CS 477:

Advanced Operating Systems

OS Design & Organization

Focus of today's lecture

- Course logistics and policy
- A brief primer on operating system
- Overview of OS design / architecture

General information

- Quick primer → Operating Systems: Three Easy Pieces
- Set of optional research papers for every week
- Course organization: Lecture, lab, exercises, project, written midterm and finals
- Staff email: cs477-staff@groupes.epfl.ch
- Instructors:

  Sanidhya Kashyap, Yujie Ren
- Current TAs:

  Yueyang Pan, Kumar Kartikeya Dwivedi, Lucas Cendez Lopes

Topics planned

- OS architecture and design
- Processes and Threads
- Scheduling
- Virtual Memory
- Storage stack
- Concurrency
- Virtualization
- Security

Mon	Tue	Wed	Thur	Fri	Sat	Sun
Sep 9	Sep 10 **LEC 1:** OS organization	Sep 11 **Lab 0**: : eBPF tutorial	Sep 12	Sep 13	Sep 14	Sep 15
Sep 16	Sep 17	Sep 18	Sep 19	Sep 20	Sep 21	Sep 22
Sep 23	Sep 24 **LEC 3:** Scheduling **DUE:** Lab0	Sep 25 **Lab 1**: : Scheduler with eBPF	Sep 26	Sep 27	Sep 28	Sep 29
Sep 30	Oct 1 **LEC 4:** Scheduling II	Oct 2	Oct 3	Oct 4	Oct 5	Oct 6
Oct 7	Oct 8 **LEC 5:** Virtual Memory **DUE:** Lab 1	Oct 9 **Course Project starts**	Oct 10	Oct 11	Oct 12	Oct 13
Oct 14	Oct 15 **LEC 6:** Virtual Memory II	Oct 16	Oct 17	Oct 18	Oct 19	Oct 20
Oct 21	Oct 22	Oct 23	Oct 24	Oct 25	Oct 26	Oct 27
Oct 28	Oct 29 **LEC 7:** Mid-term	Oct 30	Oct 31	Nov 1	Nov 2	Nov 3
Nov 4	Nov 5 **LEC 8:** File Systems and LFS	Nov 6	Nov 7	Nov 8	Nov 9	Nov 10
Nov 11	Nov 12 **LEC 9:** Storage Media	Nov 13	Nov 14	Nov 15	Nov 16	Nov 17
Nov 18	Nov 19 **LEC 10:** OS Concurrency	Nov 20	Nov 21	Nov 22	Nov 23	Nov 24
Nov 25	Nov 26 **LEC 11:** Virtualization	Nov 27	Nov 28	Nov 29	Nov 30	Dec 1
Dec 2	Dec 3 **LEC 12:** OS Security	Dec 4	Dec 5	Dec 6	Dec 7	Dec 8
Dec 9	Dec 10 **LEC 13:** Final exam	Dec 11	Dec 12	Dec 13	Dec 14	Dec 15
Dec 16	Dec 17 **LEC 14:** Project Presentations **DUE:** Course project	Dec 18	Dec 19	Dec 20	Dec 21	Dec 22

Software platforms

- We will use multiple software platforms:
  - **Moodle**: Links to software platforms you need to register
  - **ED**: Communicate with the course staff and other students
  - **Github classroom**: For submitting the lab

Lectures

- The slides will be available before the class
- We will only have live lectures
  - They will **NOT** be recorded
- You are expected to attend every lecture

Note:

- Non-overlapping: 21
- Lecture overlapping: 22
- Other overlapping: 8

Exercise sessions

- The exercise session consists of:
  - Set of subjective questions with answers discussed in the class
  - No written solution will be released
- Students are expected to attend the exercise session to ask questions

Lab / project session

- Wednesday 2–4 PM in INM 202
- TAs will be present to answer questions

Grading policy:

- Lecture notes: 5%
- Lab: 15%
- Exams: 35%
  - Midterm: 15%
  - Finals: 20%
- Project: 45%

Lecture notes (5%)

- 2–3 students work in a group to take notes of the class and write a report
- Must be submitted within a week
- We will release lecture sheet for people to sign up.

Lab (15%)

- One lab
- Lab must be solved individually without using any help from AI assistants
  - Will zero the whole lab in case we find it
- Duration: 4 weeks in total
  - Part I: Already released
  - Part II: Will be released in the third week
- Submit your code along with the report to Github classroom
- **No late days**

Exams (35%)

- Written exams
- Include questions from both lectures and labs
- Midterm (15%): In week 8, during lecture hours
  - Covers lecture content from week 1–6
  - Covers the lab
- Finals (20%): In week 14, during lecture hours
  - Covers content from the whole course
  - Covers the lab

Project (35%)

- Free to choose the topic
- Should be done in a group of **3–4**
- Each project must finish within the semester with some tangible results
- Projects topics should be relevant to the OS and must be explicitly agreed by us
- Mid evaluation by week 8/9 to see the progress
- Final presentation on week 15 (last week of the lecture)
- Must submit the report and the code by the last week of the lecture

Some project ideas

- **Scheduling:** Adaptive application-defined scheduling
- **Virtual memory:** Time series application's memory layout visualizaiton
- **Virtual memory:** Application-aware state machine replication
- **Virtual memory / FS:** Application-aware data prefetching
- **Virtual memory / FS:** Designing new page cache algorithms
- **OS:** Using speculation to speed up kernel operations
- **Security:** Intra-process isolation
- **OWN:** Your own project: New OS design, OS component design, finding issues in Linux, using ML for OS operations

Projects: Proposal

- **Due:** October 8th (tentative)
- **Format:** 1 page, free writing
- Project proposal should include the following:
  - Motivation and goal of your work
  - Problem you would like to solve (define clearly)
  - Brief summary of related work
  - Your ideas to solve the problem
  - Research plan for the project

Academic honesty!

You are allowed to collaborate on everything in this course, **BUT** you must adhere to the academic honesty policies of the university:

- You **must not share or text about labs** with others
- You **must participate in collaboration**
- You **must not use someone else's code / text** in your solution for the lab
- You **cannot force anyone** to provide you the answer

Beware!

You will work on programming assignments (labs). Please work/review your programming skills in the first two weeks by solving **lab 0**

Cheating and academic integrity violation will be reported. Each labs will be checked for plagiarism. Please check the academic integrity policy of EPFL:

https://bit.ly/3BmfHJU

You will learn a lot in this course, but you will have to work from day one!

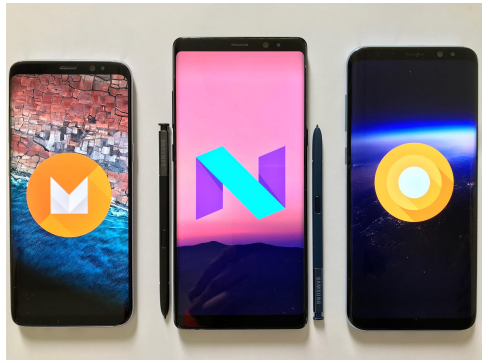
Focus of today's lecture

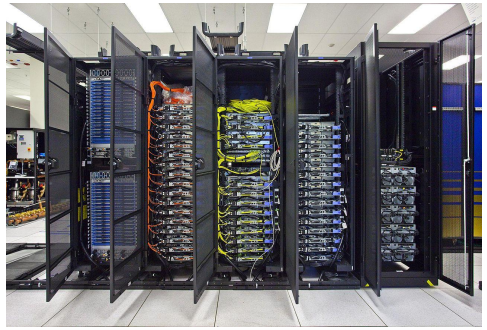
- Course logistics and policy
- **A brief primer on operating system**
- Overview of OS design / architecture

Why should you care about operating systems?

OS is everywhere!

- Almost every **device**, from your smartwatch, your smart light bulb, to your mobile phones, and laptops runs an operating system!





OS is everywhere!

- Almost every **device**, from your smartwatch, your smart light bulb, to your mobile phones, and laptops runs an operating system!

- Every **program** you will write will run on an operating system!

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

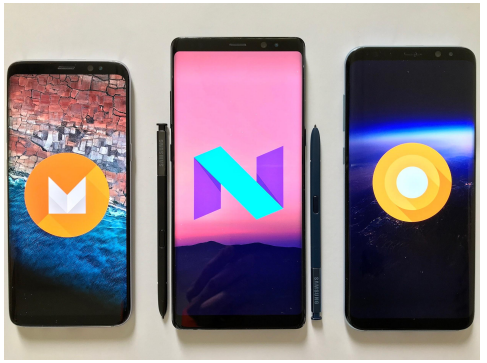
```c
#include <stdio.h>
```

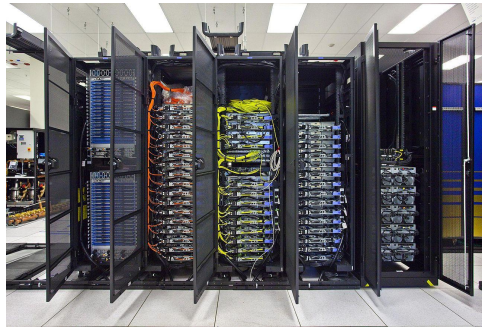
```c
#include <stdio.h>

int main(int argc, char *argv[])
{
  char *input = argv[1]; // Reading from standard input.
  while (1) {
    printf("%s\n", input); // Printing the input infinitely
  }
  return 0;
}
```

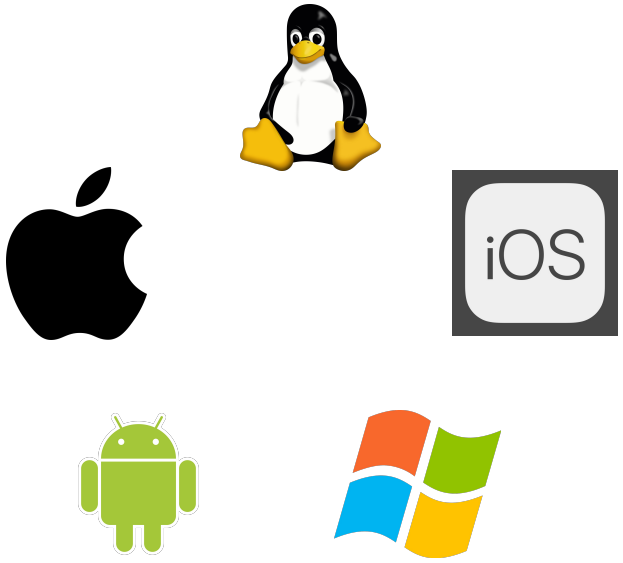
Why is it difficult to design an operating system?

1) So many different devices!





2) Communicate across devices around the world!

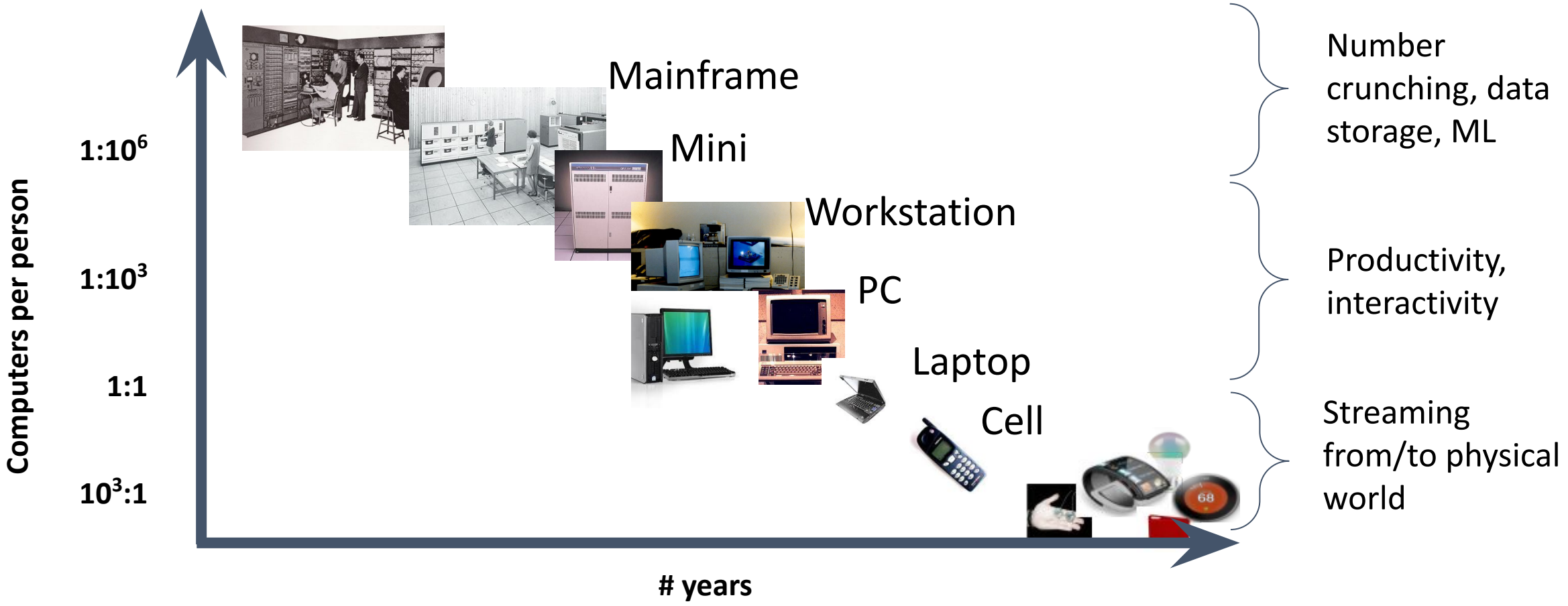


Each device has an operating system

Communicate over the Internet

Interface across huge sets of devices!

3) Bell's law



A new device class every 10 years!

4) Computer performance trends

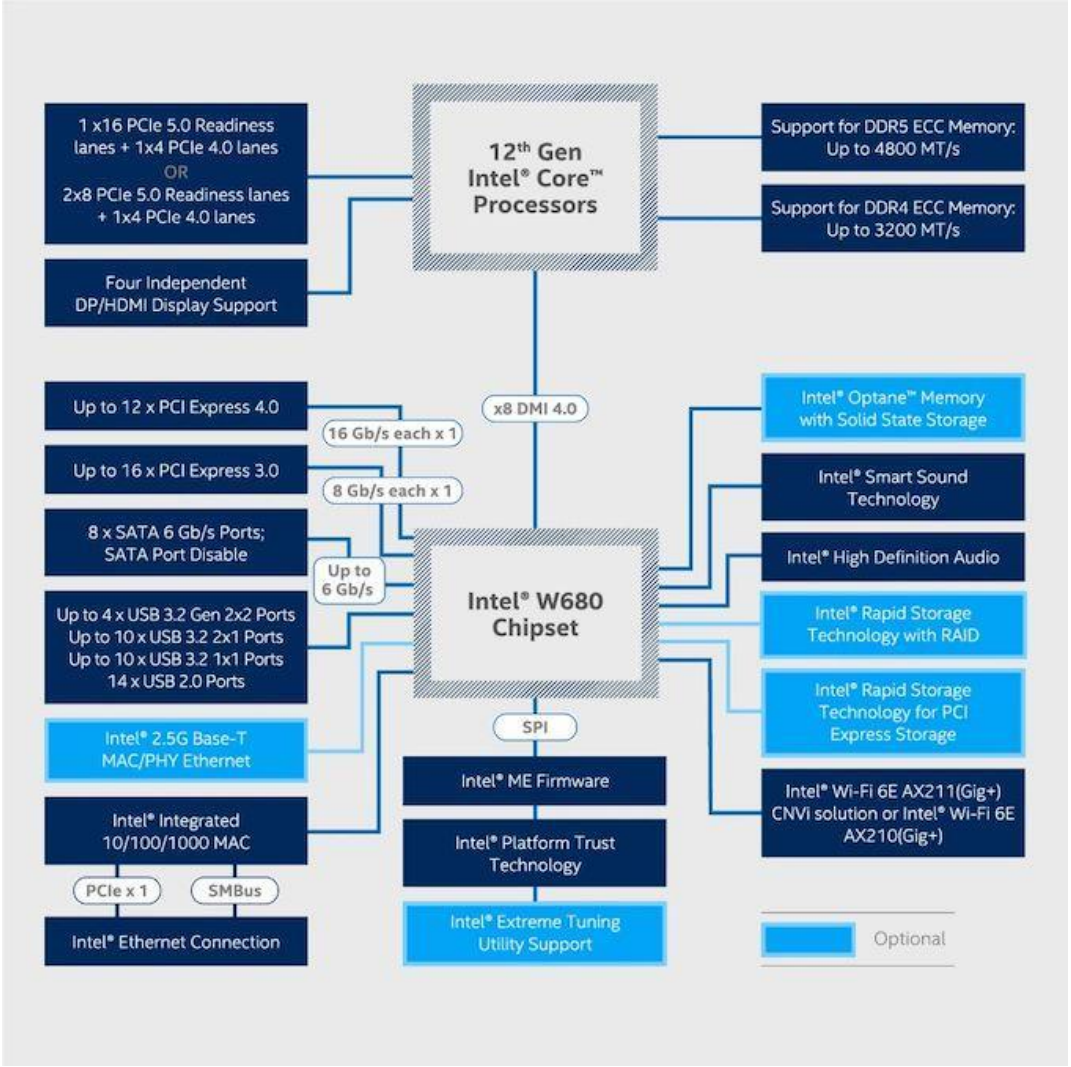
Latency Numbers Everyone Should Know

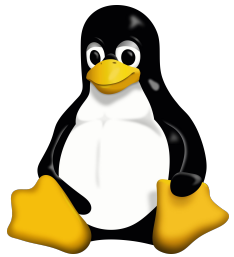
Operation	Time in ns	Time in ms (1ms = 1,000,000 ns)
L1 cache reference	1	
Branch misprediction	3	
L2 cache reference	4	
Mutex lock/unlock	17	
Main memory reference	100	
Compress 1 kB with Zippy	2,000	0.002
Read 1 MB sequentially from memory	10,000	0.010
Send 2 kB over 10 Gbps network	1,600	0.0016
SSD 4kB Random Read	20,000	0.020
Read 1 MB sequentially from SSD	1,000,000	1
Round trip within same datacenter	500,000	0.5
Read 1 MB sequentially from disk	5,000,000	5
Read 1 MB sequentially from 1Gbps network	10,000,000	10
Disk seek	10,000,000	10
TCP packet round trip between continents	150,000,000	150

New timescales keep coming up with devices

5) Hardware is getting complicated over time

- Hardware is becoming smarter
- Better reliability and security
- Better performance (more efficient code, more parallel computation)
- Better energy efficiency





What is an operating system?

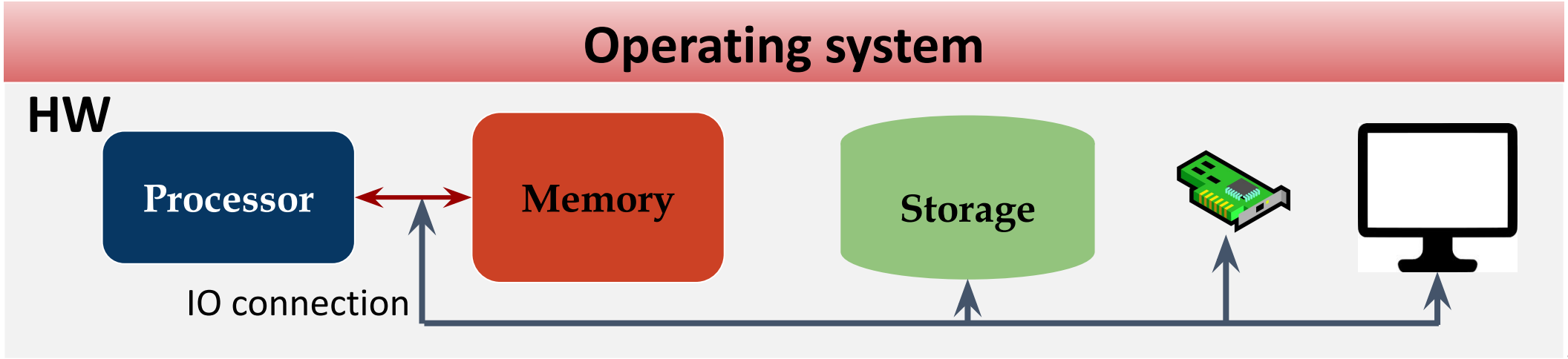
An operating system …

- A software layer that **interfaces** between diverse **hardware resources** and one or many **applications** running on the machine

Operating system

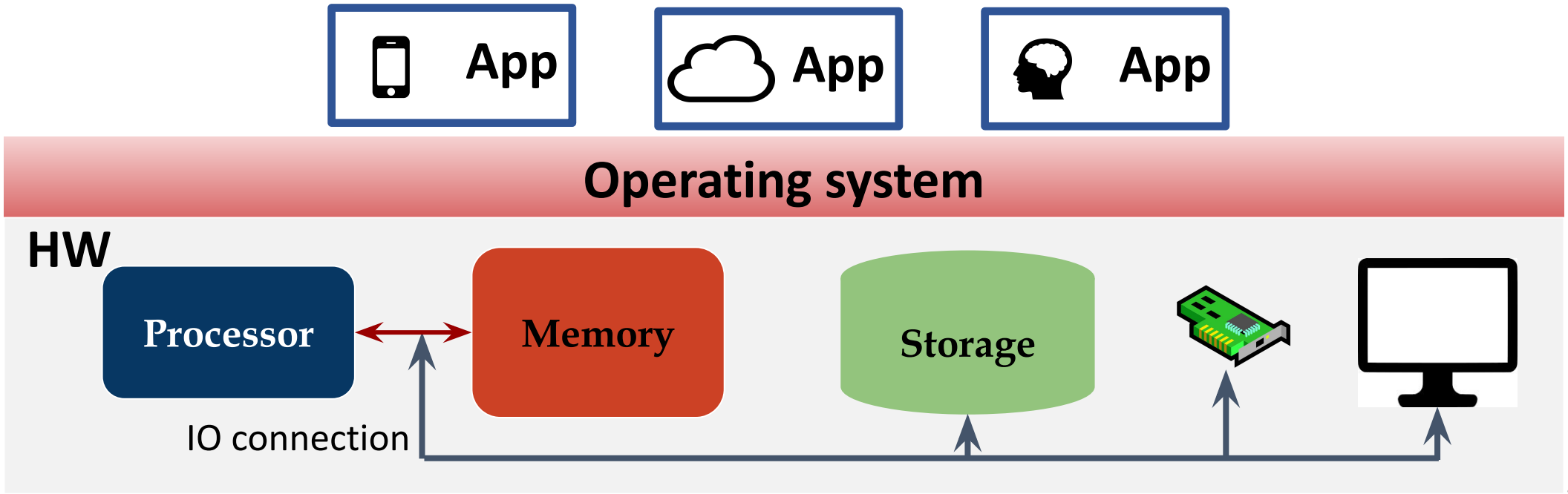
An operating system …

- A software layer that **interfaces** between diverse **hardware resources** and one or many **applications** running on the machine



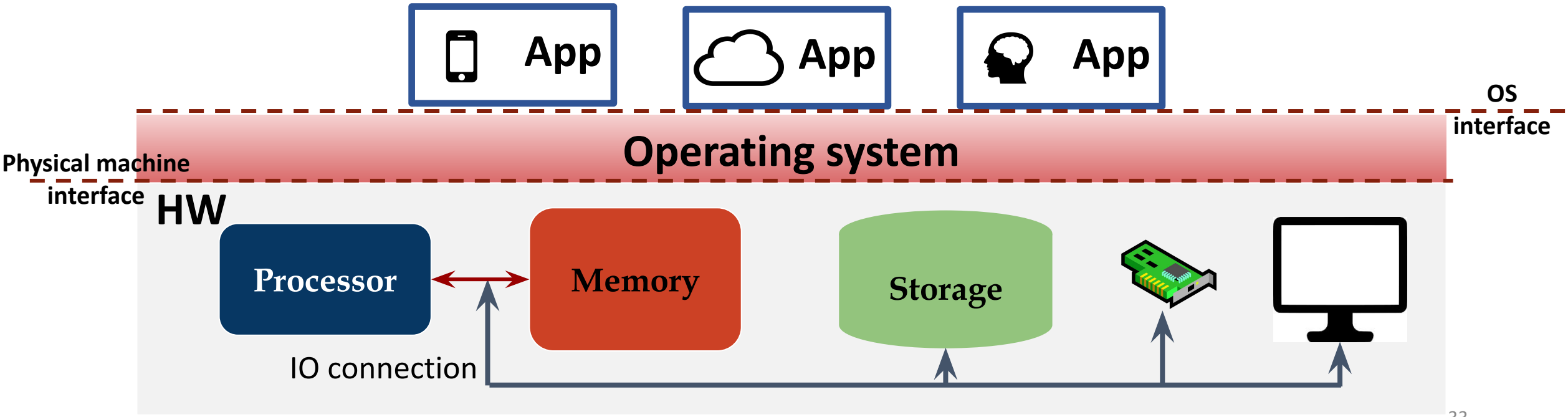
An operating system …

- A software layer that **interfaces** between diverse **hardware resources** and one or many **applications** running on the machine



An operating system …

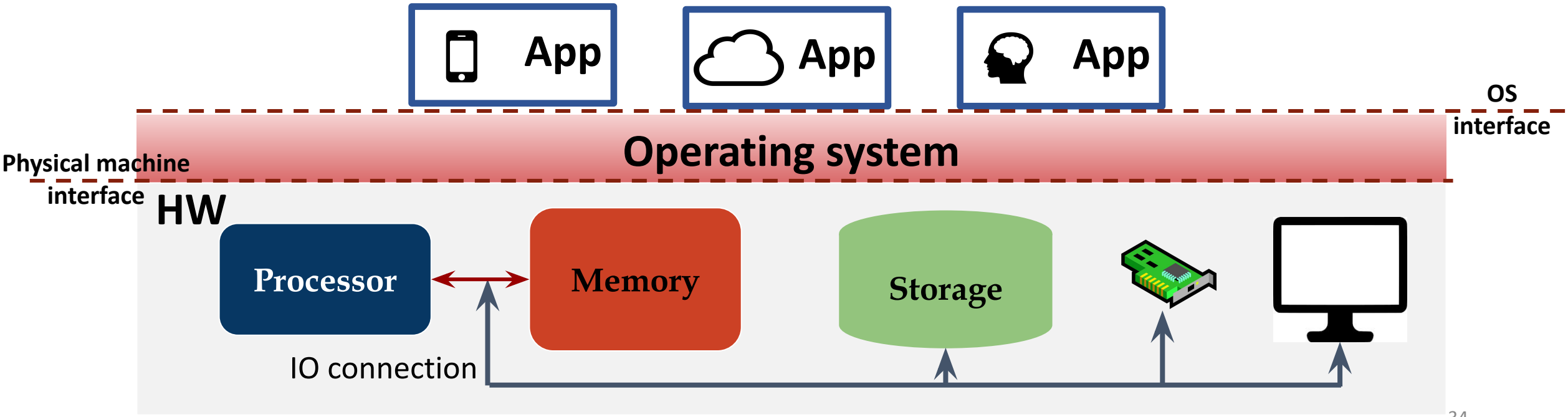
- A software layer that **interfaces** between diverse **hardware resources** and one or many **applications** running on the machine
- Implements **virtual machine** that is easier to program than raw hardware



An operating system …

- A software layer that **interfaces** between diverse **hardware resources** and one or many **applications** running on the machine
- Implements **virtual machine** that is easier to program than raw hardware

  Easier to use, simpler to code, more reliable, more secure …



What does an OS do for you?

- **Abstract** the hardware for convenience and portability
- **Multiplex** the hardware among multiple applications
- **Isolate** applications to contain bugs
- Allow **sharing** among applications

OS: Application view

- OS provides an execution environment for running programs in the form of abstractions
- Typical OS abstractions
  - Processor ⟶ Processes, threads
  - Memory ⟶ Address space (virtual memory)
  - Storage ⟶ files, directories
  - I/O devices ⟶ Files (+ ioctls)
  - Network ⟶ Files (socket, pipe …)

Today's OSes have several *Unix* features

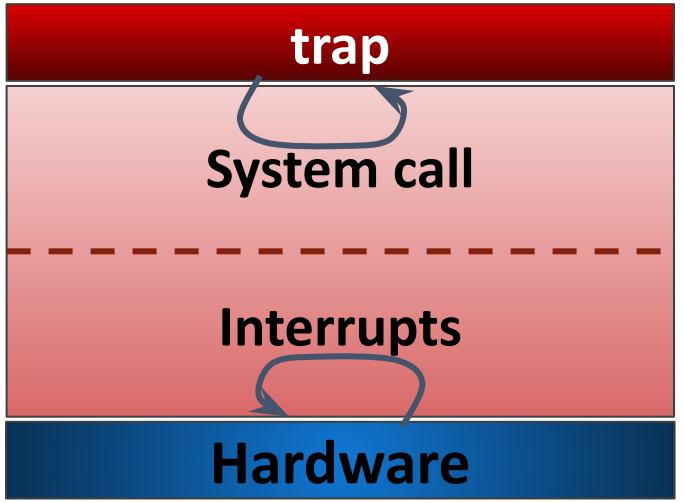
- **Process control**
  - fork(), exec(), wait(), exit()
  - Pipes for interprocess communication (IPC)
- **File systems**
- **Signals**
- **Shells**
  - Standard IO and IO redirection
  - Shell scripts

OS: System view

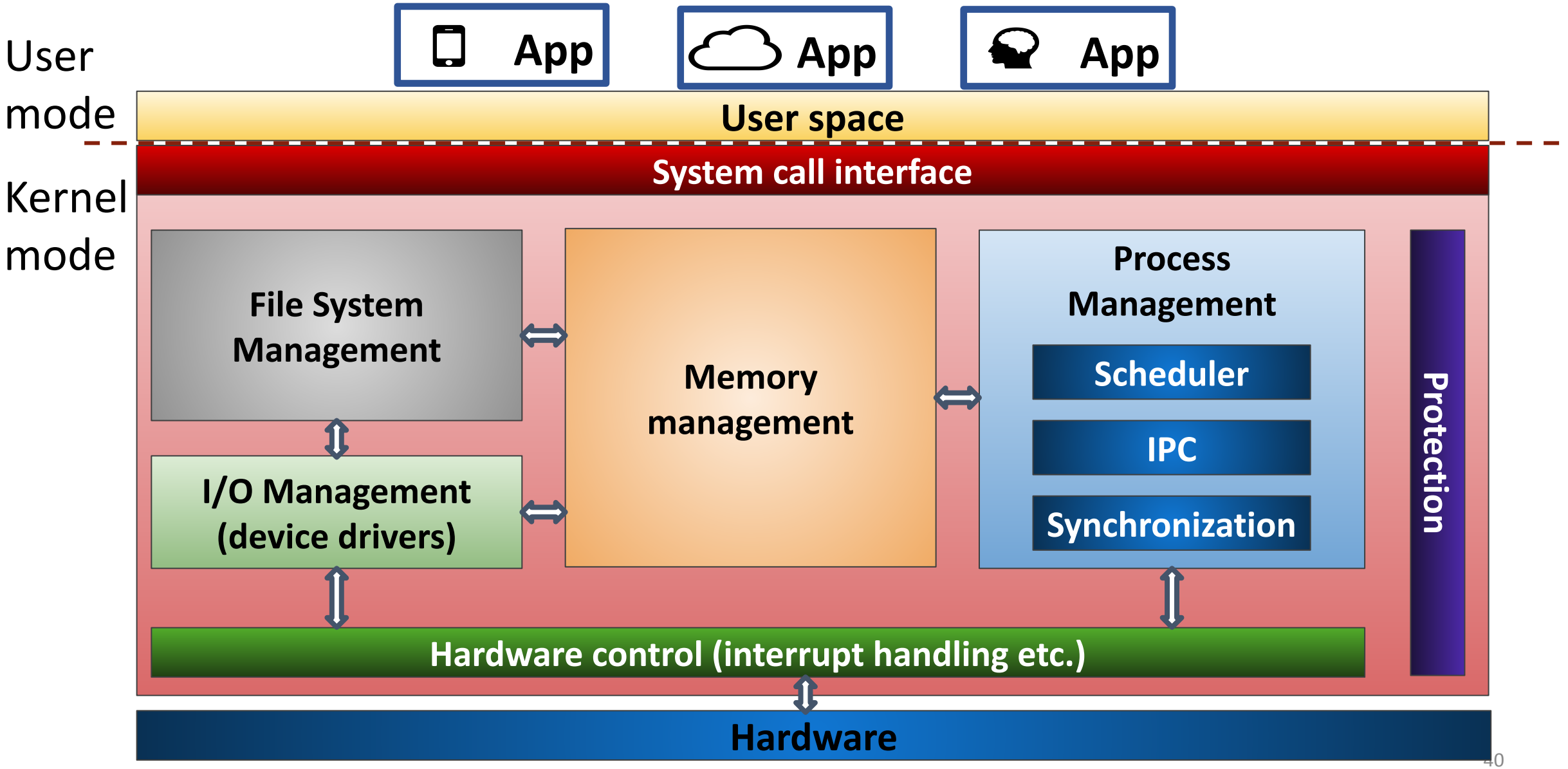
- OS manages various hardware resources
- Several metrics might be considered:
  - Sharing
  - Fairness
  - Efficiency
    - Throughput, latency, energy efficiency

OS: Implementation view

- OS is a highly concurrency program that manages both applications and hardware
- Both software and hardware interact with the OS via two events:
  - System calls
  - Interrupts



OS overview


User mode
Kernel mode
App
App
App
User space
System call interface
File System Management
Memory management
Process Management
Scheduler
IPC
Synchronization
Protection
I/O Management (device drivers)
Hardware control (interrupt handling etc.)
Hardware

User space vs. kernel space

- A CPU executes either in user space or in kernel space
- Only the code running in the kernel mode is allowed to perform **privileged** operations, such as controlling CPU or IO devices
- Hardware provides architectural support to ensure isolation
- A user space application talks to the kernel using **system call** interface
  - E.g., open(), read(), write(), close()

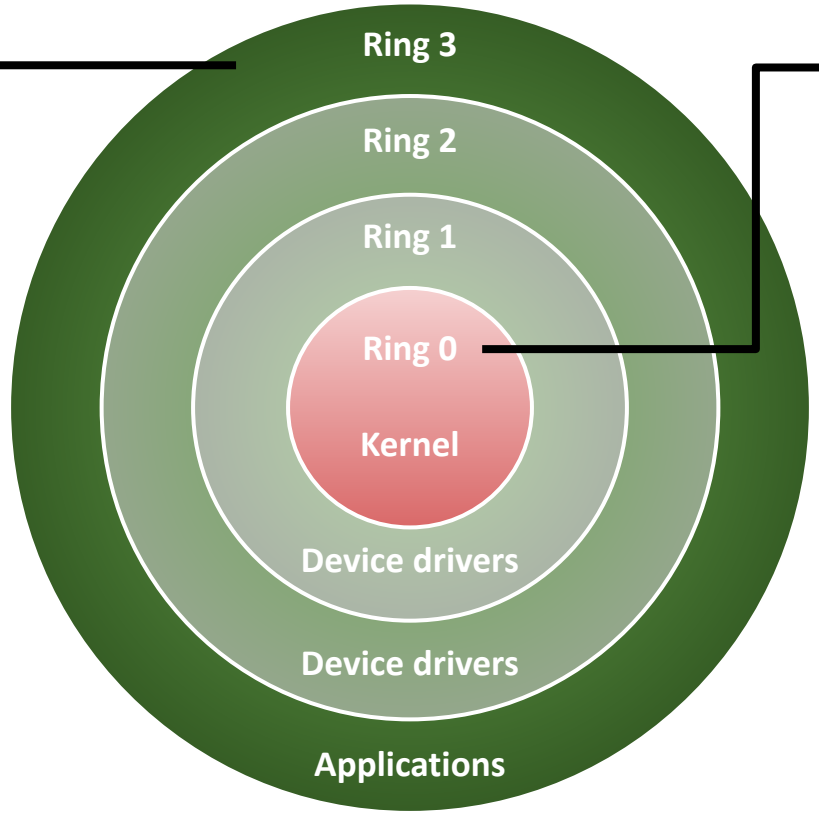
Architectural support for OS

Uses protection rings to distinguish among various modes of execution

User mode ← → **Kernel mode**

Least privileged (Ring 3)

Most privileged (Ring 0)



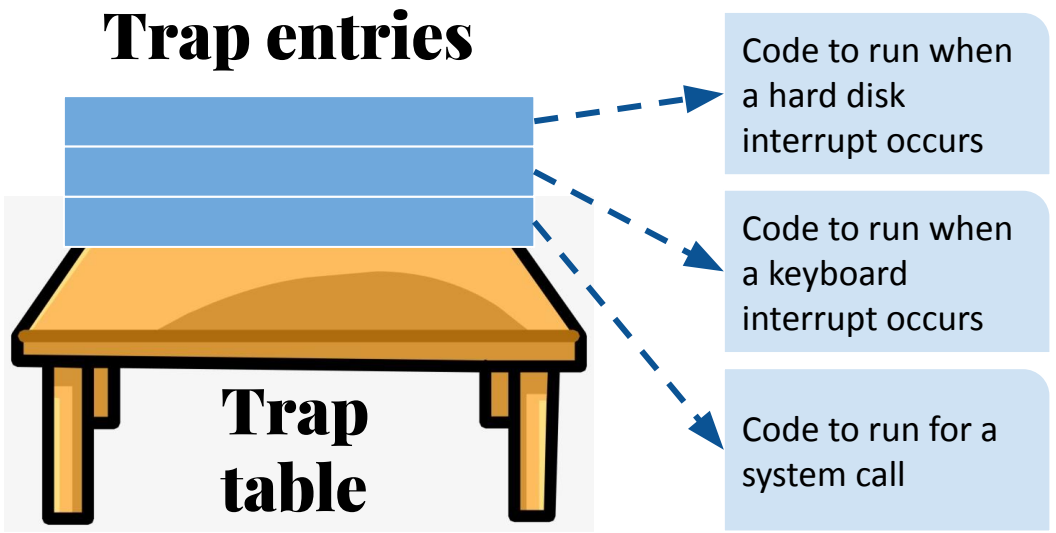
x86

User mode: Processes work in this mode; CPU executes only limited set of instructions – only allowed ones!

Kernel mode: OS works in this mode and executes instructions without checks

- Direct IO access
- Access system registers
- Memory state mgmt.

Architectural support for OS …

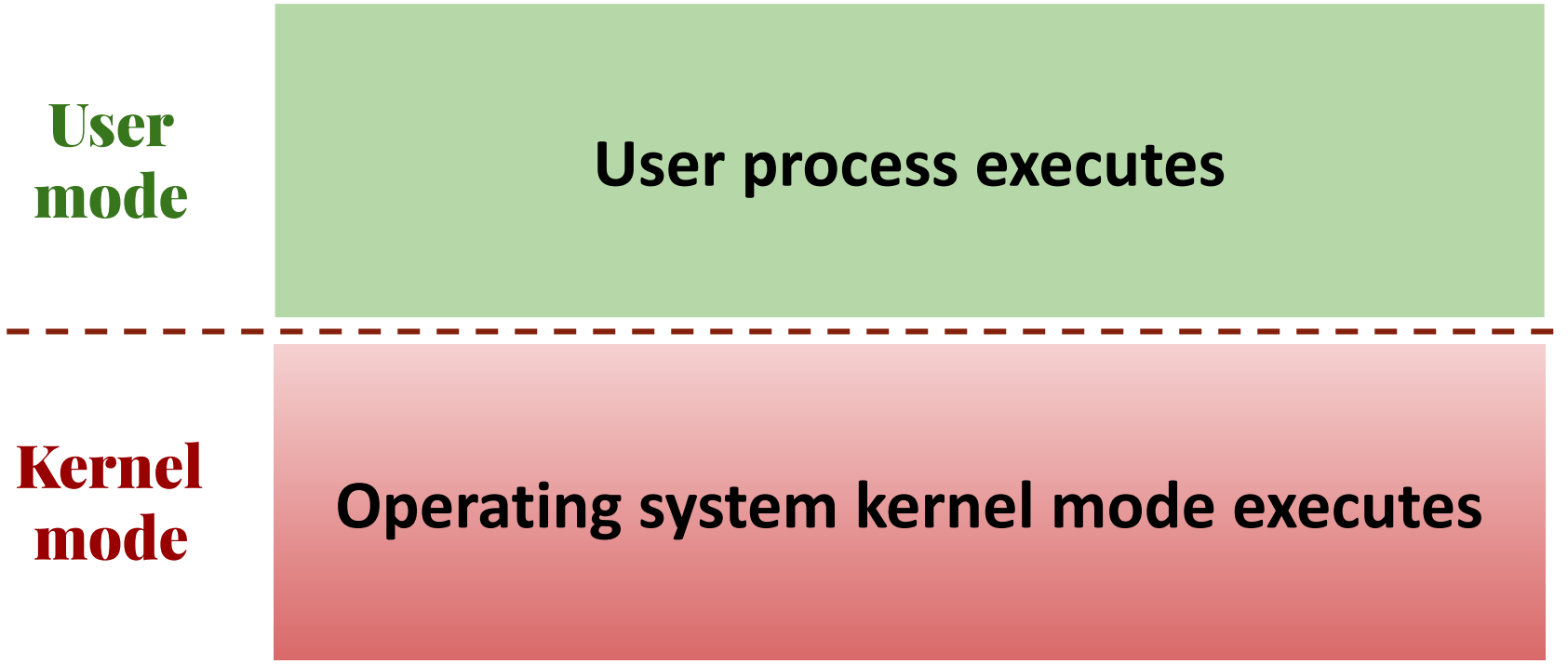


- **Interrupts**
  - Generated by hardware devices
  - Asynchronous
- **Exceptions**
  - Generated by software executing instructions
    - **Faults** (unintentional, but possibly recoverable): page faults, protection faults …
    - **Traps** (intentional): **syscall** instruction in x86
    - **Aborts** (unintentional and unrecoverable): parity error, machine error
  - Synchronous
  - Exception handling logic is similar as interrupt handling

Architectural support for OS …

- **Memory protection**
  - Segmentation / paging
- **Timer**
- **DMA (direct memory access)**
- **Atomic instructions**

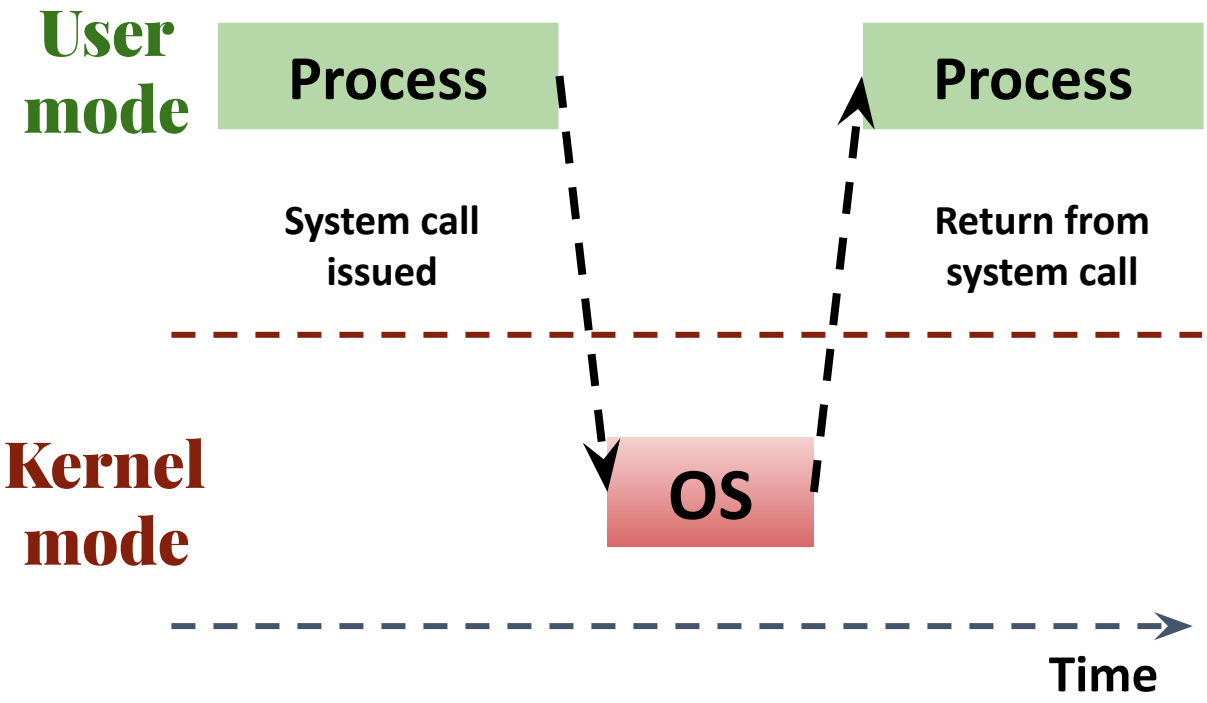
System Call: Requesting OS operations!



Requesting OS services (user mode → kernel mode)

- Processes can request OS services through the **system call API** (example: fork/exec/wait)

- **System calls transfer execution to the OS, meanwhile the execution of the process is suspended**



System calls

System calls exposes key functionalities:

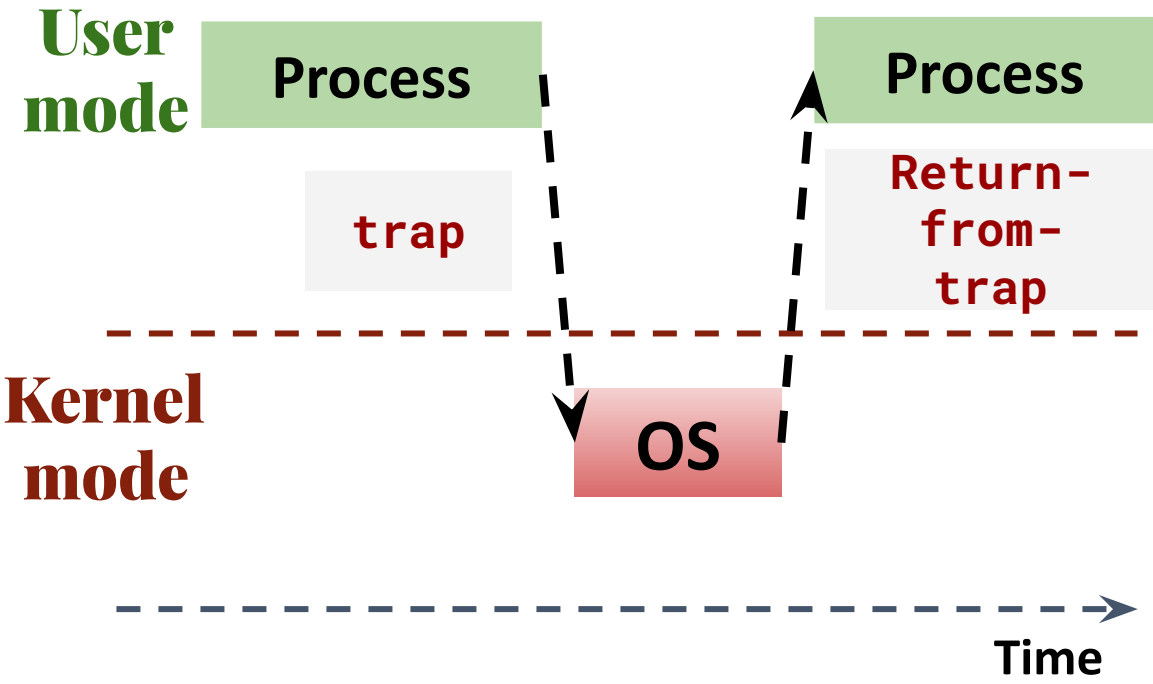
- Creating and destroying processes
- Accessing the file system
- Communicating with other processes
- Allocating memory

Most OSes provide hundreds of system calls

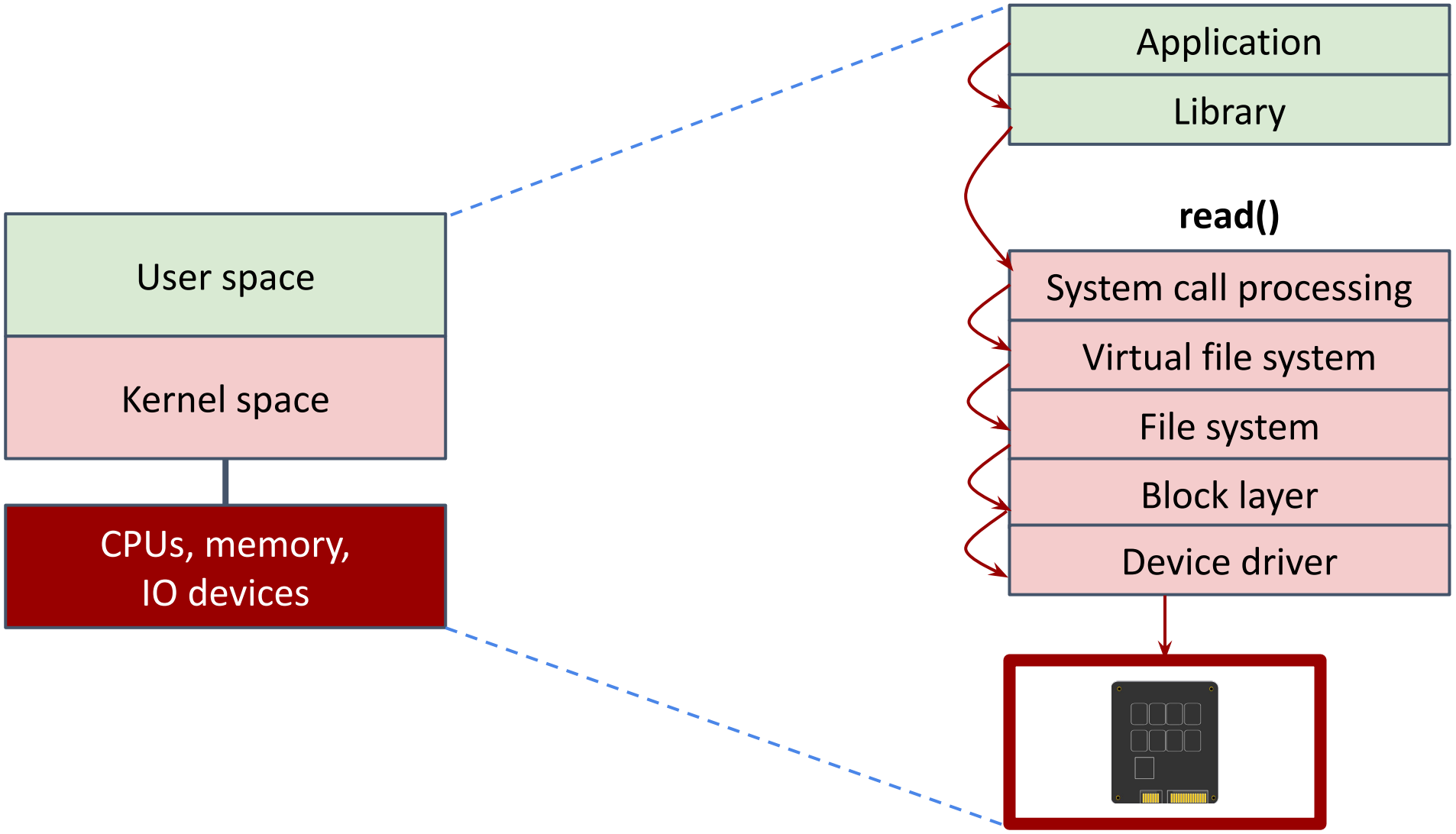
- Linux currently has more than 300+

Putting everything together for a system call

1. A system call is a **trap instruction (syscall/ecall)**
2. OS saves registers to **per-process stack**
3. Change mode from Ring 3 to Ring 0
4. Execute privileged operations
5. Change mode from Ring 0 to Ring 3
6. Restore the state of the process by popping registers in the **return from trap (iret)**



Executing a privileged operation



Traditional OS design

Traditional OS architecture: *Monolithic kernel*

- A traditional design: all of the OS operations run in **kernel, privileged mode**
  - Share the same address space
- Kernel interface ~= system call interface
- Good:
  - Easy for subsystems to cooperate (cache shared by FS and virtual memory)
  - Good performance
- Bad:
  - Dependencies between system components
  - Complex and huge; difficult to maintain due to large size
  - Leads to bugs, no isolation in the kernel

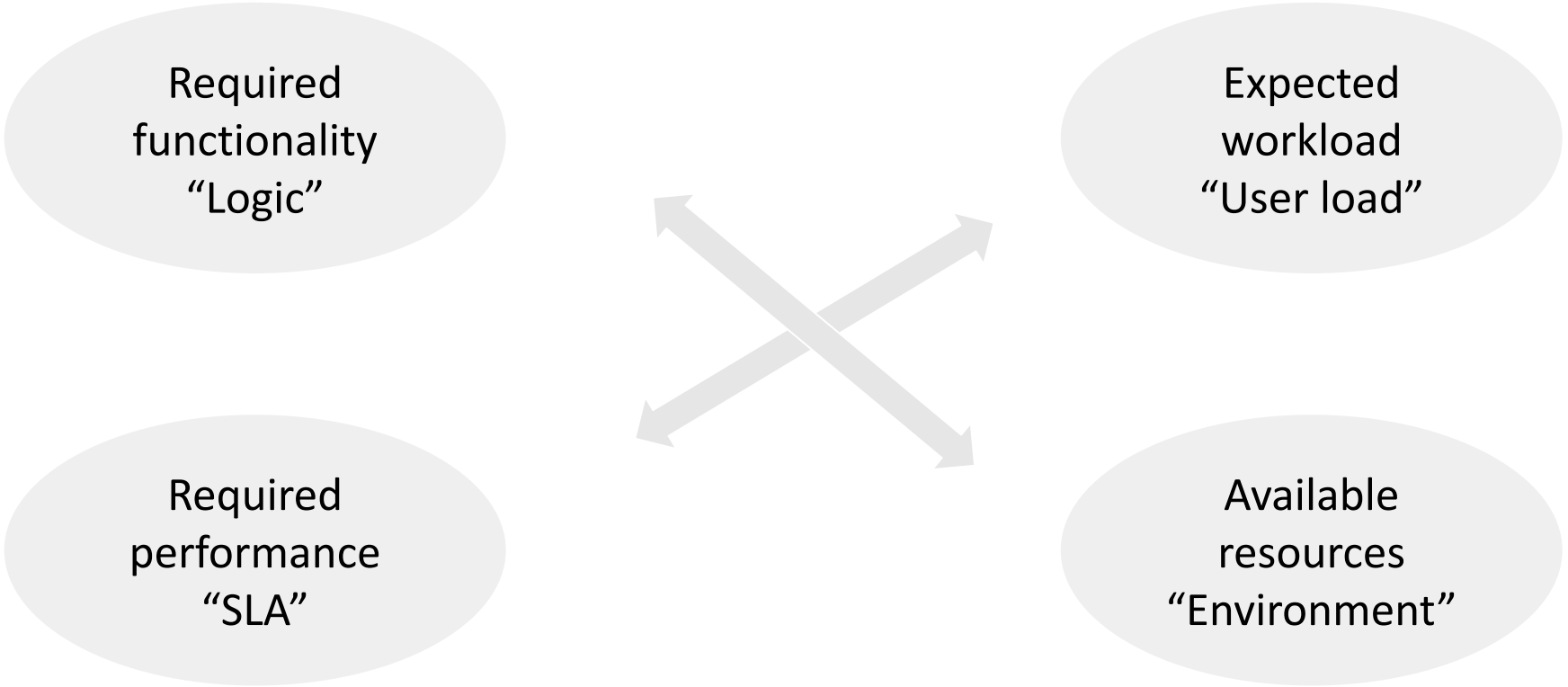
Let's take a step back and think about systems research and design in general

Systems research

- **The study of tradeoffs**
  - Functionality vs. performance
  - E.g., where to place error checking

Are there principles or rules of thumb that can help with large system design?

What is system design?



Something to do with **abstraction** (interface + policy) & **layering**

Need a balance between functionality & required perf / SLA

End-to-end principle

- Helps guide function placement among modules in a **distributed system**

- **Argument**
  - Implement the functionality in the lower layer
    - A large number of higher layers / applications use this functionality and implementing it at the lower layer improves the performance of many of them,

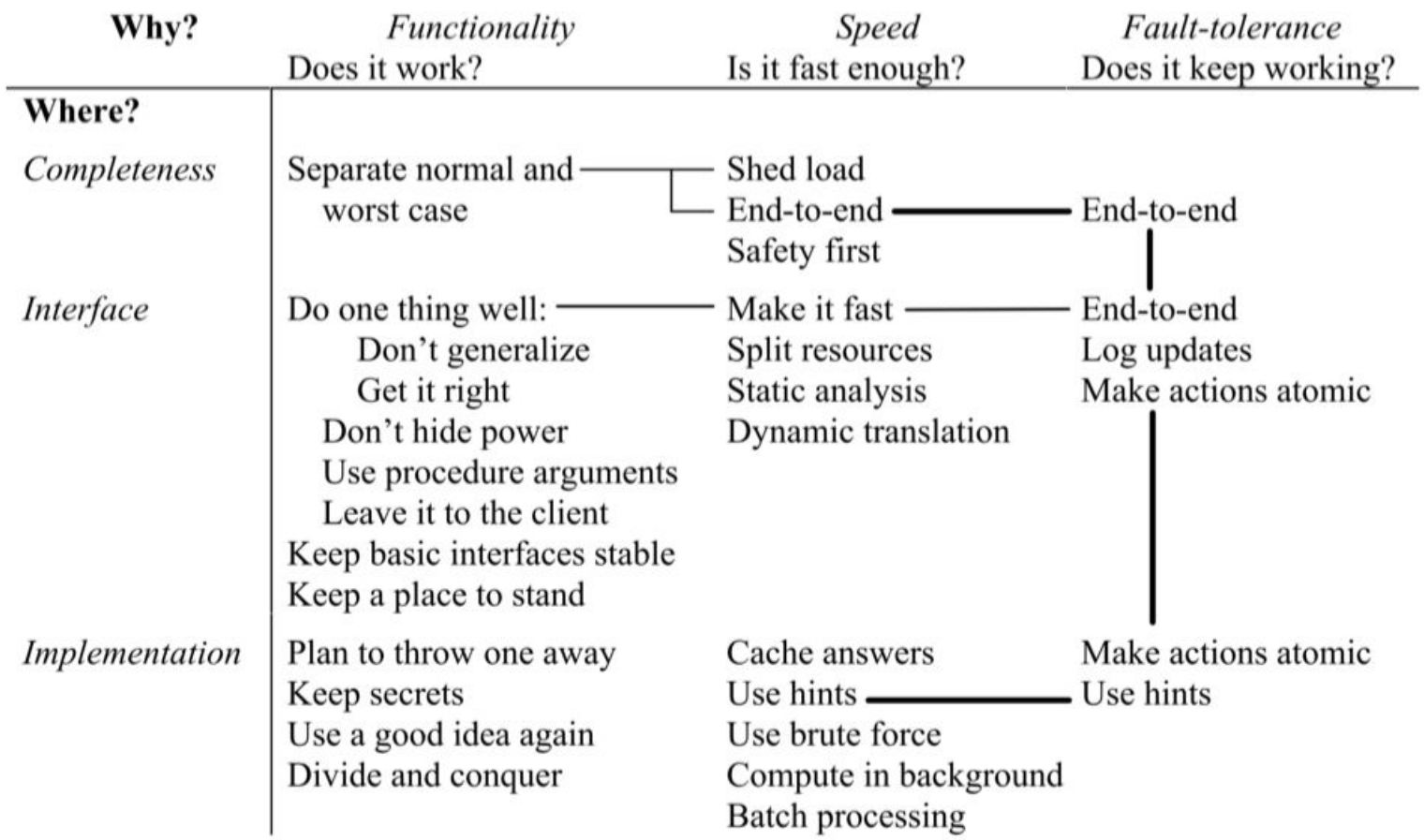
      AND
    - Does not hurt remaining applications

Difficult to reason as computer systems are complex

Rely on system design hints: *Why* and *Where*

- **Why:**
  - Functionality: does it work?
  - Speed: is it fast enough?
  - Fault-tolerance: does it keep working?
- **Where:**
  - Completeness
  - Interface
  - Implementation

System design hints from Butler Lampson



Why?	*Functionality* Does it work?	*Speed* Is it fast enough?	*Fault-tolerance* Does it keep working?
Where?			
Completeness	Separate normal and worst case	Shed load End-to-end Safety first	End-to-end
Interface	Do one thing well: Don't generalize Get it right Don't hide power Use procedure arguments Leave it to the client Keep basic interfaces stable Keep a place to stand	Make it fast Split resources Static analysis Dynamic translation	End-to-end Log updates Make actions atomic
Implementation	Plan to throw one away Keep secrets Use a good idea again Divide and conquer	Cache answers Use hints Use brute force Compute in background Batch processing	Make actions atomic Use hints

Another system design aspect: *Policy* vs. *Mechanism*

- **Policy:**
  - ***What* should be done?**
  - Policy decisions must be made for all resource allocation and scheduling problems
    - CPU scheduling: scheduling algorithm, quantum size, priority, etc.
- **Mechanism:**
  - ***How* to do something?**
  - The tool for implementing a set of policies
  - CPU scheduling: dispatcher for low-level context switching, priority queues etc.

Separating *Policy* from *Mechanism*

- **A key principle in OS design**
- Policies are likely to change depending on workloads, and also across places or over time
  - Each change in policy would require a change in the underlying mechanism
- A general mechanism, separated from policy, is more desirable
  - A change in policy would then require redefinition of only certain parameters of the system instead of resulting in a change in the mechanism
  - It is possible to experiment with new policy without breaking mechanisms

Focus of today's lecture

- Course logistics and policy
- A brief primer on operating system
- **Overview of OS design / architecture**

Traditional OS architecture: *Monolithic kernel*

- A traditional design: all of the OS operations run in **kernel, privileged mode**
  - Share the same address space
- Kernel interface ~= system call interface
- Good:
  - Easy for subsystems to cooperate (cache shared by FS and virtual memory)
  - Good performance
- Bad:
  - Dependencies between system components
  - Complex and huge; difficult to maintain due to large size
  - Leads to bugs, no isolation in the kernel

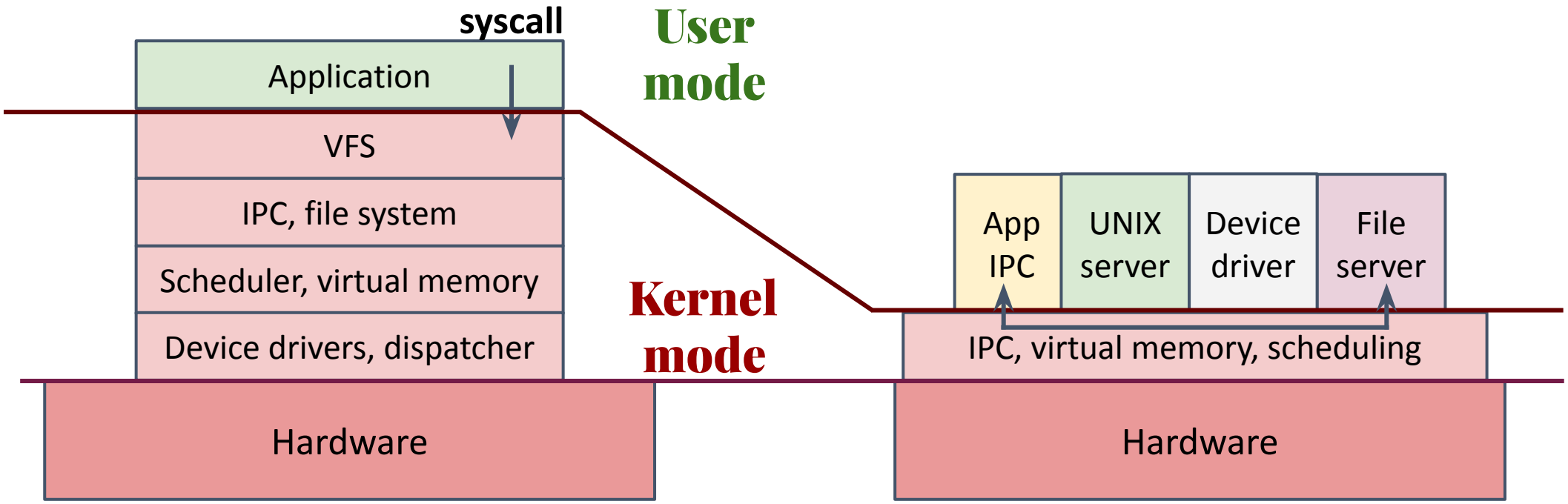
Different types of kernel designs

- Monolithic kernel
- Microkernel
- Hybrid kernel
- Exokernel

Alternative OS architecture: *Microkernel*

- Minimalist approach
  - IPC, virtual memory, thread scheduling
- Put the rest in the user space: device drivers, FS, paging in VM
- Kernel interface != system call interface
- Good:
  - More stable with less services in kernel space
  - More isolation
- Bad:
  - Lots of system calls and context switches
  - IPCs may be slow

Monolithic kernels vs. Microkernels



Alternative OS architecture: *Hybrid kernels*

- Combine the best of both worlds
  - Speed and simple design of monolithic kernel
  - Modularity and stability of a microkernel
- Still similar to a monolithic kernel
  - Disadvantages still apply here
- E.g., Windows NT, BeOS

Alternative OS architecture: *Exokernels*

- Follows end-to-end principle
  - Extremely minimal
  - Fewest hardware abstractions as possible
  - Just allocate physical resources to applications
- Disadvantages:
  - More work for application developers
  - Difficult to reason about sharing

Summary

- OS manage hardware and software
  - Abstracts resources, enables sharing and isolation of applications
- Provides basic set of abstractions on top of hardware
  - Process, address space, files, etc.
- Several OS design possible based on end-to-end, policy vs mechanism
  - Monolithic: All OS components in kernel space; complex
  - Microkernels: A minimal abstraction using IPC
  - Hybrid: Combination of both monolithic and microkernel
  - Exokernel: Extremely minimal; directly expose all resources to applications