CS162
Operating Systems and Systems Programming
Lecture 2

Protection: Processes and Kernels

Professor Natacha Crooks & Matei Zaharia

https://cs162.org/

Slides based on prior slide decks from David Culler, Ion Stoica, John Kubiatowicz, Alison Norman and Lorenzo Alvisi

Admistratrivia

Homework and Early Drop Deadline

HW0 Due 30/1

Early Drop Deadline 31/1

Should be working on Homework 0 already!

- Get familiar with all the cs162 tools

HW1 will be released on 31/1

Projects are looming

Group Formation Form (Link on EdStem) is due Feb 8th.

There is a teammate search functionality on EdStem.

Discussions are starting! First 2 optional but mandatory afterwards

Project 0 will be released on 27/1

Recall: Operating System

An operating system implements a virtual machine for the application whose interface is more convenient than the raw hardware interface
(convenient = security, reliability, portability)

Application 1	Application 2	Application 3
Operating System		
Hardware		

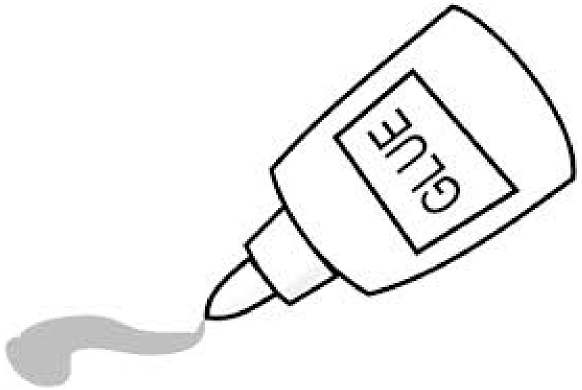
Recall: Three main hats

Referee

Manage protection, isolation, and sharing of resources

Illusionist

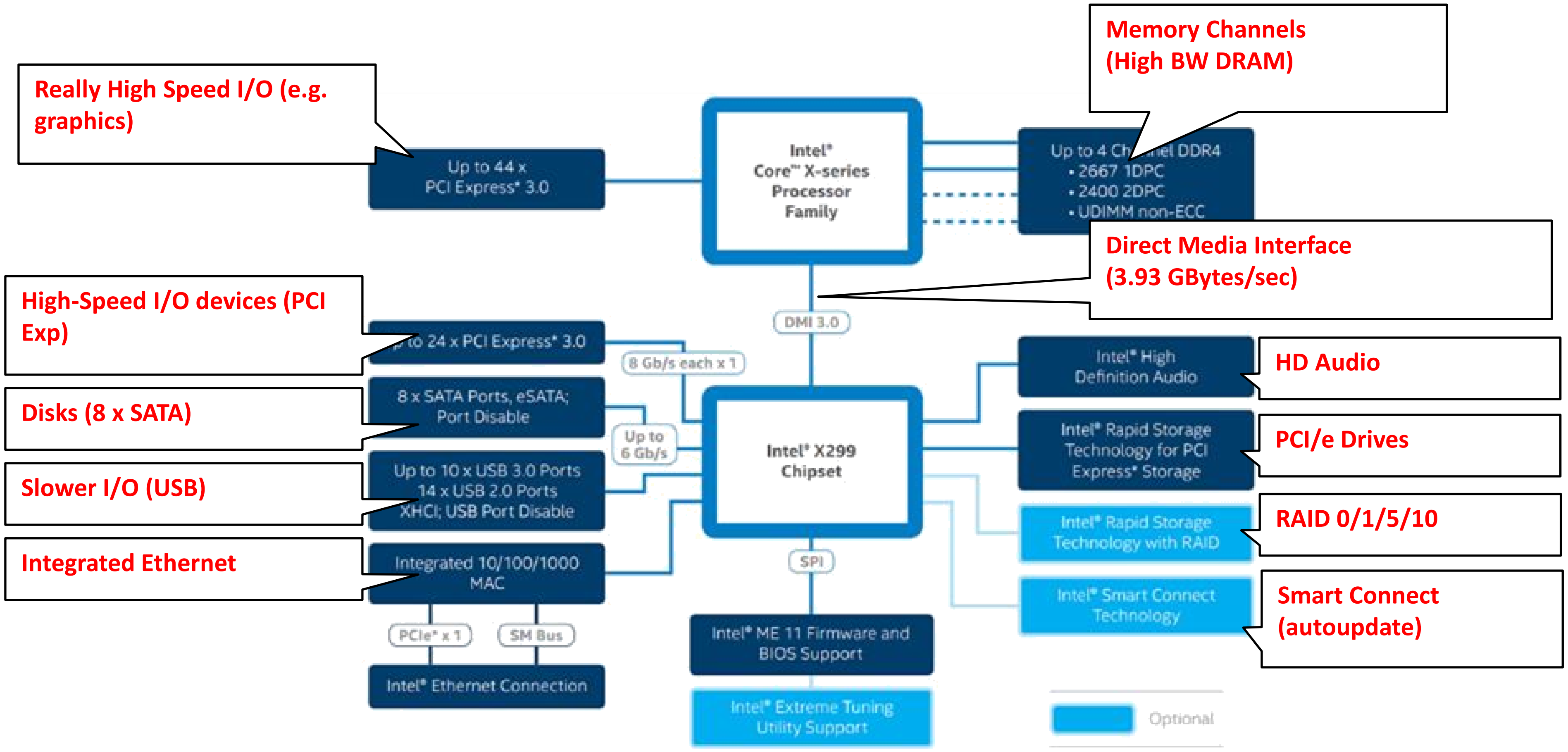
Provide clean, easy-to-use abstractions of physical resources



Glue

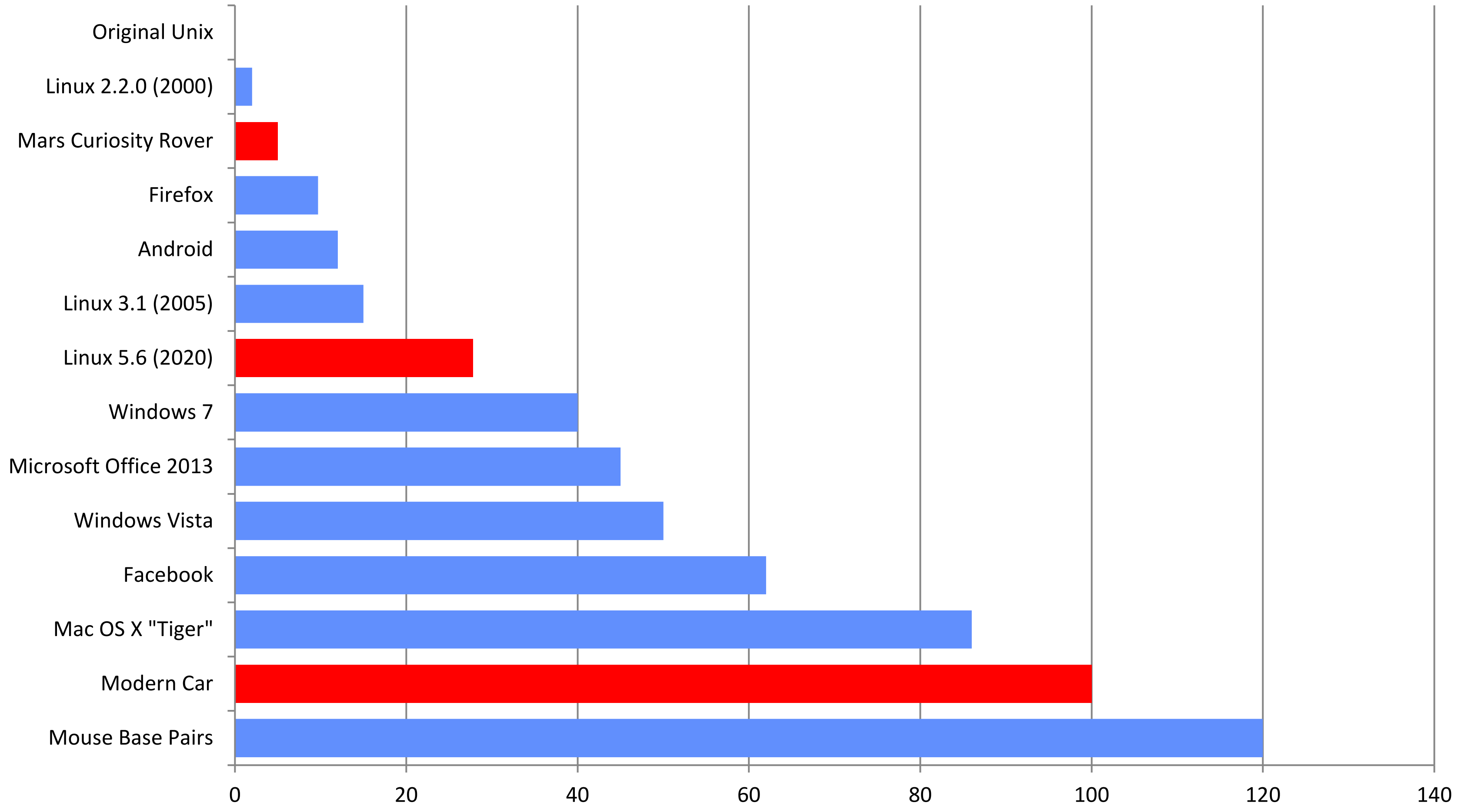
Provides a set of common services

Recall: HW Complex



Intel Skylake-X I/O Configuration

Recall: Increasing Software Complexity



Evaluation Criteria

Performance
Efficiently, with low overhead and equitably.

Reliability
System does what it supposed to do

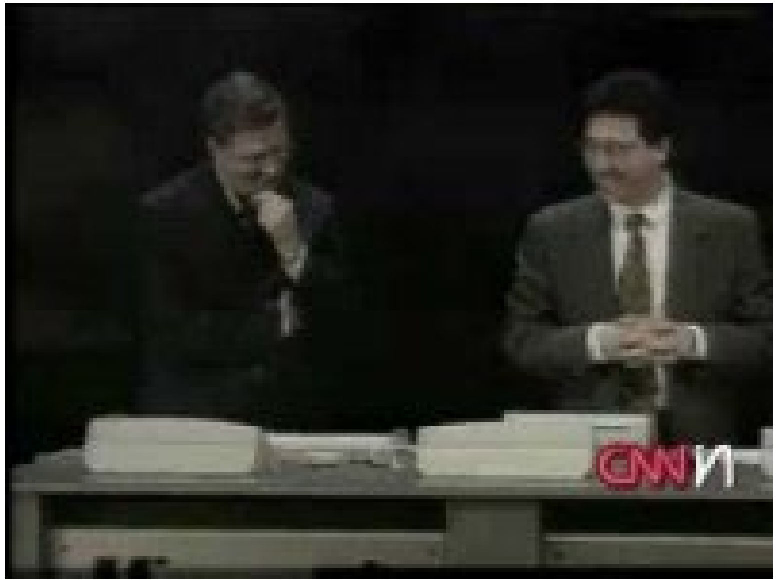
Security
Minimise vulnerability to attack

Portability
No need to change abstractions when hardware changes

Evaluation Criteria: Reliability

System does what it is supposed to do

OS failures catastrophic!



Availability: mean time to failure + mean time to repair

Evaluation Criteria: Security

Minimize vulnerability to attack

Integrity: Computer's operation cannot be compromised by a malicious attacker

Privacy: data stored on computer accessible to authorized users

Enforcement Policy

How the OS ensures only permitted actions are allowed

Security Policy

What is permitted

Evaluation Criteria: Portability

A portable abstraction does not change as the hardware changes

Can’t rewrite application (or OS!) every time

Must plan for hardware that does not exist yet!

Application

Abstract Machine Interface

Operating System

Hardware Abstraction Layer

Hardware

Three “Prongs” for the Class

Understanding OS principles

System Programming

Map Concepts to Real Code

Topic Breakdown

- Virtualizing the CPU
  - Process Abstraction and API
  - Threads and Concurrency
  - Scheduling
- Virtualizing Memory
  - Virtual Memory
  - Paging
- Persistence
  - IO devices
  - File Systems
- Distributed Systems
  - Challenges with distribution
  - Data Processing & Storage

Topic Breakdown

- Virtualizing the CPU
  - Process Abstraction and API
  - Threads and Concurrency
  - Scheduling
- Virtualizing Memory
  - Virtual Memory
  - Paging
- Persistence
  - IO devices
  - File Systems
- Distributed Systems
  - Challenges with distribution
  - Data Processing & Storage

Side Note: Mechanisms vs Policy

Mechanism

Low-level methods or protocols that implement a needed piece of functionality

A Brake Pedal!

Policy

Algorithms for making decisions within the OS.

Use the mechanism.

"I break when I see a stop sign"

Goals for Today

- What are the requirements of a good VM abstraction?
- What is a process?
- How does the kernel use processes to enforce protection?
- When does one switch from kernel to user mode and back?

Goal 1: Requirements for Virtualization

The OS will protect you

Protection is necessary to preserve the virtualization abstraction

Protect applications from other application's code (reliability, security, privacy)

Protect OS from the application

Protect applications against inequitable resource utilisation (memory, CPU time)

Goal 2: What is a Process?

A process (simplified)

A process is an instance of a running program

CPU

Memory

(address space)

Store code, data, stack, heap

Registers

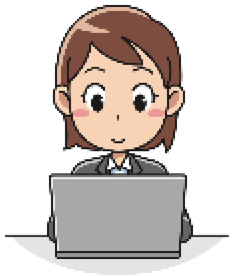
Program Counter, Stack Pointer

Regular registers

IO information

Open files (and others)

From program to process



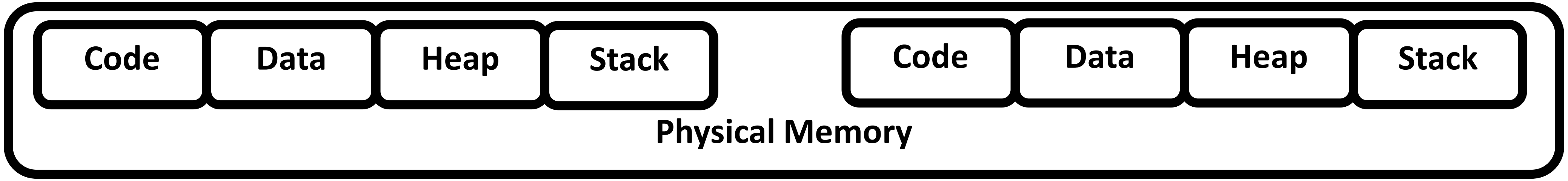
```c
#include <stdio.h>

int main()
{
    printf("Hello World");

    return 0;
}
```

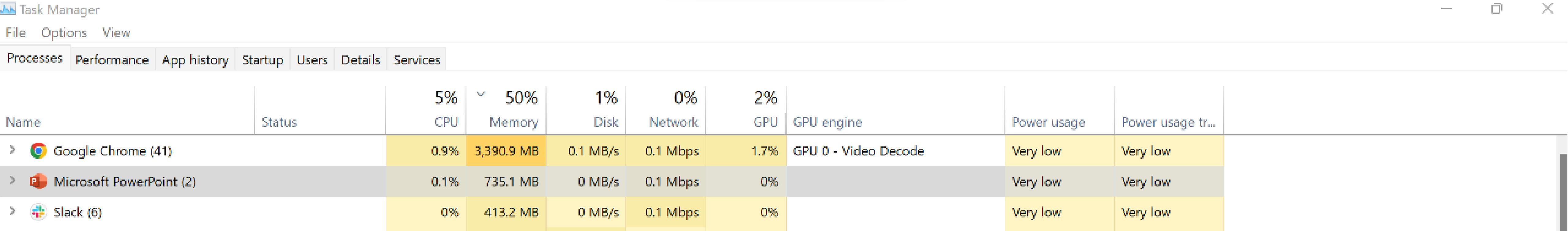
Executable image, instructions and data

./helloworld

`crooks@laptop> ./helloworld`

`crooks@laptop> ./helloworld`




Task Manager

File Options View

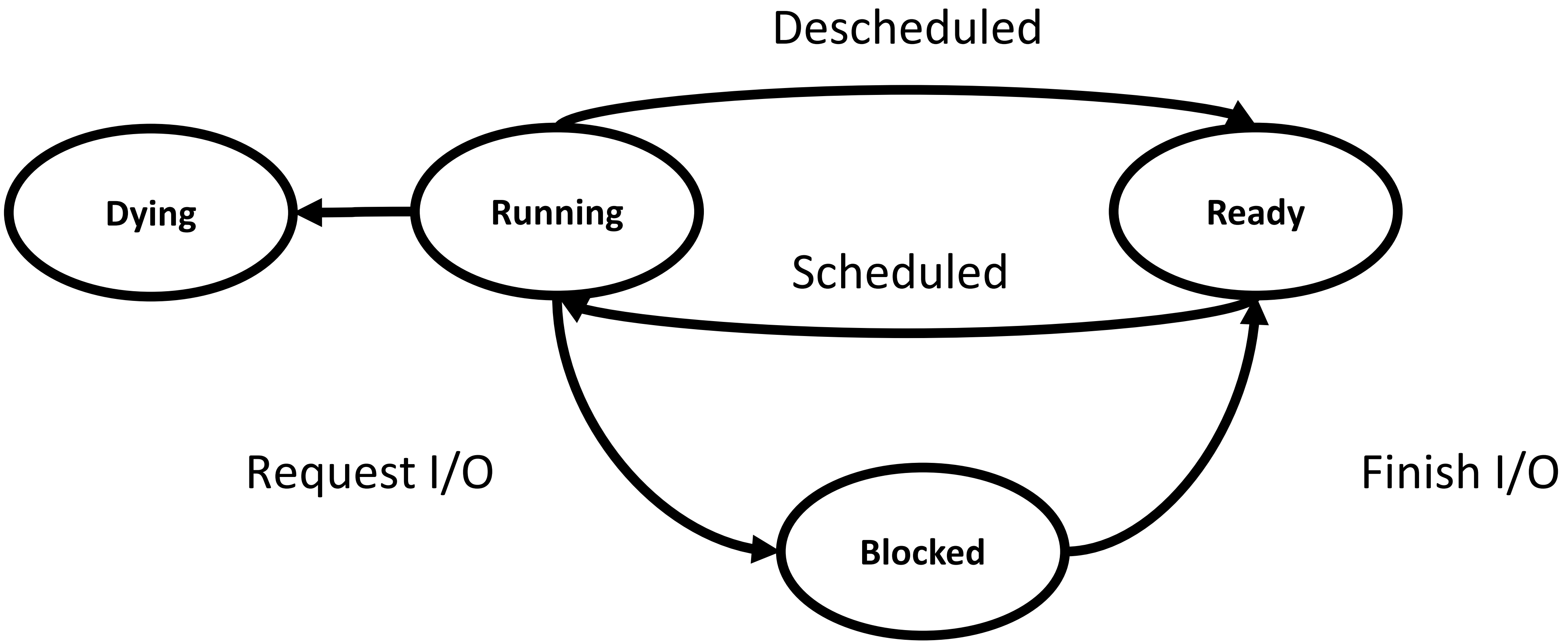
Processes | Performance | App history | Startup | Users | Details | Services

Name	Status	5% CPU	50% Memory	1% Disk	0% Network	2% GPU	GPU engine	Power usage	Power usage tr...
Google Chrome (41)		0.9%	3,390.9 MB	0.1 MB/s	0.1 Mbps	1.7%	GPU 0 - Video Decode	Very low	Very low
Microsoft PowerPoint (2)		0.1%	735.1 MB	0 MB/s	0.1 Mbps	0%		Very low	Very low
Slack (6)		0%	413.2 MB	0 MB/s	0.1 Mbps	0%		Very low	Very low

Process Life Cycle

A process can be in one of several states:

(real OSes have additional variants)



Process Management by the OS

Process Control Block (or process descriptor)

in OS stores necessary metadata

PC

Stack Ptr

Registers

PID

UID

List of open files

Process State

…

Three “Prongs” for the Class

Understanding OS principles

System Programming

Map Concepts to Real Code

Processes in the wild (well, in the kernel)

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

In Linux: `task_struct` defined in <`linux/sched.h`>

Xv6 Kernel (proc.h)

Processes in Pintos

```
struct process {
  /* Owned by process.c. */
  uint32_t* pagedir;           /* Page directory. */
  char process_name[16];       /* Name of the main thread */
  struct thread* main_thread; /* Pointer to main thread */

  /* All the fun data structures you're going to add */
};
```

Pintos (userprog/process.h)

Many Processes

Process List stores all processes

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

Xv6 Kernel (proc.c)

<u>Run Queues</u>

Lists all PCBs in **READY state**

<u>Wait Queues</u>

Lists all PCBs in **BLOCKED state**

The Illusionist and the Referee are Back

Illusionist	Referee
Give every process the illusion of running on a private CPU	Manage resources to allocate to each process
Give every process the illusion of running on private memory	Isolate process from all other processes and protect OS

Operating System Kernel

Lowest level of OS running on system.

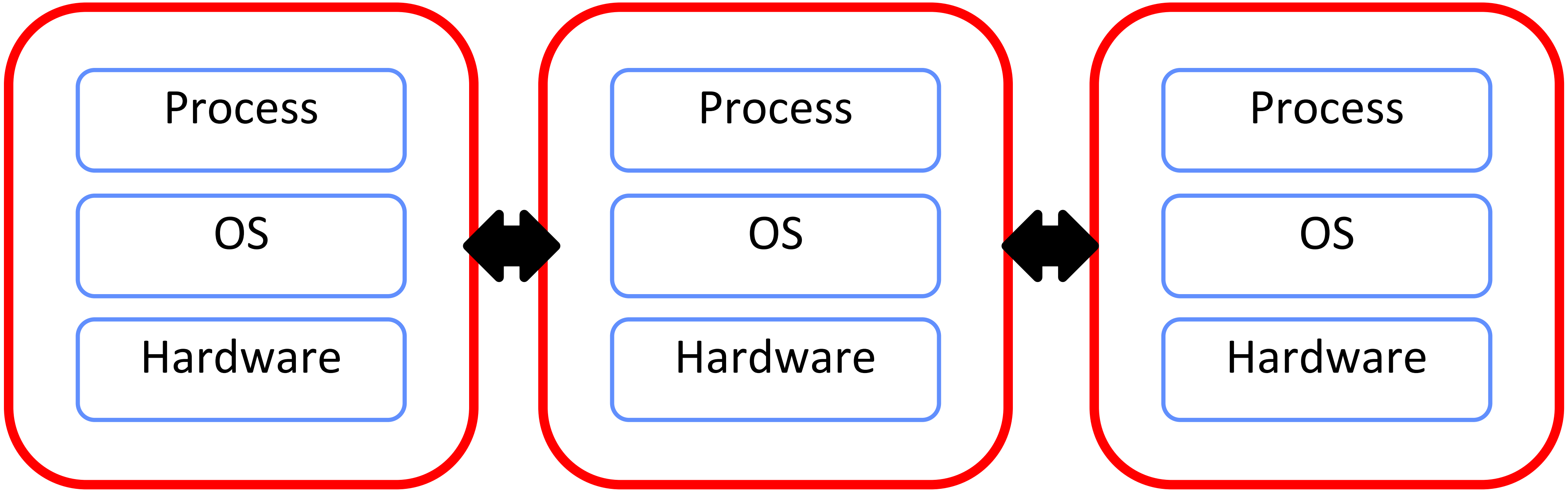
Kernel is trusted with full access to all hardware capabilities

All other software (OS or applications) is considered untrusted

Untrusted	Applications
	Rest of OS
Trusted	Operating System Kernel
Untrusted	Hardware

The Process, Refined

A executing program with restricted rights



Enforcing mechanism must not hinder functionality or hurt performance

User vs Kernel: Dr Jekyll and Mr Hyde

<u>Application/User Code (Untrusted)</u>

Run all the processor with all potentially dangerous operations disabled

<u>Kernel Code (Trusted)</u>

Runs directly on processor with unlimited rights

Performs any hardware operations

But run on the same machine!

How can the kernel enforce restricted rights?

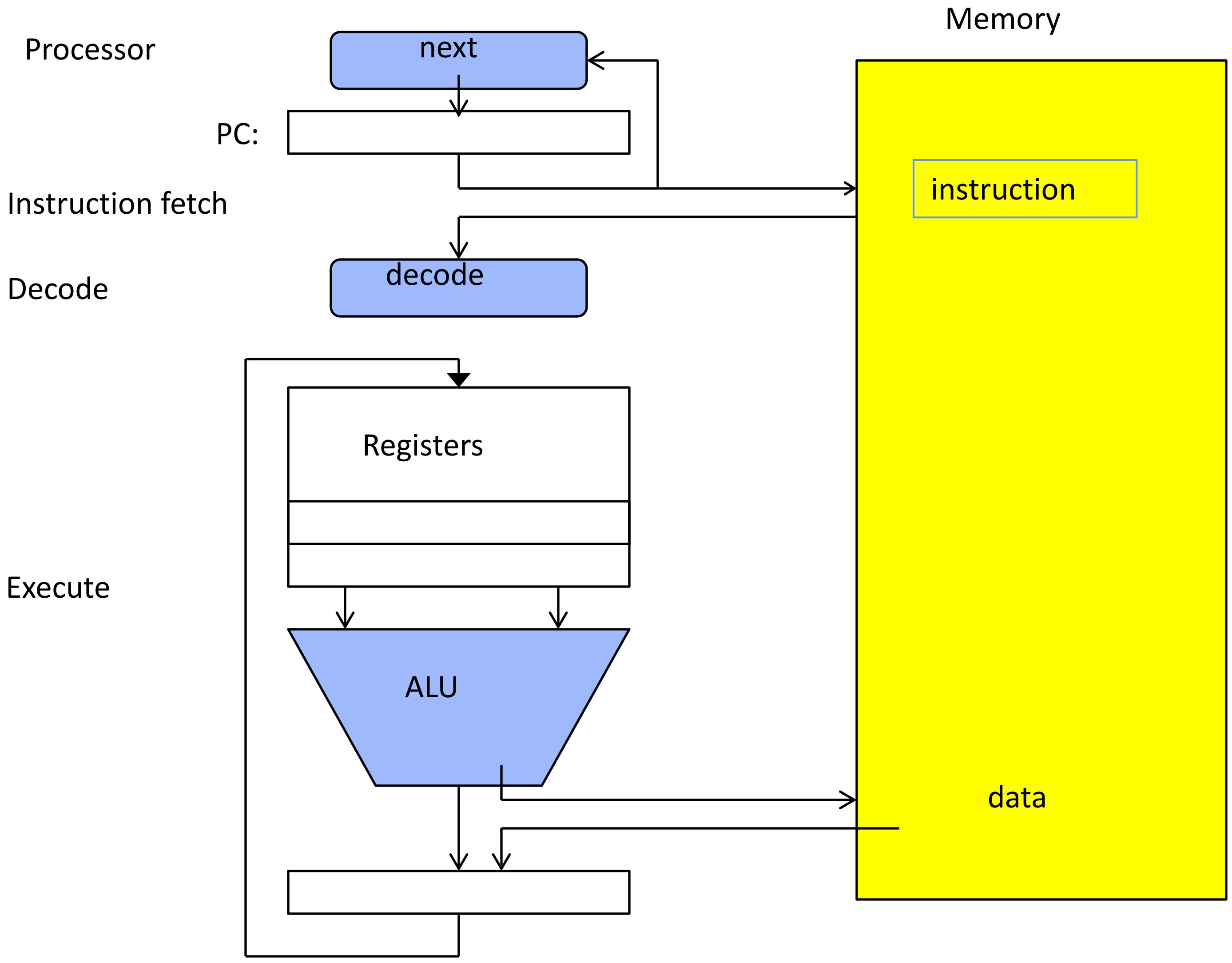
1) While preserving functionality

2) While preserving performance

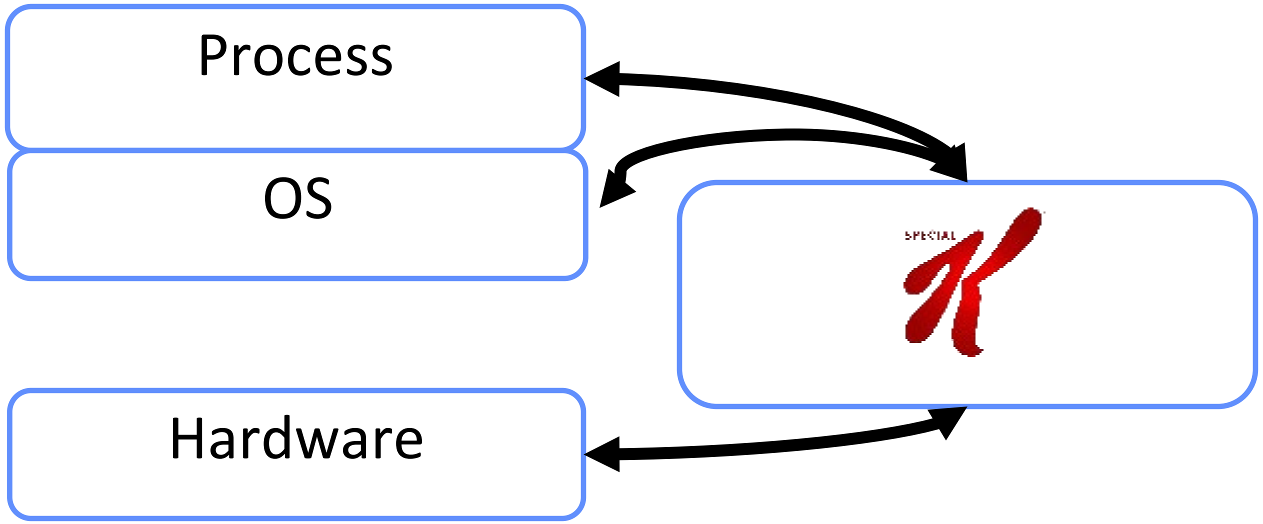
3) While preserving control

Attempt 1: Simulation

Recall: CPU Instruction Cycle (from CS61c)



Attempt 1: Simulation



Have the Kernel interpret and check every instruction!

Potential Issues:

Extremely slow! Would have to cycle through all operations, switch into the kernel, etc.

Unnecessary. Most operations are perfectly safe!

Attempt 2: Dual Mode Operation

Hardware to the rescue!

Use a bit to enable two modes of execution

<u>In User Mode</u>

Processor checks each instruction before executing it

Executes a limited (safe) set of instructions

<u>In Kernel Mode</u>

OS executes with protection checks off

Can execute any instructions

Hardware must support

1) Privileged Instructions

Unsafe instructions cannot be executed in user mode

2) Memory Isolation

Memory accesses outside a process's address space prohibited

3) Interrupts

Ensure kernel can regain control from running process

4) Safe Transfers

Correctly transfer control from user-mode to kernel-mode and back

Req 1/4: Privileged Instructions

Cannot change privilege level (set mode bit)

Cannot change address space

Cannot disable interrupts

Cannot perform IO operations

Cannot halt the processor

How can an application do anything useful …

Asks for permission to access kernel mode!

System calls Transition from user to kernel mode only at specific locations specified by the OS

Exceptions User mode code attempts to execute a privileged operation. Generates a processor exception which passes control to kernel at specific locations

More on safe control transfers later

Hardware must support

1) Privileged Instructions

Unsafe instructions cannot be executed in user mode

2) Memory Isolation

Memory accesses outside a process's address space prohibited

3) Interrupts

Ensure kernel can regain control from running process

4) Safe Transfers

Correctly transfer control from user-mode to kernel-mode and back

Req 2/4: Memory Protection

OS and applications both resident in memory

Application should not read/write kernel memory

(or other apps memory)

A Bug's Tail

The character could leave the game area and start overwriting other running programs and kernel memory.

One of the worst bugs I ever had to deal with was in this game. Once the game player made it to the Colony, every so often the system would crash and burn at totally random times. You might be playing for ten minutes when it happened or ten hours, but it would just die in a totally random way

There was a slow-moving slug like creature that knew how to follow the game player's trail. When it came across another creature, rather than bouncing off and risk losing the trail, I made it so that it would destroy the other creature and stay on target to find you. This worked great, except that on some rare occasions, this slug could do to a wall what it did to the other creatures. That is, it could delete it. This meant that the virtual door was now open for this creature to explore the rest of the RAM on the Macintosh, deleting and modifying it as it went along. Of course, it was just a matter of time before it found some juicy code. In other words, the bug was a REAL bug.

Super Mario Land 2

Mario could exit a level and explore the entire memory of the system

Virtual Memory is Hard!

- Virtualizing the CPU
  - Process Abstraction and API
  - Threads and Concurrency
  - Scheduling
- Virtualizing Memory
  - Virtual Memory
  - Paging
- Persistence
  - IO devices
  - File Systems
- Distributed Systems
  - Challenges with distribution
  - Data Processing & Storage

Hardware must support

1) Privileged Instructions

Unsafe instructions cannot be executed in user mode

2) Memory Isolation

Memory accesses outside a process's address space prohibited

3) Interrupts

Ensure kernel can regain control from running process

4) Safe Transfers

Correctly transfer control from user-mode to kernel-mode and back

Req 3/4: Interrupts

Kernel must be able to regain control of the processor

Hardware to the rescue! (Again x 2)

Hardware Interrupts

Set to interrupt processor after a specified delay or specified event and transfer control to (specific locations) in Kernel.

Resetting timer is a privileged operation