



Processes

Lecture 01 · 3.1 Process Concept

Prof. Mostafa Elhosseini

Professor of Smart Systems Engineering

What This Lecture Covers

3.1.1

The Process

How is a process different from a program?

3.1.2

Process State

What happens from creation to termination?

3.1.3

Process Control Block

How does the OS remember each process?

3.1.4

Threads

How can one process do several things?

3.1.1

The Process

A program in execution — an active entity with its own program counter, registers and memory.

Why We Need Processes

On your laptop, right now:

Watching a YouTube lecture

Downloading a PDF

Editing a Word document

Listening to music

Receiving WhatsApp notifications

QUESTION

Who manages them all, and how does the CPU know which to run?

The operating system — by organising each into a **process**.

What Is a Process?

A program that is **currently executing** .

Instructions on disk become a process once the OS gives them memory and resources.

ANALOGY

The Restaurant Analogy

COMPUTER

RESTAURANT

Program

Written recipe

Process

Cooking a meal

CPU

Chef

RAM

Kitchen workspace

Process data

Ingredients

Operating system

Kitchen manager

KEY POINT

Three chefs, one recipe: the instructions are identical, the cooking is separate.

One program can run as several processes.

Job, Task and Process

BATCH SYSTEMS

Job

Work submitted, run to completion, no interaction.

TIME-SHARING

Task

Users interact with programs while they run.

TODAY

Process

The general technical term for all of them.

“Job” survives in terms such as **job scheduling**.

Program vs. Process

	Program	Process
Nature	Passive	Active
Location	Stored on disk	Loaded in memory
CPU	Not executing	Can be scheduled
Resources	Occupies storage	Memory, CPU time
Example	Word executable	Running Word

Program Counter and Registers

PC

Program Counter

Points to the next instruction to execute.

R0...Rn

CPU Registers

Tiny, very fast storage inside the CPU.

Same movie, two viewers: one at minute 5, one at minute 40. **Same content, different progress.**

FIGURE 3.1

Layout of a Process in Memory

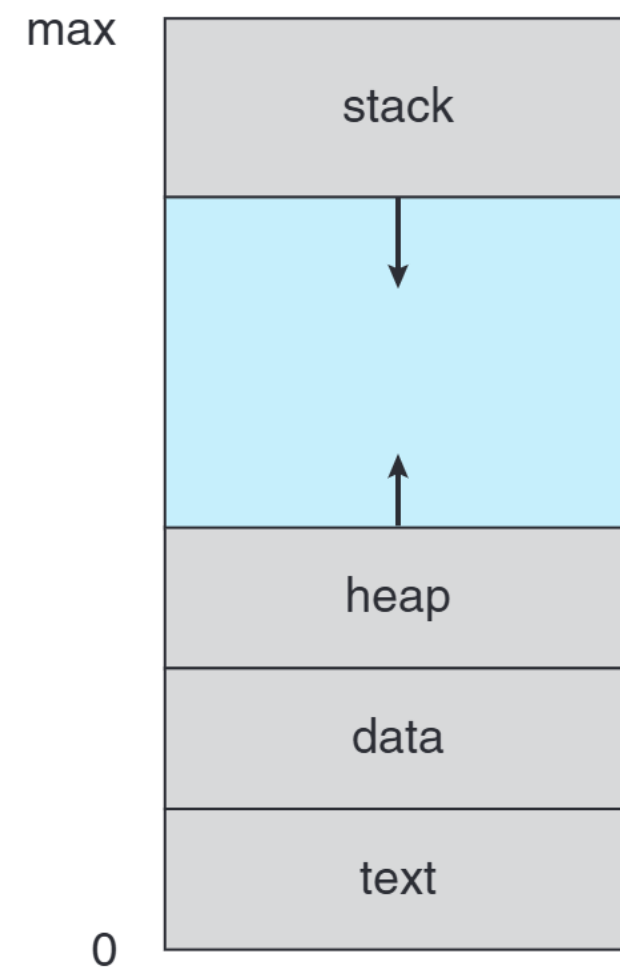


Figure 3.1 Layout of a process in memory.

stack

Function calls and local variables

heap

Memory allocated at run time

data

Global and static variables

text

Executable instructions

Text and Data Sections

FIXED SIZE

Text

The executable instructions.

```
sum = a + b;
```

Recipe: the cooking steps

FIXED SIZE

Data

Global and static variables.

```
int sc-camel-total-students = 500;
```

Restaurant: the daily-total board

Heap and Stack

GROWS ↑ AND SHRINKS

Heap

Memory requested while the program runs.

```
p = malloc(100 * sizeof(int));  
free(p);
```

GROWS ↓ AND SHRINKS

Stack

One frame per function call — Last In, First Out.

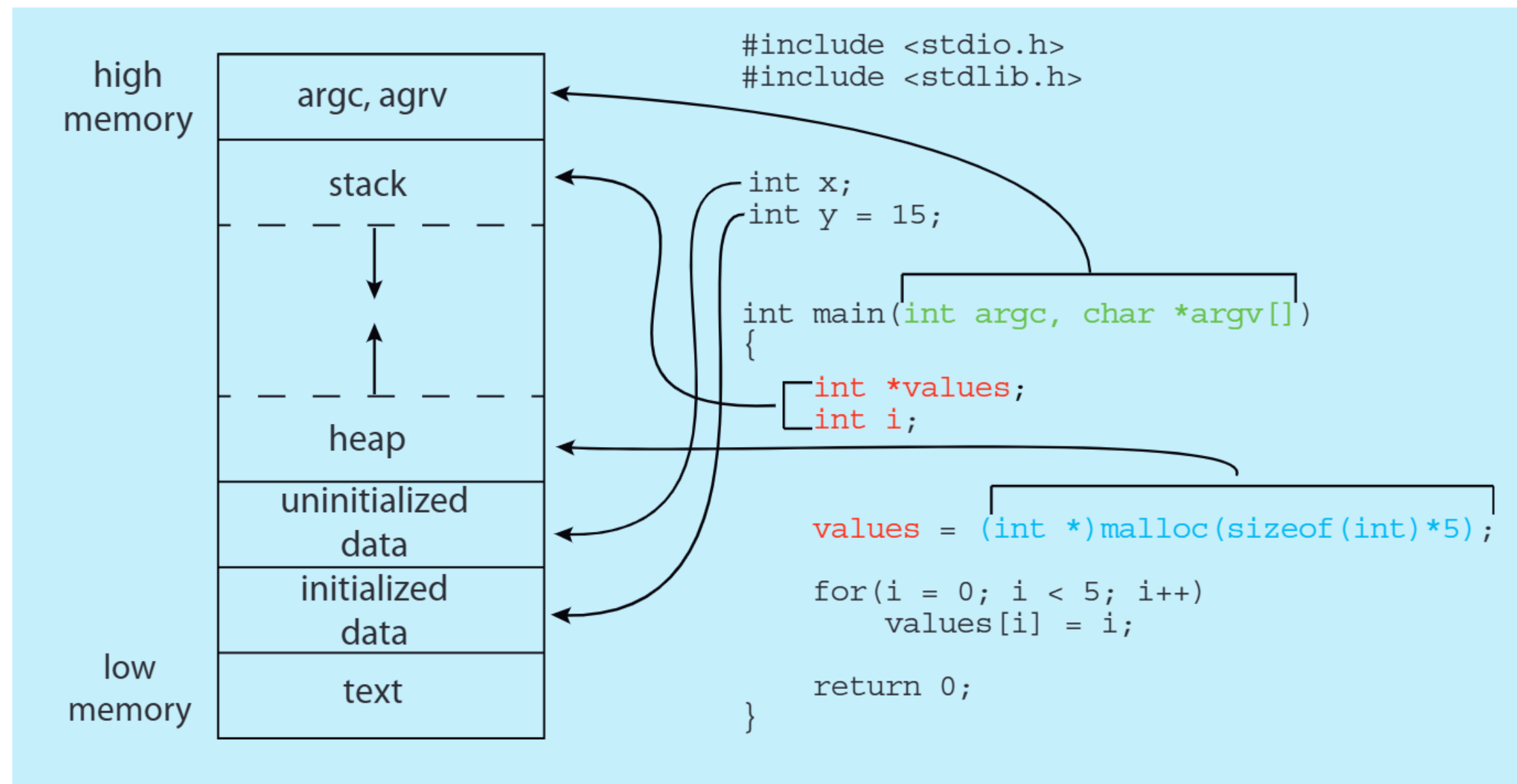
```
cookFood() ← top prepareOrder()
```

Stack vs. Heap

	Stack	Heap
Holds	Function-call information	Dynamic allocations
Managed	Automatically, per call	By allocation calls
Speed	Faster to allocate	More flexible sizes
Threads	One stack per thread	Shared by threads
Size	Usually limited	Larger objects

Memory Layout of a C Program

Global data splits into **initialized** and **uninitialized** sections; **argc**, **argv** get their own.



Measuring Sections with `size`

text	data	bss	dec	hex	filename
1158	284	8	1450	5aa	memory

text	data	bss	dec / hex
Code and read-only content	■ Initialized data Global/static variables with explicitly initialized nonzero values	Uninitialized data Block Started by Symbol	Total in decimal and hex

Same Program, Different Processes

Imagine opening the same browser application three times.

- **Process 1**: Browsing YouTube.
- **Process 2**: Accessing Blackboard.
- **Process 3**: Checking email.

Many copies of one program — each a **separate execution sequence**.

process 1

Browser A

Own data, heap, stack

process 2

Browser B

Own tabs, own cache

process 3

Browser C

Own calls and locals

SHARED

Text section — identical code

PRIVATE

Data, heap, stack — own state

Same Program, Different Processes

Hospital information system

shared text section

process 1

Doctor A

Patient A's record

process 2

Doctor B

Patient B's record

process 3

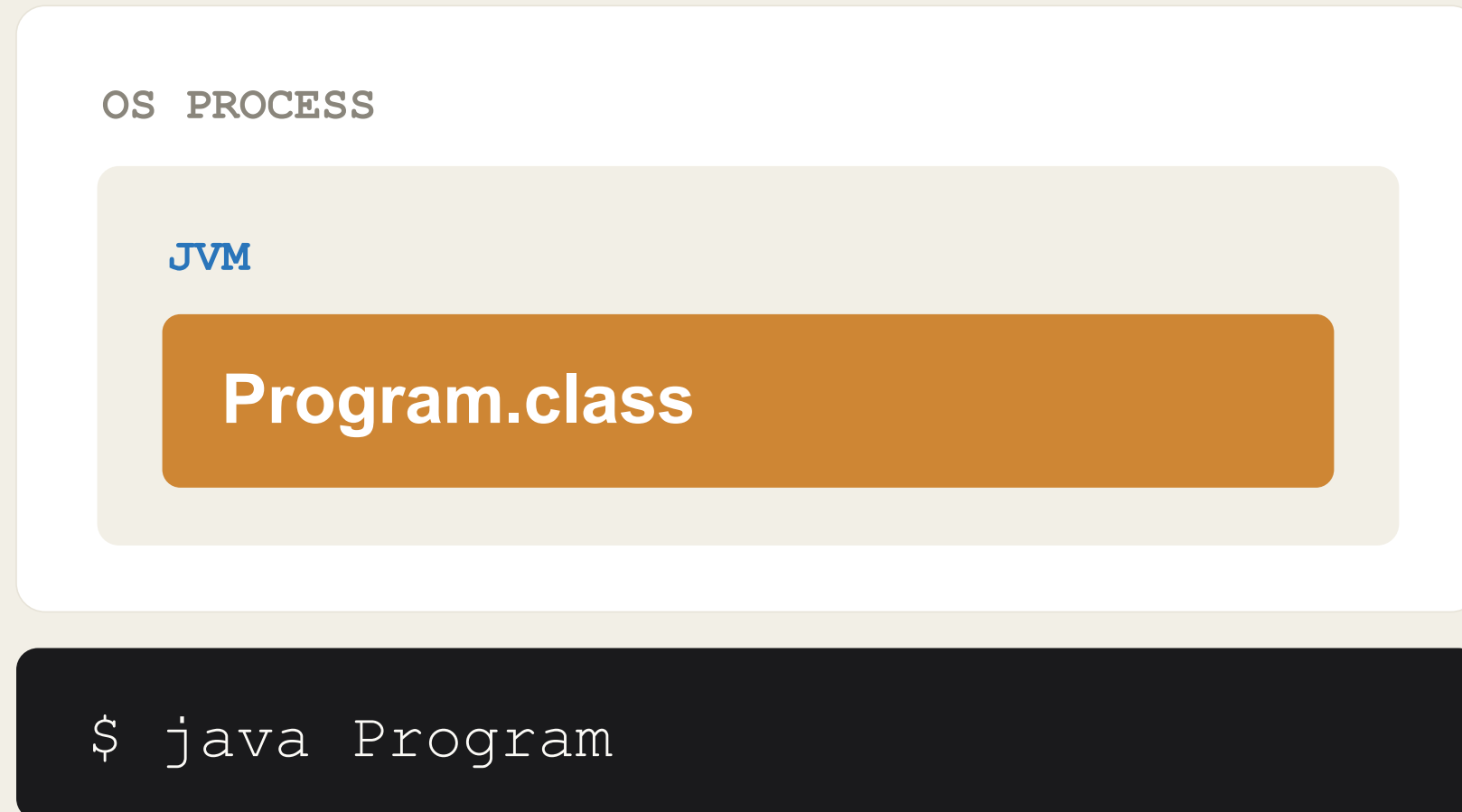
Doctor C

Patient C's record

Shared: the code. **Private:** data, heap, stack and progress.

The JVM as an Execution Environment

- A process can provide an environment in which another program executes.



One process hosting the execution of another program.

The JVM interprets or compiles bytecode into native instructions.

Parent and Child Processes

Parent

The process that creates

The manager

Child

The process it creates

The new worker

Siblings

Children of the same
parent

Co-workers

```
$ ls
```

The shell spawns a child process to run the command.

3.1.2

Process State

As a process executes, it changes state.

The Five States

New

Being created

Submits exam request

Ready

Waiting for the CPU

Waits for a free seat

Running

Instructions executing

Taking the exam

Waiting

Blocked on an event

Awaits a document

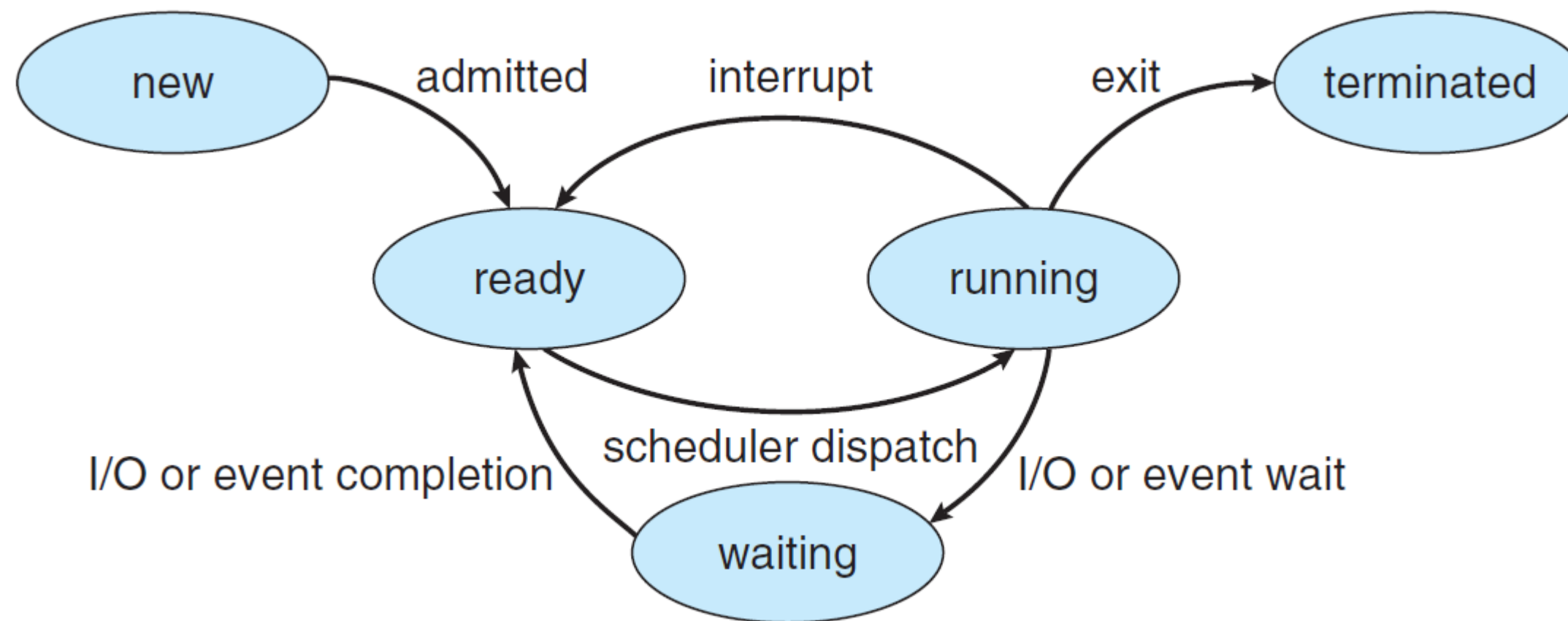
Terminated

Finished execution

Exam completed

FIGURE 3.2

Diagram of Process State



Ready vs. Waiting

Ready

Could run **now** — if given the CPU.

`needs: CPU`

Waiting

Cannot run — even with a free CPU.

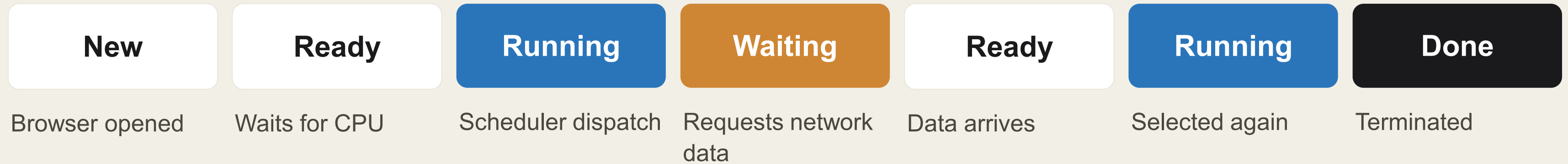
`needs: an event (e.g. I/O)`

Waiting always returns to **Ready** before **Running**.

Transitions Between States

TRANSITION	CAUSE
New → Ready	Admitted by the OS
Ready → Running	Scheduler dispatch
Running → Waiting	Needs I/O or an event
Waiting → Ready	Event completes (I/O done)
Running → Ready	Interrupt: time slice or priority
Running → Terminated	Process exits

Example: Downloading a PDF



One download, **seven** state changes.

Practice: Identify the State

- | | | |
|---|--|---------|
| 1 | Has every resource — needs only the CPU. | Ready |
| 2 | Requested disk data that hasn't arrived. | Waiting |
| 3 | Its time slice expires while executing. | Ready |
| 4 | Just launched — still being created. | New |

3.1.3

Process Control Block

Each process is represented in the OS by a PCB — also called a task control block.

FIGURE 3.3

How the OS Remembers a Process

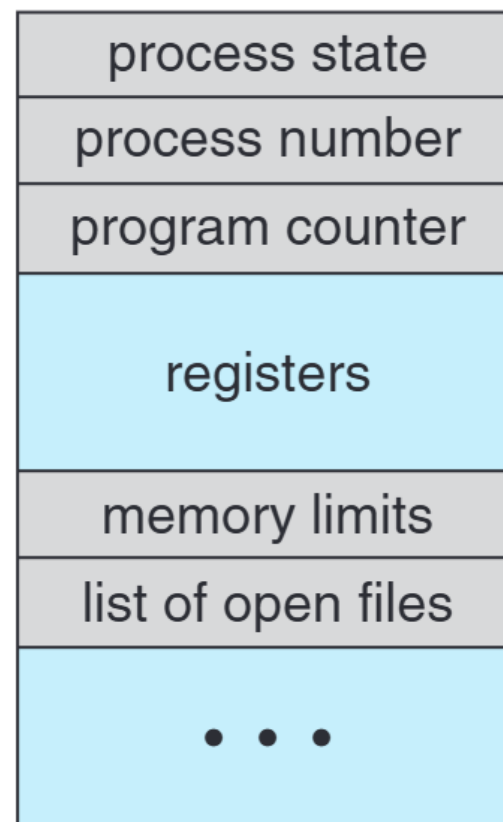


Figure 3.3 Process control block (PCB).

Every process has exactly one PCB.

It holds everything needed to **start or restart** the process.

ANALOGY

A patient's medical file — the next doctor continues from it.

What the PCB Stores

FIELD	MEANING	PATIENT FILE
Process ID	Unique identifier	Patient ID
Process state	Ready, running...	Patient status
Program counter	Next instruction	Next treatment
CPU registers	Saved CPU values	Latest results
Scheduling info	Priority, queues	Triage priority
Memory info	Mappings, limits	Assigned ward
Accounting	CPU time used	Usage records
I/O status	Open files, devices	Assigned equipment

Program Counter vs. PCB

PC

A single **CPU register**.

Points to the next instruction.

PCB

The OS's full **record** of a process.

Includes a saved copy of the PC.

The PCB is metadata — **not** the program itself.

The Context Switch



Benefit

Multitasking without lost progress

Cost

Saving and restoring is overhead

ANALOGY

A school keeps a file folder for every student: current status, class schedule, test scores, privileges, borrowed books. When a student leaves and returns, the office reads the folder to know exactly where they left off. The PCB is that folder.

Process Representation in Linux

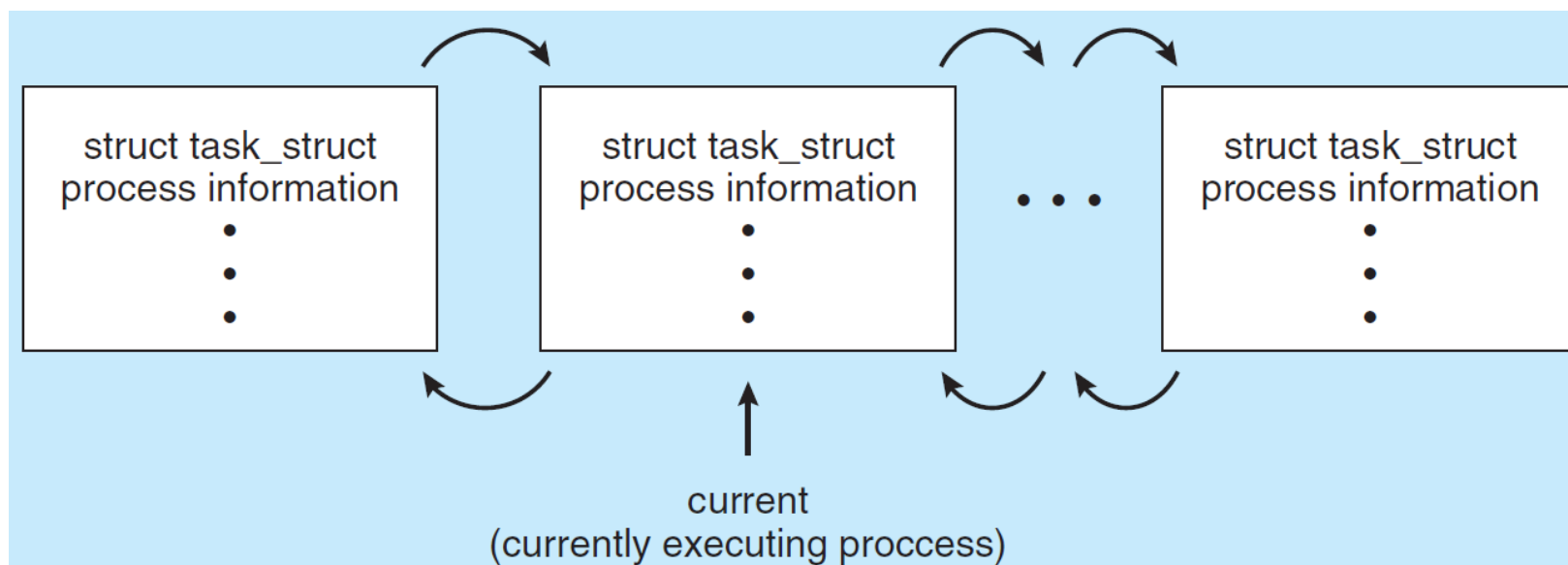
```

long state;           /* state of the process */
struct sched_entity se; /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space */

```

task_struct

Linux's PCB, in <linux/sched.h>



Doubly linked list

Holds every active process

current

Points to the running task

3.1.4

Threads

One process, several sequences of execution.

From One Thread of Execution to Many

CLASSIC MODEL

A single thread

One stream of instructions, one task at a time. The user of a word processor cannot type characters and run the spell checker at the same instant.

Especially beneficial on **multicore** systems, where multiple threads run in parallel.

MODERN EXTENSION

Multiple threads

A thread is the unit of execution inside a process: threads share code, data, memory and open files, but each keeps its own program counter, registers and stack.

On systems that support threads the **PCB is expanded** to include information for each thread — often as separate **thread control blocks (TCBs)**.

One Process, Several Activities

Word processor — one process

shared memory and files

thread 1

User input

Handles keystrokes

thread 2

Spell checker

Runs while you type

thread 3

Auto-save

Writes in background

Each thread keeps its own **PC, registers and stack**.

Process vs. Thread

	Process	Thread
Role	Resource container	Unit of execution
Memory	Own address space	Shares the process's
Creation	More expensive	Lighter
Isolation	Strong	Bugs can affect others
Analogy	The restaurant	A chef inside it

Why Threads Are Useful

Responsiveness

Work continues while one thread waits.

Efficiency

Lighter to create and switch than processes.

Parallelism

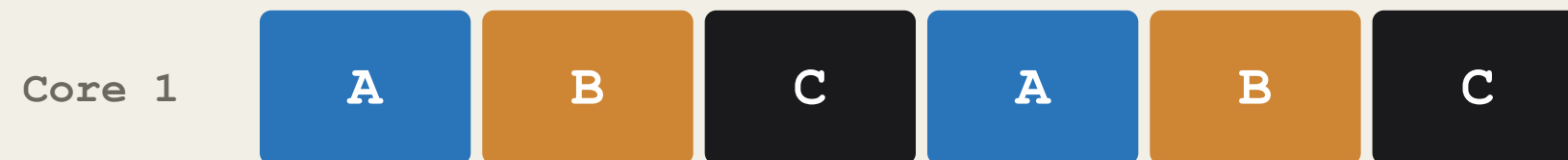
Threads run at once on multiple cores.

RISK Shared memory invites race conditions.

Concurrency vs. Parallelism

Concurrency

One core — taking turns



One chef, three dishes.

Parallelism

Three cores — same instant



Three chefs, three dishes.

Applied Scenario: Smart Hospital

CONCEPT

IN THE HOSPITAL APP

Process	The running application
Heap	The loaded X-ray image
Ready	Analysis waits for CPU time
Waiting	Image still loading from storage
PCB	The OS record for the analysis
Context switch	CPU moves to another app
Threads	UI, image loading, analysis
Parallelism	Threads computing on separate cores

RECAP

Key Takeaways

Program	Passive instructions
Memory layout	Text, data, heap, stack
PCB	The kernel’s record
Thread	Execution sharing resources

Process	A program executing
Process state	New → terminated
Context switch	Save one, restore another
Concurrency	Overlap vs. simultaneity

COMING NEXT

3.2

Process Scheduling

How the OS decides which ready process gets the CPU.