

# Fundamental concepts

## Module 1

---

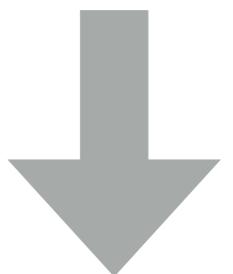
**Operating systems 2019**

**1DT044, 1DT096 and 1DT003**

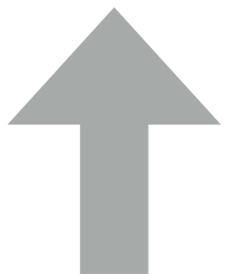
# Learning and understanding



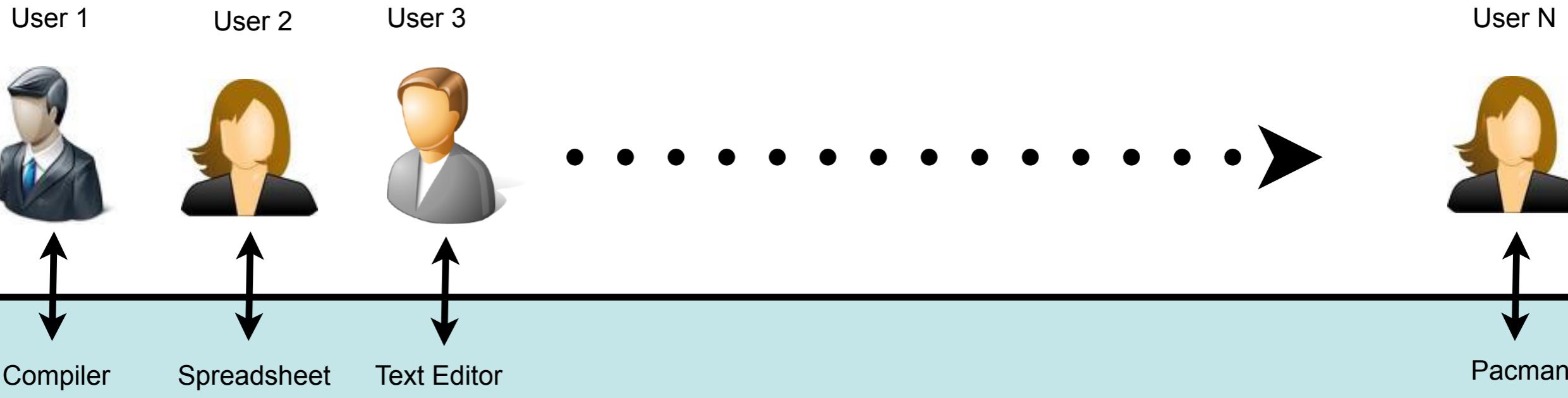
Top down



Bottom up



**One top down view  
from the previous  
lecture**



# Operating System

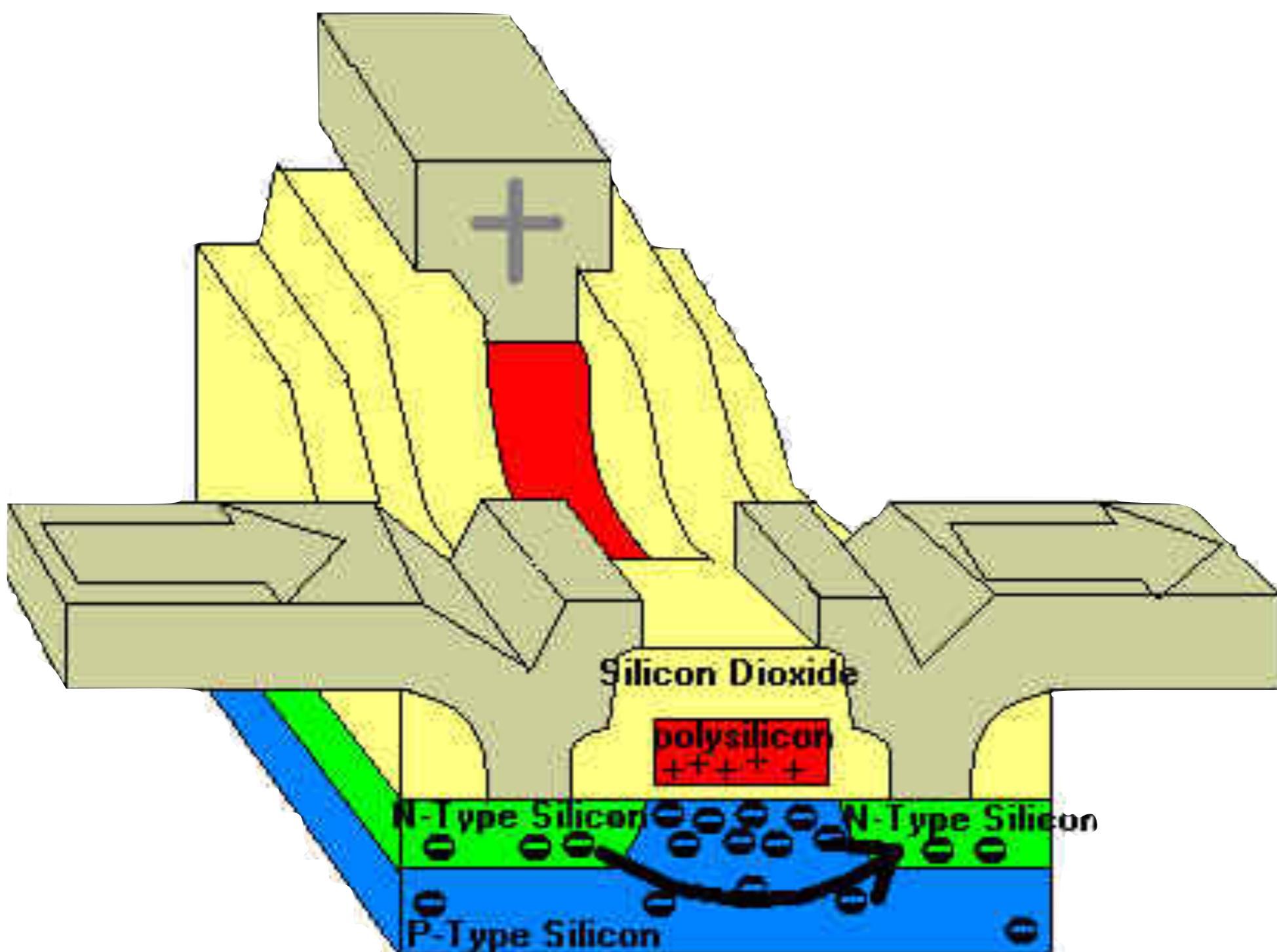
Controls the hardware and coordinates its use among the various application programs for the various users.

## Computer Hardware

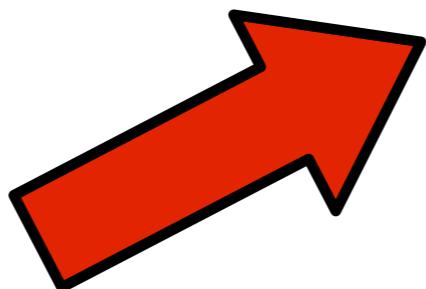
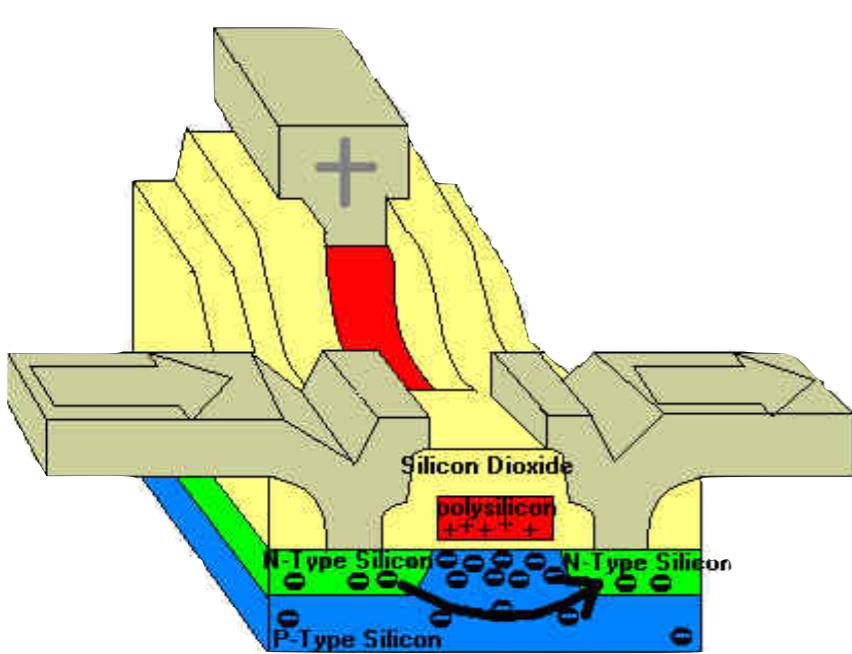


**Now, let's start  
bottom up**

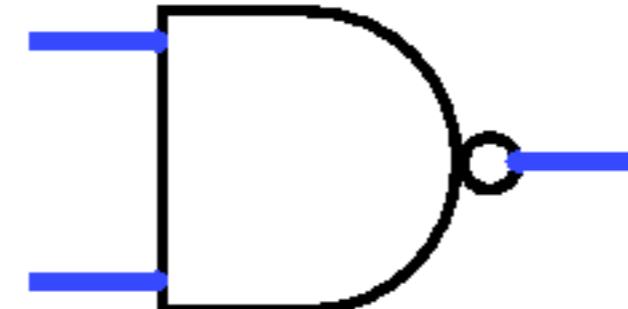
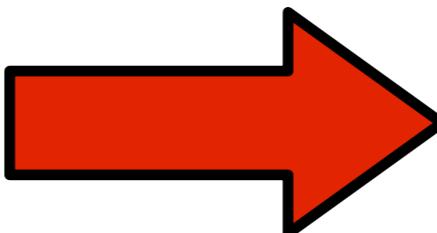
# The transistor



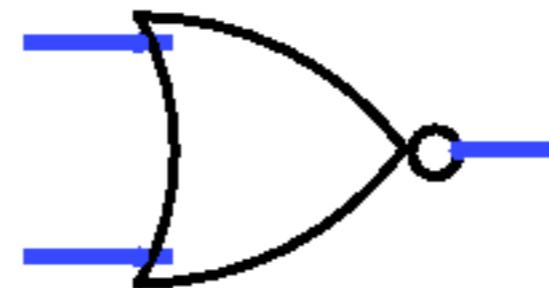
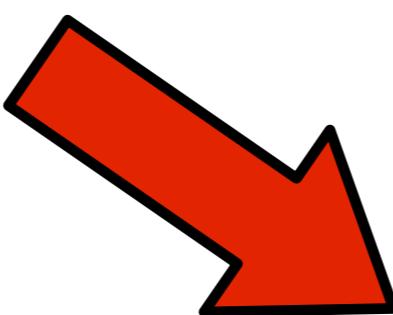
# From transistors to logical gates



NOT

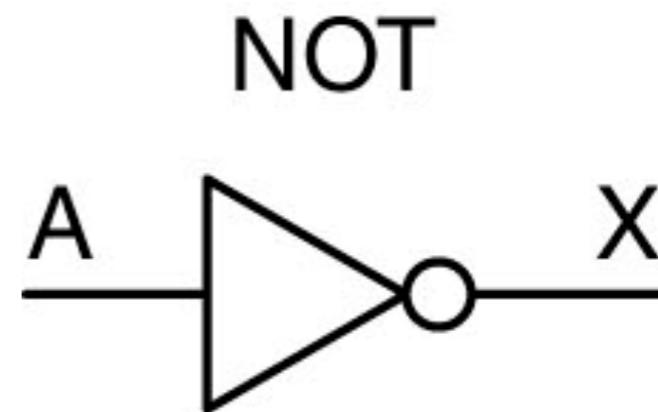
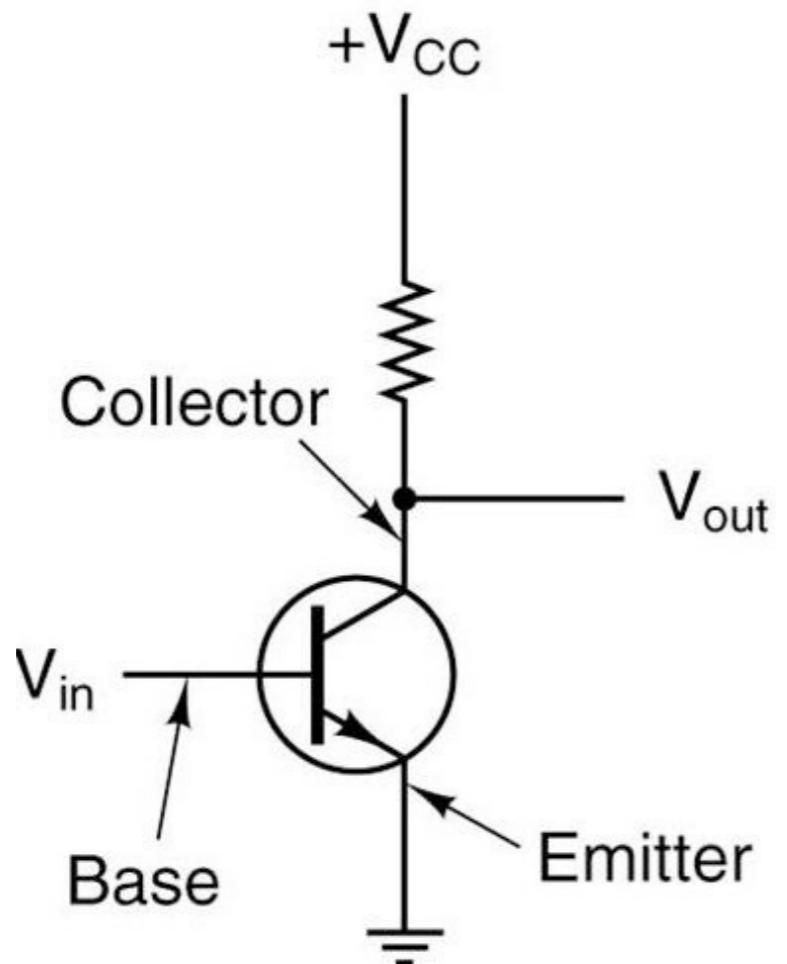


NAND



NOR

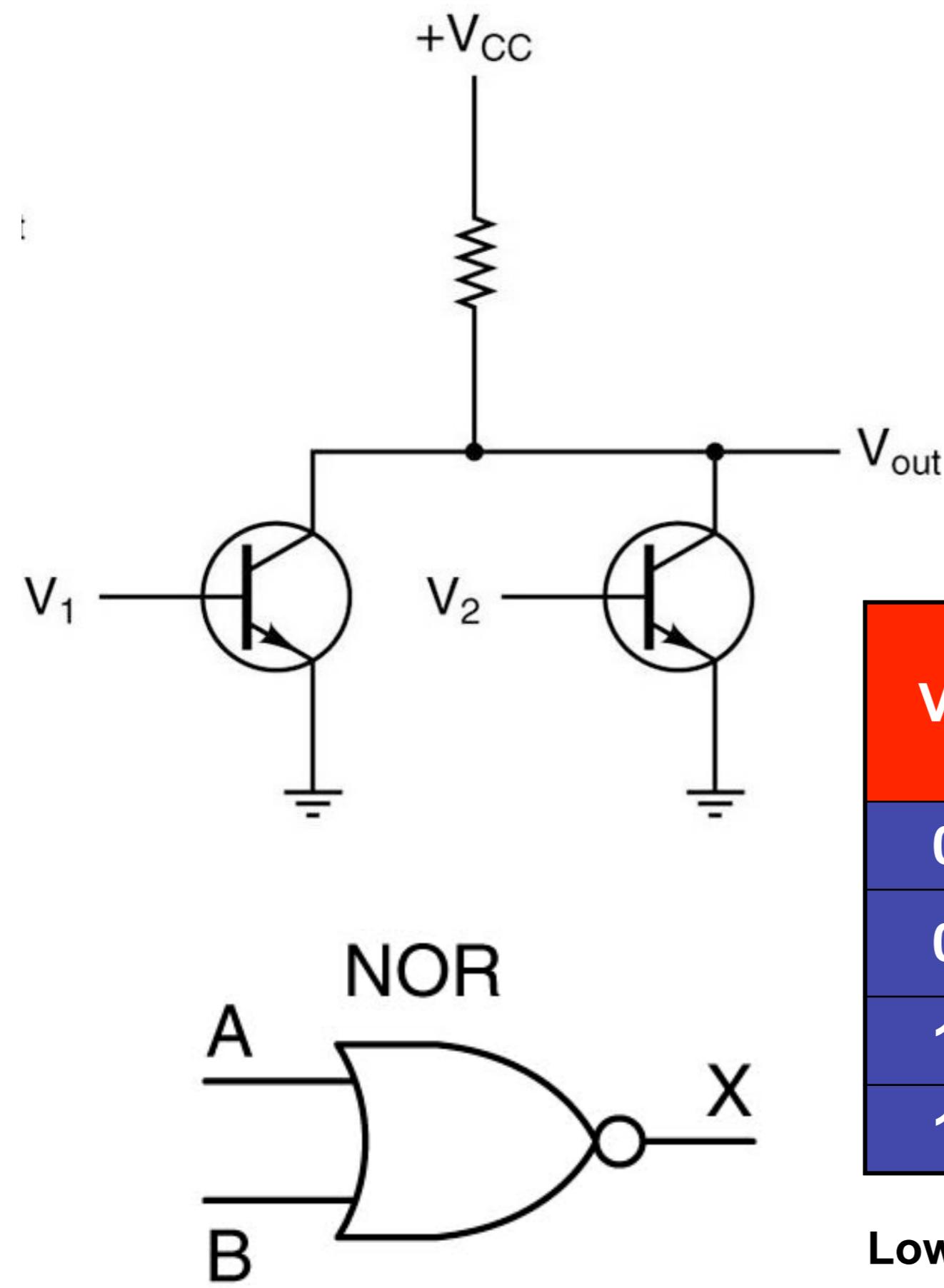
# An inverter - logical NOT



$V_{in}$	$V_{out}$
low	high
high	low

$A$	$X$
0	1
1	0

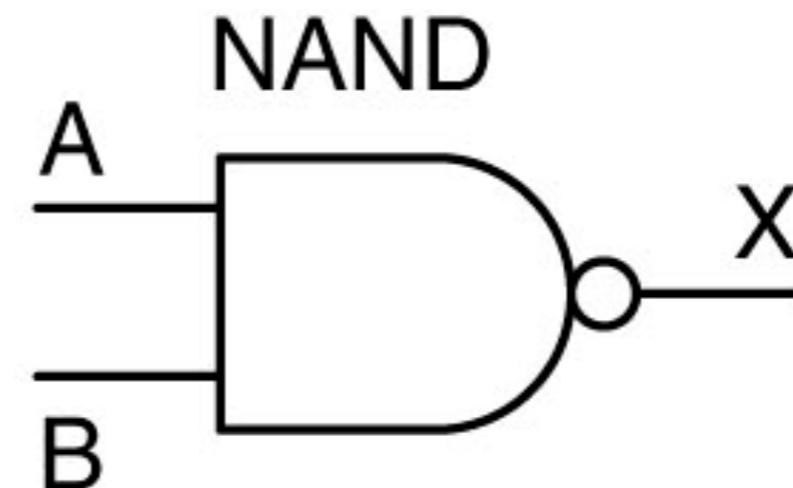
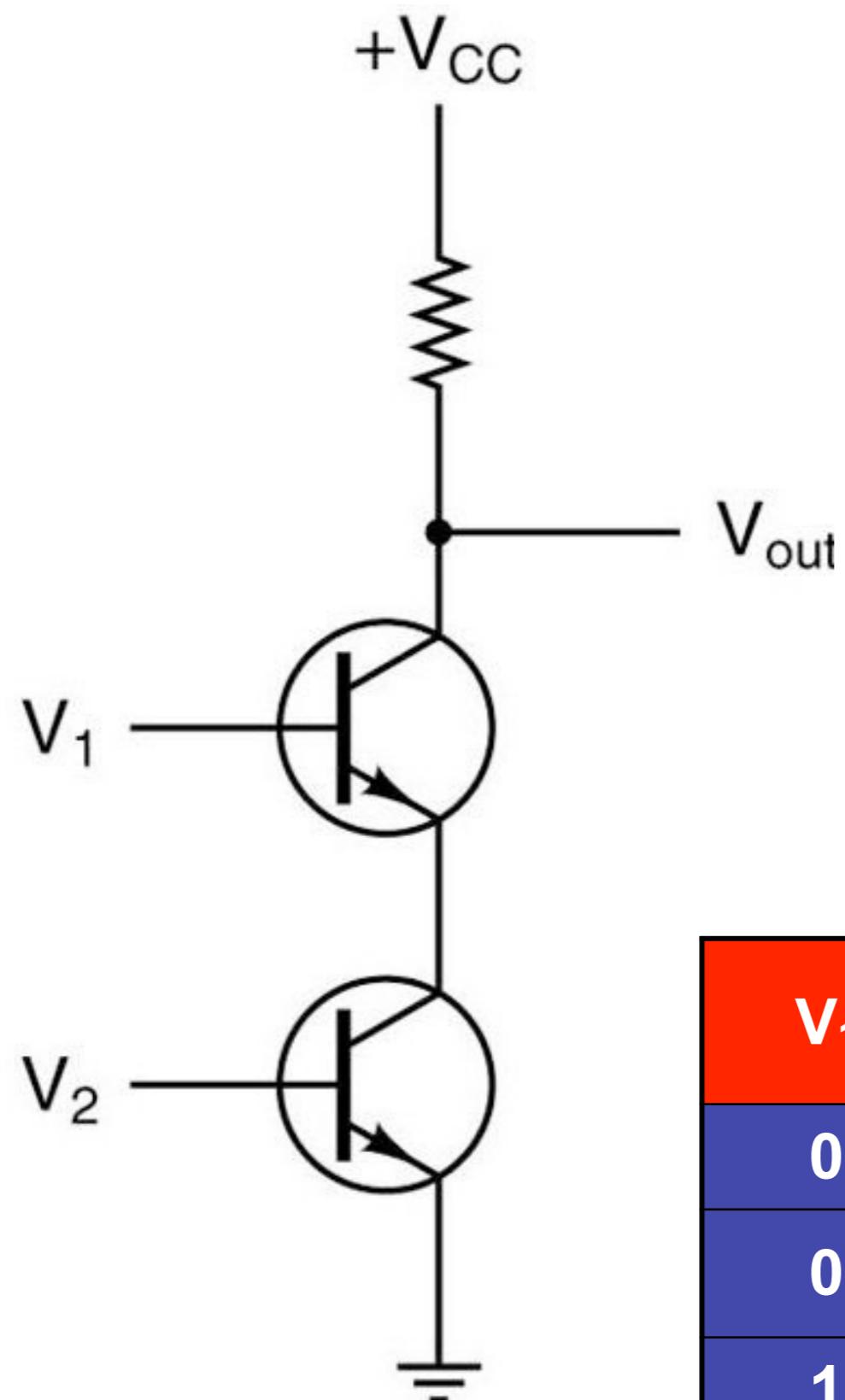
# logical NOR



$V_1$	$V_2$	$V_{out}$	$A \text{ or } B$	$a \text{ nor } b = \text{not}(A \text{ or } B)$
0	0	1	0	1
0	1	0	1	0
1	0	0	1	0
1	1	0	1	0

Low  $\rightarrow 0$    High  $\rightarrow 1$

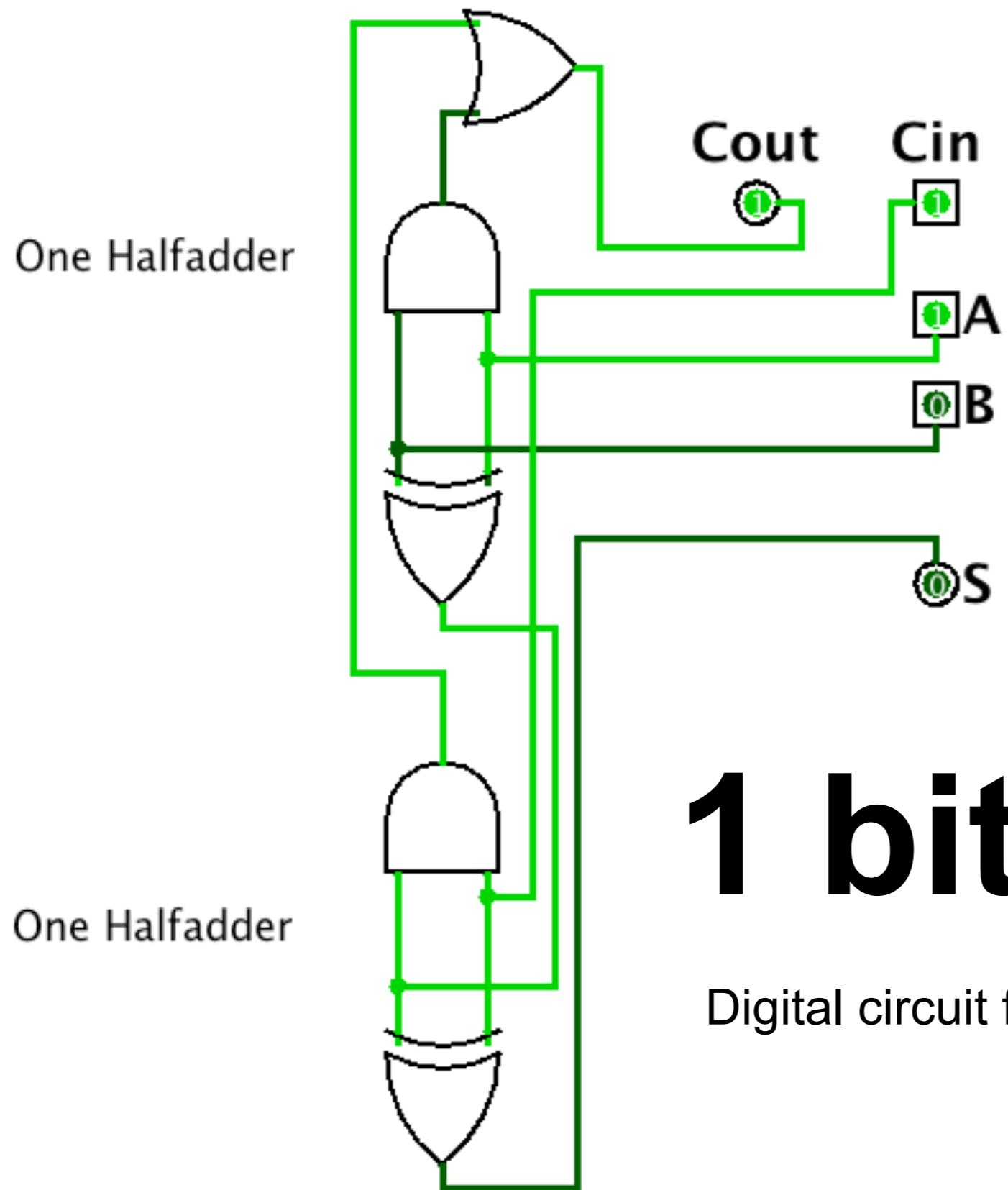
# logical NAND



$V_1$	$V_2$	$V_{out}$	A and B	$A \text{ nand } B = \text{not}(A \text{ and } B)$
0	0	1	0	1
0	1	1	0	1
1	0	1	0	1
1	1	0	1	0

Low → 0   High → 1

One OR-gate to combine the two Halfadders

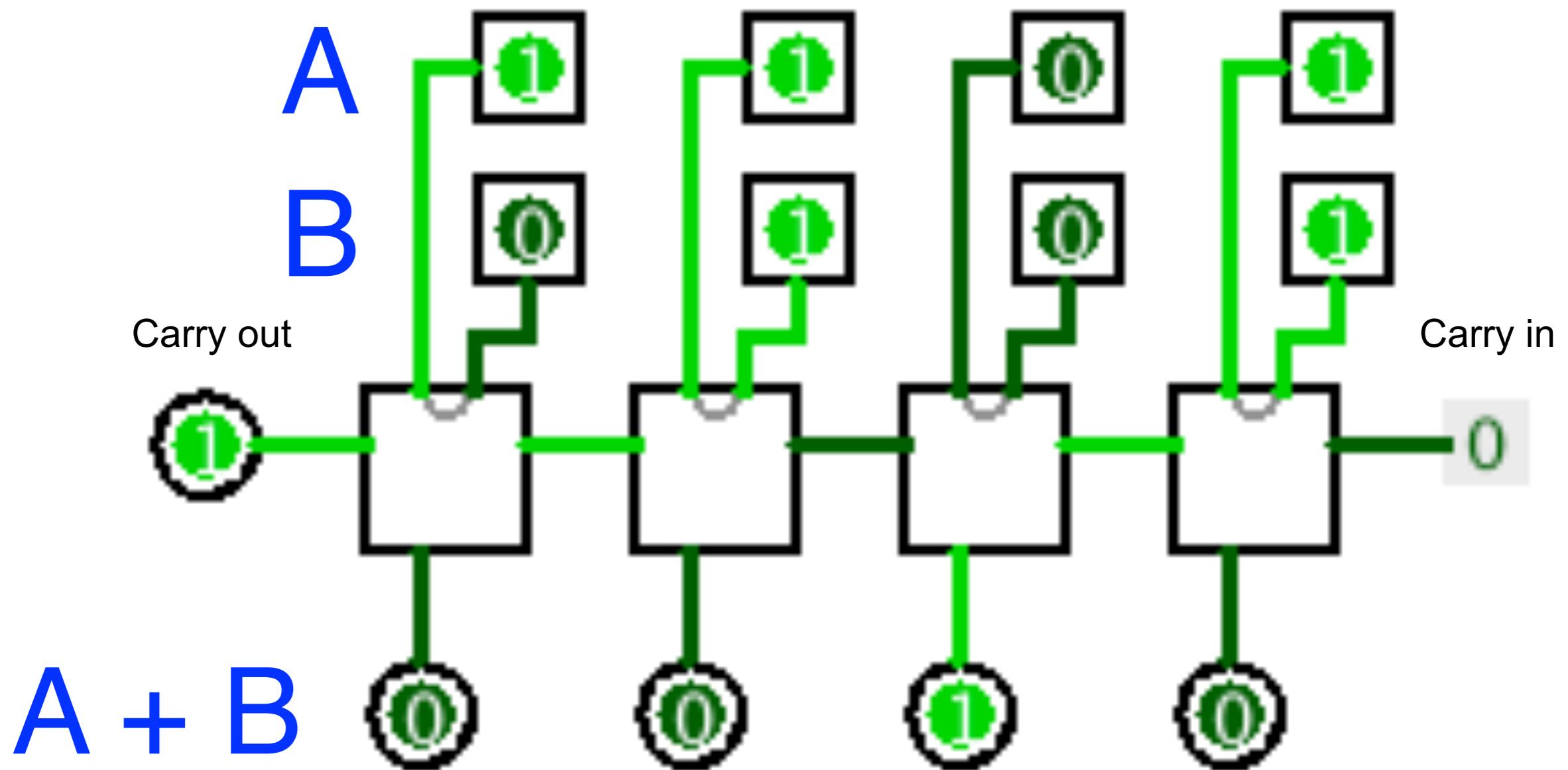


# 1 bit full adder

Digital circuit for adding two 1 bit numbers A and B

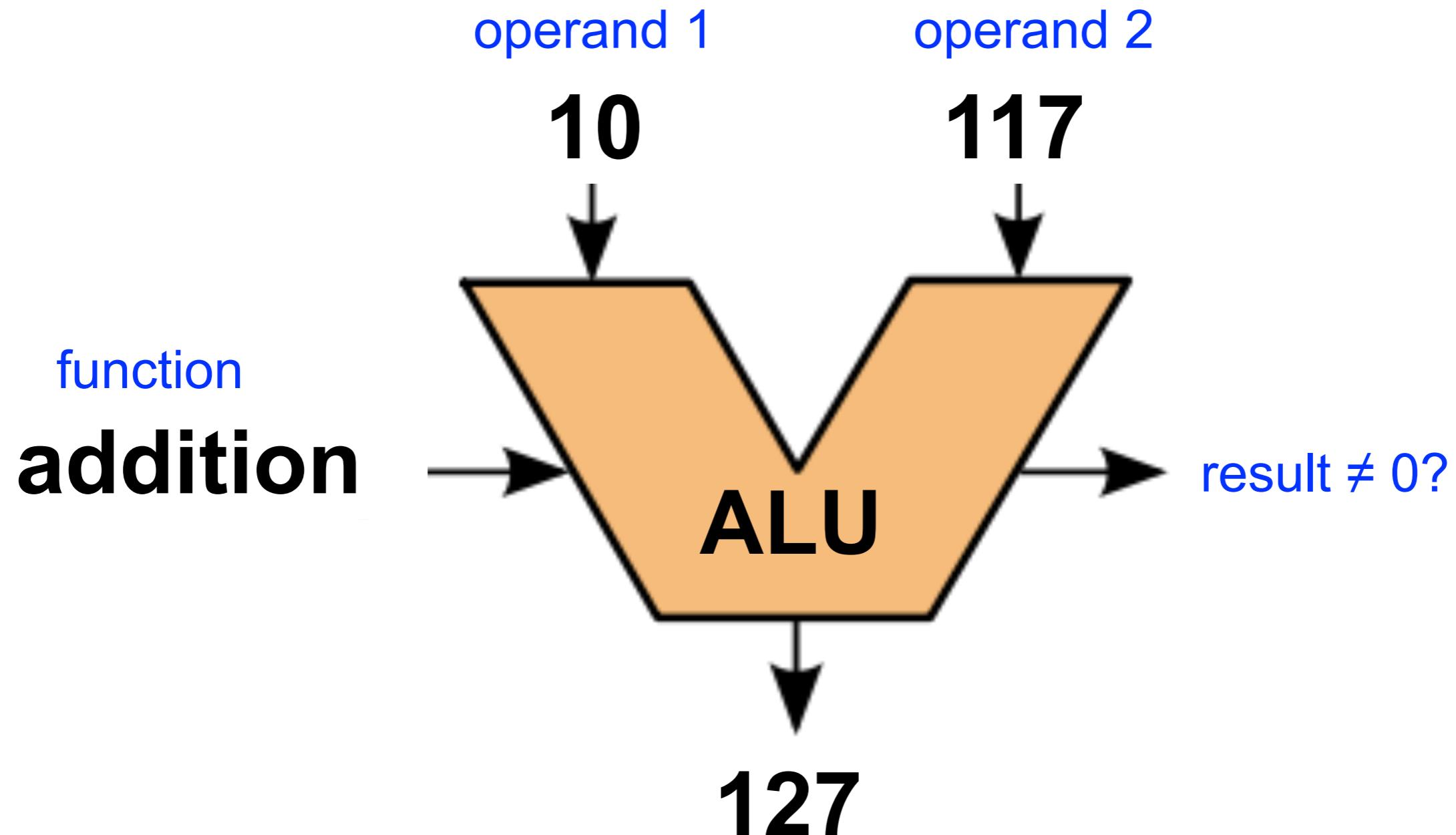
# Ripple adder

Four 1 bit full adders used to add two four bit numbers



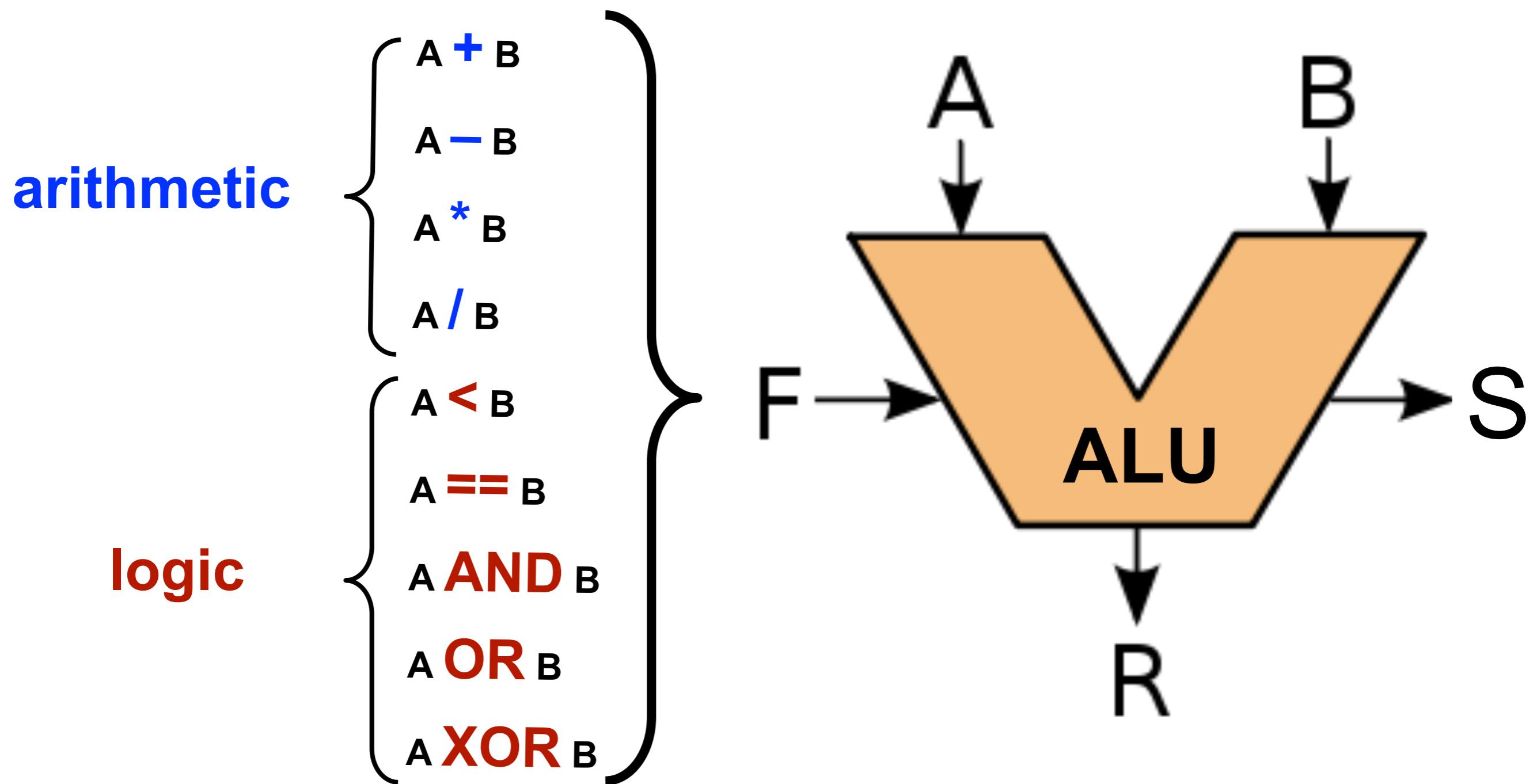
# Arithmetic Logic Unit (ALU)

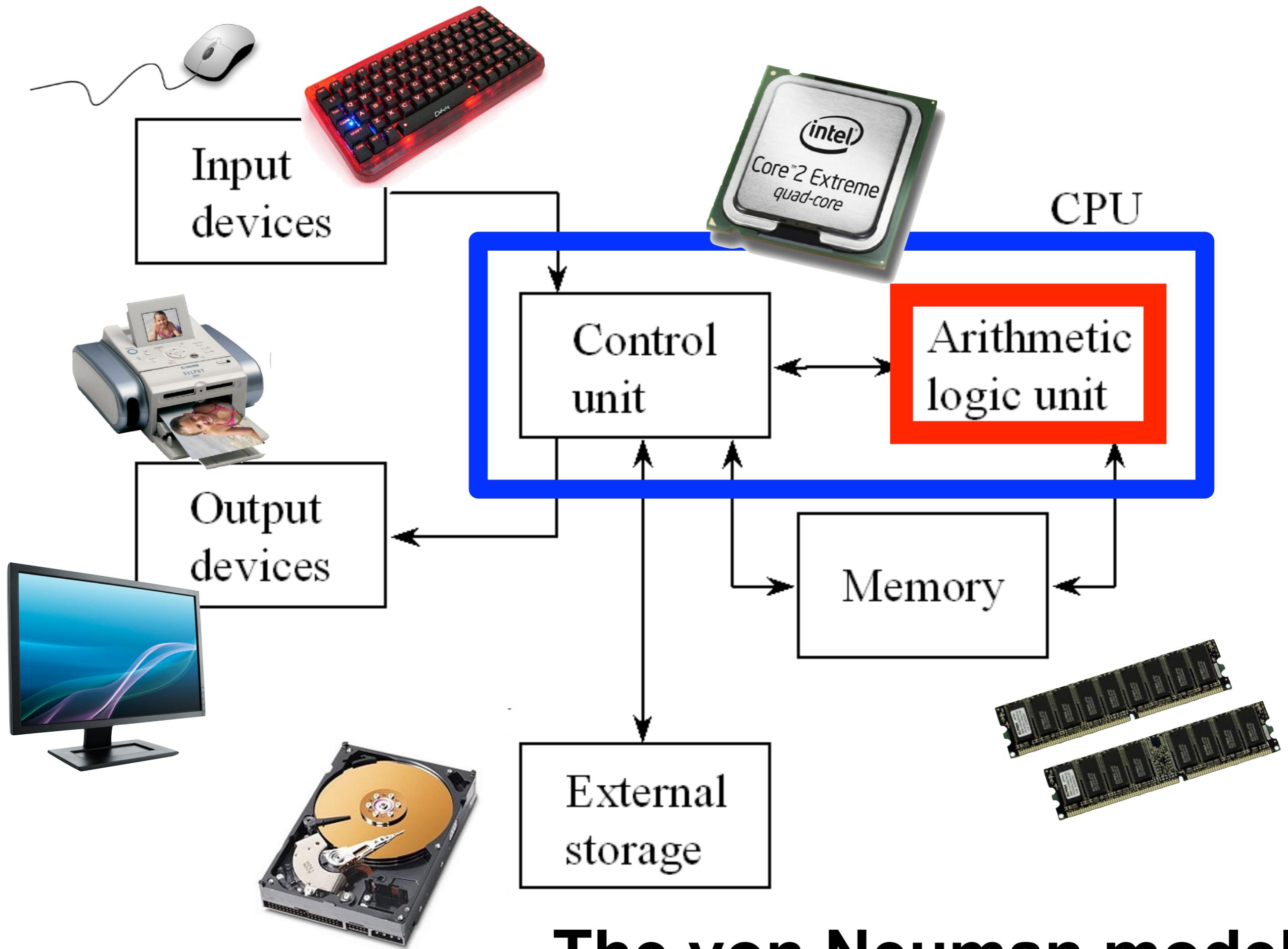
An adder is the first part in constructing a working ALU.



# Arithmetic Logic Unit (ALU)

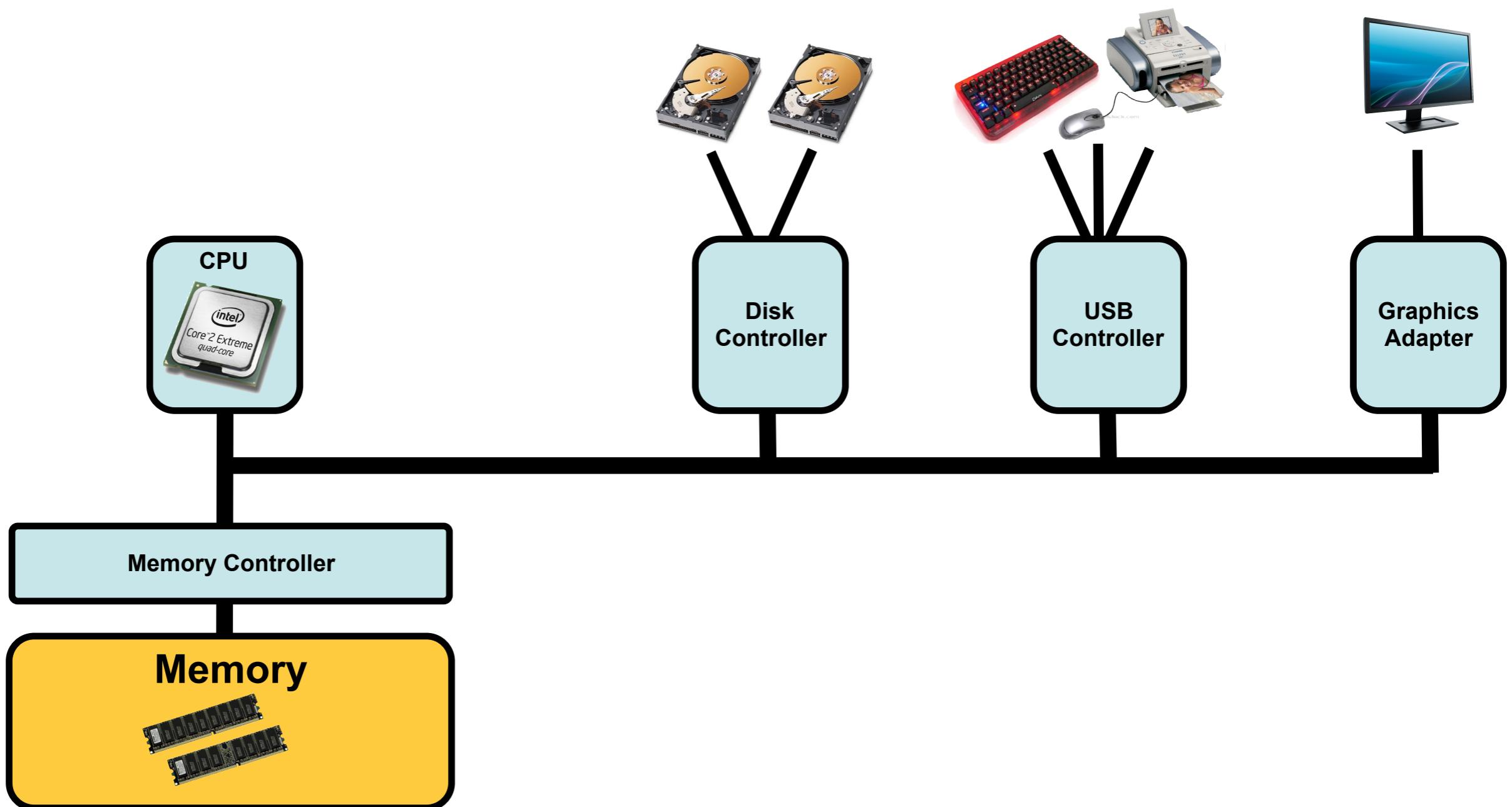
Similar to the addition circuit, circuits can be constructed for other ALU operations.





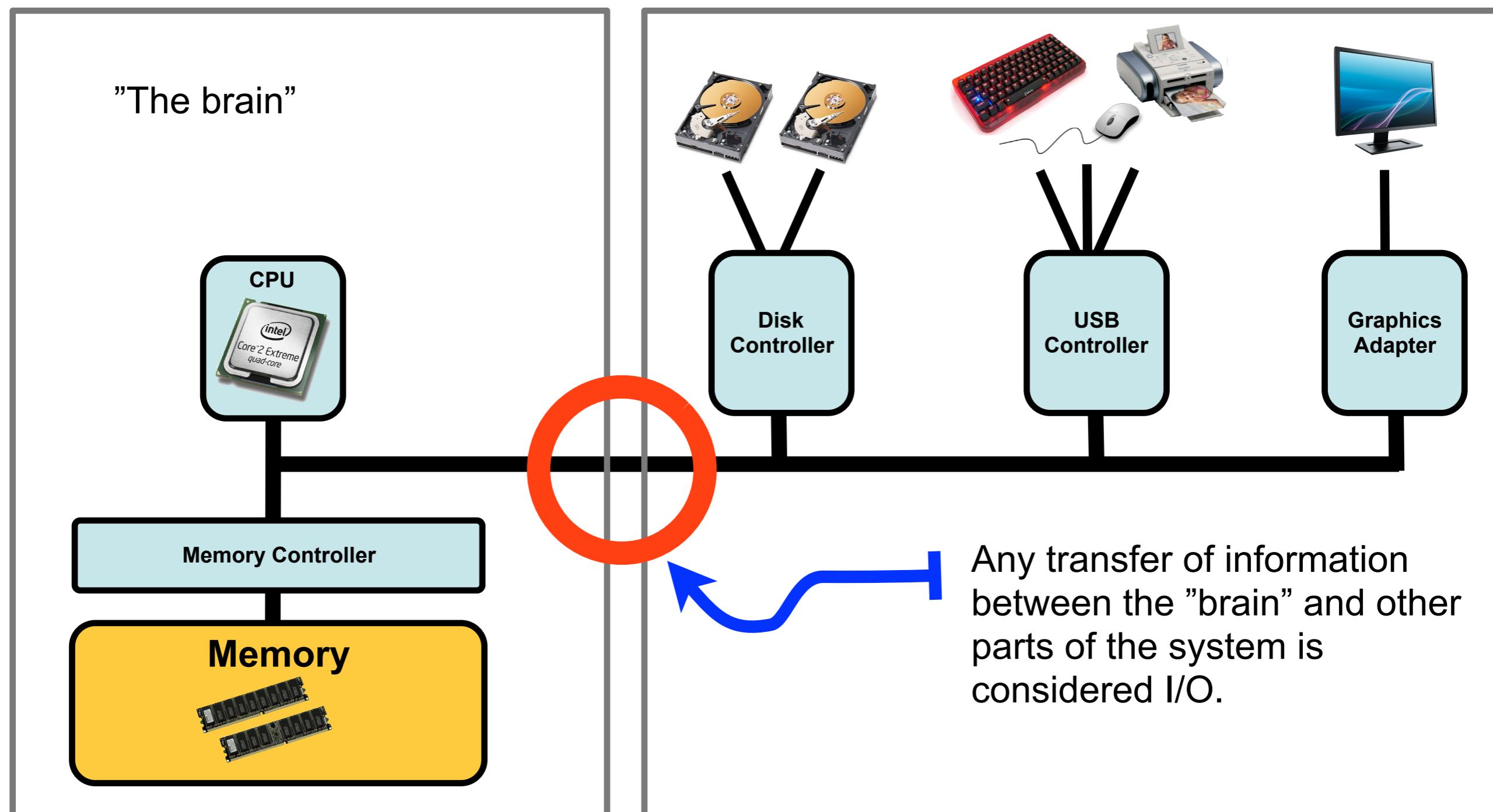
# The von Neuman modell

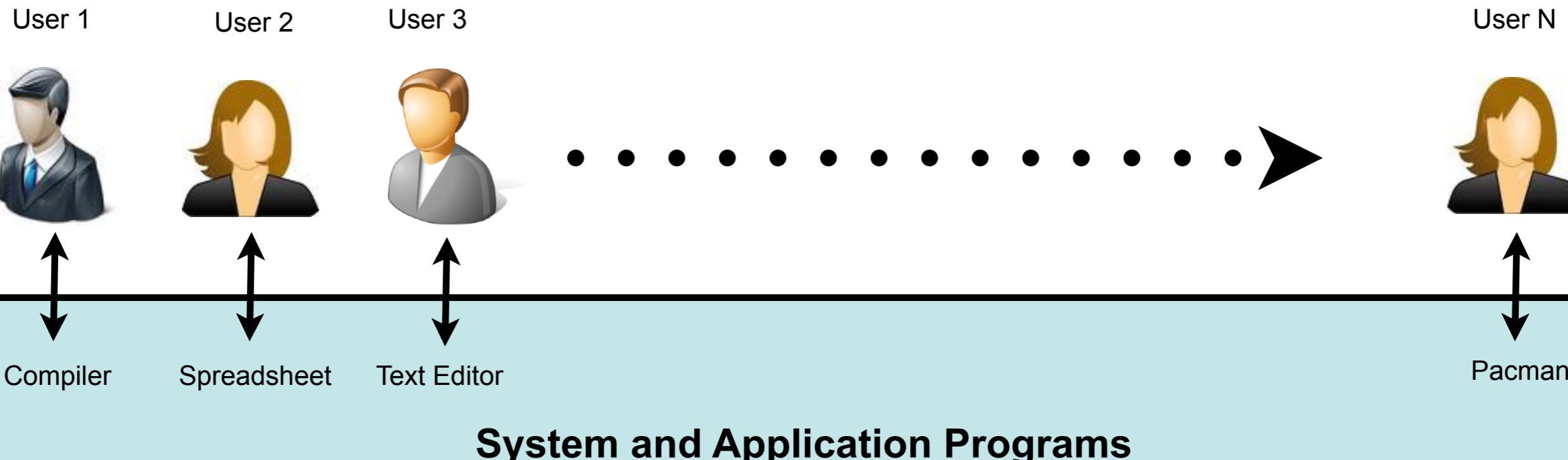
# A typical modern computer system



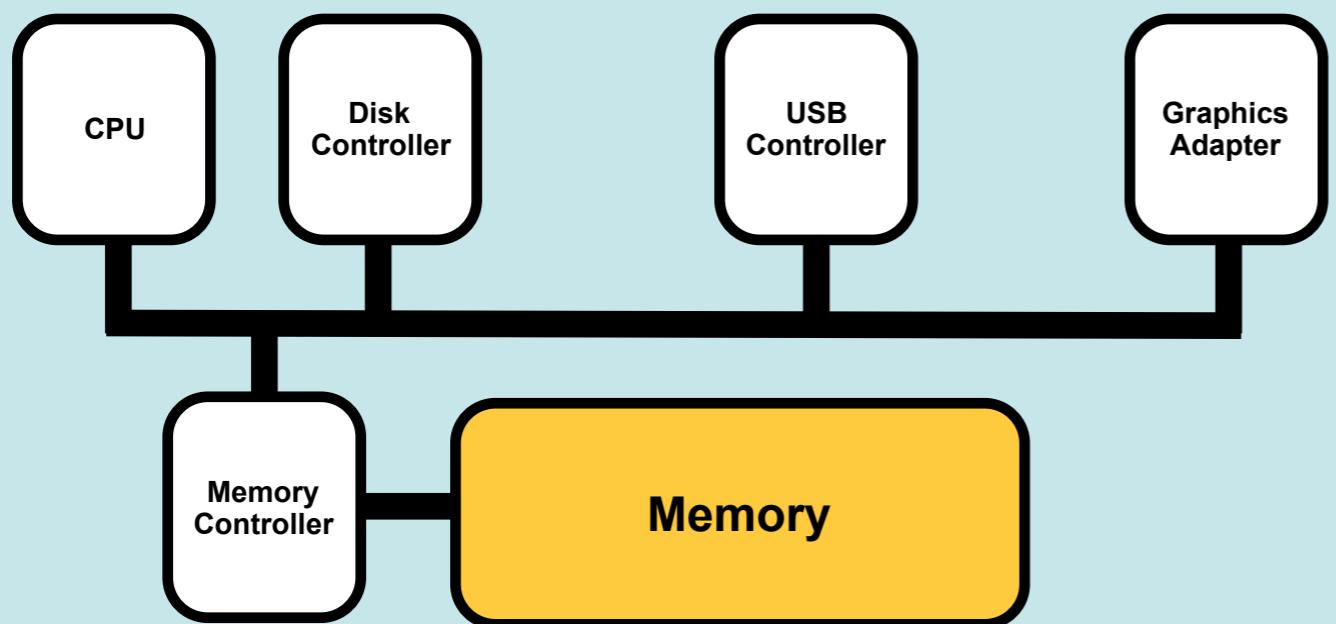
# Input and Output (I/O)

In computer architecture, the combination of the CPU and main memory (i.e. memory that the CPU can read and write to directly, with individual instructions) is considered the "brain" of a computer

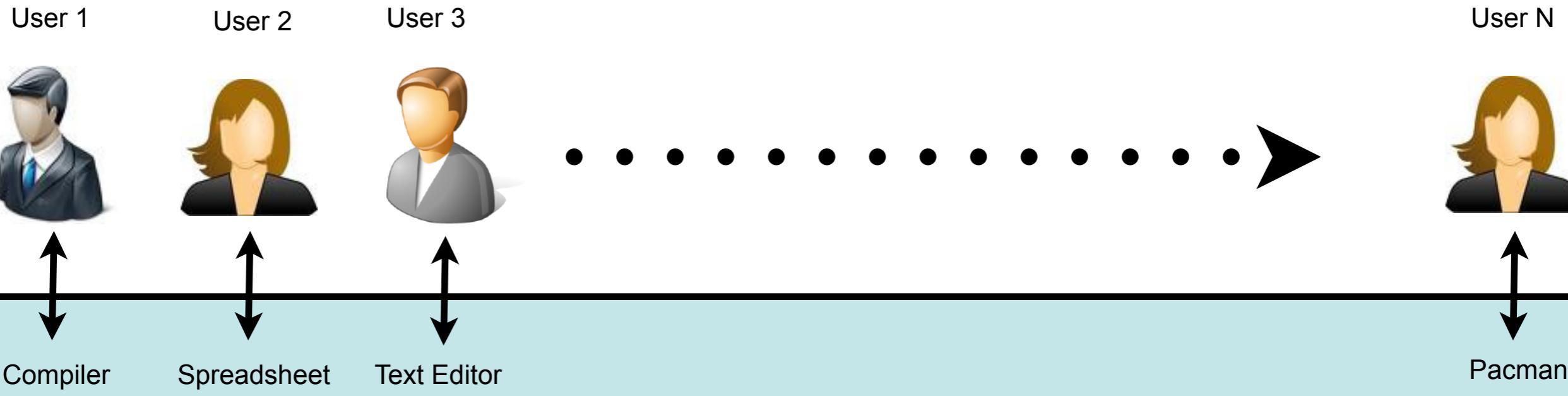




# Operating System



**Computer  
Hardware**



# Operating System

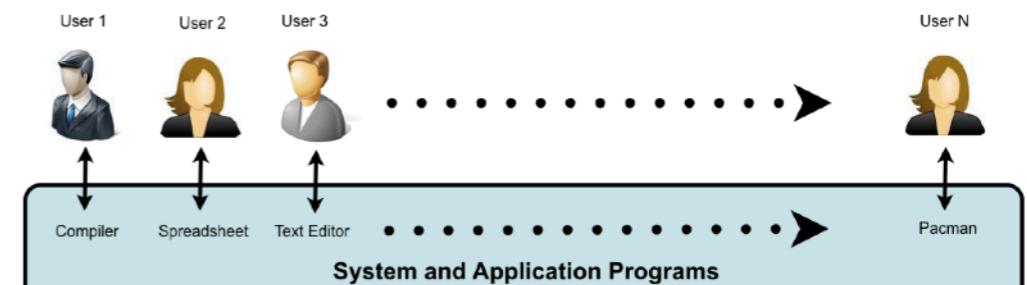
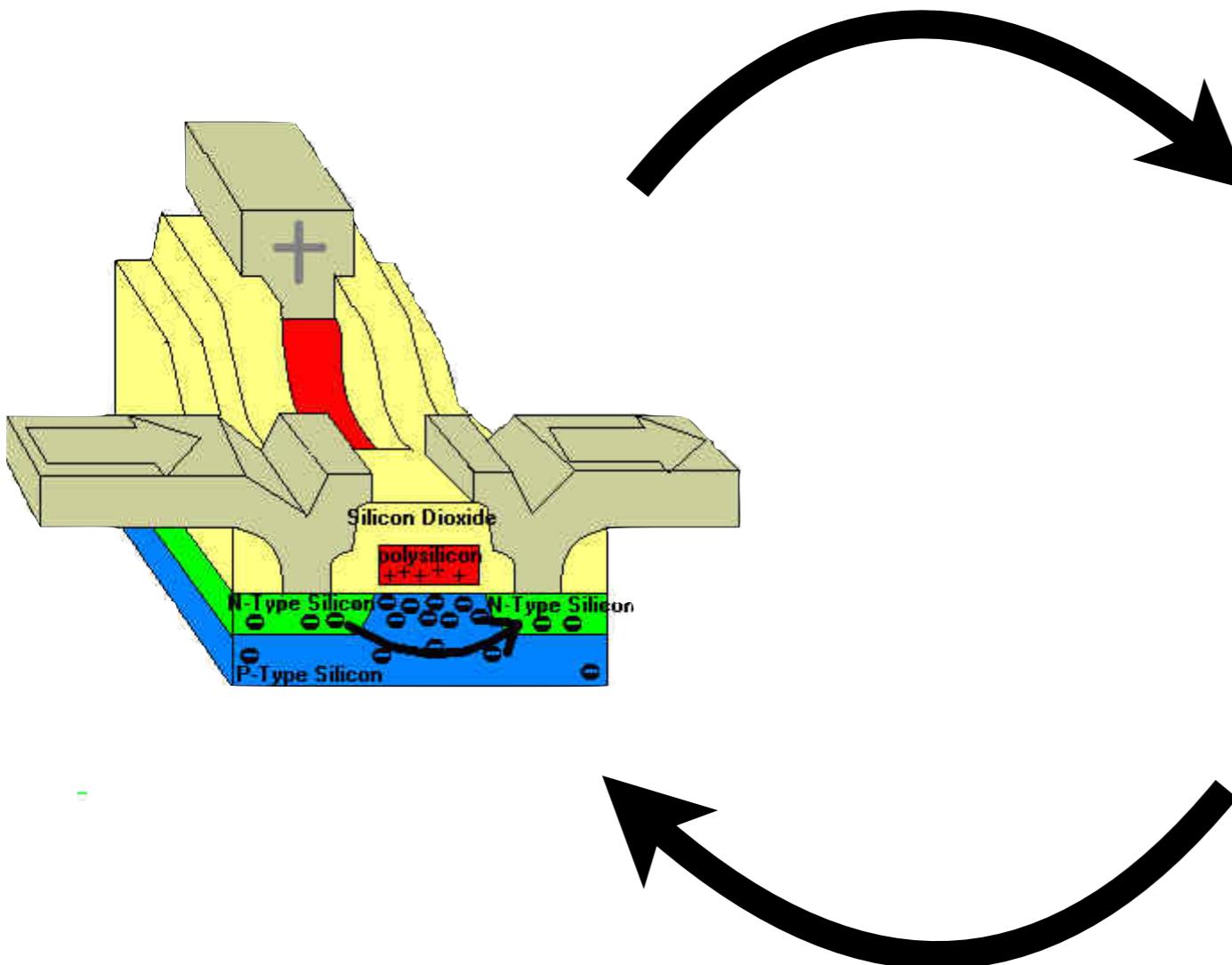
Controls the hardware and coordinates its use among the various application programs for the various users.

## Computer Hardware

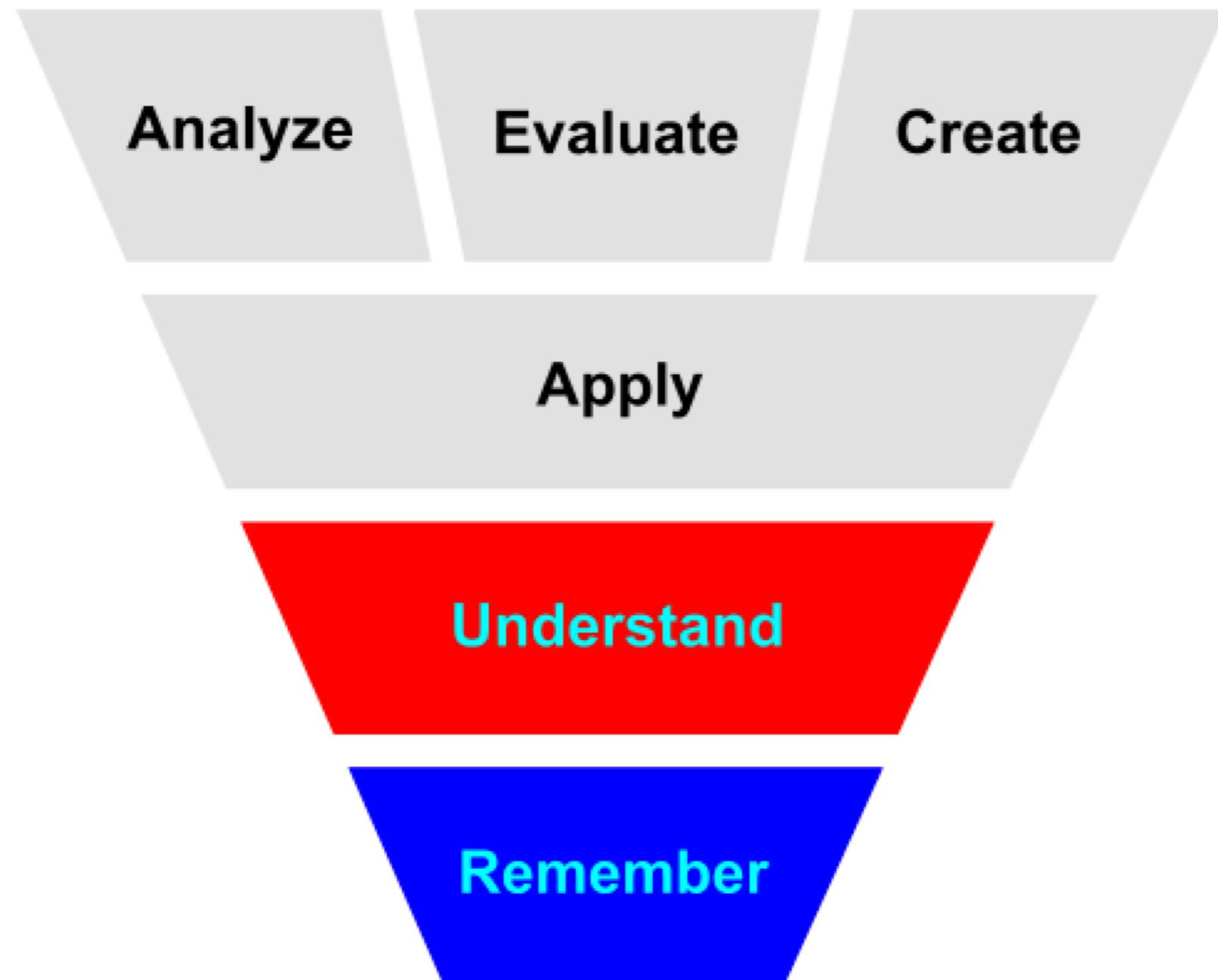


# Bottom up

# Top down



# The cognitive domain



# **Important definitions**

# CPU

The central processing unit (CPU) is the electronic circuitry within a computer that carries out the instructions of a computer program by performing the basic arithmetic, logical, control and input/output (I/O) operations specified by the instructions.

## Register

A processor register is a quickly accessible location available to a computer's central processing unit (CPU). Registers usually consist of a small amount of fast storage. A CPU only has a small number of registers.

## Memory

Memory refers to the computer hardware integrated circuits that store information for immediate use in a computer; it is synonymous with the term "primary storage". The memory is much slower than the CPU register but much larger in size.

### Sources

[https://en.wikipedia.org/wiki/Central\\_processing\\_unit](https://en.wikipedia.org/wiki/Central_processing_unit)

[https://en.wikipedia.org/wiki/Processor\\_register](https://en.wikipedia.org/wiki/Processor_register)

[https://en.wikipedia.org/wiki/Computer\\_memory](https://en.wikipedia.org/wiki/Computer_memory)

2018-01-16

2018-01-16

2018-01-16

# CPU context

At any point in time, the values of all the registers in the CPU defines the CPU context. Sometimes CPU state is used instead of CPU context.

# **Program**

A set of instructions which is in human readable format. A passive entity stored on secondary storage.

# **Executable**

A compiled form of a program including machine instructions and static data that a computer can load and execute. A passive entity stored on secondary storage.

# **Process**

A program loaded into memory and executing or waiting. A process typically executes for only a short time before it either finishes or needs to perform I/O (waiting). A process is an active entity and needs resources such as CPU time, memory etc to execute.

# Kernel

The kernel is a computer program that is the core of a computer's operating system, with complete control over everything in the system.

## **System and Application Programs**

# **Operating System**

Controls the hardware and coordinates its use among the various application programs for the various users.

## **Computer Hardware**

# System and Application Programs

## Operating System

Controls the hardware and coordinates its use among the various application programs for the various users.

### Bootstrap program



### Kernel



## Computer Hardware

# System and Application Programs

## Operating System

Controls the hardware and coordinates its use among the various application programs for the various users.

### Bootstrap program

Kept on chip (ROM or EEPROM), aka **firmware**.

Small program executed on power up or reboot.

**Initializes all aspects of the system**, from CPU register to device controllers to memory content.

Locates and **loads the kernel into memory** for execution.

### Kernel

The part of the operating system that is running at all times.

On boot, starts executing the first process such as **init**.

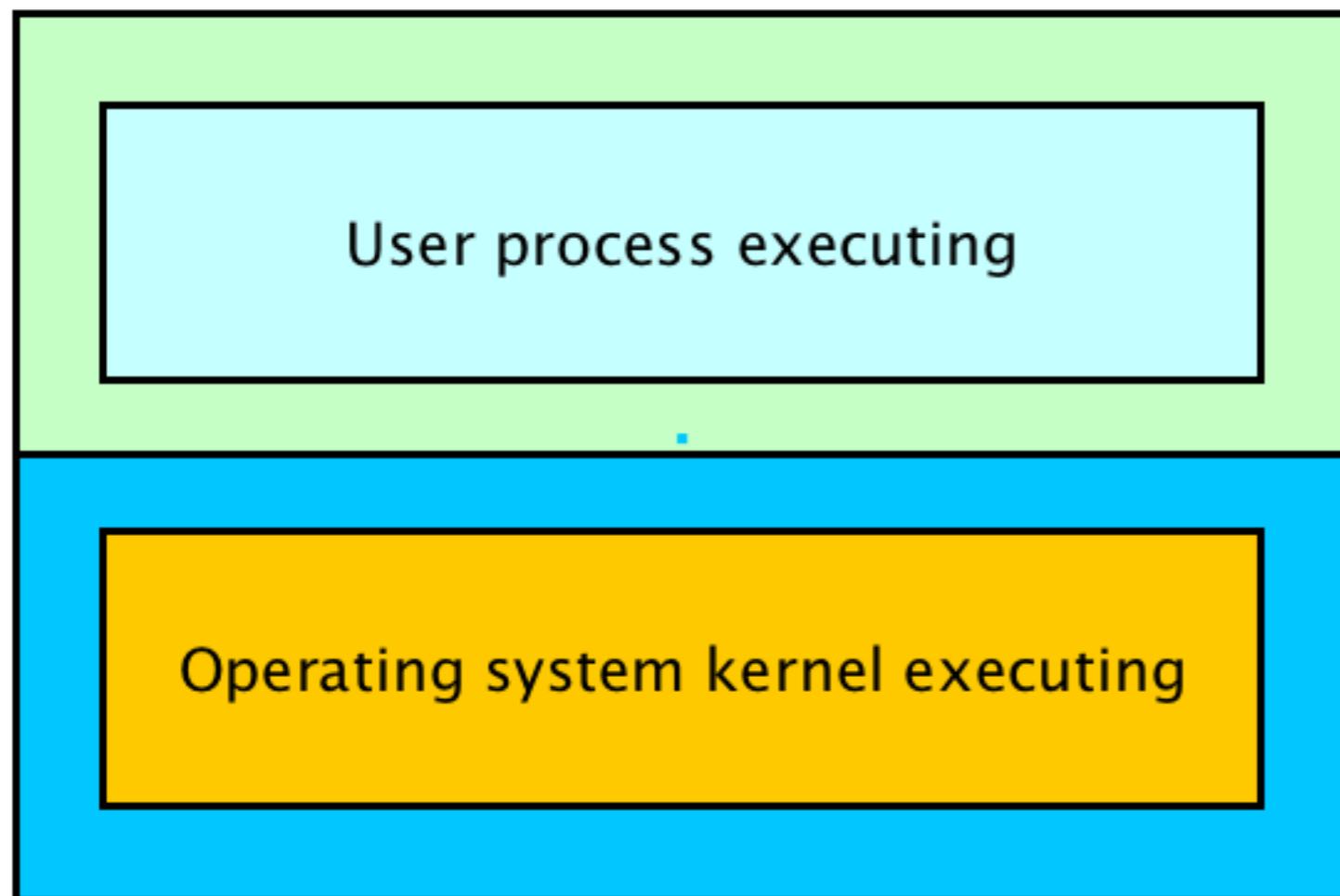
Waits for some **event** to occur ...

## Computer Hardware

# Dual mode operation

In order to protect the operating system from user processes and, two modes are provided by the hardware: **user mode** and **kernel mode**.

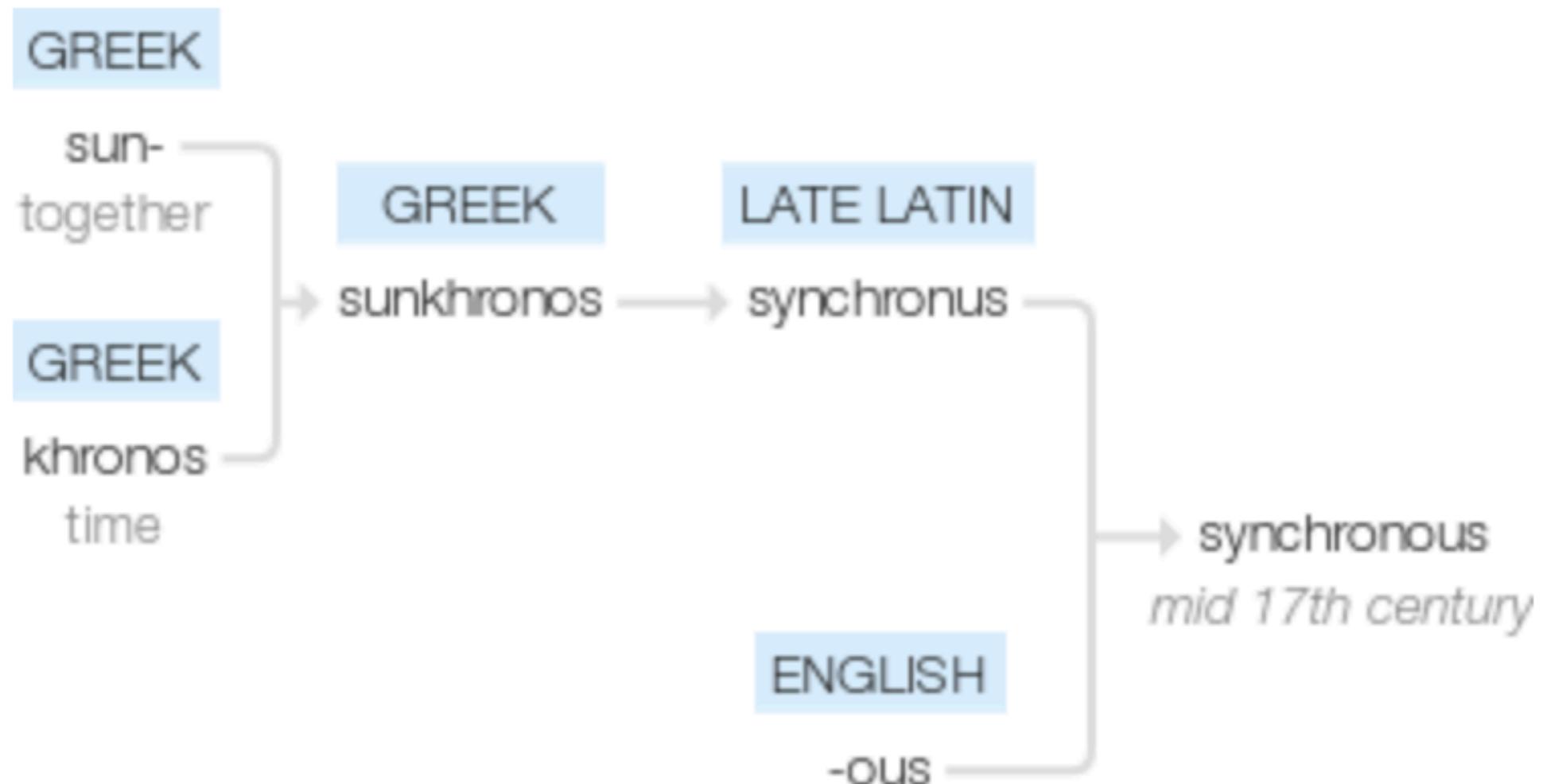
## User mode



## Kernel mode

Dual mode operation place restrictions on the type and scope of operations that can be executed by the CPU. This design allows the operating system kernel to execute with more privileges than user application processes.

# Synchronous



Mid 17th century: from late Latin *synchronous* (from Greek *sunkhronos*, from *sun-* 'together' + *khronos*'time') + *-ous*.

# Synchronous and asynchronous events

**Synchronous** means happening, existing, or arising at precisely the same time.<sup>1</sup>

If an event occurs at the same instruction every time the program is executed with the same data and memory allocation, the event is synchronous. An synchronous event is directly related to the instruction currently being executed by the CPU.

**Asynchronous** simply means “not synchronous”.

On the other hand, an asynchronous event is not directly related to the instruction currently being executed by the CPU.

1) <https://www.merriam-webster.com/dictionary/synchronous>

# **Exceptions and interrupts**

Interrupts and exceptions are used to notify the CPU of events that needs immediate attention during program execution.

# Exceptions are **internal** and **synchronous**

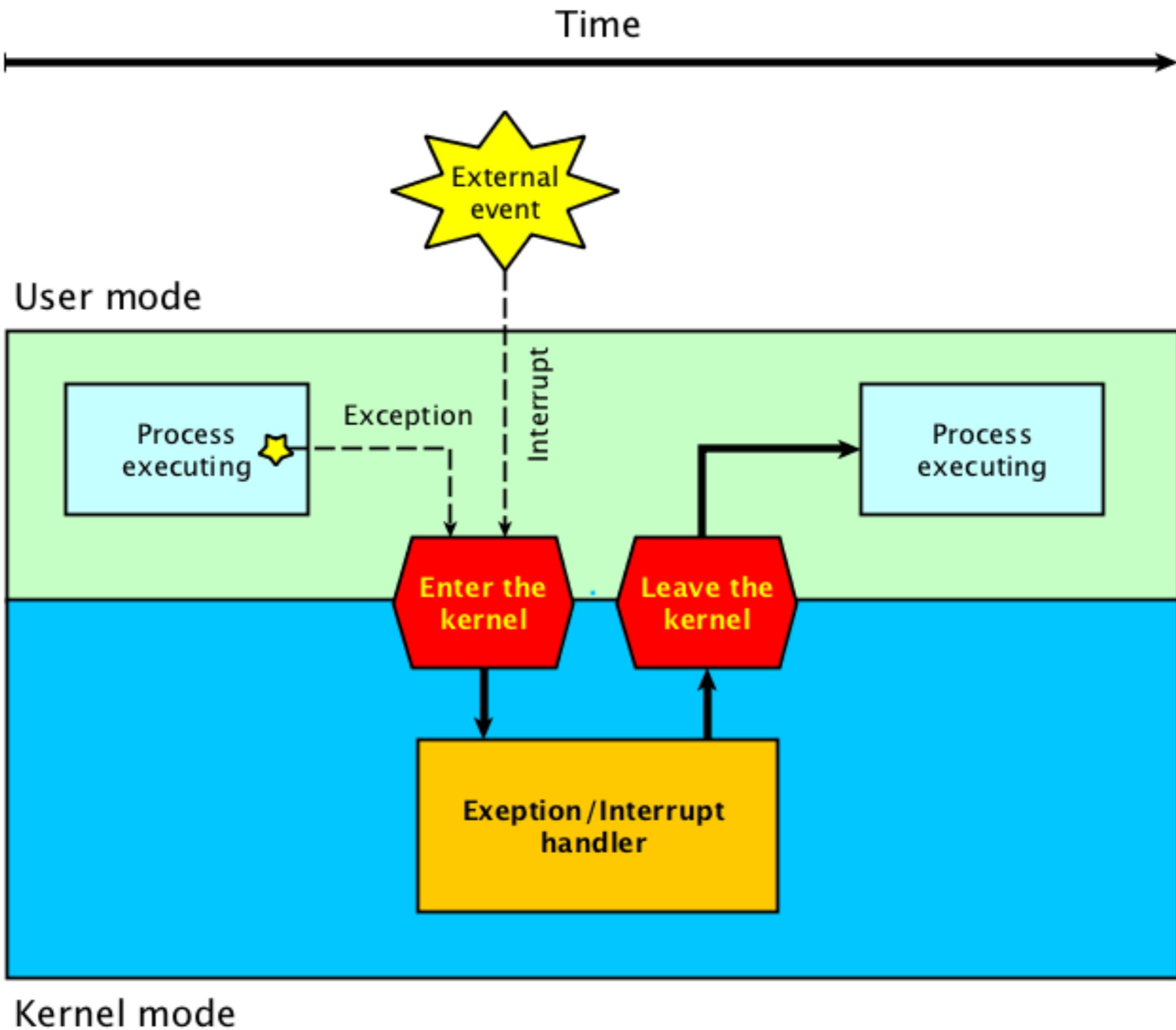
- ★ Exceptions are used to handle internal program errors.
- ★ Overflow, division by zero and bad data address are examples of internal errors in a program.
- ★ Another name for exception is trap. A trap (or exception) is a software generated interrupt.
- ★ Exceptions are produced by the CPU control unit while executing instructions and are considered to be synchronous because the control unit issues them only after terminating the execution of an instruction.

# Interrupts are external and asynchronous

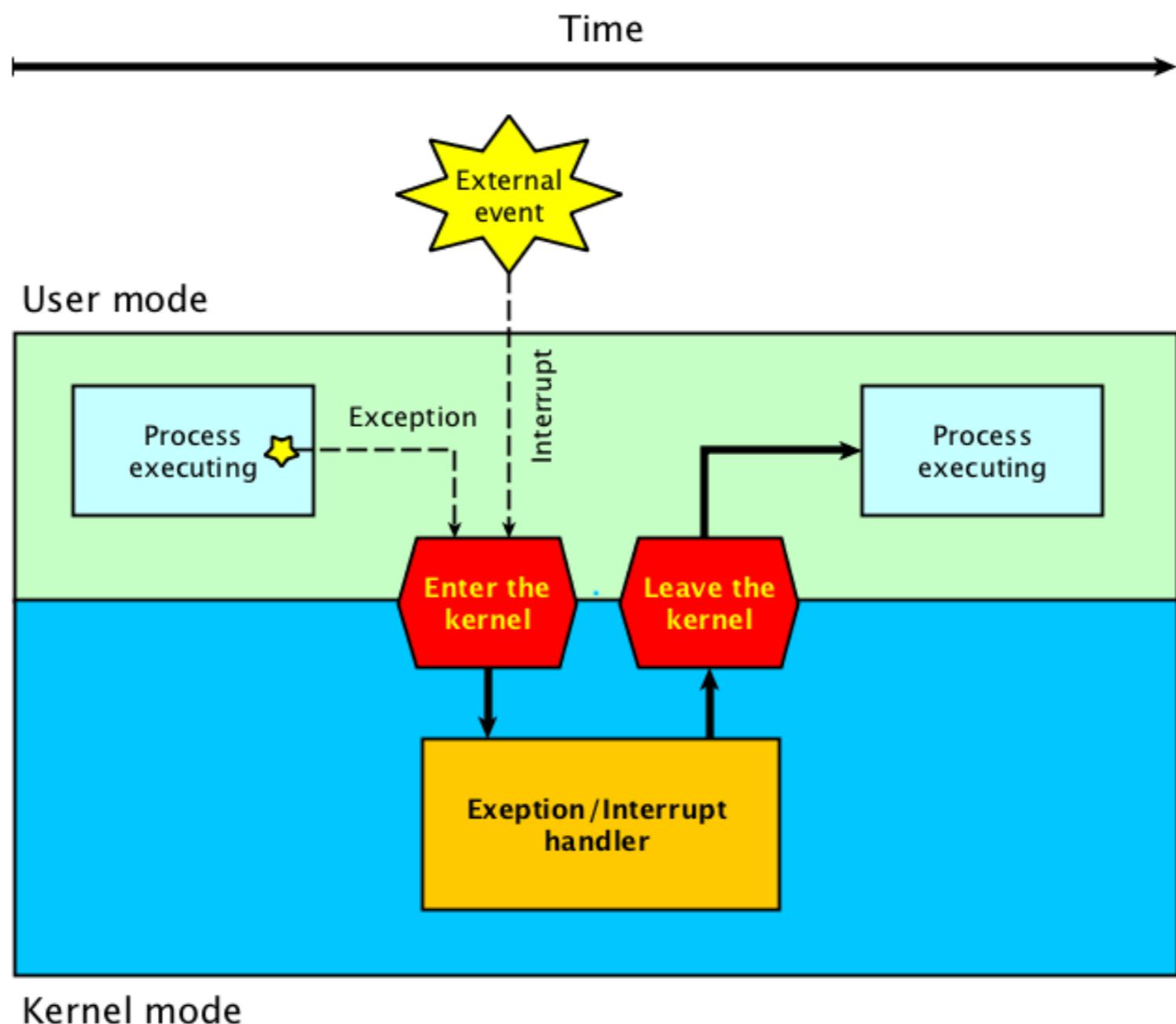
- ★ Interrupts are used to notify the CPU of external events.
- ★ Interrupts are generated by hardware devices outside the CPU at arbitrary times with respect to the CPU clock signals and are therefore considered to be asynchronous.
- ★ Key-presses on a keyboard might happen at any time. Even if a program is run multiple times with the same input data, the timing of the key presses will most likely vary.
- ★ Read and write requests to disk is similar to key presses. The disk controller is external to the executing process and the timing of a disk operation might vary even if the same program is executed several times.

# Exception and interrupt handling

- ★ When an exception or interrupt occurs, execution **transition** from user mode to kernel mode.
- ★ The **cause** of the interrupt or exception is determined.
- ★ The exception or interrupt is **handled**.
- ★ When the exception or interrupt has been handled execution **resumes** in user space.



# Problem?



# Problem

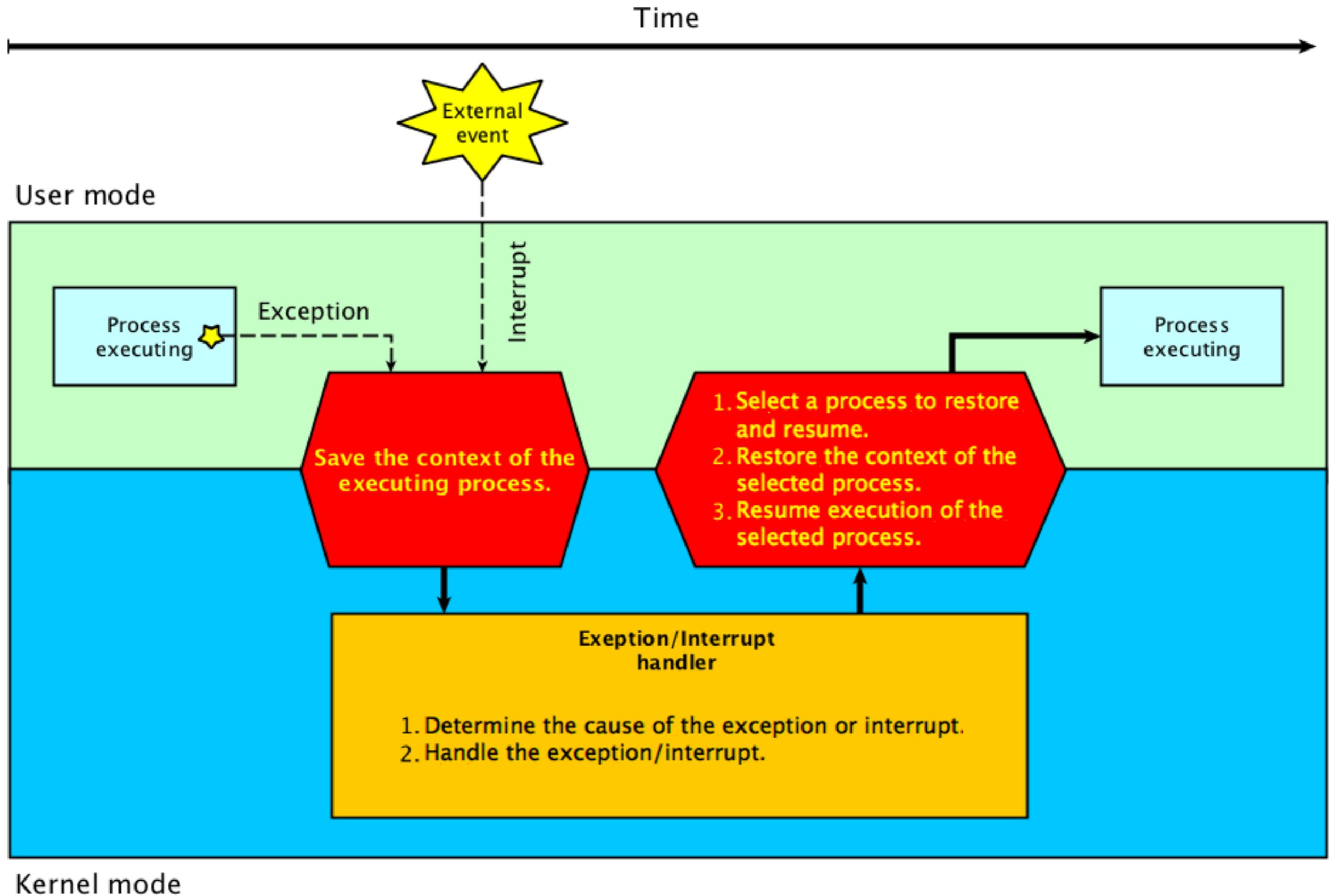
The exception/interrupt handler uses the same CPU (including the program counter and all registers) as the user processes.

## Saving cpu context

When **entering** the exception/interrupt handler, values in all registers must be **saved** to memory before the kernel can use the registers.

## Restoring cpu context

Before **resuming** execution of a user process, the values of all registers must be **restored** using the values saved to memory earlier.





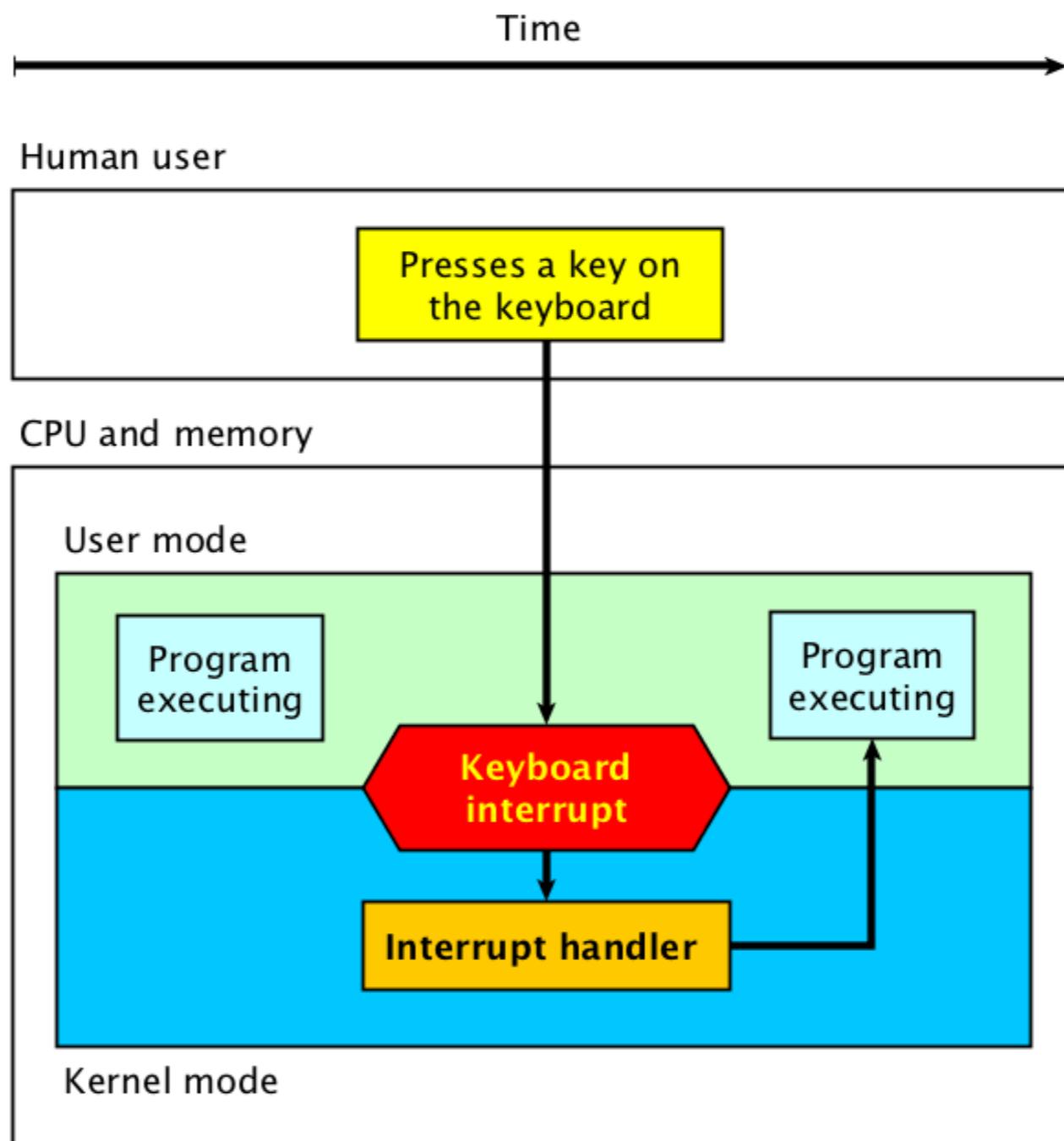
**Waiting for keyboard  
input**

# Waiting for keyboard input

- ★ Humans are very slow compared to the CPU. No matter how fast you are, the CPU will be able to execute a huge number of instructions between every key-press you make.
- ★ Is it possible to make the CPU do something useful while waiting for user input?

# Interrupt driven input

Using interrupts makes it possible for the CPU to do something useful, for example execute another program, while waiting for user input.



# **Multiprogramming**

## **Job**

In a multiprogramming system a program loaded to memory and ready to execute is called a job.

## **Execute another job while waiting for I/O**

The simple idea is to make a job wait for I/O "outside" the CPU. When a job makes a request for I/O the CPU will execute another job while the first job waits for the I/O request to complete.

## **Interrupts**

Interrupts are used to notify the system when an I/O request is completed.

# **States**

In a multiprogramming system, a job can be in one of three states.

## **Running**

The job is currently executing on the CPU. At any time, at most one job can be in this state.

## **Ready**

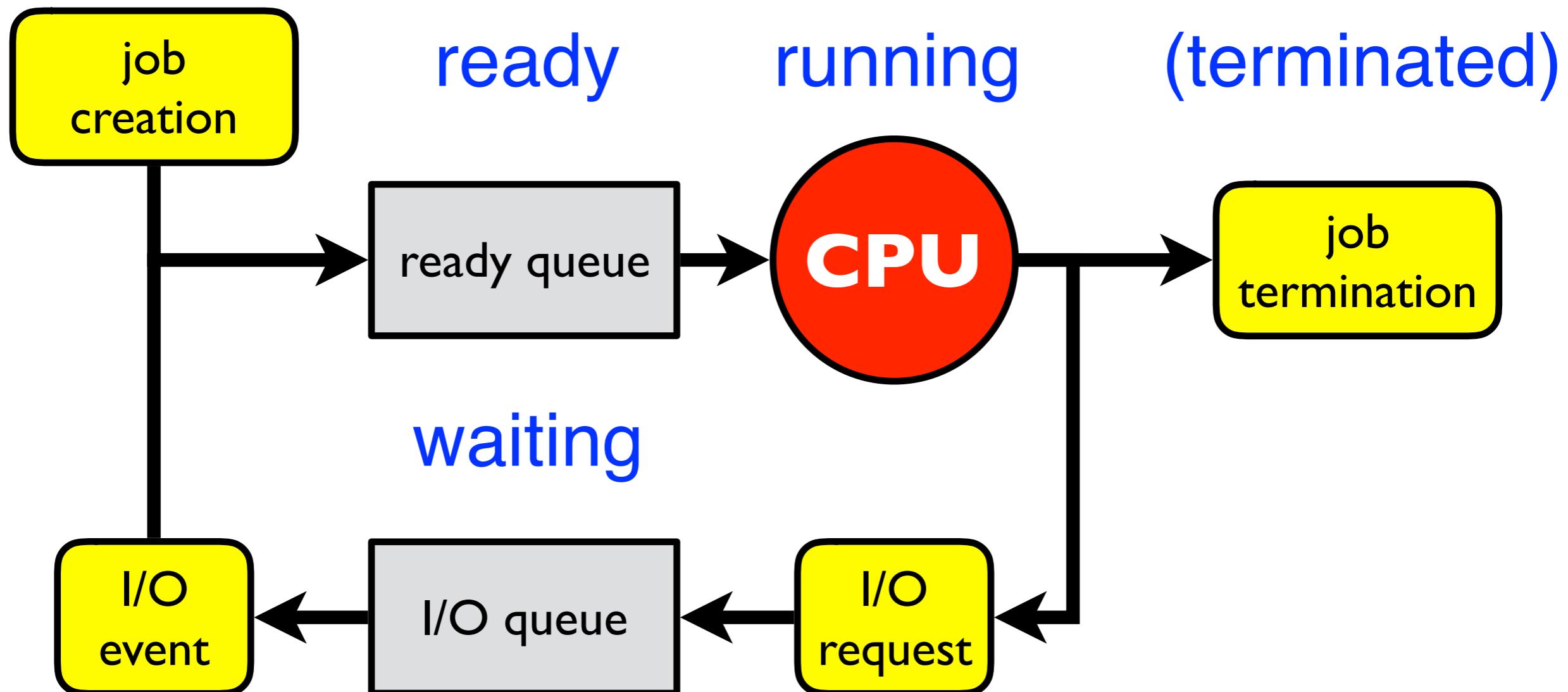
The job is ready to run but currently not selected to do so.

## **Waiting**

The job is blocked from running on the CPU while waiting for an I/O request to be completed.

# Multiprogramming

A schematic view of multiprogramming



**Interrupt** used to  
signal an external and  
asynchronous I/O  
event

# Memory

# CPU



**Job 1**

Ready

**Job 2**

Ready

**Job 3**

Ready

**Job 4**

Ready

In a multiprogramming system, several jobs are kept in memory at the same time. Initially, all jobs are in the ready state.

# Memory

# CPU

Job 2  
Ready

Job 3  
Ready

Job 4  
Ready

Job 1  
**Running**



One of the ready jobs is selected to execute on the CPU and changes state from ready to running. In this example, job 1 is selected to execute.

# Memory

# CPU



Eventually, the running job makes a request for I/O and the state changes from running to waiting.

In this example the jobs makes a request to wait for the human user to press a key one the keyboard.

# Memory

# CPU

**Job 1**  
Waiting

**Job 2**  
Ready

**Job 3**  
Ready

**Job 4**  
Ready



# Memory

# CPU

**Job 1**  
Waiting

**Job 2**  
Ready

**Job 4**  
Ready

**Job 3**  
**Running**



Instead of idle waiting for the I/O request to complete, one of the ready jobs is selected to execute on the CPU and have its state change from ready to running.

In this example job 3 is selected to execute.

# Memory

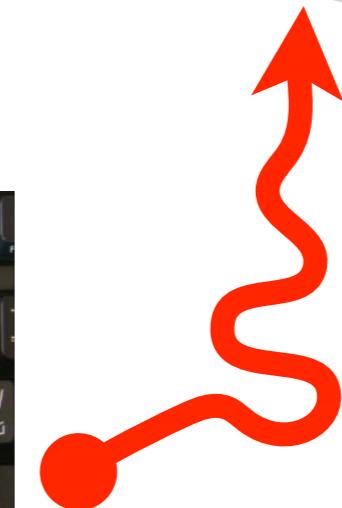
**Job 1**  
Waiting

**Job 2**  
Ready

**Job 4**  
Ready

# CPU

**Job 3**  
Running



**Interrupt**

Eventually the I/O request job 1 is waiting for will complete and the CPU will be notified by an interrupt. In this example, job 1 was waiting for a keypress on the keyboard.

# Memory

CPU

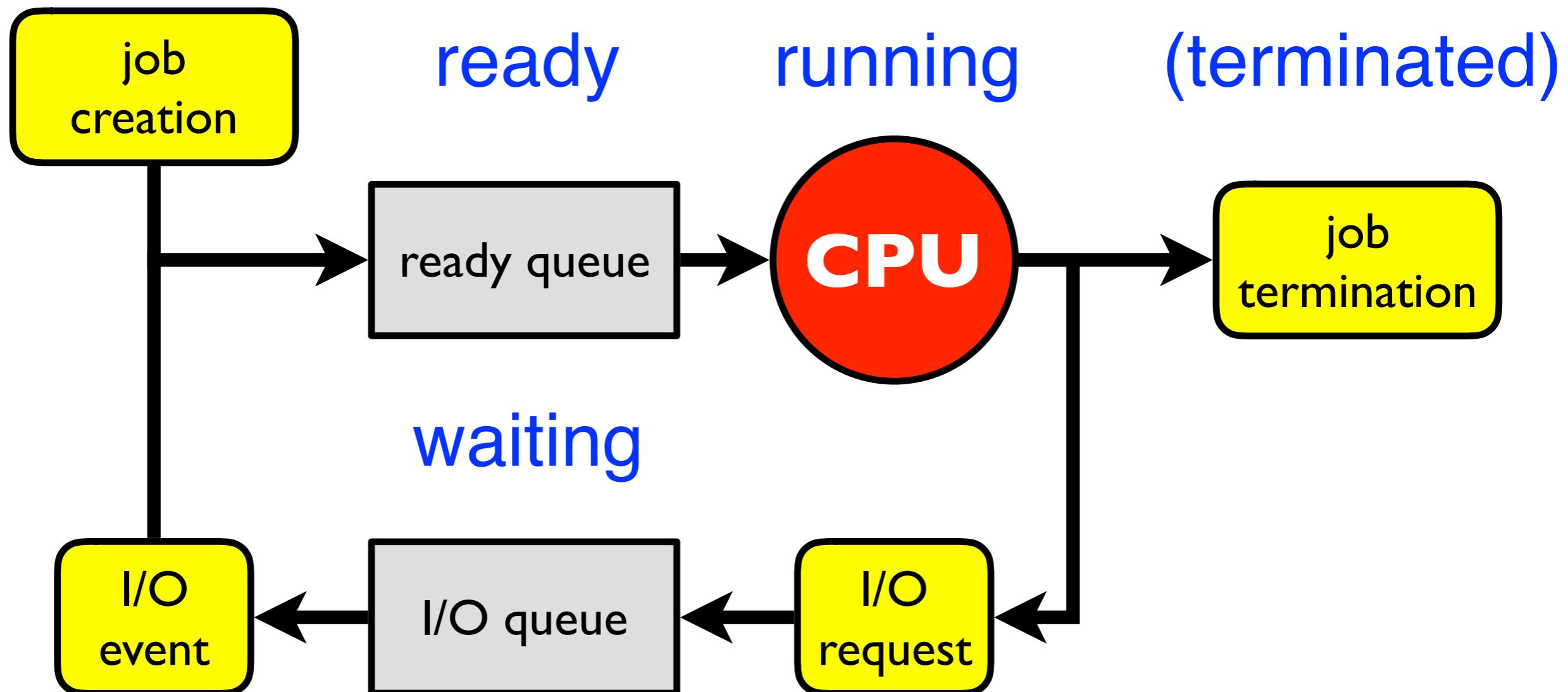


The state of the waiting job (job 1) will change from waiting to ready.

Job 3 will execute until making a request for I/O.

# Multiprogramming

A schematic view of multiprogramming



**Interrupt** used to  
signal an external and  
asynchronous I/O  
event

# Observations

**Multiprogramming** keeps the CPU busy despite individual jobs waiting for I/O.

This seems good if we want to **maximize** the **CPU utilization**.

# Problems

Each job might execute quite some time before any other job gets a spin on the CPU.

We must ensure that the operating system maintains **control over the CPU**.

We cannot allow a user program to get stuck in an infinite loop or never to request I/O.

# Solution

Use a **timer**.

The timer can be set to interrupt the currently executing job after a specified period of time if the job does not request I/O.

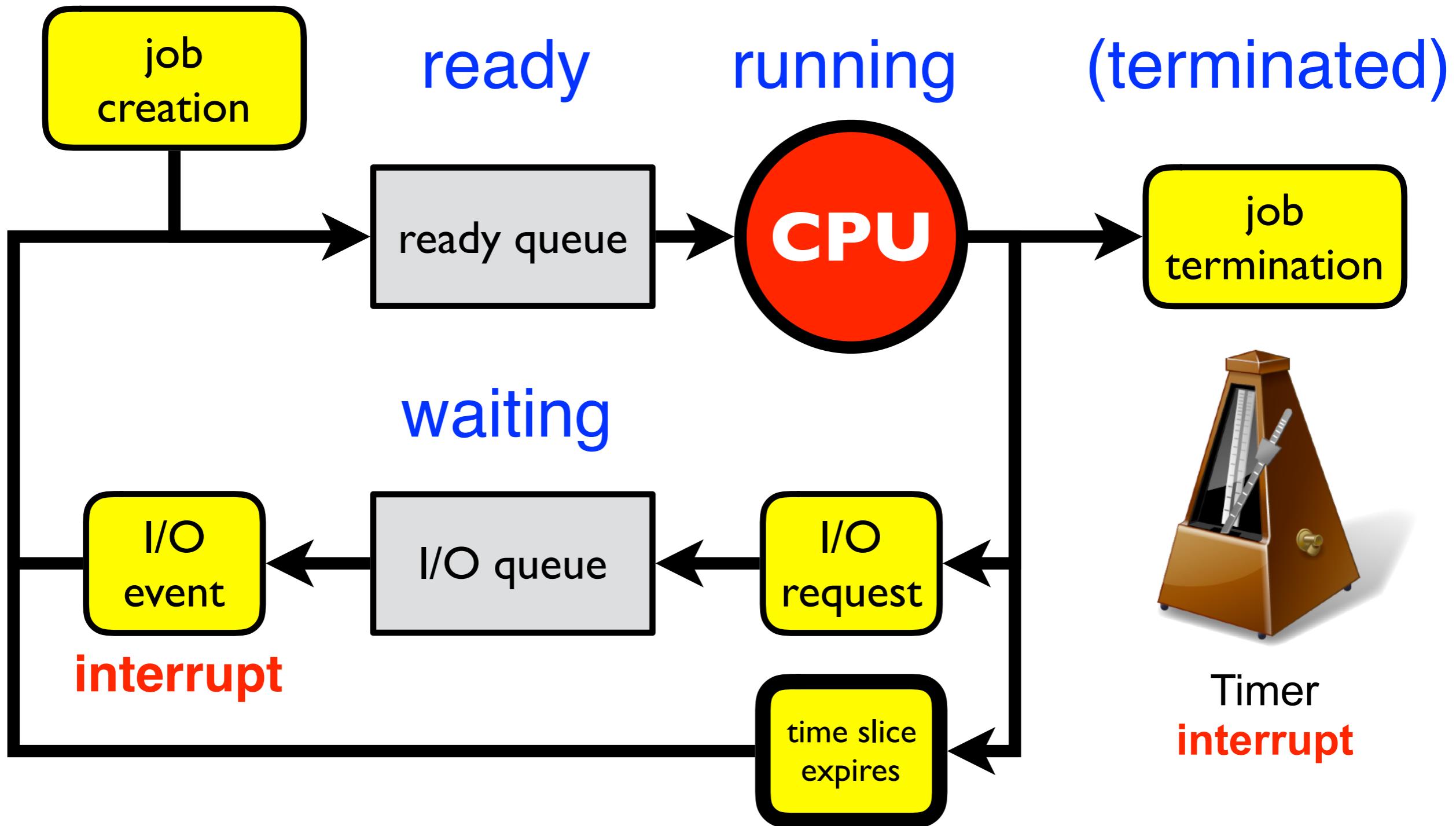
# Multitasking

Multitasking (aka **time sharing**) is a logical extension of multiprogramming where a **timer** is set to cause an **interrupt** at a regular time interval.

The running job is replaced if the job requests I/O or if the job is interrupted by the timer. This way, the running job is given a **time slice** of execution than cannot be exceeded.

# Multitasking

A schematic view of multitasking



# Design objectives

Multiprogramming and time sharing are designed to achieve different goals.

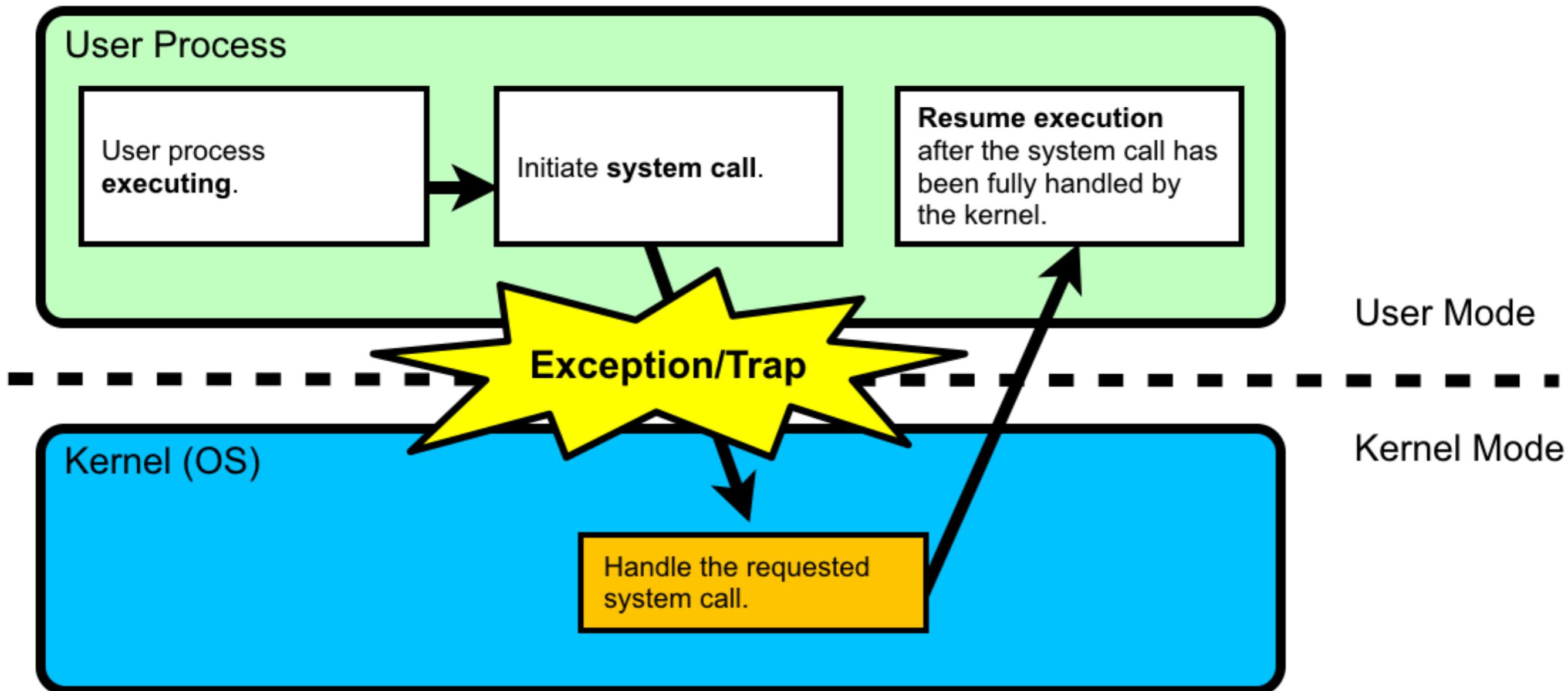
- ★ The objective of **multiprogramming** is to have some job running at all time, to **maximize CPU utilization**.
- ★ The objective of **time sharing** is **user interaction**, to switch the CPU among jobs so frequently that users can interact with each program while they are running seemingly at the same time.

**System**

**cais**

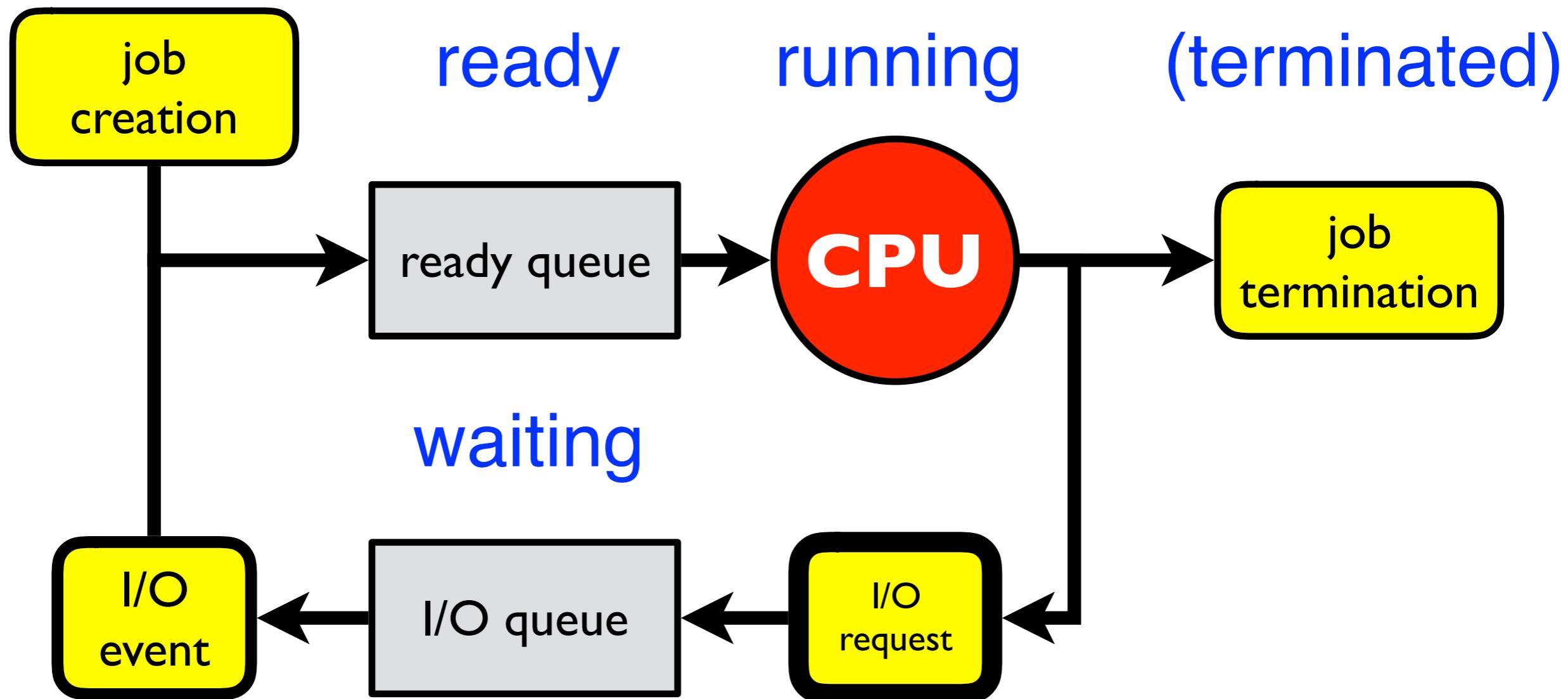
# System calls

- ★ A user program **requests service** from the operating system using system calls.
- ★ System calls are **implemented** using a special **system call exception**. Another name for exception is **trap**.
- ★ System calls forms an **interface** between **user programs** and the **operating system**.



# Multiprogramming

A schematic view of multiprogramming

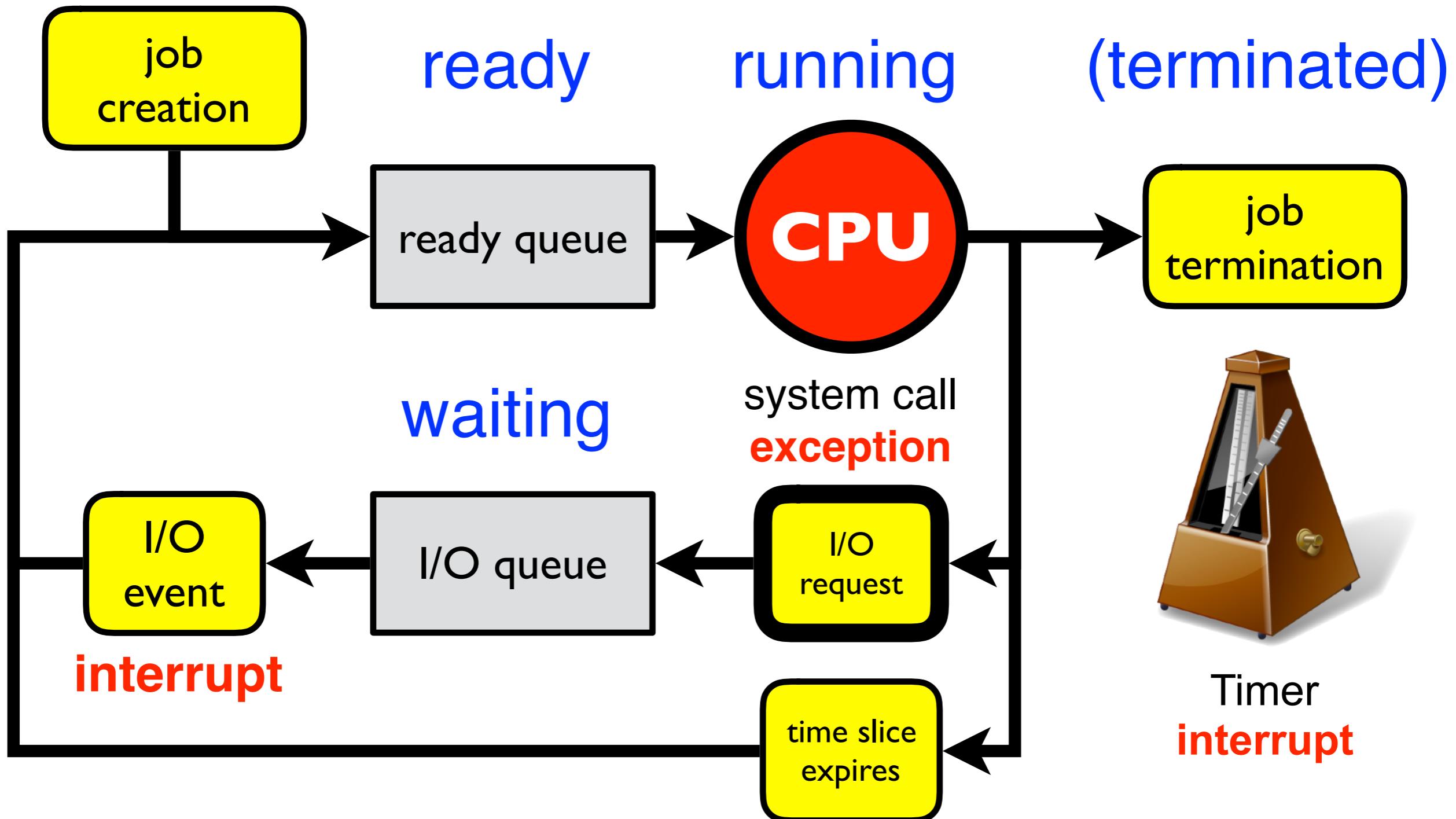


**Interrupt** used to signal  
an external and  
asynchronous I/O event

System call initiated  
using a special  
system call **exception**

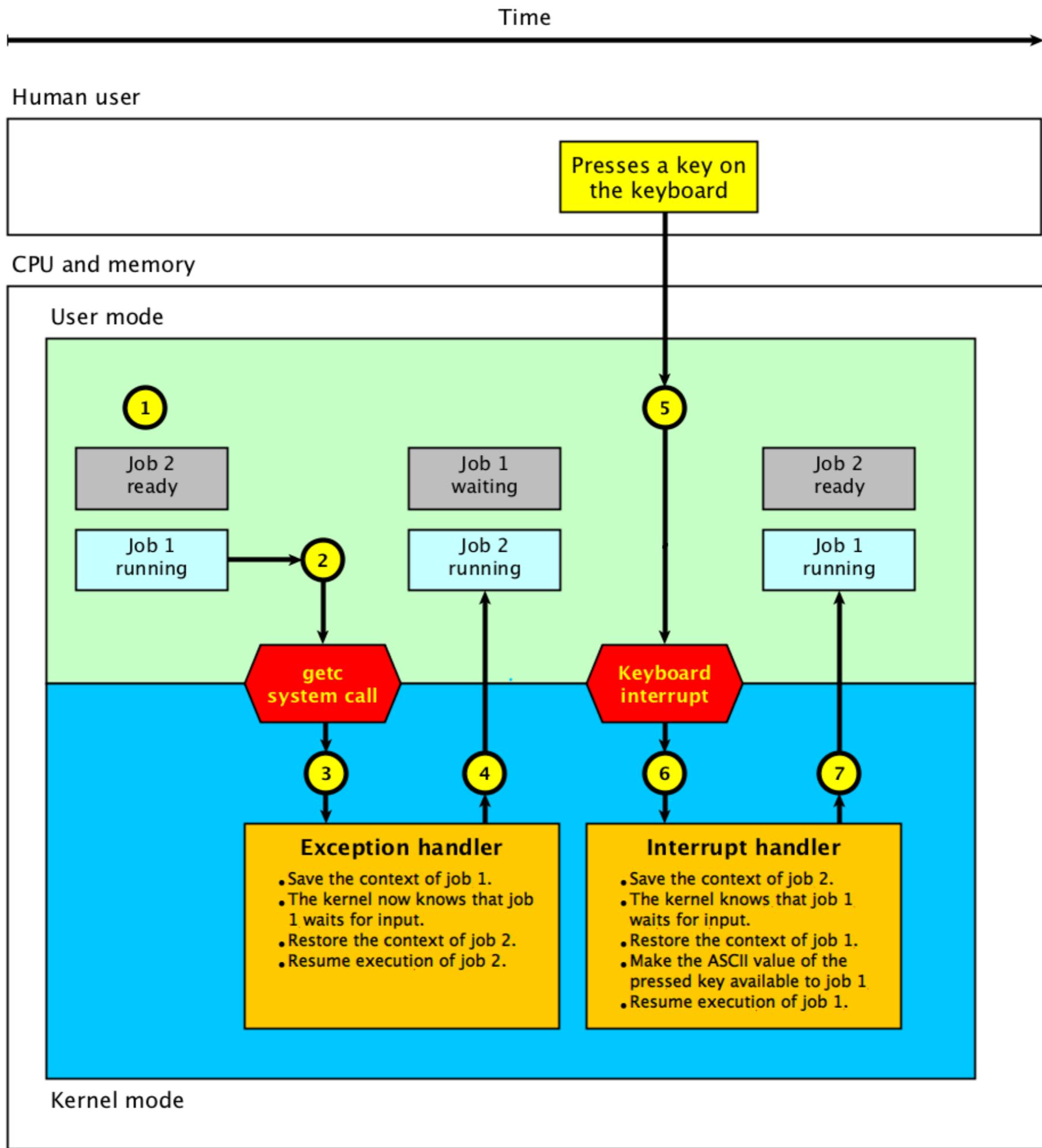
# Multitasking

A schematic view of multitasking



# Read character system call design

Let's sketch the design for a system call similar to the C library function **getc** that allows a program to read a single character typed by a human user on the keyboard.



# Problem

What if more than one process calls **getc**.  
Which process gets which key stroke?

# Read string system call design

Let's sketch the design for a system call similar to the C library function **gets** that allows a program to read a string typed by a human on the keyboard.

## Time Human user

## CPU and memory

