

Operating-System Structures

بُنَى أنظمة التشغيل

Services, interfaces, system calls, and the internal architecture of a modern kernel

PRESENTED BY

Prof. Mostafa Elhosseini

Professor of Smart Systems Engineering

Six fundamental questions

01 WHAT services does the OS provide?

Operating-System Services

02 HOW do users interact with the OS?

GUI, CLI, Shell

03 HOW do applications request OS services?

APIs and System Calls

04 HOW are programs prepared and executed?

Compilers, Linkers, Loaders

05 HOW is the OS organized internally?

Monolithic, Layered, Microkernel, Modular, Hybrid

06 HOW does the OS start, and how can we examine it?

Booting, Debugging, Monitoring

THE CENTRAL QUESTION

How should we *design, organize, build, boot, and monitor* an Operating System?

Fast

Reliable

Secure

Flexible

Easy to maintain

We know how apps **request** kernel services — now: how is the **kernel itself** designed?

User goals vs system goals

USER GOALS	SYSTEM GOALS
Easy to learn and use	Easy to design and implement
Convenient	Easy to maintain
Fast and responsive	Efficient
Reliable and safe	Flexible and reliable

Customers want
Delicious food, fast service, convenience, safety

The manager wants
Efficient operations, easy maintenance, flexibility, low costs

Both matter — but they are **not identical**.

How it is done vs what should be done

Mechanism = *HOW*

A **timer interrupt** lets the OS regain control of the CPU

The oven and its timer

Policy = *WHAT*

How long may a process run? **Which** process runs next?

The chef's choice: 15 minutes

Why separate them? Change the scheduling **policy** — keep the same timer and context-switch mechanisms.

PART 2

Five Kernel Architectures

خمس معماريات للنواة

Five ways to organize the restaurant

Five ways to organize a restaurant

01

Monolithic

One big shared kitchen

مطبخ واحد كبير ومشارك،
كل العاملين موجودين مع
بعض وبيتواصلوا مباشرة.

02

Layered

Organized departments

أقسام منظمة في مستويات، وكل
قسم يعتمد على الخدمات اللي
بتقدمها المستويات اللي تحته.

03

Microkernel

Separate kitchens with a dispatcher

مطابخ منفصلة وبينها منسق
للطلبات، كل مطبخ مسؤول عن
خدمة معينة والتواصل بيتم من
خلال رسائل.

04

Modular

Interchangeable kitchen equipment

معدات مطبخ قابلة للإضافة
والاستبدال، نقدر نضيف أو
نشيل معدات حسب الحاجة من
غير ما نعيد بناء المطبخ كله.

05

Hybrid

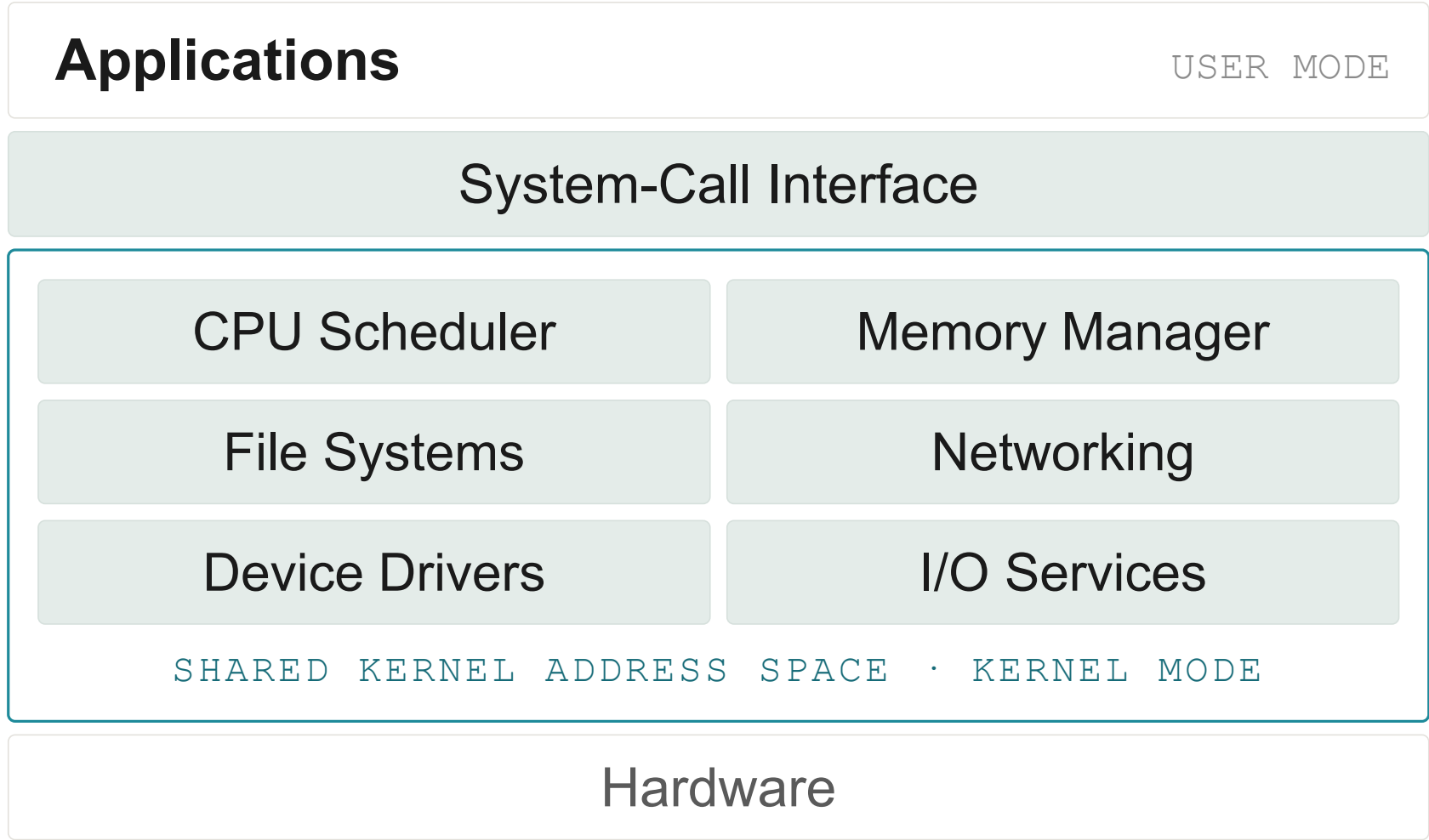
A central kitchen with specialized departments

مطبخ مركزي مع أقسام
متخصصة، بنجمع بين أكثر من
طريقة تنظيم للاستفادة من
مميزات كل واحدة.

Each organization offers different **advantages** — and different **trade-offs**.

One big kitchen

- مطبخ واحد كبير ومشارك، كل العاملين موجودين مع بعض ويتواصلوا مباشرة.



Major OS services run together in a **shared privileged kernel address space**

READING A FILE

- The application makes a **system call**
- The file system cooperates **directly** with memory management and drivers
- No separate user-space processes in between

Fast — but one mistake affects all

RESTAURANT ANALOGY

A **large open kitchen** : the chef cooks, another worker manages ingredients, someone runs the equipment — all talking **directly**.

STRENGTHS

- + Communication is **fast**
- + No message passing between departments

LIMITATIONS

- One faulty worker can **disrupt the whole kitchen**
- Faulty privileged code can **crash the OS**

Linux Monolithic — *where* major kernel components run
Modular — *how* extra functionality is loaded and organized

Hierarchical levels with defined interfaces

- أقسام منظمة في مستويات، وكل قسم يعتمد على الخدمات التي تقدمها المستويات التي تحته.

Layer N	User interface
Layer 3	File services
Layer 2	Memory management
Layer 1	Hardware management
Layer 0	Hardware — CPU, RAM, I/O

Illustrative — the division varies by design

- Each layer uses only the services of the layers **below** it
- Errors are traced to one layer, if lower layers are known to work

STRENGTHS

- + Clear structure
- + Easier debugging and maintenance

COSTS

- Traversal overhead
- Hard to assign services to layers

RESTAURANT ANALOGY

A chain of departments : waiter → coordinator → supervisor → chef.
Orderly — but a simple request passes through every desk.

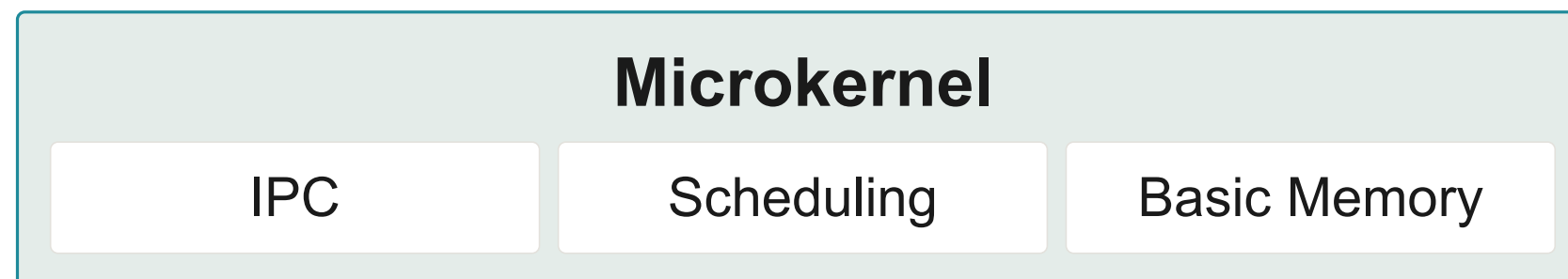
Keep only the essentials in kernel mode

- مطابخ منفصلة وبينها منسّق للطلبات، كل مطبخ مسؤول عن خدمة معينة والتواصل ويتم من خلال رسائل.

USER MODE



↓ Messages / IPC ↓



KERNEL MODE



Only a **minimal set** of fundamental mechanisms runs in Kernel Mode

- **File servers** and some **device drivers** run as independent User-Mode processes
- Historic example: **Mach** — Carnegie Mellon University

Separate kitchens, one dispatcher

RESTAURANT ANALOGY

Separate kitchens for **pizza, desserts, and main meals** — a central dispatcher coordinates. If the **bakery** breaks, the others keep cooking.

IPC — Interprocess Communication

The mechanism separate processes use to **exchange information** — a microkernel relies on it heavily

STRENGTHS

- + Better **isolation** and **reliability**
- + A crashed file service can be **restarted**

LIMITATIONS

- Extra **communication** overhead
- Extra **context switching**

Interchangeable kitchen equipment

- معدات مطبخ قابلة للإضافة والاستبدال، نقدر نضيف أو نزيل معدات حسب الحاجة من غير ما نعيد بناء المطبخ كله.

Add or remove kernel functionality **without rebuilding** the OS — Linux **Loadable Kernel Modules (LKMs)**

RESTAURANT ANALOGY

Start selling pizza? **Install a pizza oven** — no need to rebuild the kitchen. Stop? Remove it.

`lsmod`
list loaded

`modinfo`
module info

`modprobe`
load + deps

CONNECTING A USB DEVICE

- 1 User connects a USB device
- 2 Linux **USB subsystem** detects it
- 3 Linux identifies an appropriate **driver**
- 4 If it is a module not yet loaded — **it is loaded**
- 5 The driver **manages** the device

Modular $\neq$ microkernel

MODULAR LINUX KERNEL	MICROKERNEL
Modules commonly run in Kernel Mode	Many services run in User Mode
Share the kernel address space	Often separate address spaces
Direct kernel function calls	Communication through IPC
A faulty module may crash the kernel	Failures can often be isolated

Modules loaded into Linux normally remain **privileged kernel components**.

macOS and iOS: Darwin’s XNU kernel

Combines ideas from different designs to balance **performance, flexibility, and reliability**

XNU COMPONENT	MAIN RESPONSIBILITY
Mach	Tasks, threads, scheduling, virtual memory, IPC
BSD	Processes, networking, file systems, POSIX
I/O Kit	Device-driver infrastructure

RESTAURANT ANALOGY

Two departments in **one central facility** — one runs staff and resources, the other runs the food — sharing space to cut overhead.

XNU uses **Mach concepts** — but is **not** a classic microkernel of isolated user-space servers.

One app — many different smartphones

- تطبيق WhatsApp يشتغل على Samsung وHONOR وXiaomi، رغم إن كل جهاز ممكن يكون فيه Hardware مختلف.
- إزاي Android بيحل المشكلة دي؟ عن طريق تنظيم النظام في مجموعة طبقات (Layers)، وكل طبقة مسؤولة عن تقديم خدمات معينة للطبقات اللي فوقها.

Android Applications	WhatsApp, Camera, Maps
Frameworks + ART	APIs and runtime execution
Native Libraries	Bionic, SQLite, media
HAL	Standard device interfaces
Linux Kernel	Processes, memory, drivers
Hardware	CPU, camera, sensors

Different **cameras, processors, sensors,** and **manufacturers** — one platform

A **modular, monolithic Linux kernel** under a **layered** user-space platform — not a microkernel.

ART, JNI, HAL, Bionic

■ إزاي تطبيق Android بيتنفذ، ويستخدم مكتبات برمجية، ويتعامل مع Hardware مختلف من جهاز للتاني

ART

Android Runtime — runs app code with **AOT** and **JIT** compilation

AOT: Android Runtime
JIT: Just-in-Time

JNI

Java Native Interface — Java code calls native **C/C++** libraries

الشيف الأساسي محتاج مساعدة من شيف متخصص عنده مهارة أو أدوات خاصة. فبيستخدم وسيلة تواصل معاه علشان ينفذ جزء من الوصفة. الـ JNI هي وسيلة التواصل دي.

HAL

Hardware Abstraction Layer — standard interfaces to device hardware

توفير واجهة موحدة للخدمات المرتبطة بالأجهزة، مع ترك تفاصيل التنفيذ للمكونات الخاصة بكل Hardware.

Bionic

Android's **C** library implementation

Bionic هي مكتبة C الخاصة بنظام Android.

RESTAURANT ANALOGY

Two branches, **different oven brands** — yet chefs use the **same familiar controls** for temperature and time. The controls hide the differences. That is the **HAL**.

Translate the order — or build a second kitchen

- تقنية من Microsoft بتسمح لنا بتشغل برامج وأوامر Linux داخل نظام Windows، من غير ما نحتاج نثبت Linux كنظام تشغيل منفصل على الجهاز.

WSL 1	WSL 2
Linux system-call compatibility layer	Real Linux kernel
No separate Linux kernel	Kernel in a lightweight VM
Translation and emulation	Linux calls handled by Linux
More compatibility limits	Broader compatibility

RESTAURANT ANALOGY

WSL 1 — hire a **translator** for the foreign customer’s order.

RESTAURANT ANALOGY

WSL 2 — open a **separate kitchen** staffed by people who speak the language.

Same problem — **very different** architectural mechanisms.

Five architectures, five trade-offs

ARCHITECTURE	RESTAURANT	ADVANTAGE	LIMITATION
Monolithic	<i>One big open kitchen</i>	+ Performance	– Fault isolation
Layered	<i>Departments in a hierarchy</i>	+ Organization	– Layer overhead
Microkernel	<i>Separate kitchens + dispatcher</i>	+ Reliability, isolation	– IPC overhead
Modular	<i>Interchangeable equipment</i>	+ Flexibility	– Shared privileged risk
Hybrid	<i>Central kitchen + specialists</i>	+ Balanced design	– Complexity

Which architecture would you choose?

SCENARIO
1

A failed user-space service should be **restartable** without crashing the whole OS.

Monolithic

Microkernel ✓

Strictly layered

Services in separate processes improve fault isolation and recovery.

SCENARIO
2

You are extending Linux with a **device driver** without rebuilding the kernel.

Loadable Kernel Modules ✓

Microkernel IPC

Changing the CPU

Linux loads compatible kernel modules at run time.

SCENARIO
3

You want **clearly defined layers** to simplify reasoning and debugging.

Hybrid

Monolithic

Layered ✓

Services are grouped by level with defined dependencies.

Three questions

Q1 Why is a monolithic kernel often faster than a microkernel?

Components communicate through **direct calls** — no IPC or extra context switches

Q2 Why can a microkernel isolate faults better?

Many services run in **separate user-space processes**

Q3 How can Linux be monolithic and modular at once?

Modules run **in kernel space** and are **loaded dynamically**

Architecture decides **where** components run, **how** they communicate, how **failures** are isolated, and how easily the system is **extended**.

2.9 – 2.10

Building, Booting, and Debugging

البناء، الإقلاع، وتصحيح الأخطاء

Generation, boot sequence, failure analysis, and performance tuning

Building a system from scratch

F I V E S T E P S

- 1 Write or obtain the source code
- 2 Configure the system for the machine it will run on
- 3 Compile the operating system
- 4 Install it
- 5 Boot the computer and its new operating system

A configuration file drives a **full system build** , a selection of precompiled modules, or purely run-time parameters

A L I N U X K E R N E L B U I L D

```
$make menuconfig  
$make  
$make modules  
$make modules install  
$make install
```

Configuration lands in `.config` ; the build produces the kernel image `vmlinuz`

From power ON to login

01

Power ON

CPU executes firmware instructions

02

BIOS / UEFI

Initializes hardware, selects boot path

03

Bootloader

Loads the OS kernel

04

Kernel init

Memory, drivers, scheduling

05

Root file system

Made available to the OS

06

User-space init

systemd starts system services

07

Login / Desktop

Ready for interaction

Mounting the root file system is **not** the finish line — initialization continues as services start.

Opening the restaurant every morning

RESTAURANT ANALOGY

Electricity on → staff arrive → equipment starts → ingredients ready → **doors open.**

- **BIOS/UEFI** — basic startup checks
- **Bootloader** — locates and starts the OS
- **Kernel** — initializes resources
- **Init system** — starts services
- **Login** — ready for users

TERM	SIMPLE EXPLANATION
BIOS	Traditional PC firmware
UEFI	Modern firmware with boot management
GRUB	Common Linux bootloader
initramfs	Temporary early file system at startup
systemd	Linux service and init manager

Android boot arrangements vary by device

PART 6

Debugging and Performance

تصحيح الأخطاء والأداء

When something goes wrong — how do we find out what happened?

Process failure vs kernel failure

PROCESS FAILURE	KERNEL FAILURE
Usually affects one process	Can affect the entire OS
May produce a Core Dump	May produce a Kernel Crash Dump
Investigated with user-space debuggers	Needs kernel-specific techniques
Other programs keep running	System may stop or restart

PROCESS ANALOGY

One chef **drops a meal** — the restaurant keeps serving.

KERNEL ANALOGY

The **central electrical system** fails — everything stops.

Counters: measuring the system

COMMAND	WHAT IT TELLS US
<code>ps</code>	Process information
<code>top</code>	Live process and resource statistics
<code>vmstat</code>	Memory, CPU, virtual memory
<code>iostat</code>	Storage I/O statistics
<code>netstat</code>	Network information

```
$ top $ cat /proc/meminfo
```

RESTAURANT ANALOGY

The manager’s **dashboard** : active orders, kitchen load, ingredients left, busy staff, customers waiting.

`/proc`

A **virtual** file system — generated by the kernel when read, **not stored on disk**

What actually happened?

COUNTERS	TRACING
Counts and summaries	Observe particular events
How much activity occurred	When and how an event occurred
Number of disk operations	Timing of each disk request
Identify symptoms	Investigate causes

RESTAURANT ANALOGY

The dashboard says delivery is slow — now **trace one order**
: received when? cooked when? delayed where?

- `strace`
System calls made by a process
- `perf`
CPU usage and performance events
- `tcpdump`
Captures network packets
- `gdb`
Debugger: execution, variables, failures — not primarily a call tracer

Observing a live kernel

eBPF

Extended Berkeley Packet Filter — verified programs run in a **restricted environment inside the kernel**

BCC

BPF Compiler Collection — tools and libraries that make eBPF tracing easier

```
$ opensnoop $ opensnoop -p 1225
```

RESTAURANT ANALOGY

Temporary **sensors in the kitchen** — measure how long meals wait for the oven, **without stopping service**.

- Needs suitable **tools, permissions, kernel support**
- Verified $\neq$ automatically **risk-free** or **zero overhead**

Eight topics, eight questions

OS Services

What services does the OS provide?

User Interfaces

How do users interact with the OS?

System Calls & APIs

How do apps request kernel services?

Compiler, Linker, Loader

How are programs prepared and run?

Portability

Why can't every app run on every OS?

OS Structures

How should kernel parts be organized?

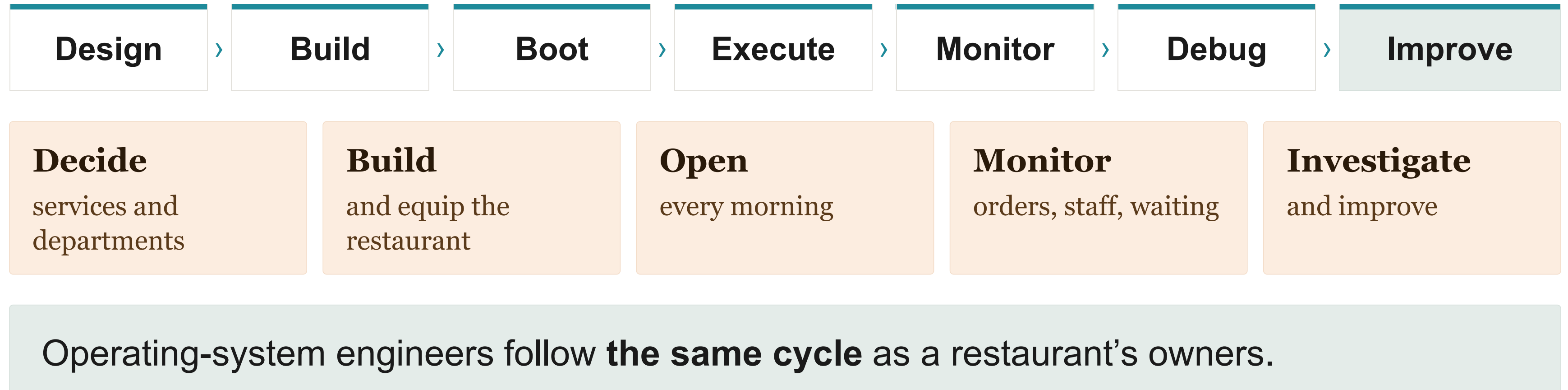
System Boot

How does the OS start?

Debugging & Monitoring

How do we investigate problems?

The life cycle of an operating system



THE MOST IMPORTANT LESSON

An OS is not just a collection of functions — its **architecture** decides how components **cooperate**, how **failures** are handled, and how the system **evolves**.

No single perfect architecture — the right design depends on the **requirements**.

What to take from Chapter 2

Services

the OS provides an environment for program execution, serving both users and the system itself

Interfaces

command interpreters, graphical interfaces, and touch screens

System calls

reached through an API in six categories; libc supplies the interface on UNIX and Linux

Around the kernel

system programs, a linker that builds executables, a loader that brings them into memory

Portability

binary formats, instruction sets, and differing system calls keep applications system-specific

Structure

monolithic for efficiency, layered for clarity, microkernel for reliability, modules for flexibility, hybrids in practice

Boot

a boot loader loads the kernel, initializes the system, and begins execution

Observability

counters for totals, tracing for events; BCC and eBPF instrument a live kernel safely