Fundamental concepts

Module 1

Lecture

Operating systems 2020

1DT003, 1DT044 and 1DT096

karl.marklund@it.uu.se

Uppsala university

Operating System I
(5 hp)

Operating systems
and process-oriented
programming
(15 hp)

Computer systems
with project work
(20 hp)

Period 3

Period 4

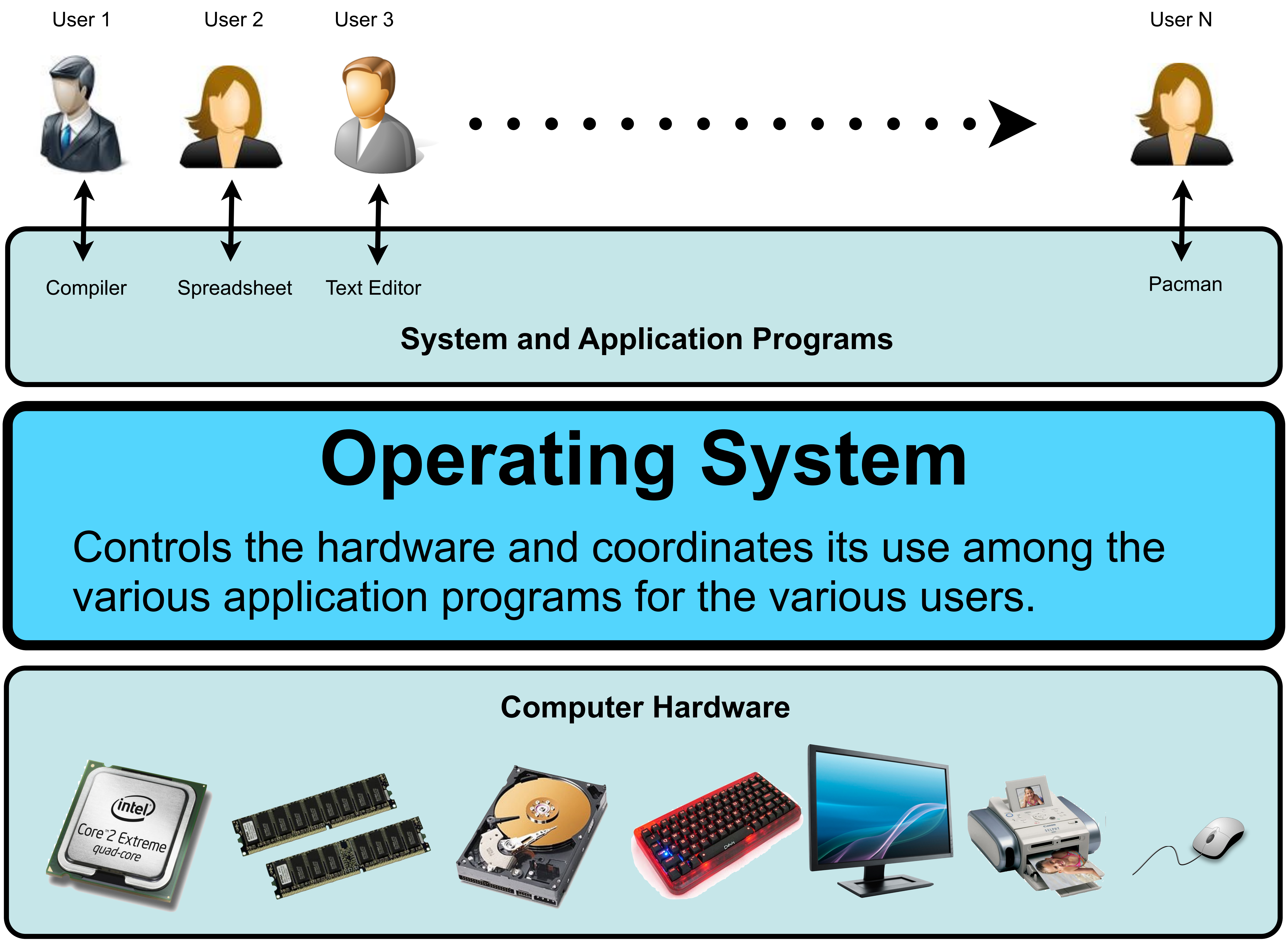
Learning and understanding

Learning and understanding

Top down

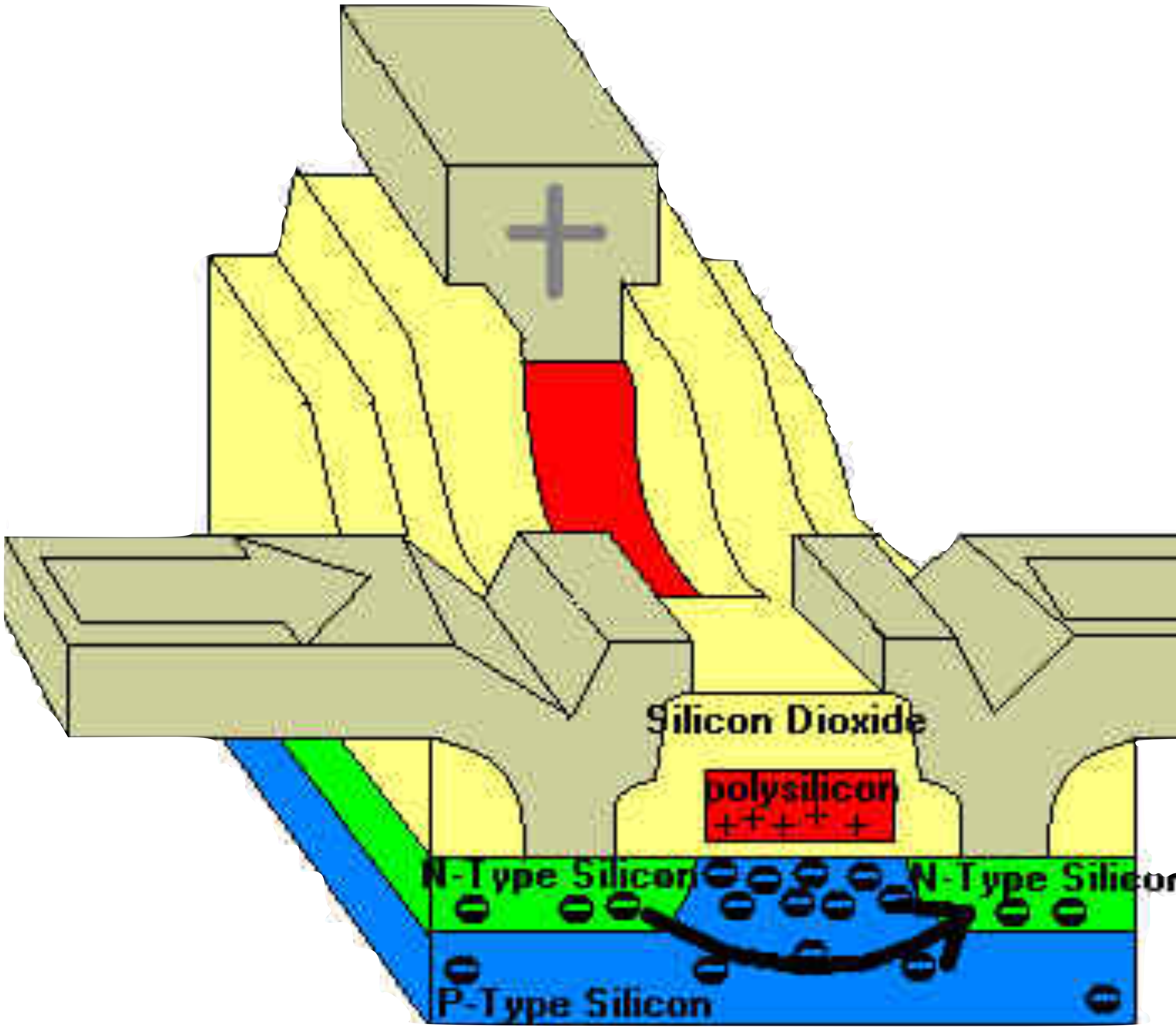
Bottom up

One top down view from the previous lecture

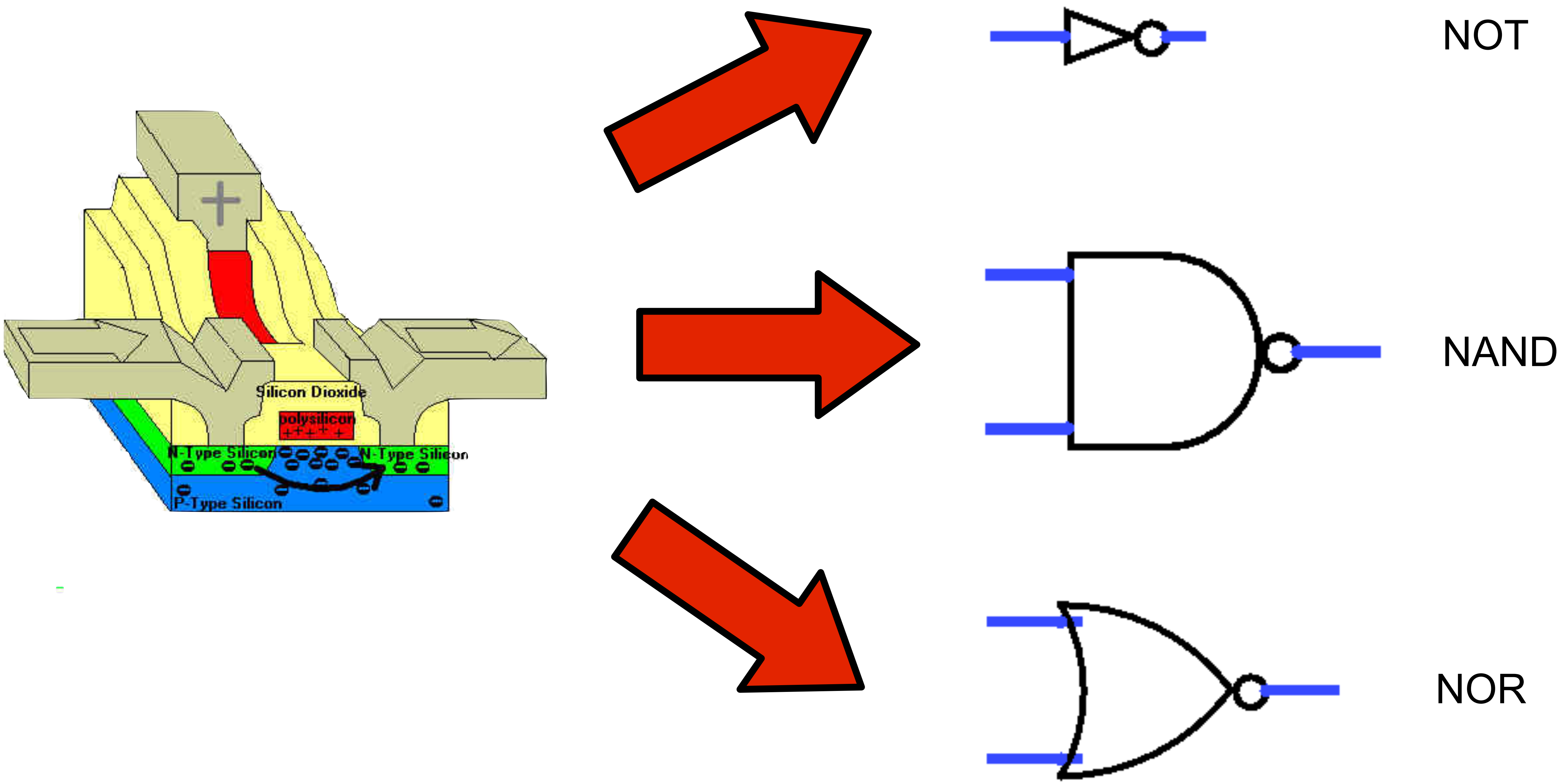

User 1
User 2
User 3
User N
Compiler
Spreadsheet
Text Editor
Pacman
System and Application Programs
Operating System
Controls the hardware and coordinates its use among the various application programs for the various users.
Computer Hardware
intel
Core 2 Extreme
quad-core

Now, let's start bottom up

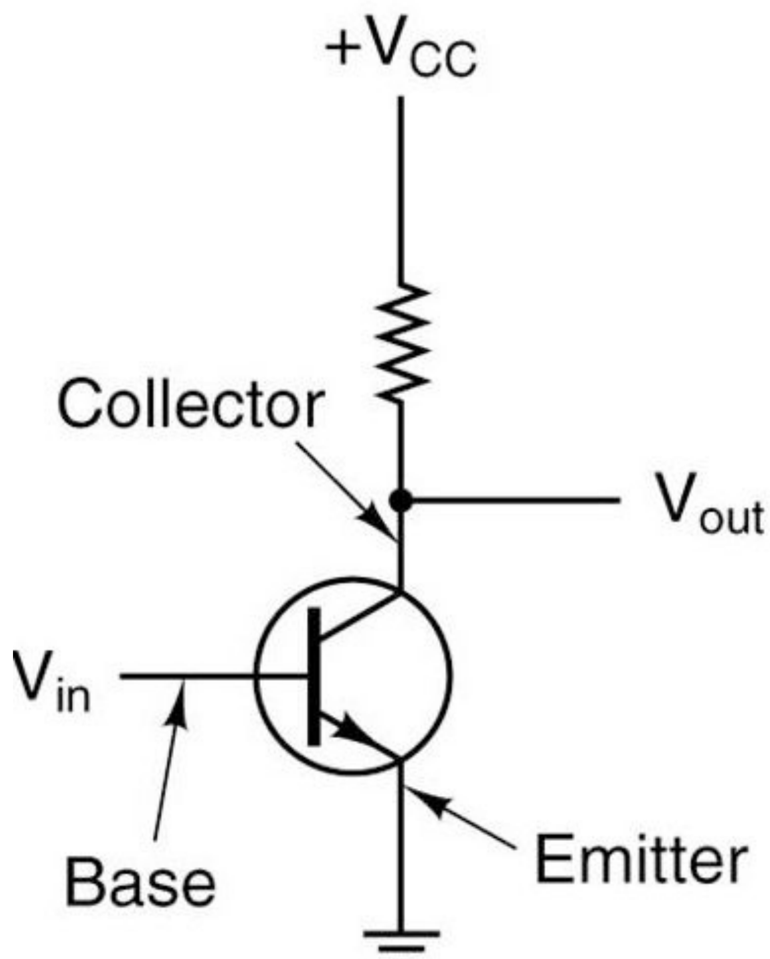
The transistor

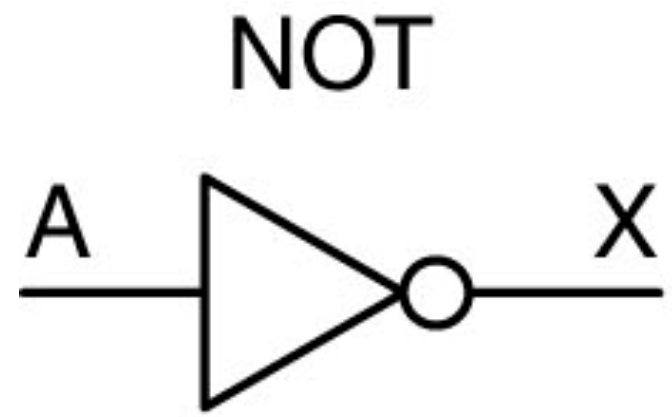


From transistors to logical gates



An inverter - logical NOT

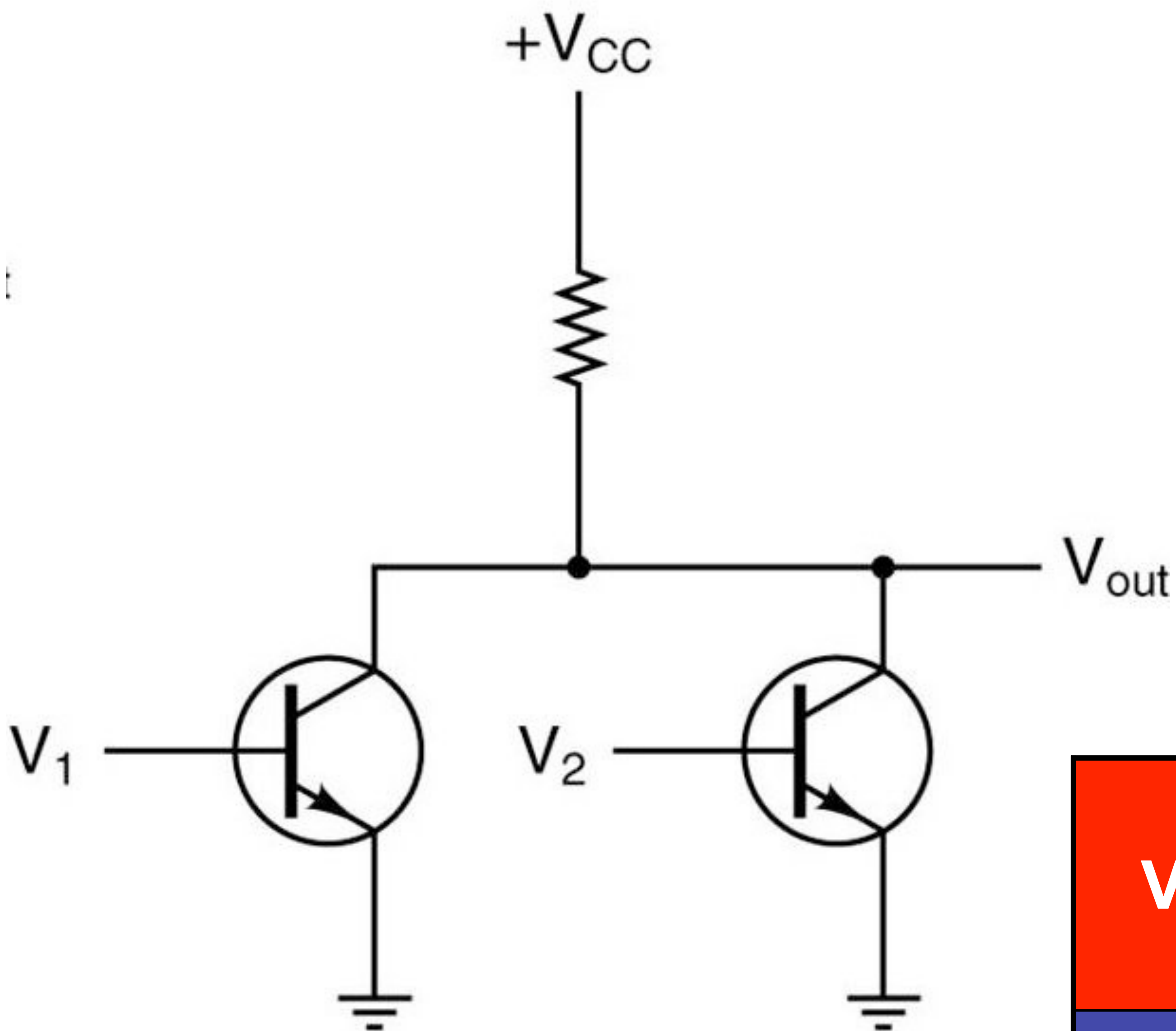


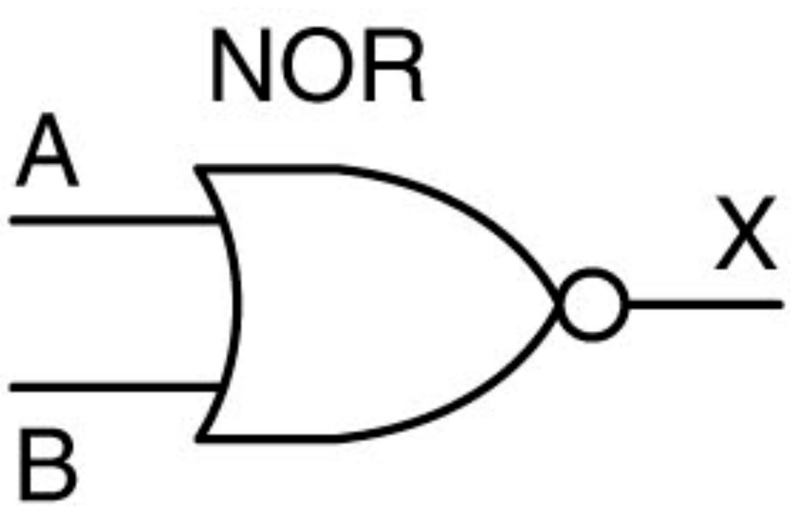


V_{in}	V_{out}
low	high
high	low

A	X
0	1
1	0

logical NOR

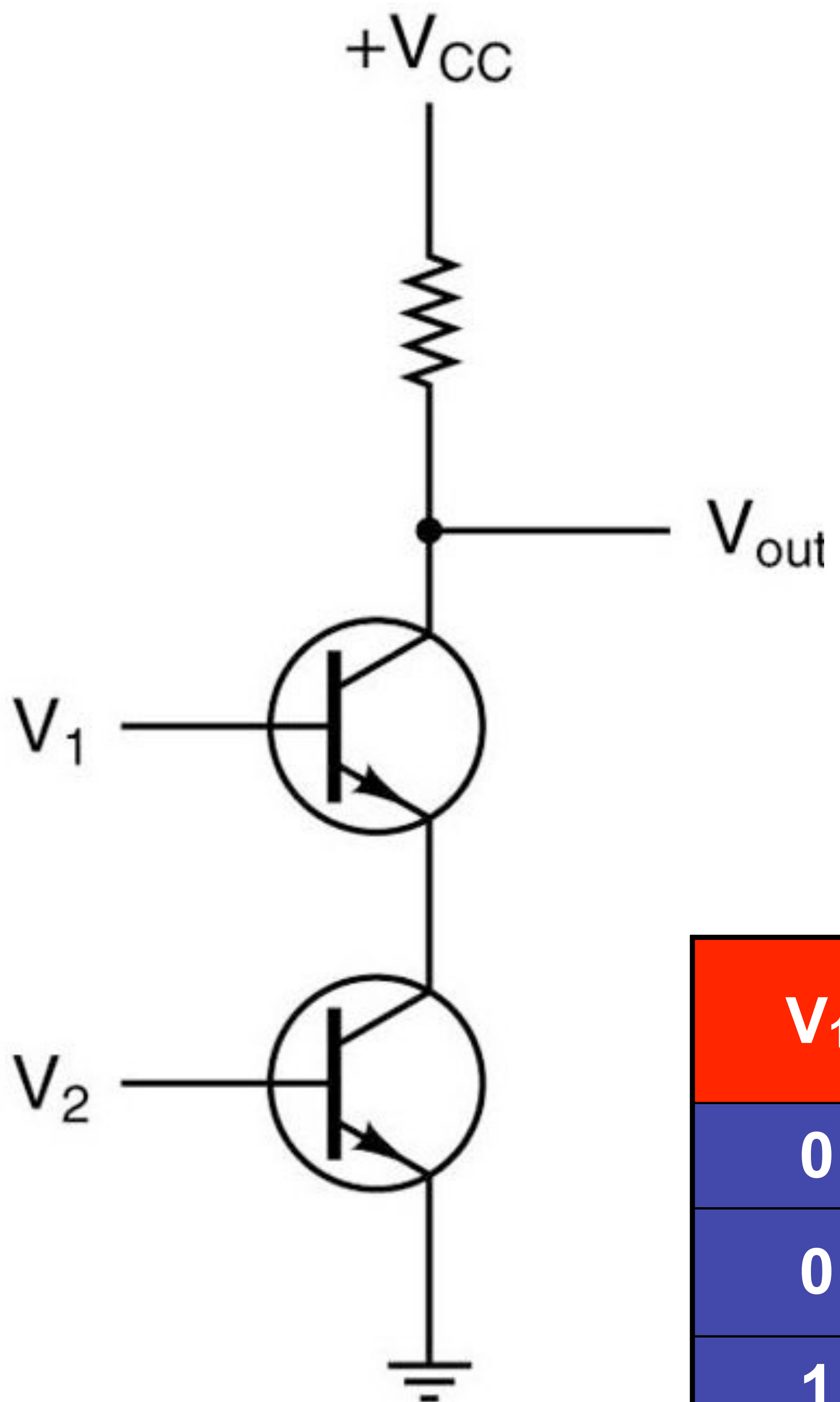


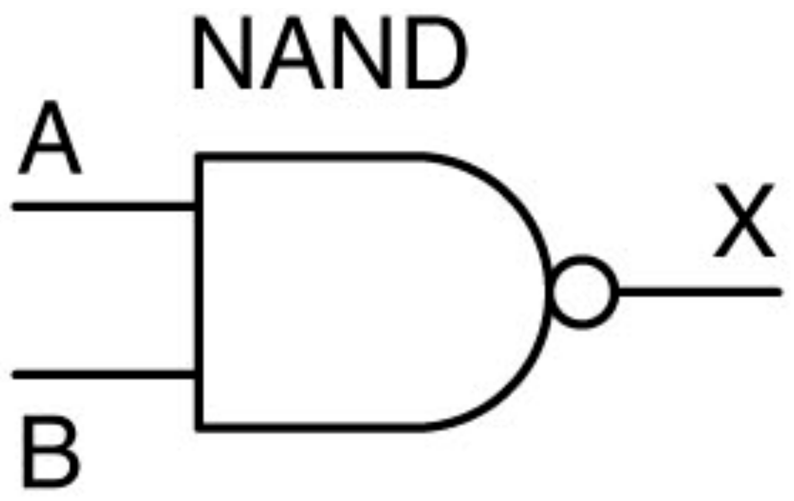


V_1	V_2	V_{out}	A or B	a nor b = not (A or B)
0	0	1	0	1
0	1	0	1	0
1	0	0	1	0
1	1	0	1	0

Low → 0 High → 1

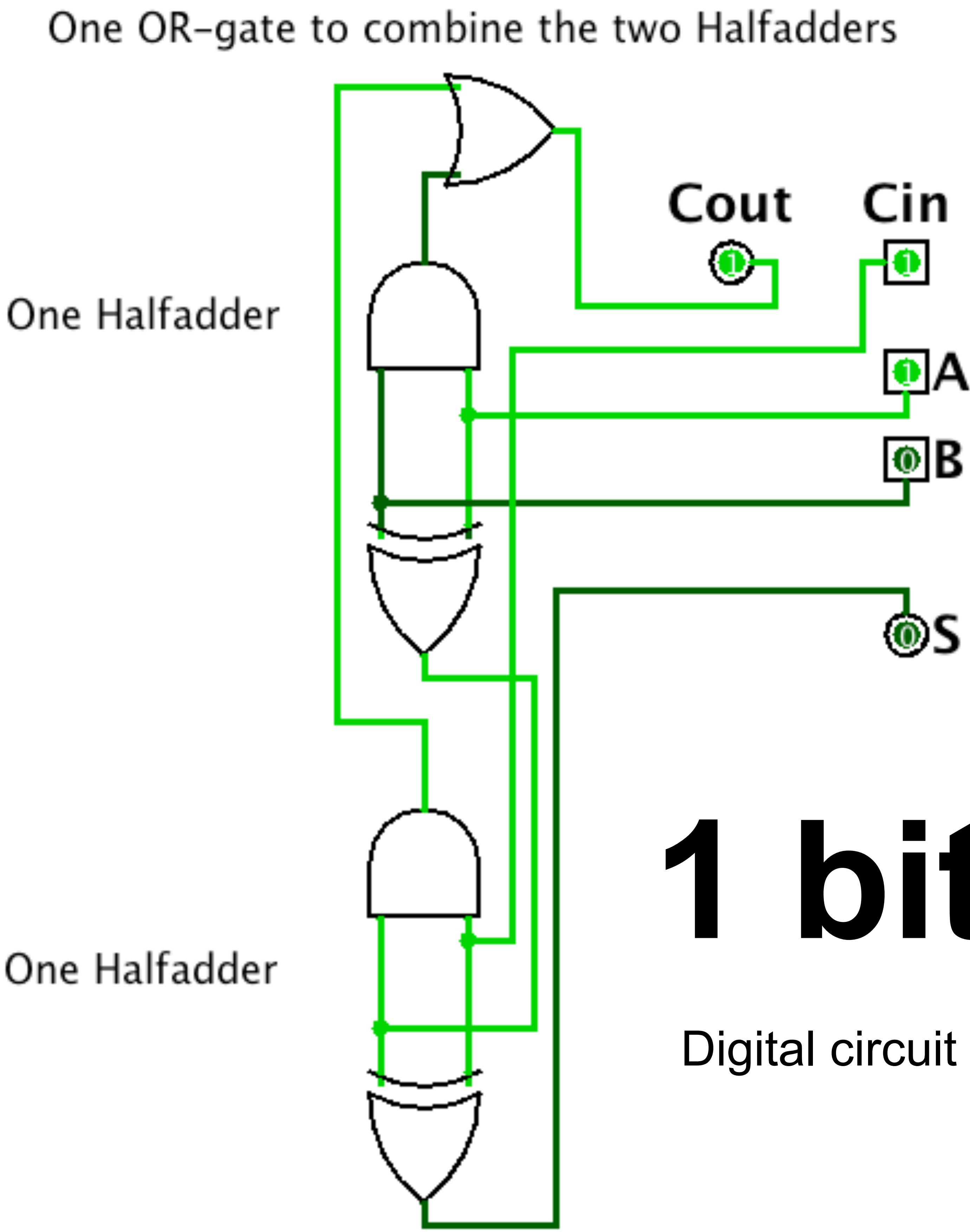
logical NAND





V_1	V_2	V_{out}	A and B	A nand B = not (A 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

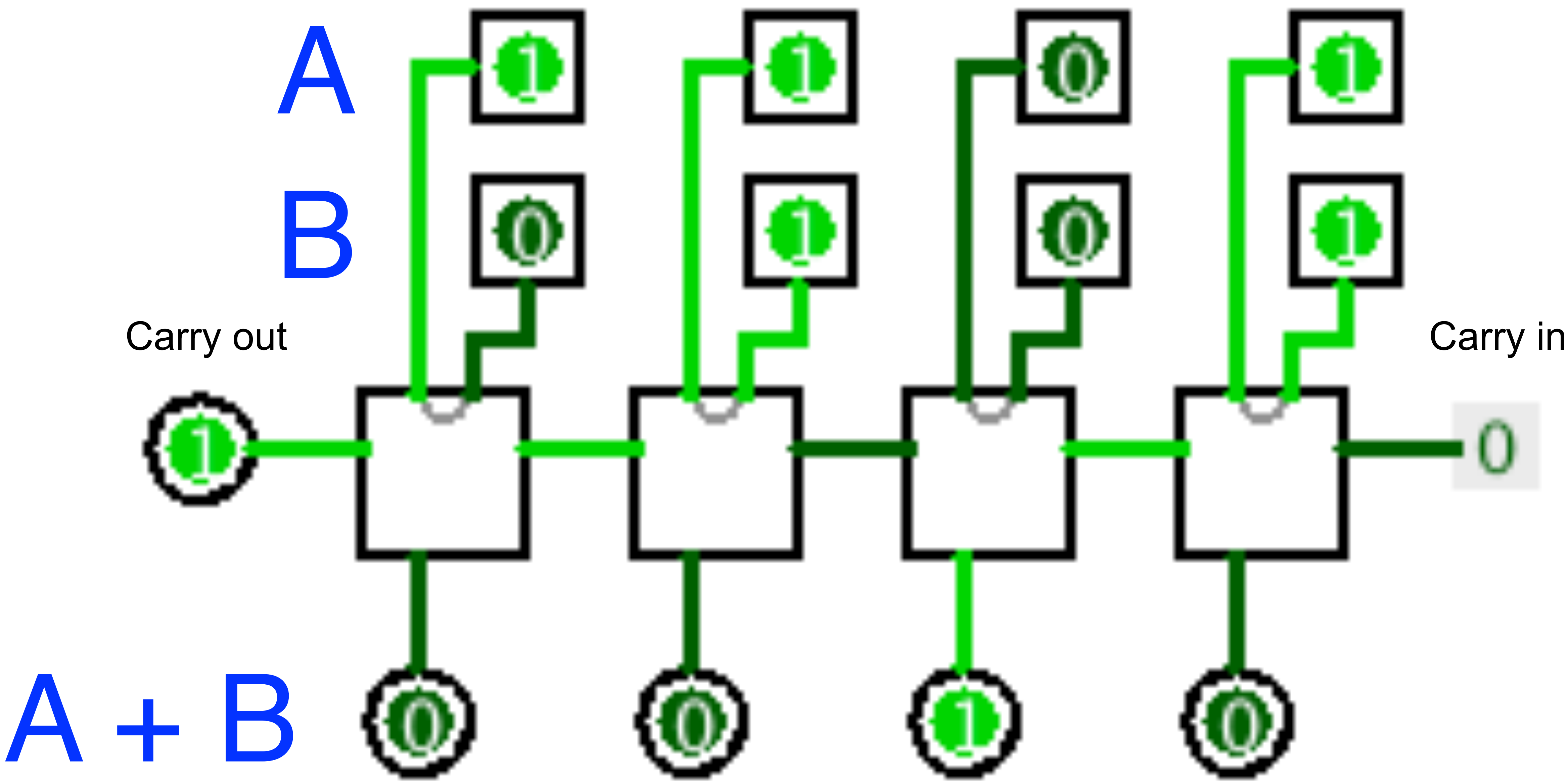


1 bit full adder

Digital circuit for adding two 1 bit numbers A and B

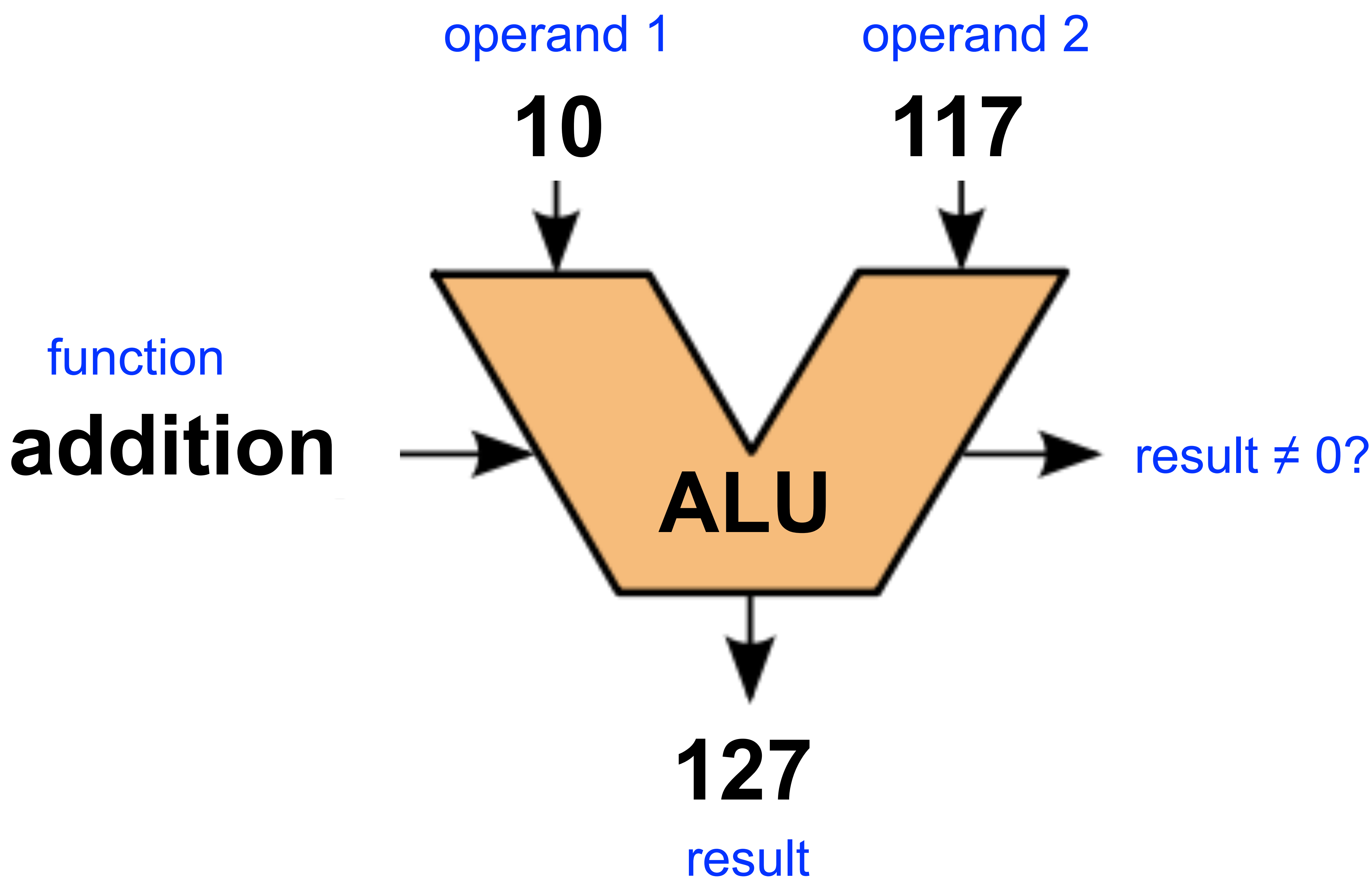
Ripple adder

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



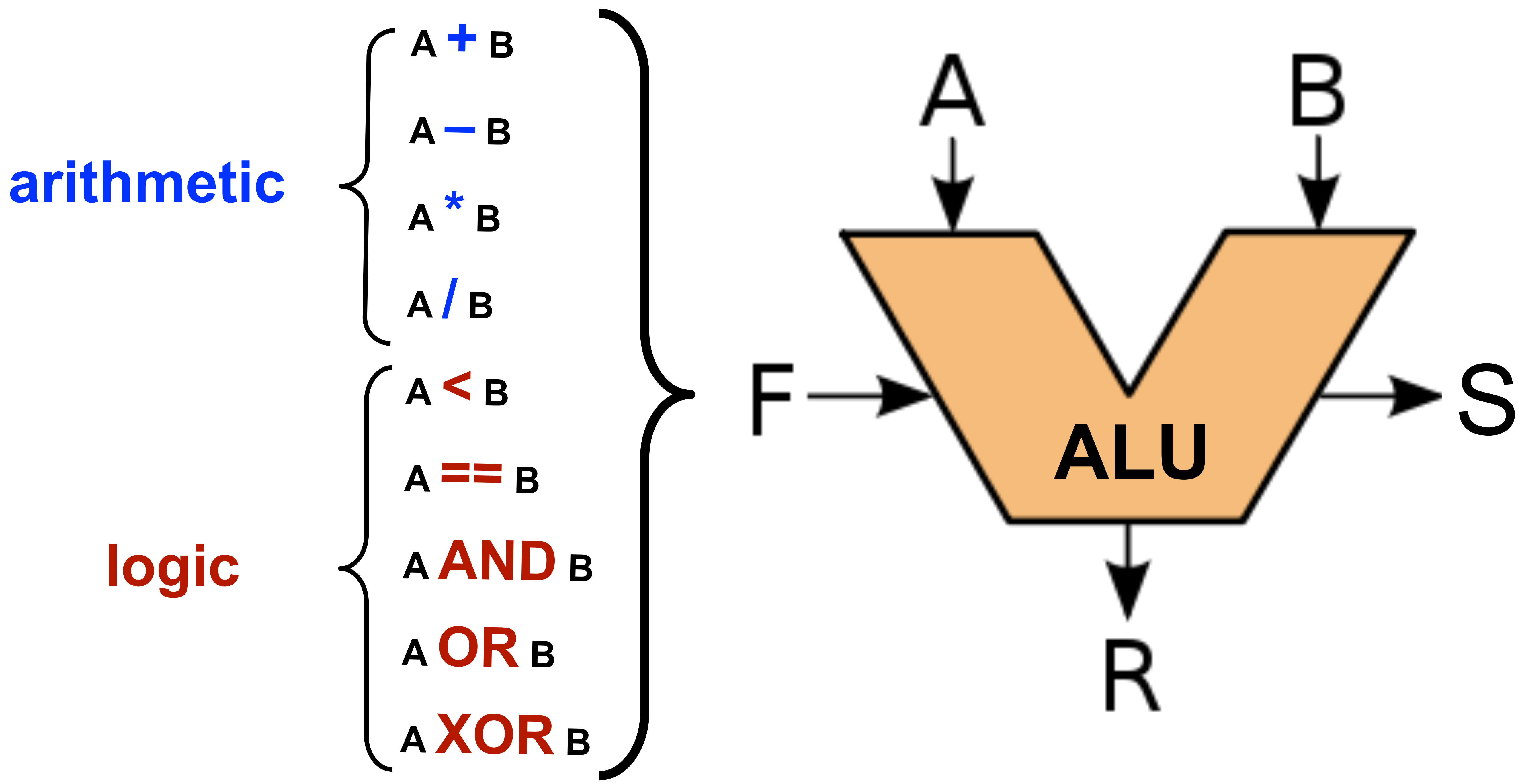
Arithmetic Logic Unit (ALU)

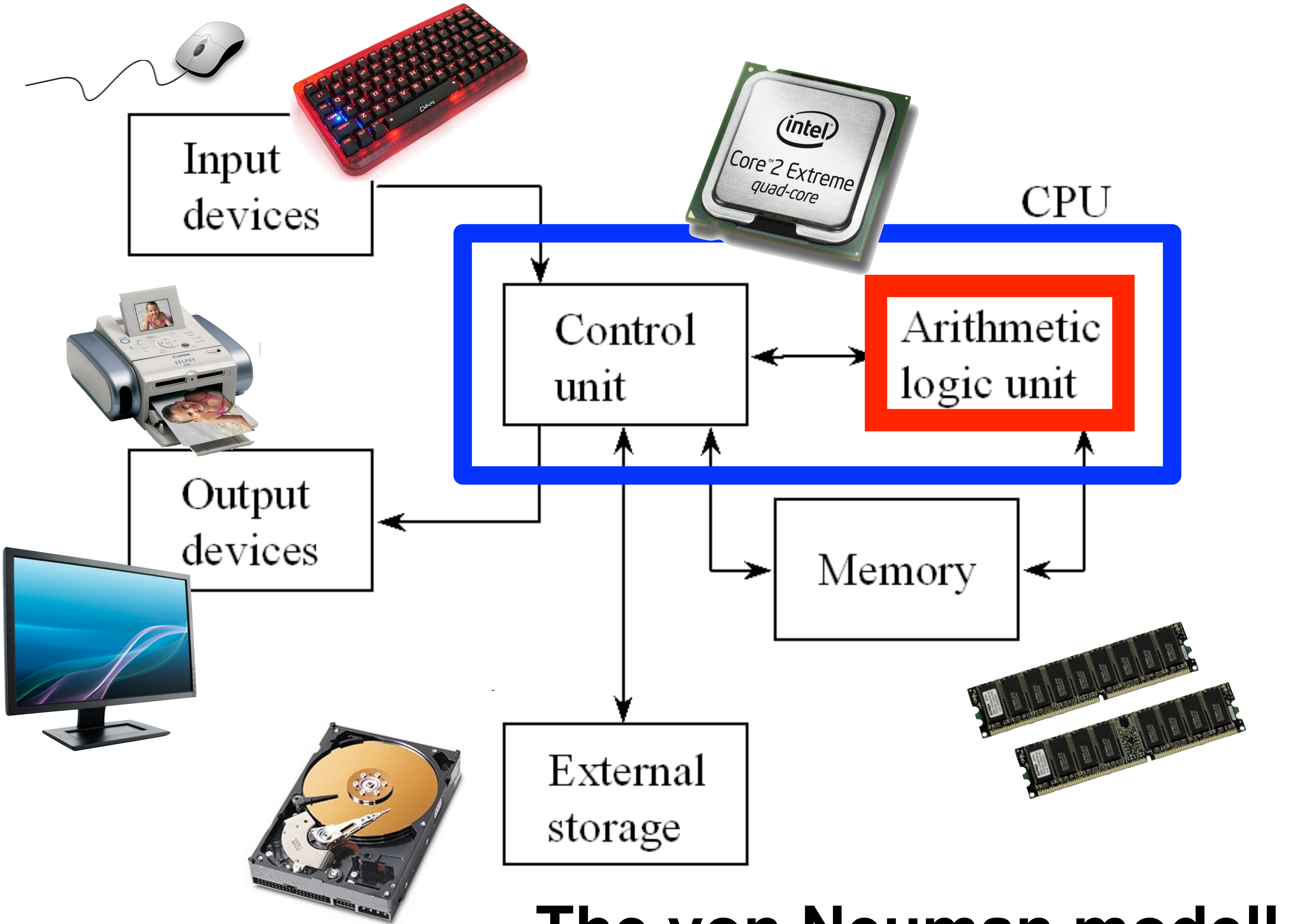
An adder is the first part in constructing a working ALU.



Arithmetic Logic Unit (ALU)

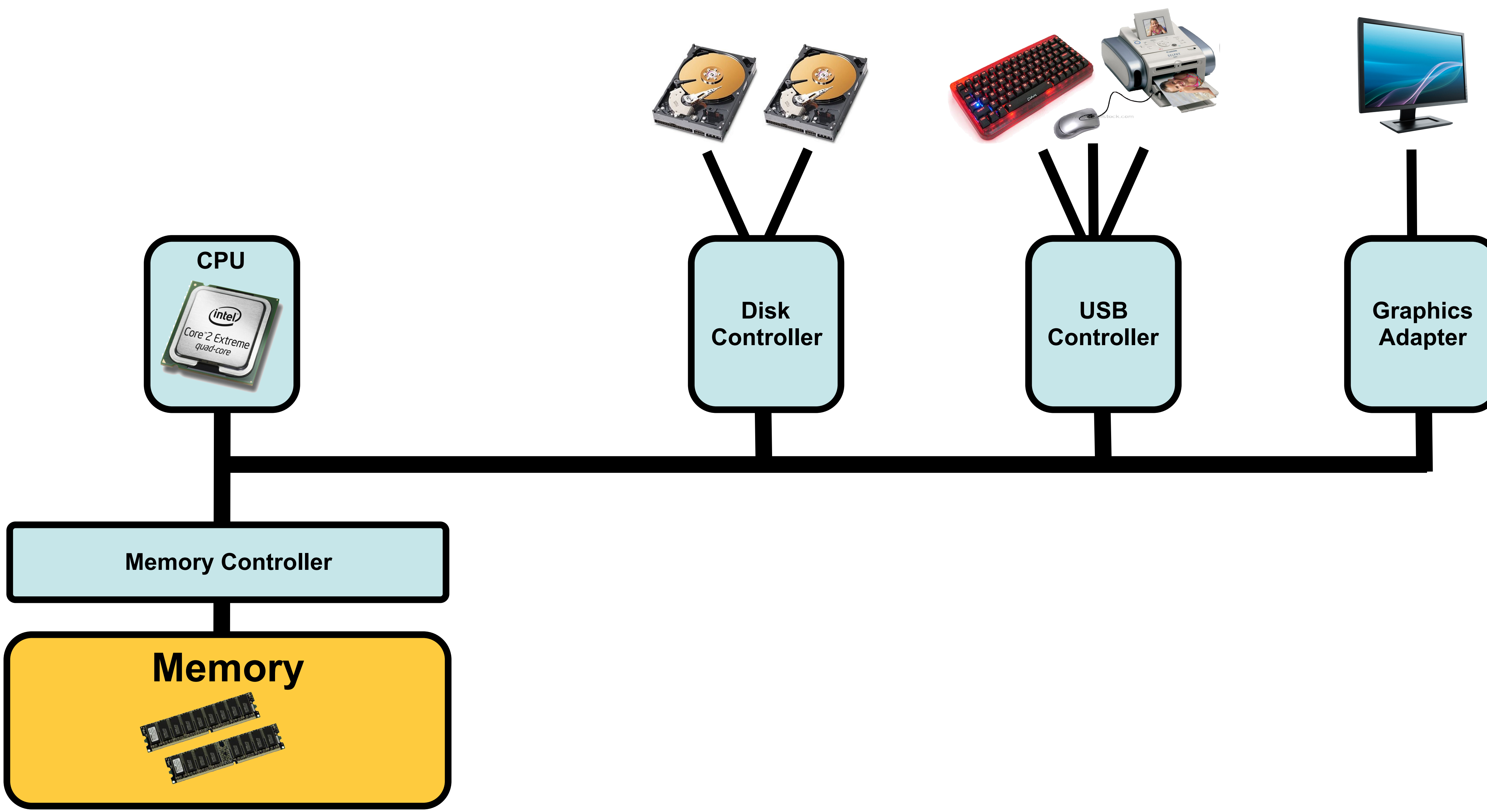
Similar to the addition circuit, circuits can be constructed for other ALU operations.





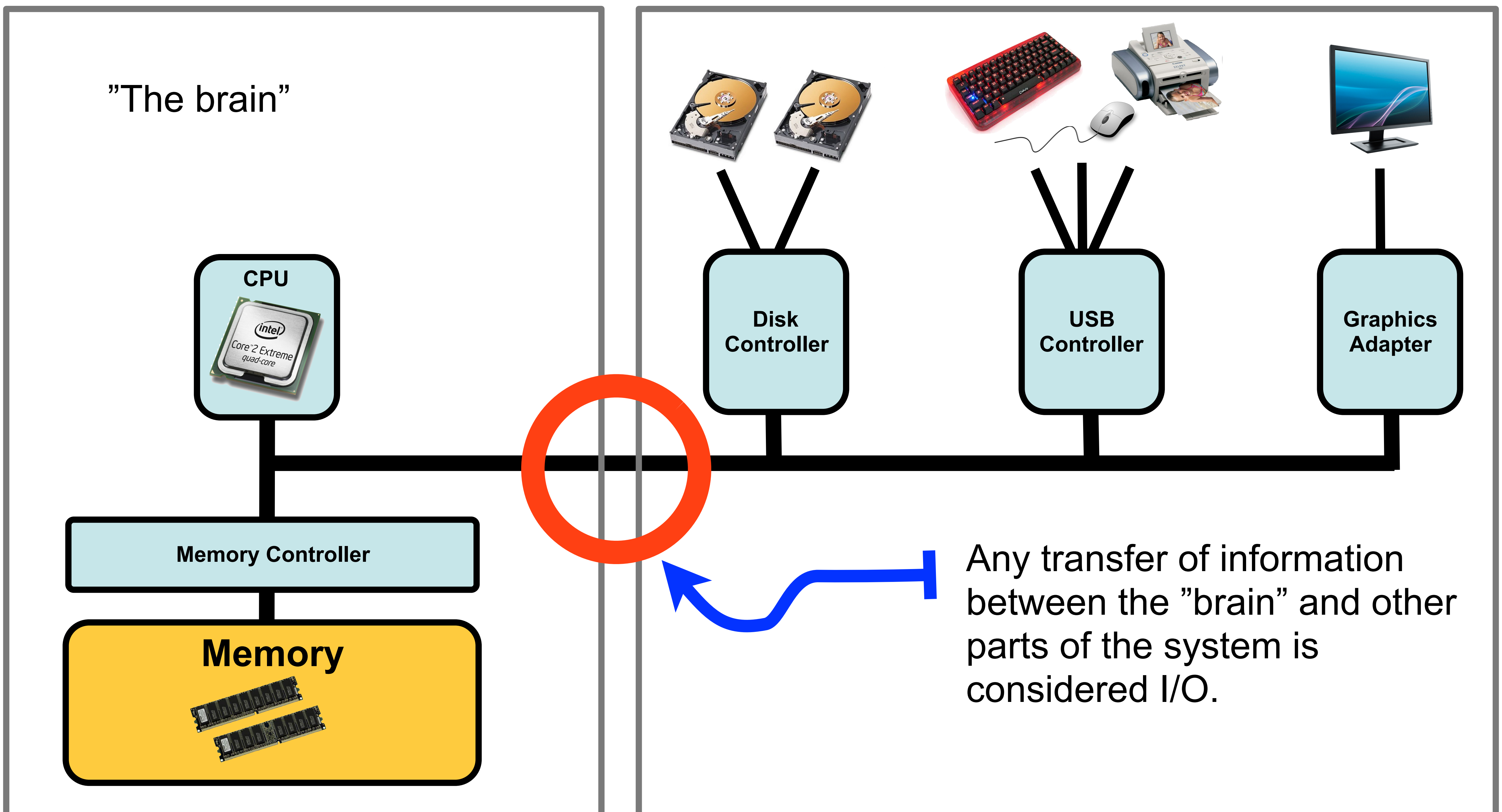
The von Neuman modell

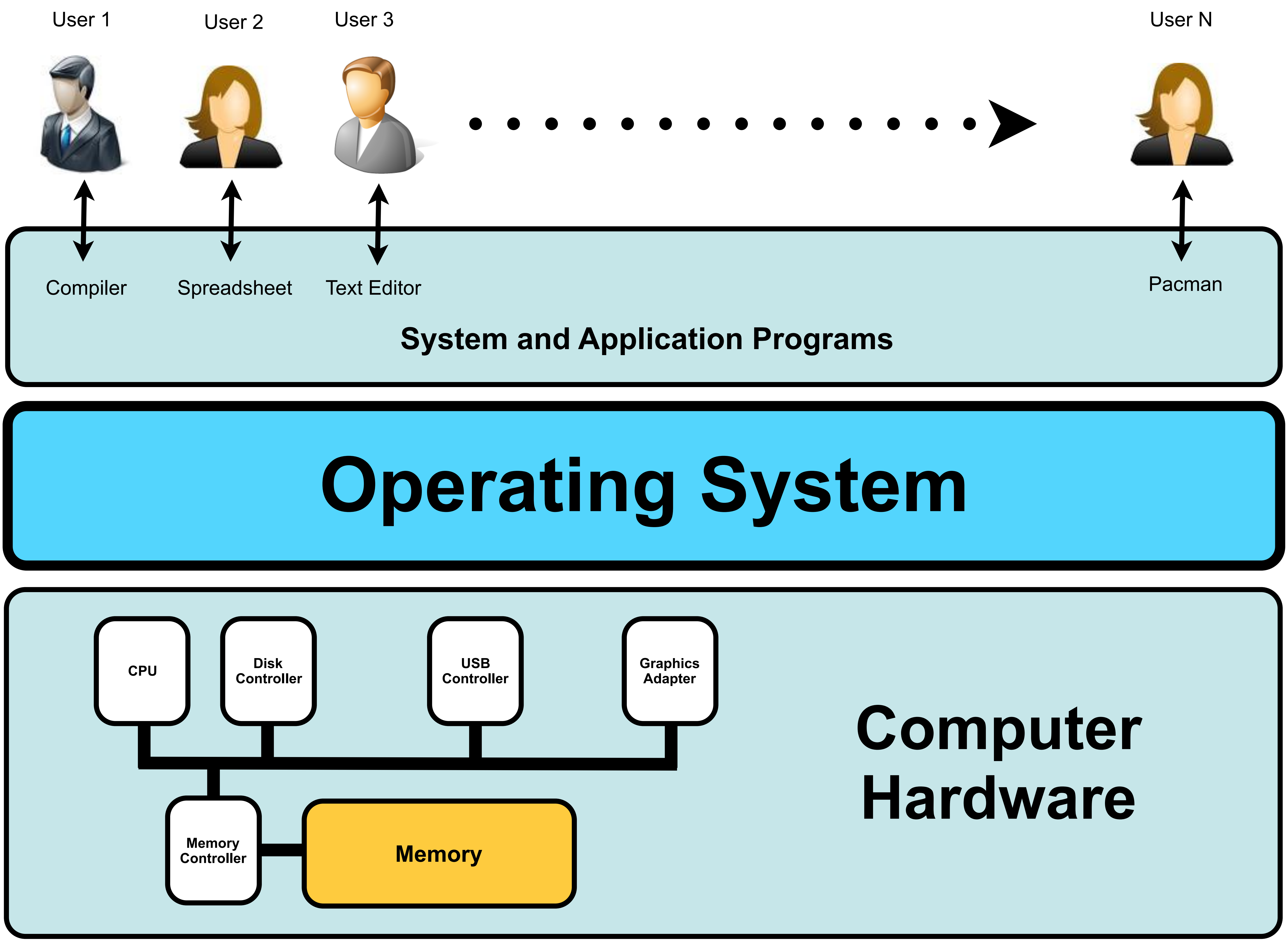
A typical modern computer system

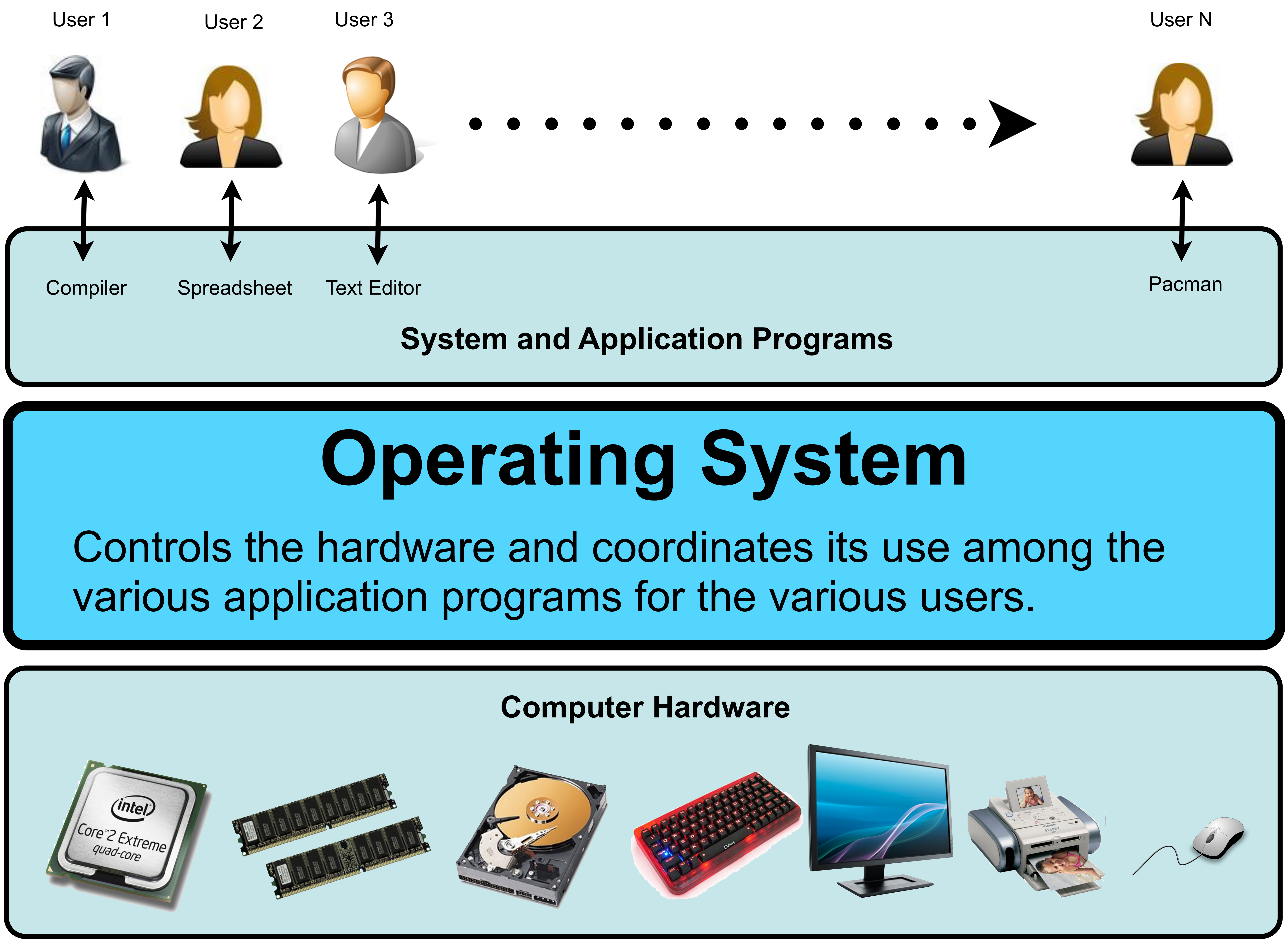


Input and Output (I/O)

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

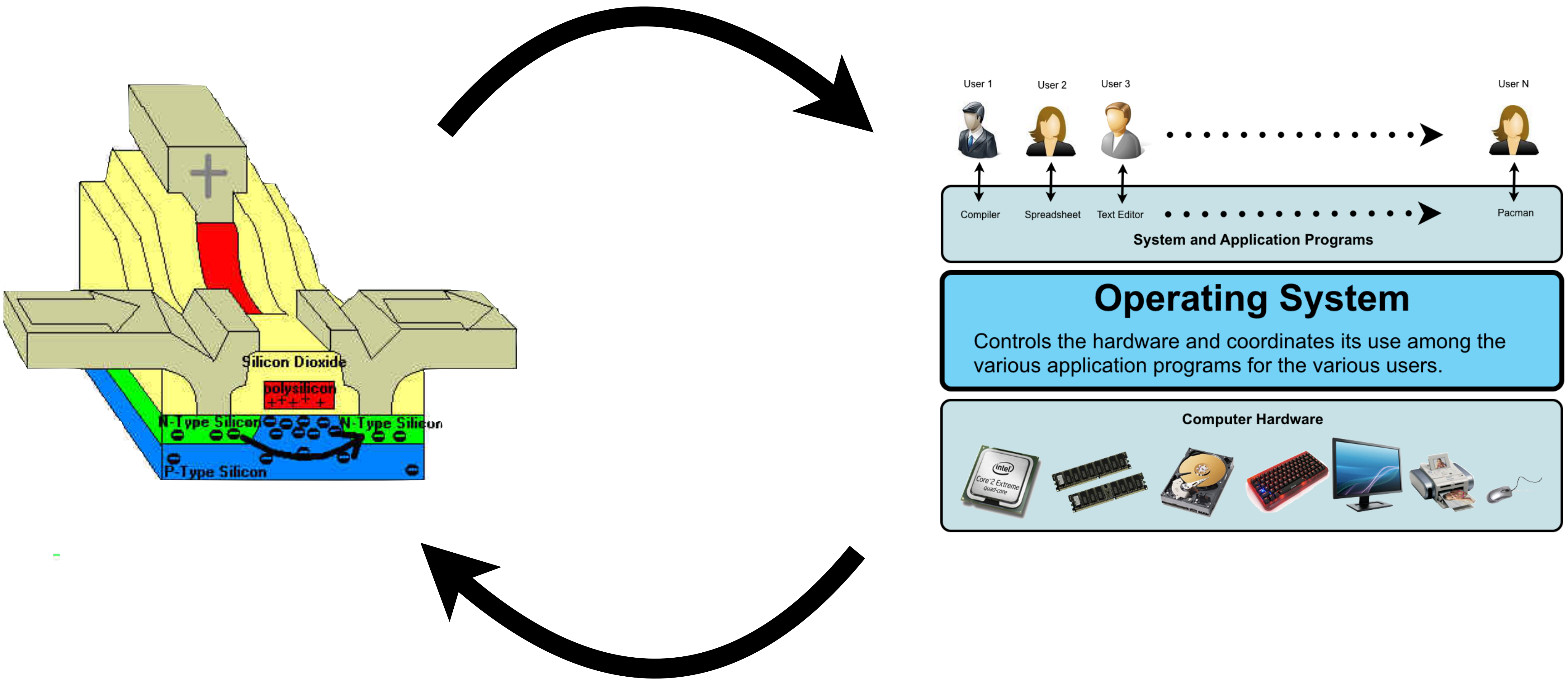



User 1
User 2
User 3
User N
Compiler
Spreadsheet
Text Editor
Pacman
System and Application Programs
Operating System
CPU
Disk Controller
USB Controller
Graphics Adapter
Memory Controller
Memory
Computer Hardware


User 1
User 2
User 3
User N
Compiler
Spreadsheet
Text Editor
Pacman
System and Application Programs
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

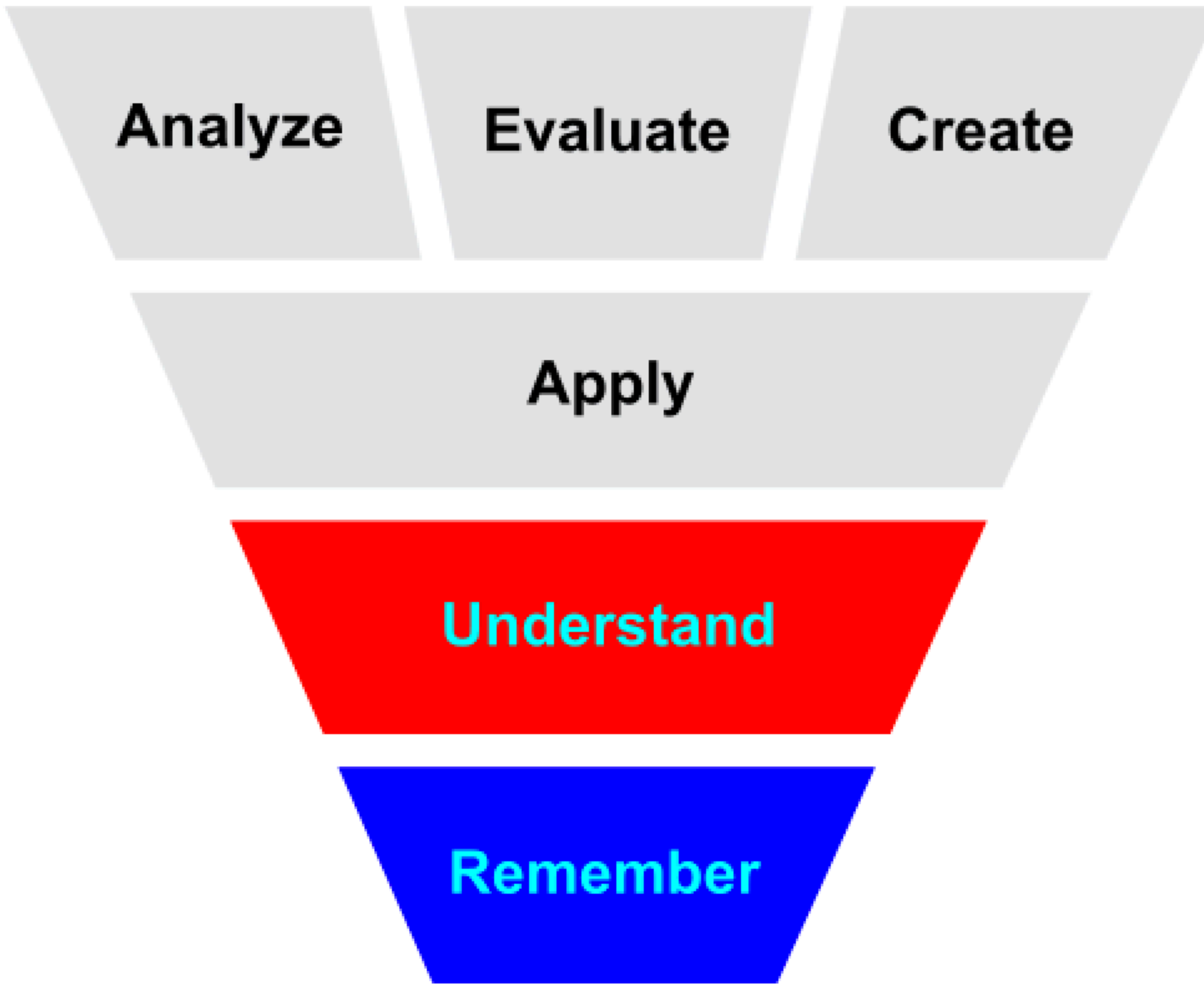


Image adopted from http://commons.wikimedia.org/wiki/File:BloomsCognitiveDomain.svg 2016-01-17

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	2018-01-16
	https://en.wikipedia.org/wiki/Processor_register	2018-01-16
	https://en.wikipedia.org/wiki/Computer_memory	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.

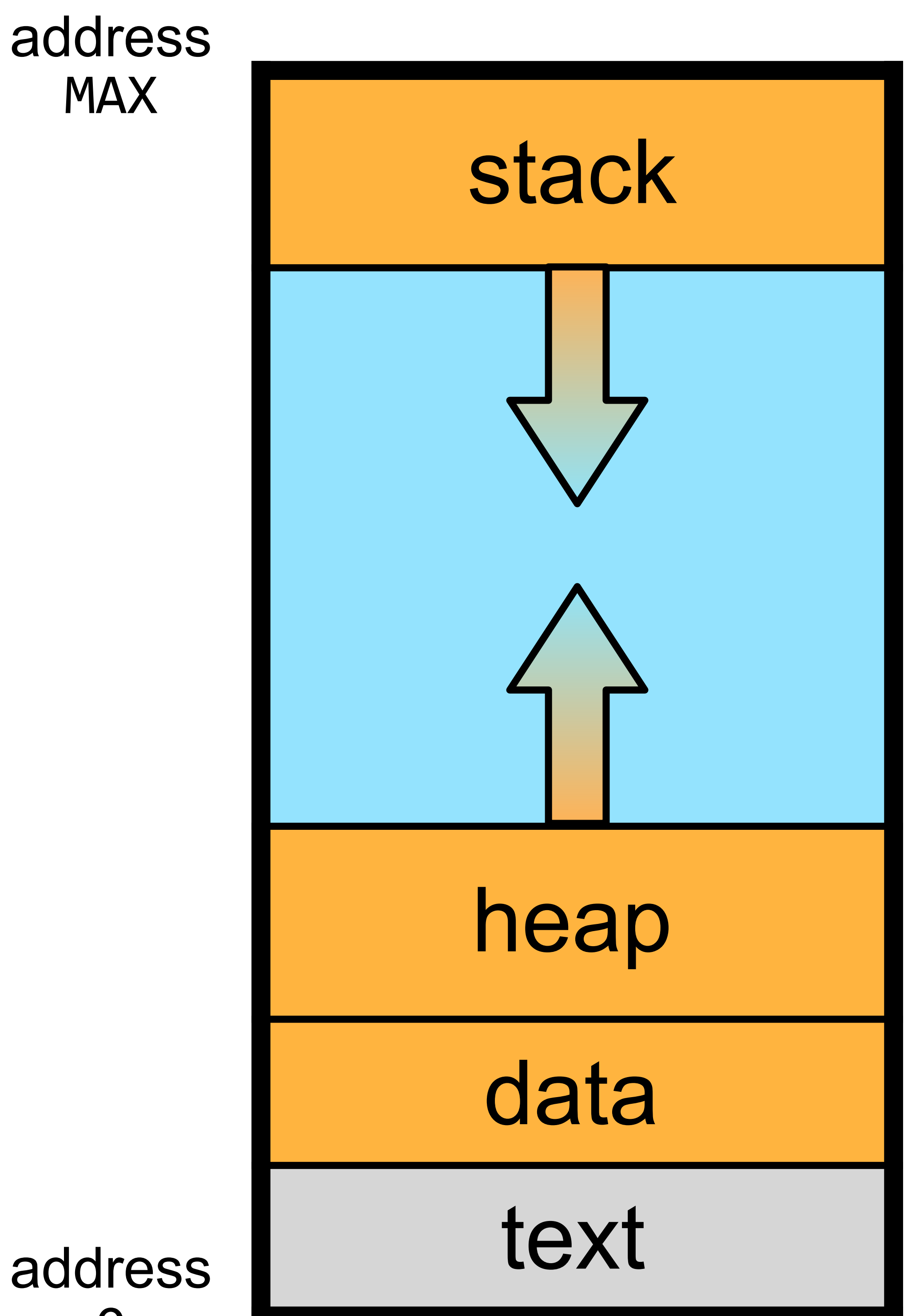
Process

An executable 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.

Process memory space

The operating system must allocate a new **memory space** for each new process. This memory space is often referred to as the process **memory image**.

A process in memory



Stack: Temporary data such as function parameters, local variables and return addresses.

The stack grows from high addresses towards lower address.

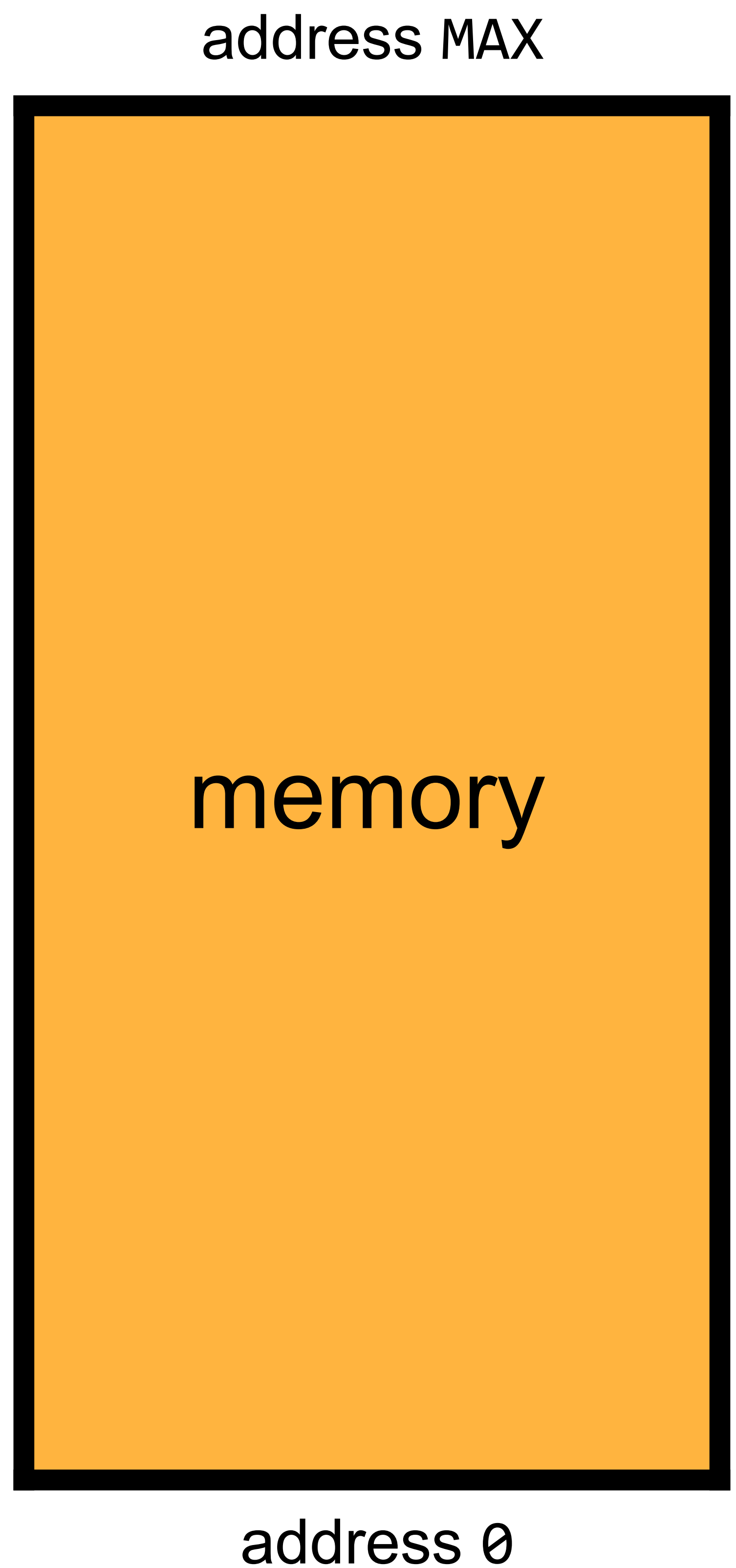
Heap: Dynamically allocated (malloc) by the program during runtime.

The heap grows from low addresses towards higher addresses.

Data: Statically (known at compile time) global variables and data structures.

Text: The program executable machine instructions.

Static memory allocation



The operating system must allocates a blob of memory for the new process memory image.

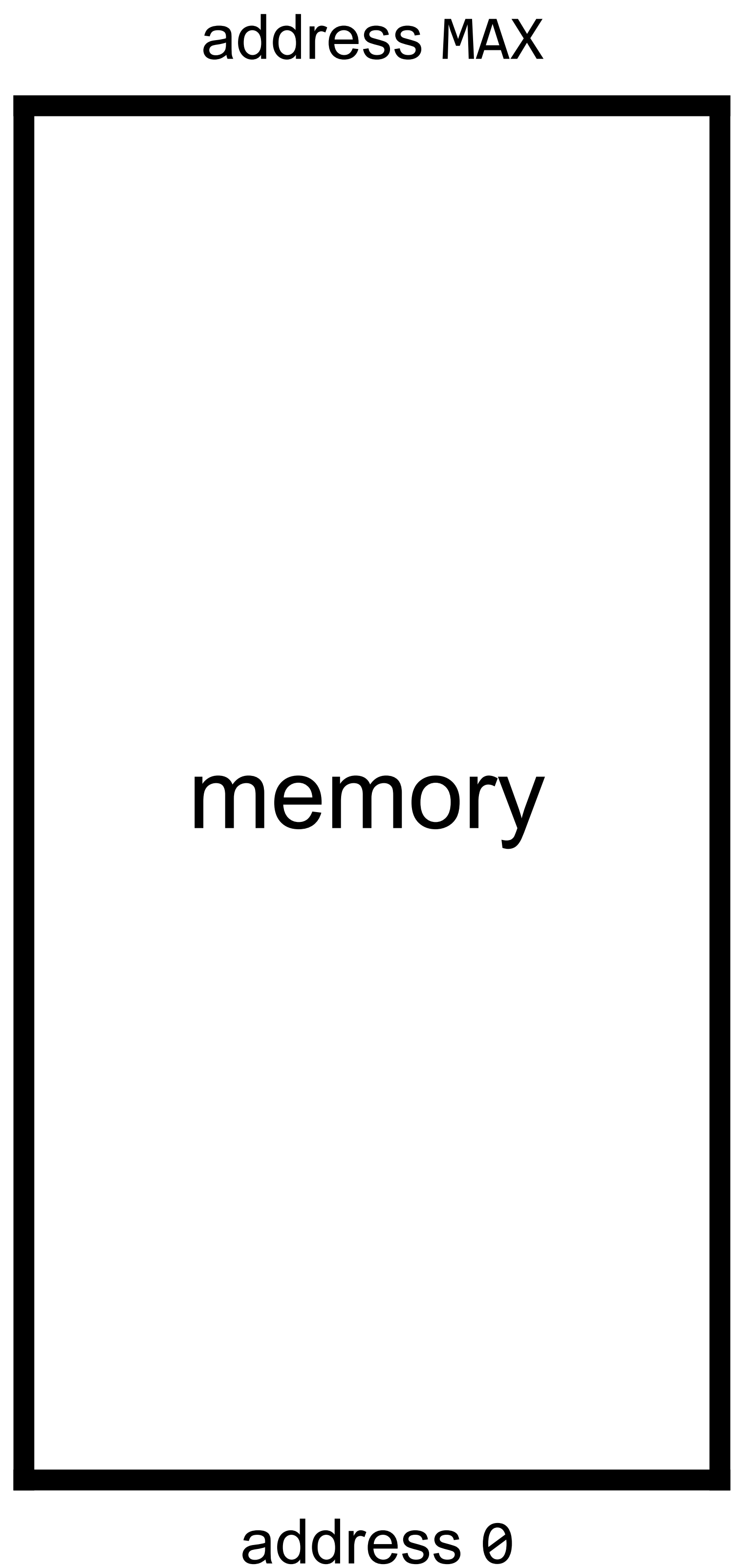
Static memory allocation

```c
// Example C program

#include <stdlib.h>

int x;

int main(void) {
  int y;
  char* str;
  str = malloc(50);
  exit(EXIT_SUCCESS);
}
```



The operating system allocates a blob of memory for the new process.

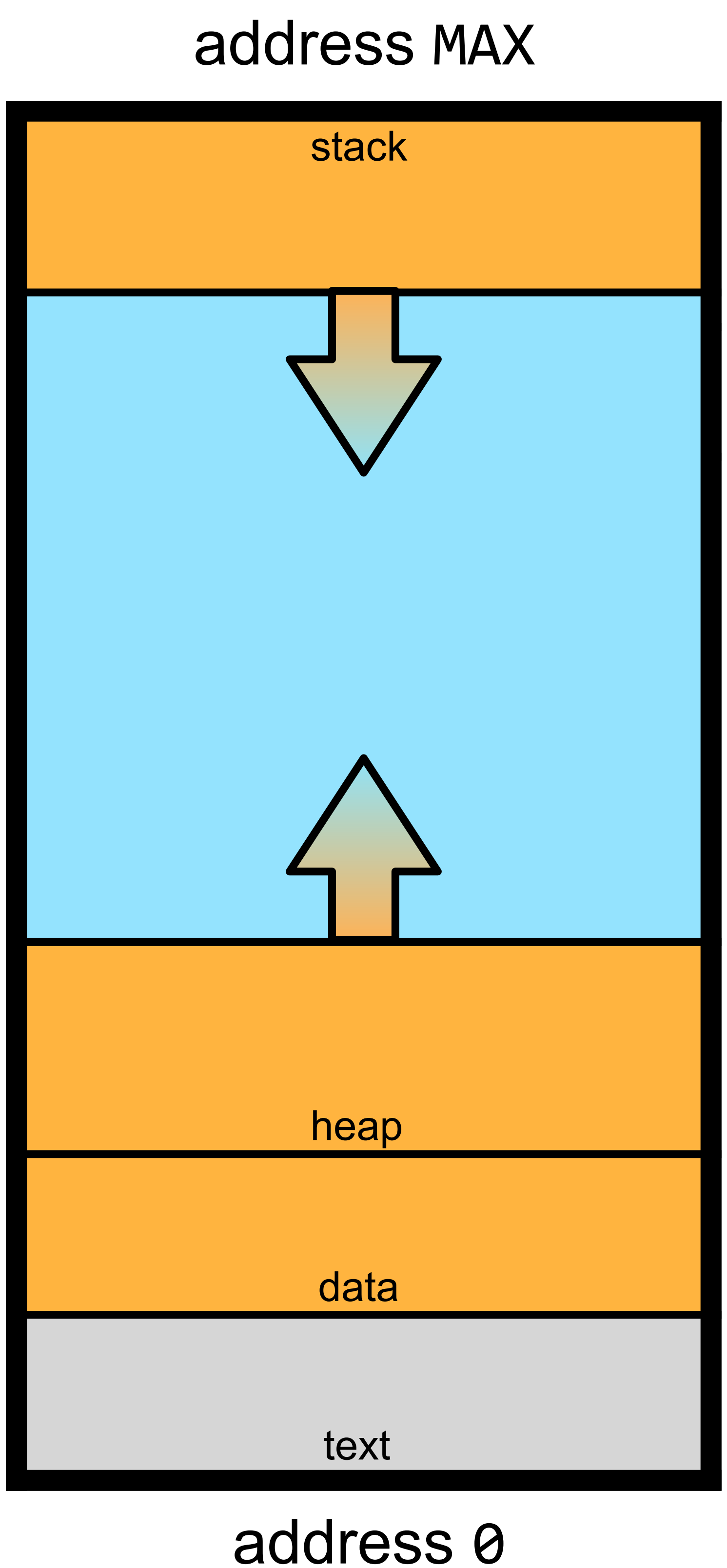
Static memory allocation

```c
// Example C program

#include <stdlib.h>

int x;

int main(void) {
  int y;
  char* str;
  str = malloc(50);
  exit(EXIT_SUCCESS);
}
```



The allocated memory is divided into the following segments:

- text
- data
- heap
- stack

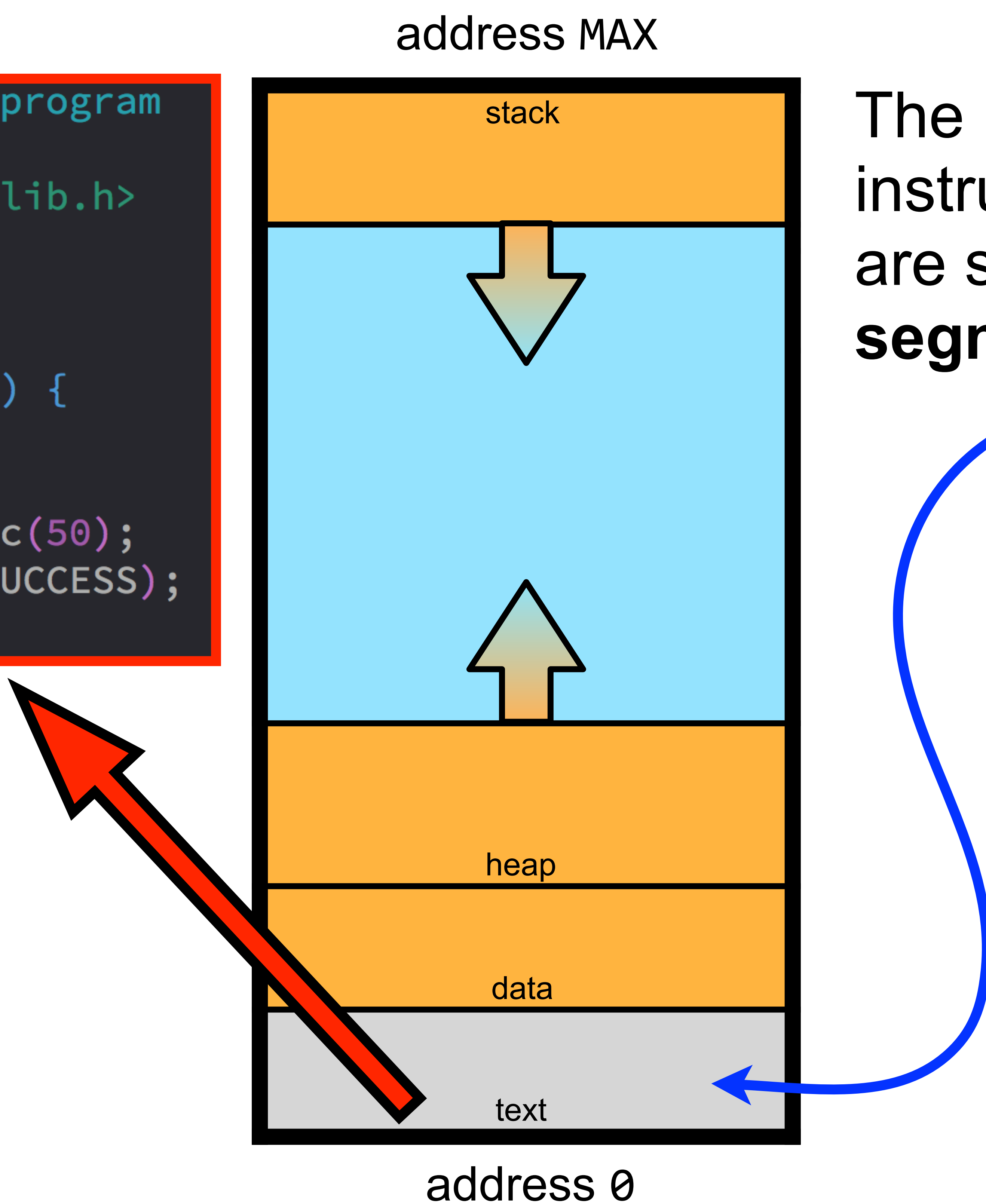
Static memory allocation

```c
// Example C program

#include <stdlib.h>

int x;

int main(void) {
  int y;
  char* str;
  str = malloc(50);
  exit(EXIT_SUCCESS);
}
```



The machine code (the instructions to be executed) are stored in the **text segment**.

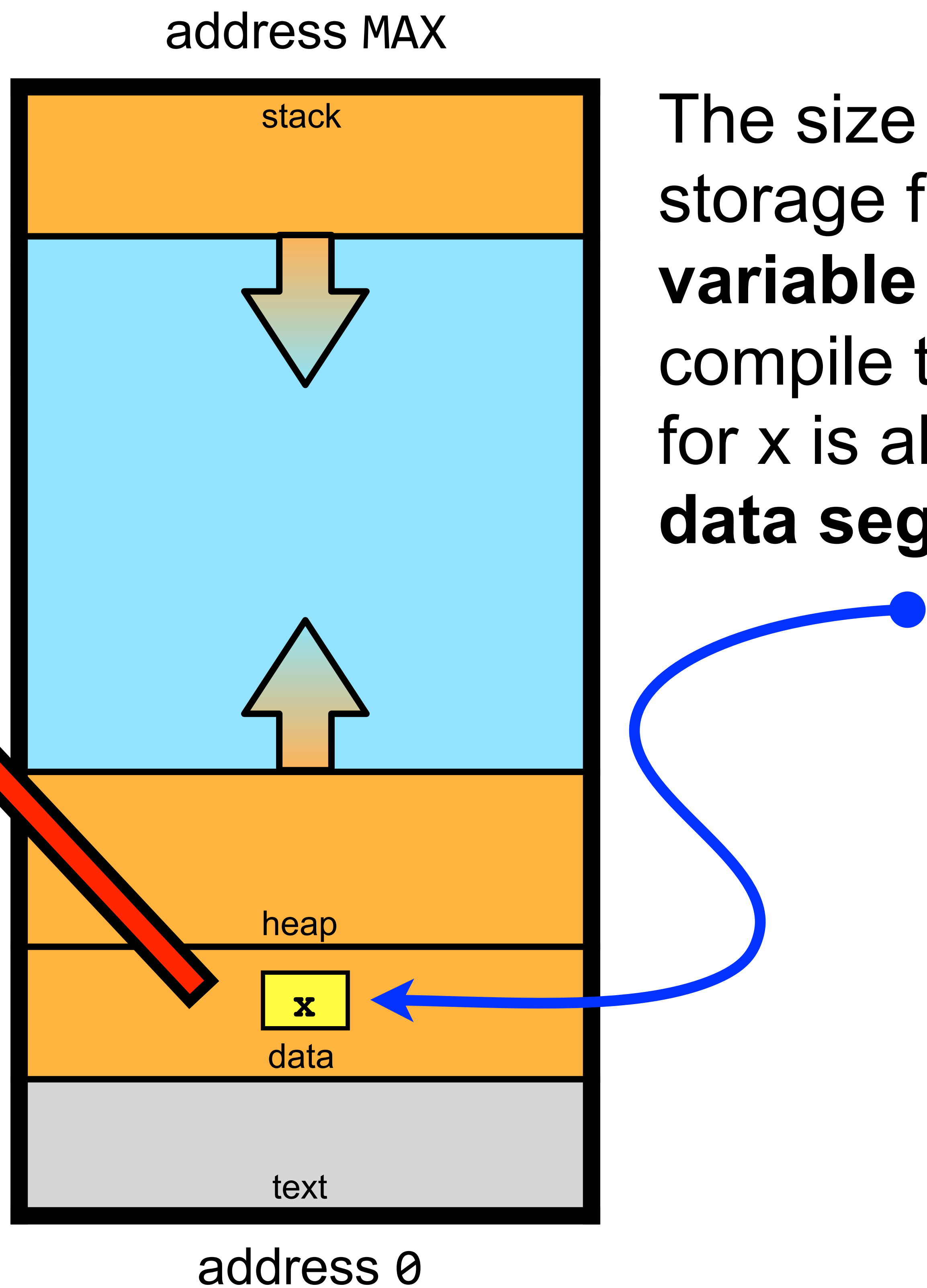
Static memory allocation

```c
// Example C program

#include <stdlib.h>

int x;

int main(void) {
  int y;
  char* str;
  str = malloc([illegible]);
  exit(EXIT_SUCCESS);
}
```



The size of the needed storage for the **global variable `x`** is known at compile time and storage for x is allocated in the static **data segment**.

Dynamic memory allocation

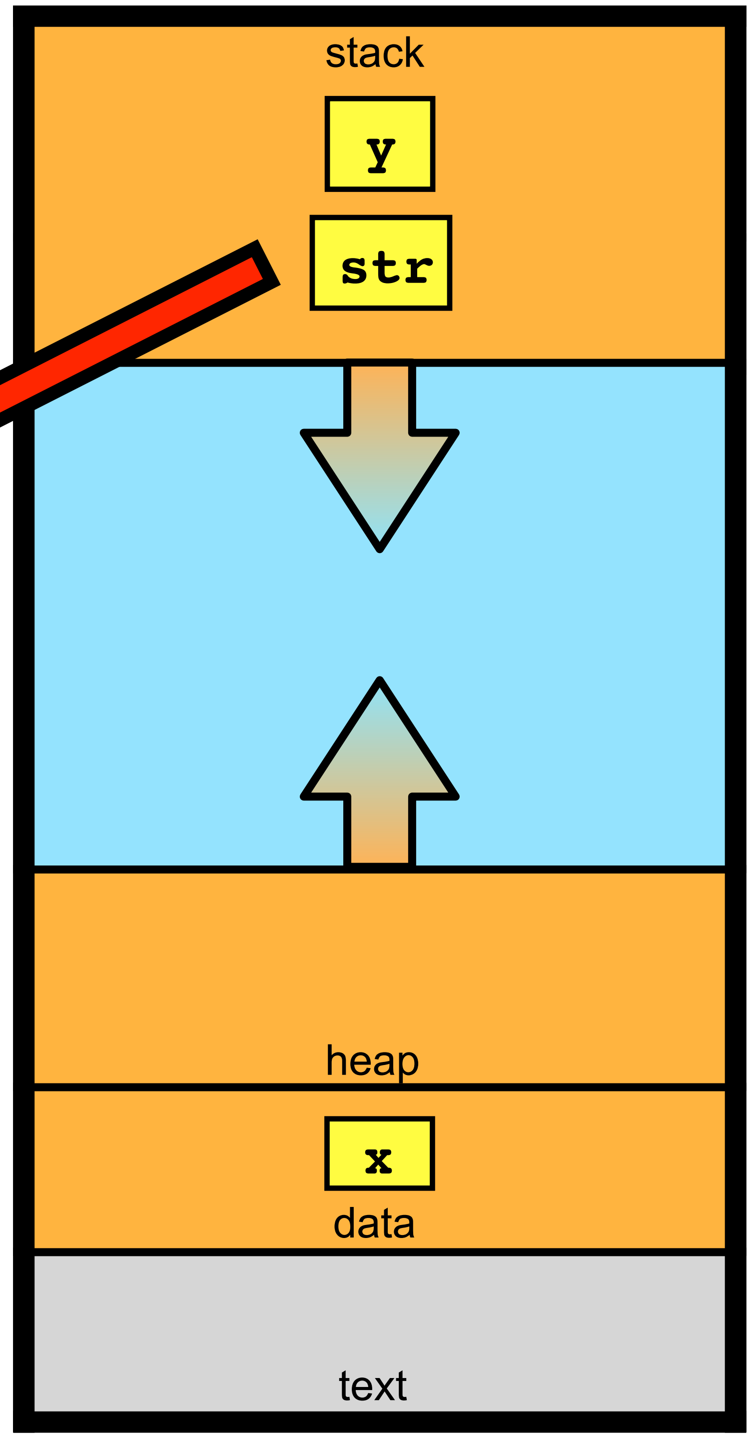
```c
// Example C program

#include <stdlib.h>

int x;

int main(void) {
  int y;
  char* str;
  str = malloc(50);
  exit(EXIT_SUCCESS);
}
```

address MAX



address 0

The size of the needed storage for the **local variables** **`y`** and **`str`** are also known at compile time but these are only needed within main and storage is allocated on the **stack**.

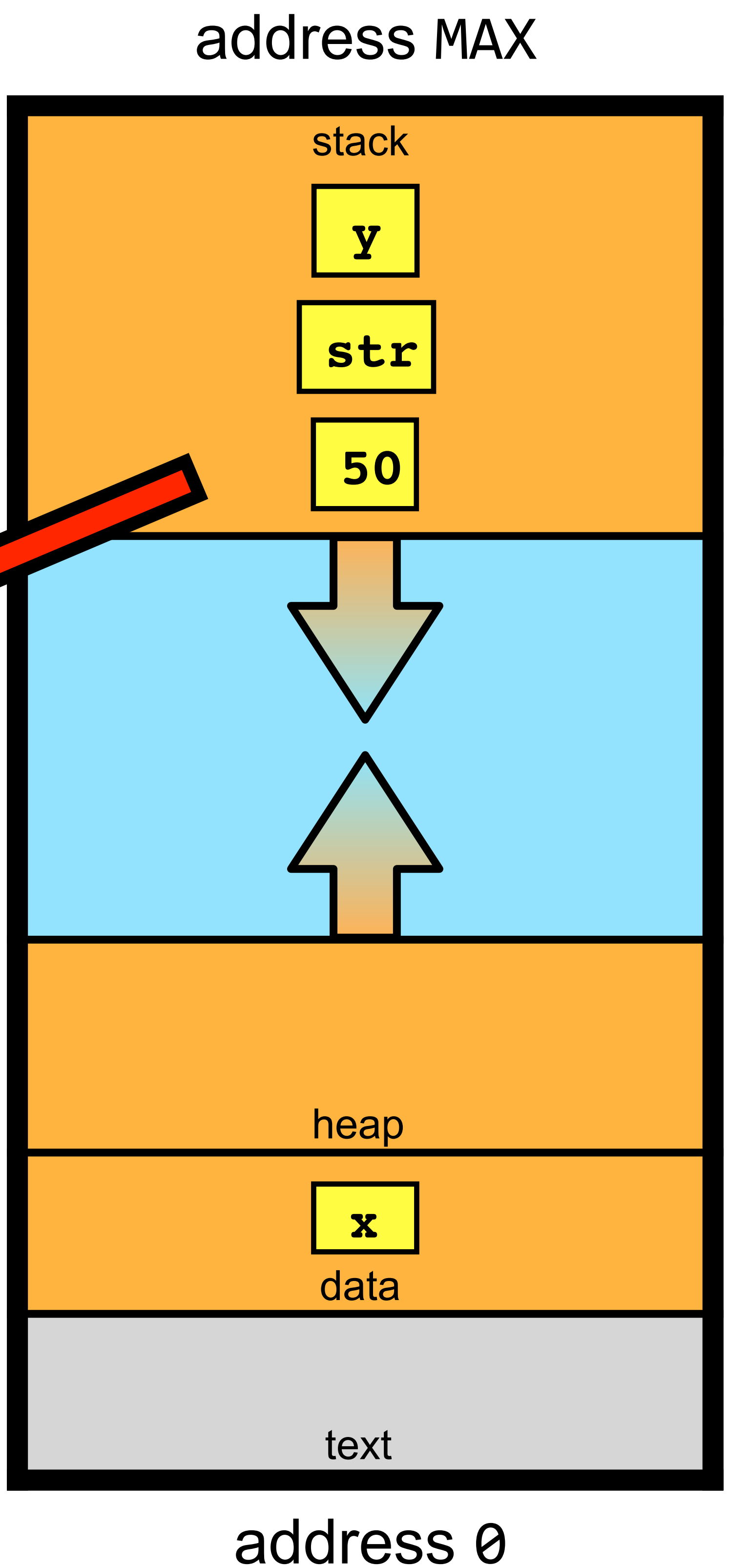
Dynamic memory allocation

```c
// Example C program

#include <stdlib.h>

int x;

int main(void) {
  int y;
  char* str;
  str = malloc(50);
  exit(EXIT_SUCCESS);
}
```



In general, the value of the **parameter** to the malloc might not be know at compile time. In this example the argument to malloc, the number 50 is pushed onto the **stack**.

Note: A compiler may also decide to put arguments in certain registers.

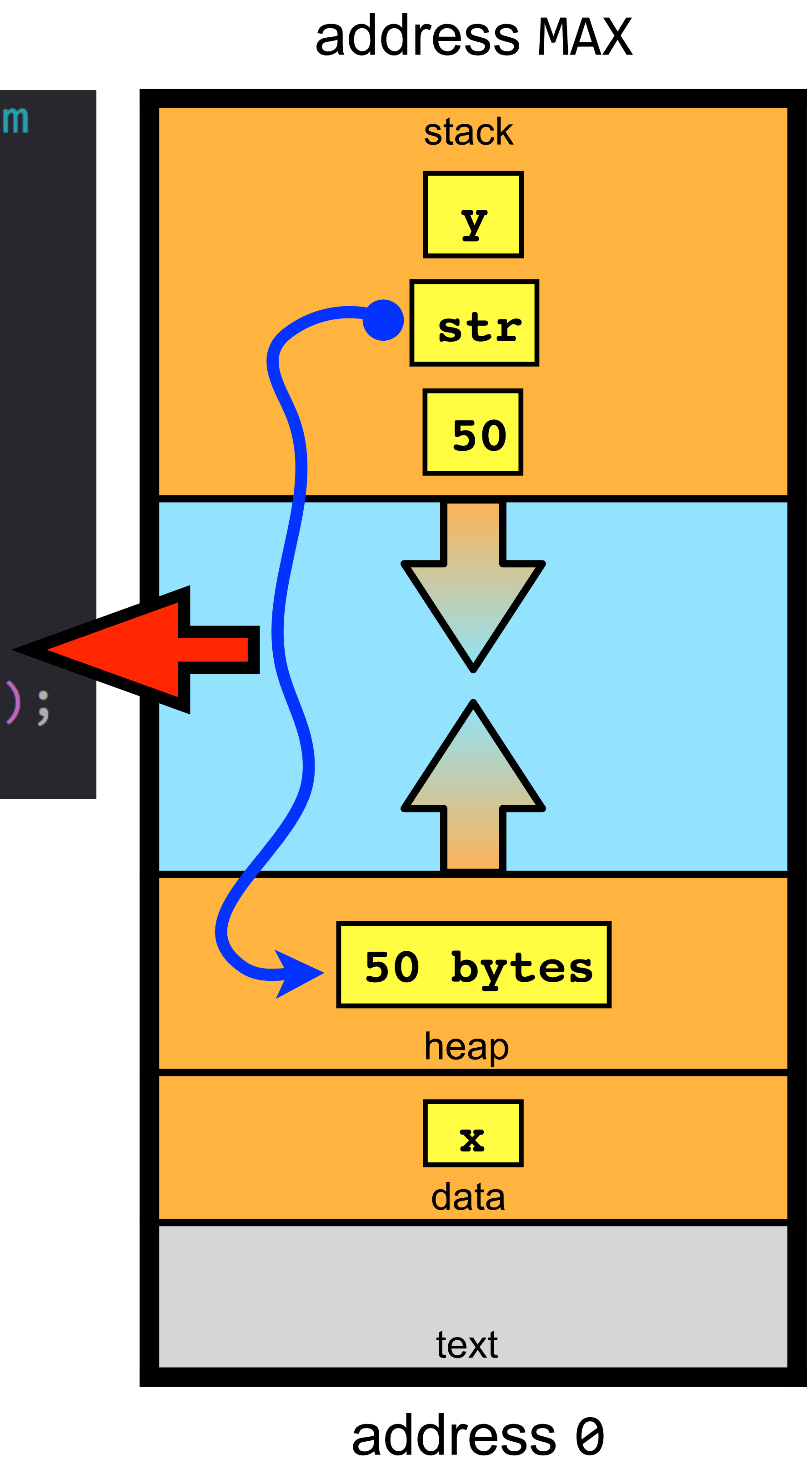
Dynamic memory allocation

```c
// Example C program

#include <stdlib.h>

int x;

int main(void) {
  int y;
  char* str;
  str = malloc(50);
  exit(EXIT_SUCCESS);
}
```



Now **`malloc`** has **dynamically** allocated memory on the heap.

The variable **`str`** stores the address to the first of the 50 bytes allocated on the heap, i.e., **`str`** is a **pointer**.

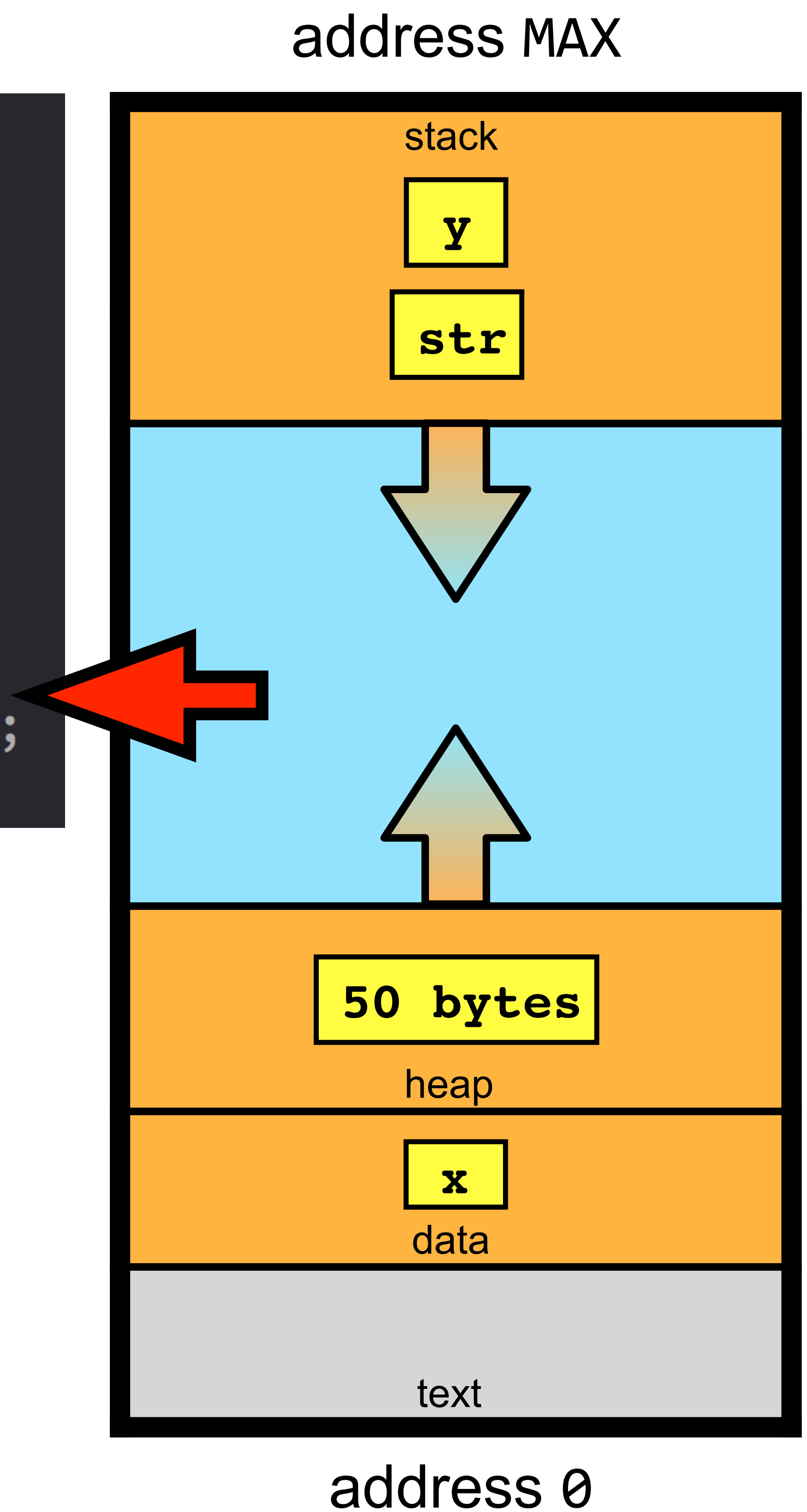
Dynamic memory deallocation

```c
// Example C program

#include <stdlib.h>

int x;

int main(void) {
  int y;
  char* str;
  str = malloc(50);
  exit(EXIT_SUCCESS);
}
```



When malloc returns, the value 50, the argument used when calling malloc is no longer needed and is popped from the stack.

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.

Source https://en.wikipedia.org/wiki/Kernel_(operating_system) 2018-01-16

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.

Kernel

Bootstrap firmware

Computer Hardware

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.

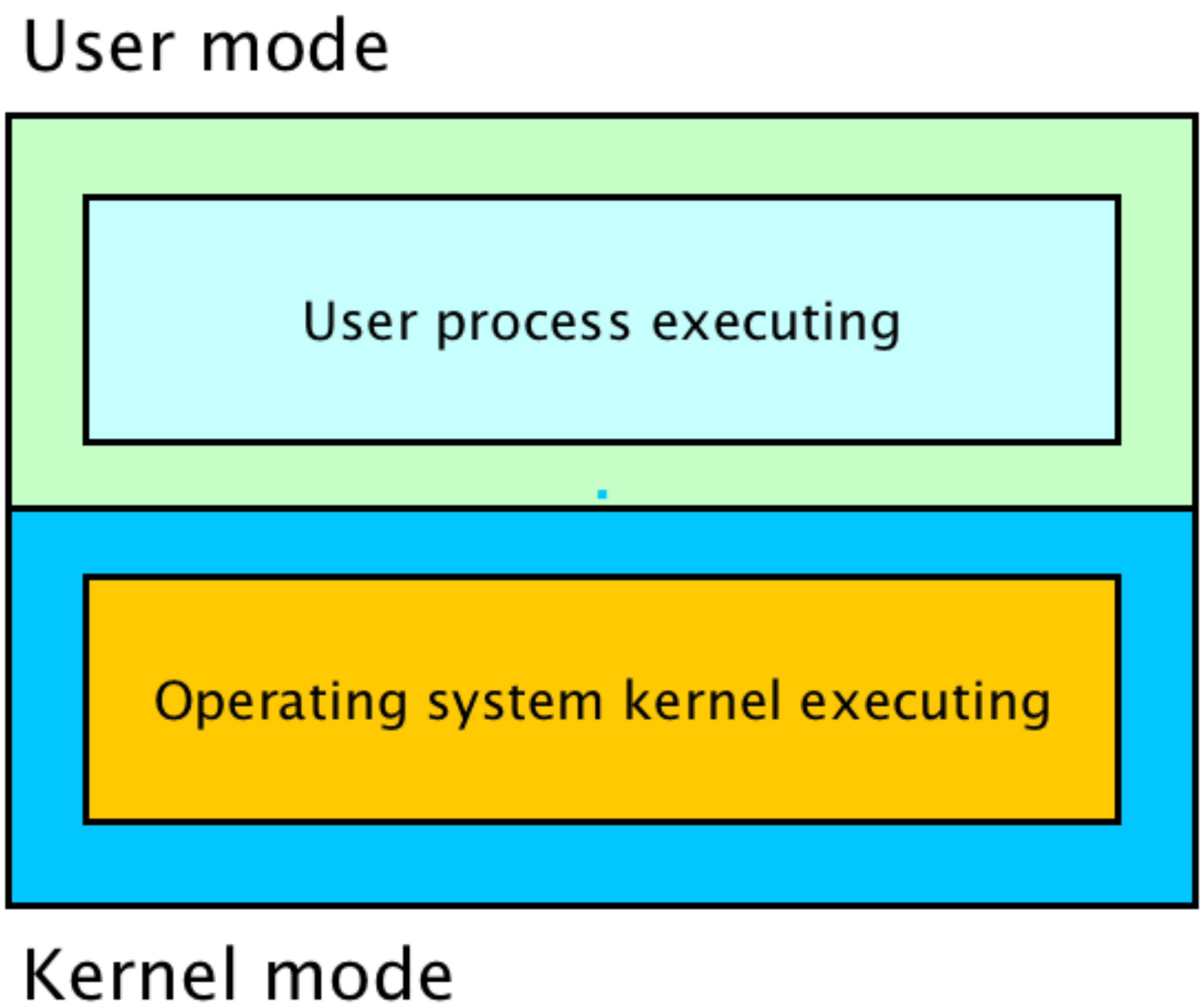
On boot, starts executing the first process such as **init**.

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

Dual mode operation

Dual mode operation

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

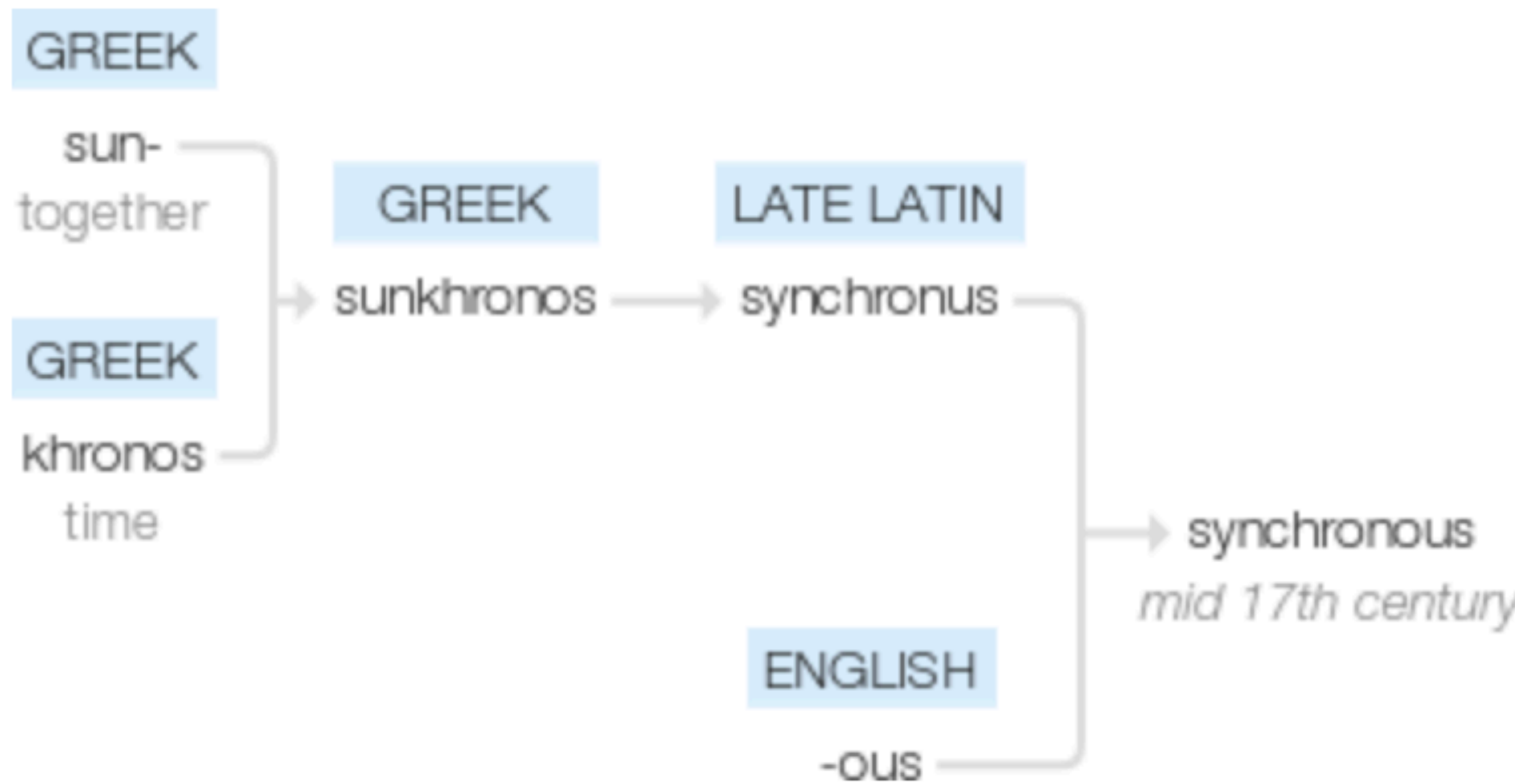


Dual mode operation place restrictions on the type and scope of operations that can be executed by the CPU.

This design allows the operating system kernel to execute with more privileges than user application processes.

Synchronous and asynchronous

Synchronous



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

Synchronous and asynchronous events

Synchronous means happening, existing, or arising at precisely the same time. [1]

> 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.

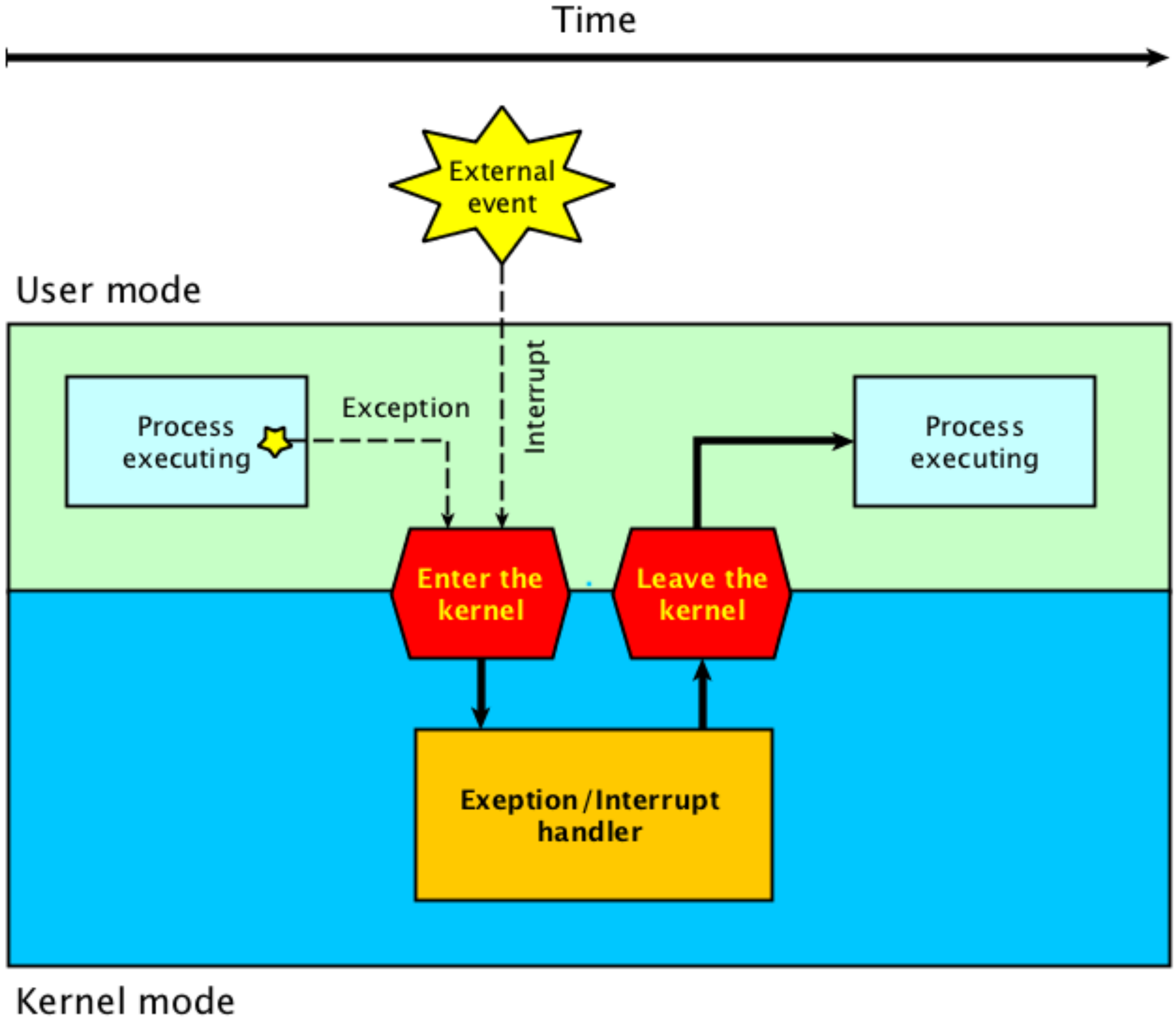
Interrupts are external and asynchronous

- ★ 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.

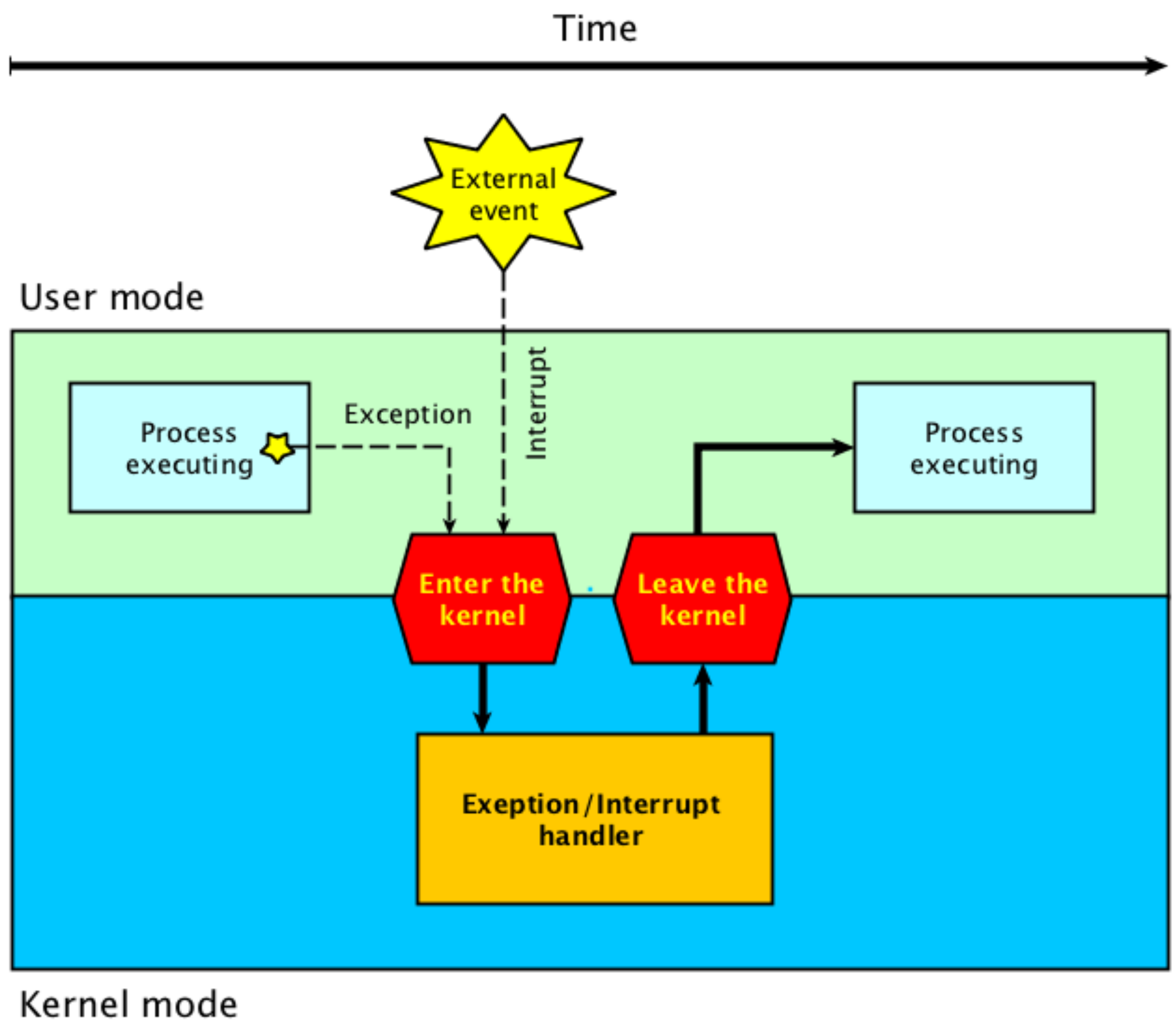
- ★ Read and write requests to disk is similar to key presses. The disk controller is external to the executing process and the timing of a disk operation might vary even if the same program is executed several times.

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.


Time
External event
User mode
Interrupt
Process executing
Exception
Process executing
Enter the kernel
Leave the kernel
Exeption/Interrupt handler
Kernel mode

Problem?


Time
External event
User mode
Process executing
Exception
Interrupt
Process executing
Enter the kernel
Leave the kernel
Exeption/Interrupt handler
Kernel mode

Problem

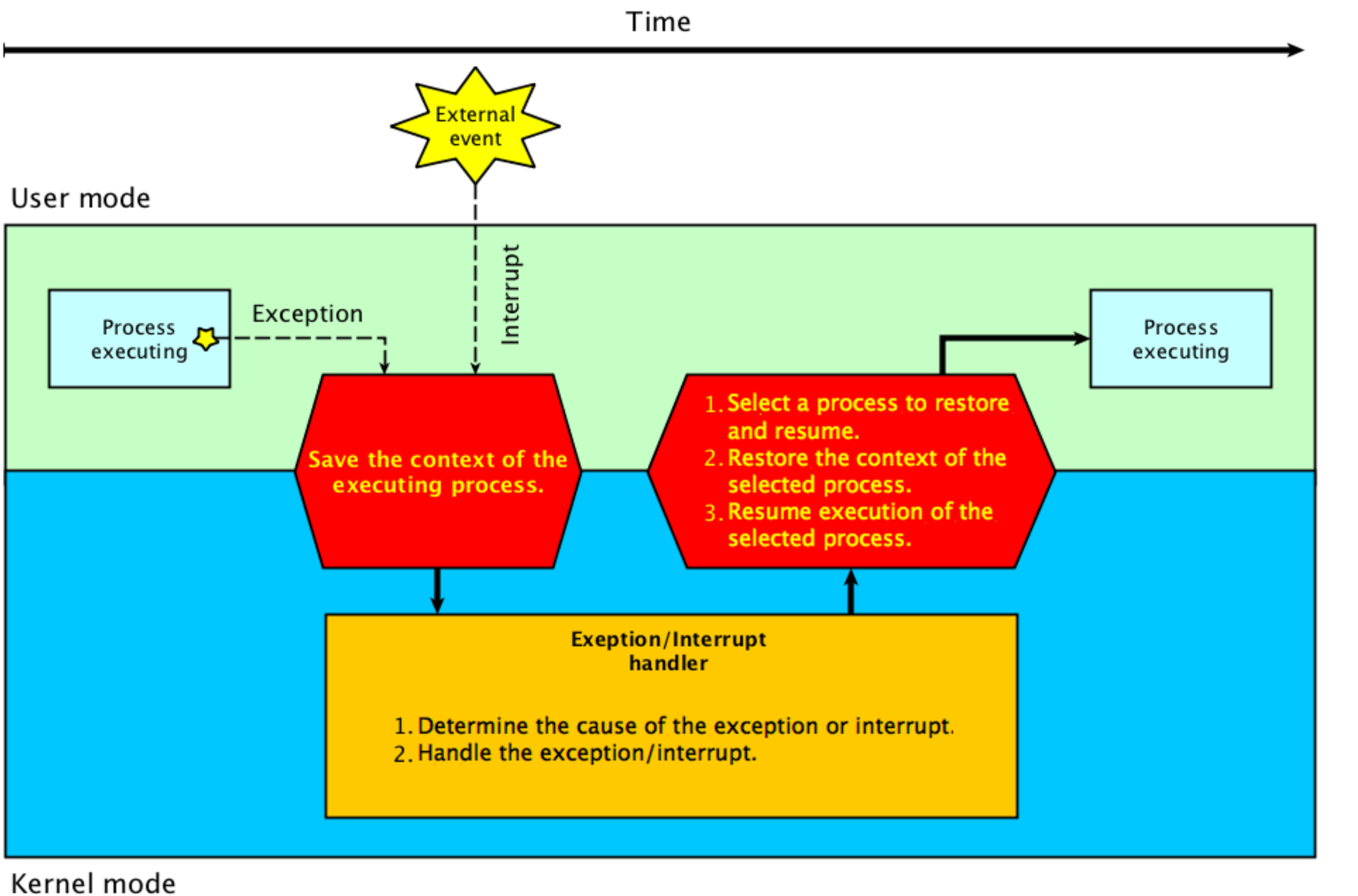
The exception/interrupt handler uses the same CPU (including the program counter and all registers) as the user processes.

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.

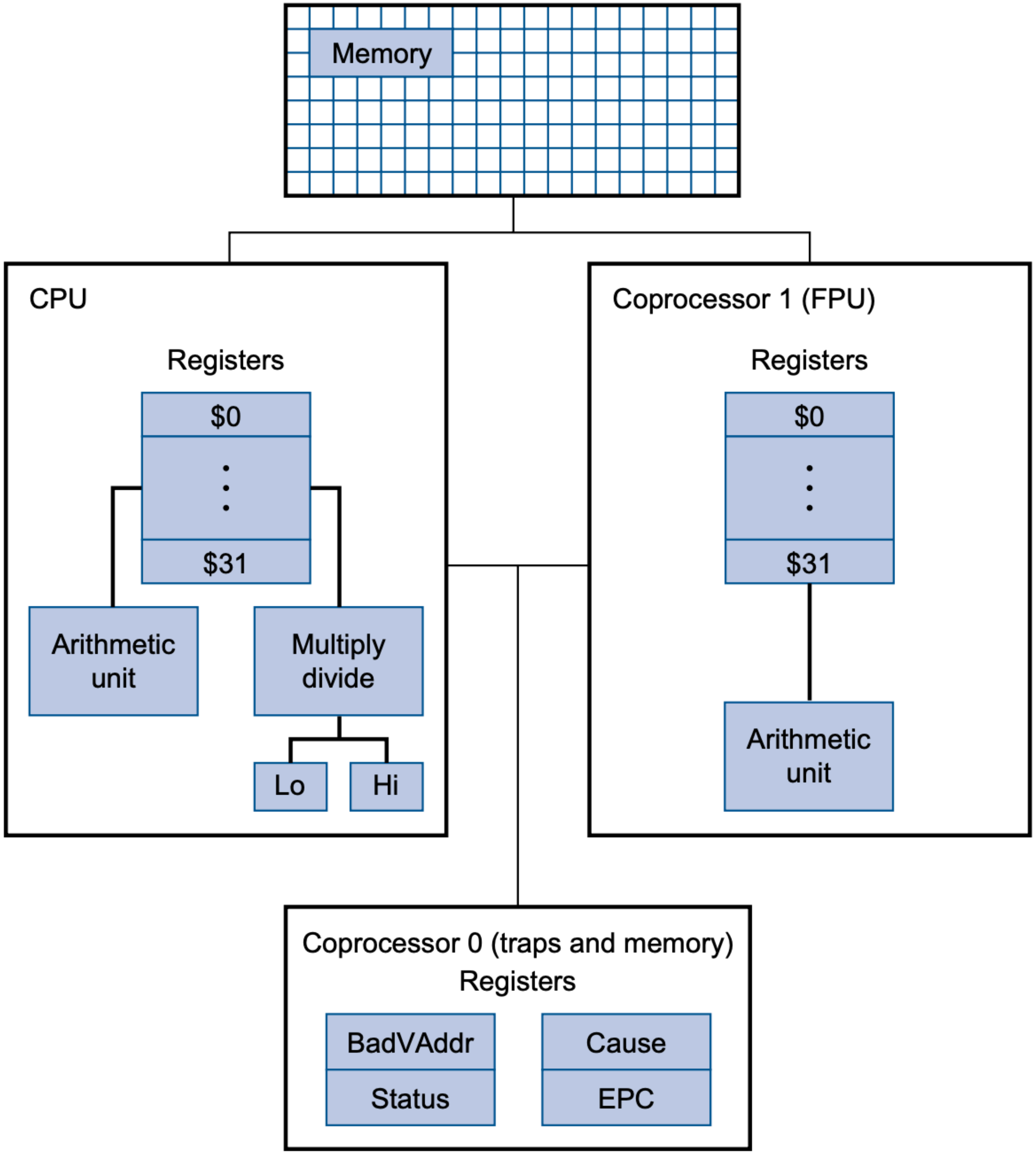
Restoring cpu context

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



Time
External event
User mode
Interrupt
Process executing
Exception
Process executing
Save the context of the executing process.
1. Select a process to restore and resume.
2. Restore the context of the selected process.
3. Resume execution of the selected process.
Exeption/Interrupt handler
1. Determine the cause of the exception or interrupt.
2. Handle the exception/interrupt.
Kernel mode

Exceptions and interrupts in Mips



Memory
CPU
Registers
$0
$31
Arithmetic unit
Multiply divide
Lo
Hi
Coprocessor 1 (FPU)
Registers
$0
$31
Arithmetic unit
Coprocessor 0 (traps and memory)
Registers
BadVAddr
Cause
Status
EPC

When an exception or interrupts occurs in Mips, the following actions are **automatically** taken.

1. The value of the **program counte**r (PC) is **stored** in the **EPC** register in coprocessor 0.
2. The cause of the exception or interrupt is stored in the coprocessor 0 **CAUSE register**.
   a. In case of an exception, the **exception code** is stored in the CAUSE register as a 5 bit value.
   b. In case of an interrupt, one of the 8 **pending interrupt** bits in the CAUSE register is set to identify the type of interrupt.
3. The **program counter** is set to the start address of the exception/interrupt handler (the **kernel entry point**) at address 0x80000180.

What happens next is up to the code in the exception/interrupt handler.

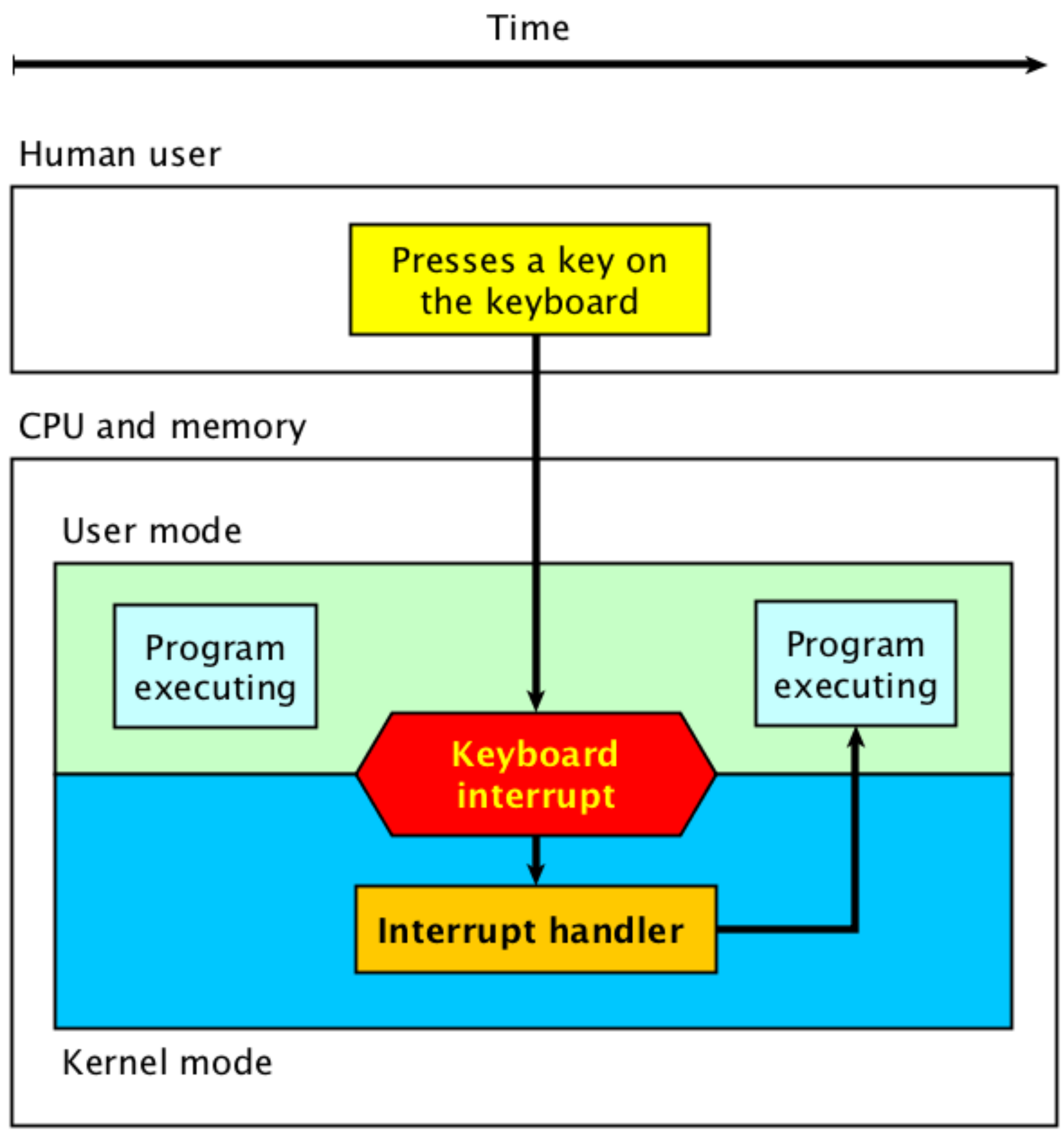
Waiting for keyboard input

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

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 the first job waits for the/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

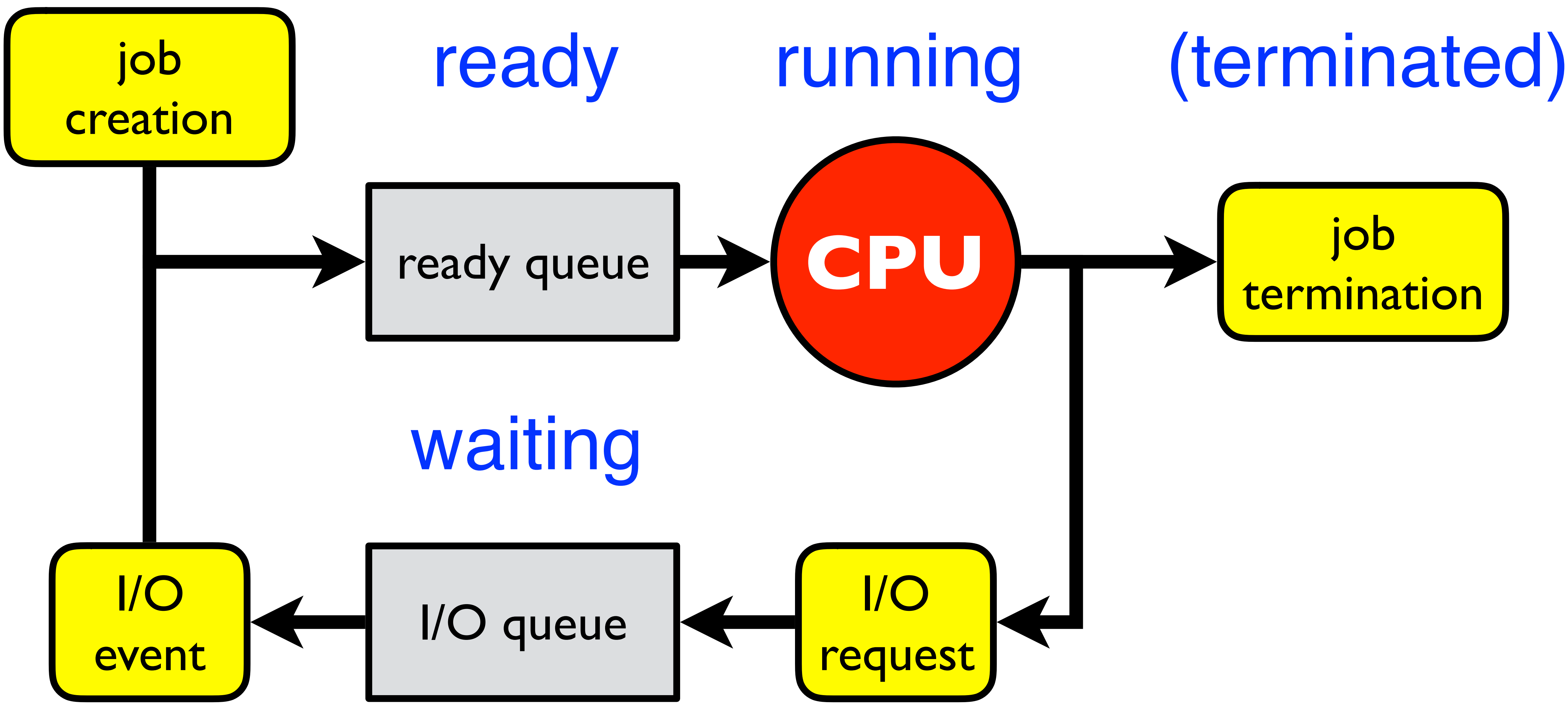
The job is ready to run but currently not selected to do so.

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

Job 1

Ready

Job 2

Ready

Job 3

Ready

Job 4

Ready

CPU

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

Memory

CPU

Job 1

Running

Job 2

Ready

Job 3

Ready

Job 4

Ready

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**

Job 1
Waiting

Job 2
Ready

Job 3
Ready

Job 4
Ready

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

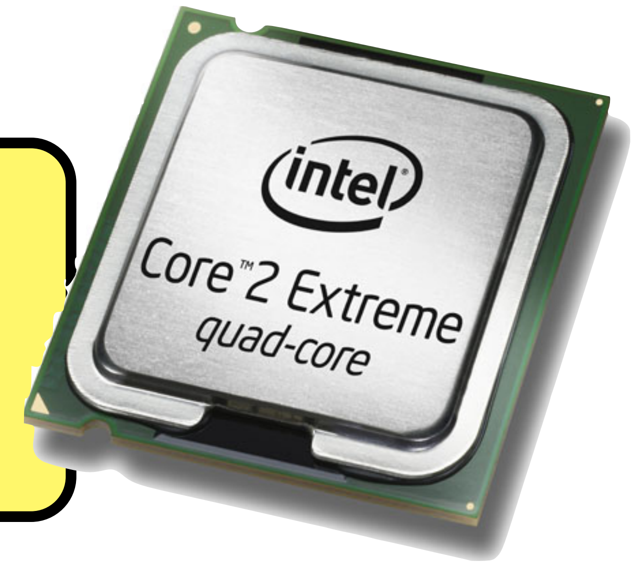
Ready

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

CPU

Job 1

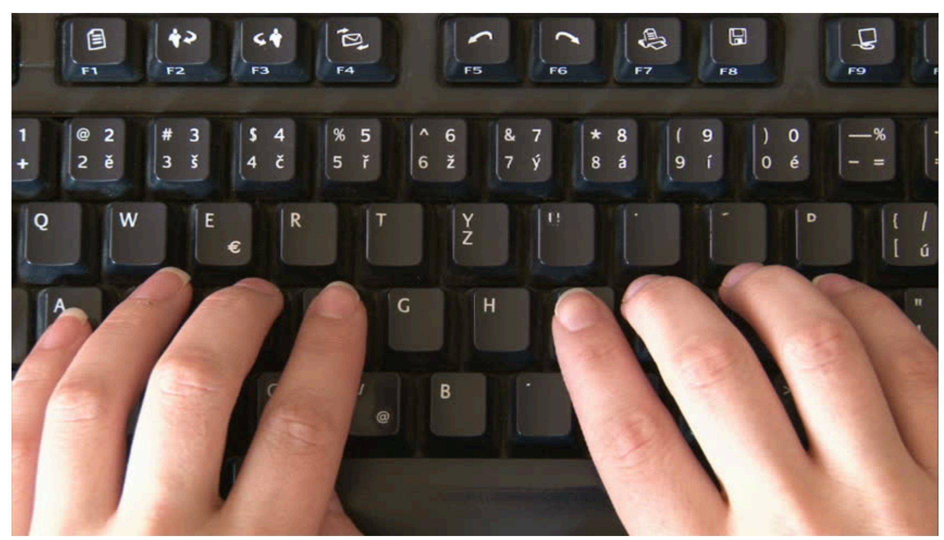
Waiting

Job 2

Ready

Job 3

Running



Interrupt

Job 4

Ready

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

Job 1

Ready

Job 2

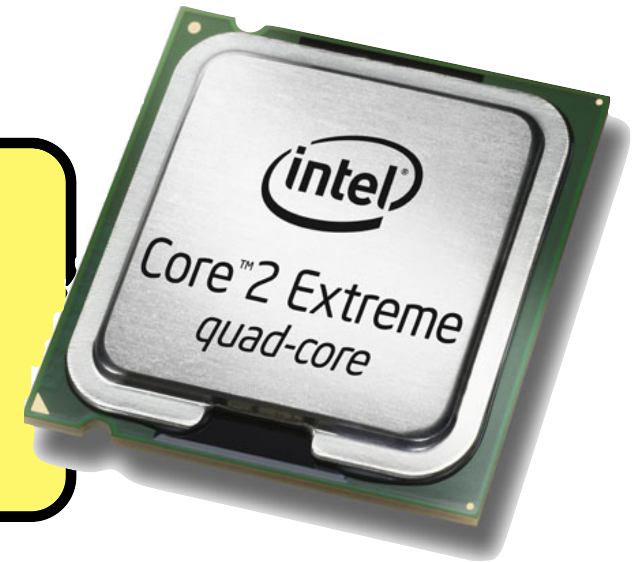
Ready

Job 4

Ready

Job 3

Running

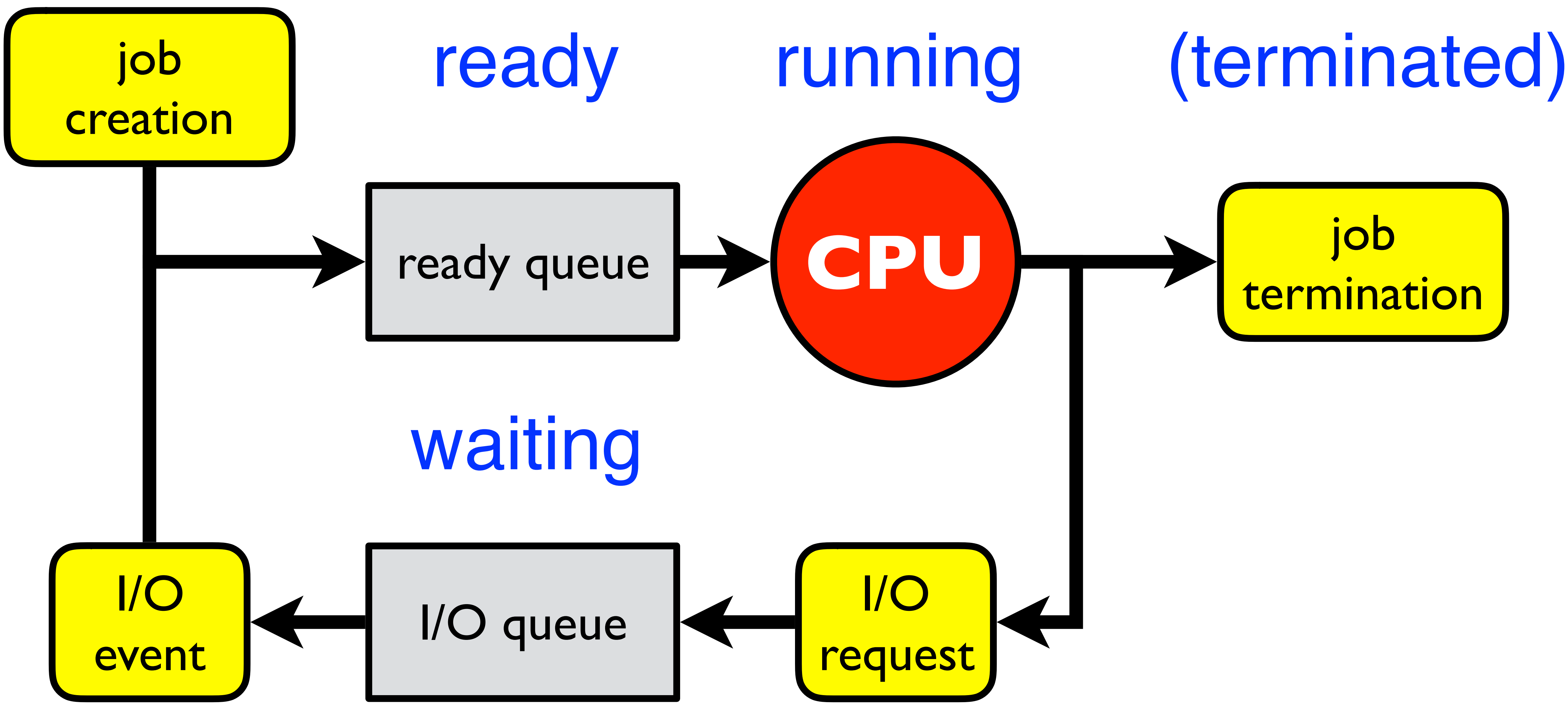


The state of the waiting job (job 1) will change from waiting to ready.

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 requesting 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

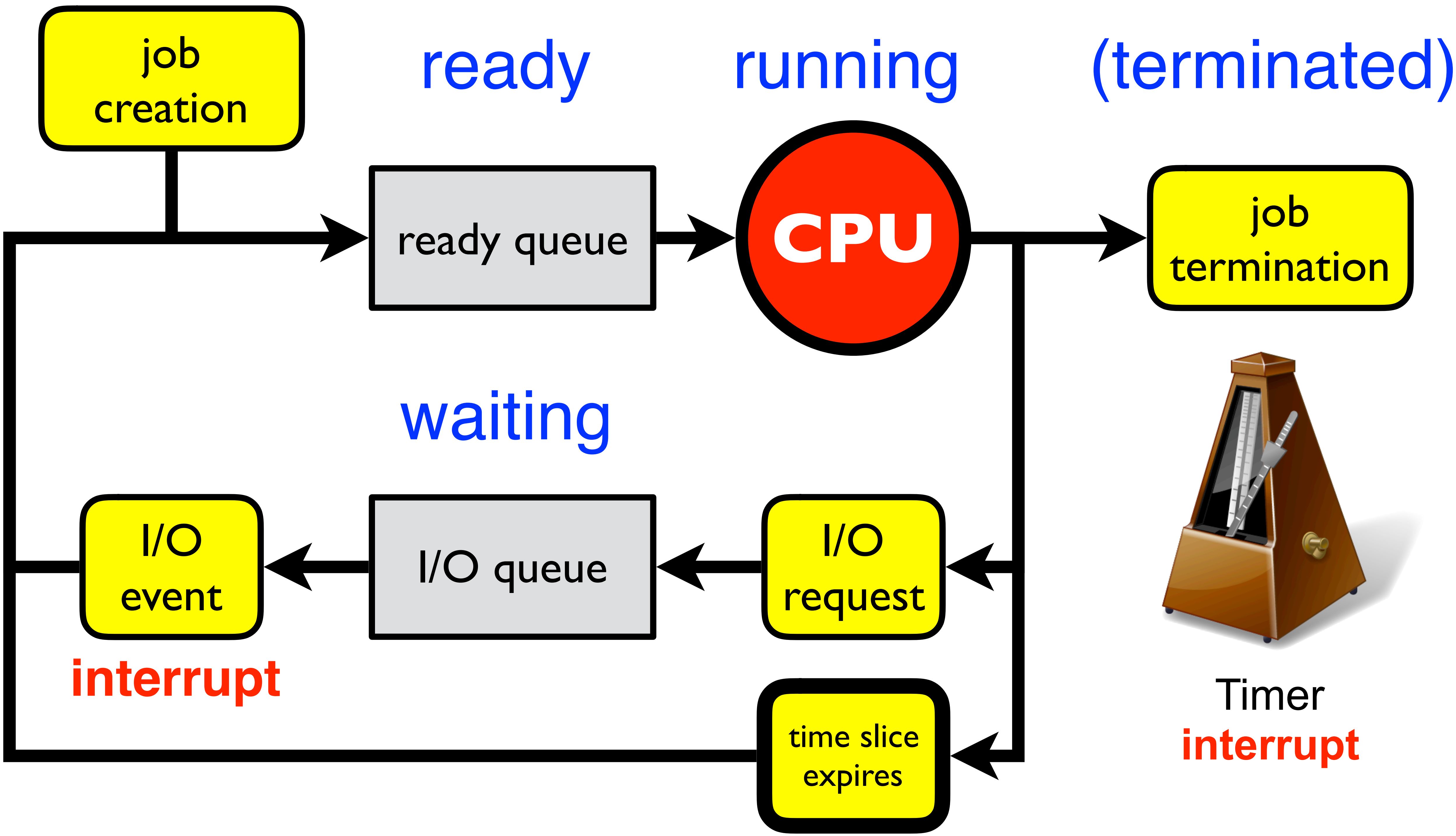
Multitasking

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

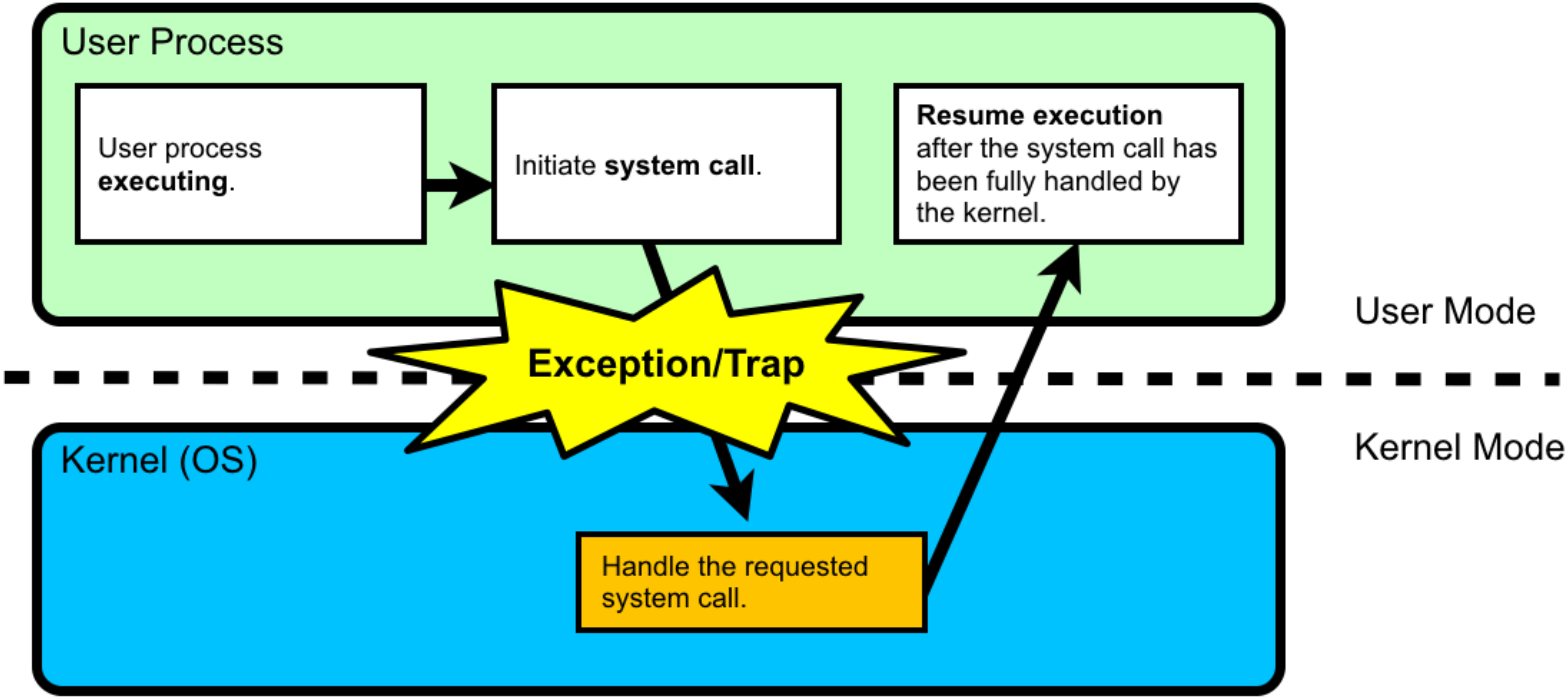
Multiprogramming and time sharing are designed to achieve different goals.

- ★ 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 calls

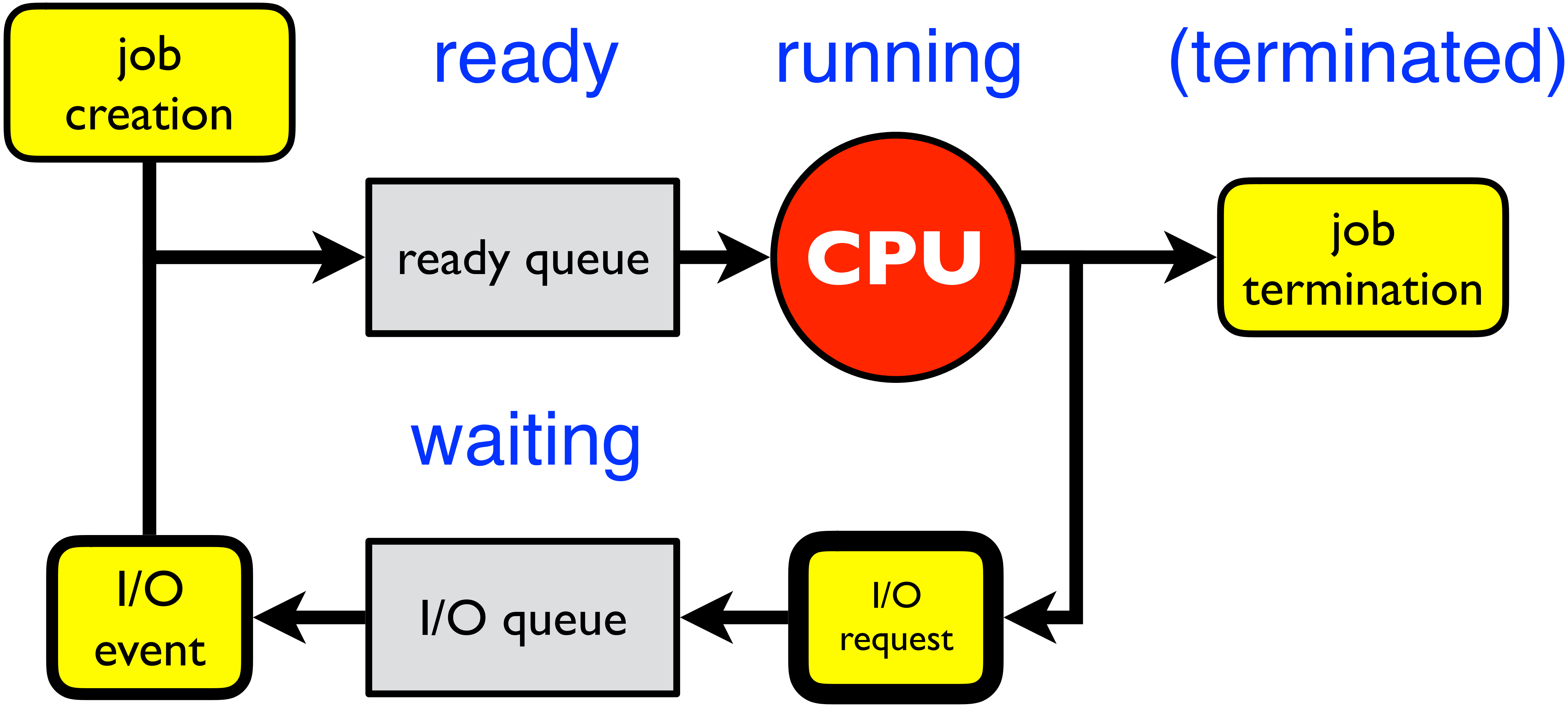
System calls

- 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**.


User Process
User process **executing**.
Initiate **system call**.
Resume execution after the system call has been fully handled by the kernel.
Exception/Trap
User Mode
Kernel Mode
Kernel (OS)
Handle the requested system call.

Multiprogramming

A schematic view of multiprogramming

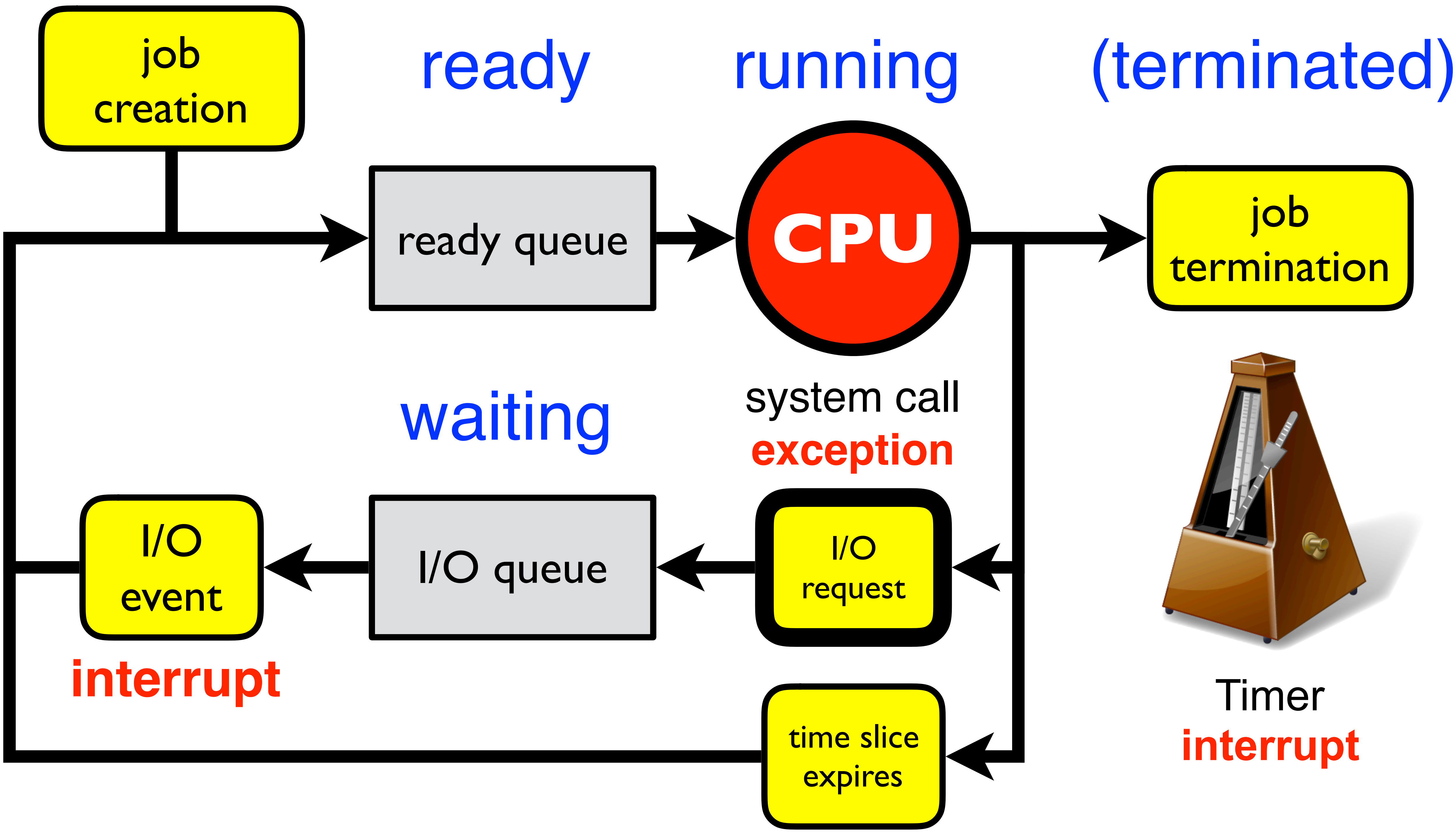


Interrupt used to signal an external and asynchronous I/O event

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

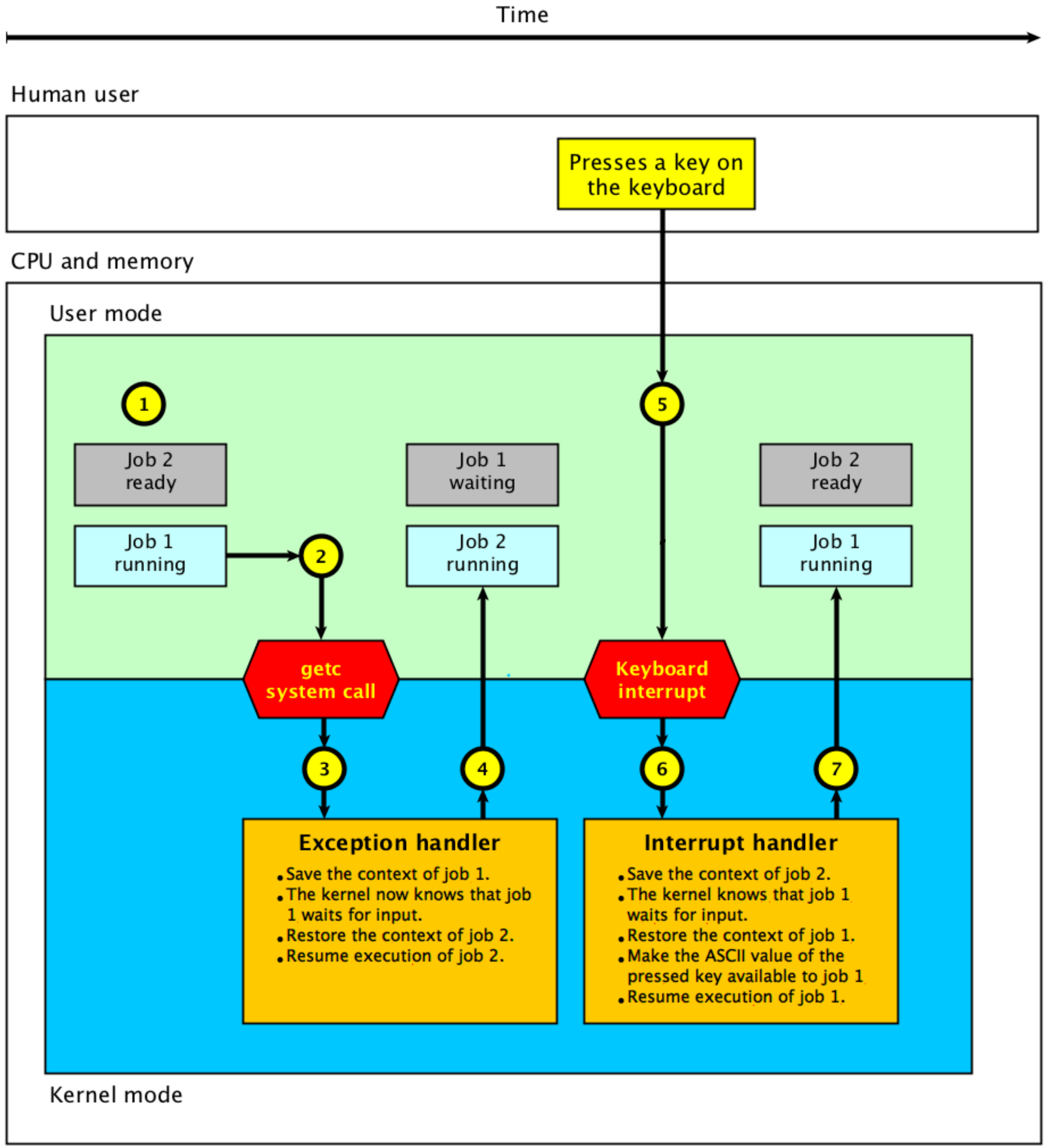
Multitasking

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.


Time
Human user
Presses a key on the keyboard
CPU and memory
User mode
1
Job 2 ready
Job 1 running
2
Job 1 waiting
Job 2 running
5
Job 2 ready
Job 1 running
getc system call
Keyboard interrupt
3
4
6
7
Exception handler
Save the context of job 1.
The kernel now knows that job 1 waits for input.
Restore the context of job 2.
Resume execution of job 2.
Interrupt handler
Save the context of job 2.
The kernel knows that job 1 waits for input.
Restore the context of job 1.
Make the ASCII value of the pressed key available to job 1
Resume execution of job 1.
Kernel mode

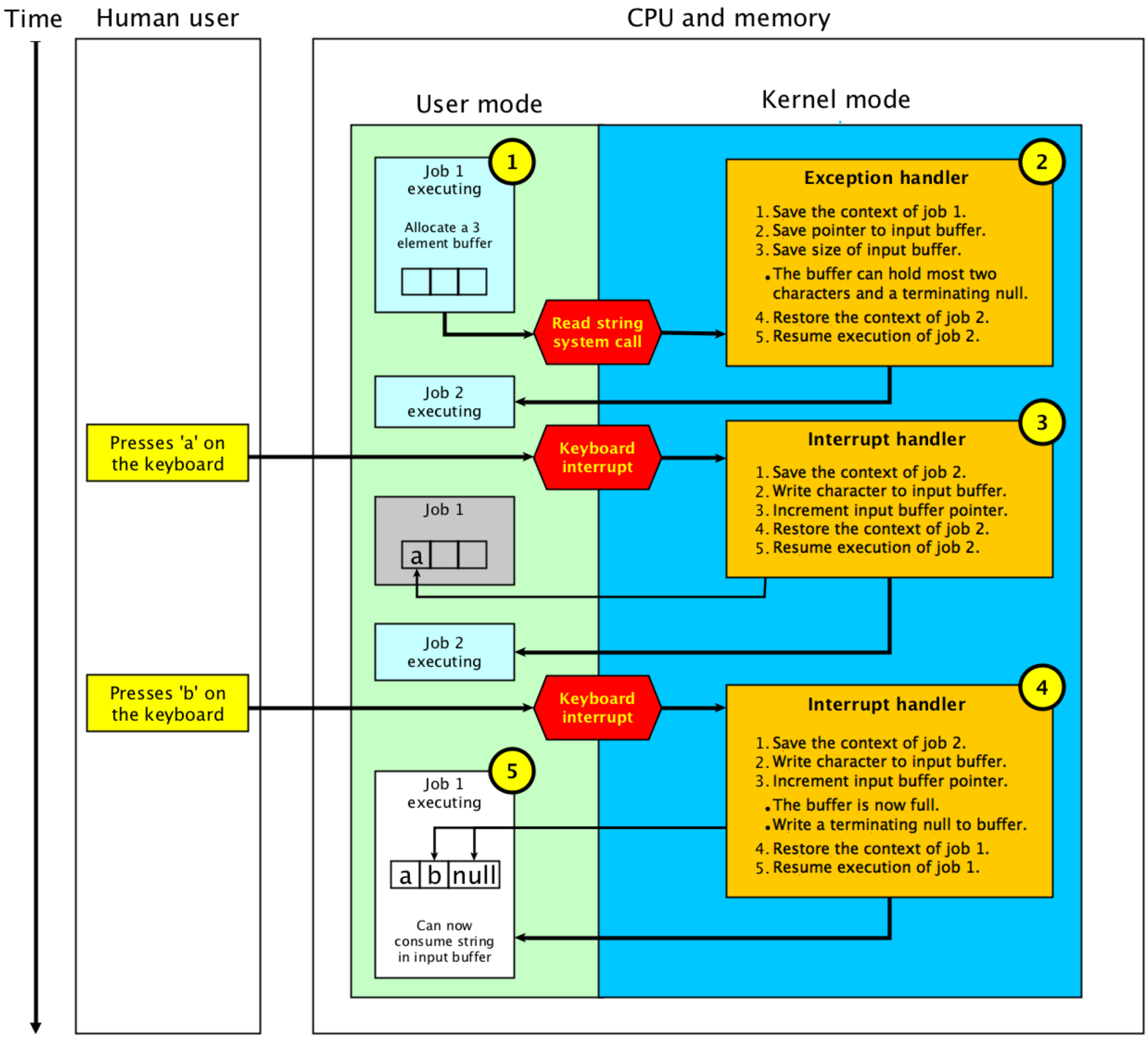
Problem

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

Read string system call design

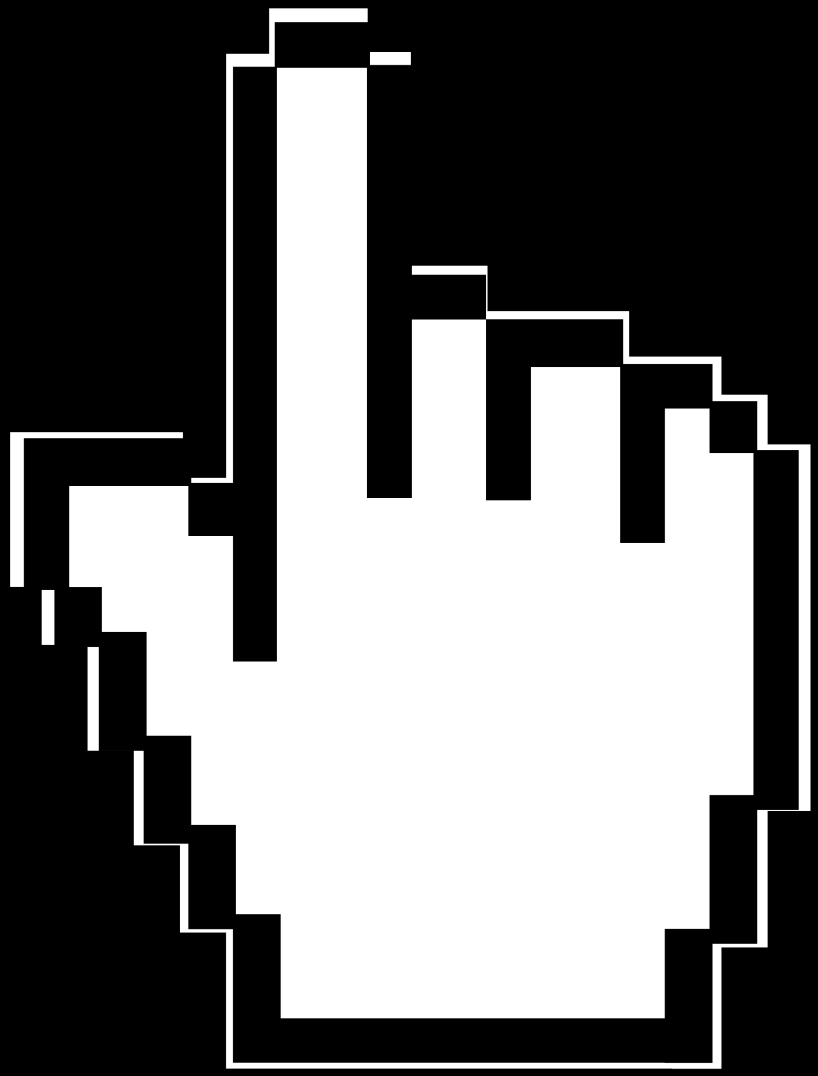
Read string system call design

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
User mode
Kernel mode
1
Job 1
executing
Allocate a 3
element buffer
Read string
system call
2
Exception handler
1. Save the context of job 1.
2. Save pointer to input buffer.
3. Save size of input buffer.
• The buffer can hold most two characters and a terminating null.
4. Restore the context of job 2.
5. Resume execution of job 2.
Job 2
executing
Presses 'a' on the keyboard
Keyboard
interrupt
3
Interrupt handler
1. Save the context of job 2.
2. Write character to input buffer.
3. Increment input buffer pointer.
4. Restore the context of job 2.
5. Resume execution of job 2.
Job 1
a
Job 2
executing
Presses 'b' on the keyboard
Keyboard
interrupt
4
Interrupt handler
1. Save the context of job 2.
2. Write character to input buffer.
3. Increment input buffer pointer.
• The buffer is now full.
• Write a terminating null to buffer.
4. Restore the context of job 1.
5. Resume execution of job 1.
5
Job 1
executing
a
b
null
Can now
consume string
in input buffer

The mouse



Does the mouse produce interrupts every time you move it?

★ https://www.quora.com/Does-the-mouse-produce-interrupts-every-time-you-move-it

How does a computer process input of a moving mouse?

★ https://stackoverflow.com/questions/32633308/how-does-a-computer-process-input-of-a-moving-mouse

If a computer mouse uses interrupts, why doesn't it slow down the PC?

★ https://www.quora.com/If-a-computer-mouse-uses-interrupts-why-doesnt-it-slow-down-the-PC

In the GitHub slide repository you also find a set of **self study** slides for module 1.