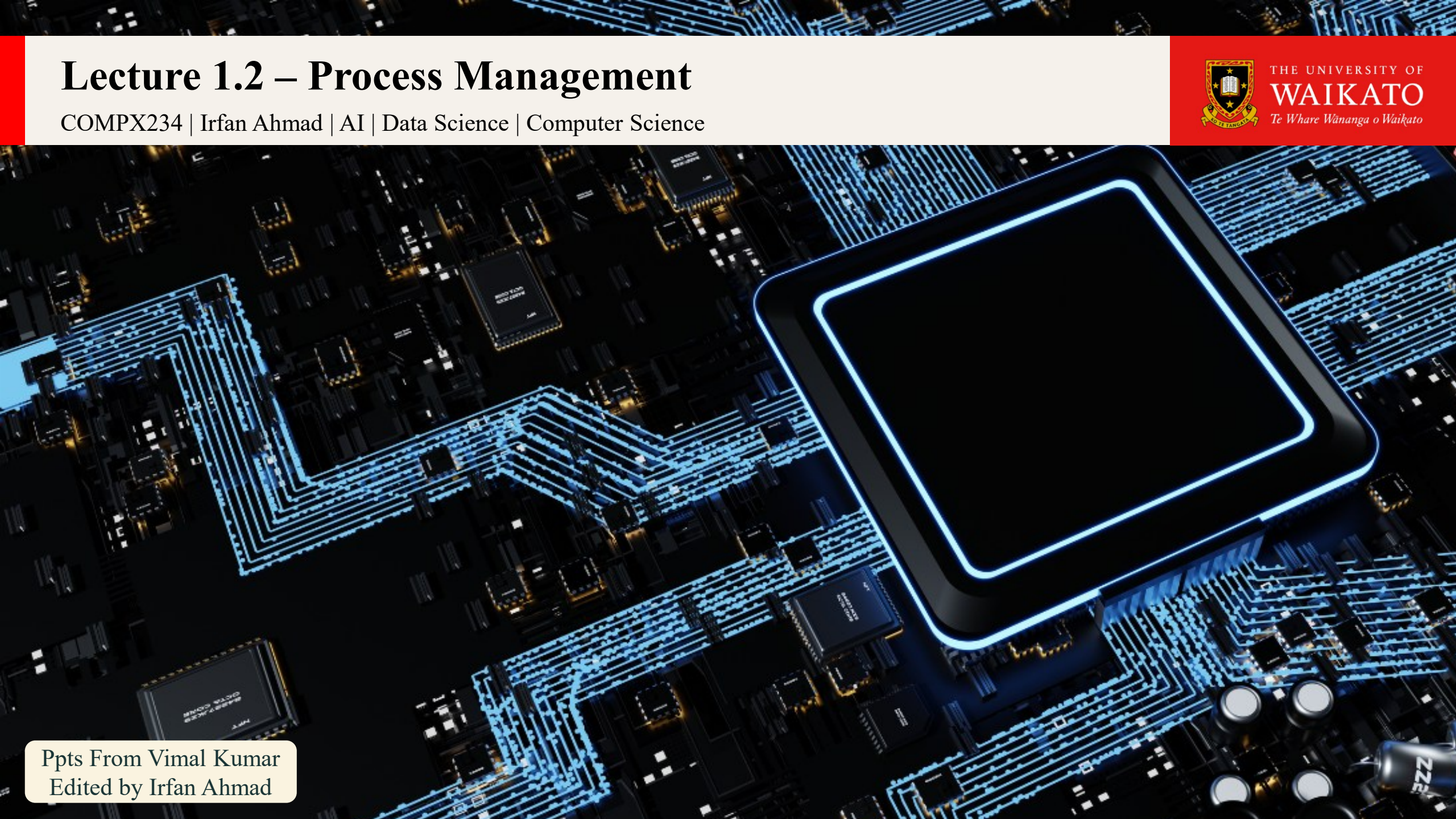
Lecture 1.2 – Process Management

COMPX234 | Irfan Ahmad | AI | Data Science | Computer Science

THE UNIVERSITY OF WAIKATO
Te Whare Wānanga o Waikato



Ppts From Vimal Kumar
Edited by Irfan Ahmad

Outlines

System Organization

Overview of computer system architecture.

Introduction to processes in computing.

Process Concept

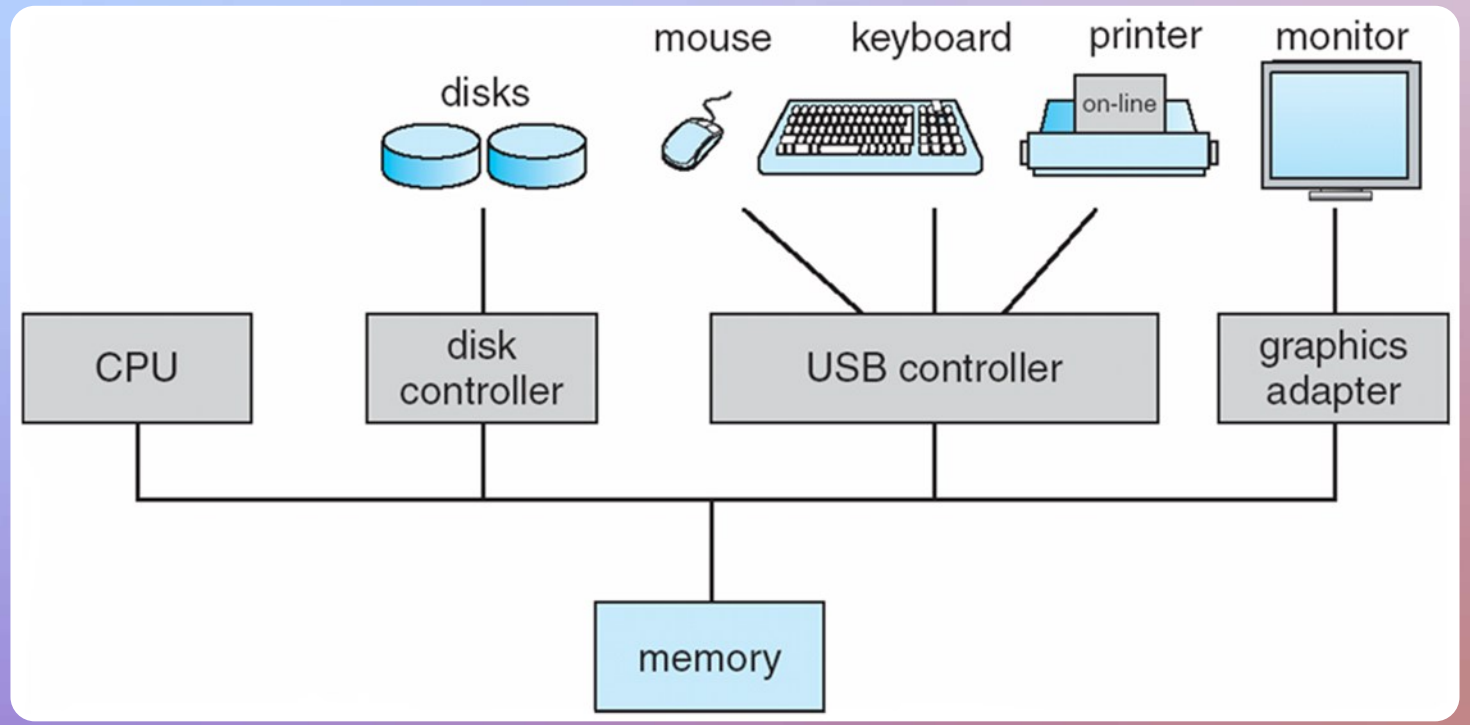
Process Scheduling

Methods for managing process execution.

Computer-System Organization

Computer-System Operation

- **Computer-system operation**
  - **One or more CPUs**, **device controllers** connect through **common bus** providing access to **shared memory**.
  - **Concurrent execution** of ***CPUs and devices*** competing for **memory cycles**.

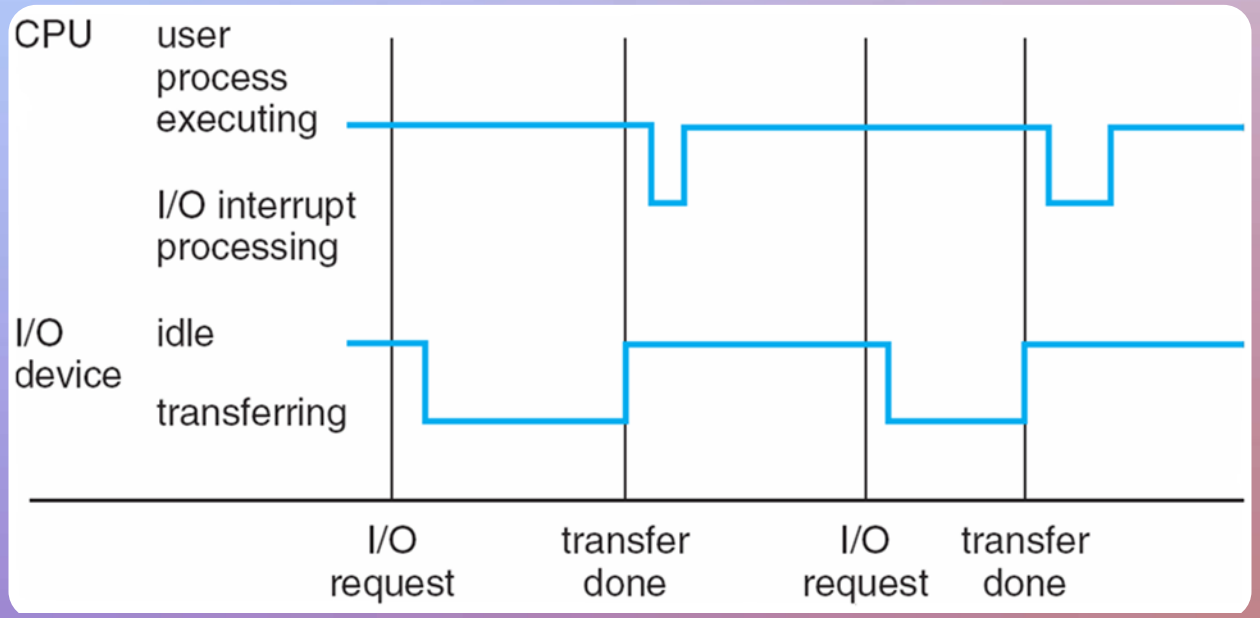


Computer-System Operation

- **I/O devices** and the **CPU** can **execute concurrently.**
- Each **device controller** oversees **a particular device** type.
- Each **device controller** has a **local buffer.**
- **CPU** moves data from/to **main memory** to/from **local buffers.**
- **I/O** is from the **device to local buffer** of controller.
- **Device controller** informs **CPU** that it has **finished its operation** by causing an **interrupt.**

Interrupts

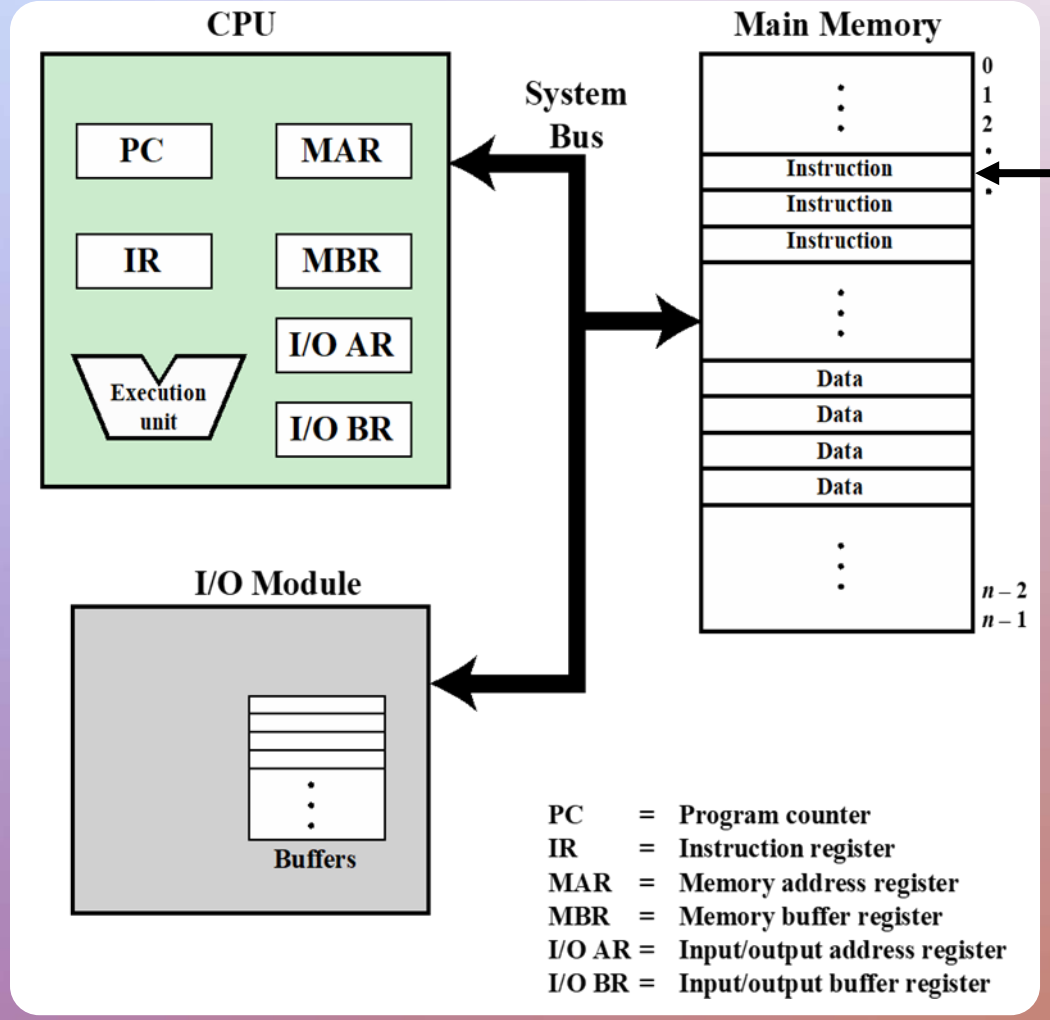
- An **operating system** is **interrupt driven**.
- **Hardware** may **trigger an interrupt** at any time by **sending a signal** to the **CPU**, usually by way of the **system bus**.
- When the **CPU is interrupted**, it **stops what it is doing** and immediately **transfers execution** to a fixed location.



How a Modern Computer Works

- Figure shows the **interplay** of ***all components*** of a **computer system**.

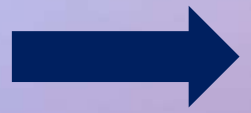
A von Neumann architecture



Program Execution

- High level language to low level language.

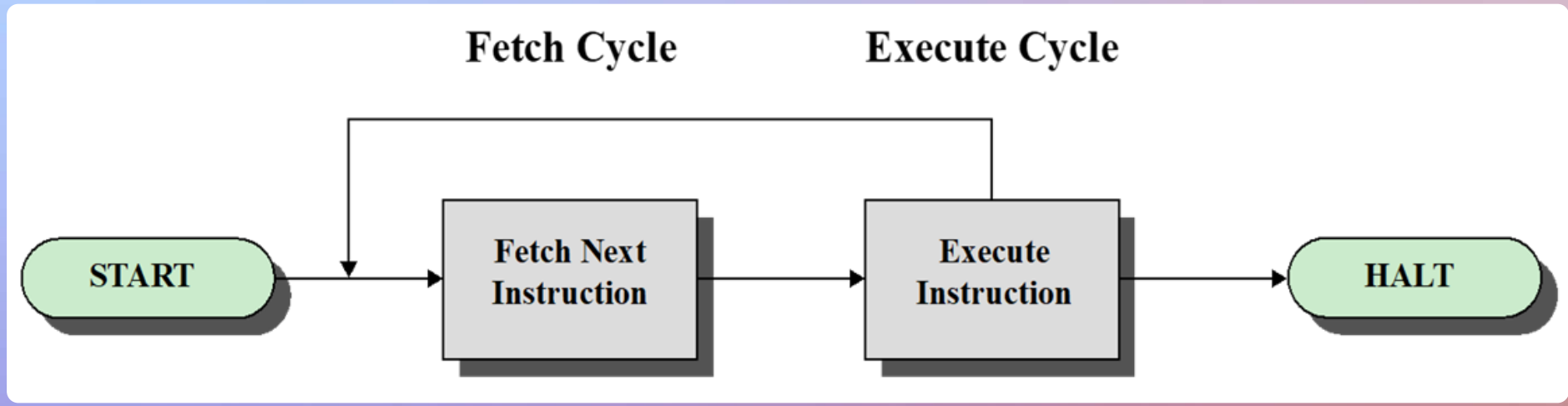
```
int total;
int i;

total = 0;
for (i = 10; i > 0; i--)
   {
     total += i;
   }
```




```
        MOV   R0, #0
        MOV   R1, #10
again   ADD   R0, R0, R1
        SUBS  R1, R1, #1
        BNE   again
halt    B     halt
```

Program Execution

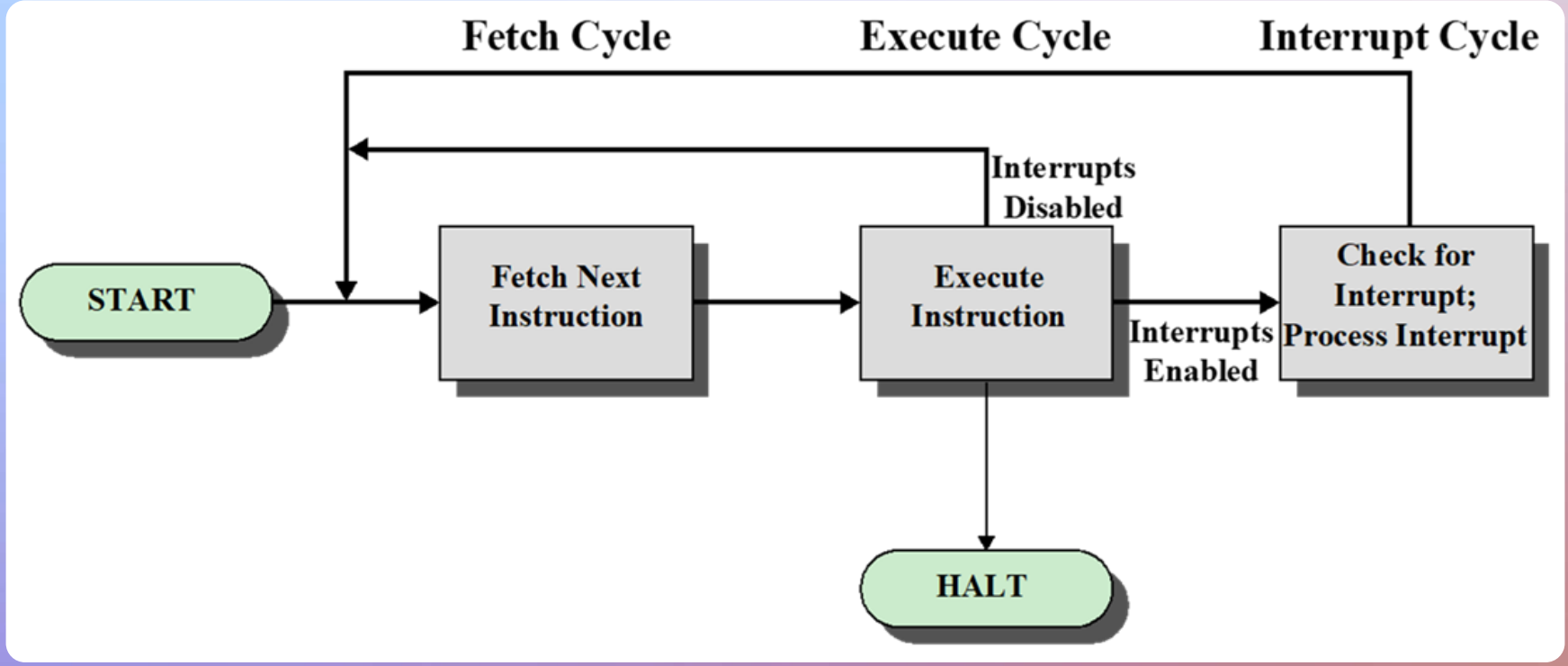
- The **processing** required **for a single instruction** is called an **instruction cycle**.
- Using the **simplified two-step description**, the **instruction cycle is depicted** in Figure below.



Basic Instruction Cycle

Program Execution

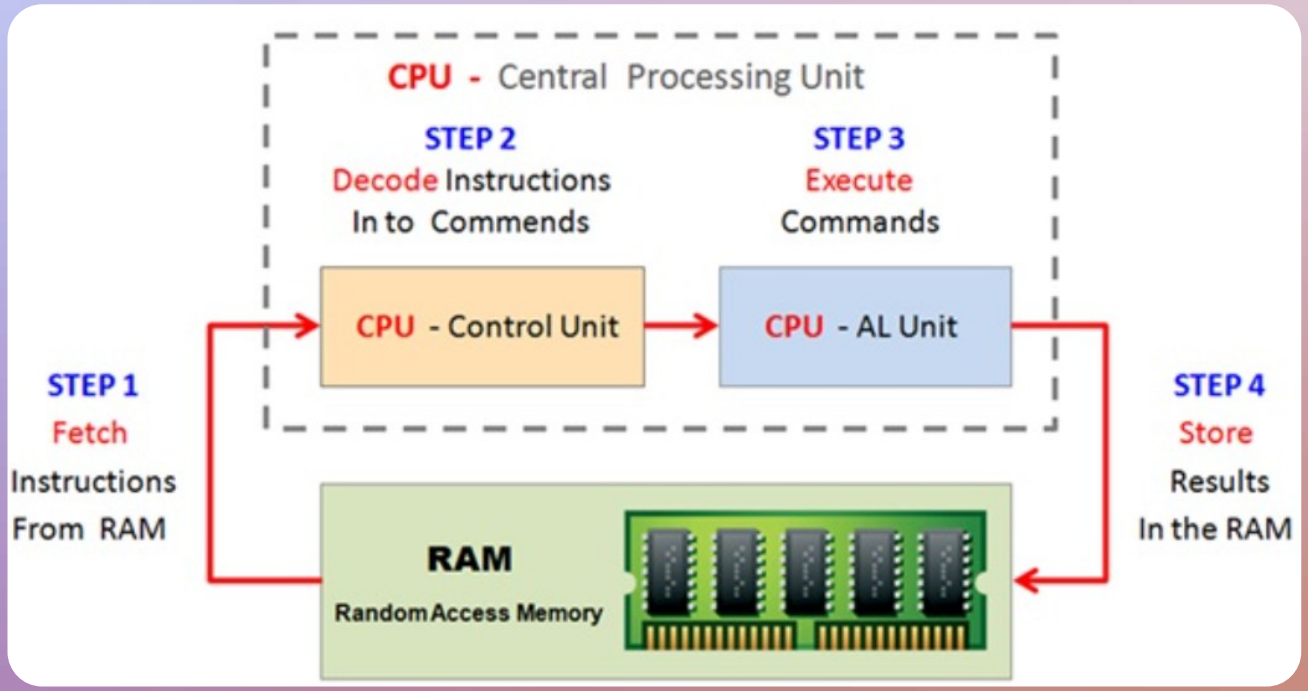
- To **accommodate interrupts**, an **interrupt cycle is added** to the instruction cycle, as shown in Figure below.



Instruction Cycle with Interrupts

Program Execution

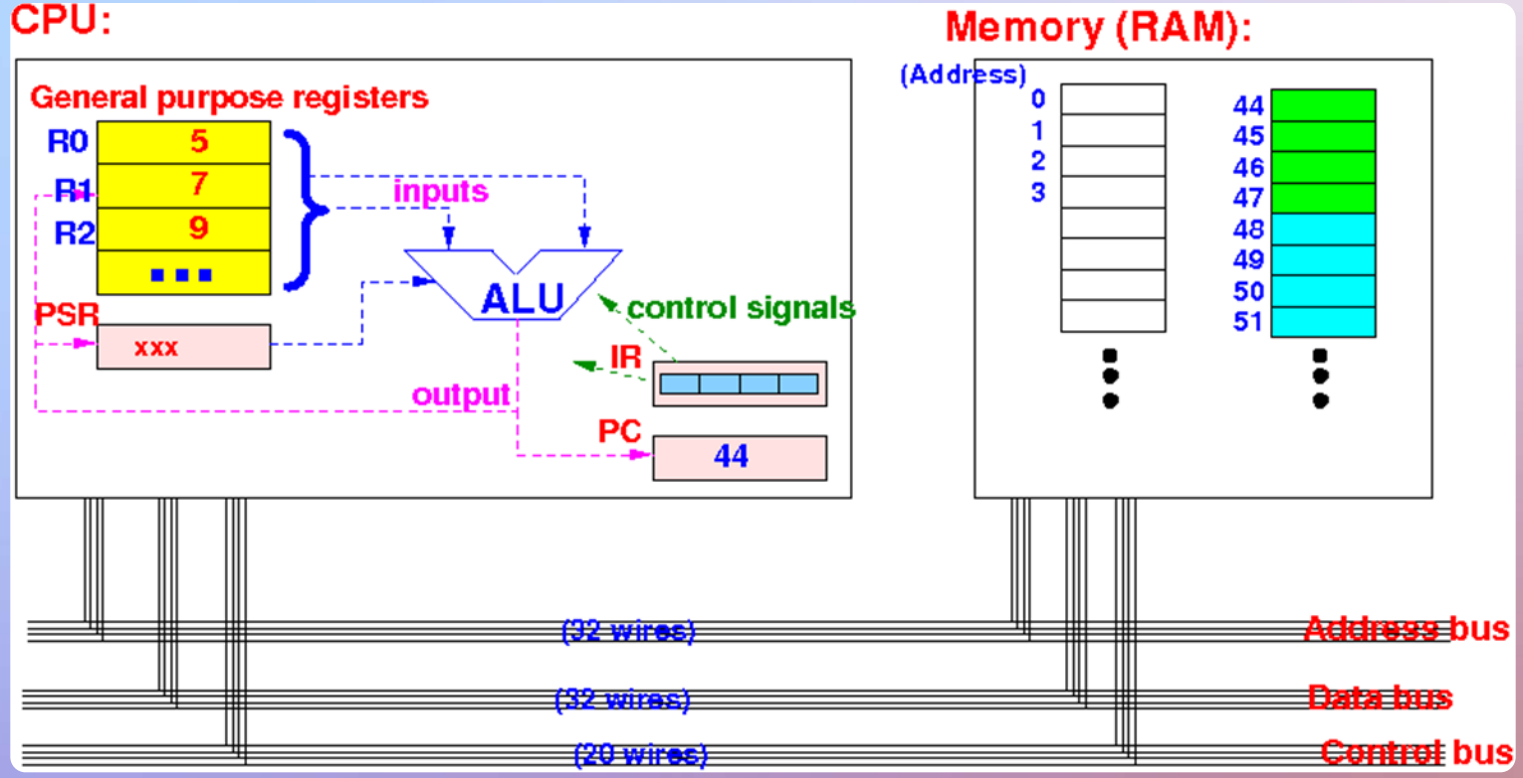
- **Execution sequence:**
  - **Fetch Instruction** at PC
  - **Decode** and **Operand Fetch**
  - **Execute** (possibly using registers)
  - **Write results** to registers/memory
  - **PC** = Next Instruction (PC)
  - Repeat



Execution Sequence

Program Execution – Details of Execution Cycle

What happens inside the **CPU** in the **instruction execution cycle**:

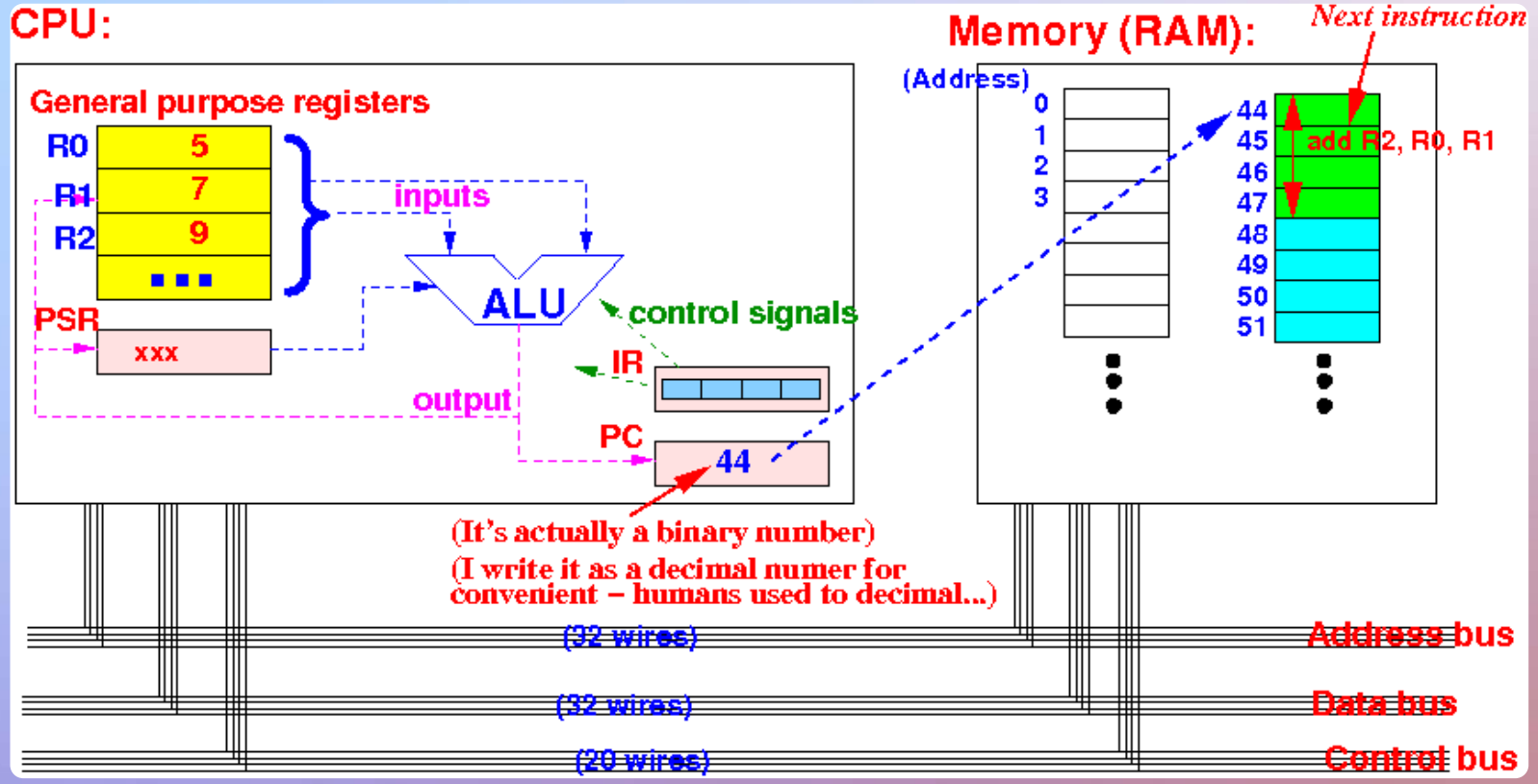


Most instructions executed by the **CPU** will result in **in-order program flow**
Example instruction that result in **in-order program flow**: **add** instruction
We will **examine** the **instruction execution cycle** when the ***next* instruction** execute is **add R2, R0, R1** (add the values in R0 and R1 and store result in register R2)

Program Execution – Details of Execution Cycle

THE UNIVERSITY OF WAIKATO
Te Whare Wānanga o Waikato

Initial situation:

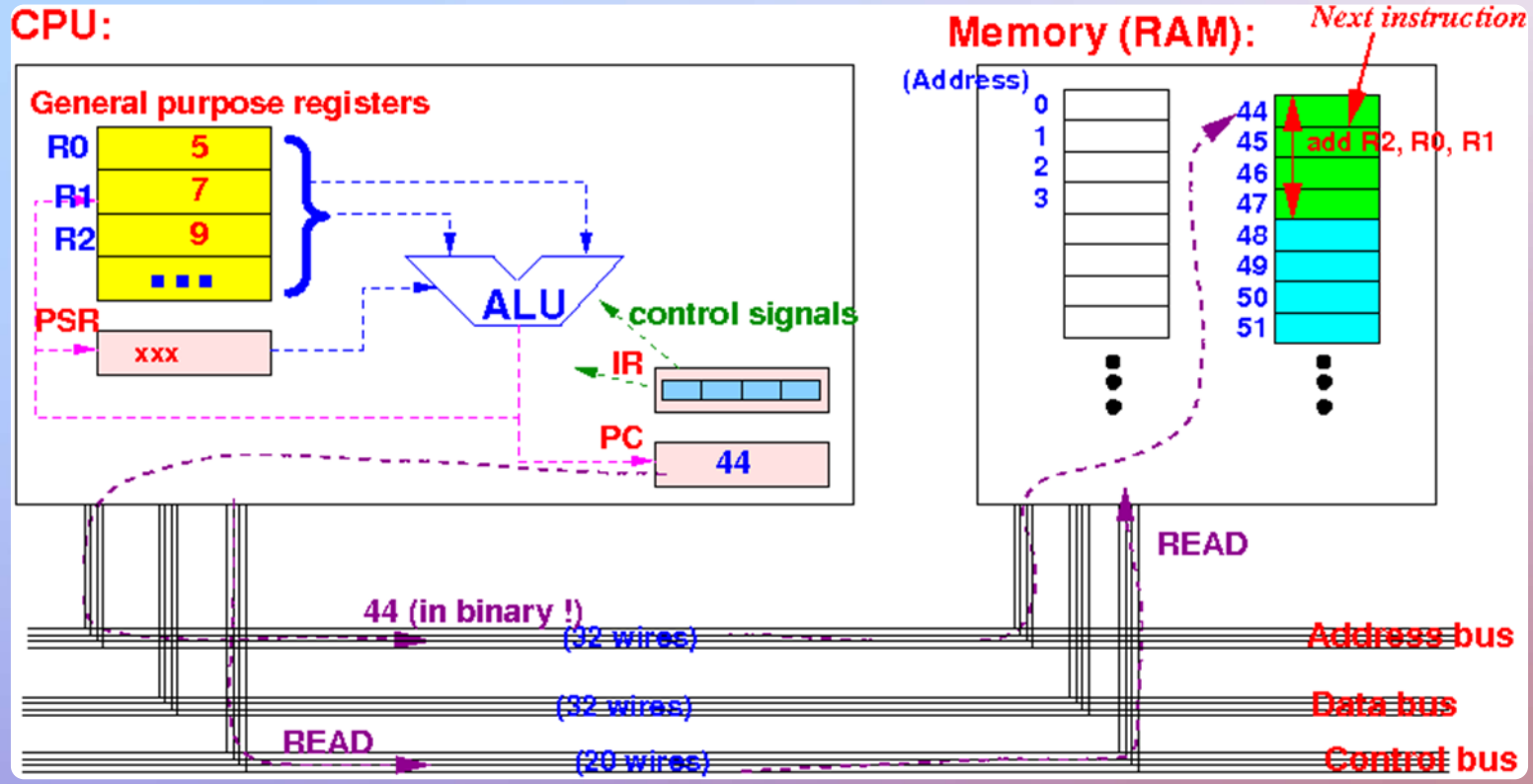


Comment:

PC = 44 (see figure)
So, the ***next*** **instruction** is **located** at **memory address 44**
We assume that **computer instructions** are **4 bytes** long
The **picture** shows that the **4 bytes at memory location 44** contains the **instruction** **add R2, R0, R1**

Program Execution – Details of Execution Cycle

How the CPU completes **Phase 1**, the **instruction fetch** step:



The **CPU** sends:

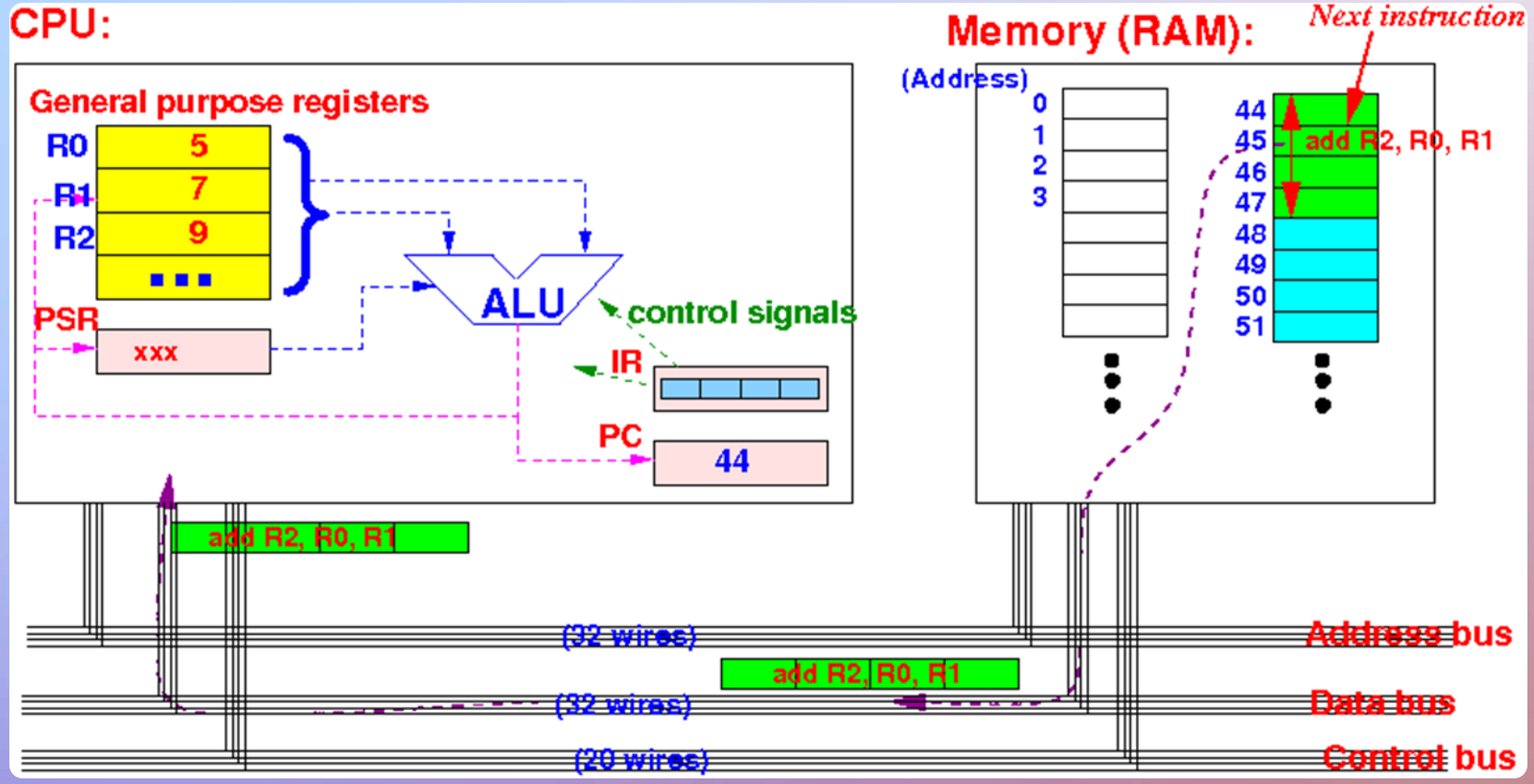
A **Read command** on the **control bus**

The **value in the PC (= 44)** on the **address bus**

These **signals** will tell the **memory** to **return the data** in its **location 44**

Program Execution – Details of Execution Cycle

How the CPU completes **Phase 1**, the **instruction fetch** step:

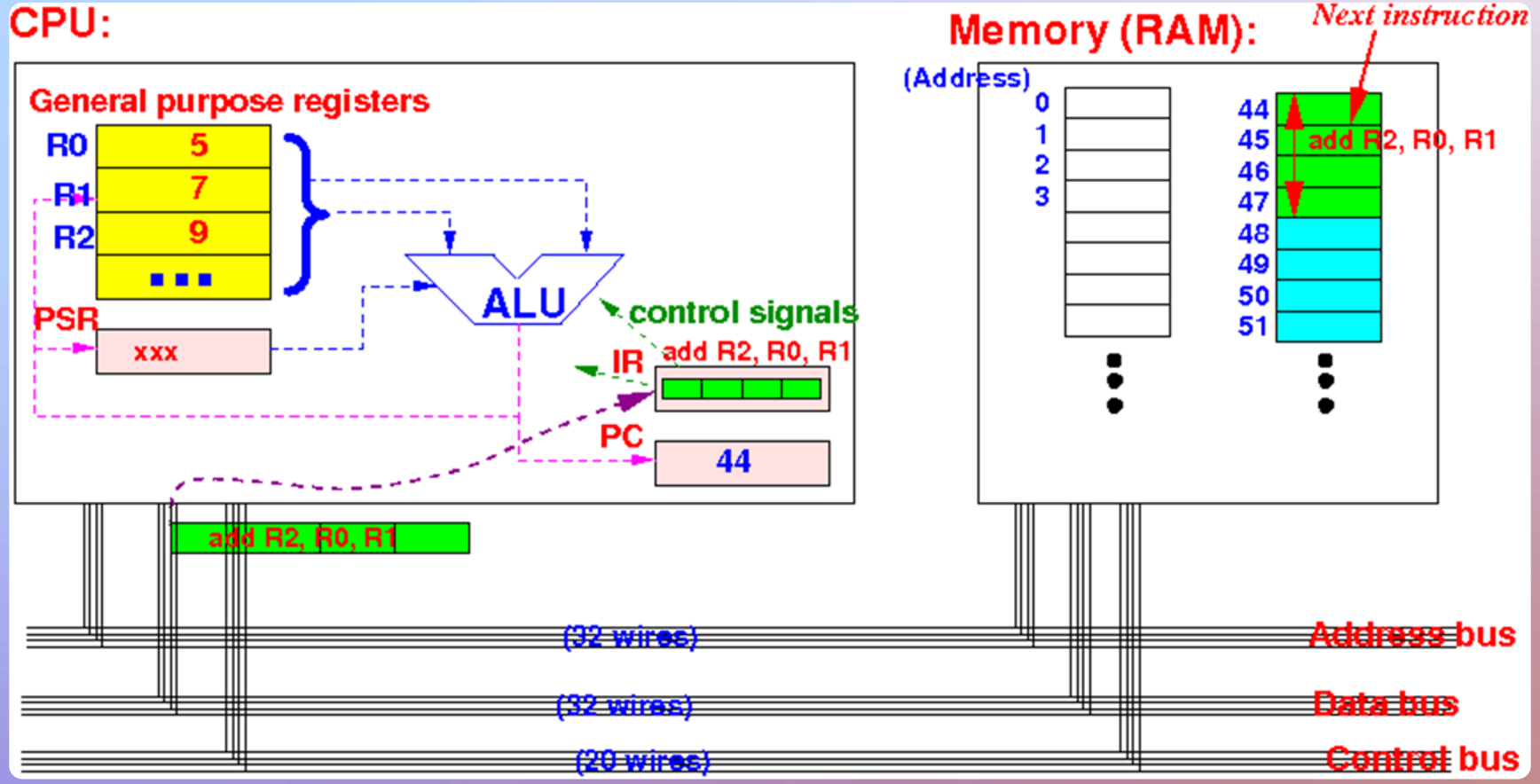


In response, the **memory** will send the **data stored in location 44** onto the **data bus**

* This **data** is the **next instruction** that the **CPU** wants to **execute** !!!

Program Execution – Details of Execution Cycle

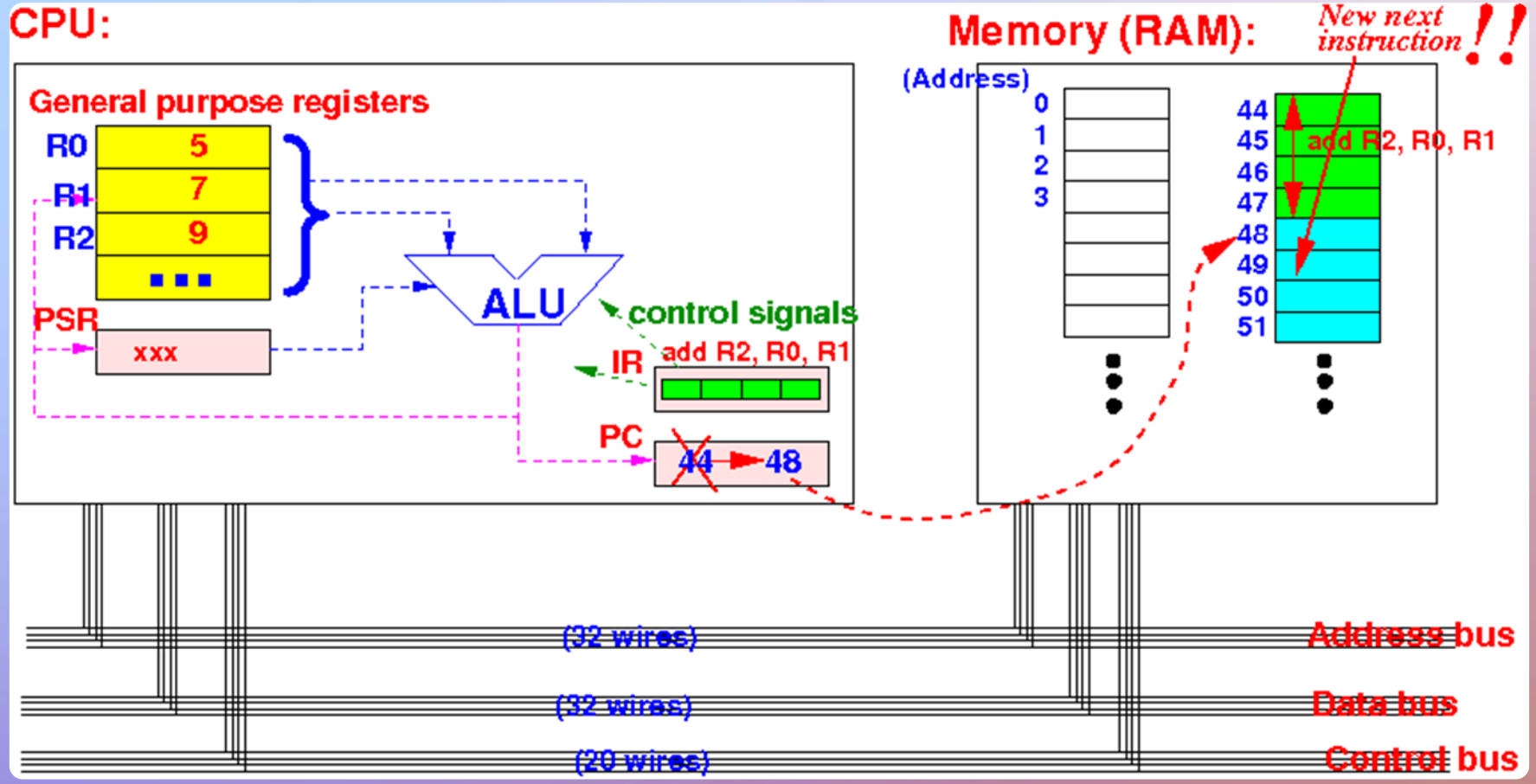
How the CPU completes **Phase 1**, the **instruction fetch** step:



The **CPU** will **capture** the **data** and **store** it in the **Instruction Register**

Program Execution – Details of Execution Cycle

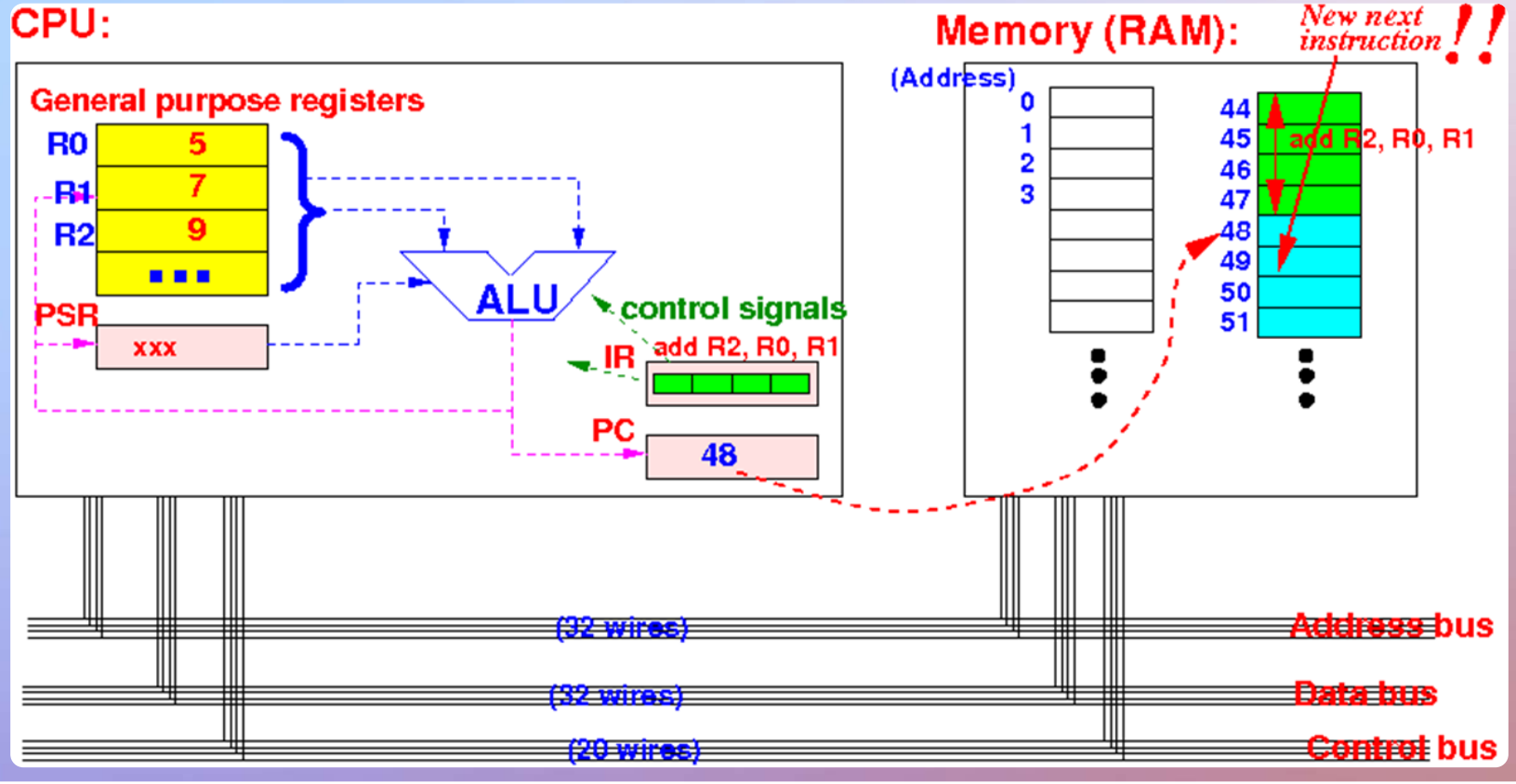
How the CPU completes **Phase 1**, the **instruction fetch** step:



Finally: the **CPU** will **increase the PC** by **4 (instruction size)** to **point** to the **(new) *next* instruction**

Program Execution – Details of Execution Cycle

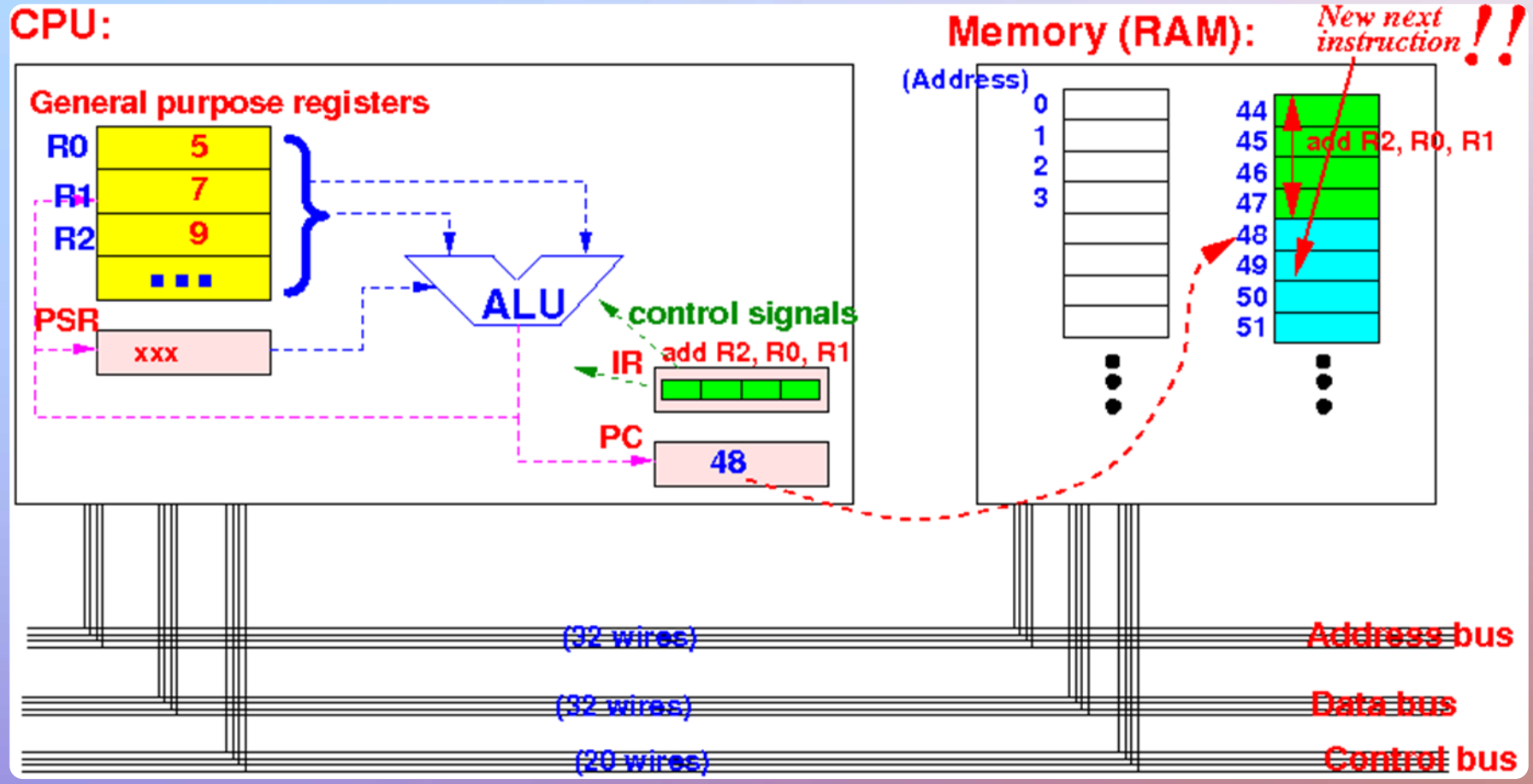
Situation after **Phase 1**, the **instruction fetch** step:



(1) The **CPU** has **fetched** the **current instruction** into the **Instruction Register (IR)**
    * The **remainder** of the **instruction execution cycle** are used to **process** the **current instruction** !!!
(2) The **PC** has been **updated** and now **points** to the **next instruction** of the ***next*** **instruction execution cycle**
    * This **implements** the **(default) in order program flow** execution ordering !!!

Program Execution – Details of Execution Cycle

How the CPU completes **Phase 2**, the **instruction decode** step:

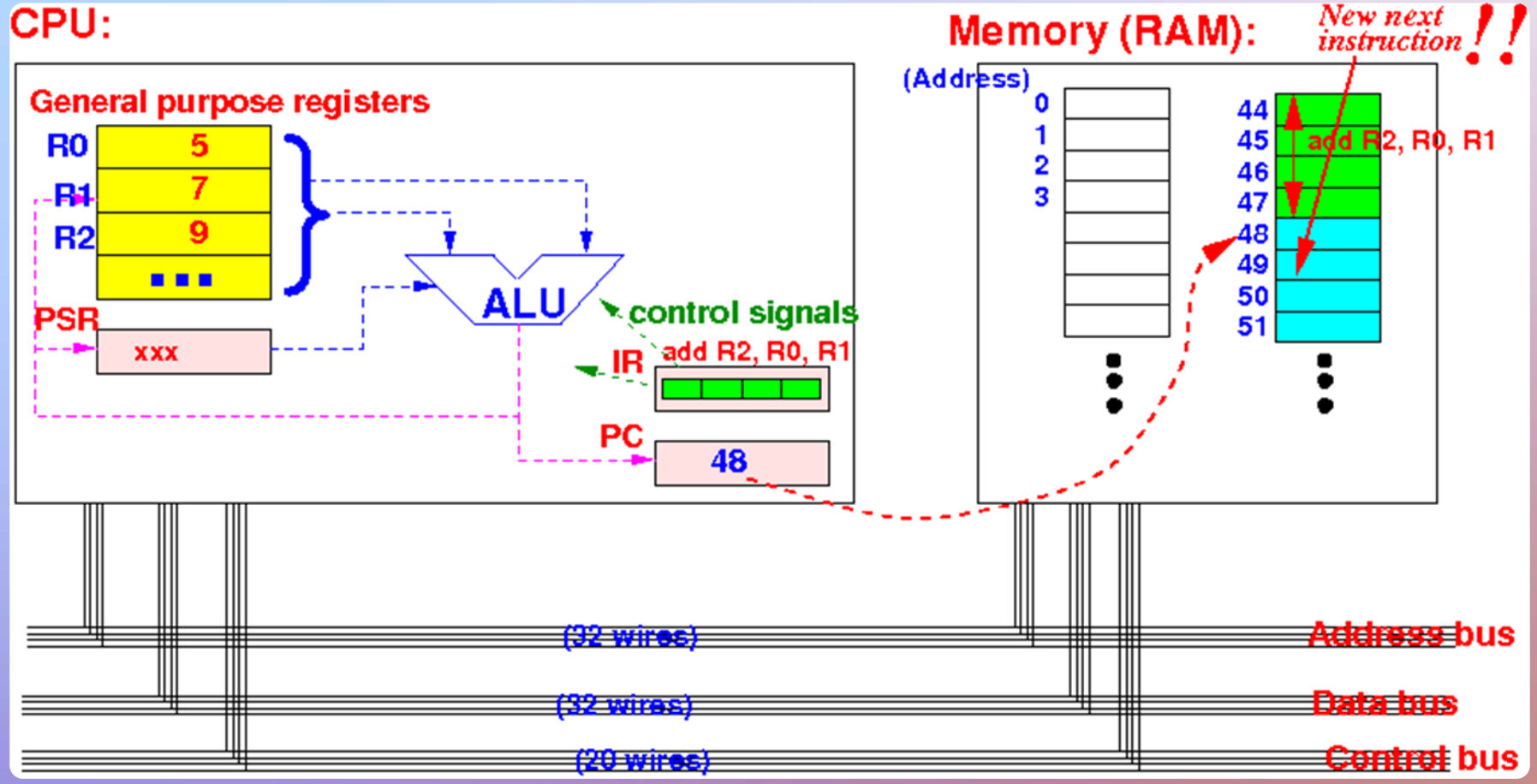


All that you need to know **for COMPX234** is:

The **CPU now** know **what operation** it needs to perform and with **which operands.**

Program Execution – Details of Execution Cycle

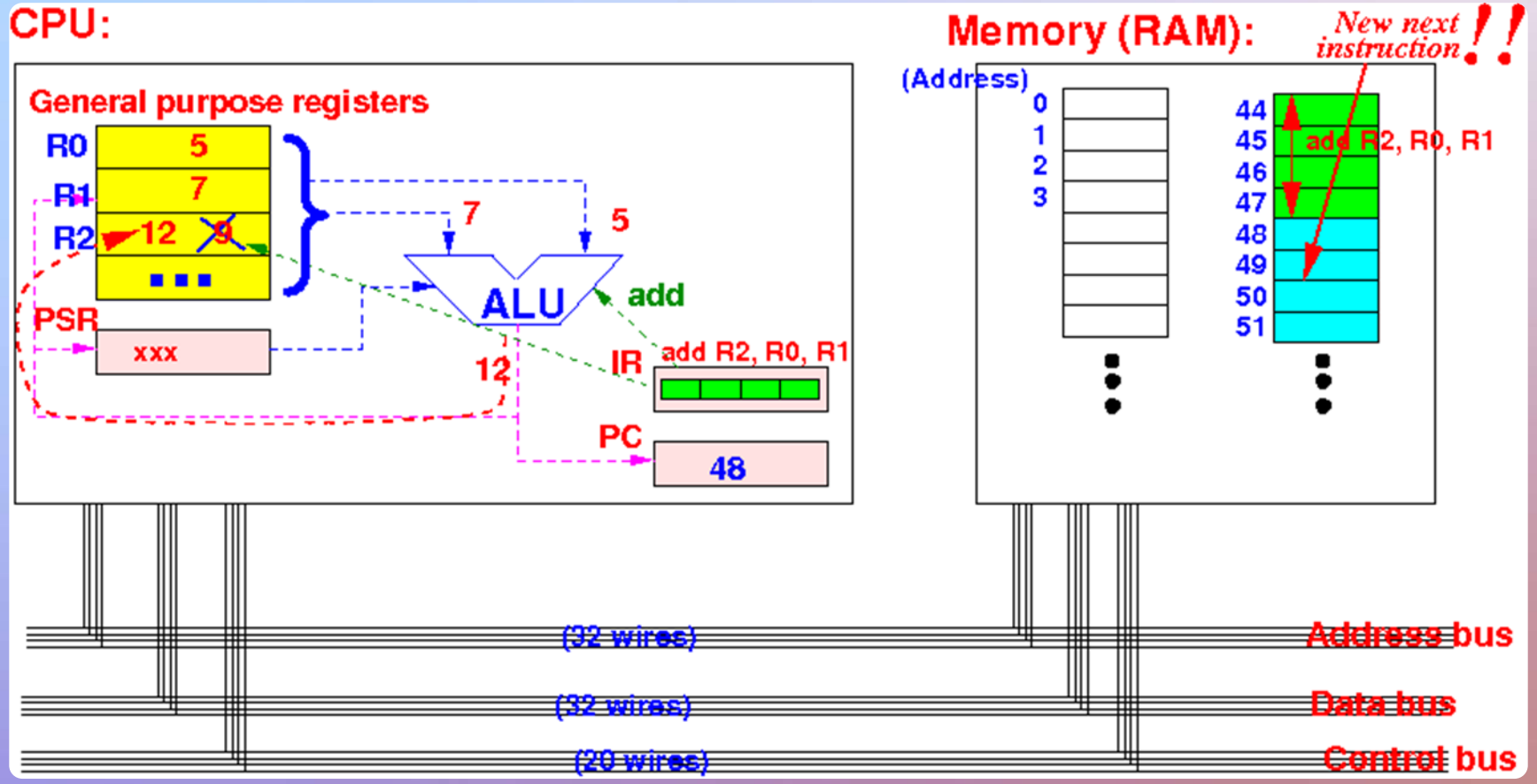
How the CPU completes **Phase 3**, the **operand fetch** step:



The **CPU** sends **control signals** to the **specific registers** and **fetch** their **values** to the **input** of the **ALU**

Program Execution – Details of Execution Cycle

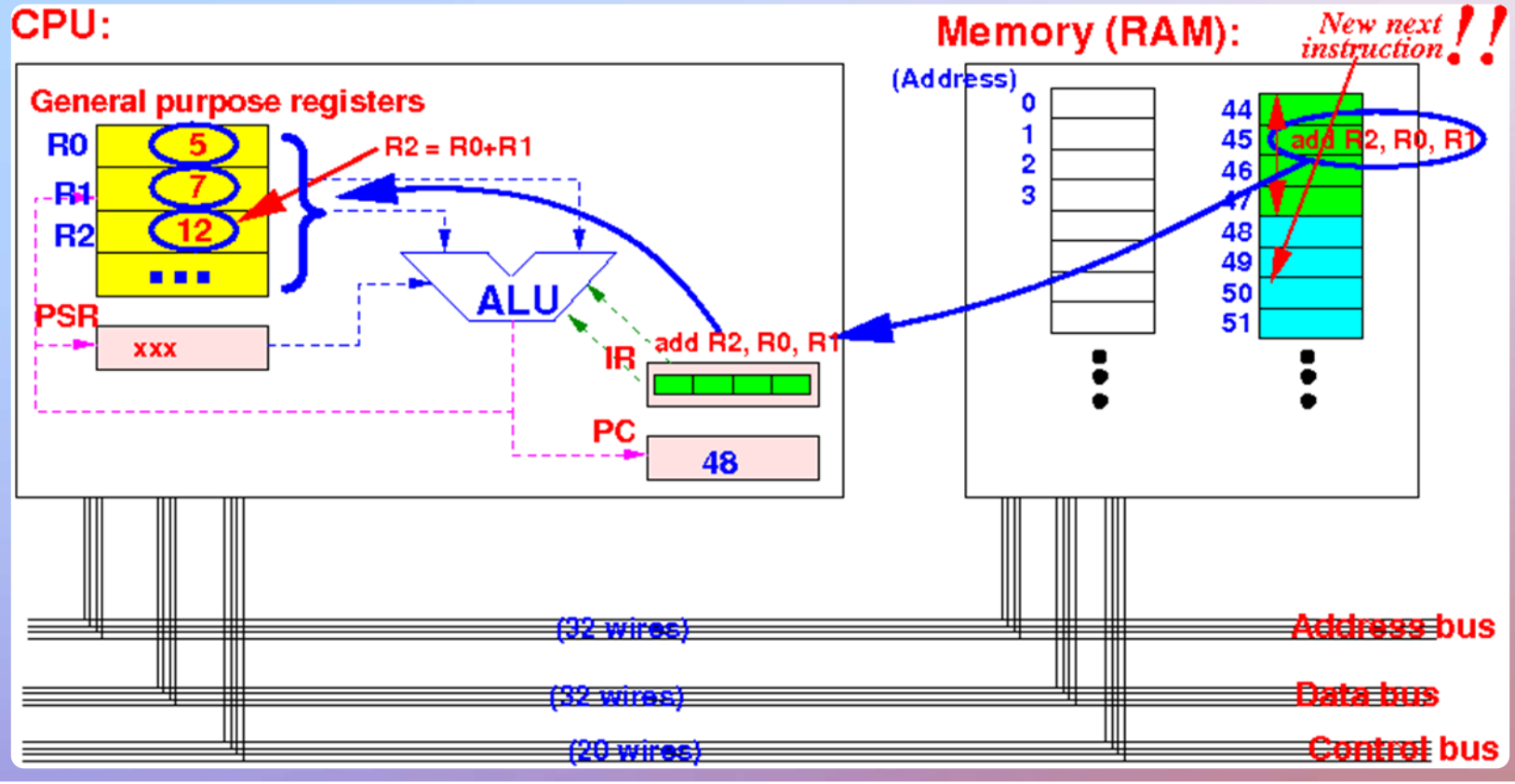
How the CPU completes **Phase 4**, the **instruction execution** step:



The **CPU** instructs the **ALU** to the **add the input values** and then **store** the result in the **specified (destination) register (R2)**

Program Execution – Details of Execution Cycle

Situation at the end of **Phase 4**, the **instruction execution** step:



The **CPU** has **finished executing** the **instruction** **add R2, R0, R1** !!!

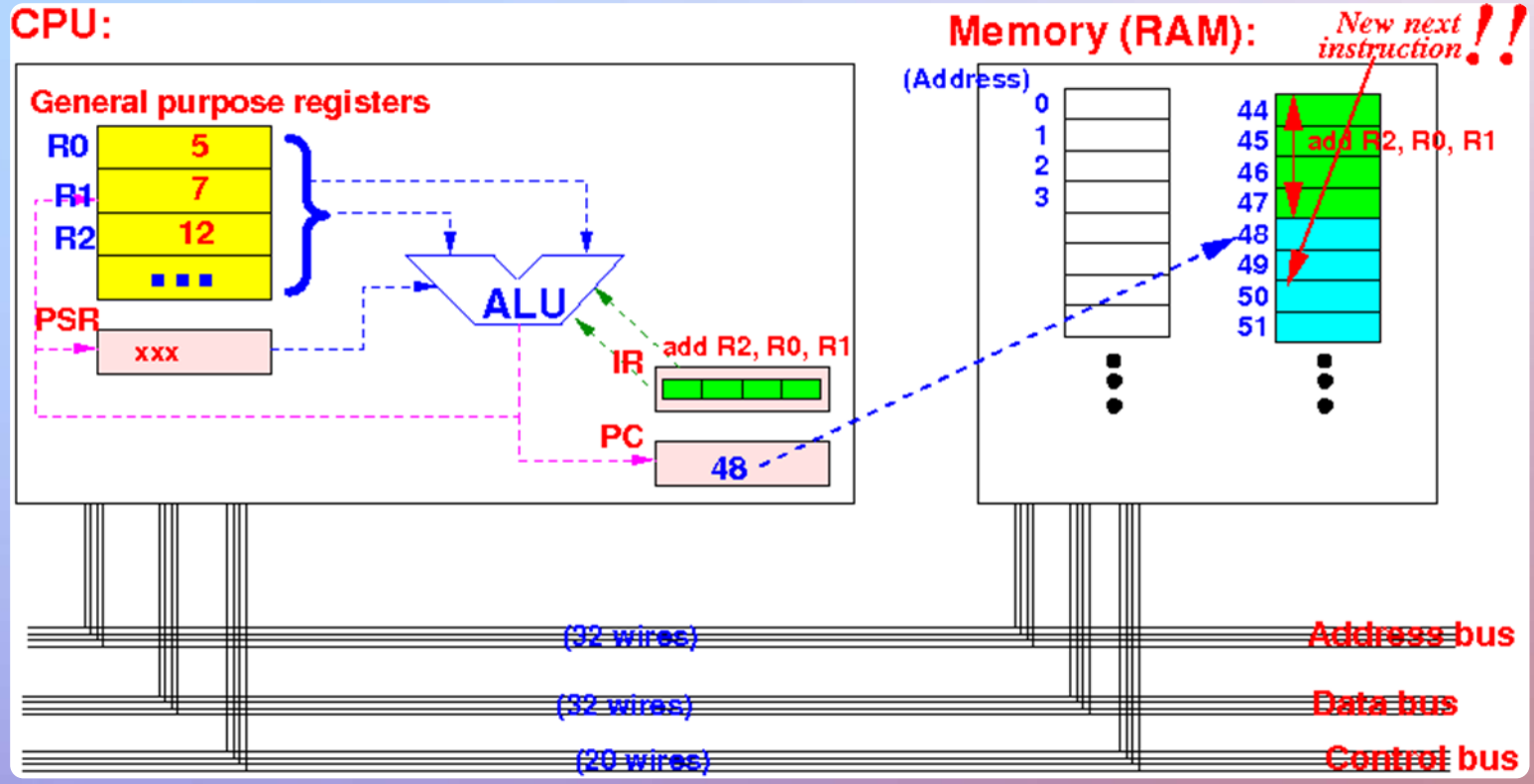
(The CPU has added the values in R0 and R1; and stored the result in register R2 !!!)

The **instruction execution cycle** has **ended** !!

(And a **new instruction execution cycle** will **begin *immediately*** !!)

Program Execution – Details of Execution Cycle

Initial situation of the ***new*** **instruction execution cycle**:

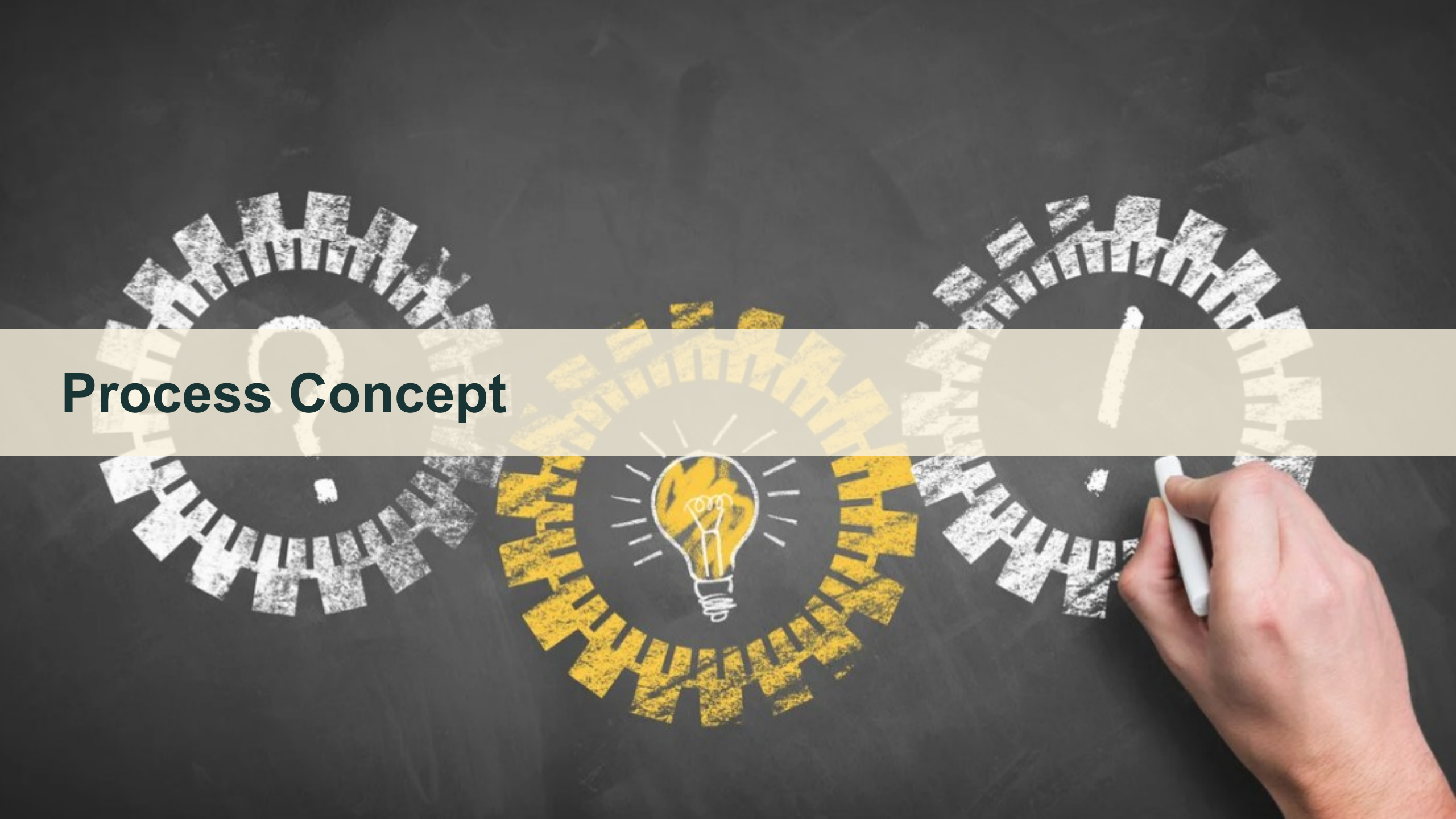


Comment:

So, the ***next*** **instruction** is **located** at **memory address 48, PC = 48** (see figure)
The **CPU** will **execute** the **"new" next instruction** in the ***new*** **instruction execution cycle** !!!

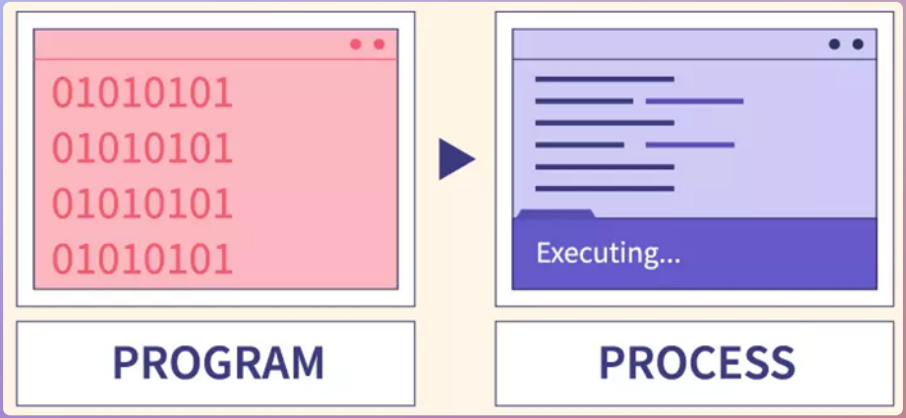
And this **sequential instruction execution** will happen **repeatedly** unless the **CPU** fetches an **instruction** that causes an **out-of-order program flow**

Process Concept

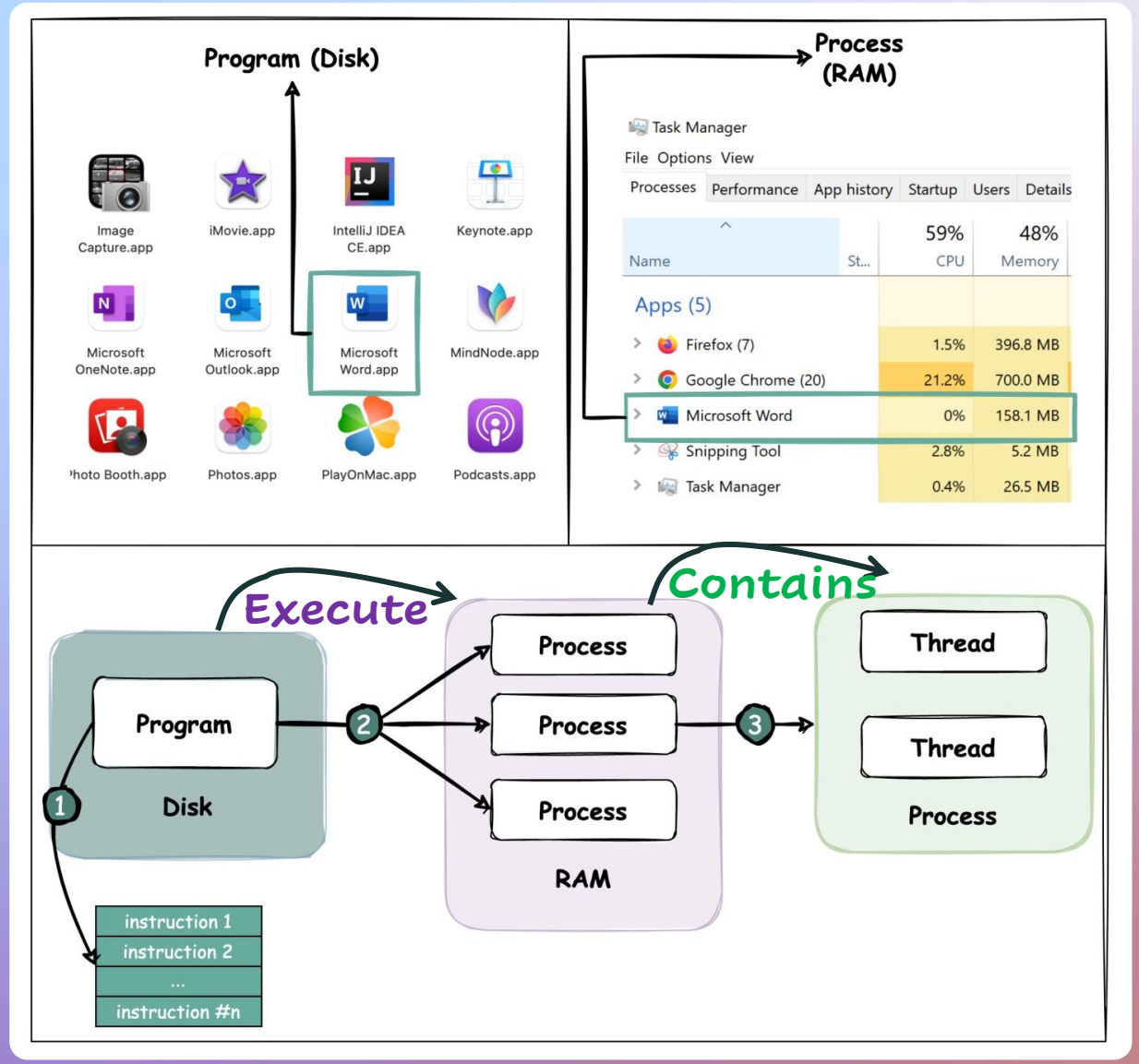


Program Vs Process

- **Process** – a **program in execution**
- **Program** is **passive entity** stored on disk (executable file), **process** is **active**
  - Program becomes process when executable file loaded into memory
- **Execution of program** started via **GUI mouse clicks**, **command line entry** of its name, etc.
- **One program** can result in the **creation of several processes**
  - E.g., Google Chrome

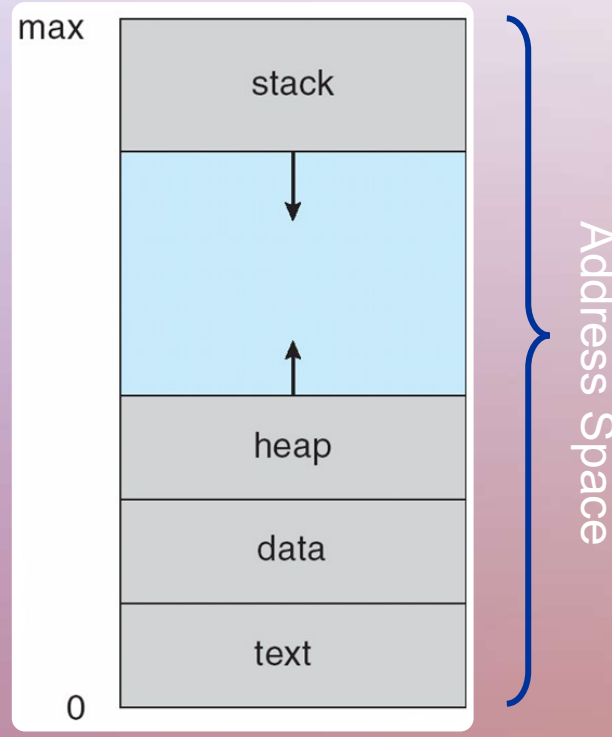


Program Vs Process



Process in Memory

- When a **program is executed**, the **resulting process** is ***allocated memory***
- The ***allocated memory*** is **divided** into **multiple sections**:
  - **Text section** — **the executable code**
  - **Data section** — the **global variables**
  - **Heap section** — **memory** that is dynamically ***allocated during run time.***
  - **Stack section** — **temporary data** when calling functions
    - Function parameters, return addresses, local variables

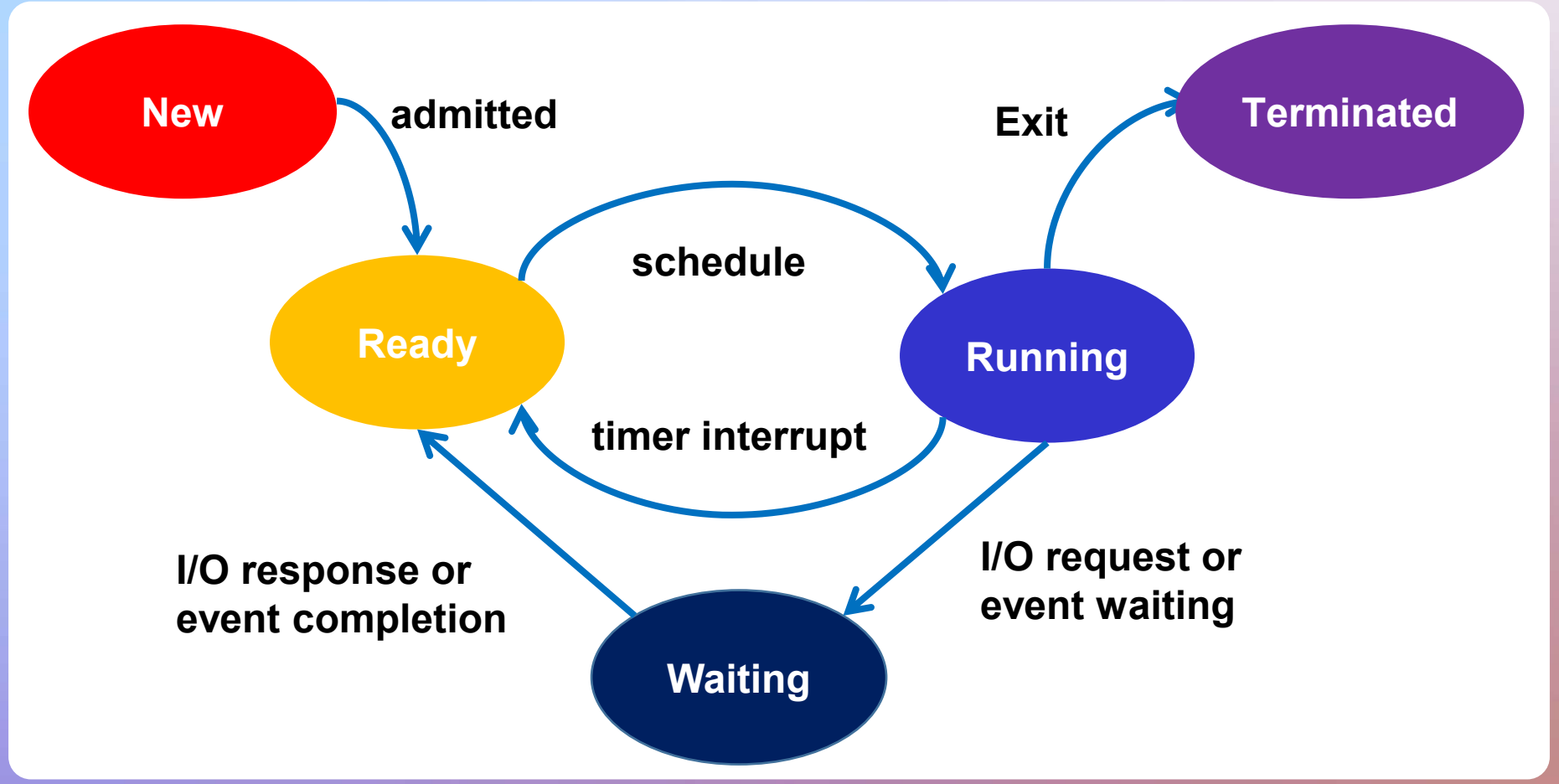


Layout of a process in memory.

Process States

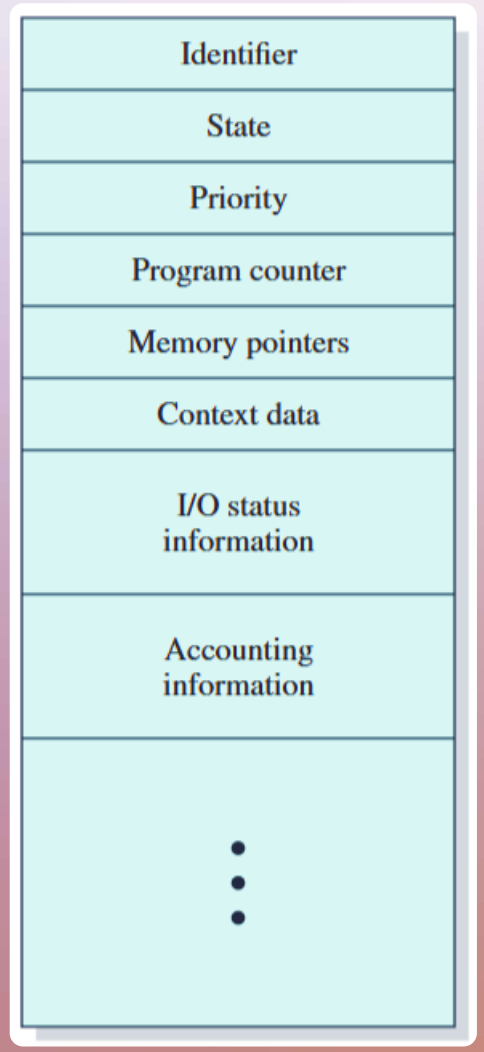
- As a **process executes**, the **operating system** keeps track of it.
- The **tracking** is done by ***using a state machine.***
- A **process** can be in one of the following **five states**:
  - **new:** The process is being created
  - **running:** Instructions are being executed
  - **waiting:** The process is waiting for some event to occur
  - **ready:** The process is waiting to be assigned to a processor
  - **terminated:** The process has finished execution

Process State Diagram



Process Control Block (PCB)

- **Data structure(s)** to keep **information** associated with **each process** (also called ***task control block***):
  - **Identifier** – A unique identifier associated with this process
  - **Process state** – running, waiting, etc.
  - **Priority** – Priority level relative to other processes.
  - **Program counter** – The address of the next instruction in the program to be executed
  - **Memory pointers** – memory allocated to the process
  - **Context data** – data that is present in registers in the processor
  - **I/O status information** – I/O devices allocated to process, list of open files
  - **Accounting information** – CPU used, clock time elapsed since start, time limits



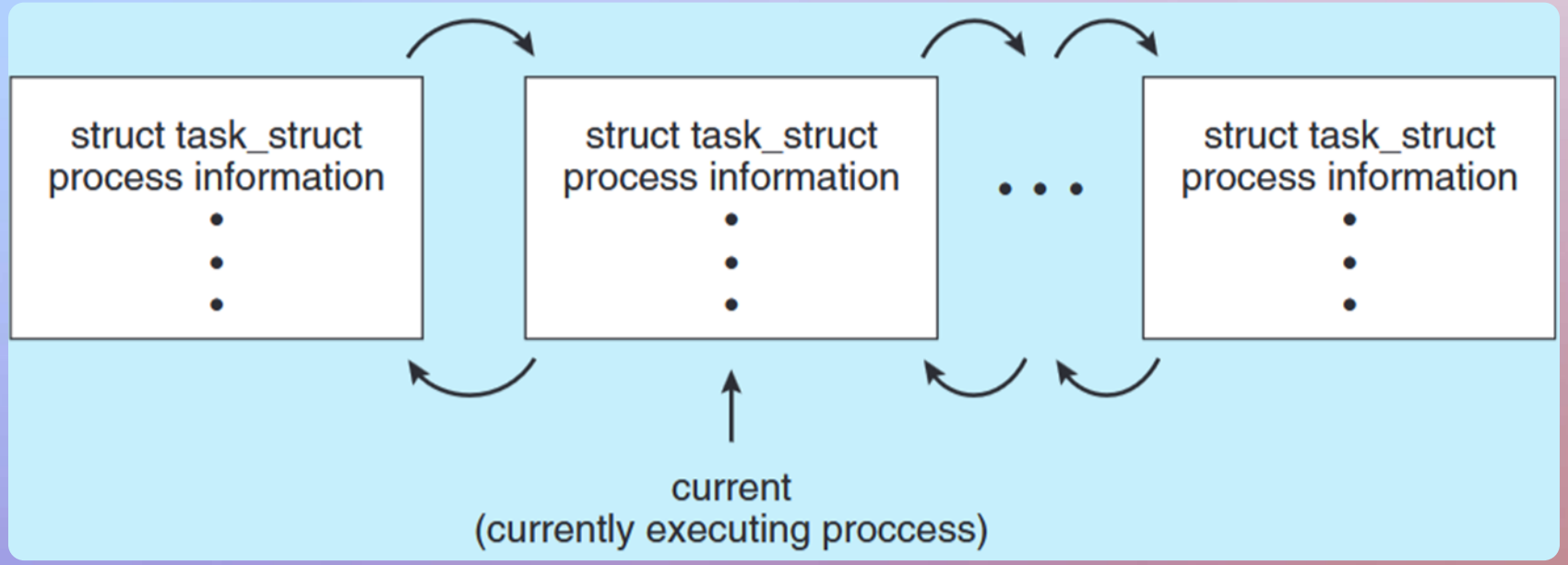
Process Representation in Linux

- Represented by the C structure **task_struct**.
- This structure contains all the necessary information for representing a process:
  - For example, the state of a process is represented by the field long state in this structure.

```
1   pid t_pid; /* process identifier */
2   long state; /* state of the process */
3   unsigned int time_slice /* scheduling information */
4   struct task_struct *parent; /* this process's parent */
5   struct list_head children; /* this process's children */
6   struct files_struct *files; /* list of open files */
7   struct mm_struct *mm; /* address space of this process */
```

Process Representation in Linux

- Within the **Linux kernel**, all **active processes** are represented using a ***doubly linked list*** of task struct. The **kernel** maintains a **pointer–current–to** the process currently executing on the system, as shown below:



Context Switch

- **Interrupts** cause **CPU** to **switch from process to process**.
- When **CPU switches** to another process, the system must **save the state** of the **old process** and **load** the **saved state** for the **new process** via a context switch.
- **Context of a process** represented in the **PCB**.
- **Context-switch** time is **overhead**; the system does **no useful work** while switching.
  - The more *complex the OS* and the *PCB* → the *longer the context switch*.
- **Time dependent** on **hardware support**:
  - Some *hardware provides* multiple *sets of registers per CPU* → *multiple contexts loaded* at once

Context Switch

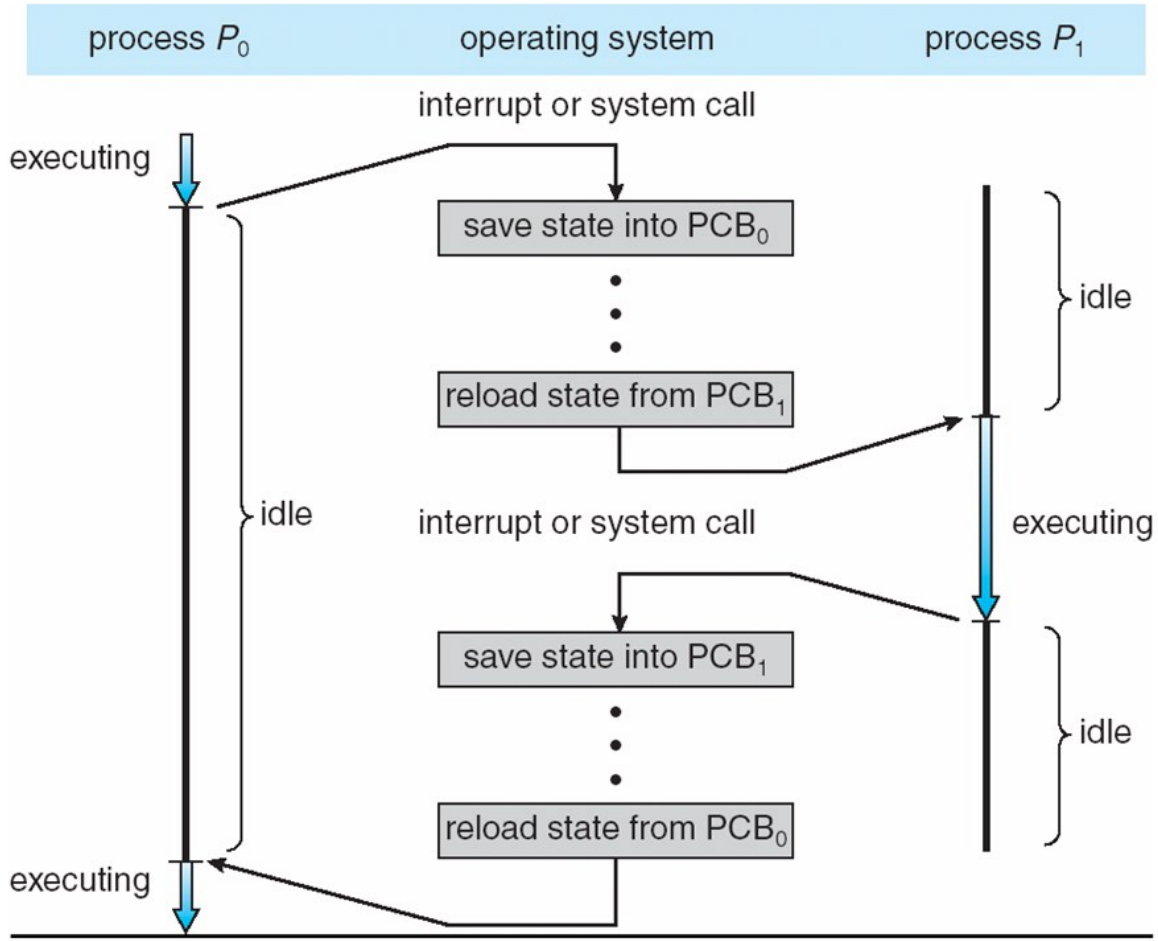
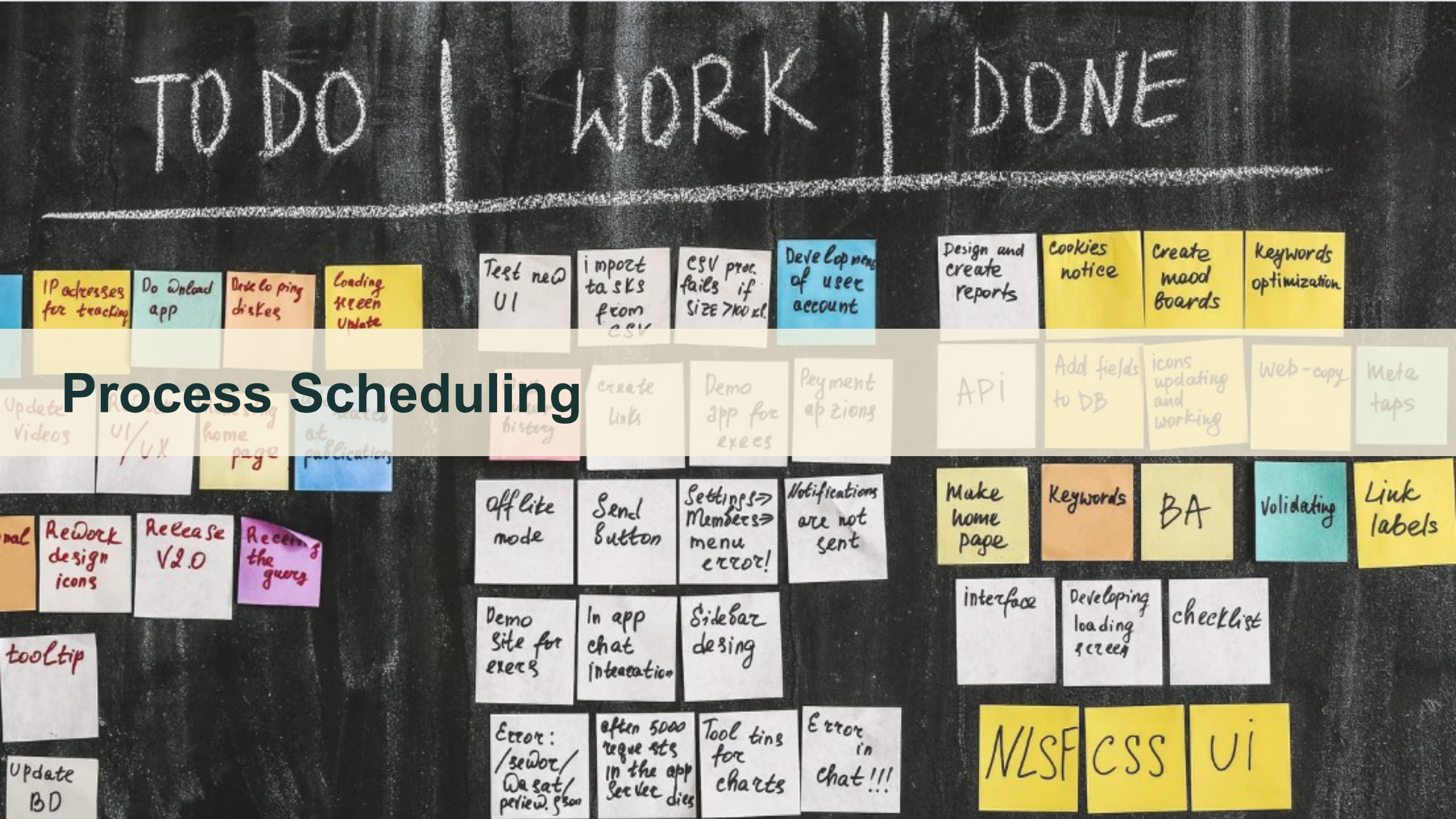


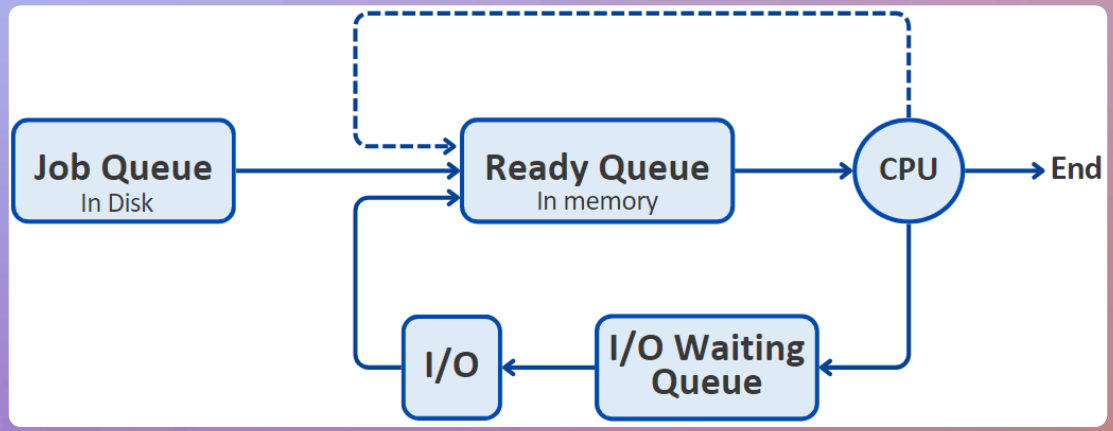
Diagram showing context switch from process to process.



Process Scheduling

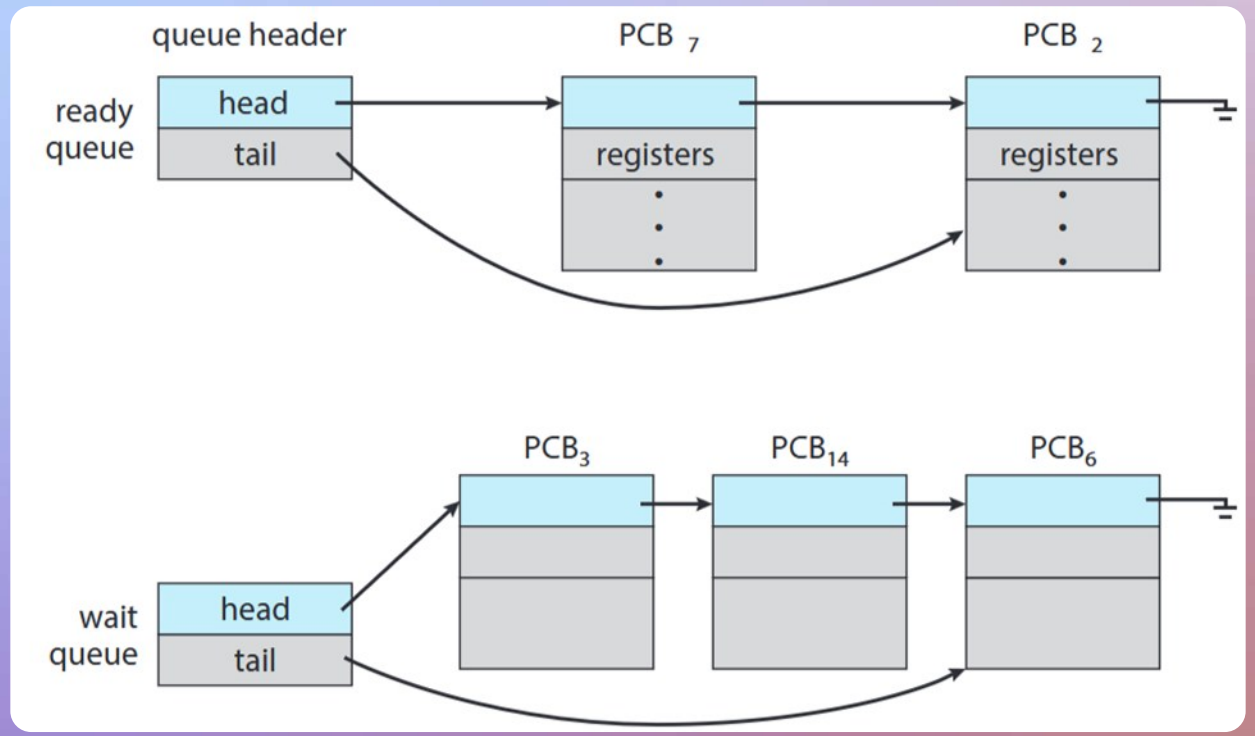
Process Scheduling

- Maximize **CPU** use, quickly **switch processes** onto **CPU** for **time sharing**.
- **Process scheduler** selects among **available processes** for **next execution** on **CPU**.
- Maintains **scheduling queues** of processes:
  - **Job queue** – set of all processes in the system (Disk)
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device
- ***Processes migrate*** among the various queues.



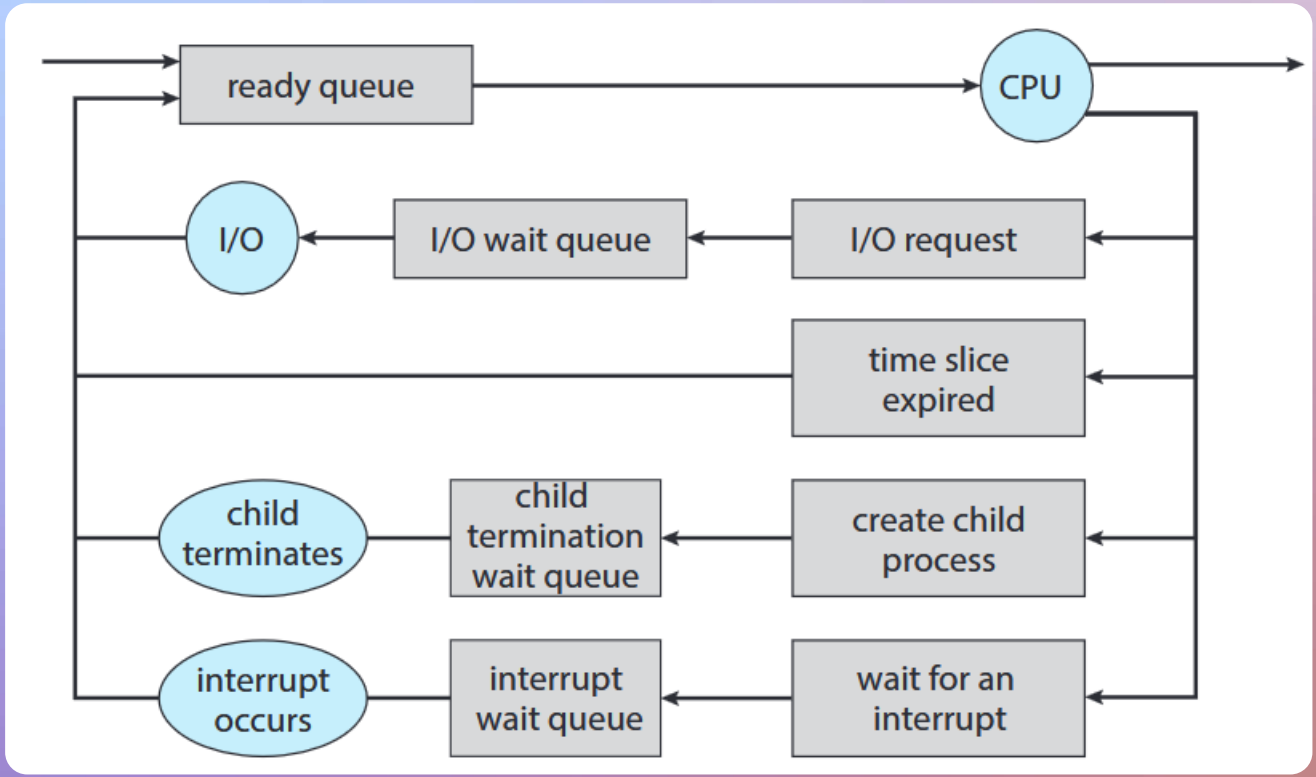
Process Scheduling

- As **processes enter the system**, they are **put into a ready queue**, where they are **ready** and **waiting** to execute on a CPU's core.
- **Processes** that are **waiting** for a **certain event to occur** — such as **completion of I/O** — are **placed** in a **wait queue**.



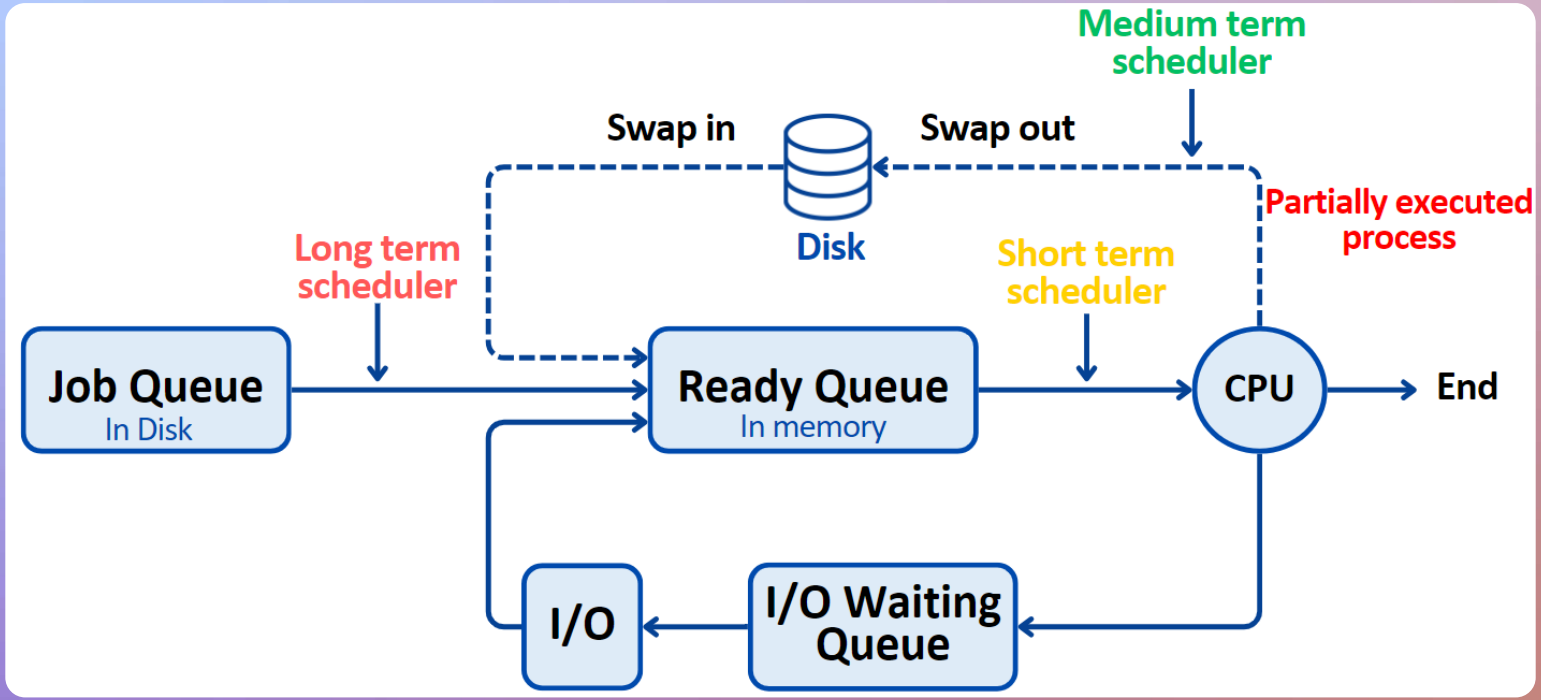
Process Scheduling

- **A common representation** of **process scheduling** is a **queueing diagram**.
  - **Two types** of queues are present: the **ready queue** and a **set of wait queues**.
  - The **circles** represent the **resources** that serve the queues.



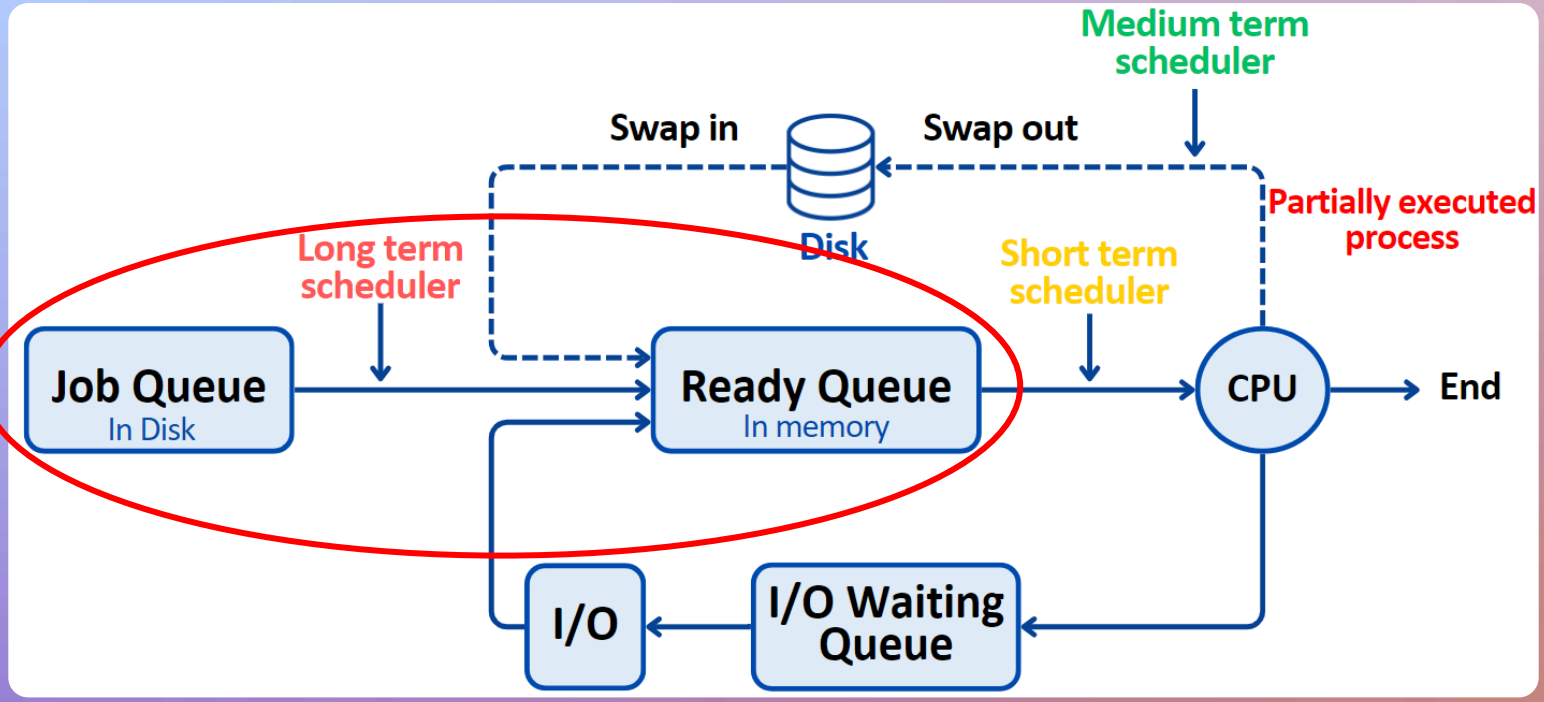
Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue.
  - Long-term scheduler is invoked infrequently (seconds, minutes) → (may be slow)
  - The long-term scheduler controls the degree of multiprogramming
  - The degree of multiprogramming refers to the number of programs that are loaded into the memory and are ready to execute at any given time



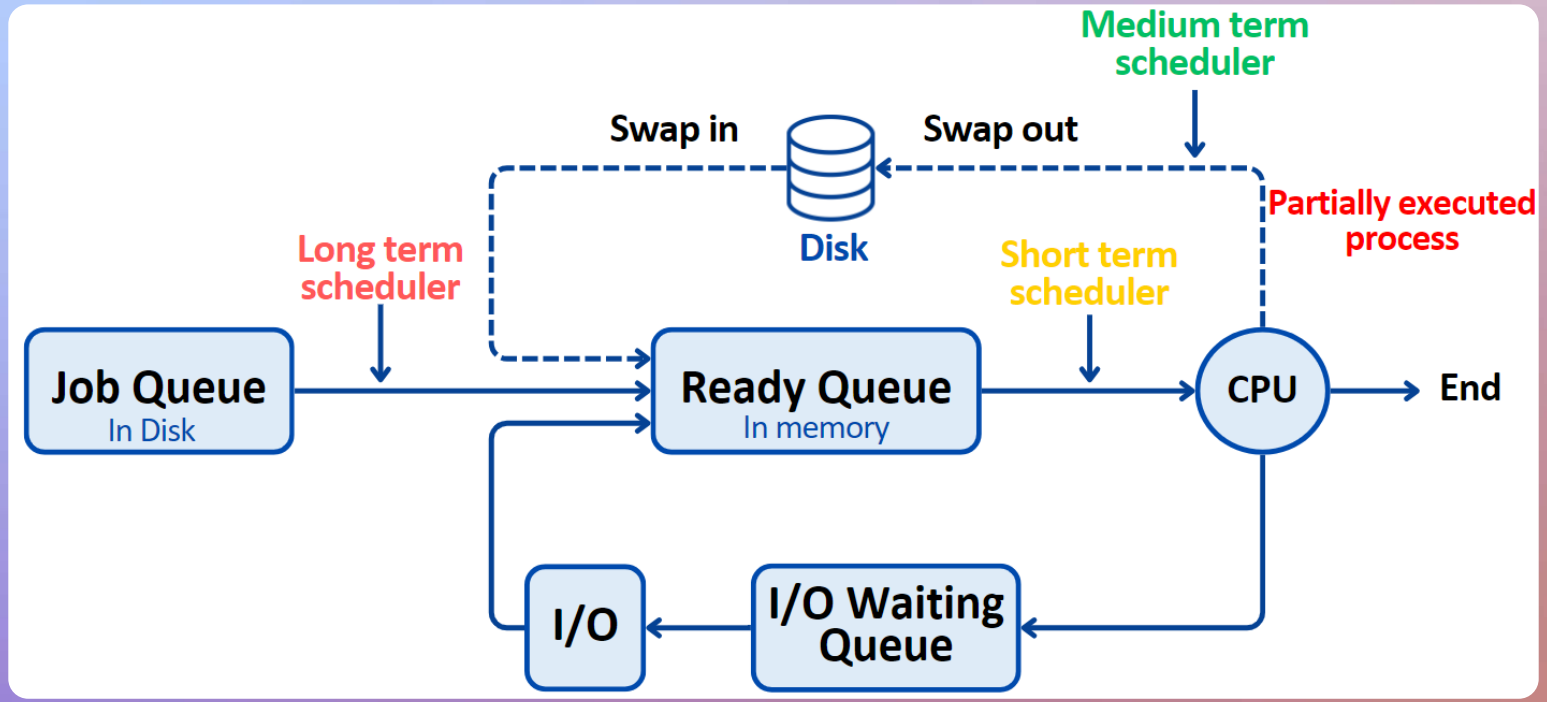
Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue.
  - Long-term scheduler is invoked infrequently (seconds, minutes) → (may be slow)
  - The long-term scheduler controls the degree of multiprogramming
  - The degree of multiprogramming refers to the number of programs that are loaded into the memory and are ready to execute at any given time



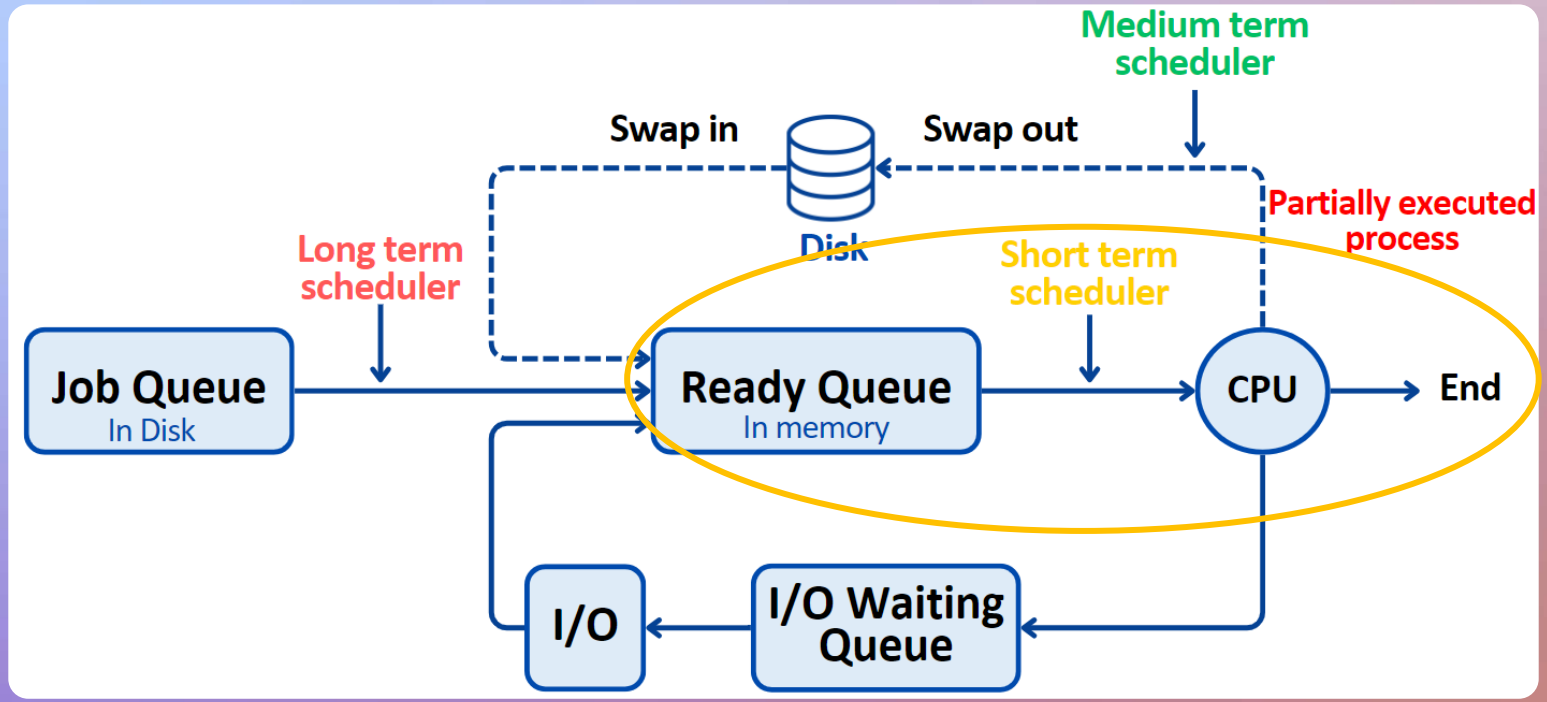
Schedulers

- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU.
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) → (must be fast)



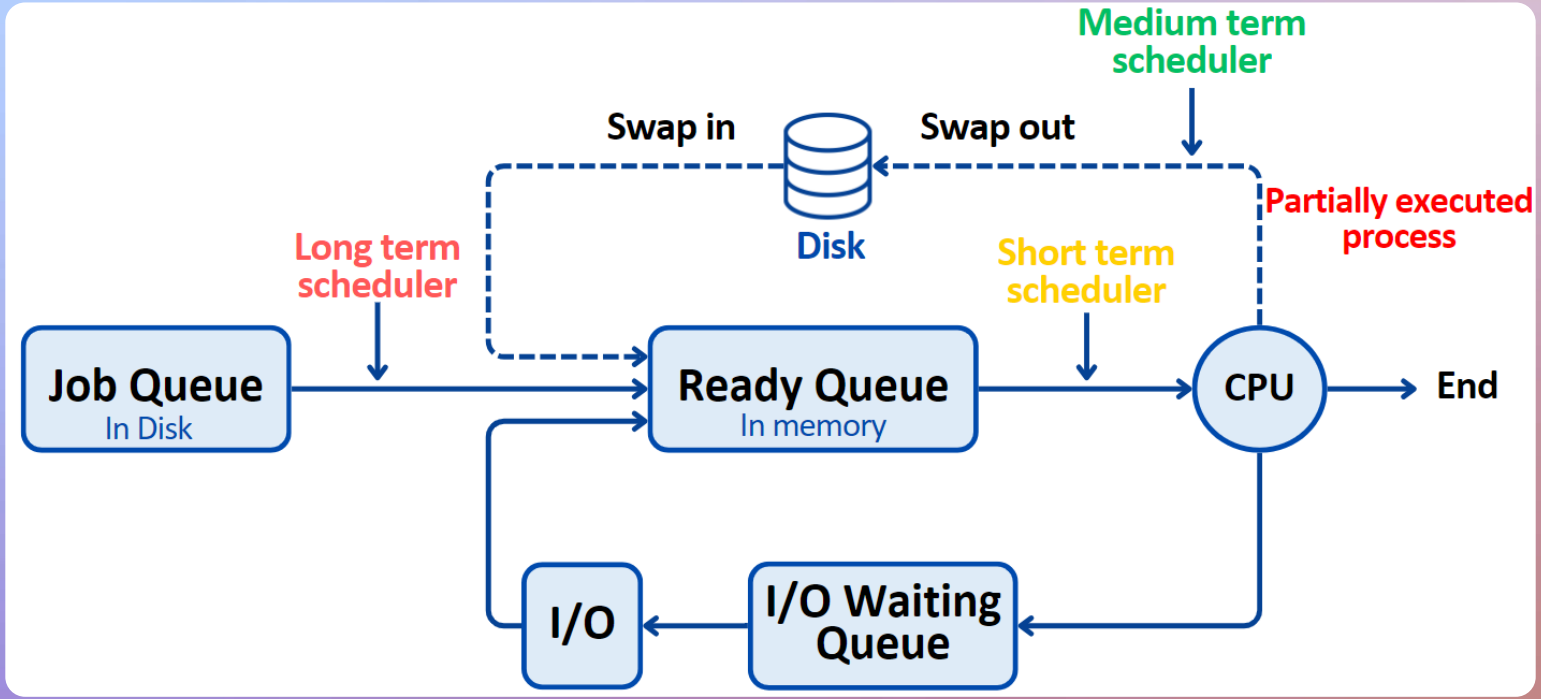
Schedulers

- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU.
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) → (must be fast)



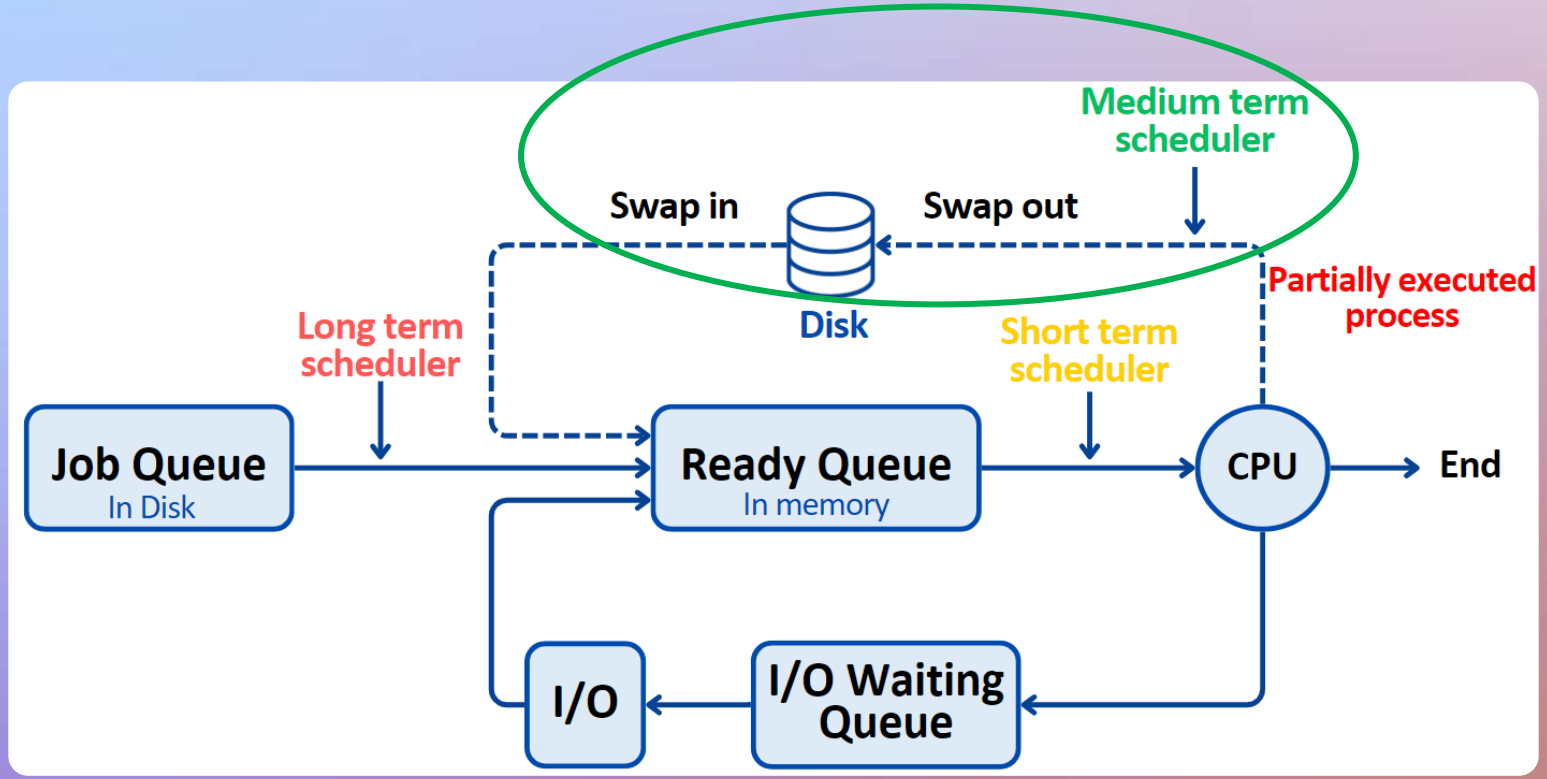
Schedulers

- **Medium-term scheduler** Temporarily removes a partially executed process (due to I/O request or by a system call) from main memory to free-up some space.
  - Can be later reintroduced in the ready queue to resume execution



Schedulers

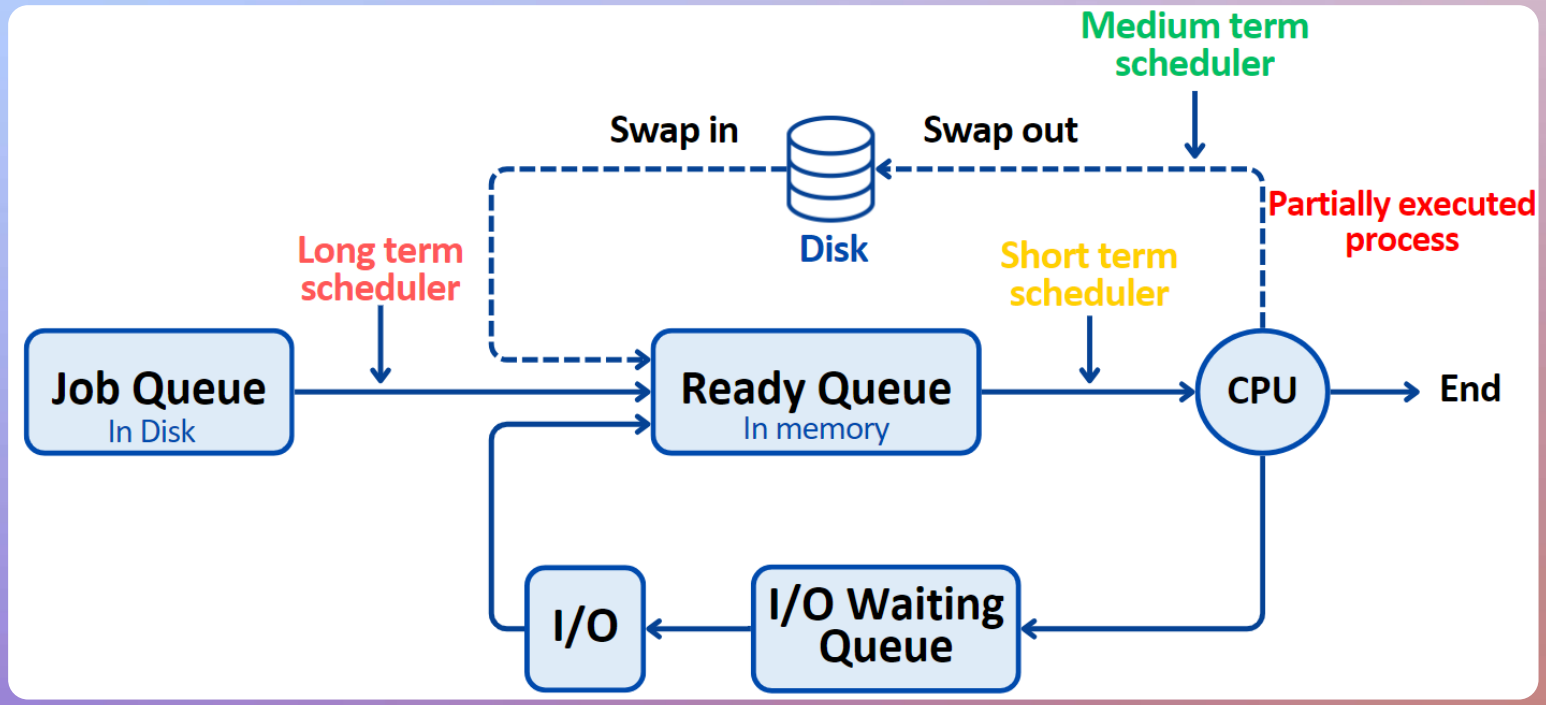
- **Medium-term scheduler** Temporarily removes a partially executed process (due to I/O request or by a system call) from main memory to free-up some space.
  - Can be later reintroduced in the ready queue to resume execution



Schedulers

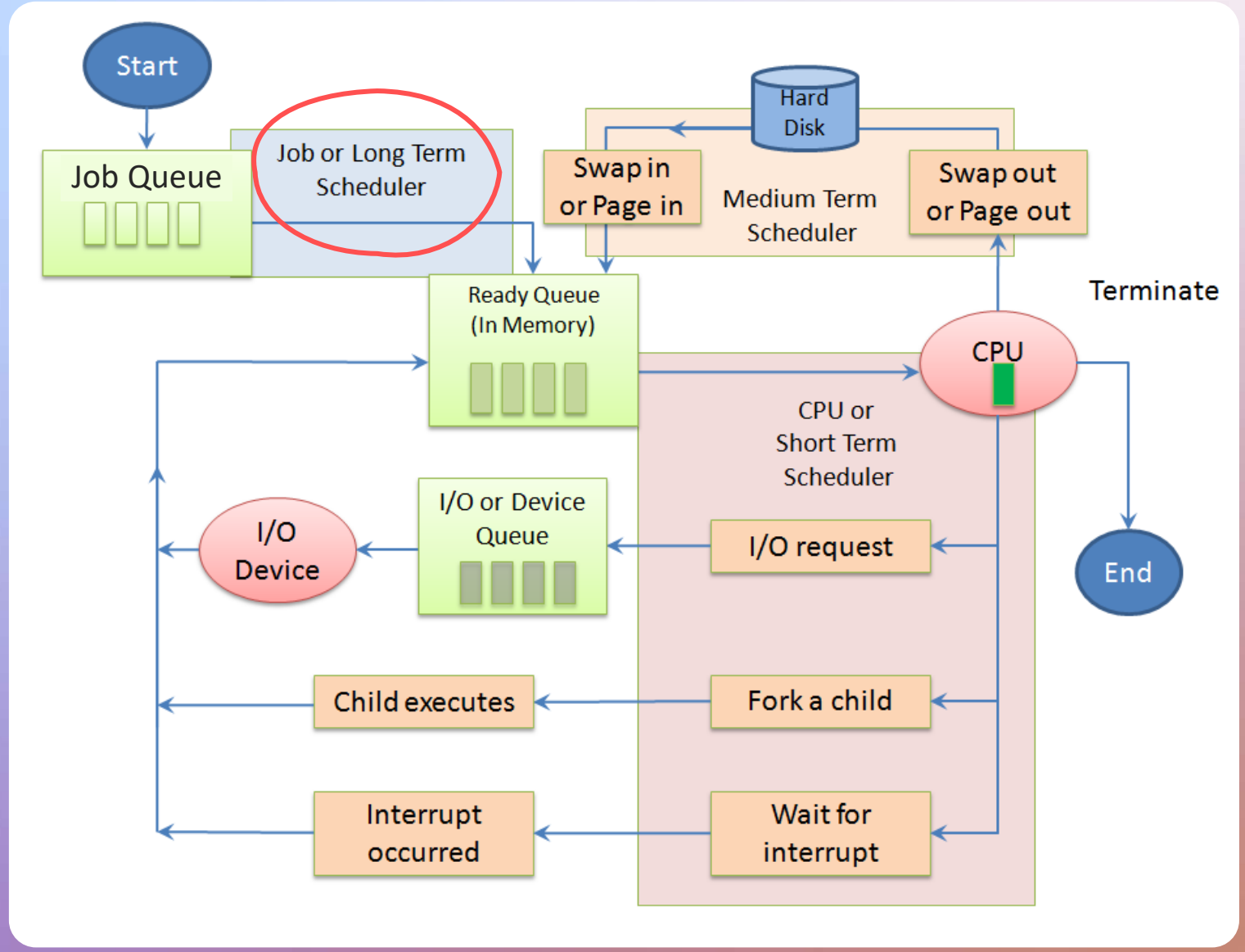
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts

Long-term scheduler strives for good process mix



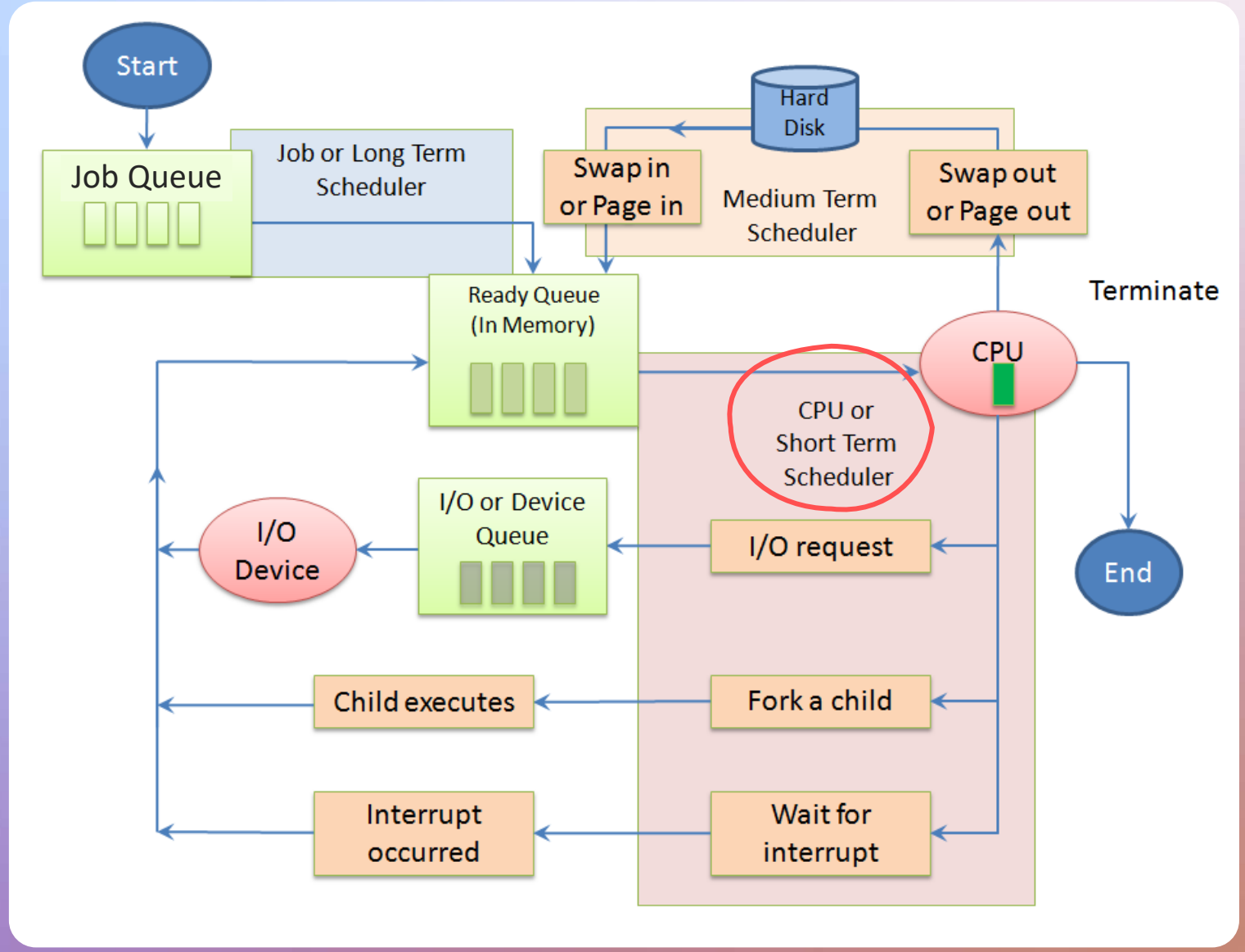
Schedulers

Scheduler and Process Scheduling Queues



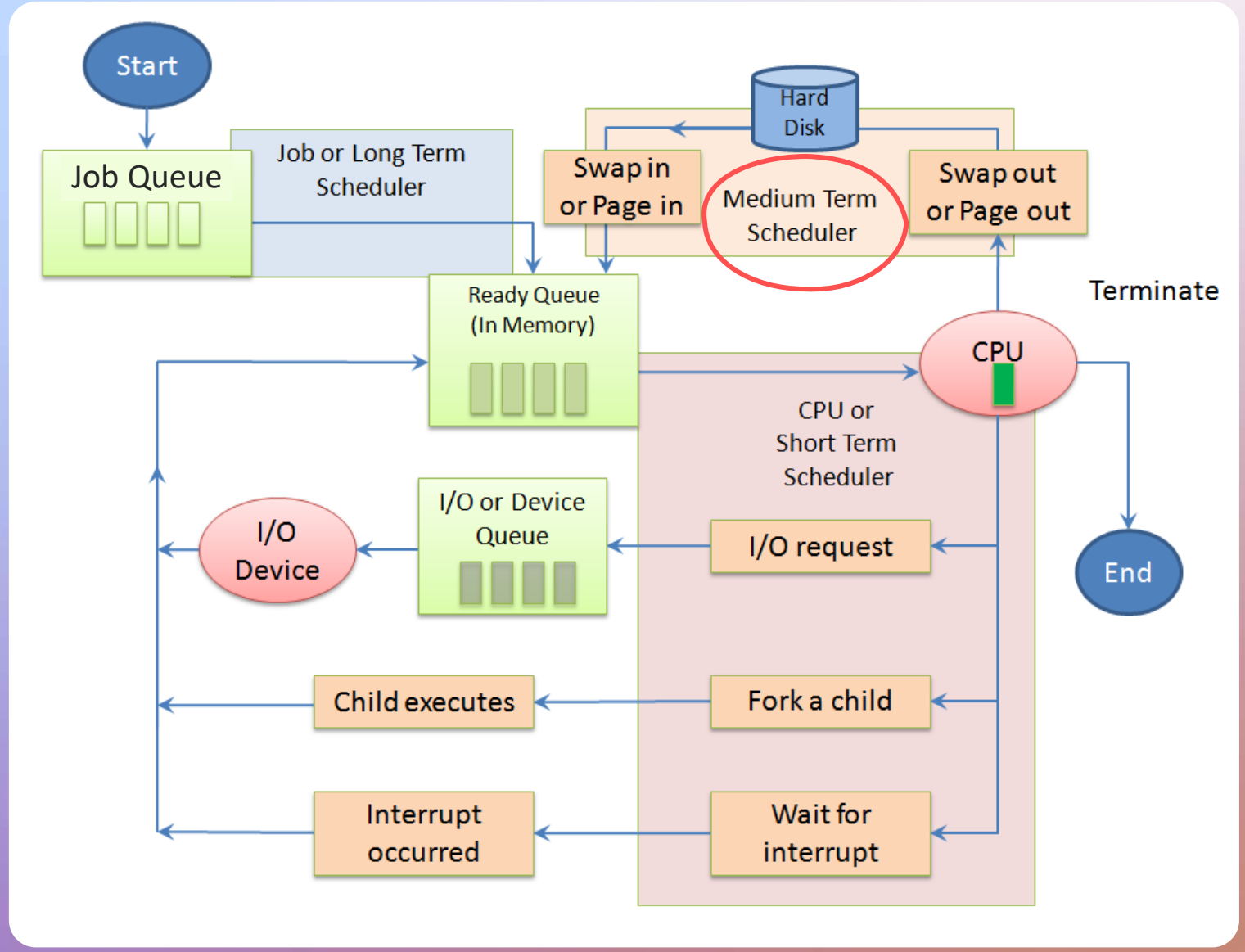
Schedulers

Scheduler and Process Scheduling Queues



Schedulers

Scheduler and Process Scheduling Queues




Questions are Welcomed