Operating Systems Overview

Chester Rebeiro
IIT Madras

Outline

- Basics
- OS Concepts
- OS Structure

What is the OS used for?

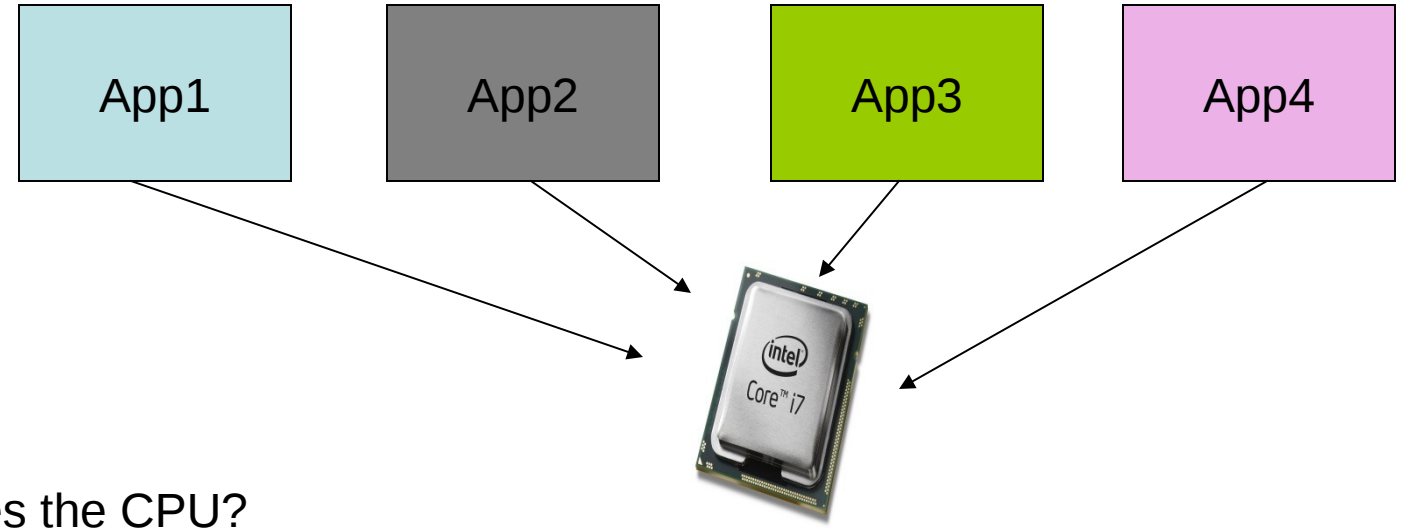
- Hardware Abstraction

  turns hardware into something that applications can use

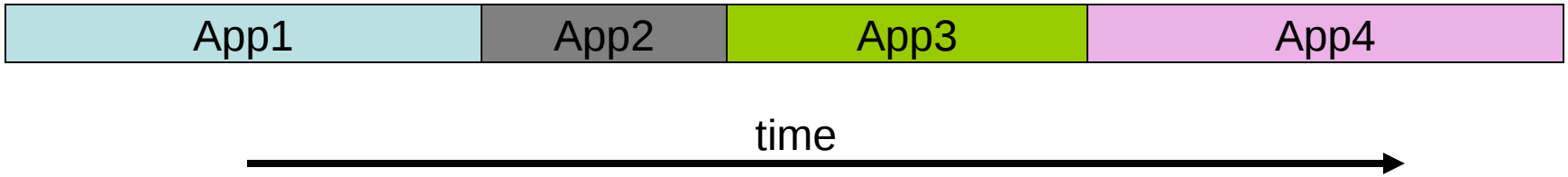
- Resource Management

  manage system's resources

Sharing the CPU

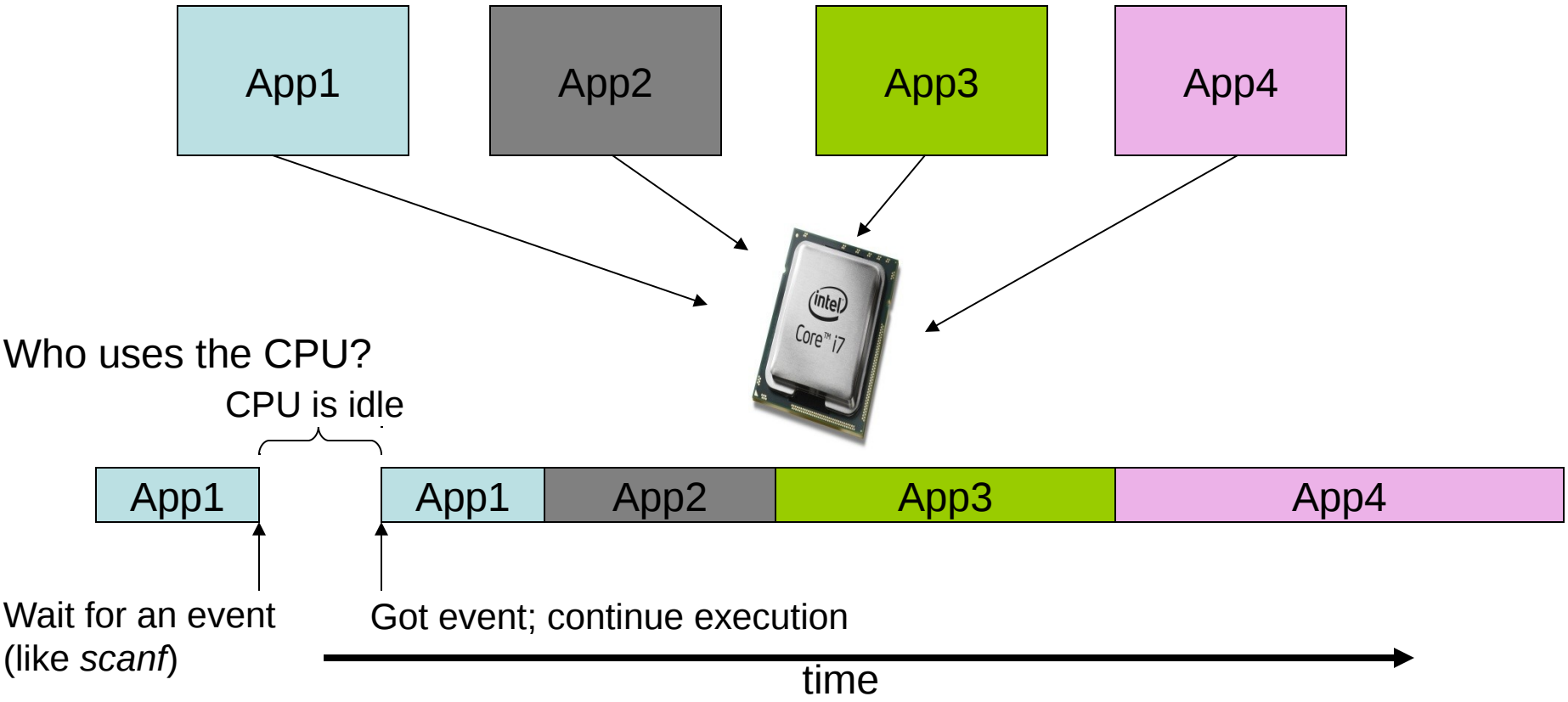


Who uses the CPU?



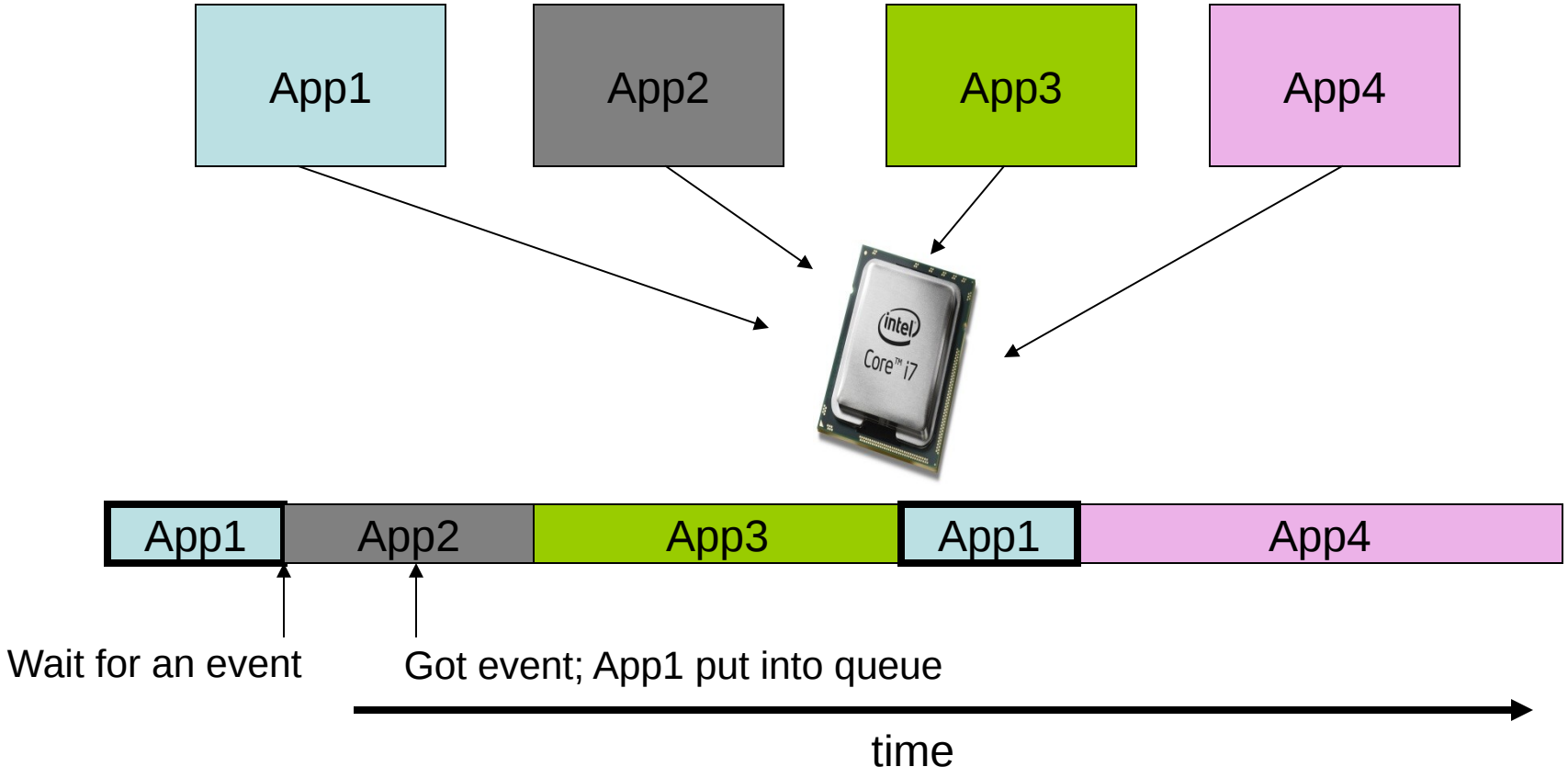
When one app completes the next starts

Idle CPU Cycles



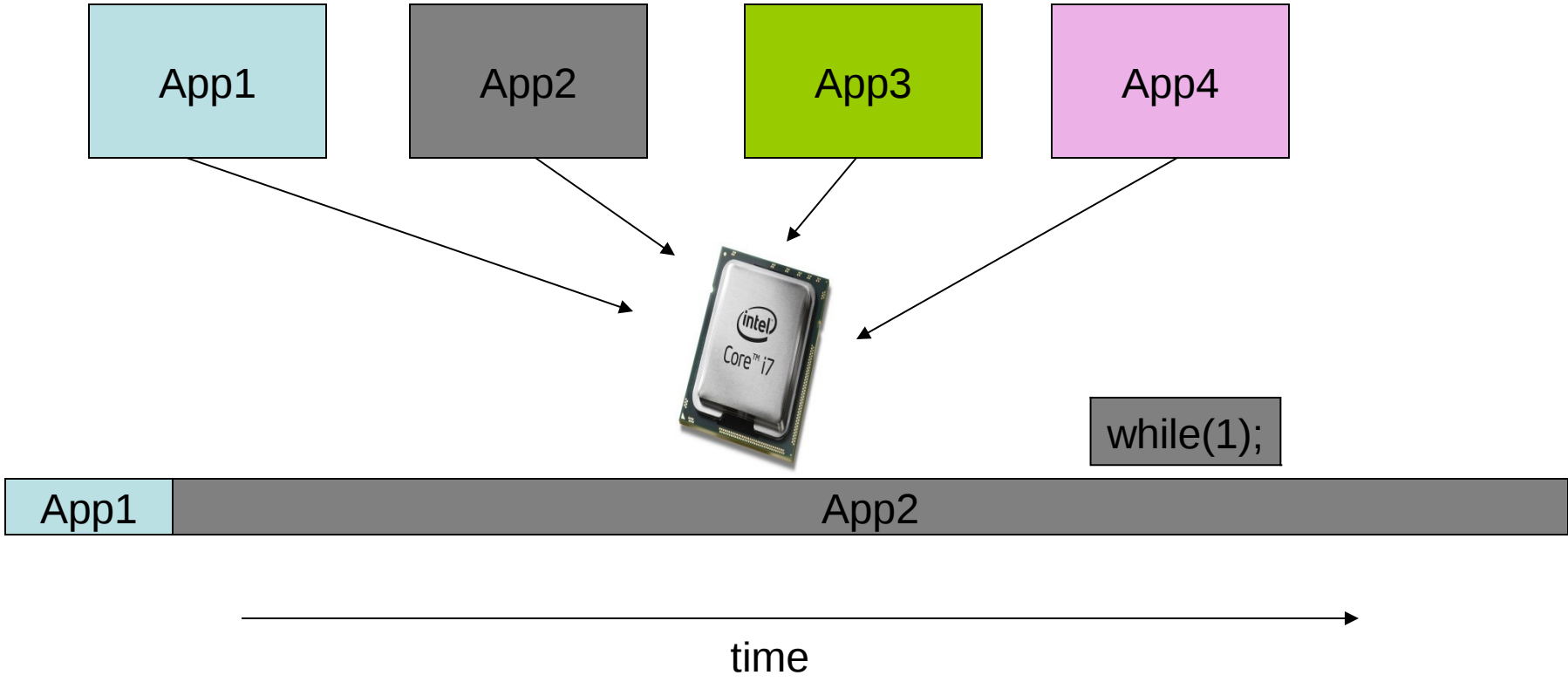
CPU is idle when executing app waits for an event.
Reduced performance.

When OS supports Multiprogramming



When CPU idle, switch to another app

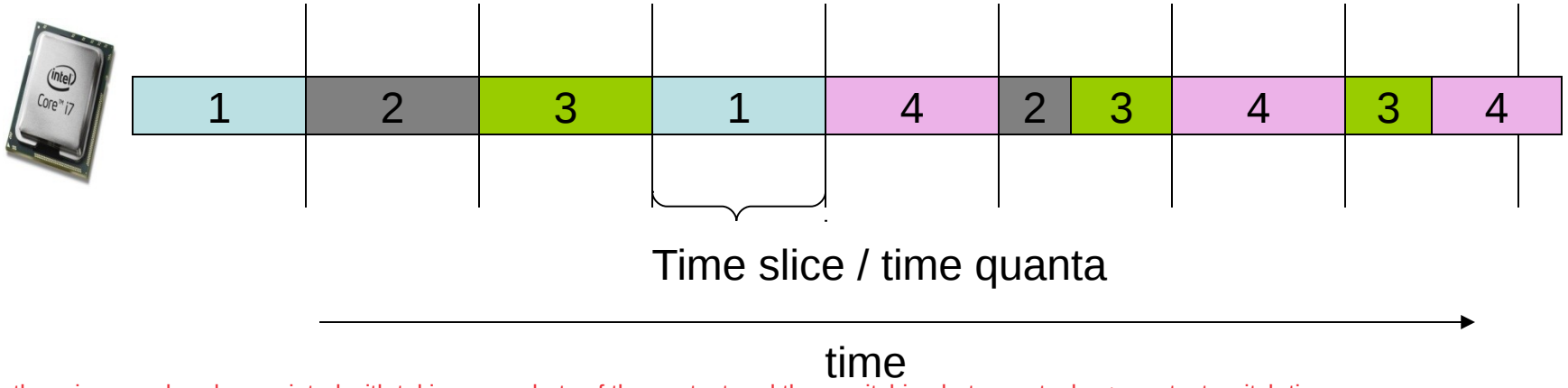
Multiprogramming could cause starvation



One app can hang the entire system

When OS supports Time Sharing (Multitasking)

- Time sliced
- Each app executes within a slice
- Gives impression that apps run concurrently
- No starvation. Performance improved



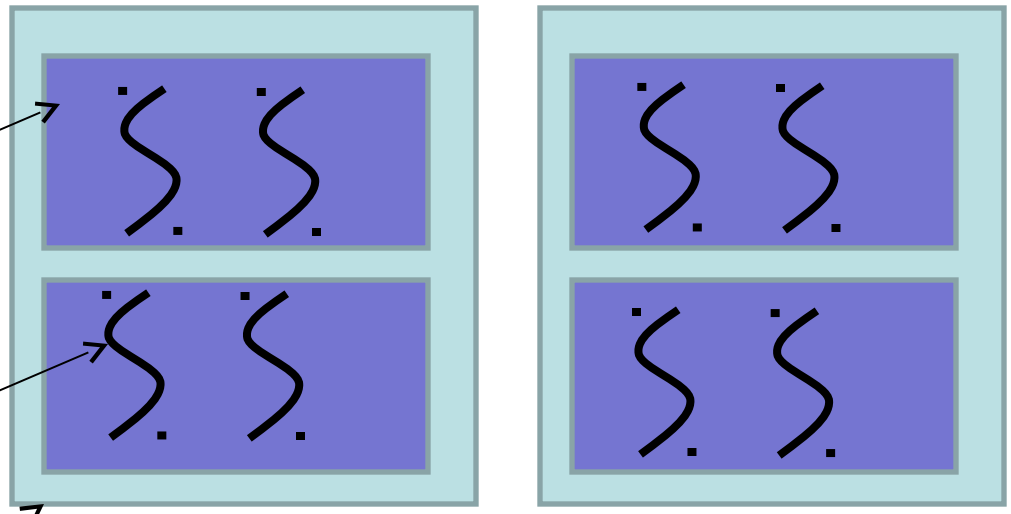
there is a overhead associated with taking snapshots of the context and then switching between tasks -> context switch time
tradeoff: if the time-slice is small, the overheads are significant. if it is too long, then there might be starvation and it might not even appear parallel

Other Shared Resources (examples)

- Printers
- Keyboards
- RAM
- disks
- Files on disk
- Networks

Multiprocessors

- Multiple processors chips in a single system
- Multiple cores in a single chip
- Multiple threads in a single core



threads are individual processes that can run on one core together

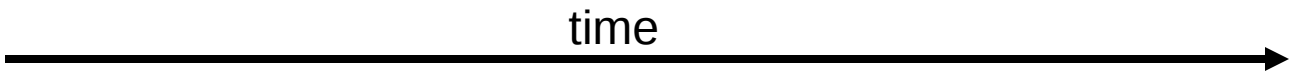
the apps prefer to run on core/socket for lower chances of cache misses and memory access requirements

Multiprocessors

- Each processor can execute an app independent of the others

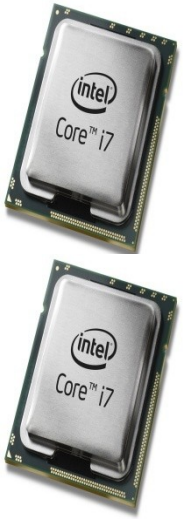
when the os has to come online and switch the apps, there is an interrupt that is triggered by the timer

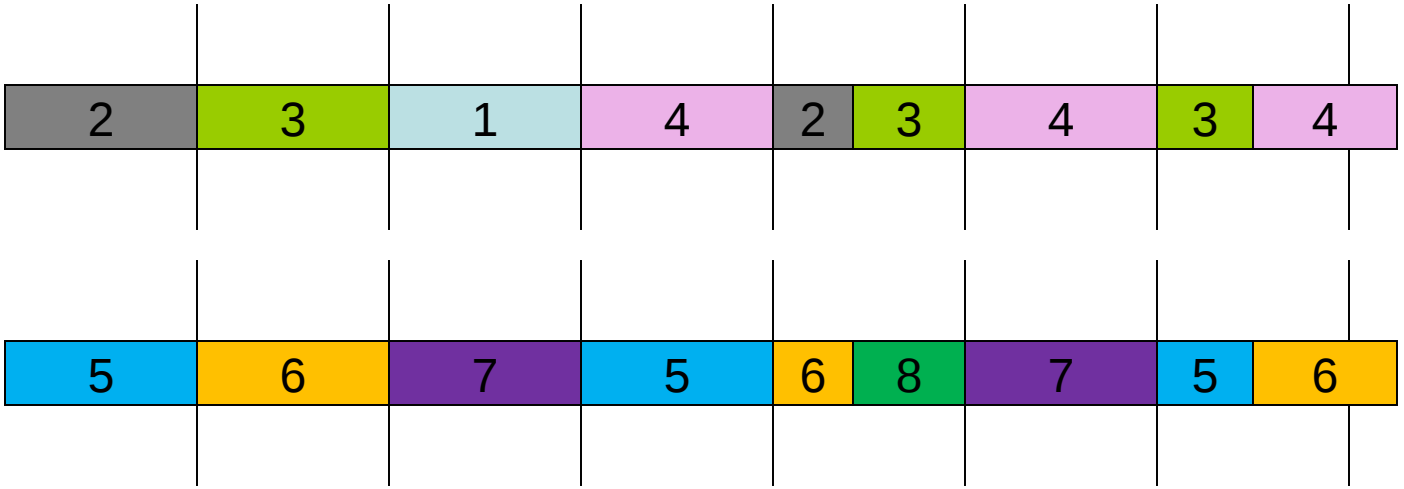
the interrupt takes the mode to the supervisor mode where the OS decides which app to run next



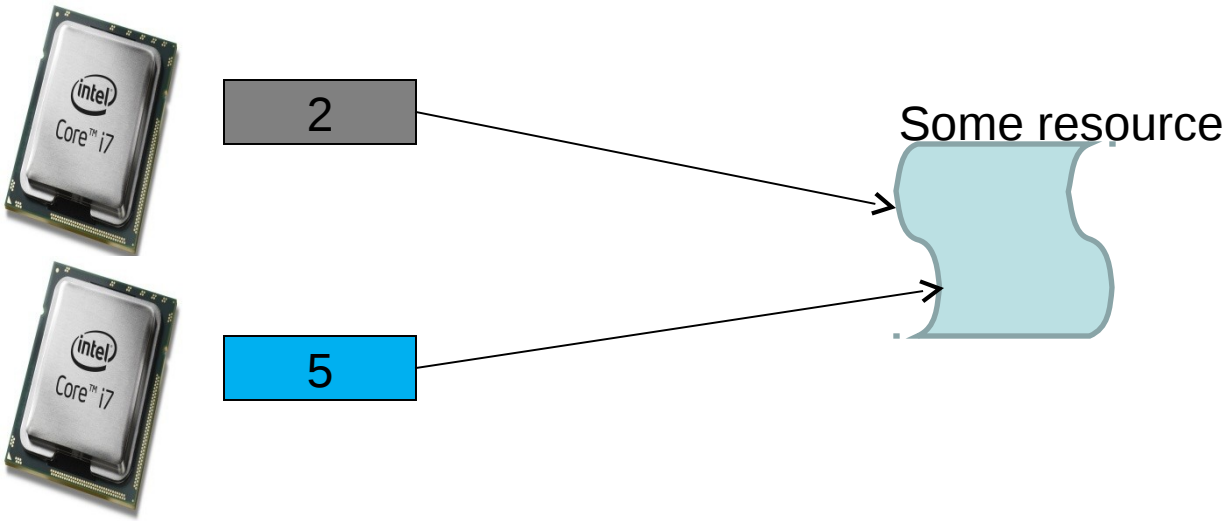
sometimes the apps can be moved across cores on instances: say the processor 1 has 4 processes ready to be run but processor 2 is free. here the OS is better off moving some process to run on another code. this could also happen when one of the cores are too hot and is about to throttle

Multiprocessors and Multithreading





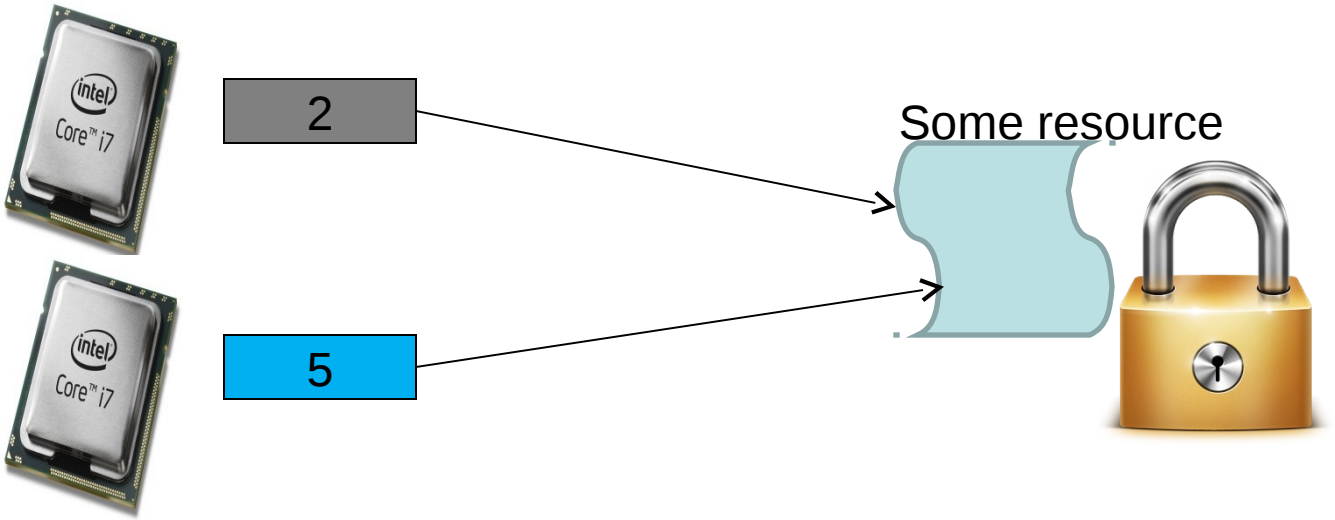
Race Conditions



- App2 and App5 want to write into some resource (like a file) simultaneously
- This results in a race condition
  - Need to synchronize between the two Apps

the idea is to enforce that only one process can access the shared resource at one time

Synchronization



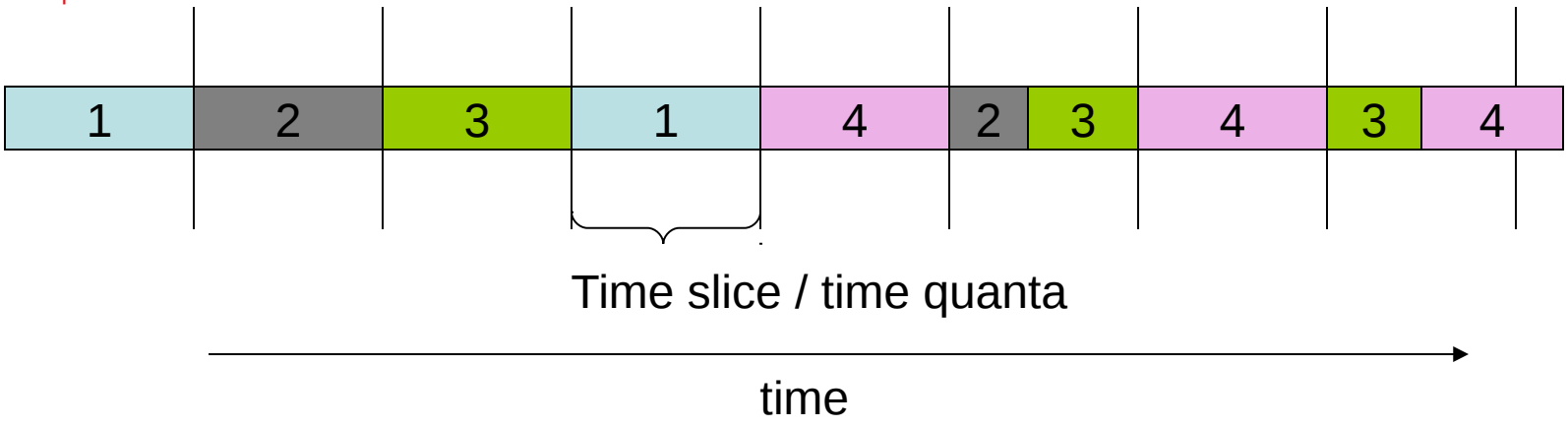
- The shared file is associated with a lock
- The lock ensures that only one App can access the resource at a time
- Sequence of Steps
  - App X locks the resource
  - App X accesses the resource, while App Y waits
  - App X unlocks the resource
  - App Y can now lock and then access the resource

when a program which has locked a resource goes on a context switch, the process might leave the resource locked itself
an example of this race would be when two resources want to put data onto the data bus -> one might overwrite the other and data inconsistency occur

Who should execute next?

- Scheduling
  - Algorithm that executes to determine which App should execute next
  - Needs to be fair
  - Needs to be able to prioritize some Apps over the others

the priority should let some processes having absolute priorities
sometimes there might be starvation - lower priority apps might not get access to the CPU at all
the os must prevent starvation and ensure fairness in access to resources



OS and Isolation

- **Why is it needed?**
  - Multiple apps execute concurrently, each app could be from a different user. Therefore needs isolation.
  - Preventing a malfunctioning app from affecting other apps

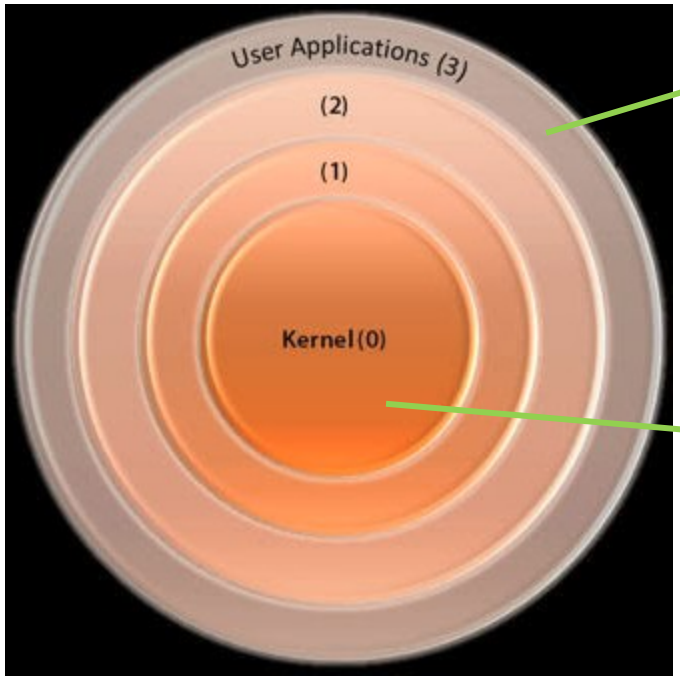
OS Isolation

- First ensure that the OS itself runs in a protected mode

the x86 arch uses rings as levels of privileges
the outer rings have lower privileges and are user level applications
the inner rings have greater privileges and can do more things than the lower ones

present day x86 arch has an SMM which is like an level -1

riscv has user, supervisor and machine modes
the machine mode has advanced privileges



Least privileged

Most privileged

the super privileged modes perform things very critical to the system - like integrity checks of the booting OS or validating updates to the booting OS

an app in the user mode must not be able to affect the OS itself or change it

Program Isolation

- Use virtual memory to ensure programs are isolated from each other
- Set page permissions
  - Execute, read only, read-write

using privilege modes protects the OS from the apps, but what protects apps from each other?
this is done by isolating the programs by providing virtual memories for each

OS and Security

- Why is it needed?
  - Defend against internal or external attacks from viruses, worms, identity theft, theft of service.
- How is it achieved?
  - Access Control
  - Passwords and Cryptography
  - Biometrics
  - Security assessment

Access Control

- Only authorized users can access files and other resources

	File 1	File 2	File 3	Program 1
Ann	own read write	read write		execute
Bob	read		read write	
Carl		read		execute read

Security Assessment

- How secure is my system?
- Can be done by
  - mathematical analysis
  - Manual / semi-automated verificiation method

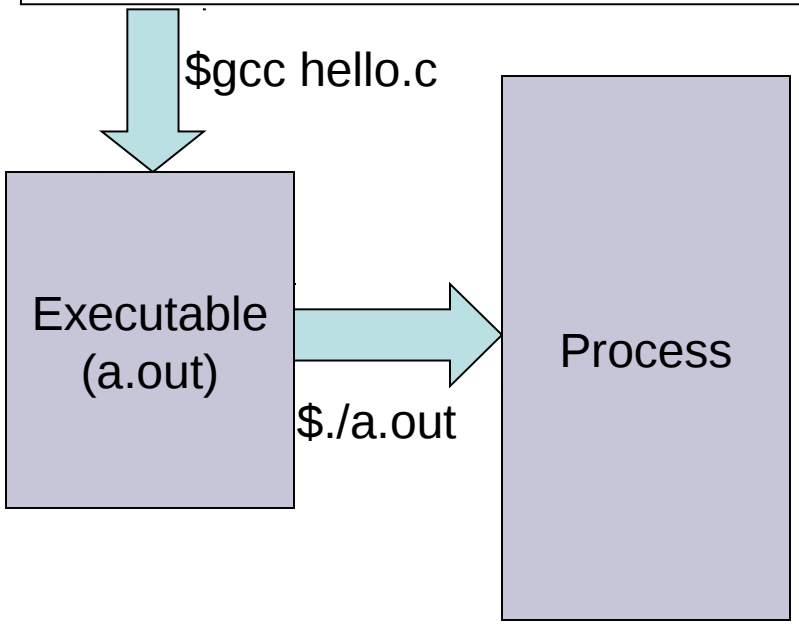
Outline

- Basics
- OS Concepts
- OS Structure

Executing Apps (Process)

```c
#include <stdio.h>

int main(){
  char str[] = "Hello World\n";
  printf("%s", str);
}
```



this metadata is added by the OS when an executable is run

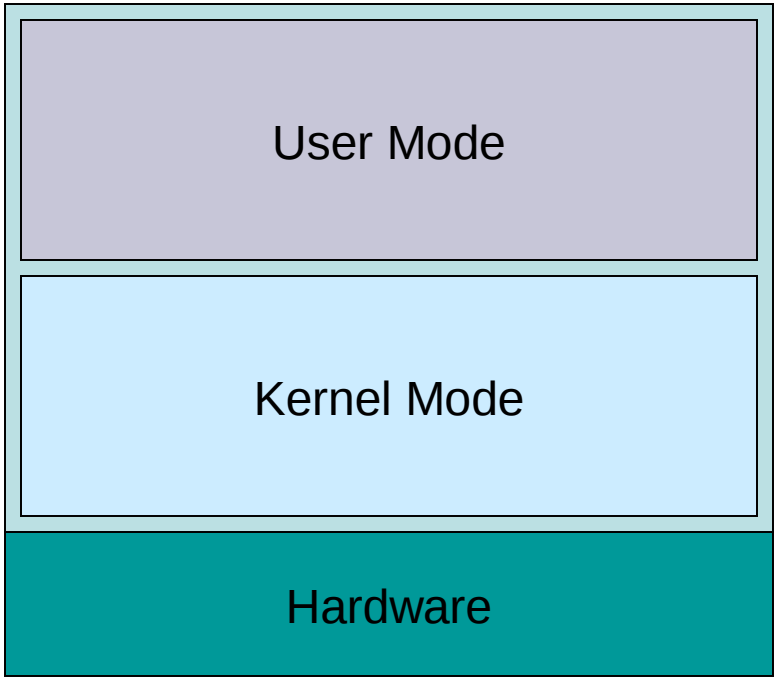
- Process
  - A program in execution
  - Comprises of
    - Executable instructions
    - Stack function calls, local variables etc
    - Heap dynamic memory allocated by malloc etc
    - State
  - State contains : registers, list of open files, related processes, etc.

where is the OS involved here?
1. ./a.out tells the OS the process must be run -> the OS creates env to run
2. when printf is called the OS sends the message to the file descriptor

a rule of thumb is that whenever hardware needs to get involved the OS gets involved as well

Operating Modes

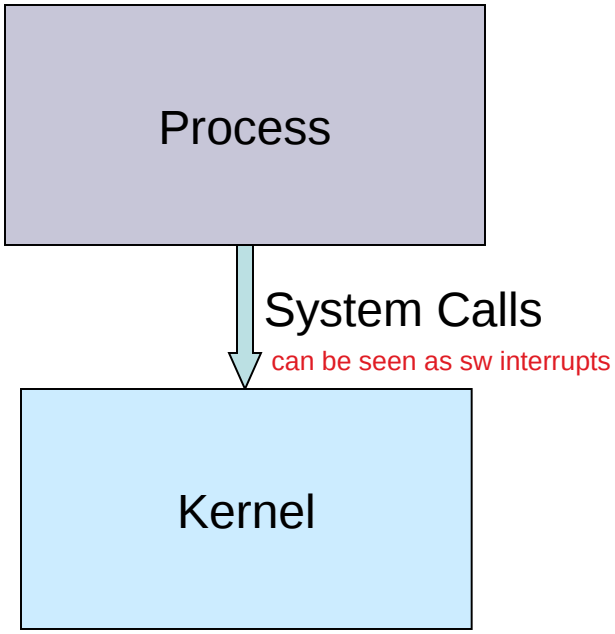
- User Mode
  - Where processes run
  - Restricted access to resources
  - Restricted capabilities
- Kernel mode a.k.a. Privileged mode
  - Where the OS runs
  - Privileged (can do anything)



the machine mode is also used during booting of the system and runs the integrity checks, then moves to sv mode where the kernel runs
when the system boots up, the cpu runs at the kernel layer first and then runs the init process which spawns the other necessities
these include the network card interface and all of that

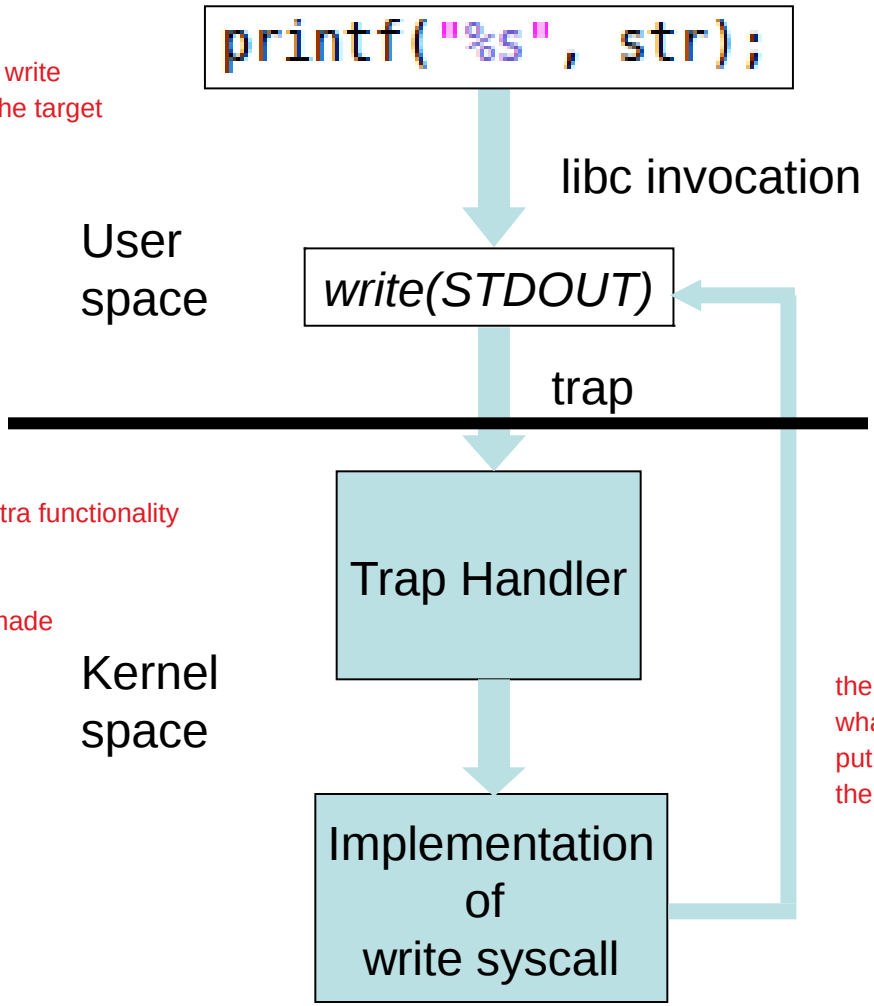
Communicating with the OS (System Calls)

- System call invokes a function in the kernel using a Trap
- This causes
  - Processor to shift from user mode to privileged mode
- On completion of the system call, the execution gets transferred back to the user mode process



Example (write system call)

the write system call can be used to write
a stream of bytes to be written into the target
file descriptor number



in riscv traps are implemented with ecalls
ecalls are enviroment calls (wtf?)

the ecall moves the processor to the machine mode. after this, it delegates the work to the OS in the supervisor mode which implements further processes

the trap handler has a bunch of extra functionality
all system calls invoke a trap
the trap handler saves the context
it also finds out what syscall was made
then it invokes the implementation
of the said system call

the writing on to the monitor is same as what is done on all memory mapped devices
put it on the video memory and move on
the monitor handles further

System Call vs Procedure Call

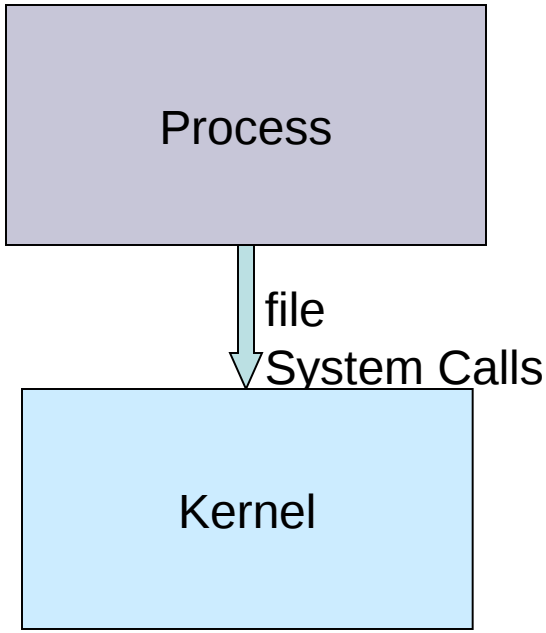
System Call	Procedure Call
Uses a TRAP instruction (such as int 0x80)	Uses a CALL instruction
System shifts from user space to kernel space	Stays in user space (or kernel space) … no shift
TRAP always jumps to a fixed addess (depending on the architecture)	Re-locatable address

System Call Interfaces

- System calls provide users with interfaces into the OS.
- What set of system calls should an OS support?
  - Offer sophisticated features
  - But yet be simple and abstract whatever is necessary
  - General design goal : rely on a few mechanisms that can be combined to provide generality

Files

- Data persistent across reboot
- What should the file system calls expose?
  - Open a file, read/write file, creation date, permissions, etc.
  - More sophisticated options like seeking into a file, linking, etc.
- What should the file system calls hide?
  - Details about the storage media.
  - Exact locations in the storage media.

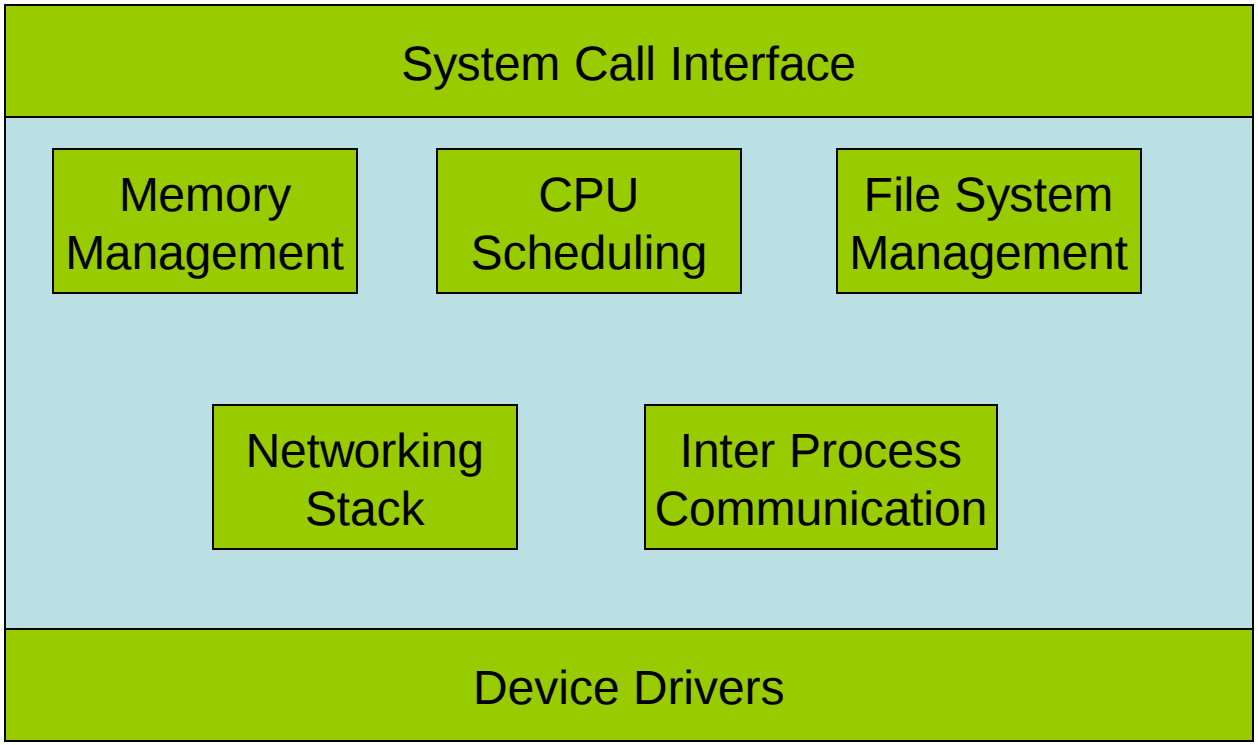


Outline

- Basics
- OS Concepts
- **OS Structure**

What goes into an OS?

the sci is used to talk to the user applications

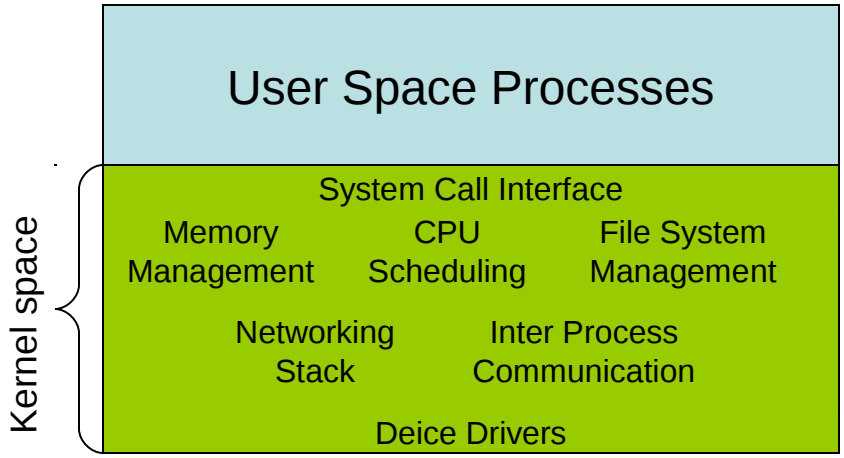


the device drivers let the OS talk to hardware underlying the system

the networking stack is used to handle network connections and requests like a web browser

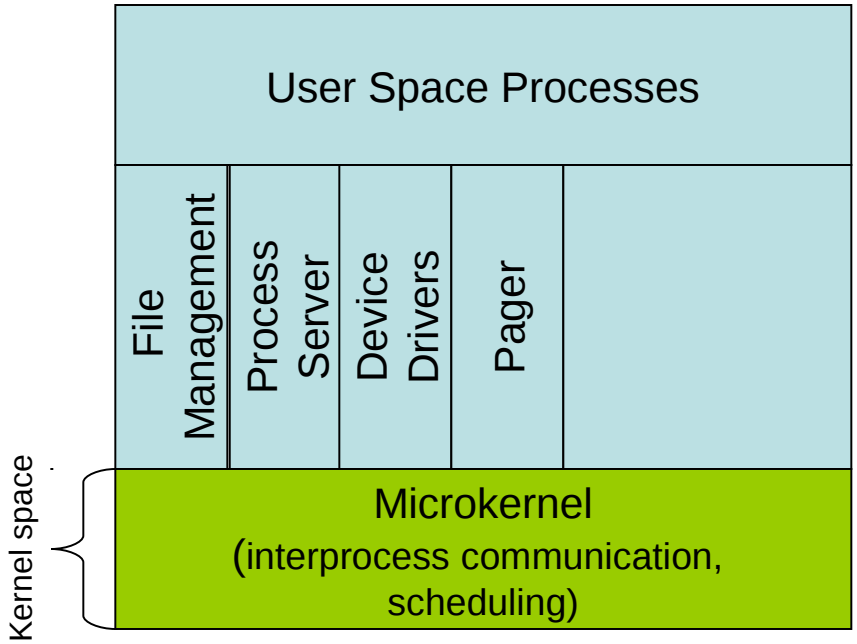
because processes are isolated from each other, the only way for them to talk to each other would need the use of IPC

OS Structure : Monolithic Structure



- Linux, MS-DOS, xv6
- All components of OS in kernel space
- Cons : Large size, difficult to maintain, likely to have more bugs, difficult to verify
- Pros : direct communication between modules in the kernel by procedure calls

OS Structure : Microkernel



Eg. QNX and L4

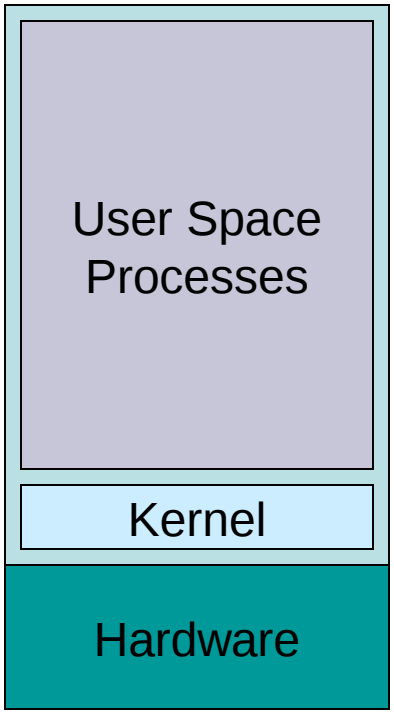
- Highly modular.
  - Every component has its own space.
  - Interactions between components strictly through well defined interfaces (no backdoors)
- Kernel has basic inter process communication and scheduling
  - Everything else in user space.
  - Ideally kernel is so small that it fits the first level cache

in this approach, any comms between two segments needs to go through the microkernel -> may lead to large overheads
microkernel offers more security because just compromising one unit does not compromise the entire kernel -> because the kernel is small
it is likely to have far fewer bugs than for a monolithic kernel

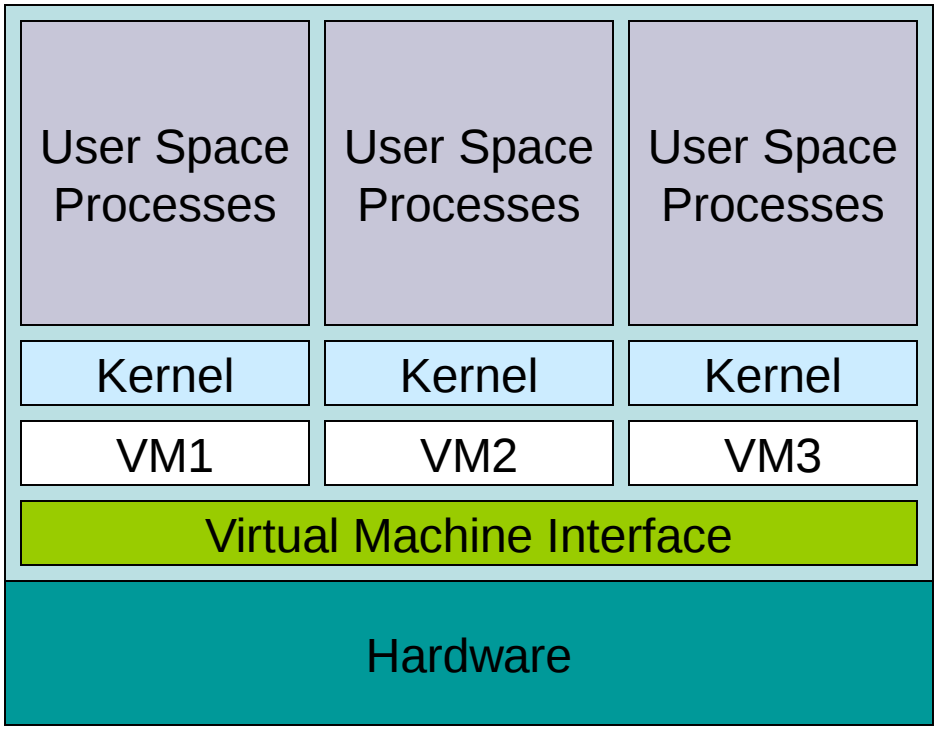
Monolithic vs Microkernels

	Monolithic	**Microkernel**
Inter process communication	Signals, sockets	Message queues
Memory management	Everything in kernel space (allocation strategies, page replacement algorithms,)	Memory management in user space, kernel controls only user rghts
Stability	Kernel more ‘crashable’ because of large code size	Smaller code size ensures kernel crashes are less likely
I/O Communication (Interrupts)	By device drivers in kernel space. Request from hardware handled by interrupts in kernel	Requests from hardware converted to messages directed to user processes
Extendibility	Adding new features requires rebuilding the entire kernel	The micro kernel can be base of an embedded system or of a server
Speed	Fast (Less communication between modules)	Slow (Everything is a message)

Virtual Machines



No virtual Machines



With virtual Machines

for next class

- Please revise / learn
  - memory management in Intel i386 (especially GDTs, page tables, and page size extensions)

    (http://www.logix.cz/michal/doc/i386/chp05-00.htm)
  - Real mode and protected mode in Intel i386

    (Shifting from real mode to protected mode)