CS4410

Operating Systems

Lecture 3:
Four fundamental OS concepts
Abstractions I: Threads

Rachit Agarwal

Context for today's lecture

- Last lecture (and early parts of today's lecture):
  - Study some of the building blocks of an OS
  - **Understand "why" we need these building blocks**
  - And what are the **conceptual challenges in designing them**

- **Today, and next couple of lectures**
  - Understand the abstractions offered by the OS
    - **Threads**, Process, Virtual memory, Files, Sockets, Signals, ..
  - **Why** they are designed the way they are designed
  - What are the **tradeoffs** in different design decisions
  - Some interesting details on how they are implemented

Goal of Today's Lecture

- Wrap up discussion on four fundamental concepts in OS
- Deeper dive into threads

Recall: What does an OS do?

- Enables **convenient "abstractions"** for applications to access hardware
  - **CPU:** threads
  - **Memory:** virtual memory
  - **Storage devices:** files
  - **Network:** sockets
  - **Server:** collection of resources needed by an application (processes, VM,..)
- **Manages** hardware resources
  - Resource **allocation, sharing and isolation**
- Implements **common services** for applications
  - Security, protection and authentication
  - Reliability
  - Communication
  - Input/output operations
  - Program execution
  - ....

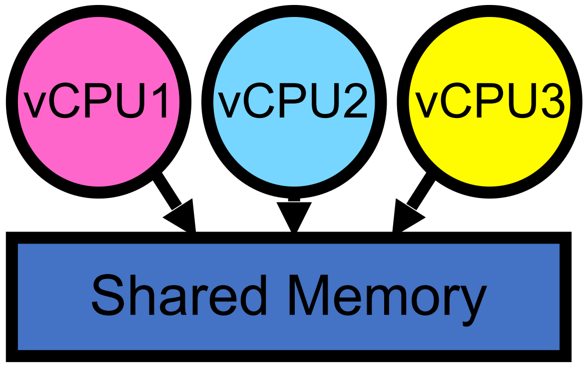
Recall: Four Fundamental OS Concepts

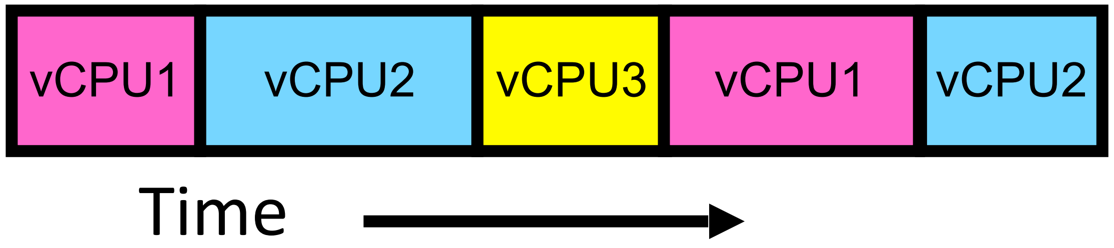
- **Thread: Execution Context**
  - A single, sequential execution context
- **Address space** (with **translation**)
  - Program's view of memory is distinct from physical memory
- **Process: an instance of a running program**
  - Address Space + One or more Threads + …
- **Protection/Isolation**
  - Only the “system” can access certain resources
  - Combined with translation, isolates programs from each other

Recall: Threads

- **Definition: A single, sequential *execution context***
    - A "virtual" core
    - Executes a sequence of instructions, in order, on a physical core
        - Only one thing happens at a time

- **Why threads?**
    - *Statistical multiplexing:* improved utilization of physical cores





- **Challenges:**
    - synchronization (correctness), scheduling (performance)

Recall: Virtual address space

- **Physical** address space: where the data *actually* resides
- “**Virtual”** address space: where the program *thinks* the data resides
- **Why virtual address space?**
    - *Statistical multiplexing*: improved utilization of physical memory
    - *Protection/Isolation* (not yet covered)
    - ….
- **Challenges?**
    - Efficient address translation

Recall: Challenge of efficient address translation

- Programs use virtual addresses

- As a program runs, virtual addresses translated to physical addresses

- Address translation must be extremely light-weight (in the common case)
    - To keep the overheads low

- Two ideas:
    - Perform address translation in hardware
    - Maintain a lookup table (virtual —> physical)

- To achieve efficiency:
    - Small size of lookup table (why?)
    - Fast algorithms to perform a lookup

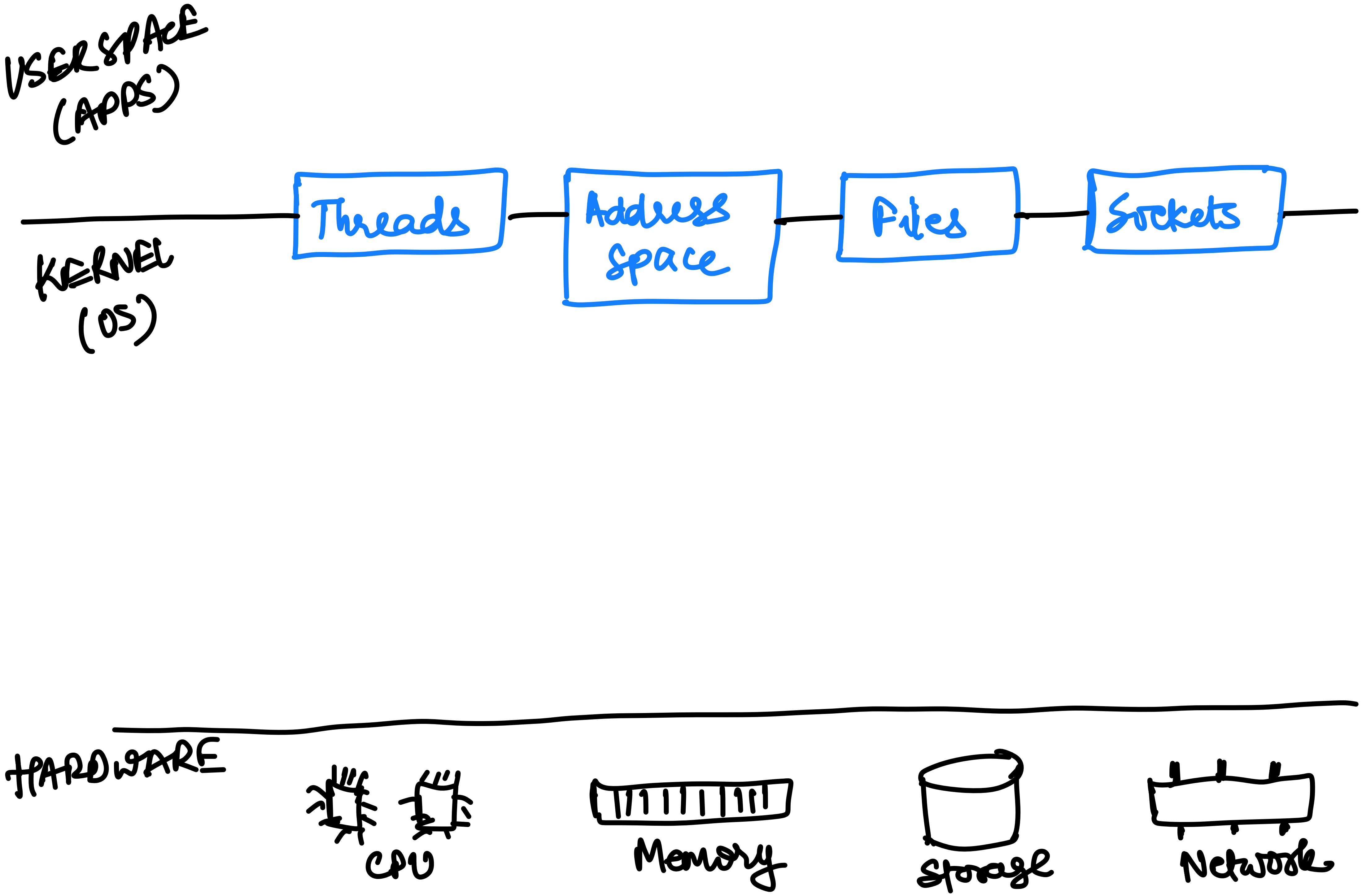
Achieving efficiency using "pages"

- Divide virtual address spaces into contiguous chunks of fixed size (say X)
  - Call each chunk a page (usually X = 4096 bytes)
- Map each page to 4KB of *contiguous* physical address space
- If page size is X, a virtual address v is at
  - (assuming addresses/offsets start with 0)
  - page number: **floor(v/X)**
  - Offset: **v - X*floor(v/X) - 1**
  - E.g., X=4096; v = 4097 is on page 1, offset 0
- Pages enable efficiency:
  - Smaller lookup table size
    - Reduced by a factor of X
    - Compared to mapping each individual address
  - Enable faster algorithms to perform a lookup (later)

Questions?

Today: Four Fundamental OS Concepts

- **Thread: Execution Context**
  - A single, sequential execution context
- **Address space** (with **translation**)
  - Program's view of memory is distinct from physical memory
- **Process: an instance of a running program**
  - Address Space + One or more Threads + …
- **Protection/Isolation**
  - Only the “system” can access certain resources
  - Combined with translation, isolates programs from each other

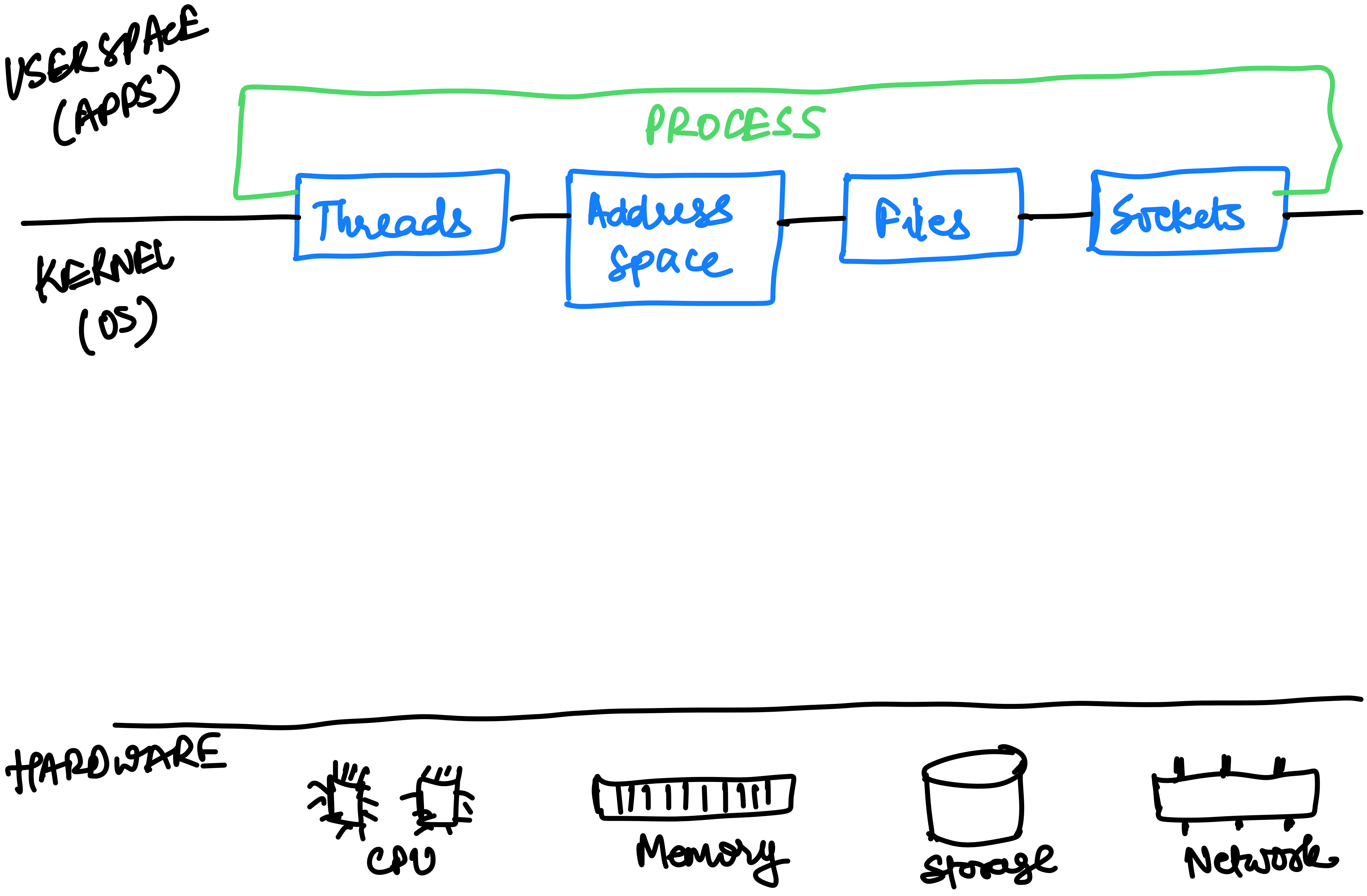


USERSPACE
(APPS)
KERNEL
(OS)
Threads
Address Space
Files
Sockets
HARDWARE
CPU
Memory
Storage
Network

Process

- **Definition: execution environment with restricted rights**
  - One or more threads
  - Execution state: everything that can affect, or be affected by, a thread
    - Code, data, registers, call stack, files, sockets, etc.
  - Part of the process state is "owned" by individual threads
  - Part is shared among all threads in the process

- **Each process has a "state"—Process control block (PCB)**
  - Execution state for each thread
  - Scheduling information
  - Information about memory used by the process
  - Information about files, sockets, etc.
  - ….


USERSPACE (APPS)
PROCESS
Threads
Address Space
Files
Sockets
KERNEL (OS)
HARDWARE
CPU
Memory
Storage
Network

Evolution of OS process model

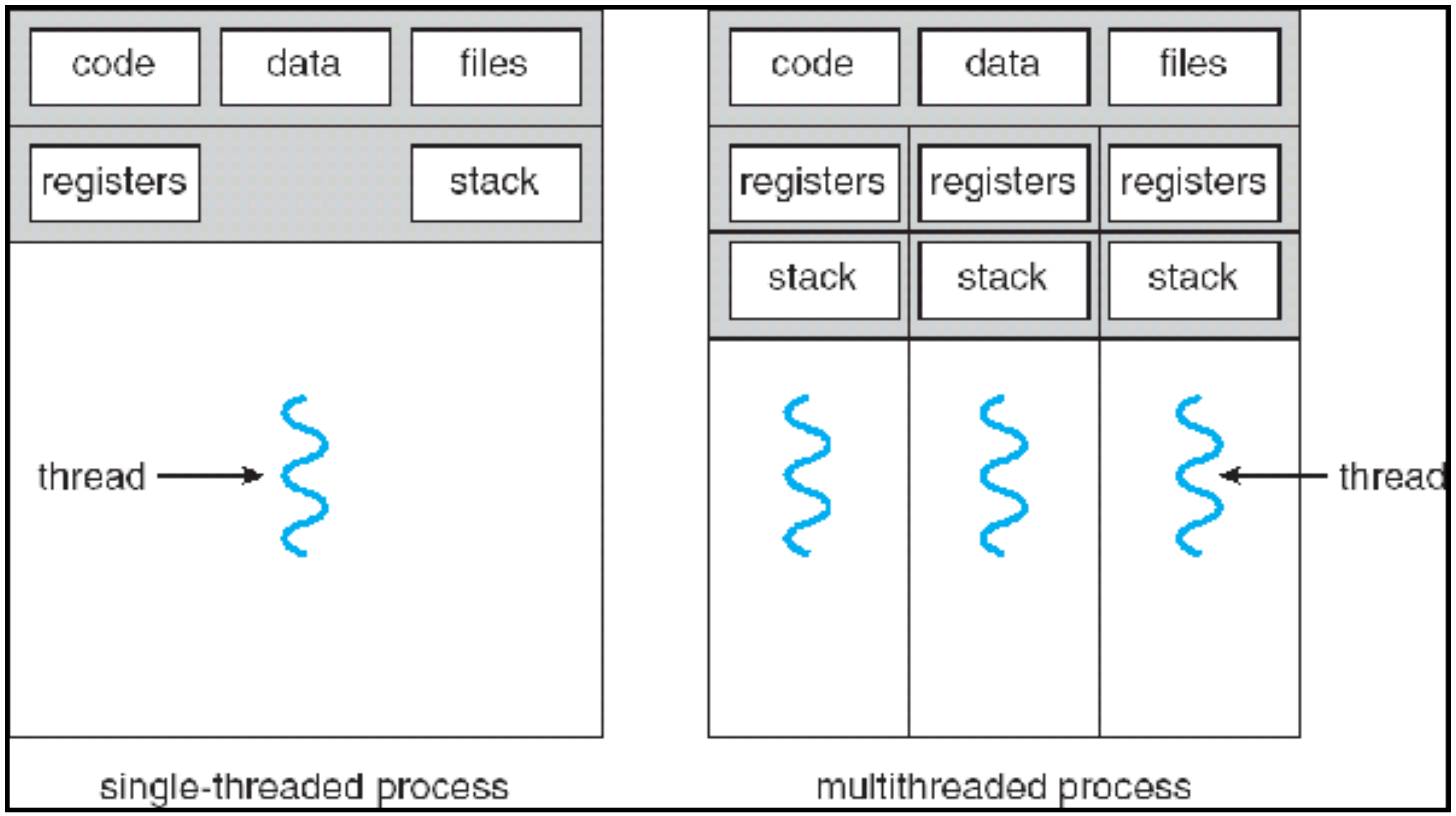
- **Early operating systems: single tasking**
  - Single process, single thread
  - “switch” applications over long timescales
  - Problem?

- **Late 1970s: multitasking**
  - Multiple processes, single thread per process
  - Share resources across processes
  - Problem?

- **1990s: multitasking, multithreading**
  - Multiple processes, multiple threads
  - Challenges?

Single and Multithreaded Processes

- Why have multiple threads within the same process?
- Threads encapsulate concurrency

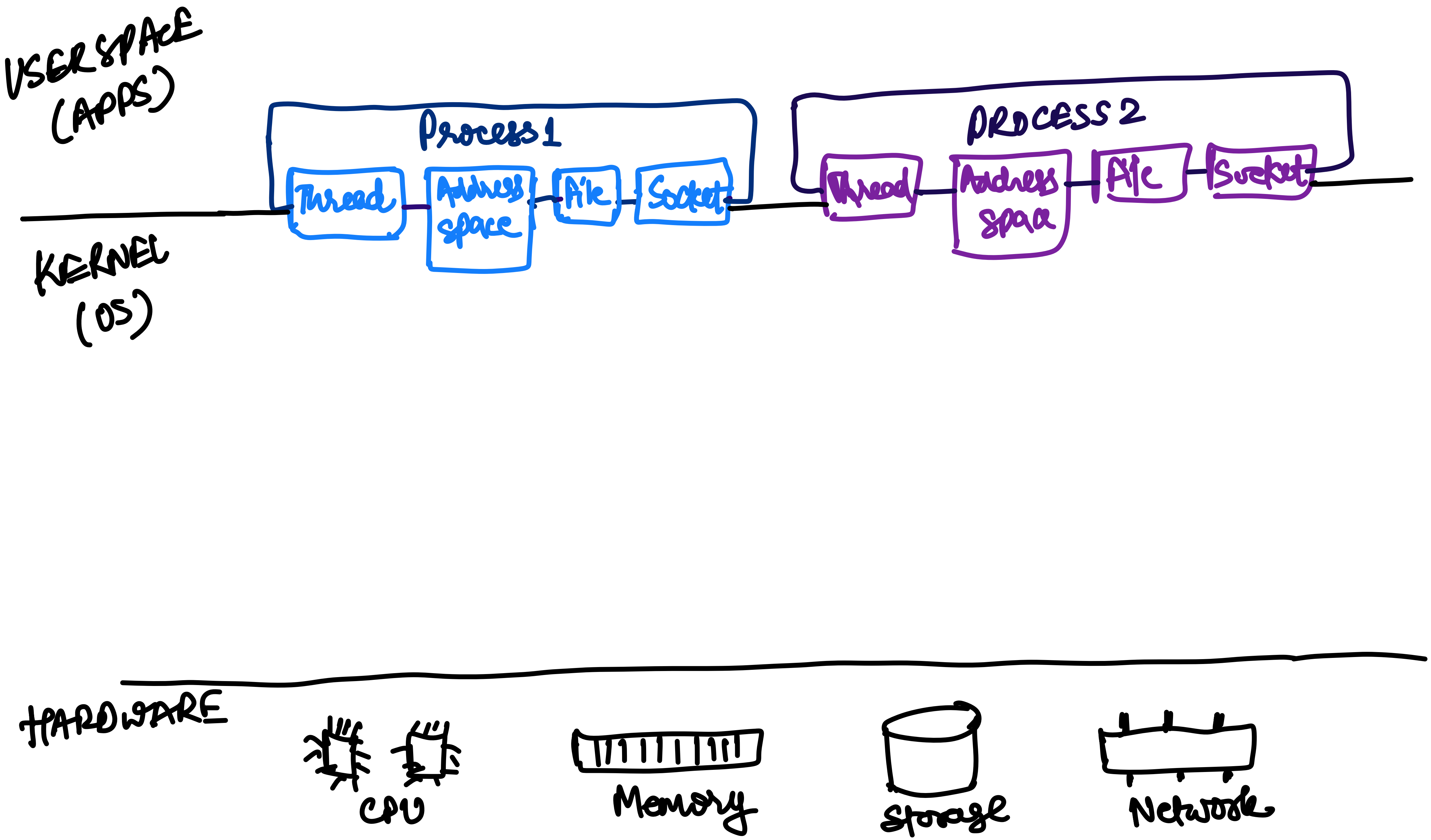


Questions?

Today: Four Fundamental OS Concepts

- **Thread: Execution Context**
    - A single, sequential execution context
- **Address space** (with **translation**)
    - Program's view of memory is distinct from physical memory
- **Process: an instance of a running program**
    - Address Space + One or more Threads + …
- **Protection/Isolation**
    - Only the “system” can access certain resources
    - Combined with translation, isolates programs from each other

An OS may run multiple concurrent processes



The core challenge with multiple processes?

- Protection/Isolation/Sharing
  - **Reliability:** buggy processes can only hurt themselves
  - **Security:** a process does not have to trust other processes
  - **Fairness:** a good granularity to enforce fair utilization of resources

- **Mechanisms to enable isolation:**
  - **Virtualization**
    - Virtual cores, virtual address space (in particular)
  - **Dual mode operations**
    - Only the OS can access certain resources

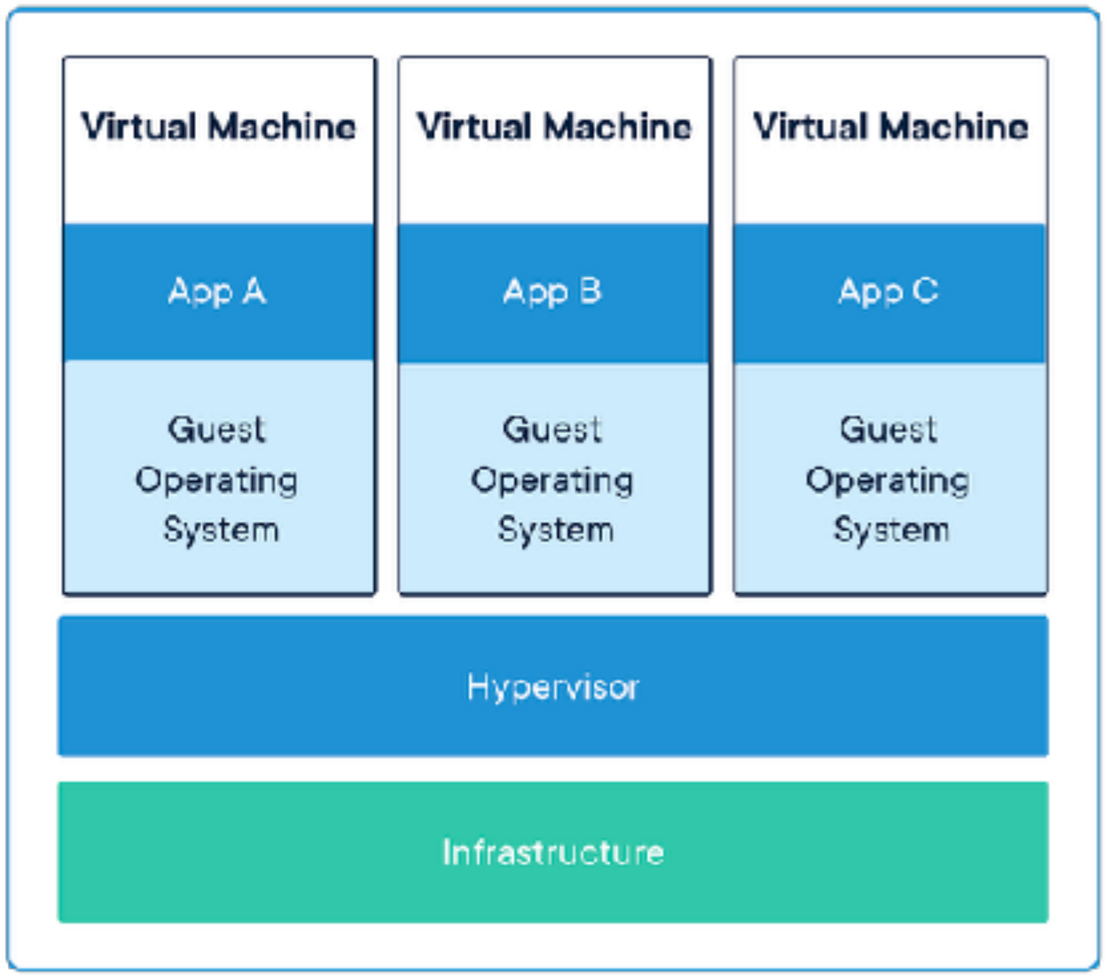
Dual mode operation

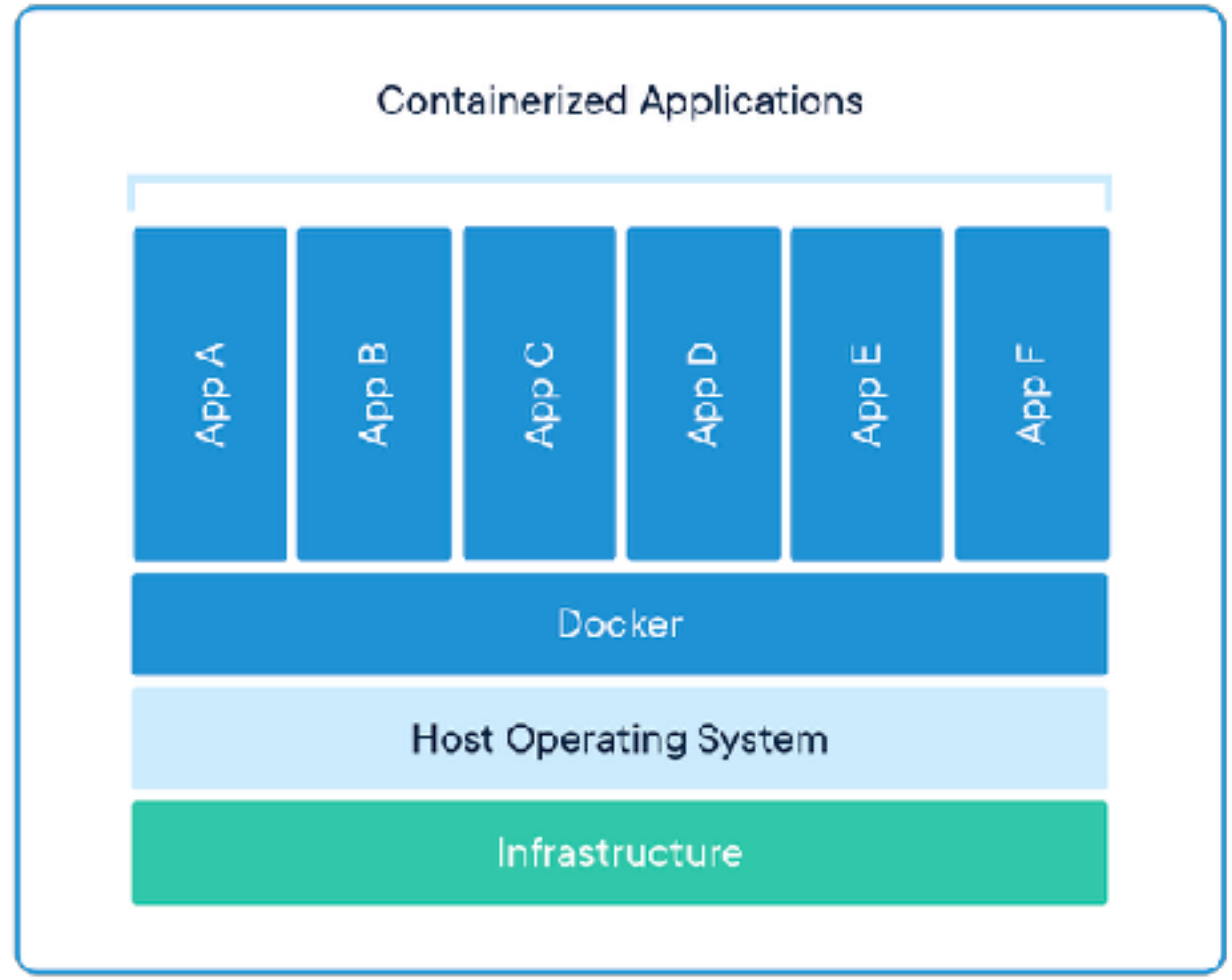
- **Hardware provides at least two modes of operations:**
    - Kernel mode (or “supervisor” / “protected” mode)
    - User mode
- **Processes (i.e., programs you run) execute in user mode**
    - Certain operations are prohibited when running in user mode
        - *E.g.*, changing the page table pointer
    - To perform privileged actions, processes request services from the OS
- **Kernel executes in kernel mode**
    - Performs privileged actions to support running processes
    - Configures hardware for proper protection (e.g., address translation)
- **“Controlled” transitions between user mode and kernel mode**
    - System calls, interrupts, exceptions

User to Kernel Mode Transfers

- **Syscalls**
  - Process requests a system service, e.g., exit
  - Like a function call, but “outside” the process
- **Interrupts**
  - External asynchronous event
  - e.g., I/O operations
- **Trap or exception**
  - Internal synchronous event in process
  - e.g., protection violation (segmentation fault), divide-by-zero, …

Additional layers of protection for modern systems





- **In many modern large-scale deployments**
  - Run a complete OS in a "virtual machine"
  - Package all libraries associated with an application into a "container"

- More on this later in the course

Questions?

Abstraction I: Threads

Diving one more level deeper: Threads

- **Thread:** A single, sequential execution context
    - A single execution sequence that can be scheduled independently
- Provide a mechanism for **concurrency** and **parallelism**
- Protection is an orthogonal concept
    - A protection domain can contain one thread or more

Concurrency vs. Parallelism

- **Concurrency** is about handling multiple things
- **Parallelism** is about doing multiple things *simultaneously*
- Example: Two threads on a single-core system without hyperthreading…
  - … execute concurrently …
  - … but *not* in parallel
- What does it mean to run two threads concurrently?
  - Scheduler is free to run threads in any order and interleaving
  - Thread may run to completion or time-slice in chunks

Need for Threads

- Consider the following program:

```
main() {
    ComputePI();
    PrintClassList("classlist.txt");
}
```

- What output do you expect?
- Would the program ever print out class list?
    - No! Why?
    - **`ComputePI`** would never finish

With Threads

- Version of program with threads (loose syntax):

```
main() {
     create_thread(ComputePI());
     create_thread(PrintClassList("classlist.txt"));
}
```

- What output do you expect?

- Now, you would actually see the class list
  - But only "now and then"
  - **Illusion: infinite number of processors (potentially varying speeds)**

- `create_thread`: Spawns a new thread running the given procedure
  - Should behave as if another CPU is running the given procedure

Threads Mask "Idle" periods

- A thread is in one of the following three states:
  - RUNNING — running
  - READY — eligible to run, but not currently running
  - BLOCKED — ineligible to run

- If a thread cannot proceed (e.g., waiting for an I/O request to be finished)
  - The OS marks it as BLOCKED

- Once the thread is ready, the OS marks it as READY
  - Can now be scheduled

- Once the thread is scheduled, the OS marks it as RUNNING
  - Actually using the physical core now

Another example for Threads

- Version of program with threads (loose syntax):

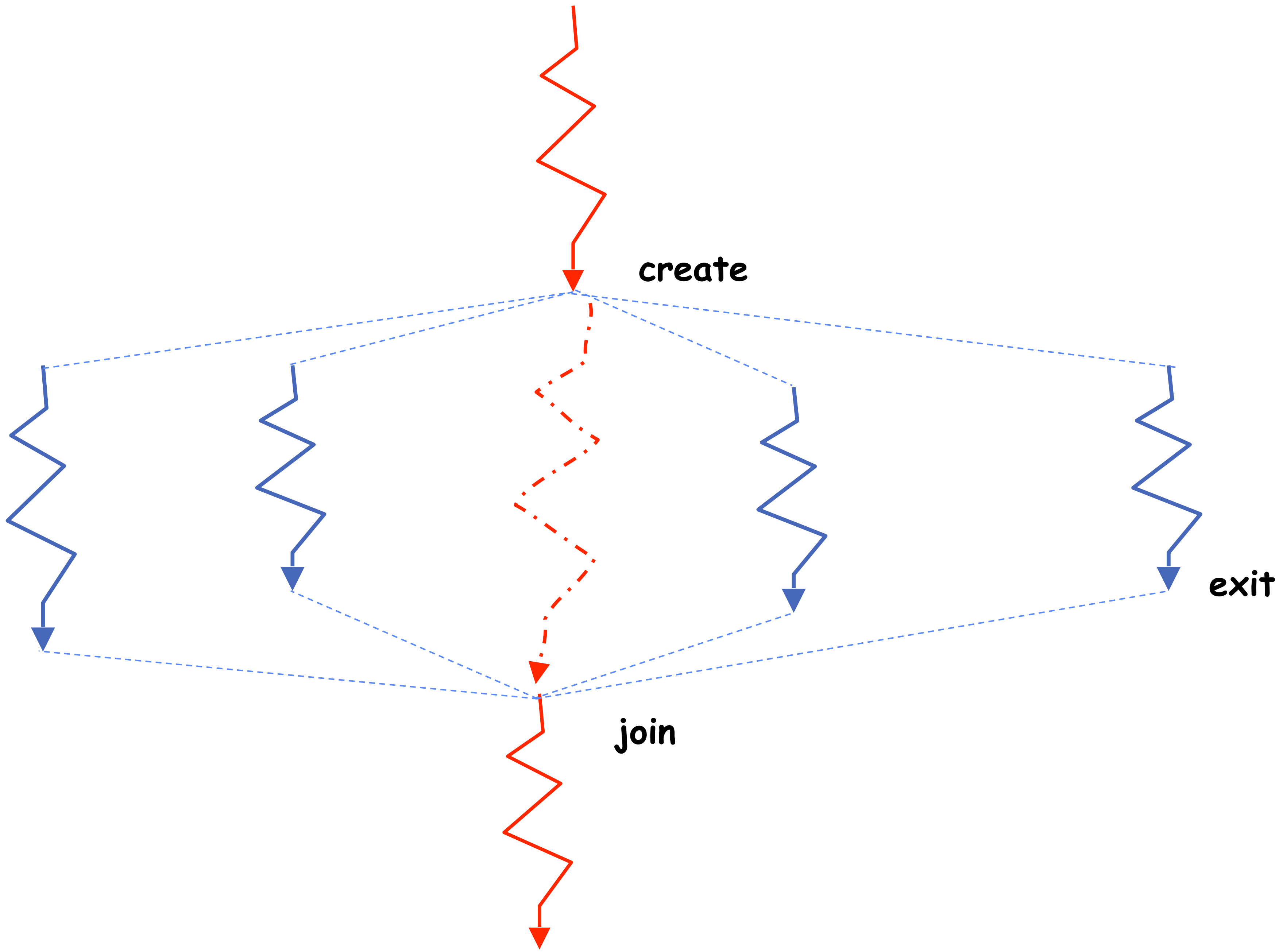
```
main() {
    create_thread(RenderUserInterface);
    create_thread(PrintClassList(“classlist.txt”));
}
```

- What is the behavior here?
    - Still respond to user input
    - While reading file in the background

Multithreaded Programs

- When you compile a C program and run the executable
  - It creates a process that is executing that program
- Initially, this new process has *one thread* in its own address space
  - With code, globals, etc. as specified in the executable
- How can we make a multithreaded process?
  - A process can issues *syscalls* to create new threads
  - These new threads are part of the process:
    - They share its address space

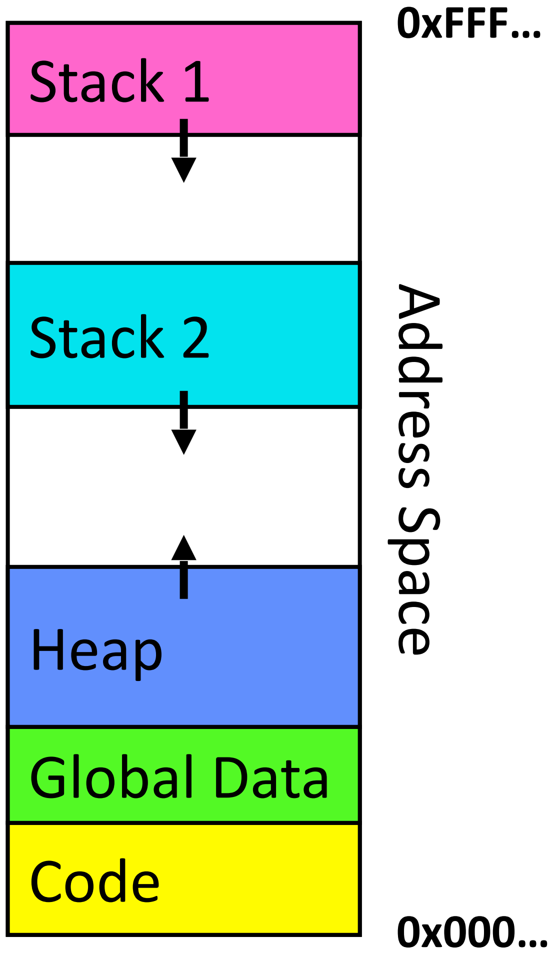
New Idea: Fork-Join Pattern



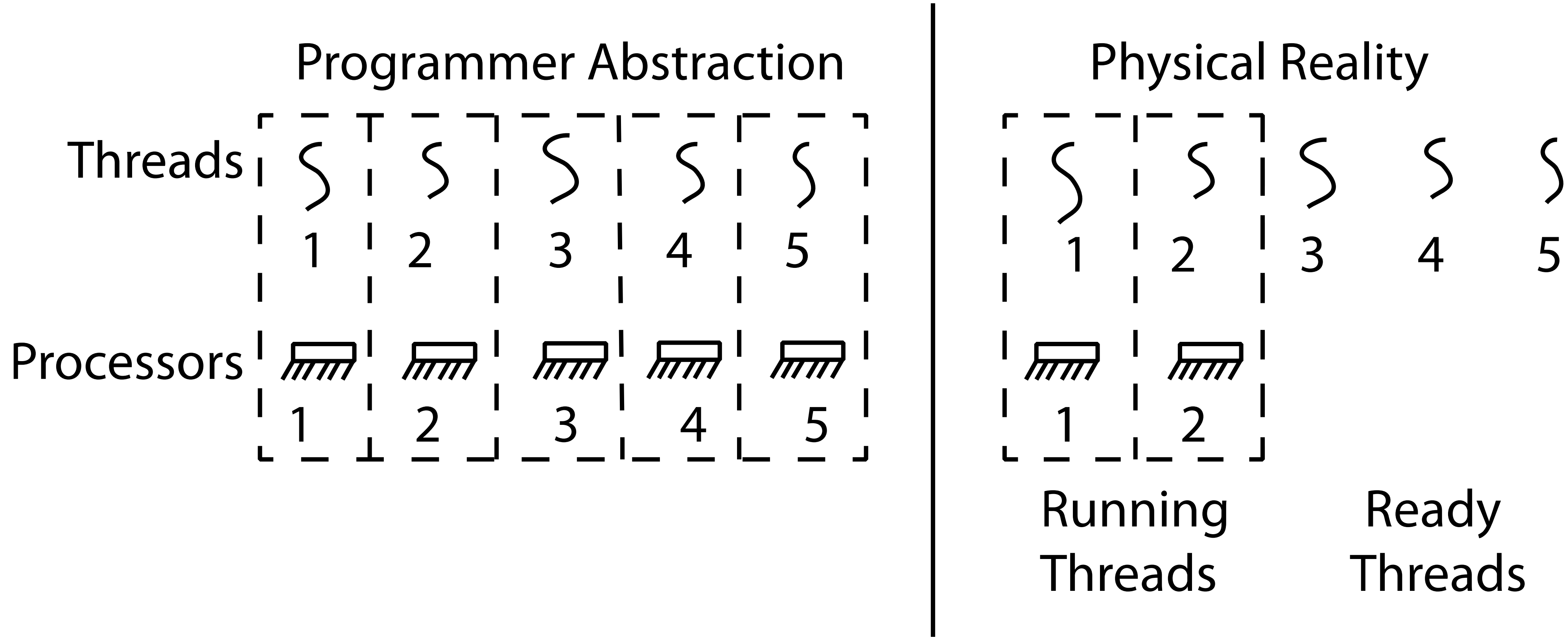
- Main thread *creates* (forks) collection of sub-threads passing them args to work on…
- … and then *joins* with them, collecting results.

Memory Layout with Two Threads

- Two sets of CPU registers
- Two sets of stacks
- Issues:
  - How do we position stacks relative to each other?
  - What maximum size should we choose for the stacks?
  - What happens if threads violate this?
  - How might you catch violations?



Thread Abstraction



- **Illusion: infinite number of processors, potentially varying speeds**
- Reality: threads execute with variable "speed"
  - Why?
  - Depends on scheduling policies
- Programs must be designed to work with any schedule

Programmer vs. Processor View

Programmer's View

.
.
.
x = x + 1;
y = y + x;
z = x +5y;
.
.
.

Possible Execution #1

.
.
.
x = x + 1;
y = y + x;
z = x + 5y;
.
.
.

Possible Execution #2

.
.
.
x = x + 1
....
thread is suspended
other thread(s) run
thread is resumed
....
y = y + x
z = x + 5y

Possible Execution #3

.
.
.
x = x + 1
y = y + x
....
thread is suspended
other thread(s) run
thread is resumed
....
z = x + 5y

Correctness with Concurrent Threads

- Goal: Correctness by Design
  - What makes this a challenging goal?
- Non-determinism:
  - Scheduler can run threads in any (non-deterministic) order
    - Why?
  - Scheduler can switch threads at any time
    - Why?
- Independent Threads
  - No state shared with other threads
  - Deterministic, reproducible conditions
- Cooperating Threads
  - Shared state between multiple threads

Race Conditions

- Initially x == 0 and y == 0

Thread A	**Thread B**
`x = 1;`	`y = 2;`

- What are the possible values of x below after all threads finish?
- Must be **1**. Thread B does not interfere with Thread A.

Race Conditions

- Initially x == 0 and y == 0

Thread A

```
x = y + 1;
```

Thread B

```
y = 2;
y = y * 2;
```

- What are the possible values of x below?
- 1 or 3 or 5 (non-deterministic)
- Race Condition: Thread A “races” against Thread B!

Definitions

- **Synchronization**:
  - Thread coordination, usually regarding shared data
- **Mutual Exclusion:**
  - Ensuring only one thread does a particular thing at a time
  - Type of synchronization
- **Critical Section:**
  - Part of code that can be executed by exactly one thread at once
  - Result of mutual exclusion
- **Lock:**
  - An object that can be held by only one thread at a time
  - Provides mutual exclusion