Introduction to Operating Systems

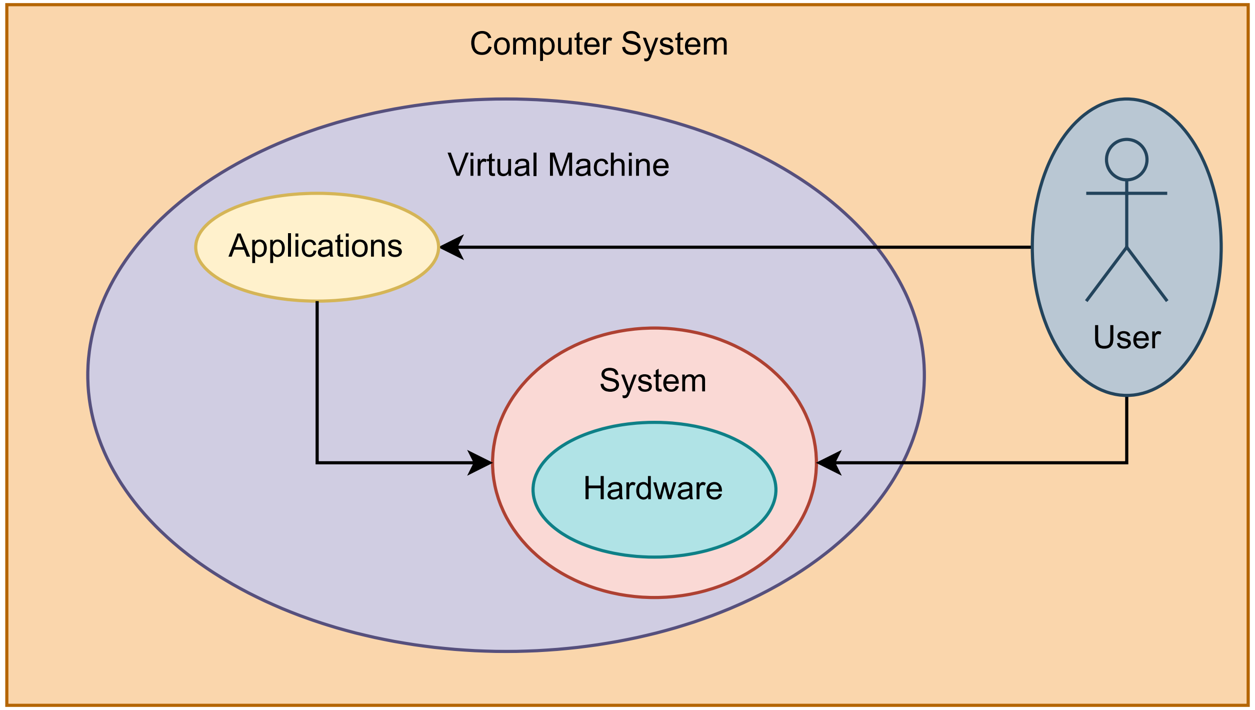
Chapter 1. Overview

1. Why an Operating system (OS)?

A computer is a complex machine that consists of one or more processors, memory, clocks, terminals, disks, and more. Besides this, not every user is familiar with the hardware.

This problem has been solved by adding a layer of software on top of the hardware to hide the complex mechanisms from the end users.

On the eyes of its users, the computer is a virtual machine much easier to understand and use.



The two main purposes of hiding the hardware from the users and programmers are:

Abstracting Complexity

Performing a task of abstraction to set aside the very concrete hardware and have a more global and simplified view of the computer.

How? Multiple levels/layers of abstraction.

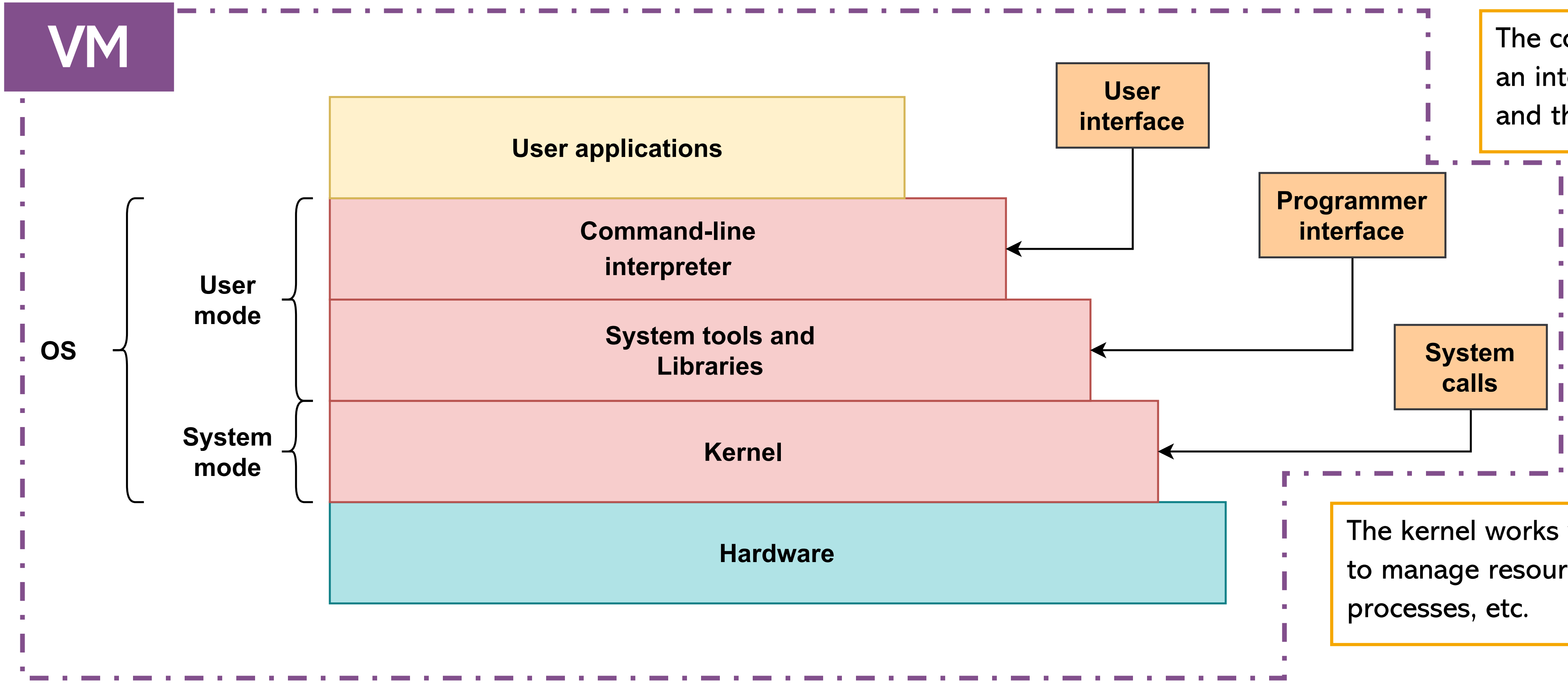
Security and Control

Providing a new and safer mode of working when code belonging to the OS is executed.

It is convenient that only certain users have access to the elements of the system, or the normal functionality of the entire computer could be in danger.

How? Authentication and access rights.

2. Virtual Machine



The command interpreter acts as an interface between the users and the system.

Tools and libraries provide users and programmers with access to essential functionalities and resources.

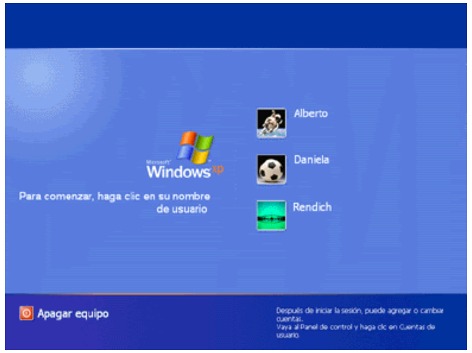
The kernel works with the hardware to manage resources, schedule processes, etc.

User viewpoint

- **Work Session**

  In a multiuser system, users access the system through a work session. This work session creates a personalized environment with the authorized resources.

  1. **Login:** the system asks for the username and password.
  2. **Work:** the user interacts with the shell.
  3. **Logout:** the user indicates they want to leave the session.



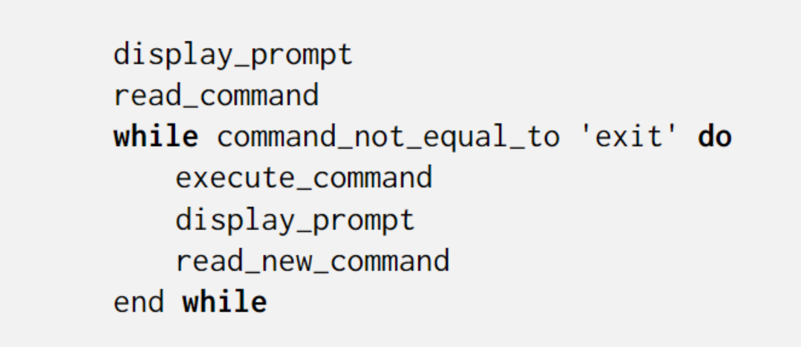
Para comenzar, haga clic en su nombre de usuario

Después de iniciar la sesión, puede agregar o cambiar cuentas.
Vaya al Panel de control y haga clic en Cuentas de usuario.

- **Command-line Interpreter (Shell)**

  The command-line interpreter is a program responsible for interpreting and communicating to the operating system what actions the user intends to perform in the system.

  The command-line interpreter recognizes internal commands (executed directly by the interpreter), and external commands (located in directories across the file system).



```
display_prompt
read_command
while command_not_equal_to 'exit' do
    execute_command
    display_prompt
    read_new_command
end while
```

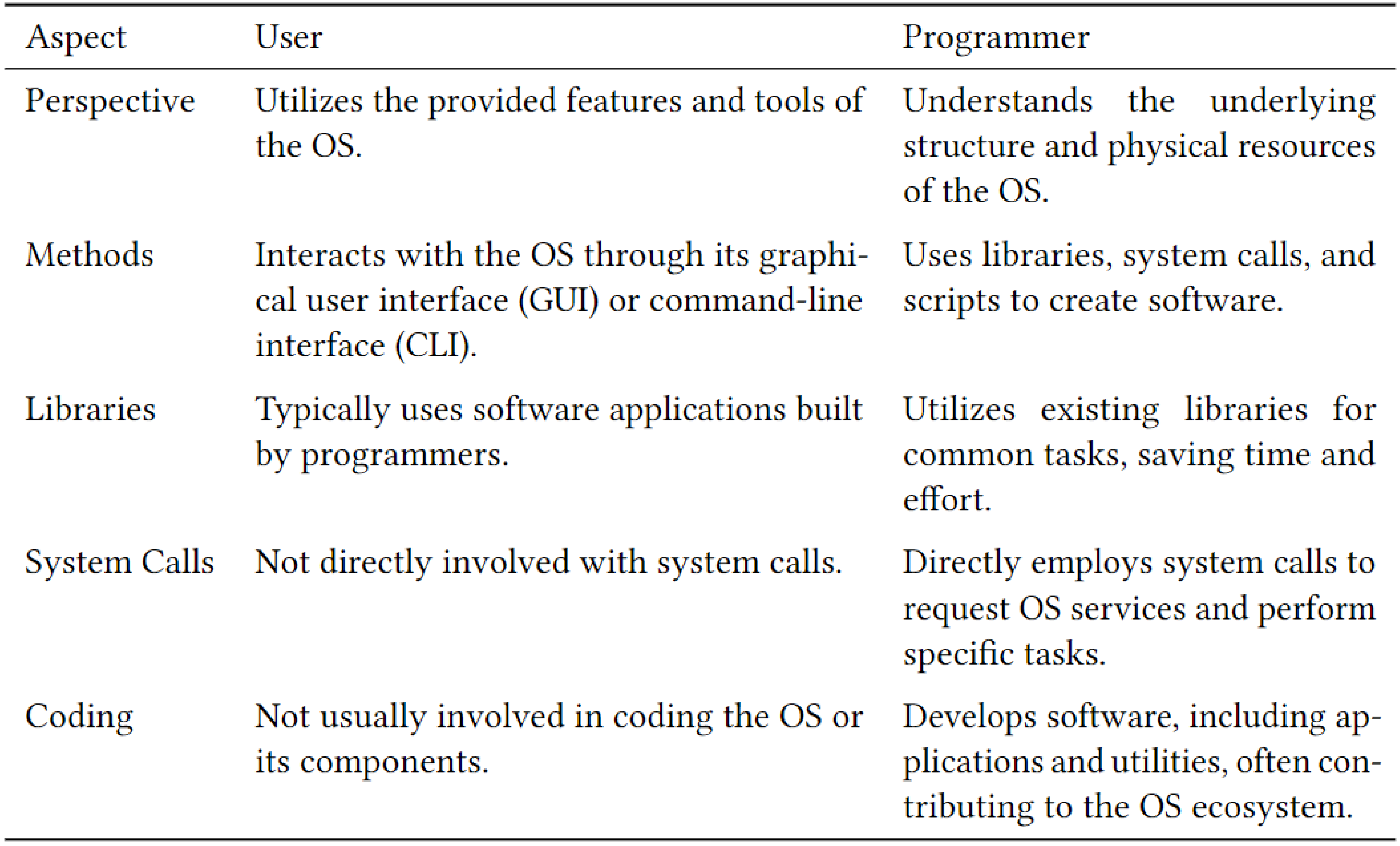
It works as a cyclic process

Programmer viewpoint

- **System Calls or Traps:**
  - They do not include new code to our programs.
  - They transfer control to a kernel-level program during execution.
  - User applications invoke system calls when requesting services from the operating system.
  - Common examples of system calls include opening and closing files, reading and writing data, creating processes…

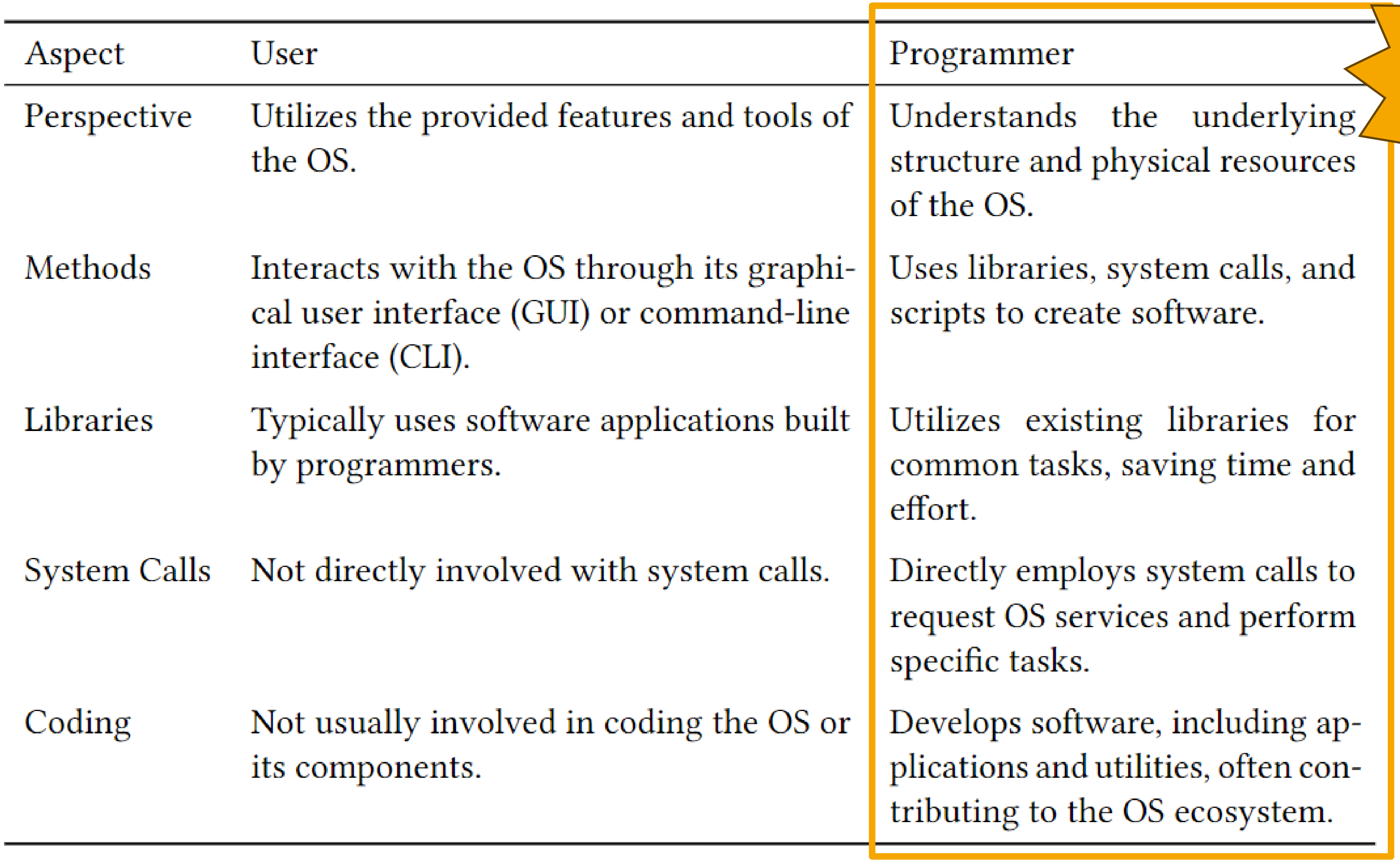
- **Libraries:**
  - They are collections of commonly used procedures and functions that are referenced and incorporated into applications.
  - Code integration occurs during the compilation process.
  - Types:
    - Standard libraries (I/O, data structures…)
    - Specialized libraries (mathematics, graphics…)

In summary:



Aspect	User	Programmer
Perspective	Utilizes the provided features and tools of the OS.	Understands the underlying structure and physical resources of the OS.
Methods	Interacts with the OS through its graphical user interface (GUI) or command-line interface (CLI).	Uses libraries, system calls, and scripts to create software.
Libraries	Typically uses software applications built by programmers.	Utilizes existing libraries for common tasks, saving time and effort.
System Calls	Not directly involved with system calls.	Directly employs system calls to request OS services and perform specific tasks.
Coding	Not usually involved in coding the OS or its components.	Develops software, including applications and utilities, often contributing to the OS ecosystem.

In summary:

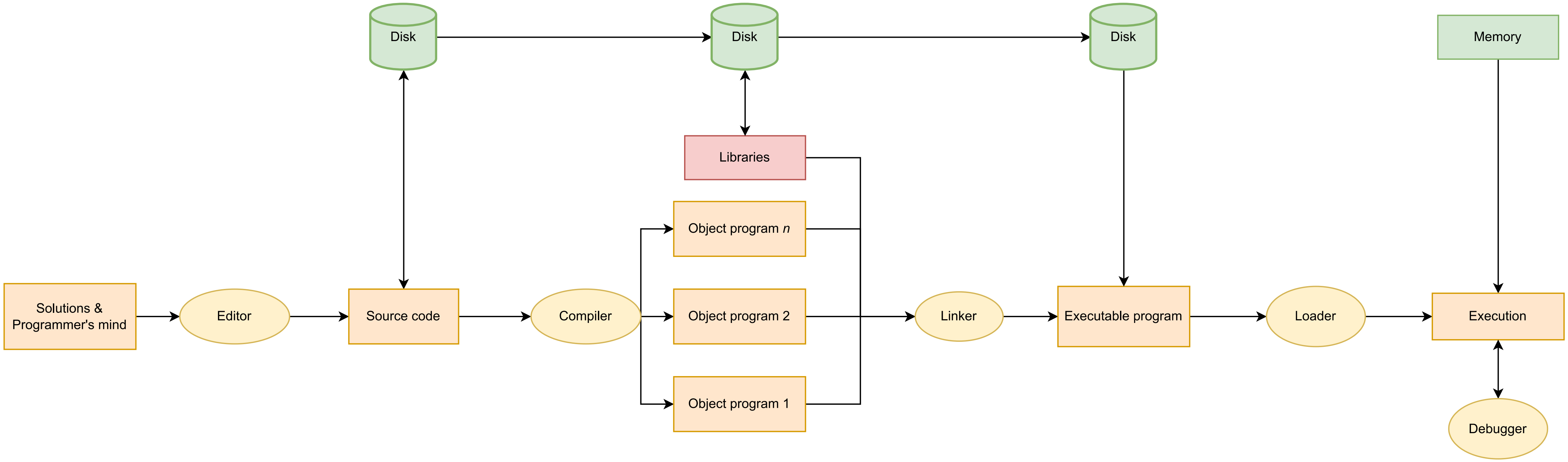


Aspect	User	Programmer
Perspective	Utilizes the provided features and tools of the OS.	Understands the underlying structure and physical resources of the OS.
Methods	Interacts with the OS through its graphical user interface (GUI) or command-line interface (CLI).	Uses libraries, system calls, and scripts to create software.
Libraries	Typically uses software applications built by programmers.	Utilizes existing libraries for common tasks, saving time and effort.
System Calls	Not directly involved with system calls.	Directly employs system calls to request OS services and perform specific tasks.
Coding	Not usually involved in coding the OS or its components.	Develops software, including applications and utilities, often contributing to the OS ecosystem.

How do we create software?

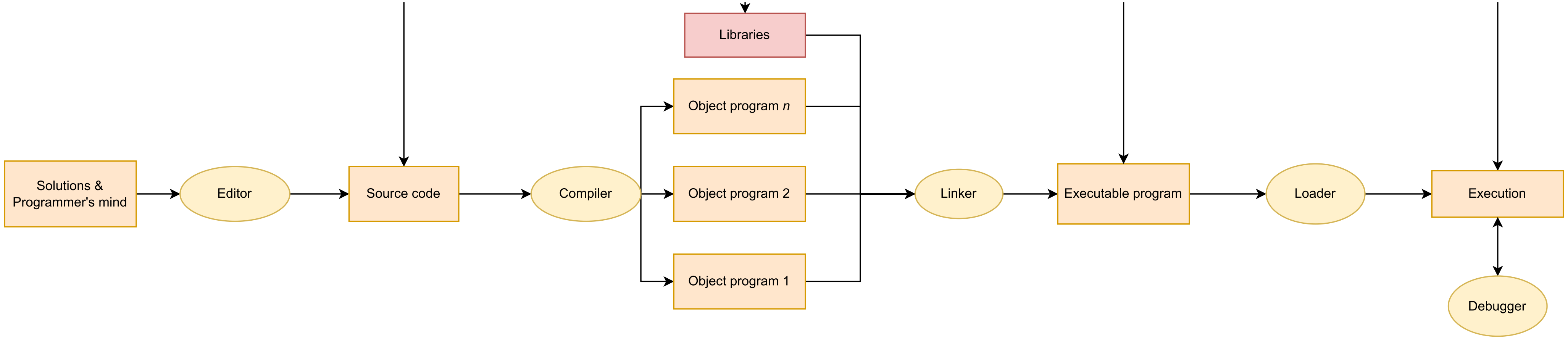
3. Stages of a program execution

It starts with the programmer writing code in a high-level language and ends with the execution of the program.



System utilities

- **Text editors:** Used by programmers for writing and editing the source code of a program.
- **Compilers:** Programs that translate high-level source code written by developers into machine code or an intermediate language.
- **Linkers:** Applications that combine multiple object files, generated by the compiler, and system libraries into a single executable program.
- **Loaders:** System components responsible for loading the executable program into memory, allocating system resources, and preparing the program for execution, among other tasks.
- **Debuggers:** A set of tools and applications to control the execution of a program in order to identify and resolve errors (bug).
- **Libraries:** Collections of pre-written code that can be reused in many programs.



Compilation

High-level program

Product.c

```
c = 0;
for(i=0; i<b; i++){
      c = c + a;
}
```

Assembly language

```
sum:          MOV zero,c
              MOV zero,i
for:          CMP i,b
              BEQ end
              ADD a,c
              ADD one,i
              CMP X,X
              BEQ for
end:
```

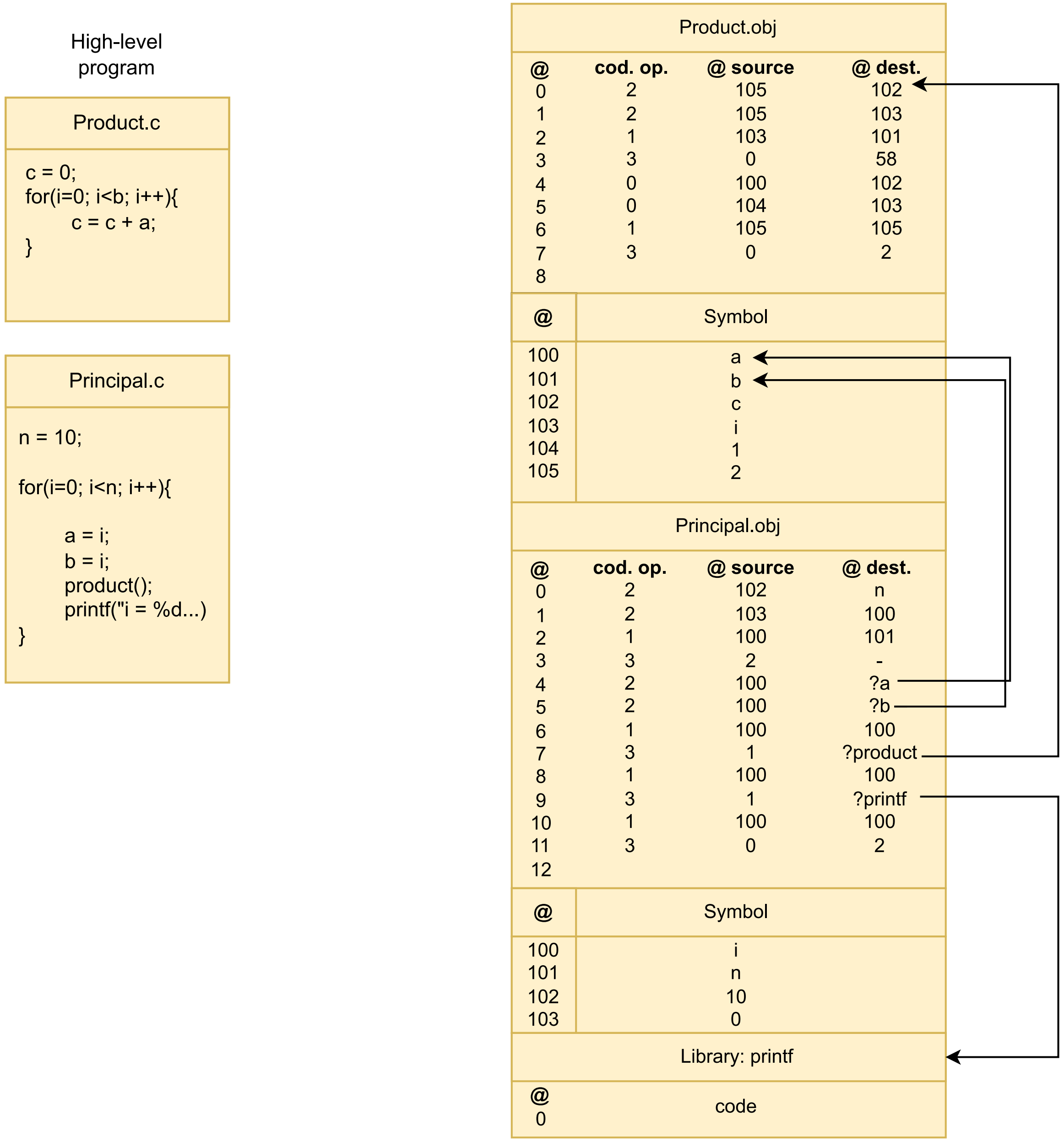
Machine language

Product.obj

@	**cod. op.**	**@ source**	**@ dest.**
0	2	105	102
1	2	105	103
2	1	103	101
3	3	0	58
4	0	100	102
5	0	104	103
6	1	105	105
7	3	0	2
8			

@	Symbol
100	a
101	b
102	c
103	i
104	1
105	2

The compiler translates a program written in a high-level programming language into an object program or machine code.



High-level
program

Product.c

```c
c = 0;
for(i=0; i<b; i++){
        c = c + a;
}
```

Principal.c

```c
n = 10;

for(i=0; i<n; i++){

        a = i;
        b = i;
        product();
        printf("i = %d...)
}
```

Product.obj

@	cod. op.	@ source	@ dest.
0	2	105	102
1	2	105	103
2	1	103	101
3	3	0	58
4	0	100	102
5	0	104	103
6	1	105	105
7	3	0	2
8			

@	Symbol
100	a
101	b
102	c
103	i
104	1
105	2

Principal.obj

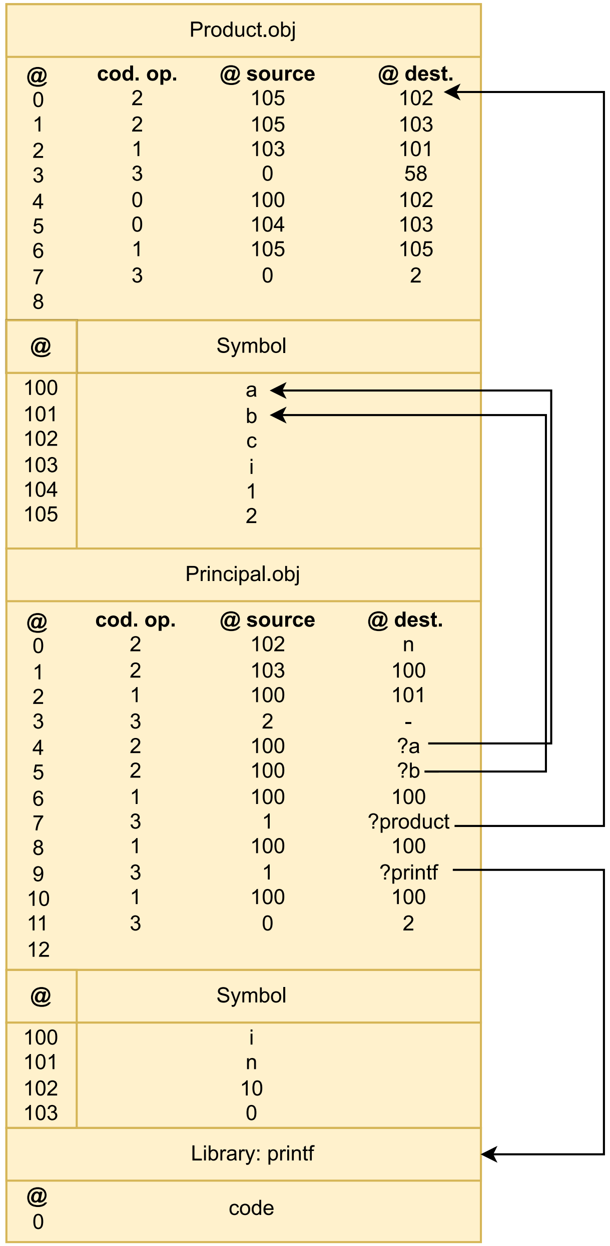
@	cod. op.	@ source	@ dest.
0	2	102	n
1	2	103	100
2	1	100	101
3	3	2	-
4	2	100	?a
5	2	100	?b
6	1	100	100
7	3	1	?product
8	1	100	100
9	3	1	?printf
10	1	100	100
11	3	0	2
12			

@	Symbol
100	i
101	n
102	10
103	0

Library: printf

@	code
0	

The linker is responsible for resolving external references among program modules and establishing the logical memory addresses.



Product.obj

@	cod. op.	@ source	@ dest.
0	2	105	102
1	2	105	103
2	1	103	101
3	3	0	58
4	0	100	102
5	0	104	103
6	1	105	105
7	3	0	2
8			

@	Symbol
100	a
101	b
102	c
103	i
104	1
105	2

Principal.obj

@	cod. op.	@ source	@ dest.
0	2	102	n
1	2	103	100
2	1	100	101
3	3	2	-
4	2	100	?a
5	2	100	?b
6	1	100	100
7	3	1	?product
8	1	100	100
9	3	1	?printf
10	1	100	100
11	3	0	2
12			

@	Symbol
100	i
101	n
102	10
103	0

Library: printf

@	
0	code

Executable.exe

mark,
system version, other data

@	cod. op.	@ source	@ dest.
0	2	105	102
1	2	105	103
2	1	103	101
3	3	2	-
4	0	100	102
5	0	104	103
6	1	103	103
7	3	0	2
8	2	108	102
9	2	109	106
10	1	106	107
11	3	2	-
12	2	106	100
13	2	106	101
14	1	106	106
15	3	1	0
16	1	106	106
17	3	1	21
18	1	106	106
19	3	0	10
20	x	x	x
21		Printf	
22			
23		Code	
24			
25		(RET)	

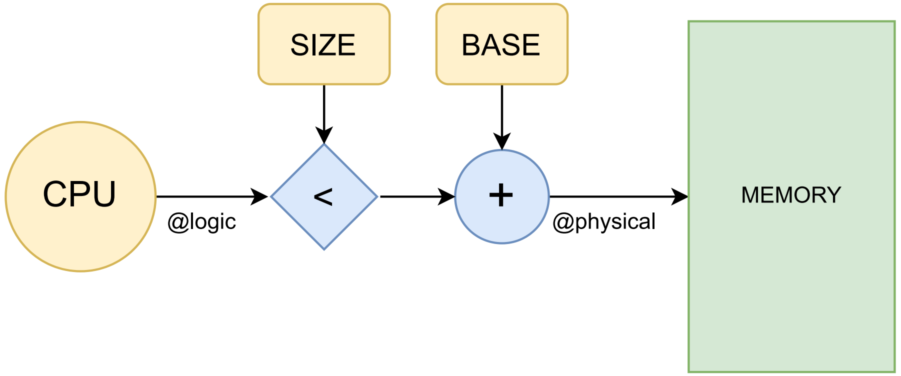
@	Symbol
100	a
101	b
102	c
103	i1
104	1
105	0
106	i2
107	n
108	10
109	0

After combining all the modules, including libraries, we get a final executable program.

4. Memory address

A relocatable program is one that can be executed at various memory addresses, and these addresses are determined by the loader during the loading process. That is why we distinguish between logic and physical addresses.

- Logic addresses: generated by the CPU and used by a program during its execution. They refer to locations within the program's own address space, as if it was the only program running on the system.
- Physical addresses: represent the actual locations in the computer's physical memory (RAM) where data and instructions are stored during program execution.



Coming: Memory Management

See you soon! ☺