# Operating Systems Overview

Chester Rebeiro
IIT Madras

# Outline

- Basics
- OS Concepts
- OS Structure

# What is the OS used for?

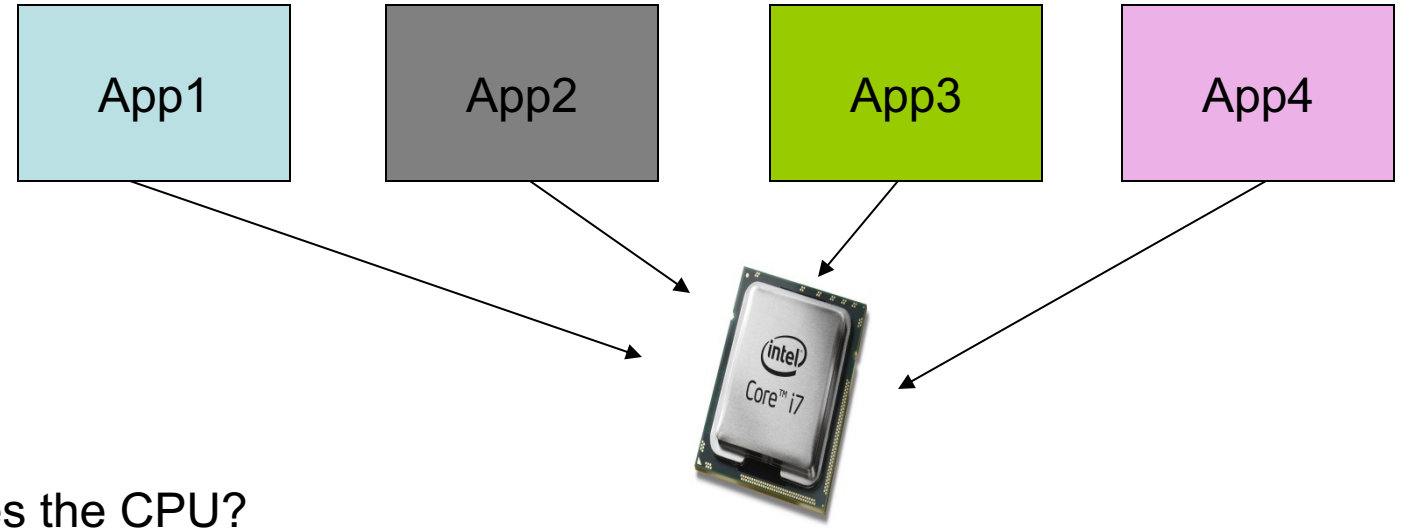
- Hardware Abstraction

  *turns hardware into something that applications can use*

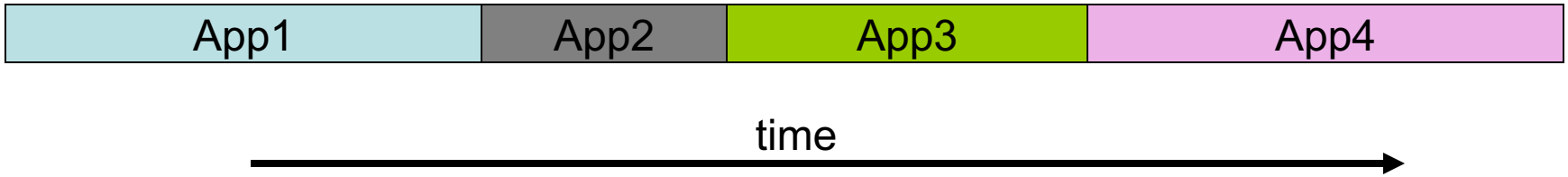
- Resource Management

  *manage system's resources*

# Sharing the CPU

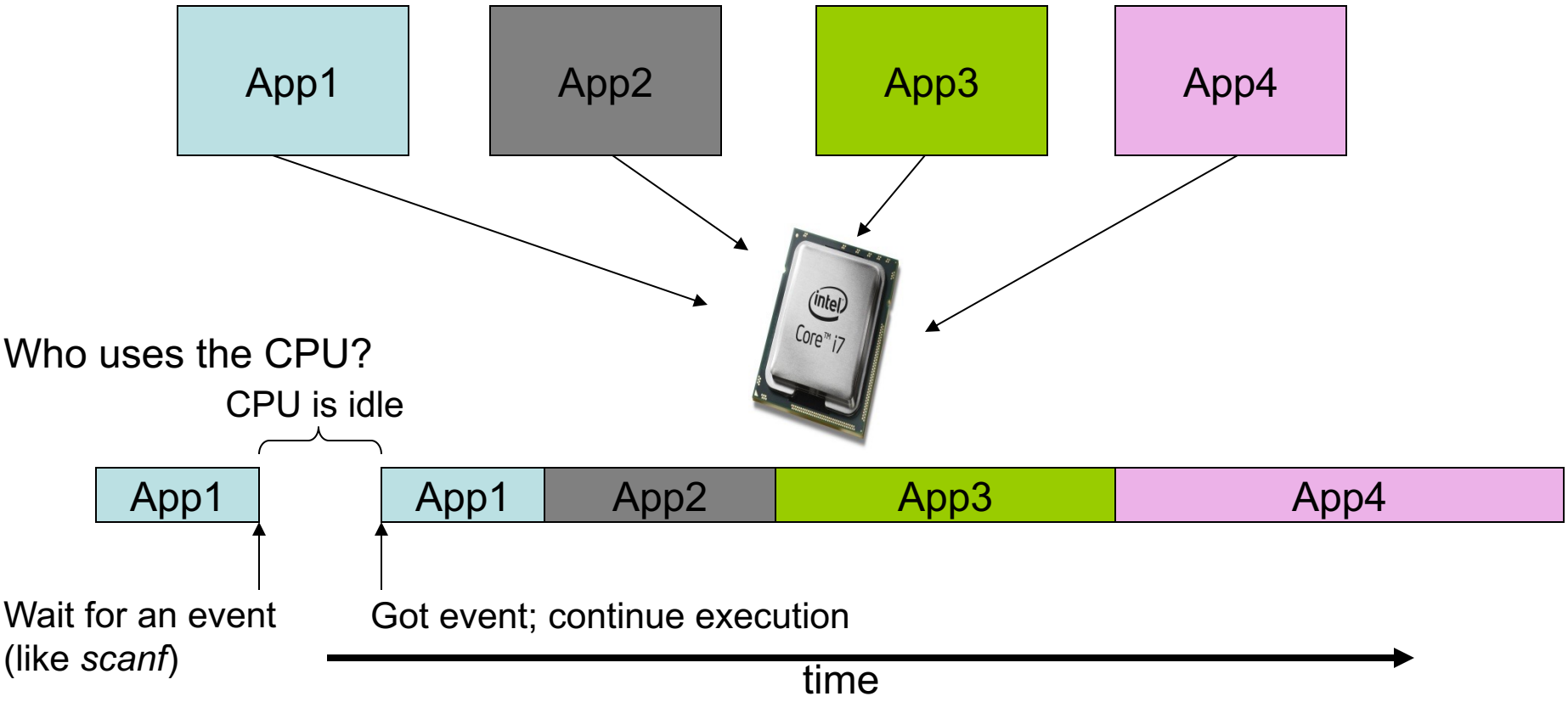



Who uses the CPU?



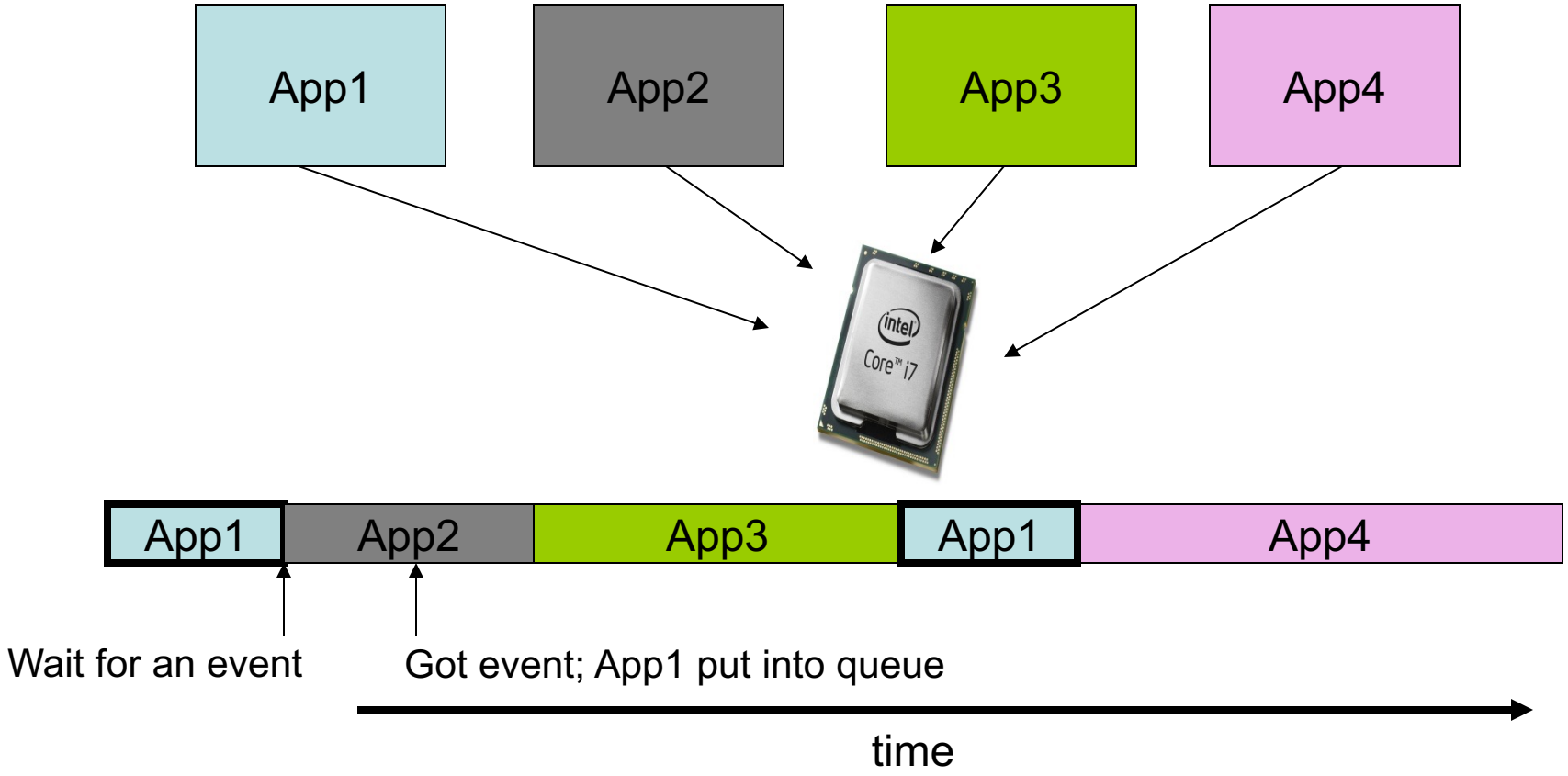

When one app completes the next starts

# Idle CPU Cycles



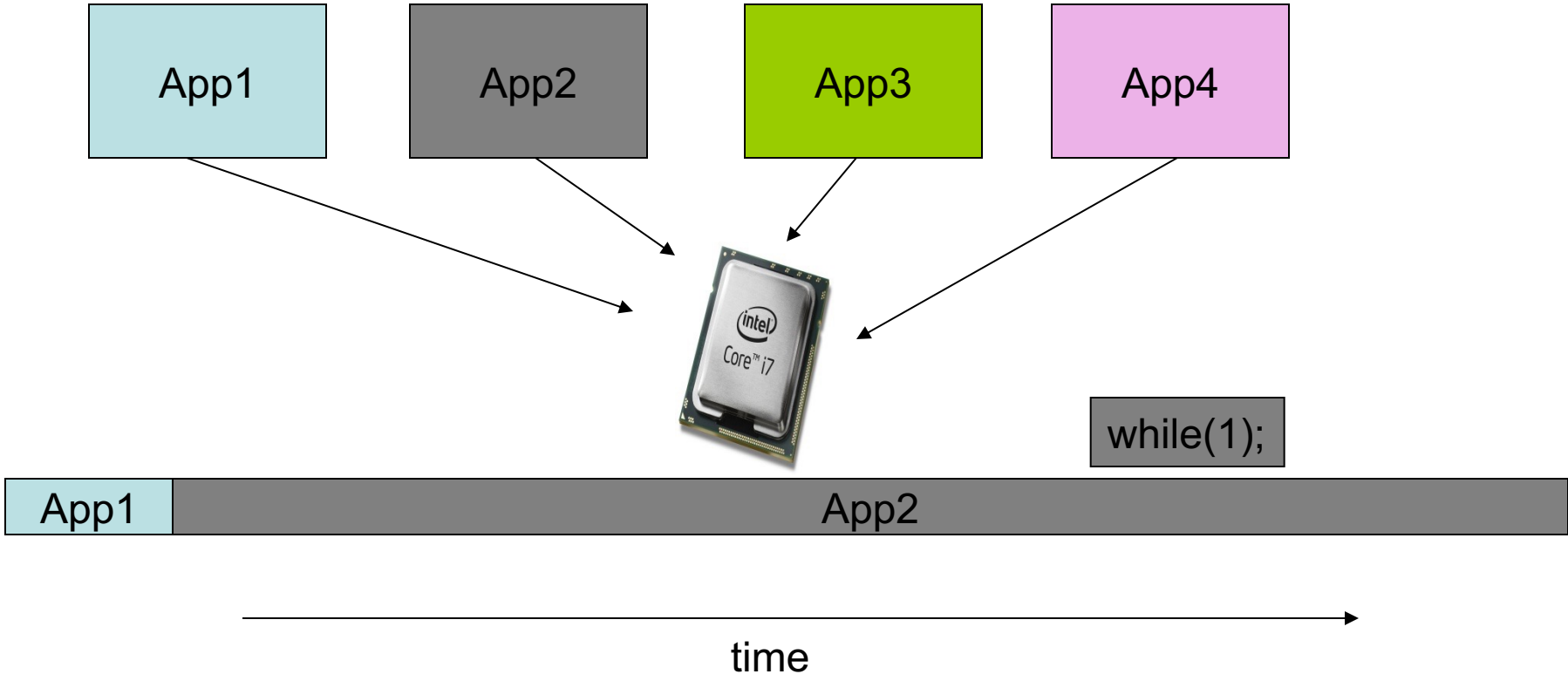

CPU is idle when executing app waits for an event.
Reduced performance.

# When OS supports Multiprogramming




When CPU idle, switch to another app

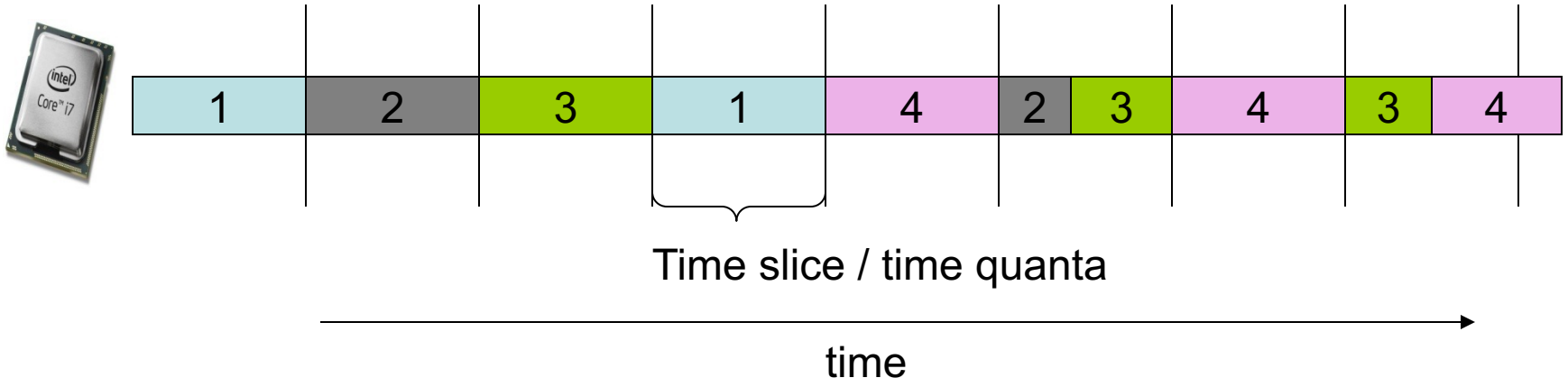
# Multiprogramming could cause starvation




One app can hang the entire system

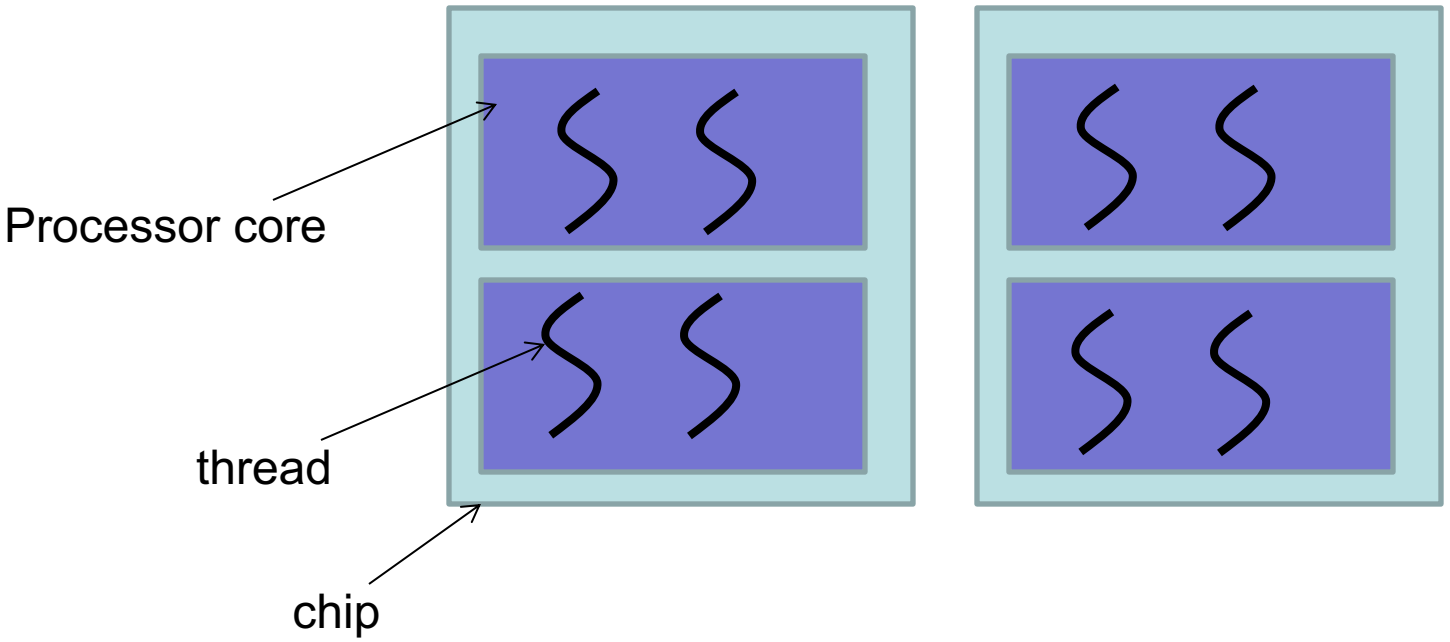
# When OS supports Time Sharing (Multitasking)

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



# Multiprocessors

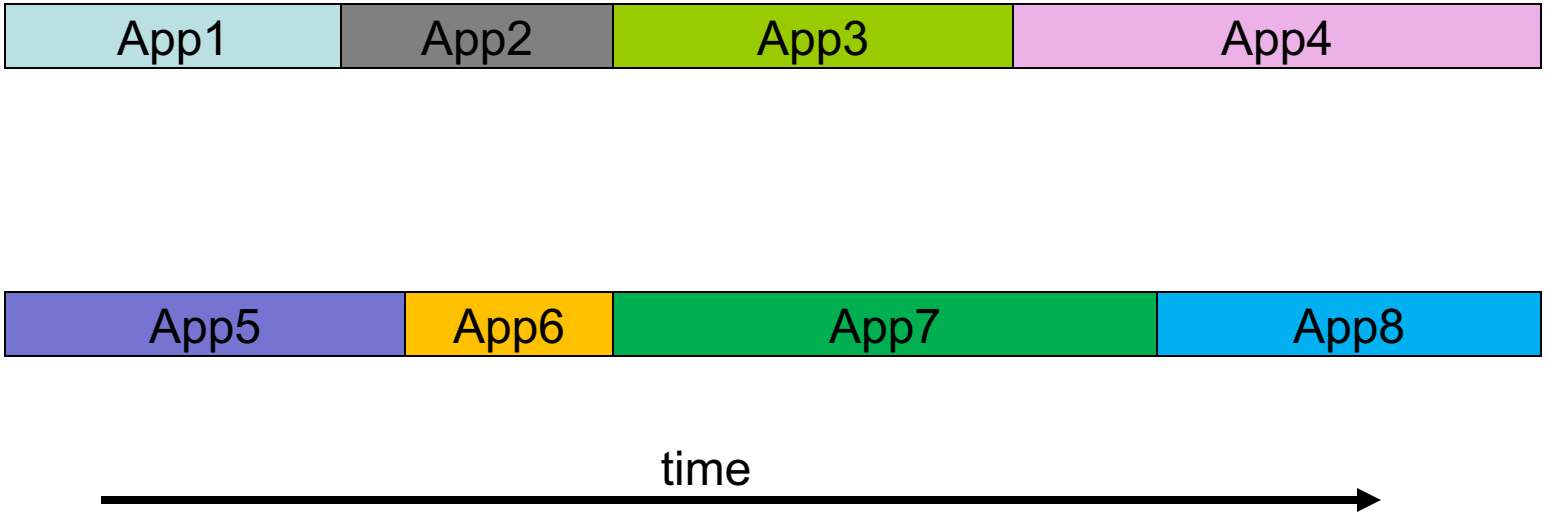
- Multiple processors chips in a single system
- Multiple cores in a single chip
- Multiple threads in a single core



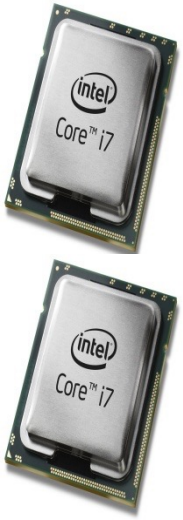
# Multiprocessors

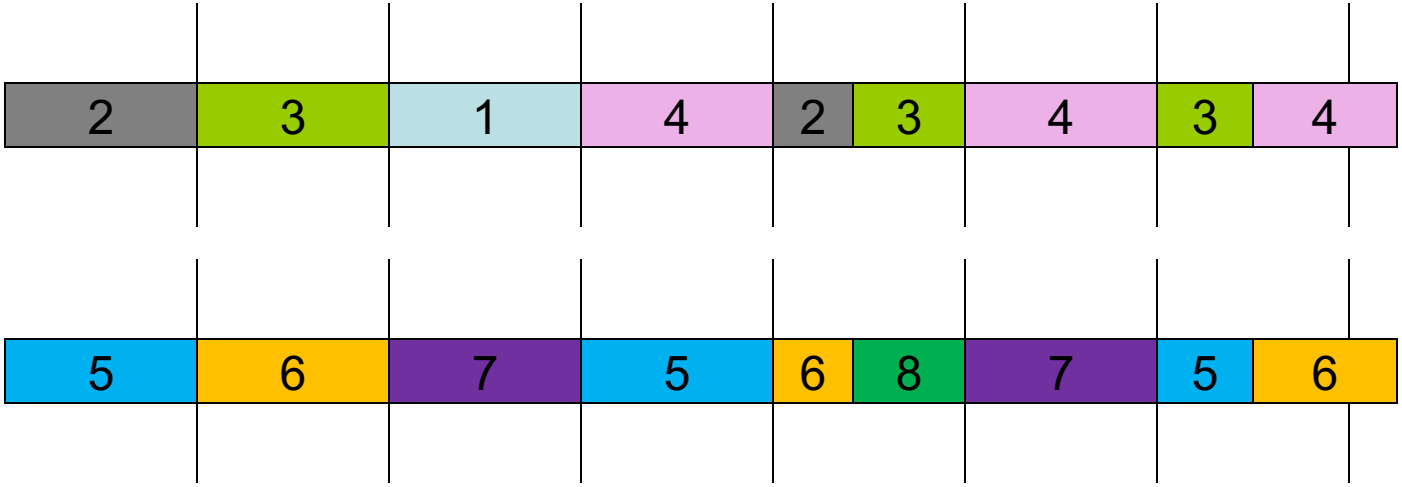
- Each processor can execute an app independent of the others



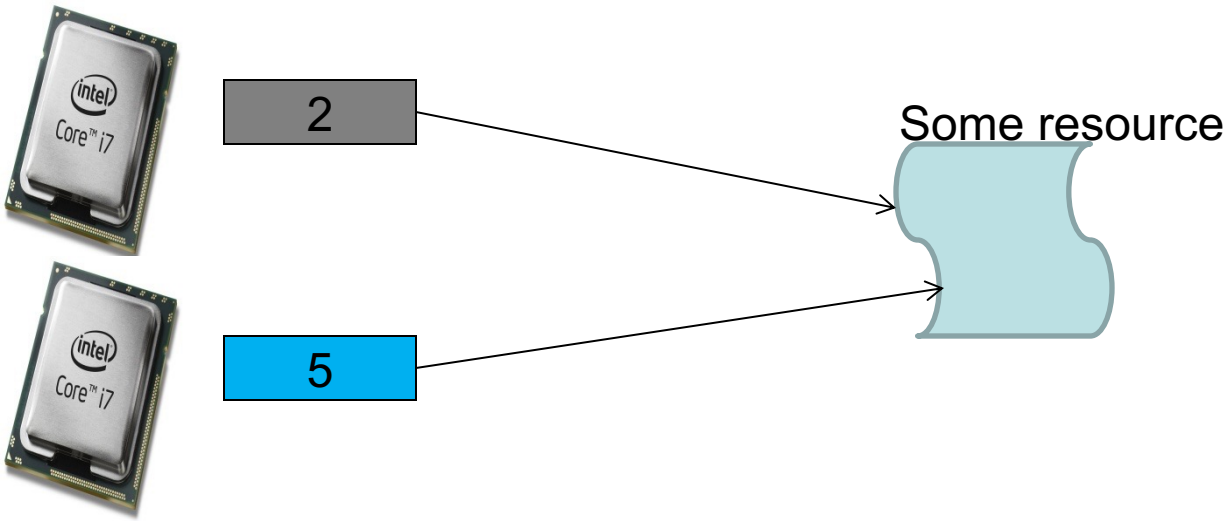


# Multiprocessors and Multithreading



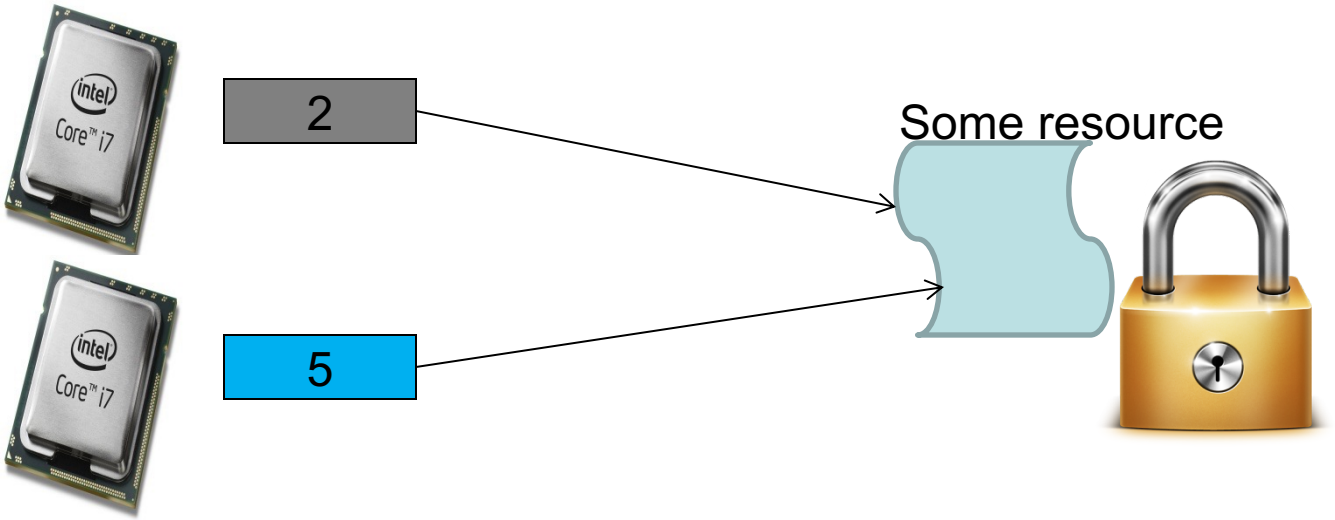



# Race Conditions




- 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

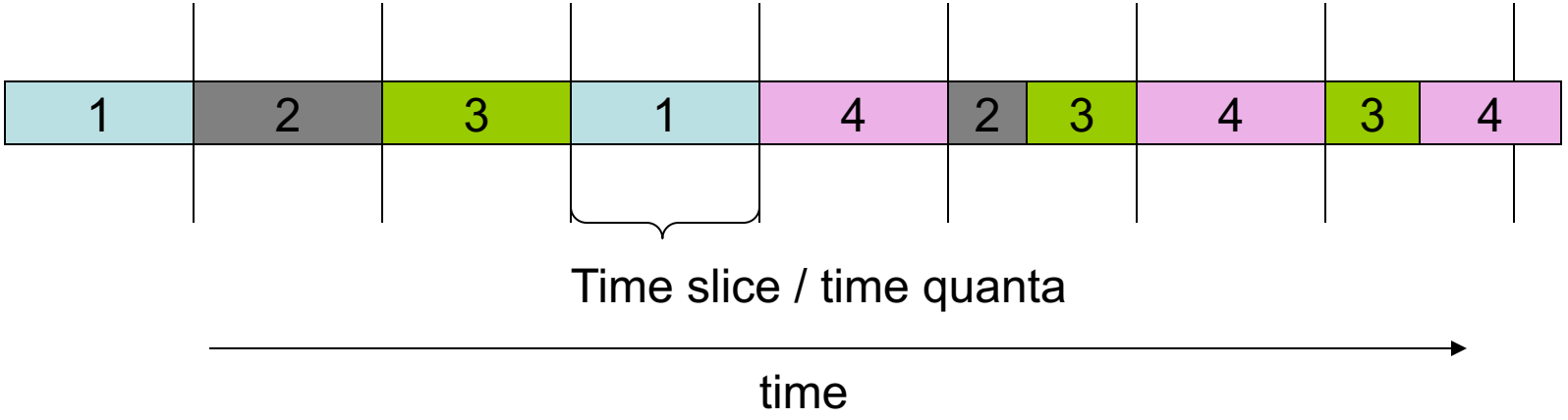
# Synchronization




- 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

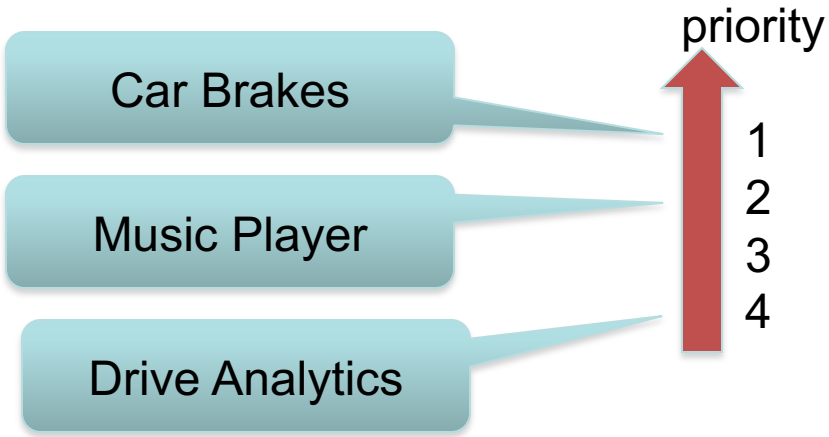
# Who should execute next?

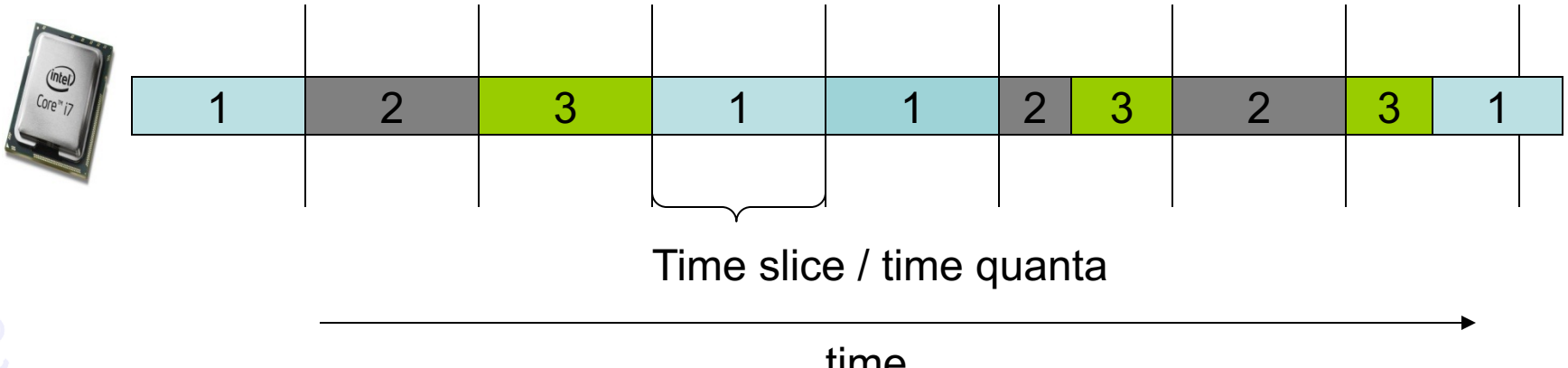
- Scheduling
  - Algorithm that executes to determine which App should execute next
  - Needs to be fair
  - Needs to be able to prioritize some Apps over the others



# Priority Based Systems

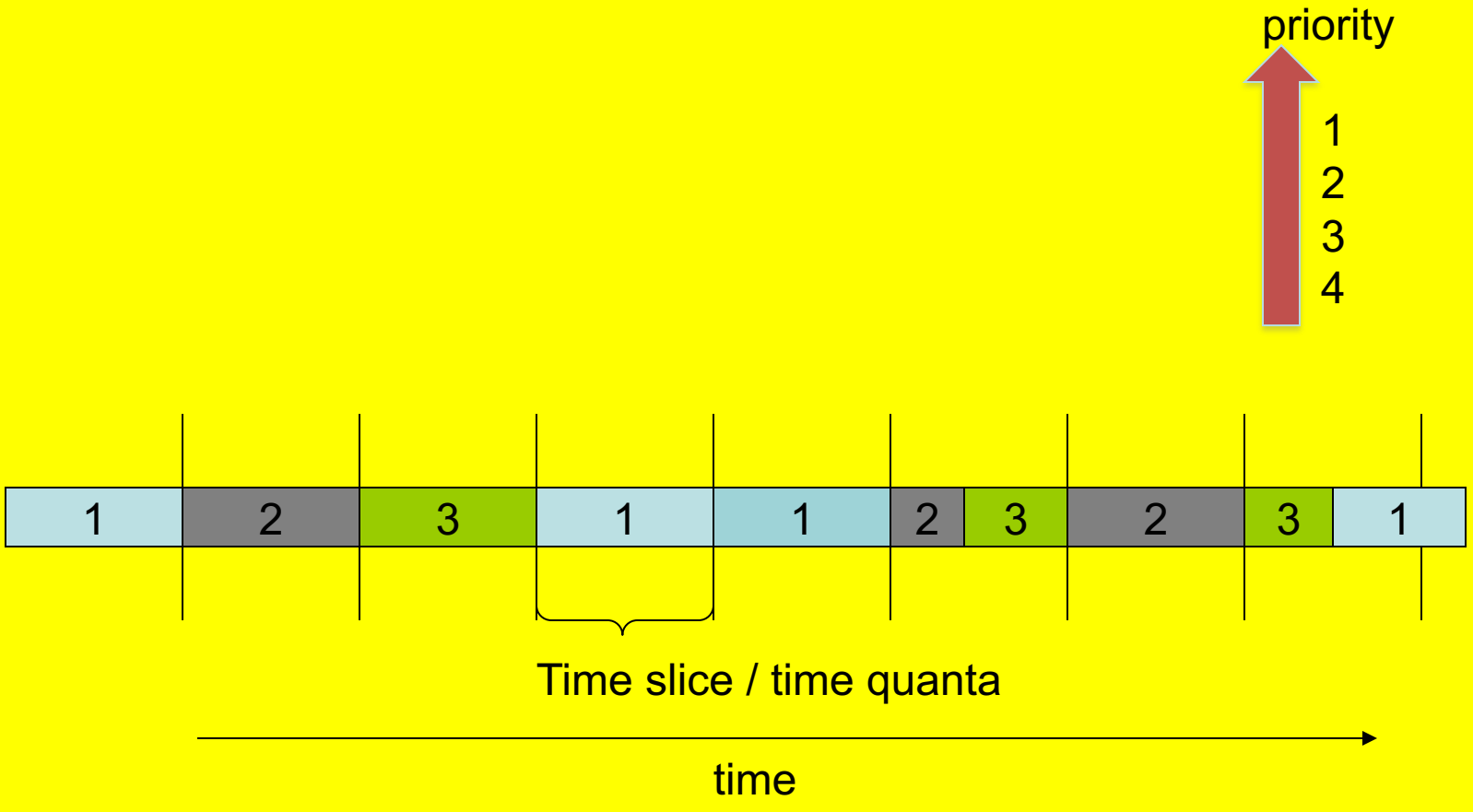
- Based on user choice or a heuristics, some apps are given higher priority compared to others.



# L3_1: Drawback of priority based systems?

- What is the potential issue(s) that can crop up with priority based systems?

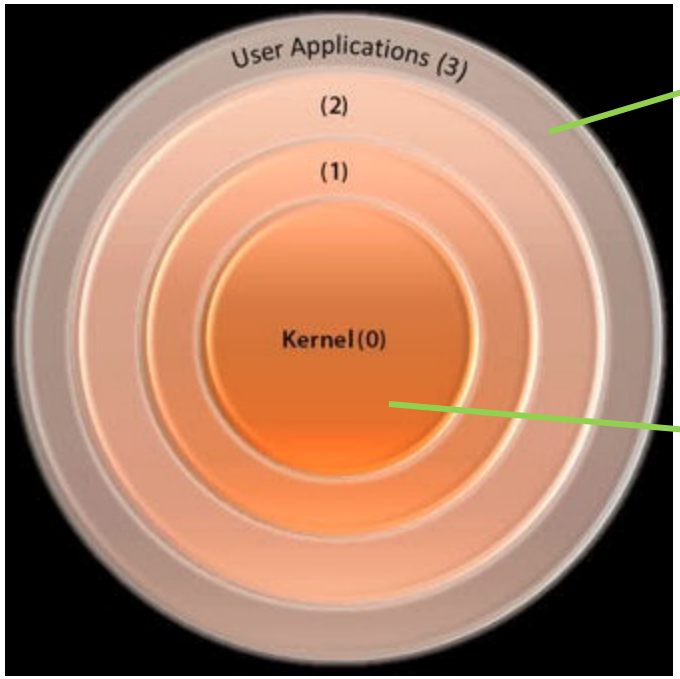
# OS and Isolation

- **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

- Ensure that the OS itself runs in a protected mode

X86 model

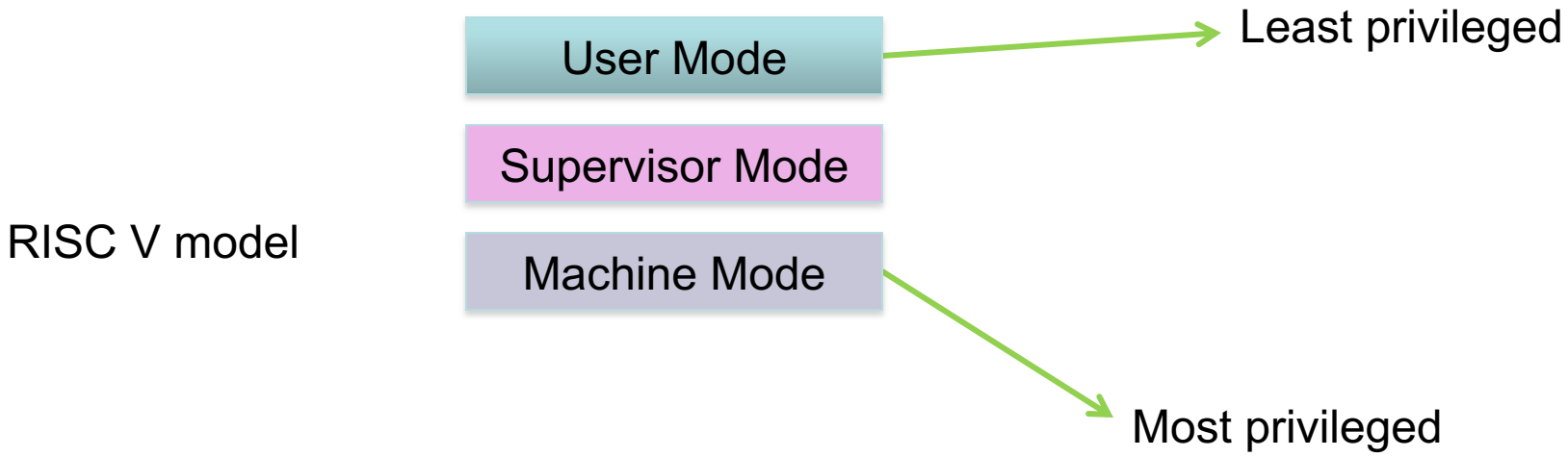



Least privileged

Most privileged

# OS Isolation

- Ensure that the OS itself runs in a protected mode



# Program Isolation

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

# 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

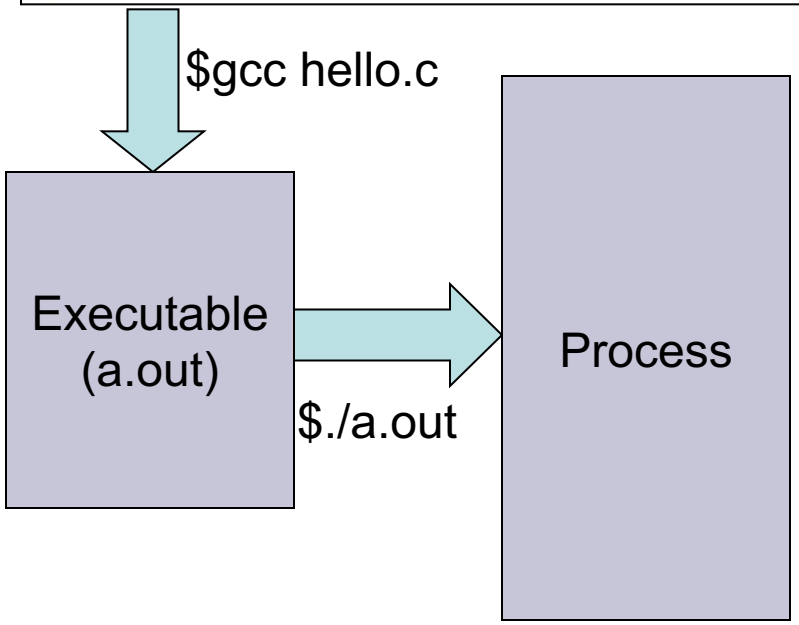
| | File 1 | File 2 | File 3 | Program 1 |
|---|---|---|---|---|
| **Ann** | own<br>read<br>write | read<br>write | | execute |
| **Bob** | read | | read<br>write | |
| **Carl** | | read | | execute<br>read |

# Outline

- Basics
- OS Concepts
- OS Structure

# Executing Apps (Process)

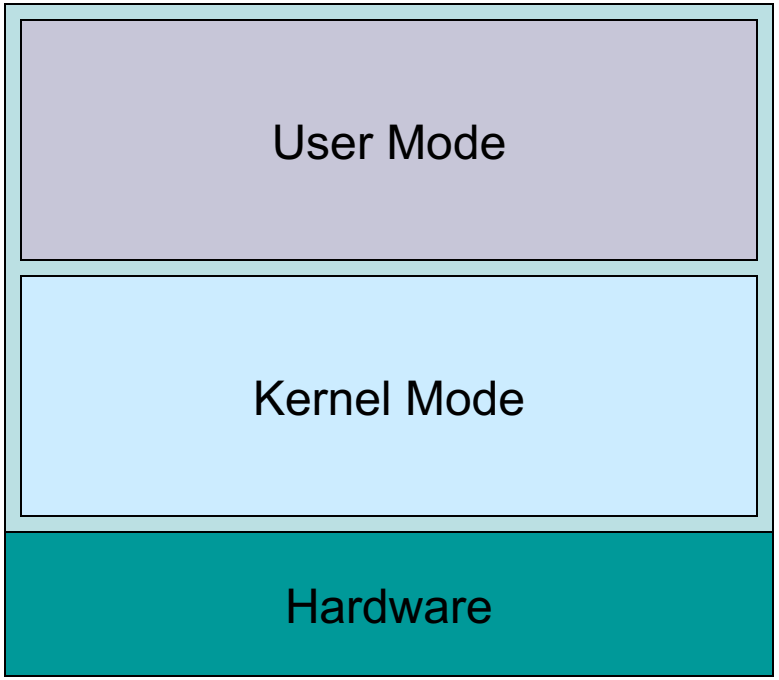
```c
#include <stdio.h>

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



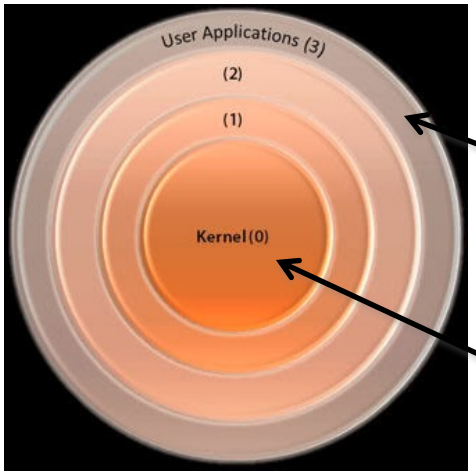

- Process
  - A program in execution
  - Comprises of
    - Executable instructions
    - Stack
    - Heap
    - State
  - State contains : registers, list of open files, related processes, etc.

# Operating Modes

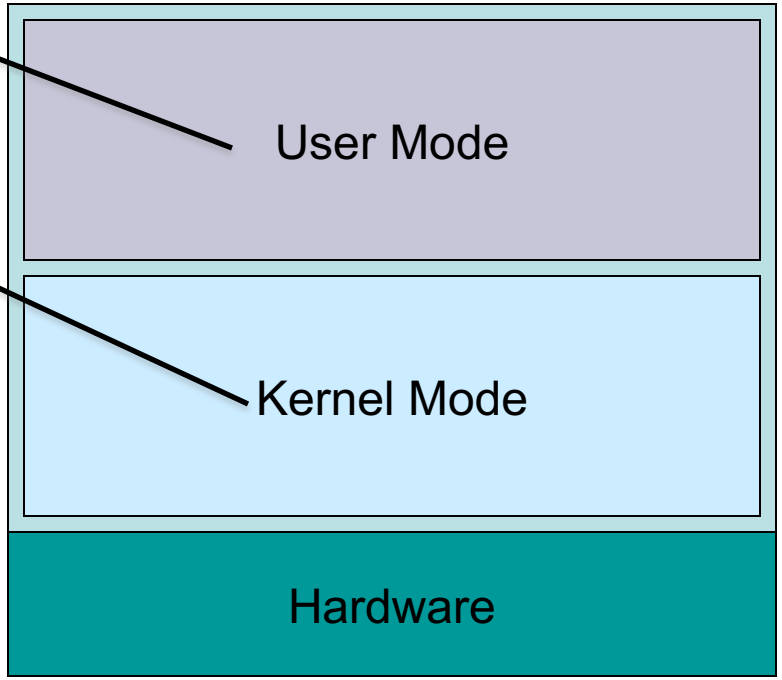
- User Mode
  - Where processes run
  - Restricted access to resources
  - Restricted capabilities
- Kernel mode a.k.a. Privileged mode
  - Where the OS runs
  - Privileged (can do anything)



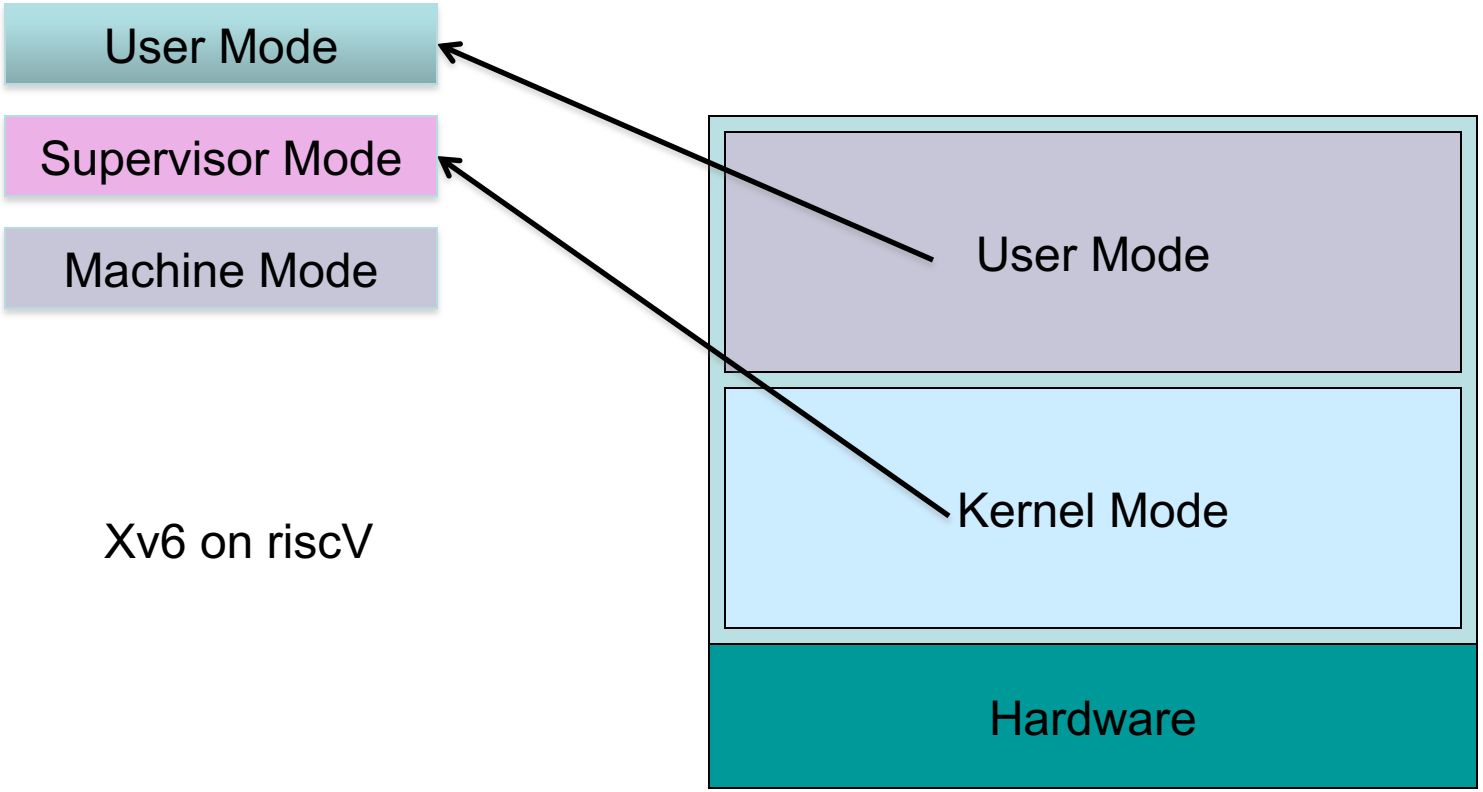
# Operating Modes




Linux on x86



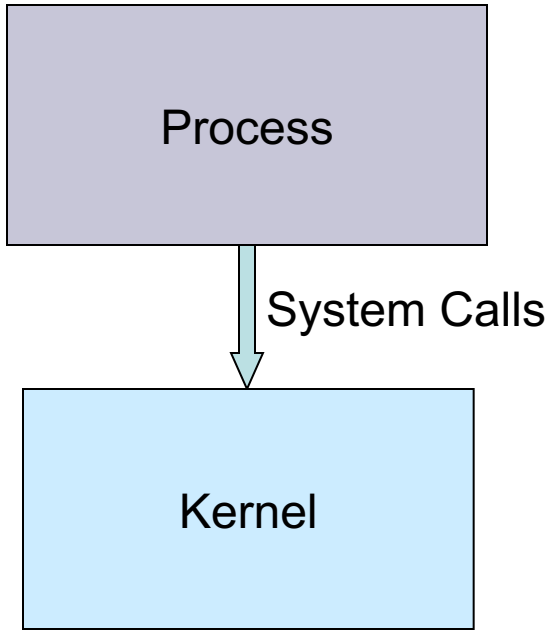
# Operating Modes



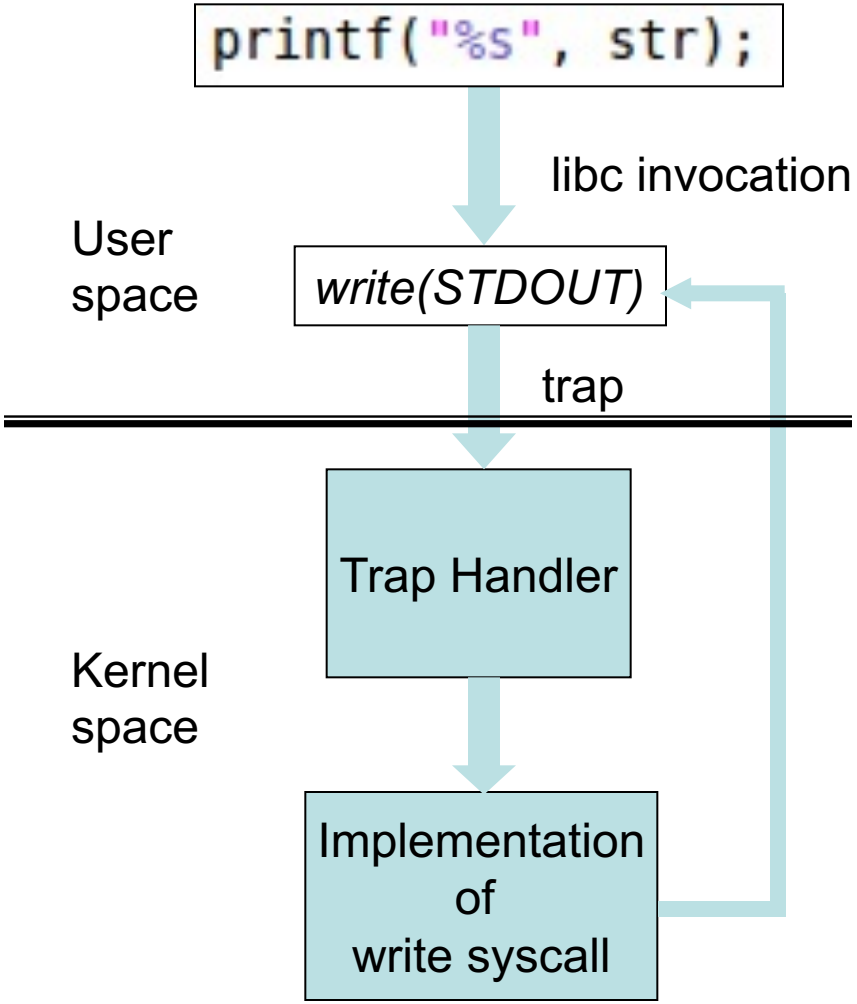

Xv6 on riscV

# Communicating with the OS (System Calls)

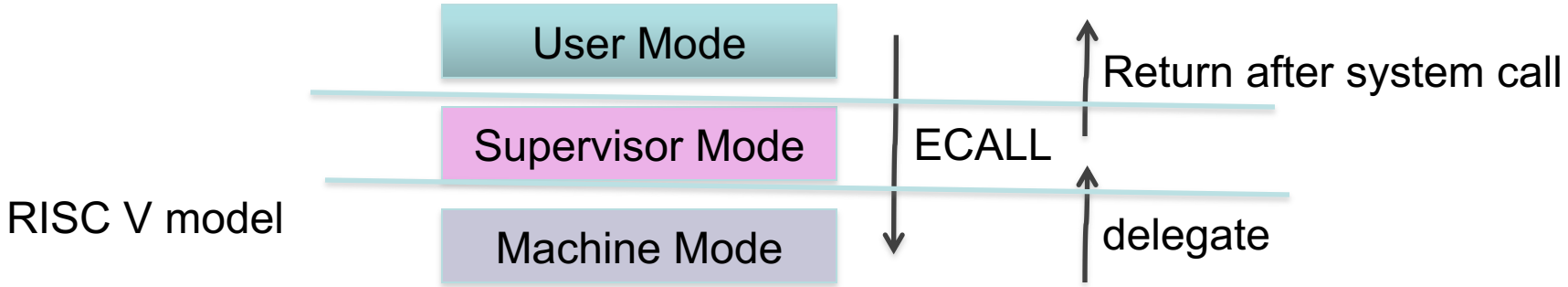
- System call invokes a function in the kernel using a Trap
- This causes
  - Processor to shift from user mode to privileged mode
- On completion of the system call, the execution gets transferred back to the user mode process



# Example (write system call)



# System calls in riscV




ECALL stores the current mode in a register called mstatus
It also stores the current program counter in register EPC
(exception program counter)

# System Call vs Procedure Call

| System Call | Procedure Call |
| --- | --- |
| Uses a TRAP instruction<br>(such as int 0x80 in x86) or<br>ECALL in RISC V | Uses a CALL instruction |
| System shifts from user space to kernel space | Stays in user space (or kernel space) |
| TRAP always jumps to a fixed addess (depending on the architecture) | Re-locatable address |

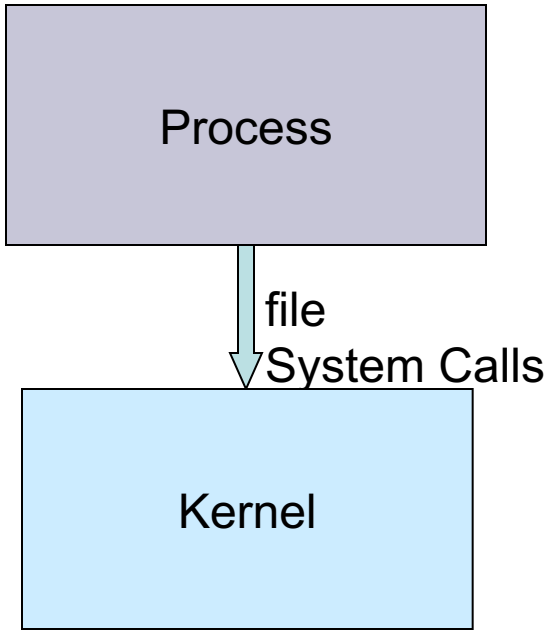
# System Call Interfaces

- 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

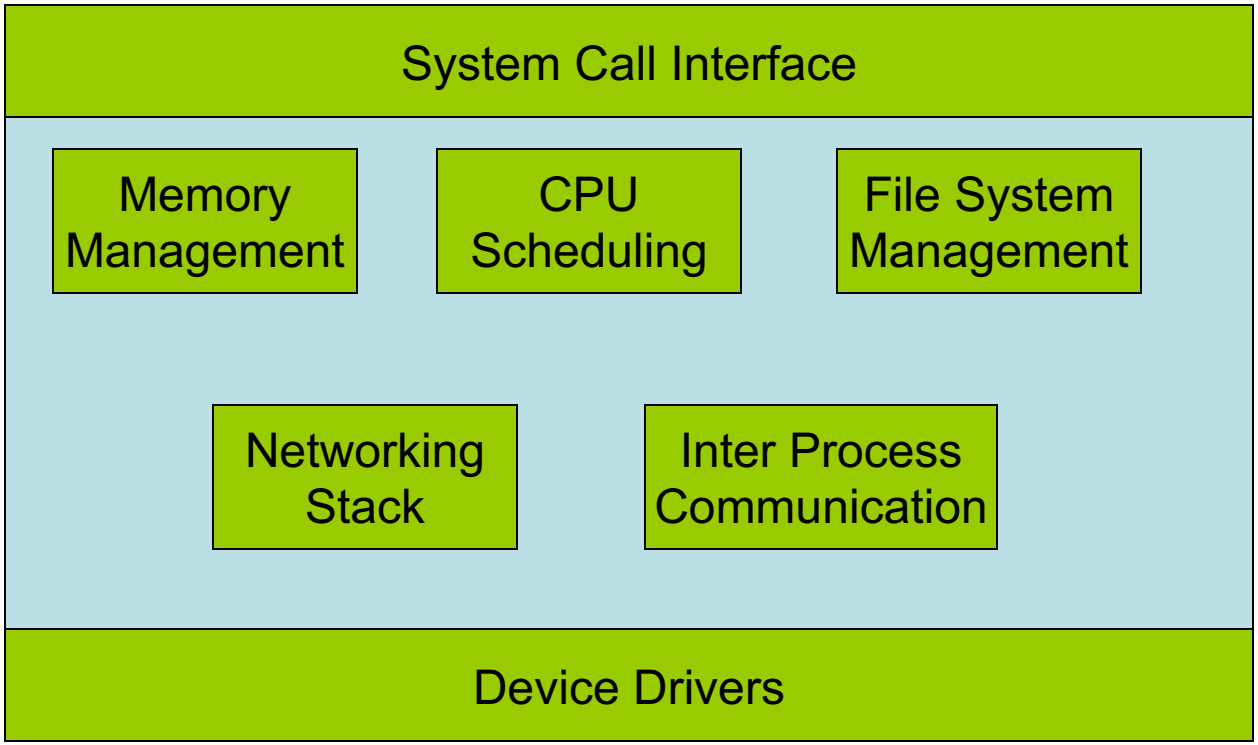
- Data persistent across reboot
- What should the file system calls expose?
  - Open a file, read/write file, creation date, permissions, etc.
  - More sophisticated options like seeking into a file, linking, etc.
- What should the file system calls hide?
  - Details about the storage media.
  - Exact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



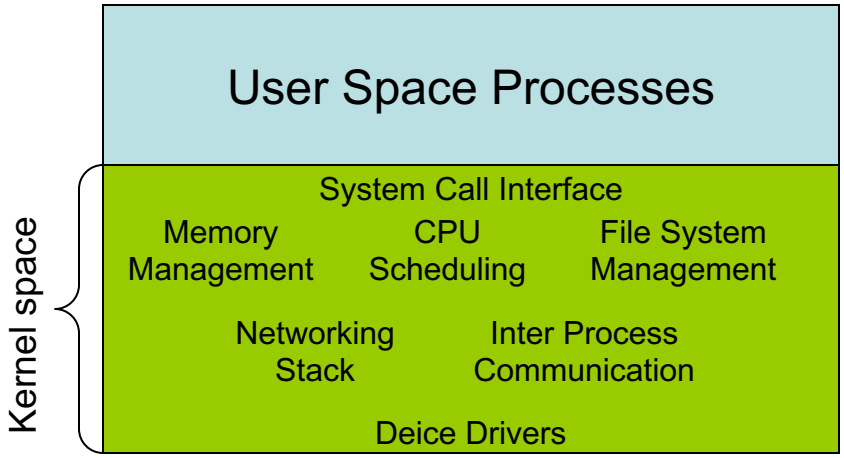
# Outline

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

# What goes into an OS?

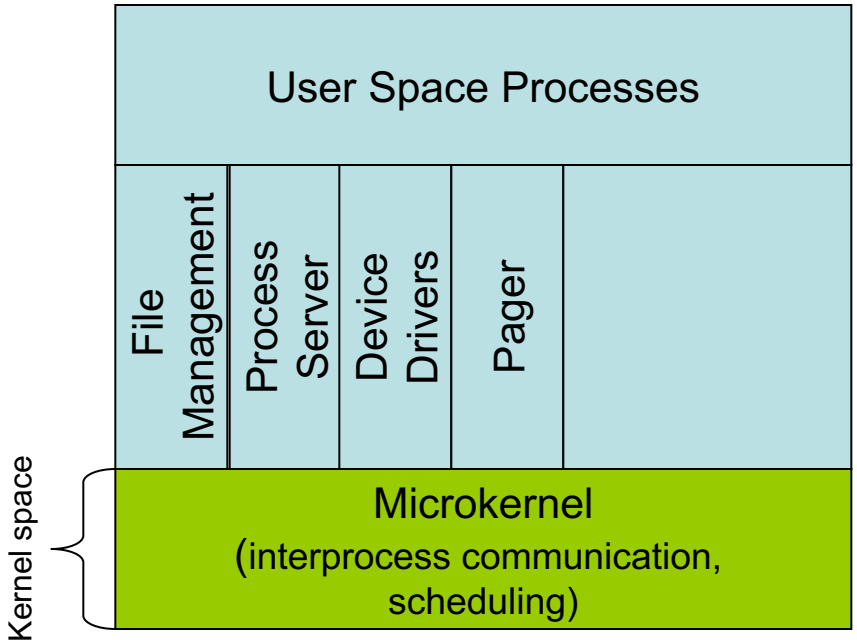


# OS Structure : Monolithic Structure




- 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



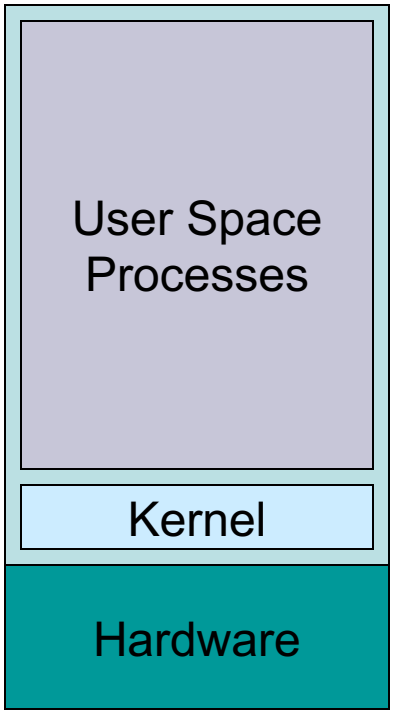

Eg. QNX and L4

- 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

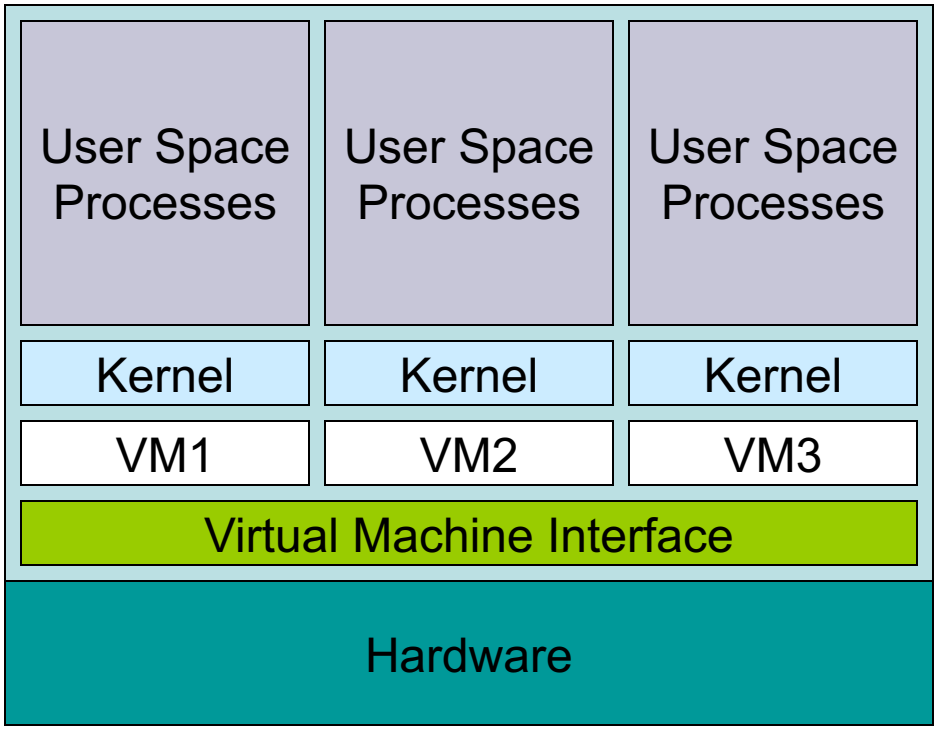
# Monolithic vs Microkernels

| | **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