Systems , in: Encyclopedia of Computer Science, Ralston, A., Reilly, E. D. (Editors), IEEE Press, 1993, pp.980

- It is an extended machine (top-down view)
  - Hides the messy details
  - Presents a virtual machine, easier to use
- It is a resource manager (bottom-up view)
  - Each program gets time/space on the resource



Where do we find OSes?

Computers



Phones



Game Boxes



Cars

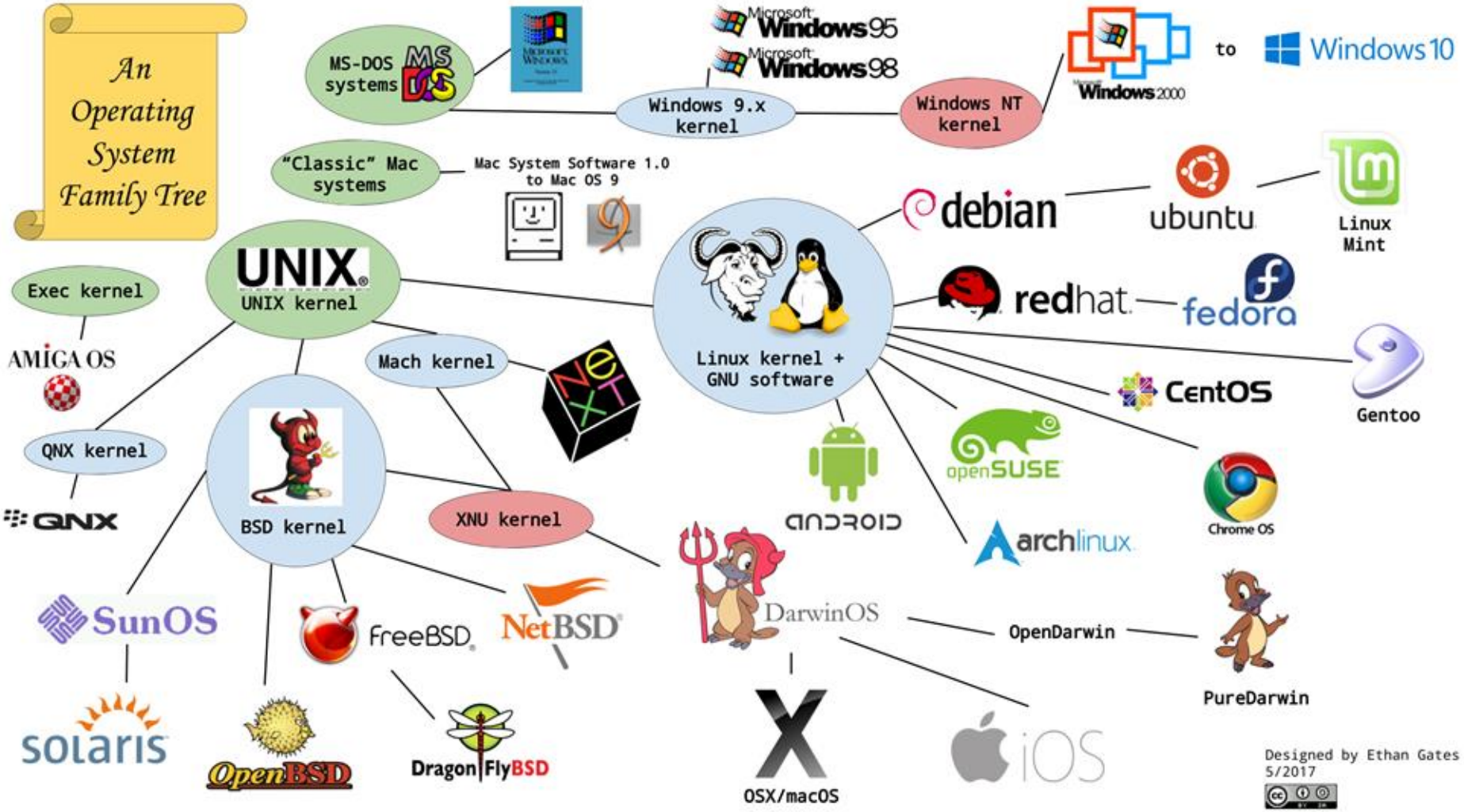


cameras,
other vehicles/crafts,
set-top boxes,
watches,
sensors,
… → **EVERYWHERE**

Operating System Categories

- Single-user, single-task:
  historic, and rare (only a few old PDAs use this)
- Single-user, multi-tasking:
  PCs and workstations may be configured like this (typically, phones today)
- Multi-user, multi-tasking:
  used on large, old mainframes; and handhelds, PCs, workstations and servers today
- Distributed OSes:
  support for administration of distributed resources
- Real-time OSes:
  support for systems with real-time requirements like cars, nuclear reactors, etc.
- Embedded OSes:
  built into a device to control a specific type of equipment like cellular phones, microwaves, washing machines, etc.

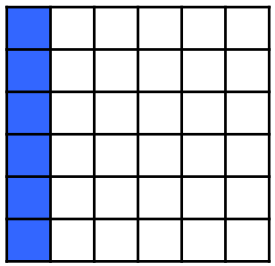
OS family tree – most systems are related



So, why study OSes?

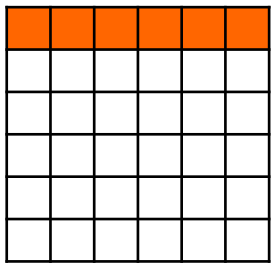
- "I will never write an operating system from scratch, nor even touch a line of code in the kernel"
- "Operating systems have existed for decades, what more can be added?"
- "I just need to know the API to give the system commands in order to store my data and run my programs"
- "Writing programs in Java or Python is very easy, and I do not need to know anything about operating systems to make it work"
- Consider the following example, does it matter which one to use?:

```
int data[15000][15000];
for (j = 0; j < 15000; j++)
  for (i = 0; i < 15000; i++)
      data[i][j] = i*j;
```



	on my mac: **datasize**	
3.9s	15.000	1.2s
12.2s	25.000	3.5s
22.1s	30.000	4.7s
49.3s	40.000	10.1s

```
int data[15000][15000];
for (j = 0; j < 15000; j++)
  for (i = 0; i < 15000; i++)
      data[j][i] = i*j;
```



So, why study OSes?

Failing to meet the Technical Challenges...



... influence your everyday life

– **latency can literally kill**!

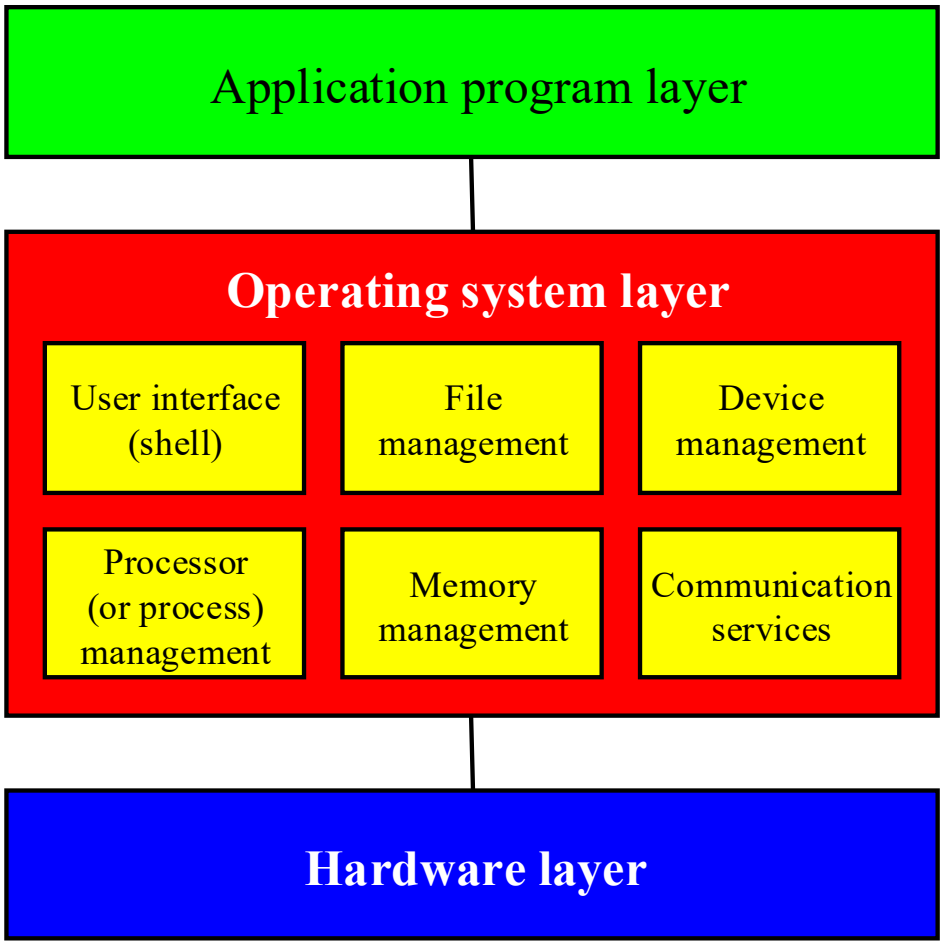
So, why study OSes?

- To do things right and efficient, one must know how computers and operating systems work (and networks and …)
  - operating systems provide magic to provide "infinite" CPU cycles, "endless" memory, transparent access to devices, networked computing, etc.
  - operating systems manage concurrency and sharing
  - understand the tradeoffs between performance and functionality, division of labor between HW and SW
- *OSes are found everywhere and are therefore key components in many systems*

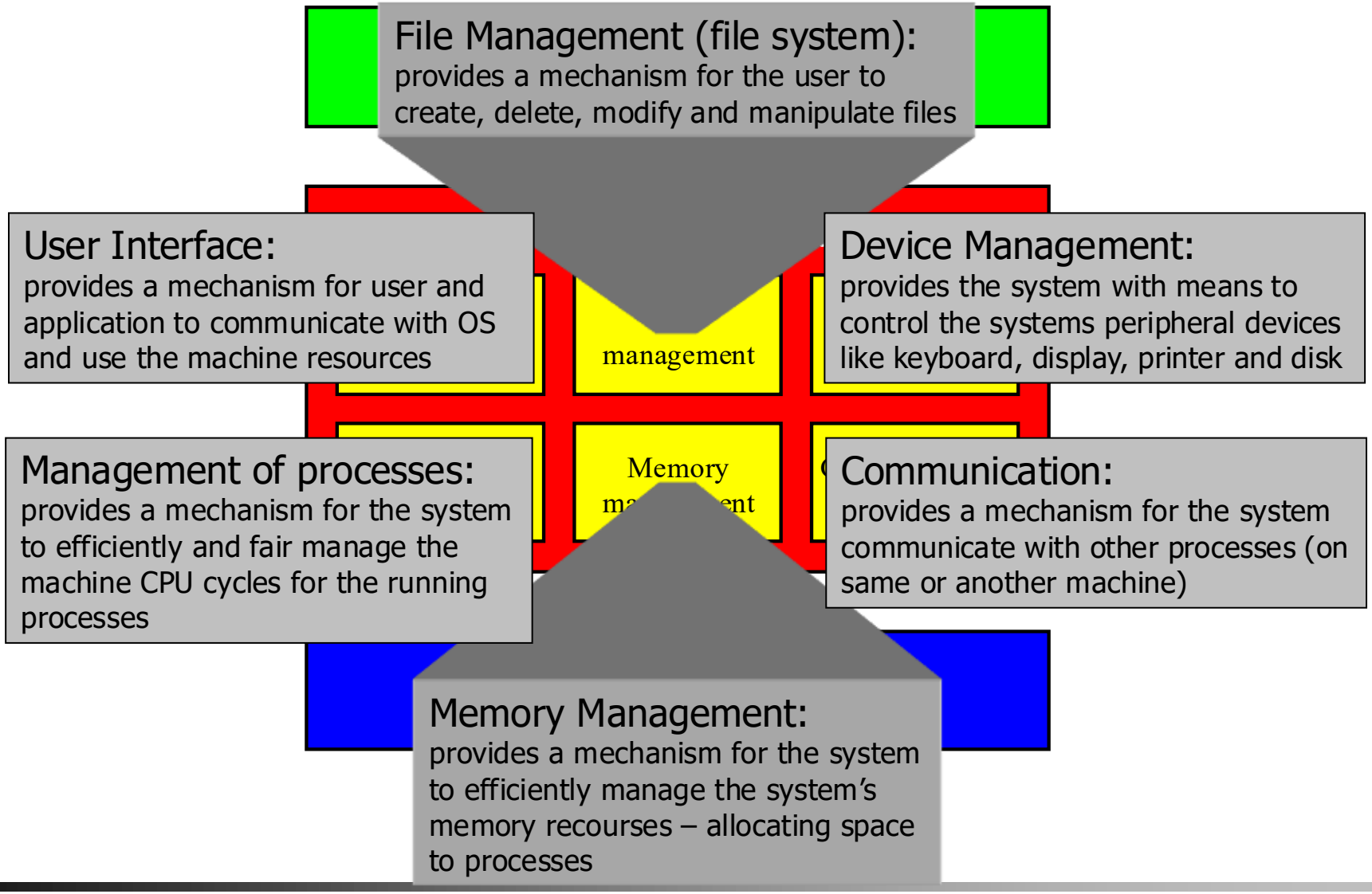


Primary Components in OSes

- "Visible" to user
  - Shell
  - File system
  - Device management
- "(Semi)Transparent"
  - Processor management
  - Memory management
  - Communication services

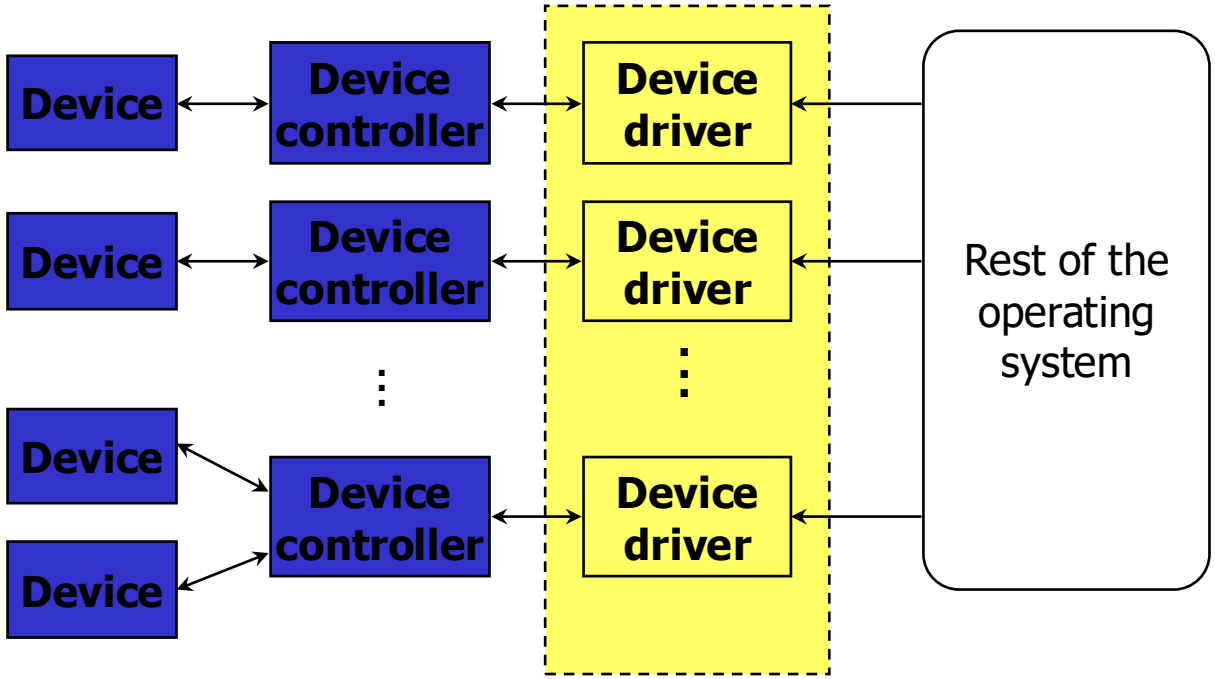


Primary Components in OSes



Device Management

- The OS must be able to control pheripal devices such as disks, keyboard, network cards, screen, speakers, mouse, ...
  - large diversity
  - varying speeds
  - different access modes
- The software talking to the controller and giving commands is called a **device driver**



- A huge amount of code (50-95% of the OS code!!??)

(As of 2021, the 5.11 release of the Linux kernel had around 30.34 million lines of code. Roughly 14% of the code is part of the "core," including architecture-specific code, kernel code, and memory management code, while **60% is drivers**.)

Interfaces

- The connection between the user and the computer
- The OS incorporates logic that support interfaces with both hardware and applications, e.g.,
  - graphical user interface (GUI)
    - interface consisting of windows, icons, menus and pointers
    - often not part of the OS (at least not kernel), but an own program
  - command line interface, e.g., a shell
  - application programming interface (API)
  - …
- Example: X (see `man X`)
  - network transparent window system running on most ANSI C and POSIX (portable OS interface for UNIX) compliant systems
  - uses inter-process communication to get input from and send output to various client programs
- Several other examples:
  Desktop Window Manager (Windows), Wayland (Linux)

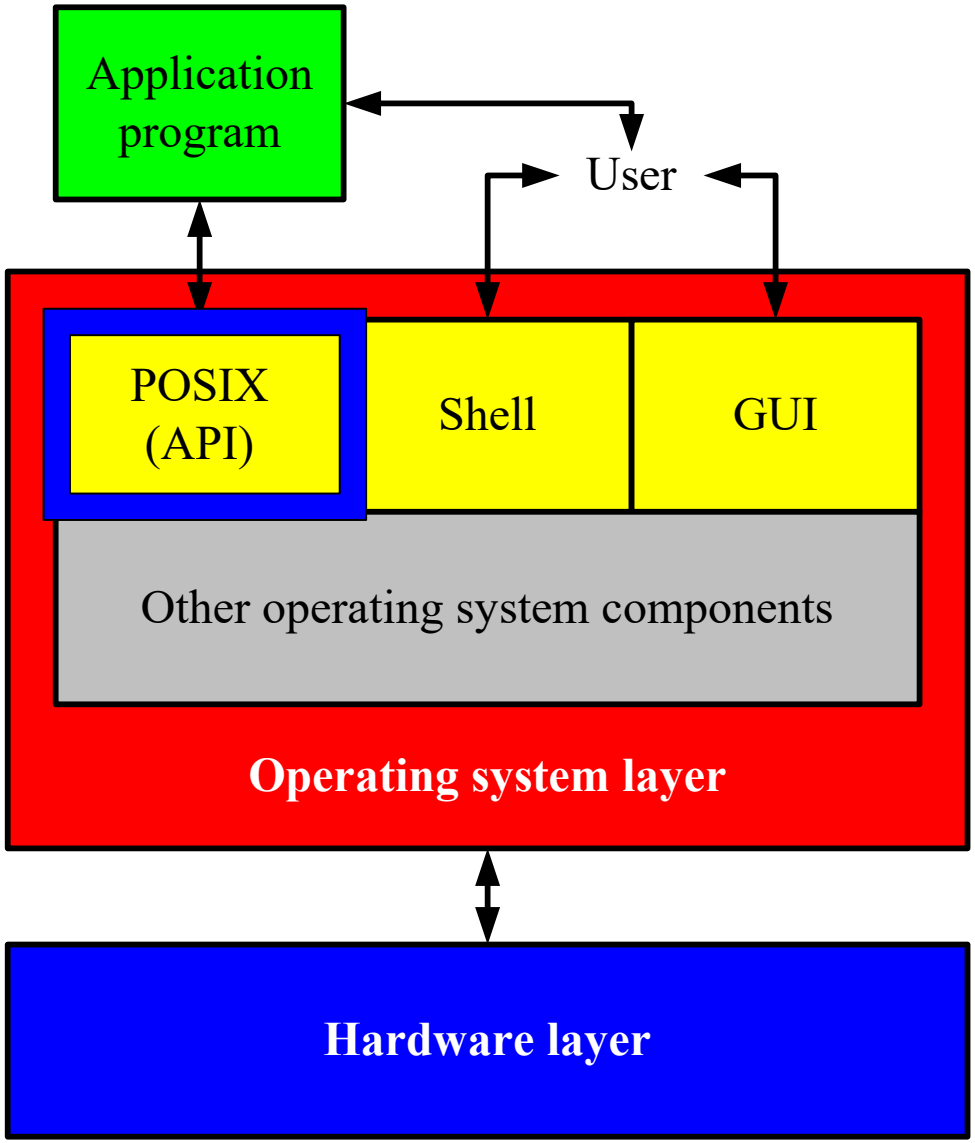
UNIX Interfaces

Applications access HW through an API consisting of a set of routines, protocols and other tools
(e.g., POSIX – portable OS interface for UNIX)

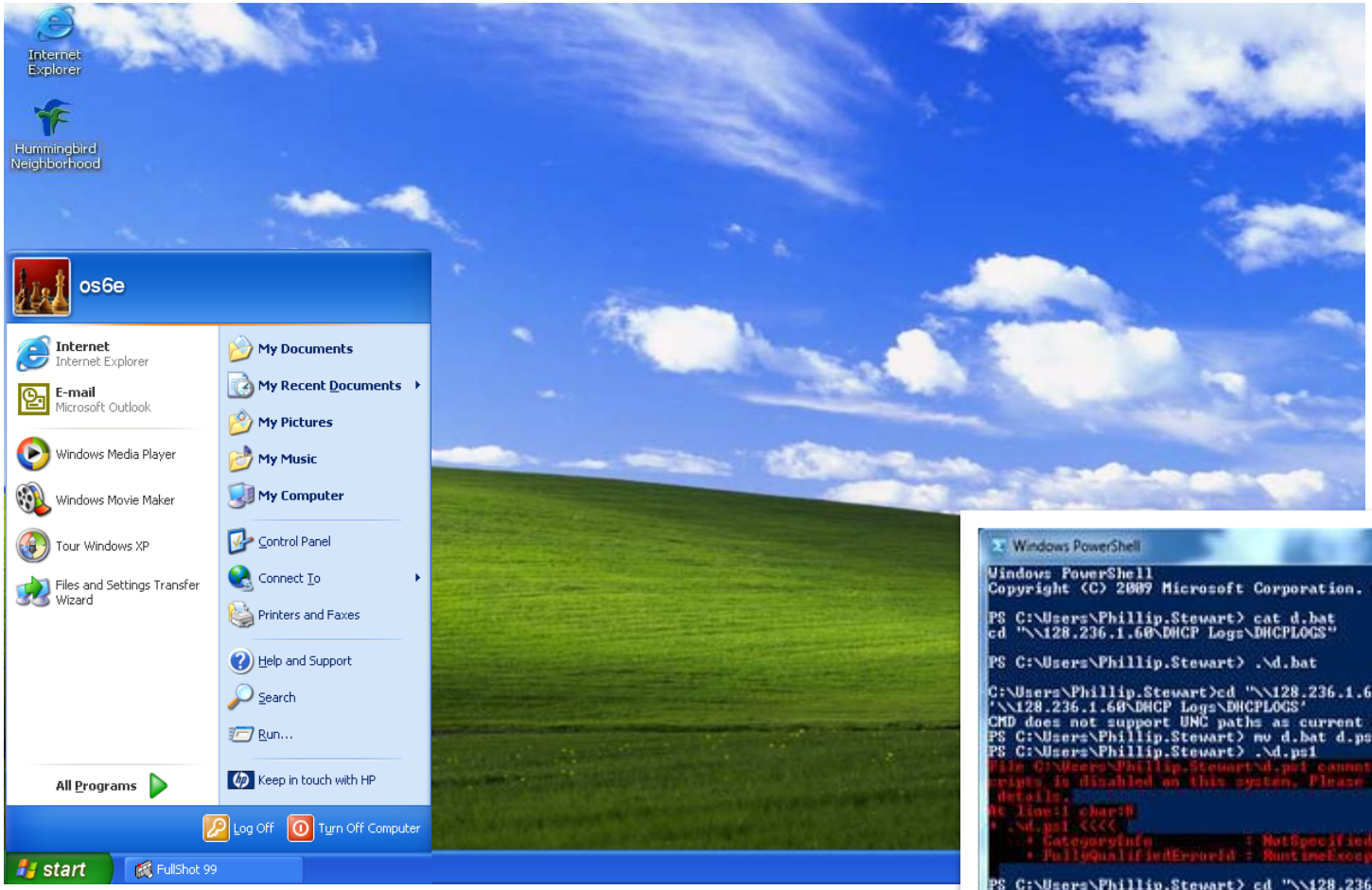
A user can interact with the system through the application interface or using a command line prosessed by a shell (not really a part of the OS)

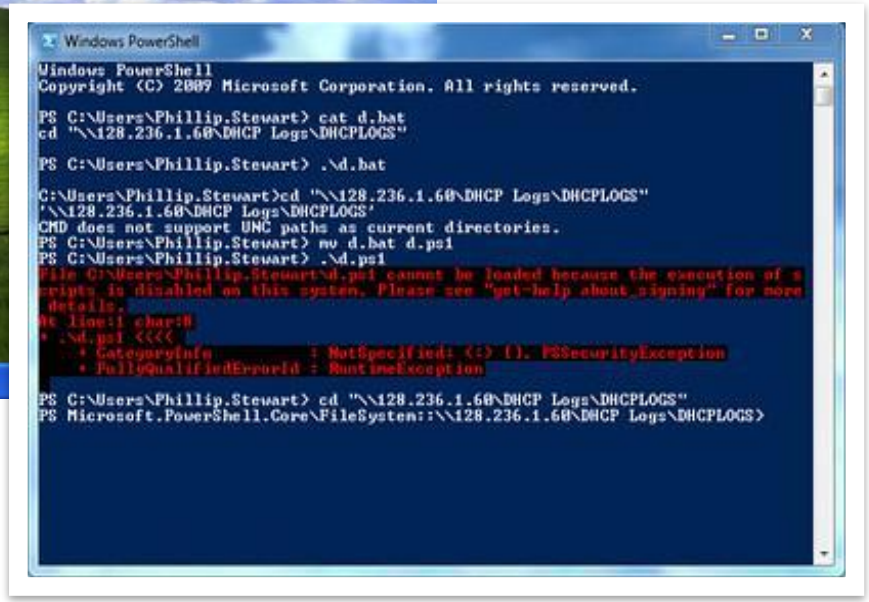
A plain command line interface may be hard to use. Many UNIX systems therefore have a standard graphical interface (X Windows) which can run a desktop system
(like KDE, Gnome, Fvwm, Afterstep, Awsome, bspwm, …)

Windows is more or less similar…



Users: Sample Windows Interfaces





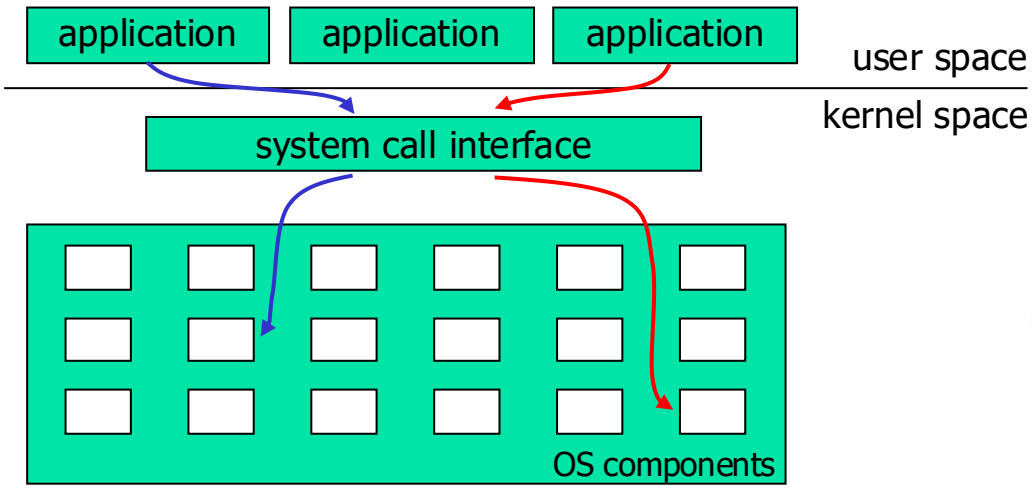
Programming: System Calls

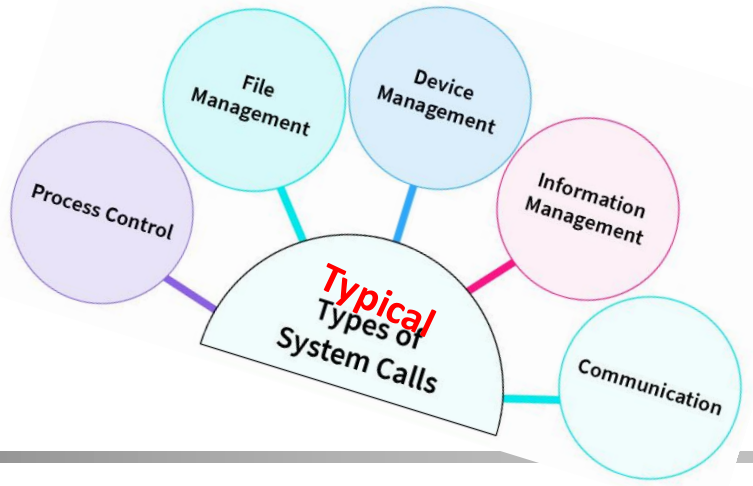
- The interface between the OS and users is defined by a set of **system calls**

- Making a system call is similar to a procedure/function call, but system calls enter the kernel:

Linux:
x86 v2.4.19 *entry.S* → 242
x86 v3.0-rc4 *syscall_table_32.S* → 347
x86 v3.16.2 syscall_32.tbl → 353

FreeBSD:
v9 *syscalls.c* → 531

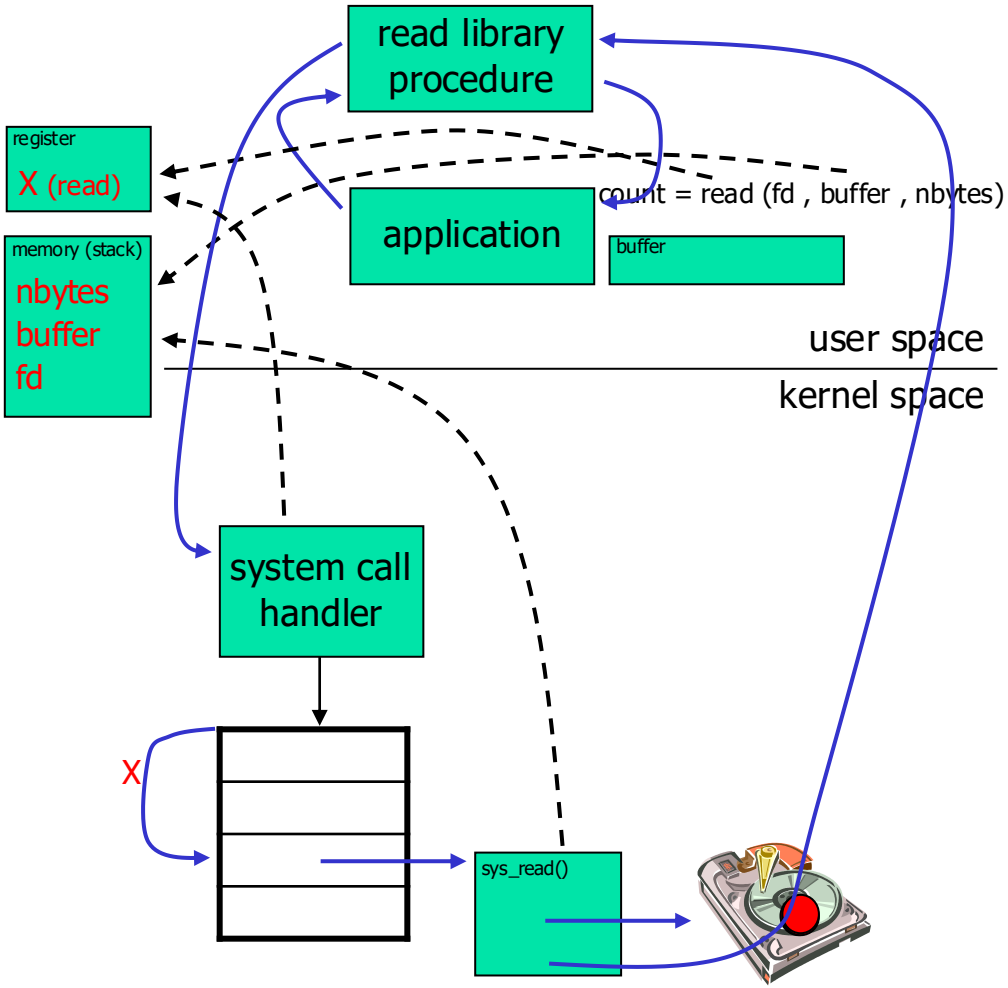




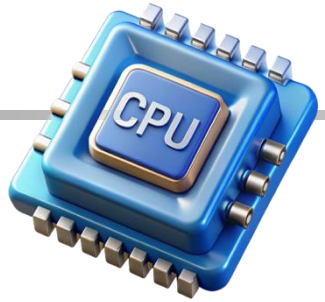
Programming: System Calls

- C example:
  `count = read(fd,buffer,nbyte)`

1. push parameters on stack
2. call library code
3. put system call number in register
4. call kernel (TRAP)
   - ✓ kernel examines system call number
   - ✓ finds requested system call handler
   - ✓ execute requested operation
5. return to library and clean up
   - ✓ increase instruction pointer
   - ✓ remove parameters from stack
6. resume process



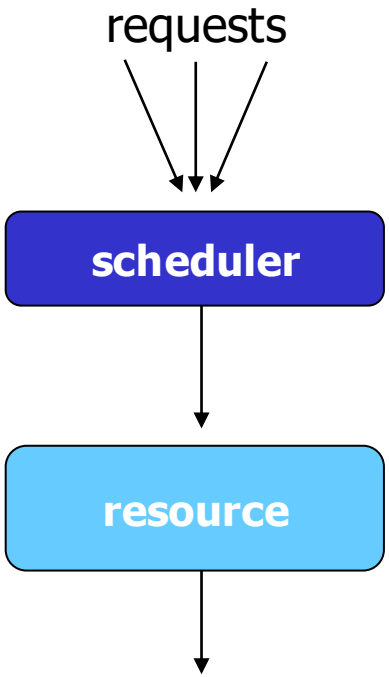
CPU Scheduling



- A task is a schedulable entity that can run on the CPU
  (a process executing a job, e.g., a user program)

- In a multi-tasking system, several tasks may wish to use the CPU simultaneously

- A scheduler decides which task that may use the CPU,
  i.e., determines order
  by which requests are serviced,
  using a scheduling algorithm



Scheduling: goals

- A variety of (contradicting) factors to consider
  - treat similar tasks in a similar way
  - no process should wait forever
  - short response times ($\text{time}_{\text{response given}} - \text{time}_{\text{request submitted}}$)
  - maximize throughput
  - maximum resource utilization (100%, but 40-90% normal)
  - minimize overhead
  - predictable access
  - …

- Several ways to achieve these goals, …

  …but hard (impossible?) to achieve all …

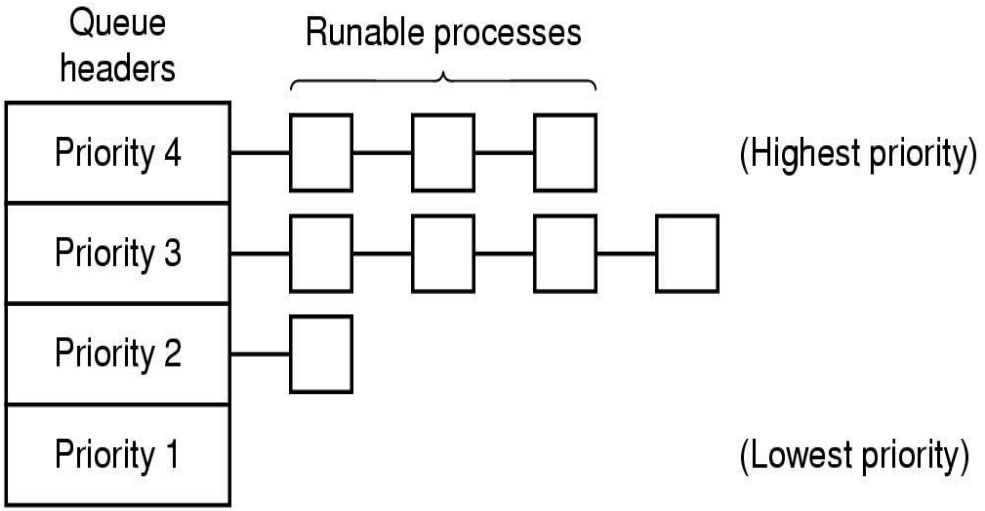
  …but which criteria is most important, most reasonable?

  …depends on who you are, environment, purpose of the system, …!

Scheduling: Priorities

- Assign each process a priority
- Run the process with highest priority in the ready queue first

- Multiple queues, often round-robin in each

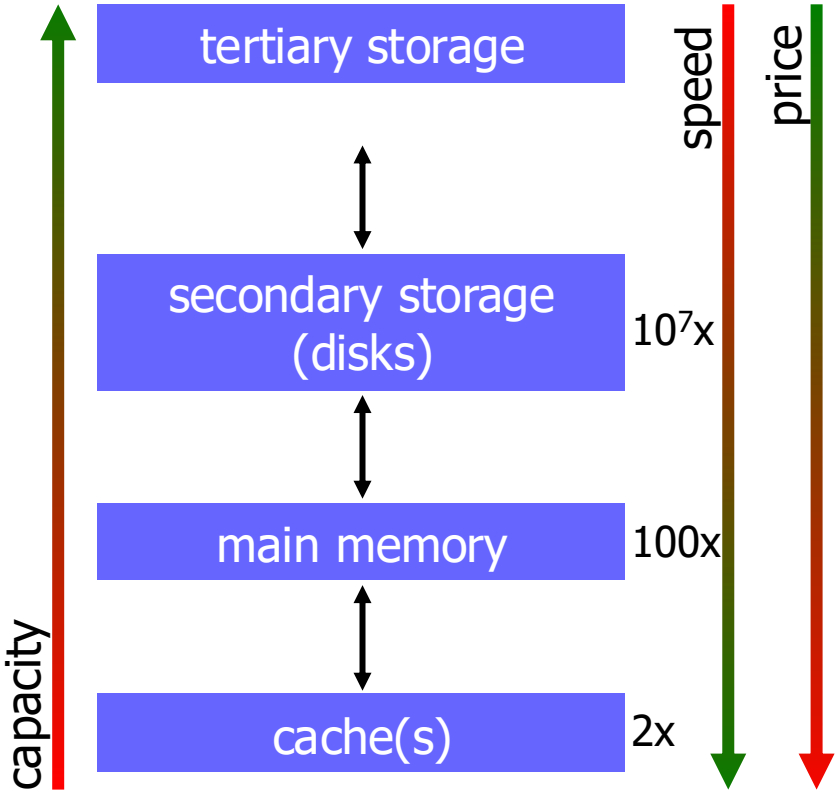


- Advantage
  - (Fairness)
  - Different priorities according to importance

- Disadvantage
  - Starvation (so maybe use dynamic priorities?)

Memory Hierarchies

- A process typically needs a lot of memory:
  We can't access the disk every time we need data
- Typical computer systems therefore have several different components where data may be stored
  - different capacities
  - different speeds
  - less capacity gives faster access and higher cost per byte
- Lower levels have a copy of data in higher levels
- A typical memory hierarchy:



Memory Management

- Memory management is concerned with managing the systems' **memory resources**
  - allocate space to processes
  - protect the memory regions
  - provide a virtual view of memory giving the impression of having more than the number of available bytes
  - control different levels of memory in a hierarchy

The Challenge of Multiprogramming

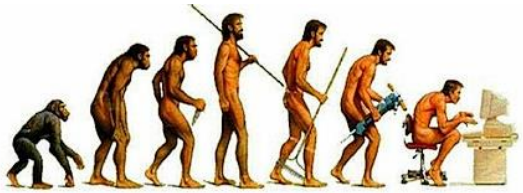
- Many "tasks" require memory
  - several processes concurrently loaded into memory



  - memory is needed for different tasks within a process

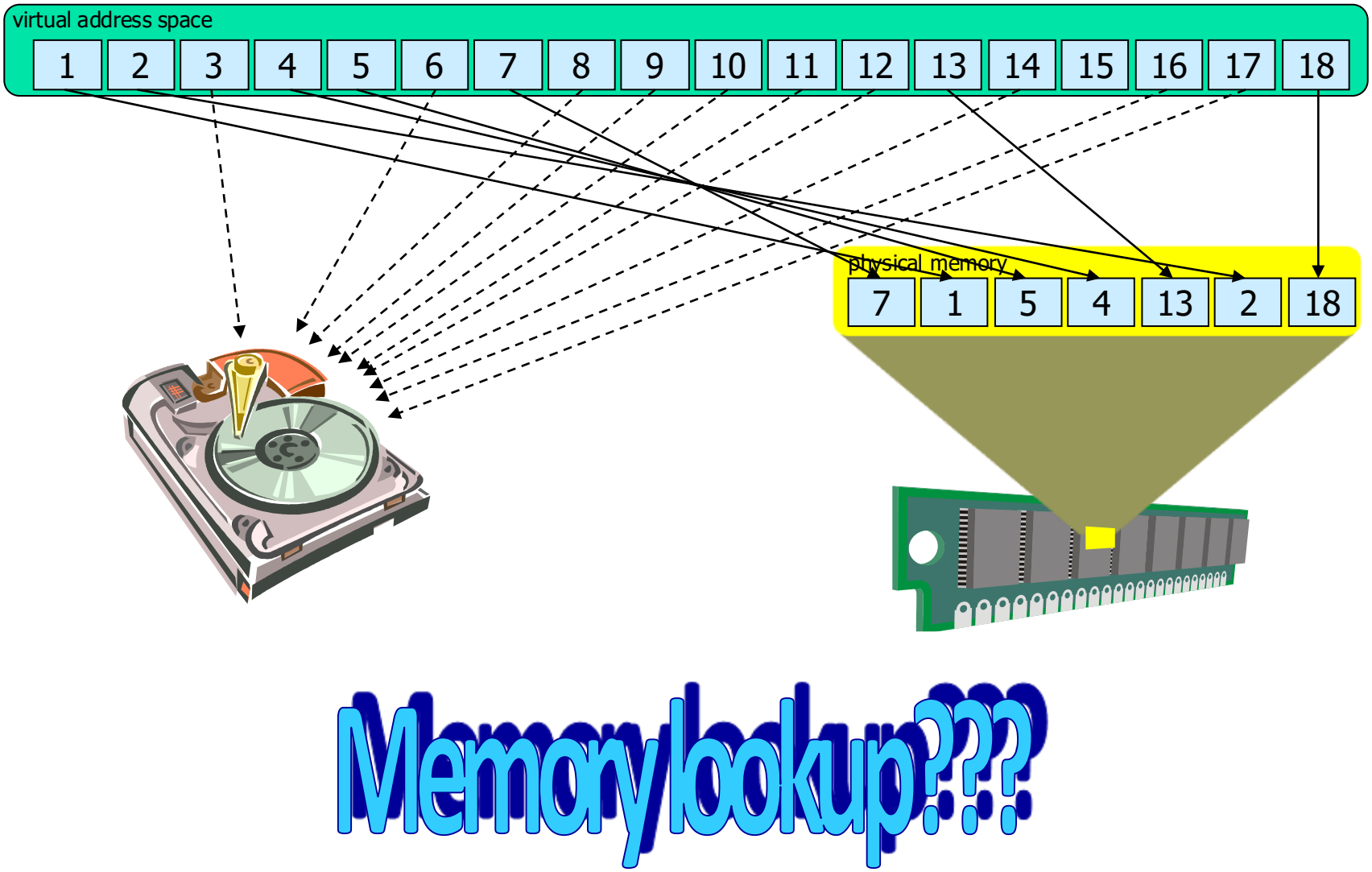


  - process memory demand may change over time



➥ **OS must arrange (dynamic) memory sharing**

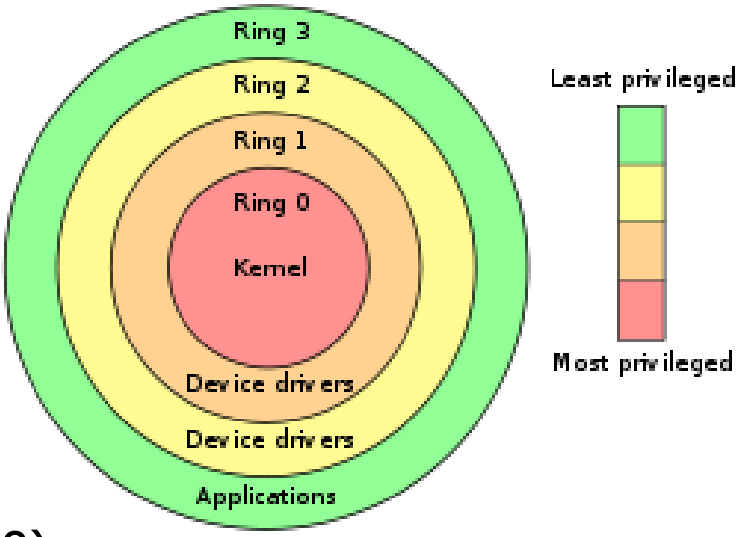
Managing hierarchies and the "limitless-illusion"



User Level vs. Kernel Level (Protection)

- Many OSes distinguish user and kernel level, i.e., due to security and protection

- Usually, applications and many sub-systems run in user mode (level 3)
  - protected mode
  - not allowed to access HW or device drivers directly, only through an API
  - access to assigned memory only
  - limited instruction set



- OSes run in kernel mode (under the virtual machine abstraction, level 0)
  - real mode
  - access to the entire memory
  - all instructions can be executed
  - bypass security

Summary



- OSes are found "everywhere" and provide virtual machines and work as a resource manager
- Hides complexity at many levels
- Users access the services using an interface like system calls
- Various components managing the different resources efficiently and fair
- Provides safety between processes and users
- More OS?
  - *IN2140 – Introduksjon til operativsystemer og datakommunikasjon*
  - *IN3000 – Operativsystemer*