Alternative OS Process Models

COS 417: Operating Systems

Spring 2025, Princeton University

Processes, Revisited

Processes, Revisited

Multiplexing

Share a **single physical** resource among **multiple processes**.

Processes, Revisited

Multiplexing

Share a **single physical** resource among **multiple processes**.

Virtualization

Take an existing resource and transform it into an (often) more general, powerful and easy to use **virtual** form of itself.

Processes, Revisited

Multiplexing

Share a **single physical** resource among **multiple processes**.

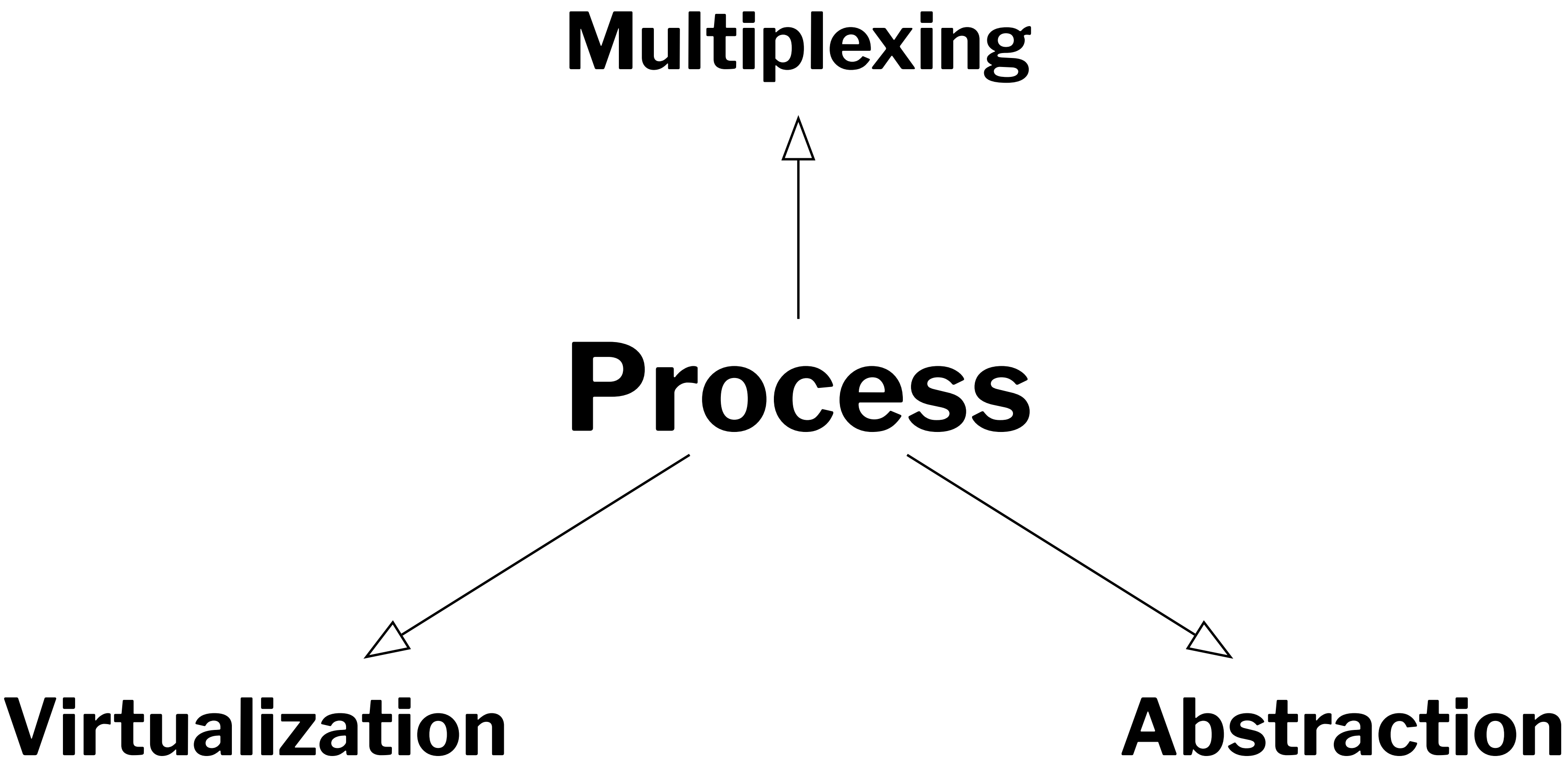
Virtualization

Take an existing resource and transform it into an (often) more general, powerful and easy to use **virtual** form of itself.

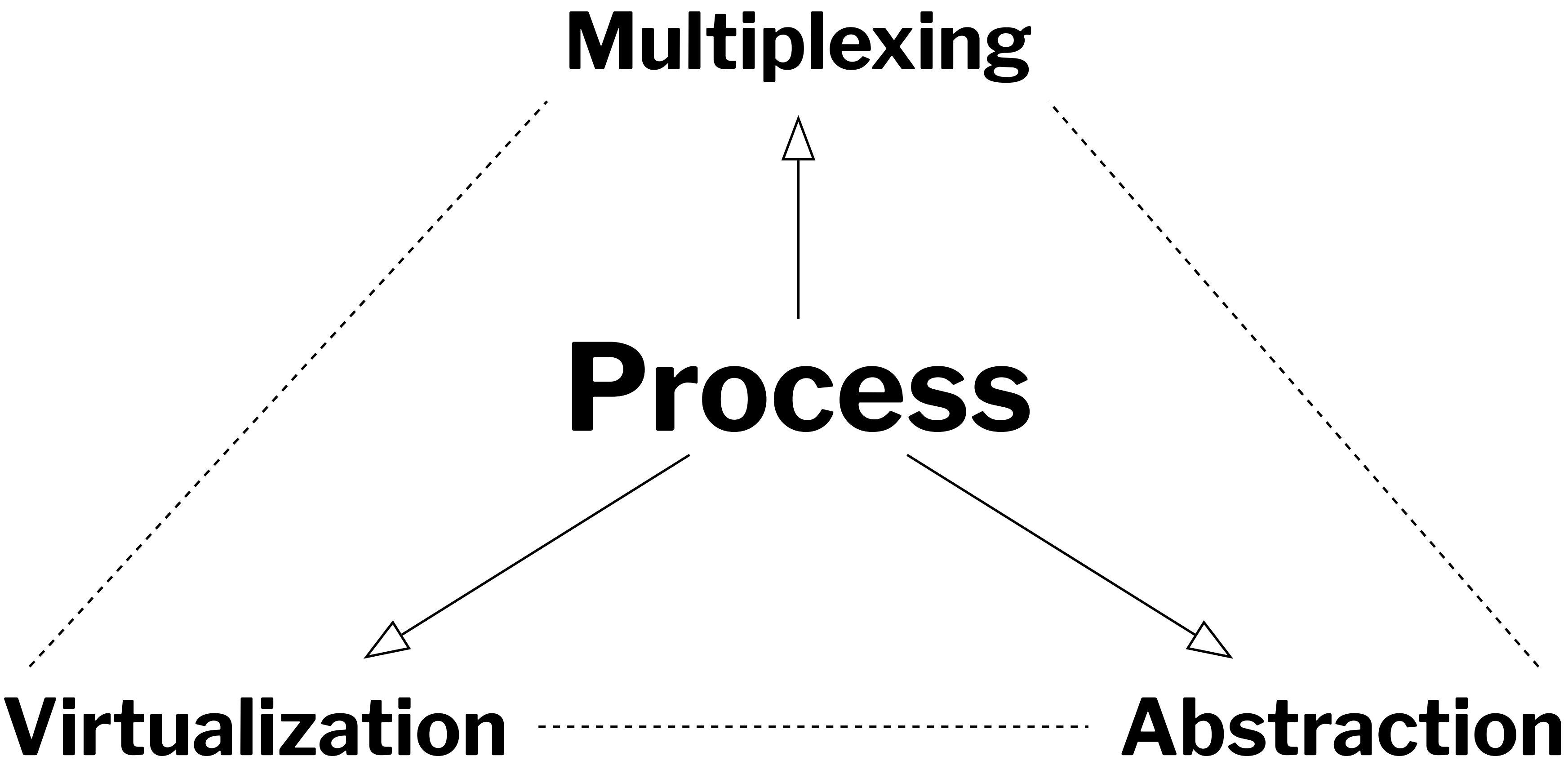
Abstraction

Take a *low level* resource and use it to provide a **higher level** (e.g., easier or more expressive) interface that is significantly different.

Processes provide *all* of the above



Processes provide *all* of the above



Through these properties, processes can...

Through these properties, processes can...

Introduce Concurrency

Run your web browser, music player and weather app side by side, without explicit coordination.

Through these properties, processes can...

Introduce Concurrency

Run your web browser, music player and weather app side by side, without explicit coordination.

Provide Isolation

Processes cannot (generally) access each other's data, and a *stuck* or crashed process does not affect others.

Through these properties, processes can...

Introduce Concurrency

Run your web browser, music player and weather app side by side, without explicit coordination.

Provide Isolation

Processes cannot (generally) access each other's data, and a *stuck* or crashed process does not affect others.

Enable Portability

Processes program against an *abstract* machine (e.g., `fork`, `wait`).

Processes are pretty great!

So … why not just use them?

why are we still talking about this?

Diverse requirements & constraints!

All computers run multiple apps, no?

Diverse requirements & constraints!

In many systems: single application!

A ton of systems just run a single application! They don't need to run **independent** processes concurrently (*i.e., without coordination*).

Diverse requirements & constraints!

In many systems: single application!

A ton of systems just run a single application! They don't need to run **independent** processes concurrently (*i.e., without coordination*).

But there's no disadvantages to processes!

Diverse requirements & constraints!

In many systems: single application!

A ton of systems just run a single application! They don't need to run **independent** processes concurrently (*i.e., without coordination*).

Real systems only have finite resources!

Overprovisioning of resources (e.g., memory, CPU) can lead to "starvation". Apps can get sluggish, miss important deadlines (like buffering video), and even need to be terminated by the OS.

Diverse requirements & constraints!

What about performance and overheads?

Diverse requirements & constraints!

Processes themselves introduce overheads.

Virtualizing resources of a machine by **context switching** between processes takes time. Tracking process state consumes memory.

Diverse requirements & constraints!

Processes themselves introduce overheads.

Virtualizing resources of a machine by **context switching** between processes takes time. Tracking process state consumes memory.

Virtualization and abstraction can impact performance.

Using virtualized resources or high-level abstractions can prevent an application from taking advantage of the hardware's full potential.

Diverse requirements & constraints!

Processes themselves introduce overheads.

Virtualizing resources of a machine by **context switching** between processes takes time. Tracking process state consumes memory.

Virtualization and abstraction can impact performance.

Using virtualized resources or high-level abstractions can prevent an application from taking advantage of the hardware's full potential.

At least processes provide isolation & security!

Diverse requirements & constraints!

Processes themselves introduce overheads.

Virtualizing resources of a machine by **context switching** between processes takes time. Tracking process state consumes memory.

Virtualization and abstraction can impact performance.

Using virtualized resources or high-level abstractions can prevent an application from taking advantage of the hardware's full potential.

Process isolation is often imperfect.

Side channels leak information between processes (like `wait` time).

UNIX Processes are *still* pretty neat!

But there's a plethora of other approaches,
each with their own tradeoffs!

Alternative OS Process Models

Desktop-class Operating Systems

Not much variety. Linux, Android, macOS, iOS, … all UNIX-inspired (`fork + exec` model). Windows has a `spawn`-like API (`CreateProcess`).

Alternative OS Process Models

Desktop-class Operating Systems

Not much variety. Linux, Android, macOS, iOS, … all UNIX-inspired (`fork + exec` model). Windows has a `spawn`-like API (`CreateProcess`).

Embedded, Cloud, Accelerators…

A ton of different approaches!

Work around one or more of the issues mentioned previously.

Let's Explore the Design Space

Do we need processes at all?

Many applications don’t need **virtualization** or **multiplexing**!

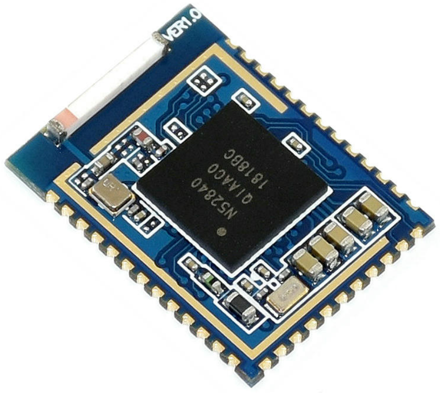
Can you think of examples?

Do we need processes at all?

Many applications don’t need **virtualization** or **multiplexing**!

Do we need processes at all?

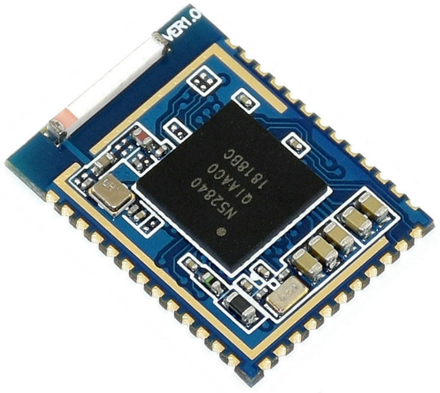
Many applications don't need **virtualization** or **multiplexing**!



Do we need processes at all?

Many applications don't need **virtualization** or **multiplexing**!

```
def lightswitch_main():
  state = False # off on startup
  while True:
    if button.read() == True:
      # button was pressed!
      state = not state
      broadcast_new_state(state)
```



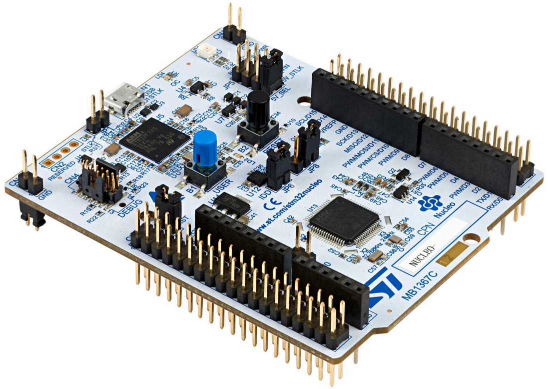
Typically referred to as "**bare metal programming**". A single process has full, direct and sole control over the hardware.

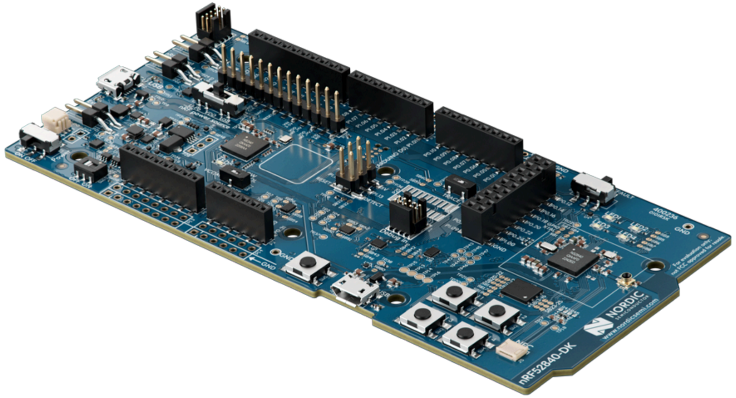
Bare-metal programs can still use abstractions!

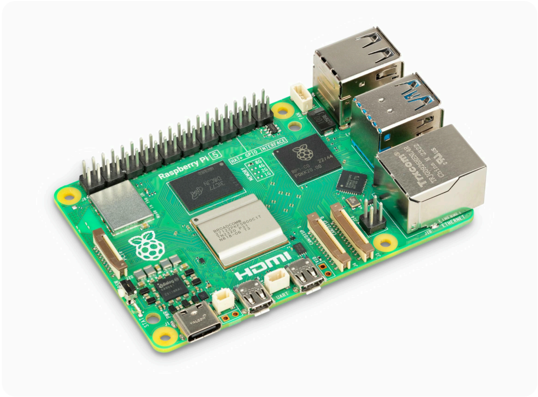
Write **portable** code using functions like

`button.read()`, `broadcast_new_state()`, ...

Run on many **different** hardware systems:







Bare-metal programs can still use abstractions!

Write **portable** code using functions like

`button.read(), broadcast_new_state(), ...`

Library Operating Systems and Unikernels

Provide abstractions similar to those of full OSes.

Do **not** run multiple independent or interacting applications!

Enables applications to have **more predictable timing**, **fixed resource allocations** and a **high degree of control** over the hardware.

Multiplexing without Virtualization

Basic assumption so far: we have more applications than processors, so we have to virtualize CPUs...

Multiplexing without Virtualization

Basic assumption so far: we have more applications than processors, so we have to virtualize CPUs…

What if we had more CPUs than applications?

Multiplexing without Virtualization

Basic assumption so far: we have more applications than processors, so we have to virtualize CPUs...

What if we had more CPUs than applications?

Multiplexing without Virtualization

Basic assumption so far: we have more applications than processors, so we have to virtualize CPUs...

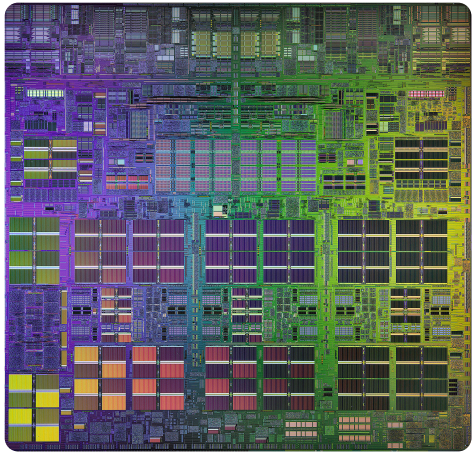
What if we had more CPUs than applications?

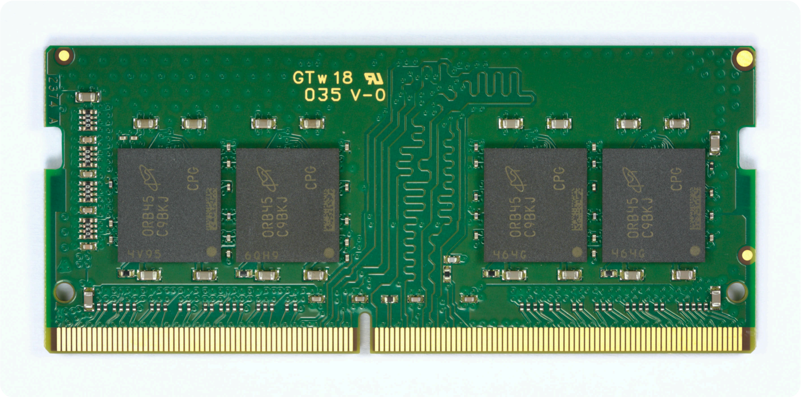
Logical Partitions

Divide a physical system into multiple partitions (*slices*), each running their own process / OS.

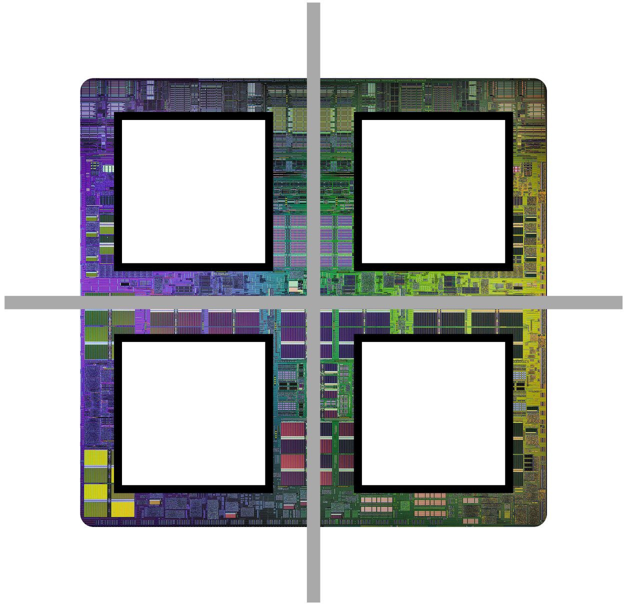
Each paritition has **direct**, but **restricted** access to the underlying hardware, constrained to their assigned physical partition.

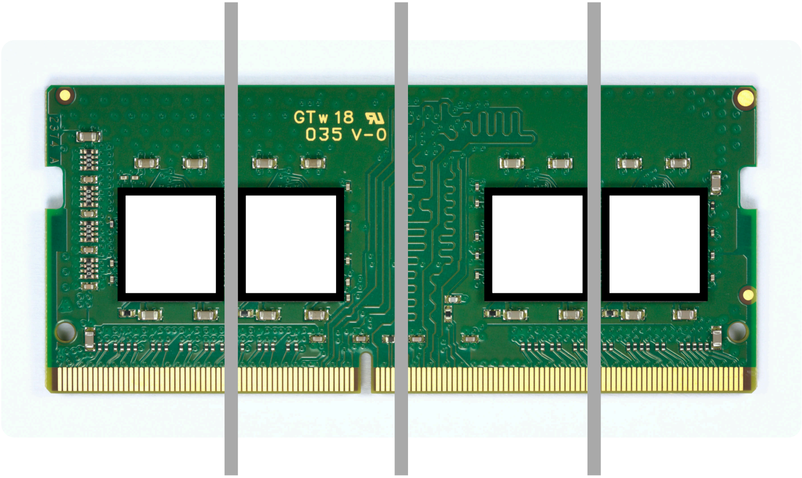
Logical Partitions, Illustrated



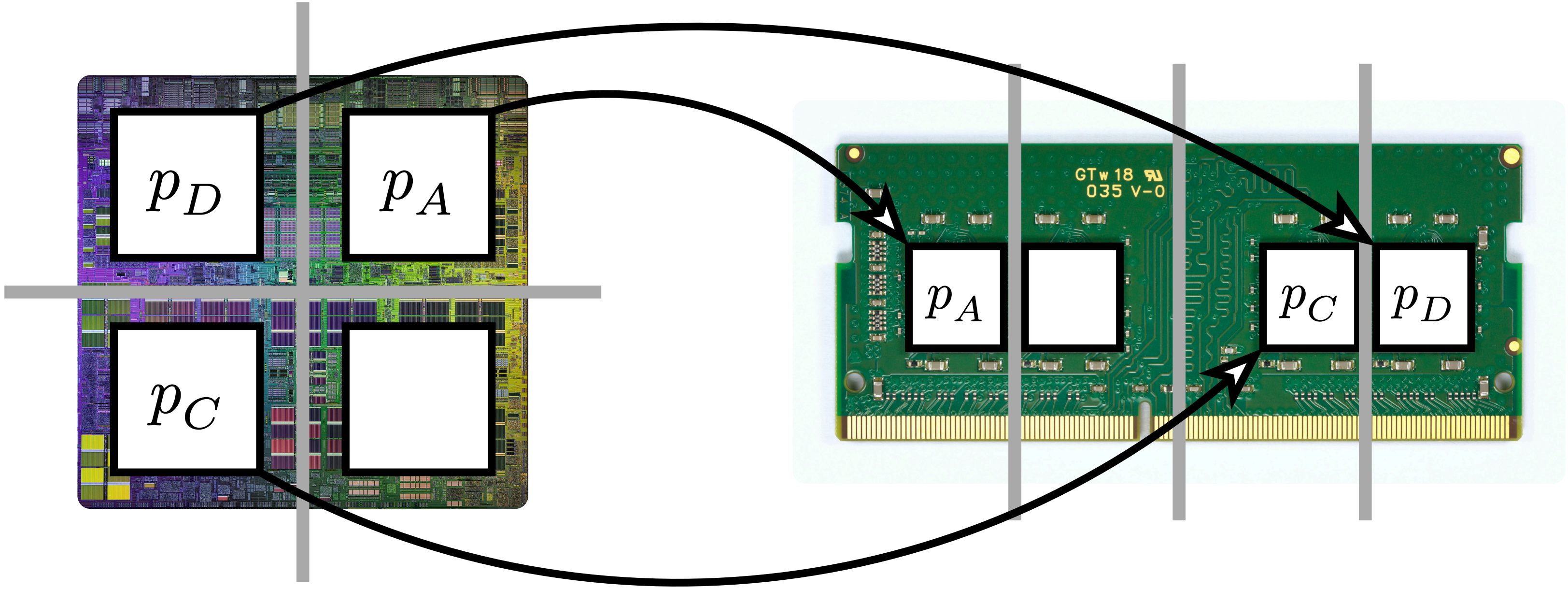


Logical Partitions, Illustrated

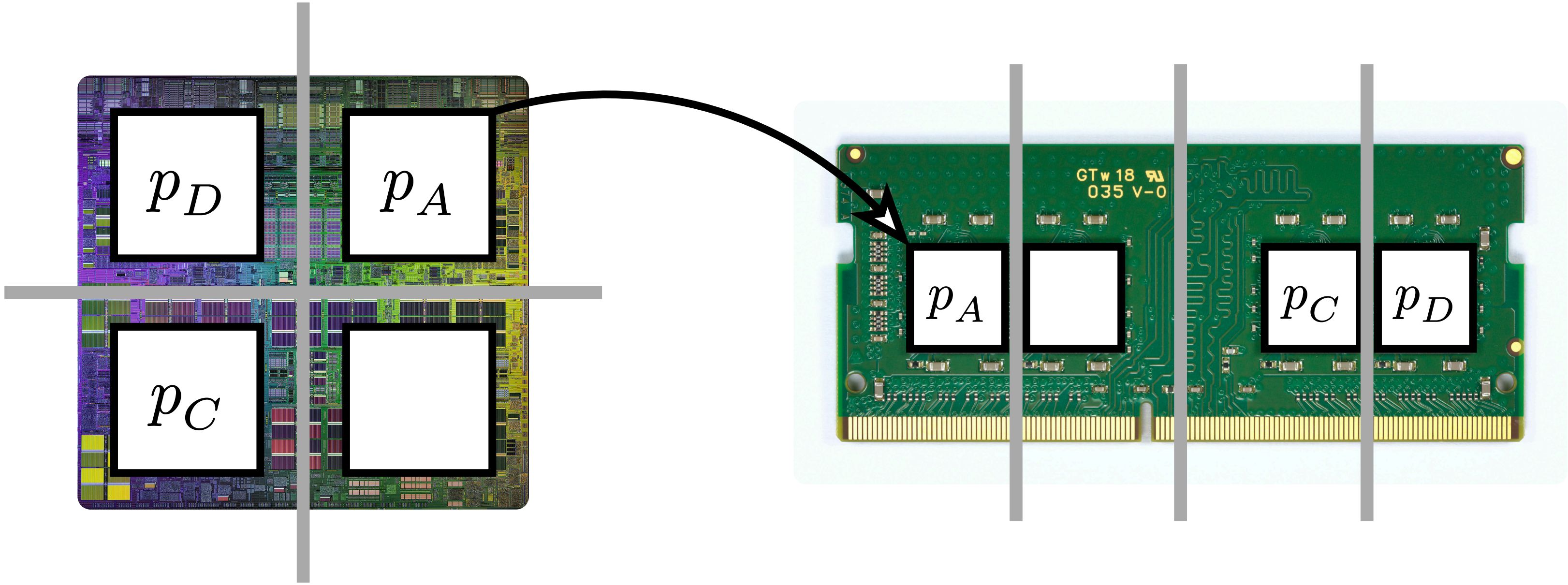




Logical Partitions, Illustrated

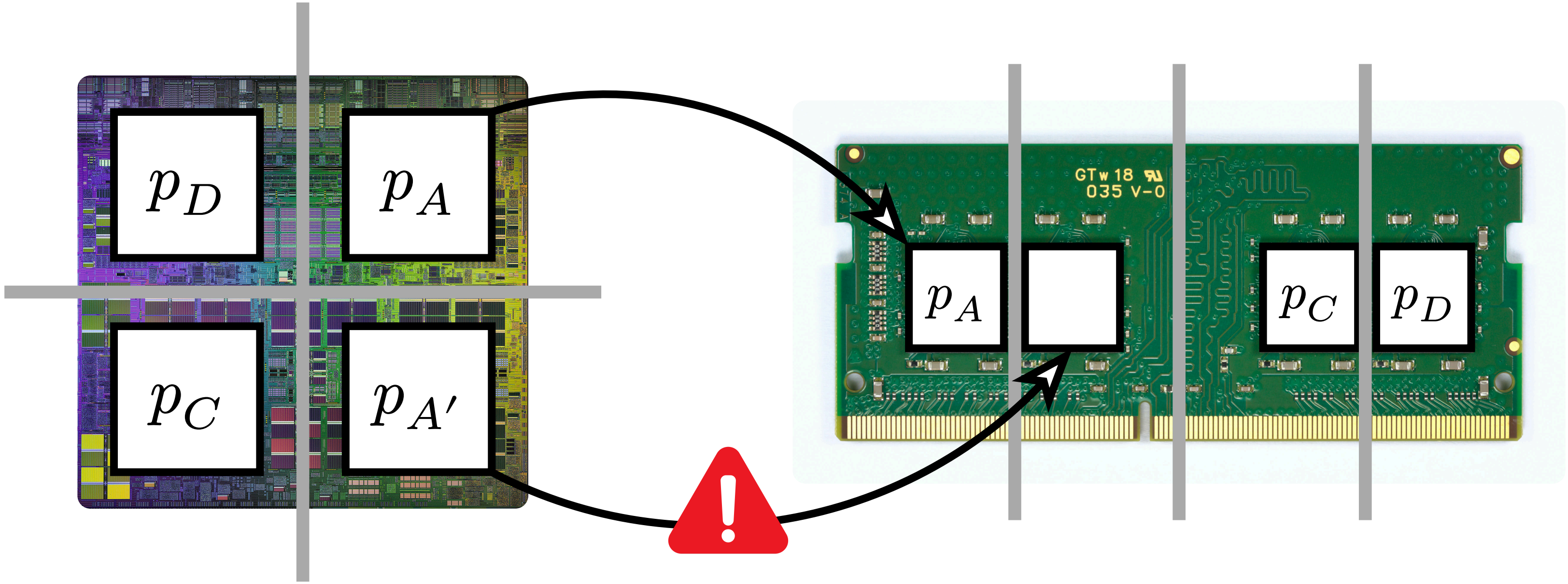


Logical Partitions, Illustrated

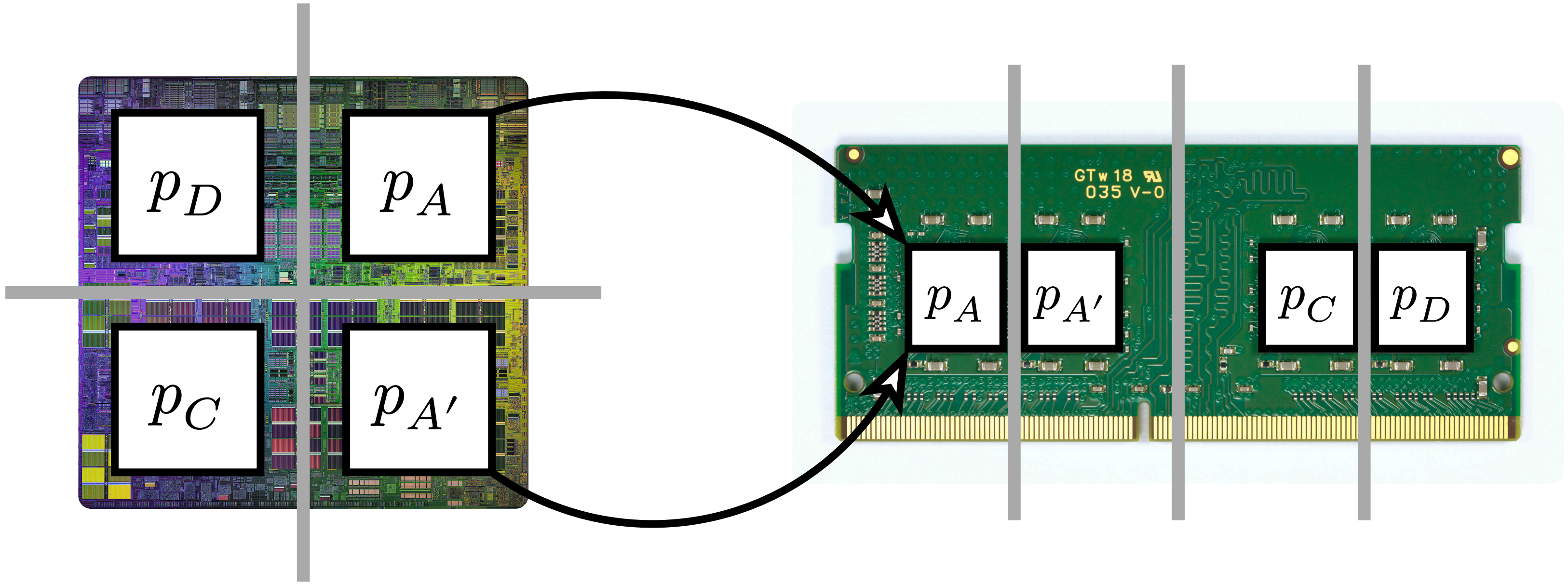


Can this model support `fork`?

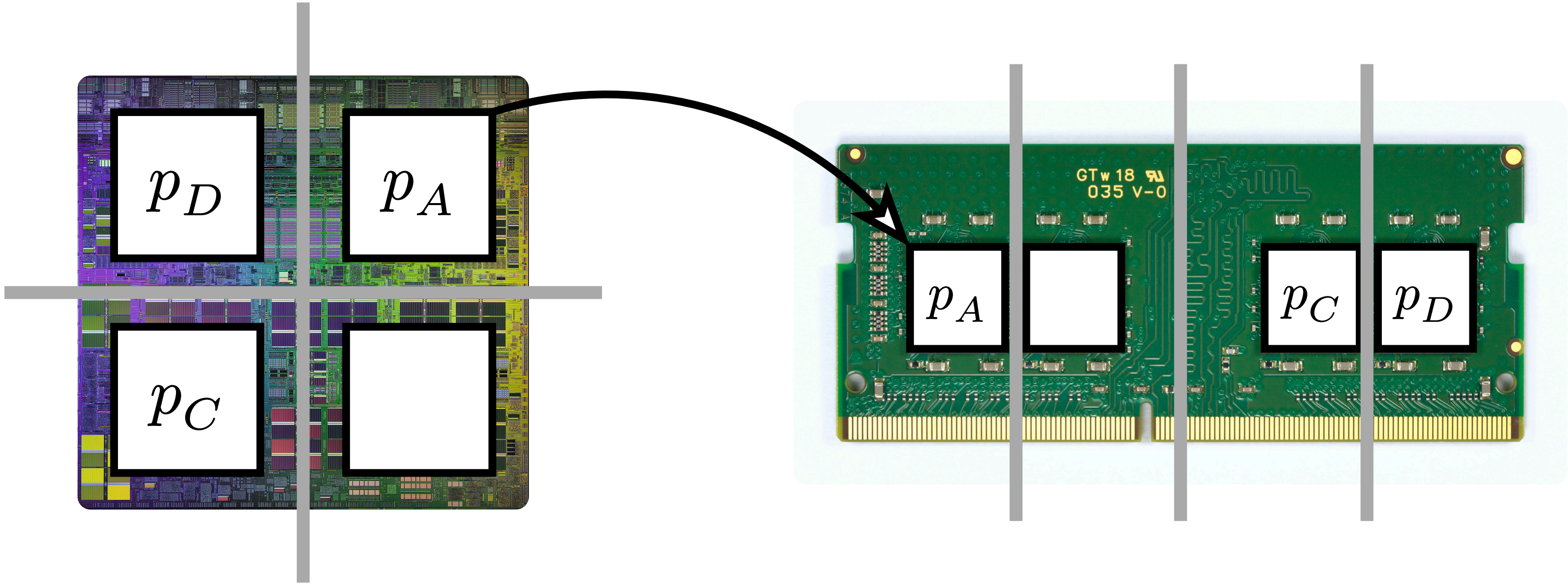
Logical Partitions, Illustrated



Logical Partitions, Illustrated

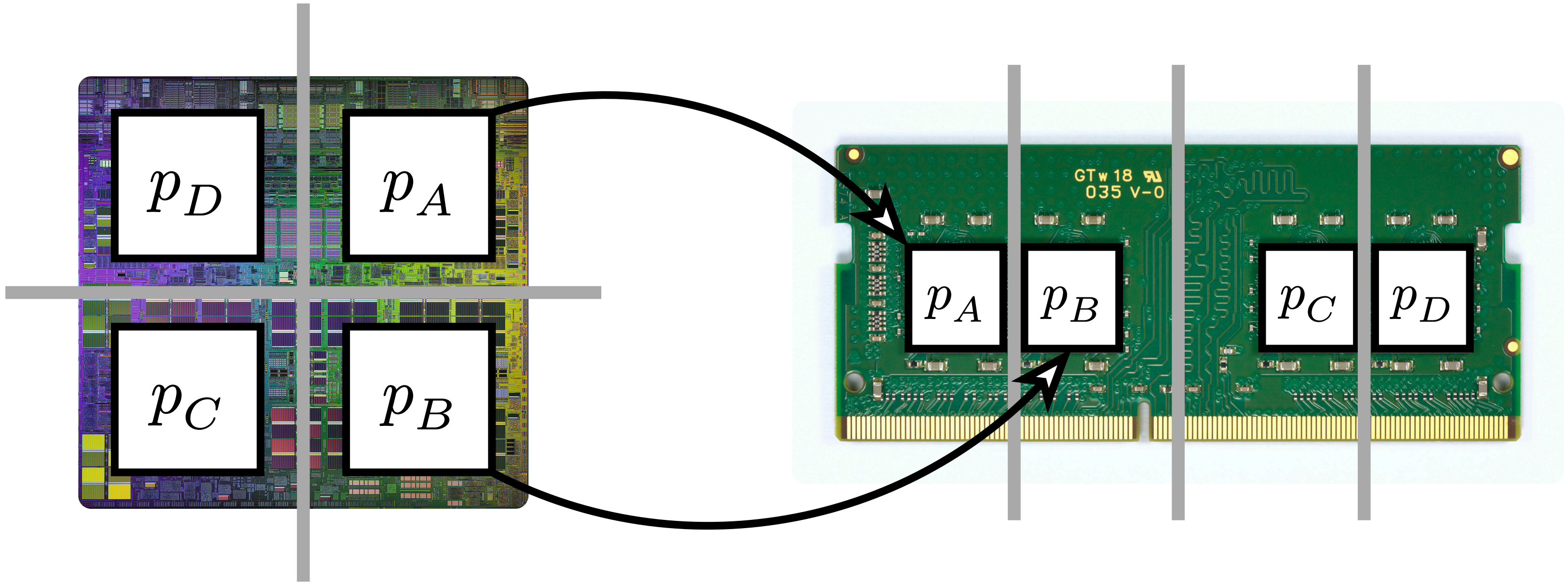


Logical Partitions, Illustrated



What about spawn?

Logical Partitions, Illustrated



Logical Partitions can be Wasteful

Logical Partitions **cannot overprovision**, **avoid side channels** such as through timing, and have **predictable performance & timing**.

Logical Partitions can be Wasteful

Logical Partitions **cannot overprovision**, **avoid side channels** such as through timing, and have **predictable performance & timing**.

For the majority of applications, they are **wasteful**: applications rarely keep a CPU busy for long periods of time. Your computer runs *thousands* of processes.

How can we retain these benefits, without wasting so many resources?

Static Processes for Predictable Behavior

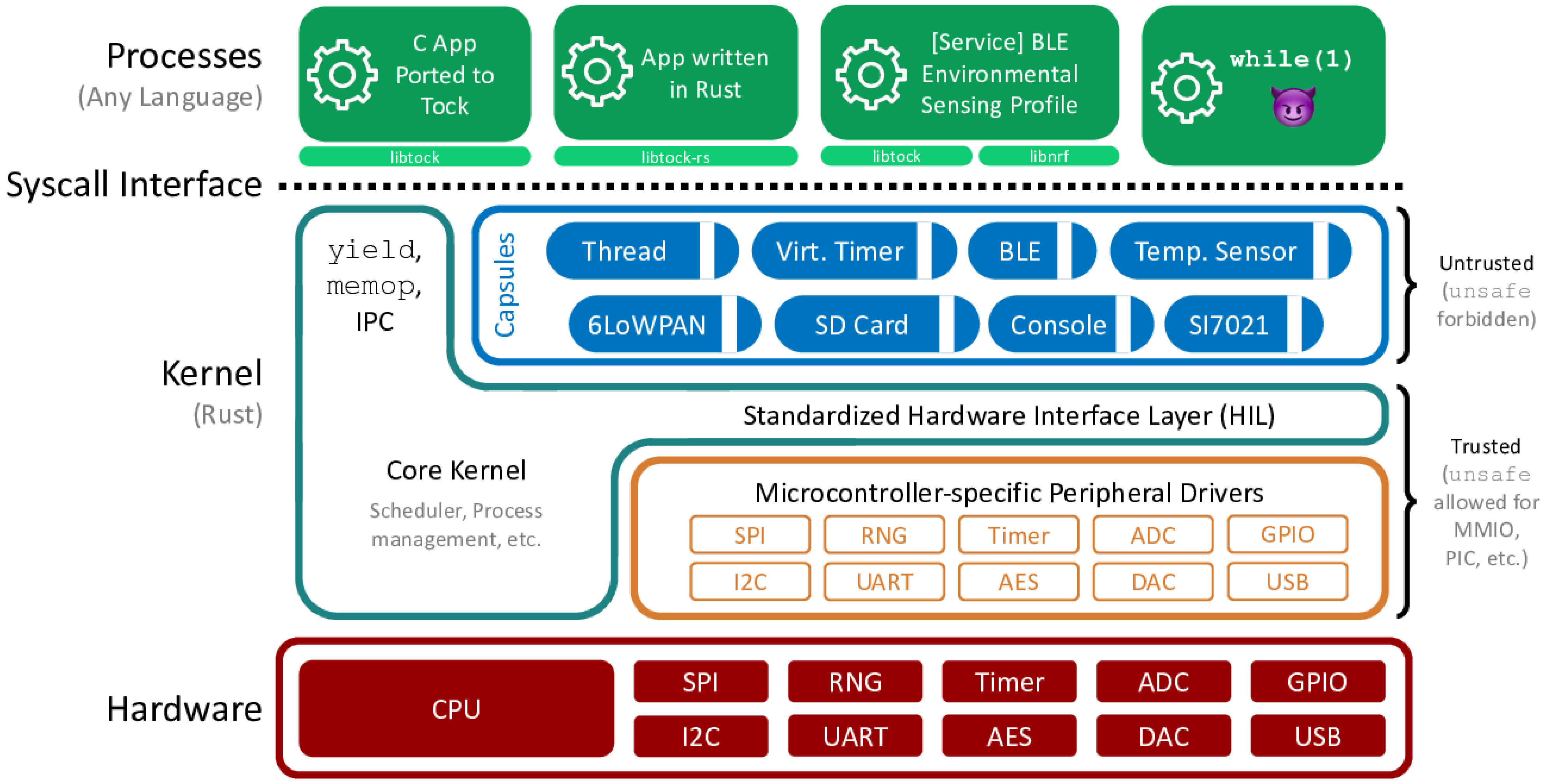
→ Virtualize the CPU according to a **fixed** schedule, among a **static** set of processes!

Formally verified *microkernel* **OS**

Used in safety-critical, **real-time** domains, like automotive ECUs. Formal correctness proof.

Memory safe embedded OS

Used in security *root of trusts* (RoTs). You might be running Tock in your laptop today!

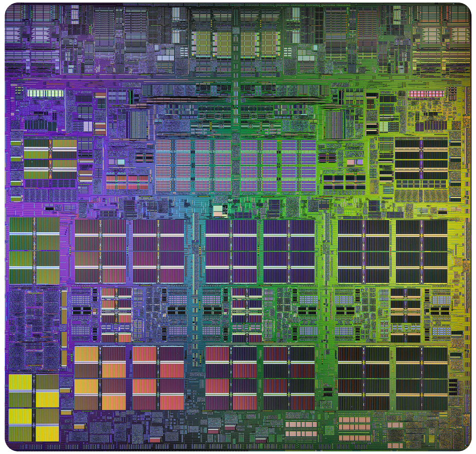


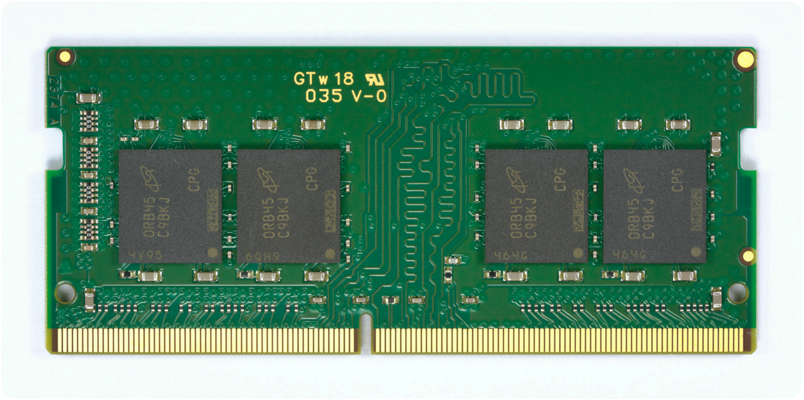
Processes
(Any Language)
C App Ported to Tock
libtock
App written in Rust
libtock-rs
[Service] BLE Environmental Sensing Profile
libtock
libnrf
while(1)
Syscall Interface
Kernel
(Rust)
yield, memop, IPC
Capsules
Thread
Virt. Timer
BLE
Temp. Sensor
6LoWPAN
SD Card
Console
SI7021
Untrusted
(unsafe forbidden)
Standardized Hardware Interface Layer (HIL)
Core Kernel
Scheduler, Process management, etc.
Microcontroller-specific Peripheral Drivers
SPI
RNG
Timer
ADC
GPIO
I2C
UART
AES
DAC
USB
Trusted
(unsafe allowed for MMIO, PIC, etc.)
Hardware
CPU
SPI
RNG
Timer
ADC
GPIO
I2C
UART
AES
DAC
USB

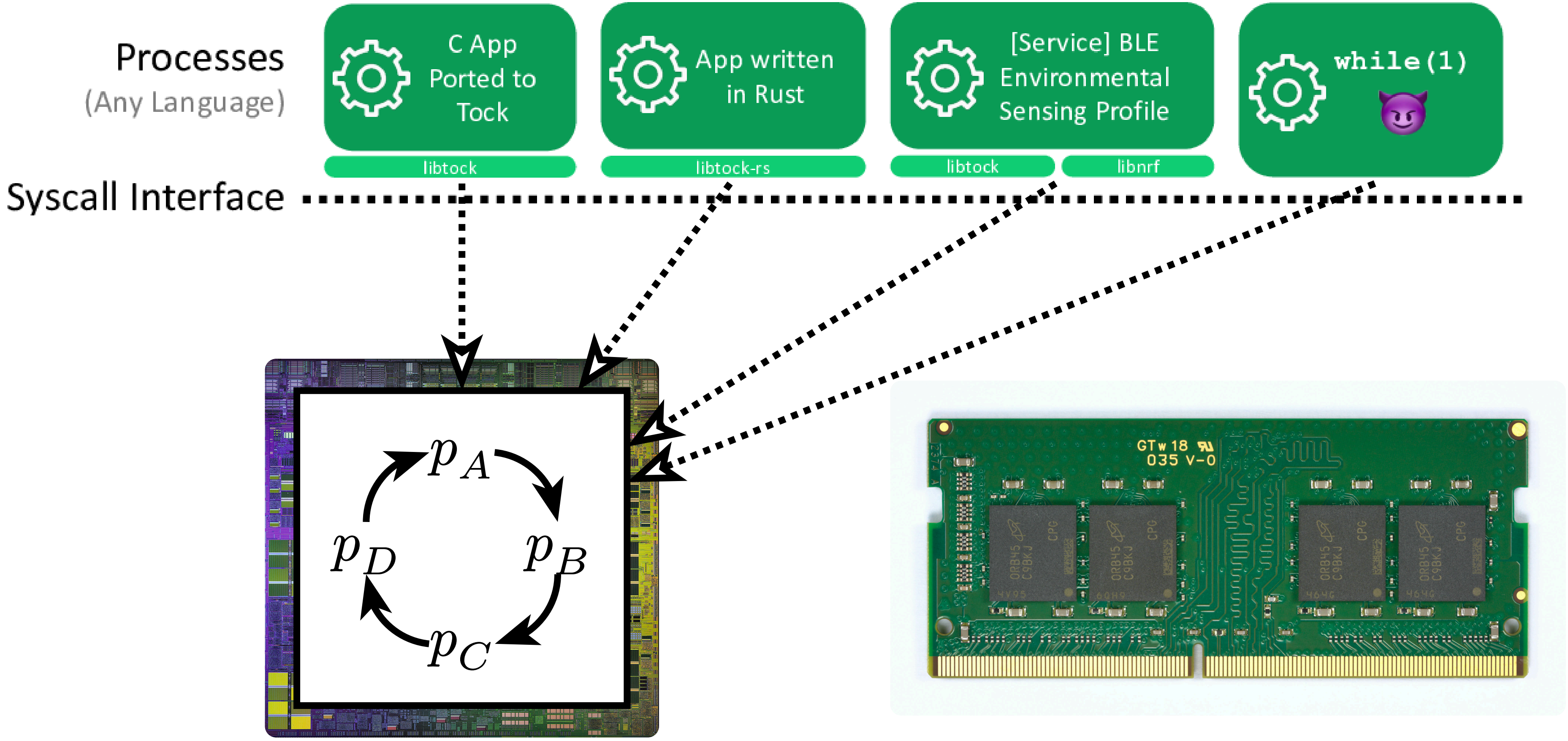
Processes
(Any Language)

C App Ported to Tock	App written in Rust	[Service] BLE Environmental Sensing Profile	`while(1)` 😈
libtock	libtock-rs	libtock / libnrf	

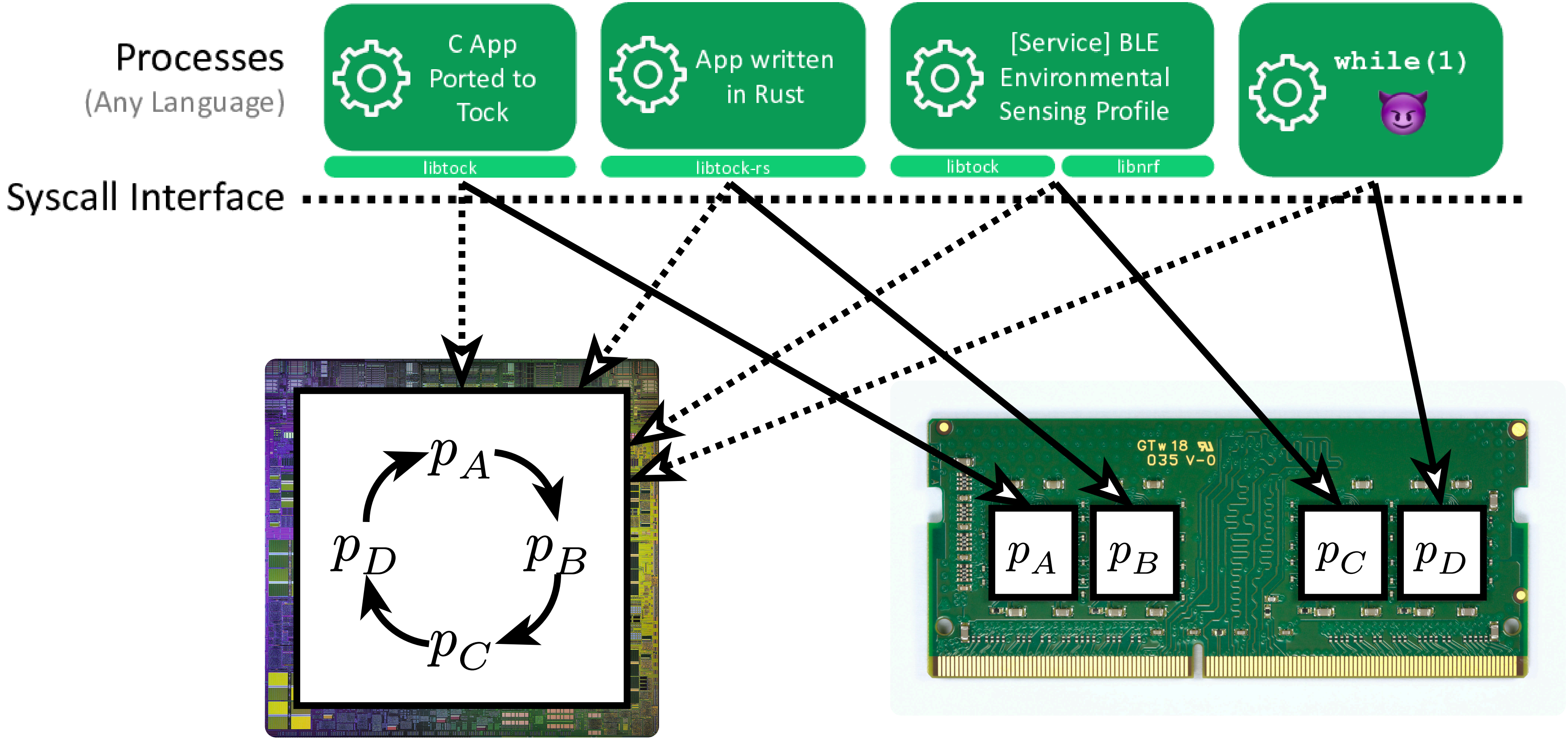
Syscall Interface







Processes
(Any Language)
C App Ported to Tock
App written in Rust
[Service] BLE Environmental Sensing Profile
while(1)
libtock
libtock-rs
libtock
libnrf
Syscall Interface
p_A
p_B
p_C
p_D



Can this model support `fork`, or `spawn`?

Preemptive vs. Cooperative Scheduling

Possible timing constraints?

```
def temperature_alert(trip):
  while True:
    cur = sensor.readC()
    if cur > trip_point:
      send_alert(cur)
```

Preemptive vs. Cooperative Scheduling


```
def temperature_alert(trip):
  while True:
    cur = sensor.readC()
    if cur > trip_point:
      send_alert(cur)
```

Possible timing constraints:

- Sample sensor n times per sec
- Max delay between sampling the sensor and sending alert
- No interruption when sending alert (e.g., sending a series of wireless packets)

Can `temperature_alert()` meet these constraints?

Preemptive vs. Cooperative Scheduling


```
def temperature_alert(trip):
  while True:
    cur = sensor.readC()
    if cur > trip_point:
      send_alert(cur)
    yield_control()
```

Possible timing constraints:

- Sample sensor `n` times per sec
- Max delay between sampling the sensor and sending alert
- No interruption when sending alert (e.g., sending a series of wireless packets)

Can `temperature_alert()` meet these constraints?

Preemptive vs. Cooperative Scheduling

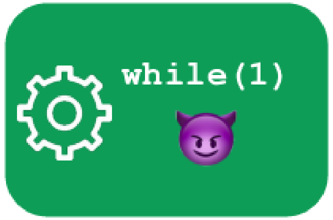

```
@sched(priority = HIGHEST)
def temperature_alert(trip):
  while True:
    cur = sensor.readC()
    if cur > trip_point:
      send_alert(cur)
    yield_control()
```

Possible timing constraints:

- Sample sensor `n` times per sec
- Max delay between sampling the sensor and sending alert
- No interruption when sending alert (e.g., sending a series of wireless packets)

Can `temperature_alert()` meet these constraints?

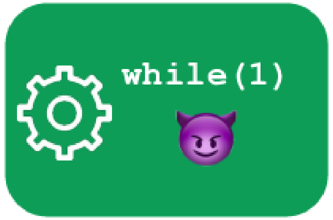
Preemptive vs. Cooperative Scheduling




```
@sched(priority = HIGHEST)
def temperature_alert(trip):
  while True:
    cur = sensor.readC()
    if cur > trip_point:
      send_alert(cur)
    yield_control()
```

```
def evil_proc():
  while True:
    pass
```

Preemptive vs. Cooperative Scheduling




```
@sched(priority = HIGHEST)
def temperature_alert(trip):
  while True:
    cur = sensor.readC()
    if cur > trip_point:
      send_alert(cur)
    yield_control()
```

```
def evil_proc():
  while True:
    pass
```

In a cooperatively scheduled system, a single stuck process can bring it to a halt!

Preemptive vs. Cooperative Scheduling

Tradeoff between

timing guarantees for each process

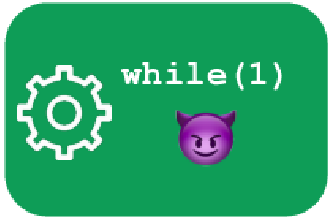
and

whole-system liveness.

Preemptive vs. Cooperative Scheduling


```
@sched(priority = HIGHEST)
def temperature_alert(trip):
  while True:
    cur = sensor.readC()
    if cur > trip_point:
      send_alert(cur)
    yield_control()
```

Cooperatively scheduled




```
def evil_proc():
  while True:
    pass
```

Preemptively scheduled

Virtualization

Transform an underlying resource into an (often) more general, powerful and easy to use **virtual** form of itself.

An important tool for **portability**!

Example: Virtual Memory

Creates a **virtual** address space that does not correspond to any single whole or partial physical resource.

Its interface does *not* introduce higher-level abstractions, like a `fork` system call or files in a file system.

Virtualization is Ubiquitous

Process Virtual Machines

Virtualization is Ubiquitous

Process Virtual Machines

Web Browsers run JavaScript and WebAssembly

Reexpose host CPU (`x86`, `AMD64`, `aarch64`) as a **virtual machine** that can interpret and execute code provided by a website.

Virtualization is Ubiquitous

Process Virtual Machines

Web Browsers run JavaScript and WebAssembly

Reexpose host CPU (`x86`, `AMD64`, `aarch64`) as a **virtual machine** that can interpret and execute code provided by a website.

Apple Rosetta 2

Translation layer for running AMD64 legacy macOS applications on new M-series Apple SoCs implementing the ARM `aarch64` architecture.

Virtualization is Ubiquitous

System Virtual Machines

Virtualization is Ubiquitous

System Virtual Machines

QEMU – Quick Emulator

Runs as a process, and provides **virtualized** (emulated) versions of **all resources** required to **run a full operating system**.

You'll be using QEMU in this class!

Can provide a virtual *guest* system similar to your *host* computer (fast), or emulate an entirely different system architecture (slower).

UNIX Processes

Covered in depth in prior lectures.

Provide all three of multiplexing, virtualization and abstraction.

Cloud Computing

It's just other people's computers!

It's a little more complex! A "cloud" is a lot like an operating system.

It Multiplexes: Running many tenants on shared infrastructure.

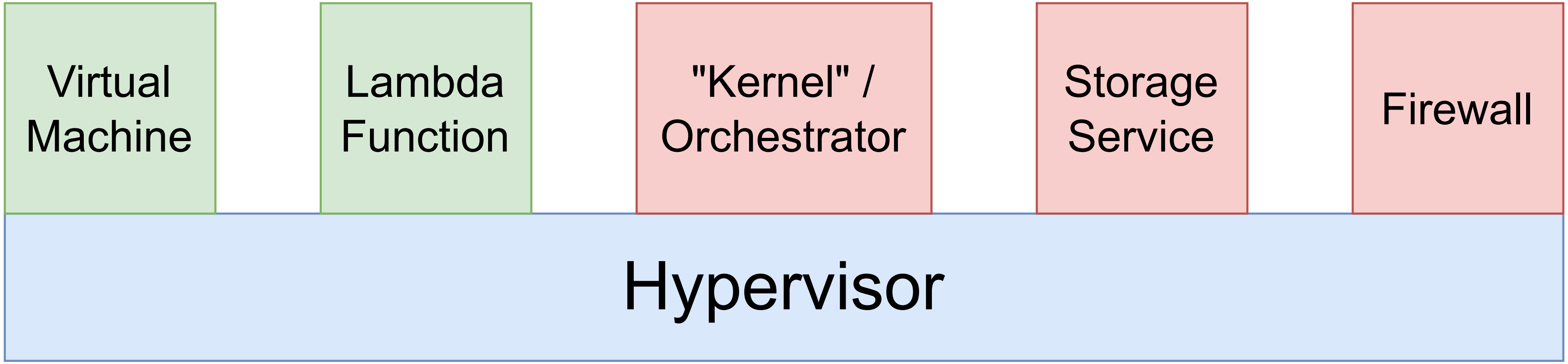
It Virtualizes: Providing virtual machines and resources to support a diverse set of workloads and enable "migrations".

It Abstracts: High-level concepts like "lambda functions" and "object storage" on top of physical resources like CPUs and disks.

Not just a single *process*: variety of offerings, different properties.

Departure from Kernel–Userspace Model

Kernel no longer behaves like a “library” to processes, like in LDE. It does not run *underneath* a process, it runs **next to it**.



Meta-Layering: Each *tenant* may itself run a full operating system.