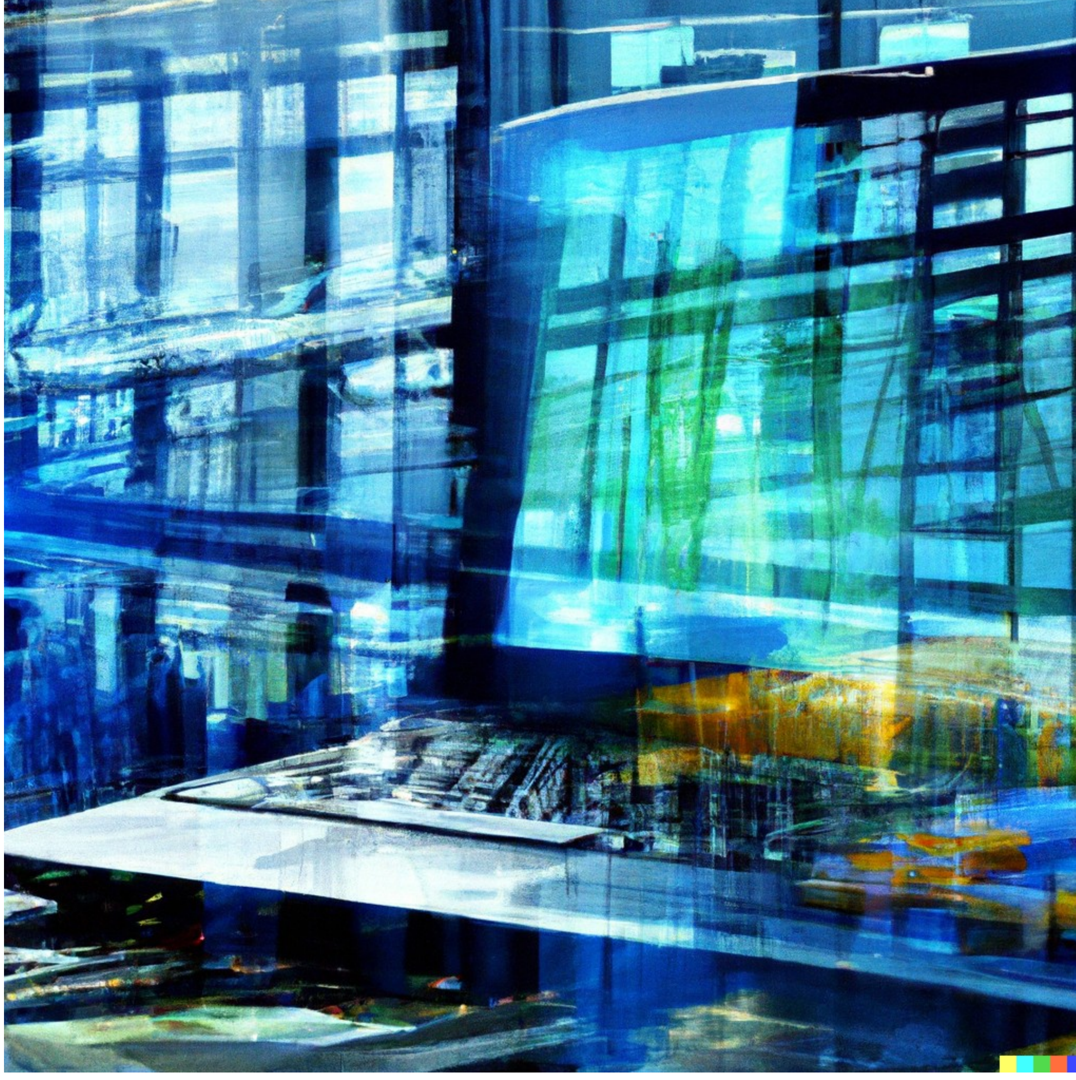
Operating Systems Processes

Víctor Robles Forcada
Operating Systems Group
Department of Architecture and Technology of
Computer Systems (DATSI)
ETSIINF – UPM



Index

General Process Concepts

General Concepts
Target Domain

Process Definition
Defining Process Hierarchy
Understand the lifecycle of a process

- Understanding a Process's Environment

Process Concept

Simplified Process **Definition**: Running Program

More Precise Process Definition: OS-Managed Processing Unit

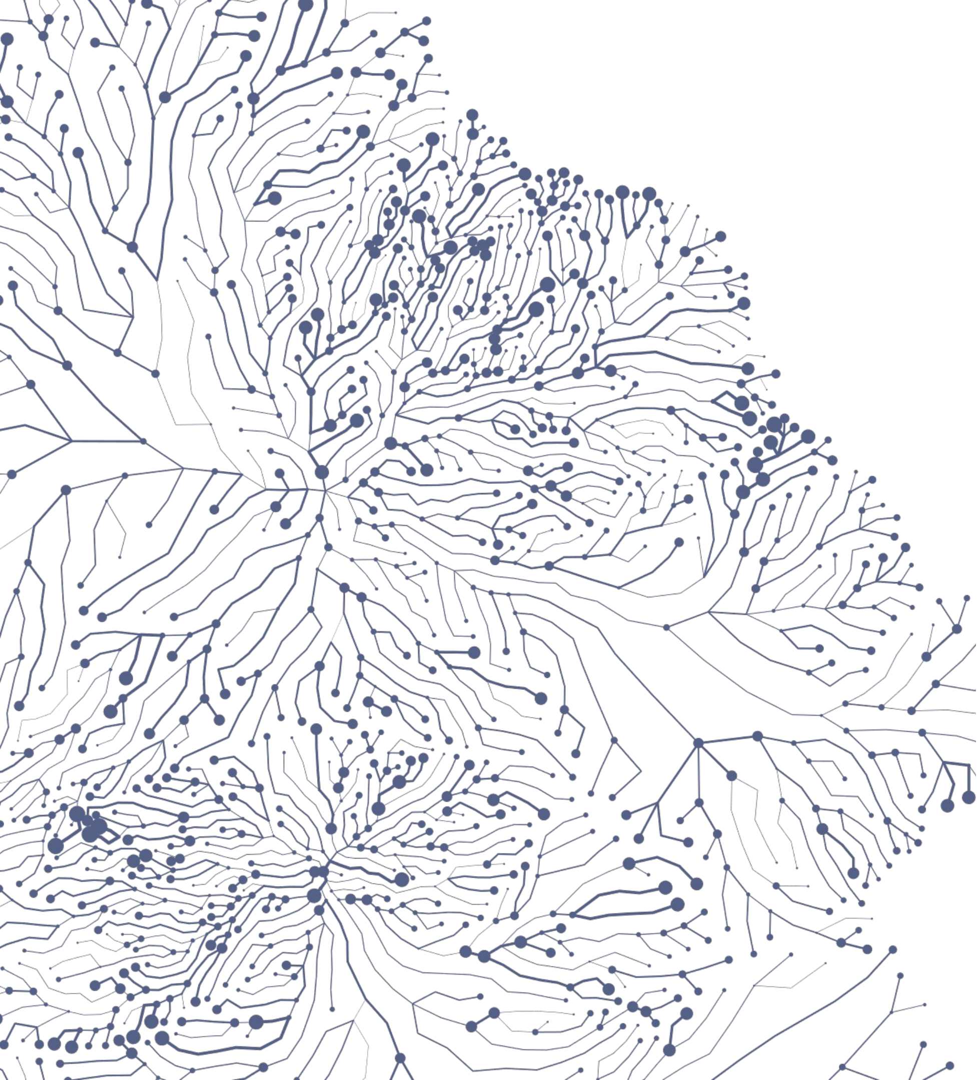
OS makes every running program believe it has its own machine

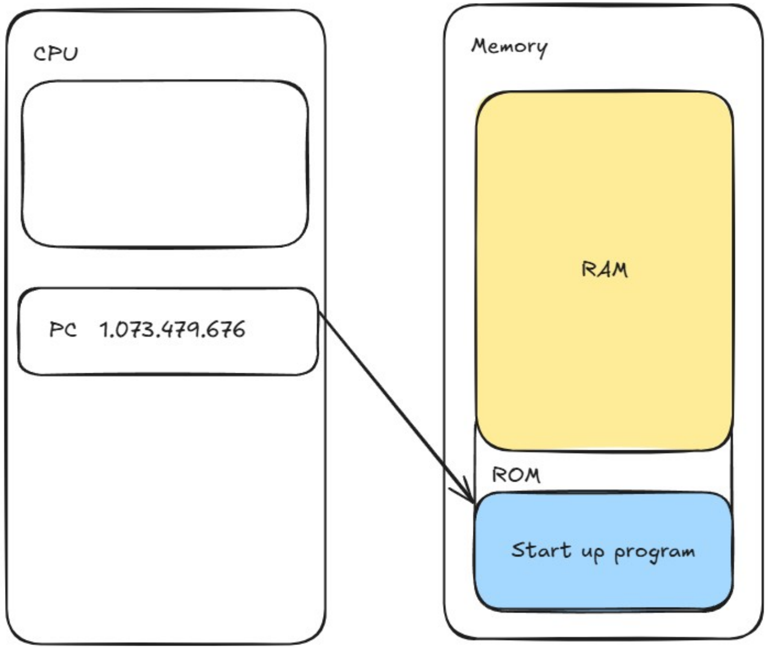
- Standalone Execution Context
- Except for communication mechanisms provided by OS

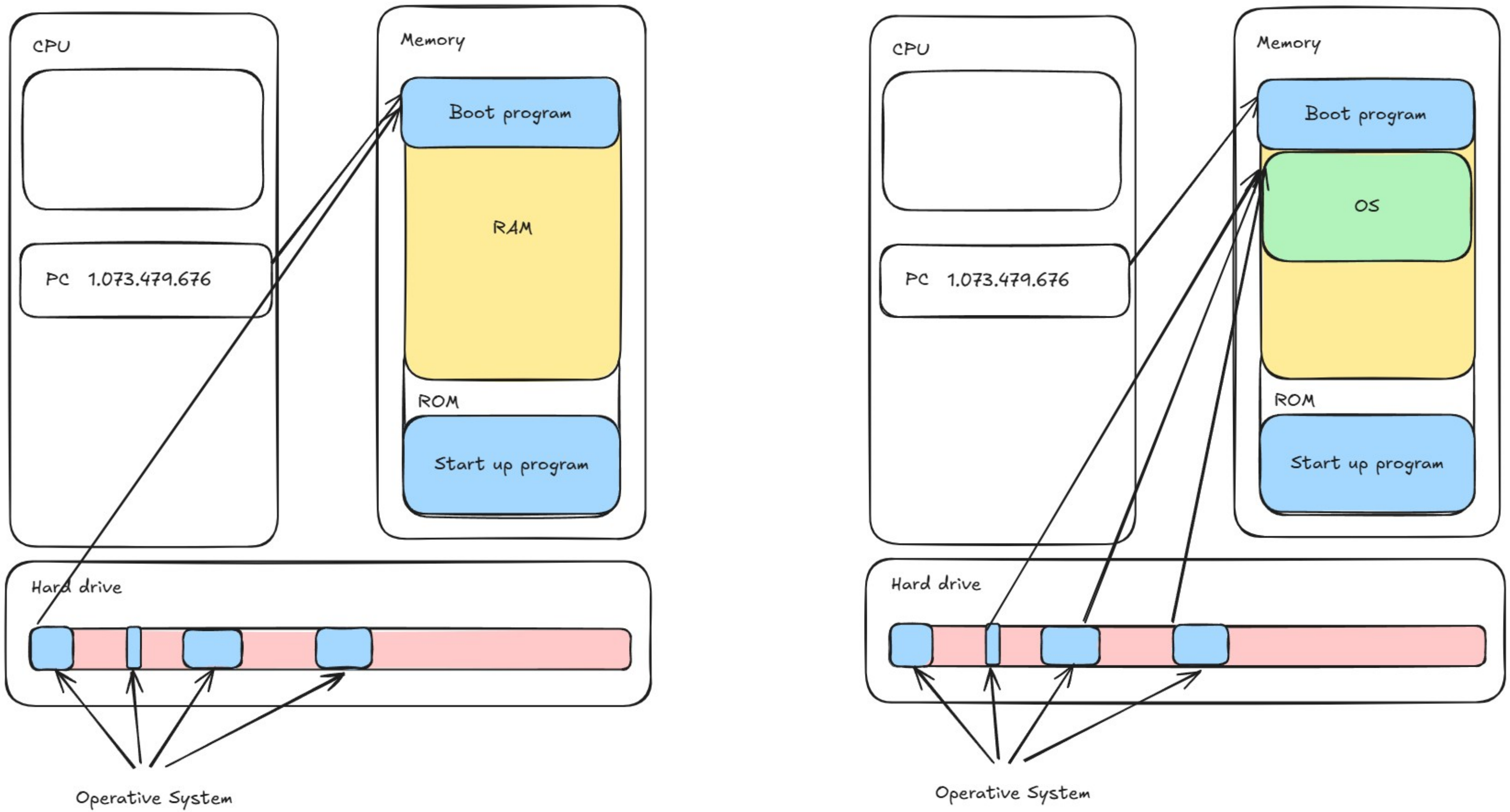
Multiple processes running the same program

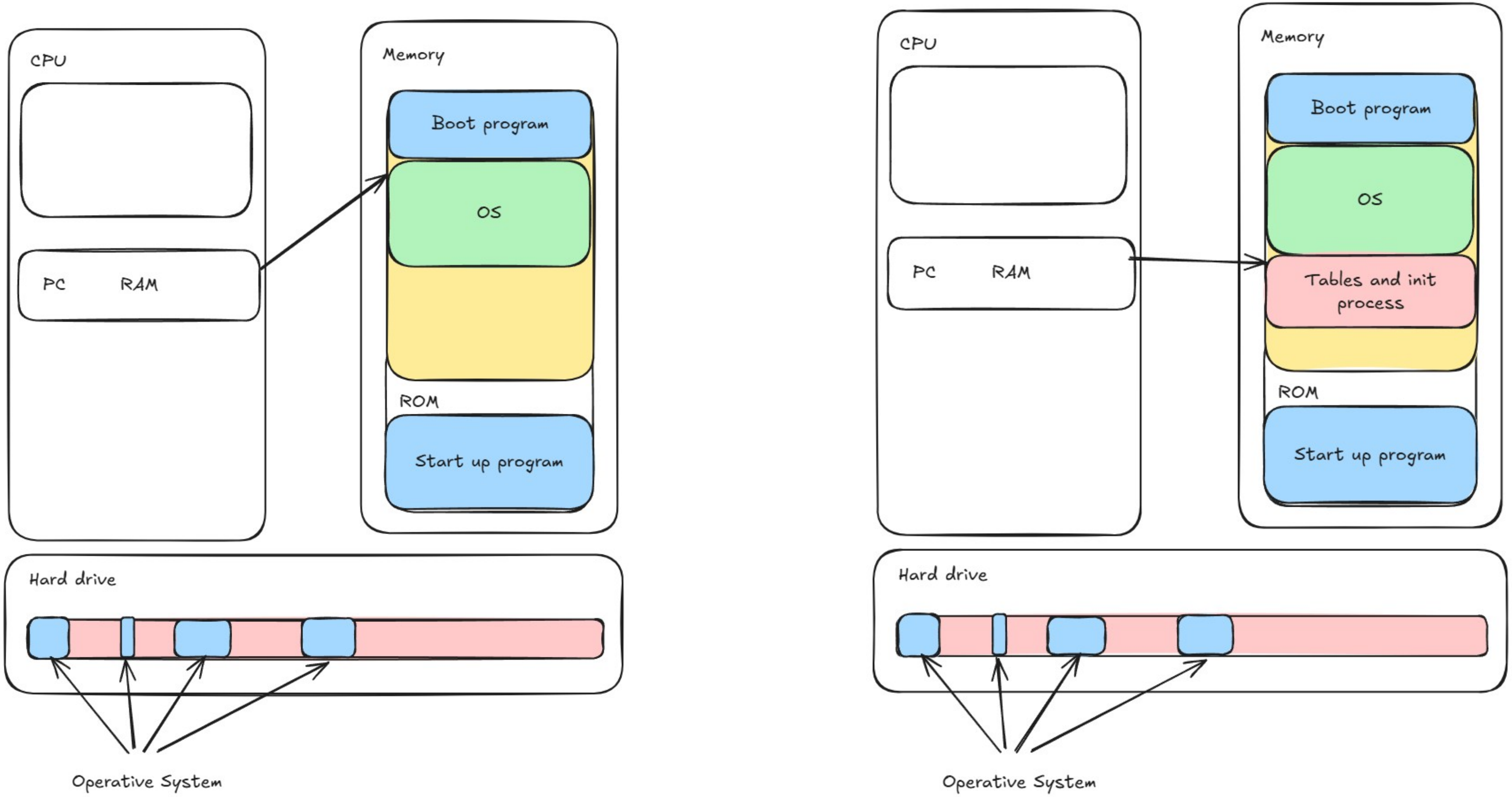
On UNIX, process can run multiple programs during its lifetime

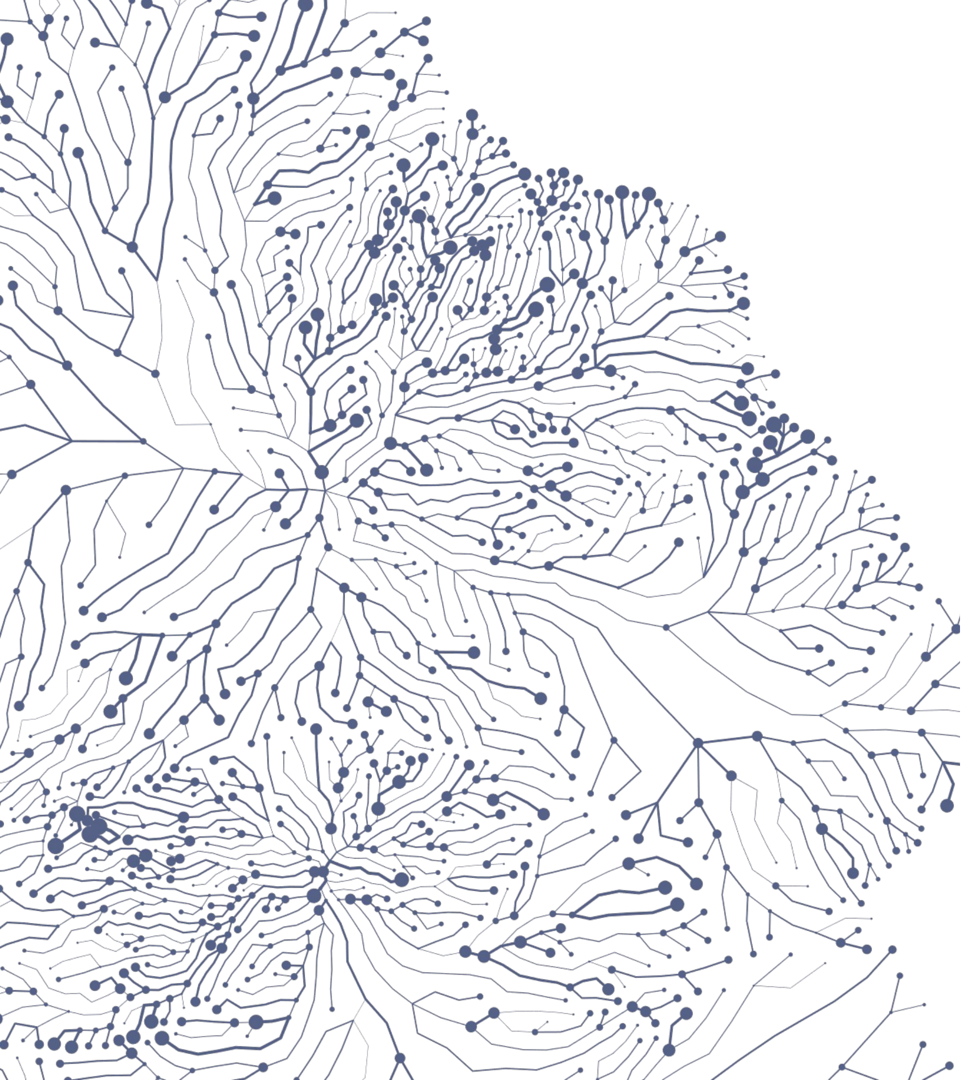
- fork() -> New process – same program
- exec(...) -> Same process – new program

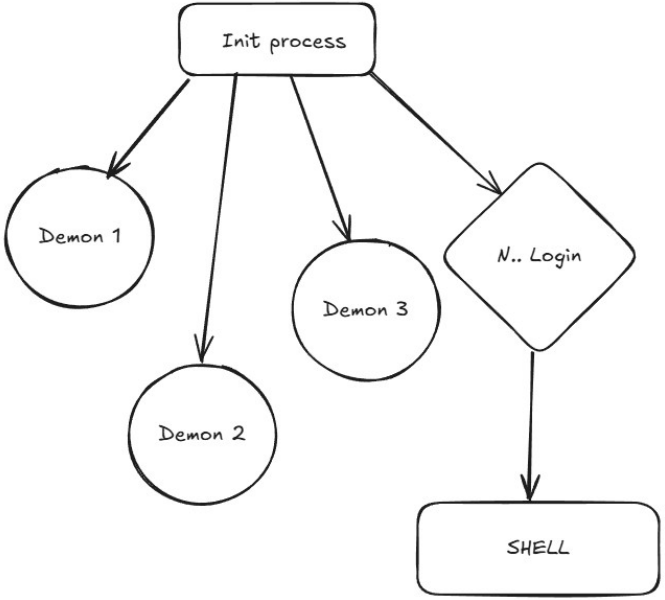



CPU
PC 1.073.479.676
Memory
RAM
ROM
Start up program


CPU
PC 1.073.479.676
Memory
Boot program
RAM
ROM
Start up program
Hard drive
Operative System
CPU
PC 1.073.479.676
Memory
Boot program
OS
ROM
Start up program
Hard drive
Operative System


CPU
PC
RAM
Memory
Boot program
OS
ROM
Start up program
Hard drive
Operative System
CPU
PC
RAM
Memory
Boot program
OS
Tables and init process
ROM
Start up program
Hard drive
Operative System




Init process
Demon 1
Demon 2
Demon 3
N.. Login
SHELL

Process Hierarchy

Process Family

Child Process

Parent Process

(Brother Process)

- (Grandfather Process)

Life of a process

Creates

Runs

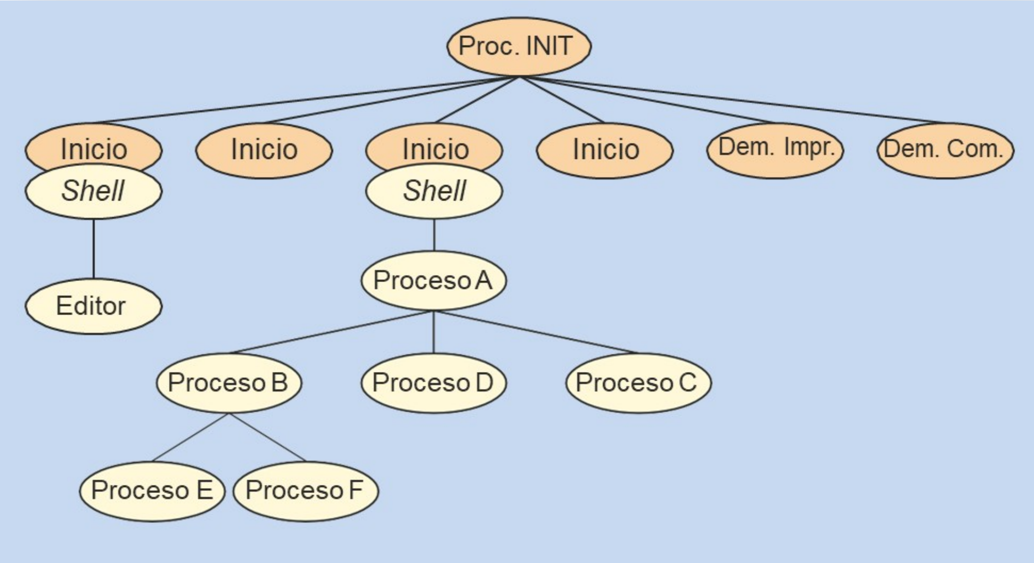
- Die or End

Process Execution

Interactive

- Non-interactive (batch or background)

Process group, dependent on each startup



Process Environment

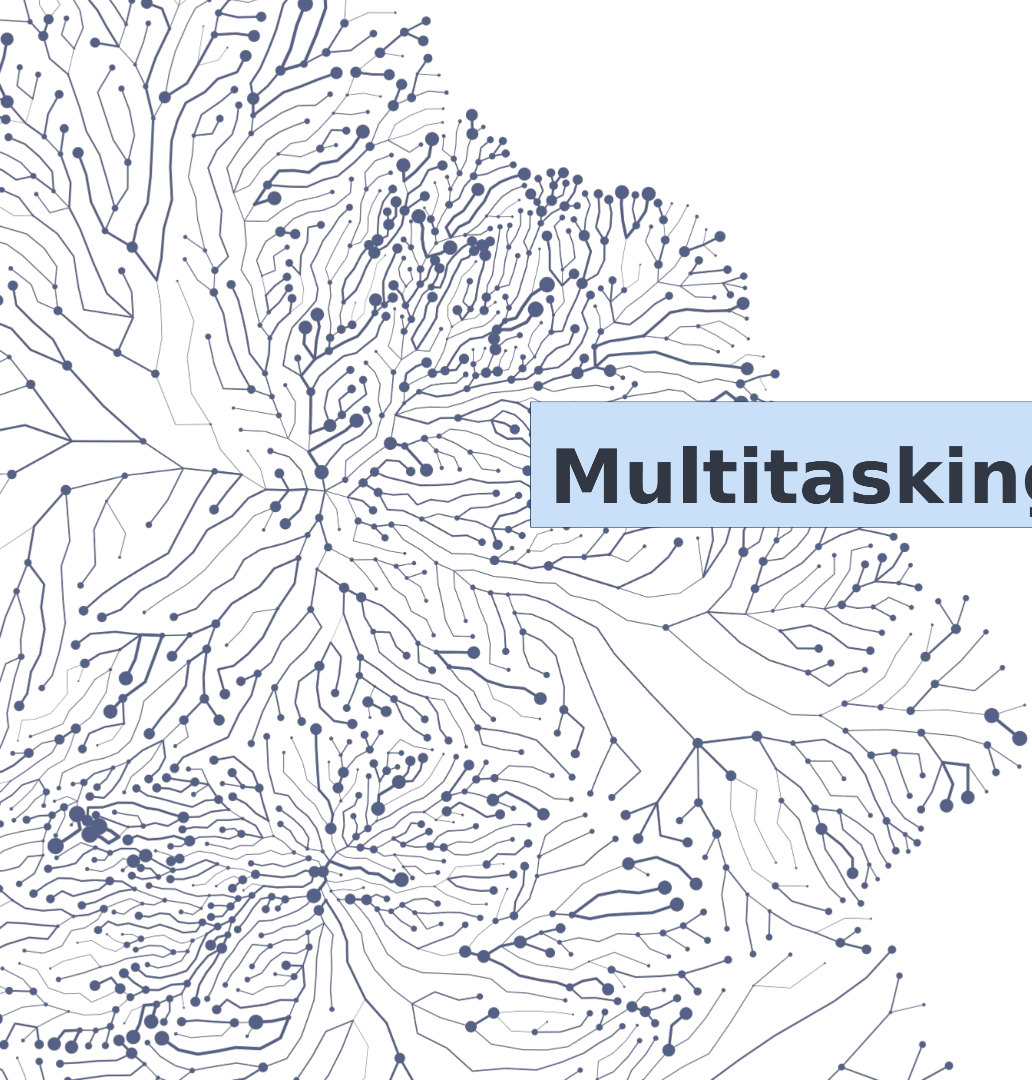
- A set of environment variables, which are used to run a program
- Name-Value table that is passed to the process in the Stack at its creation

the child process

- r Removing

HOME, the user's home directory
USER, logged-in user
PATH, directory prefix to find executables
TERM, terminal type in shell execution
PWD, Current Working Directory
SHELL, a command interpreter currently in use

- Examples of use
- env (list of environment variables)
- echo $PATH (PATH Environment Variable)
    - $PATH = $PATH:xxx (add new directory to current PATH)

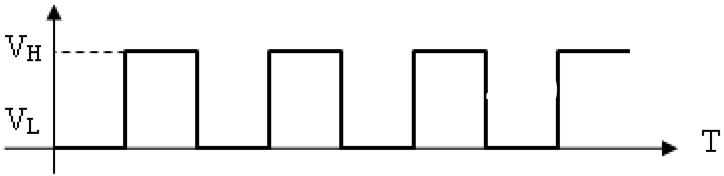


Multitasking

Multitasking
Target Domain

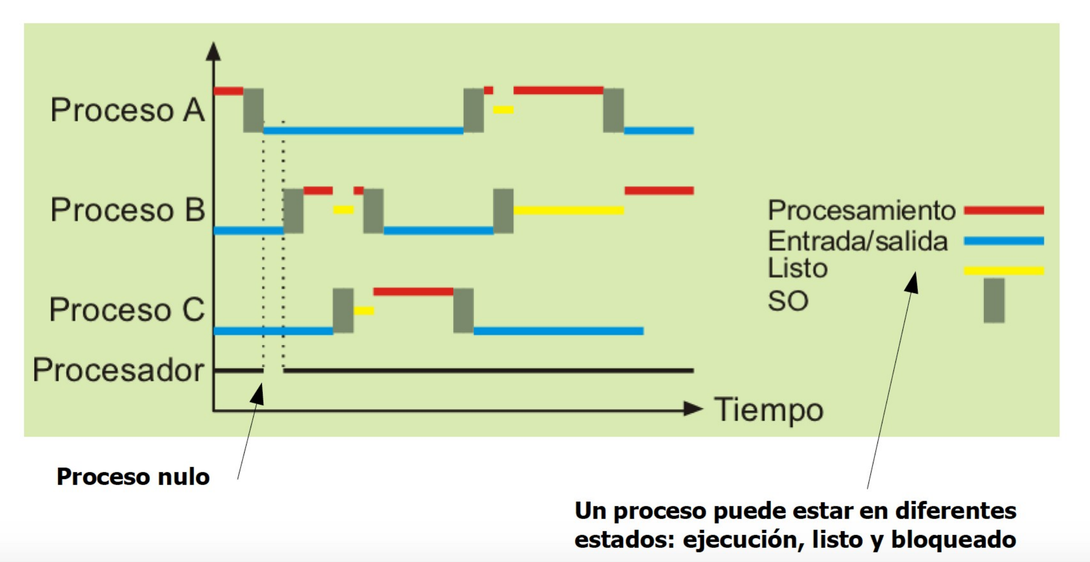
Multitasking concept

- Learn about the benefits of multiprogramming
- Processor clock speed



Multitasking Concept

Multitasking is the ability of an operating system to run multiple processes "at the same time" on a single computer, sharing system resources such as CPU, memory, and I/O devices



Advantages of Multitasking

Advantages of multiprogramming

Facilitates programming by dividing programs into processes

Enables simultaneous interactive service of multiple users efficiently

Take advantage of the time processes spend waiting for their I/O operations

- Increases CPU usage

Degree of multiprogramming

- Number of active processes in main memory

Memory Needs

System without virtual memory. The main memory must have the capacity to store the OS and all processes

Process Implementation

Process Implementation
Target Domain

Understanding Process States
Processor Multiplexing Concept
Understand the process information stored by the operating system
Processor State Concept
Understanding Context Shifts

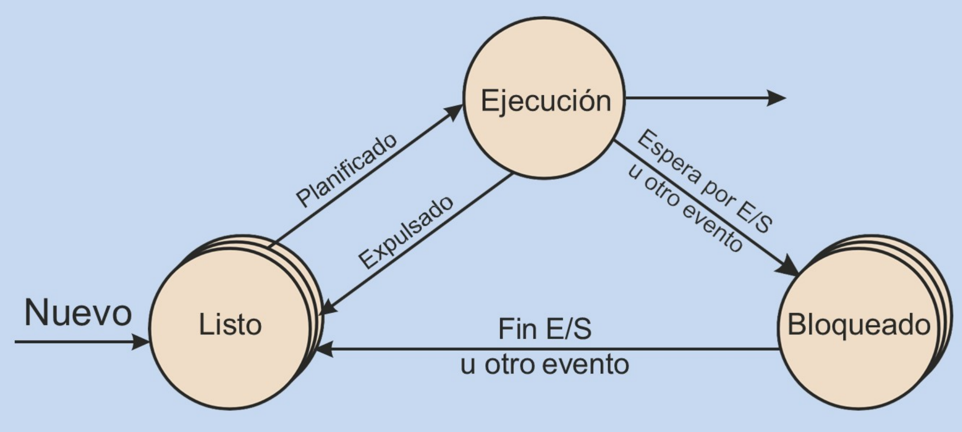
- Understanding the creation of a process

States of a Process

Execution: one per processor

Blocked: Waiting for I/O or event. For example, pause()

- Ready to Run



Scheduler: An OS module that decides which ready process to run

Trigger: An OS module that controls the selected process

- A process can be ejected because it has spent a certain processor time or because another, more priority process,

Processor Multiplexing

- OS distributes N processors among existing M processes (M>>N usually)
  - OS assigns and revokes processor usage to each process
- OS must store and manage the information of each process, called the Process Context
  - Processor registers
  - Your open files, your current directory, etc.
  - Process Memory Image
- OS must ensure that when it resumes running a process, the context of that process is "installed"

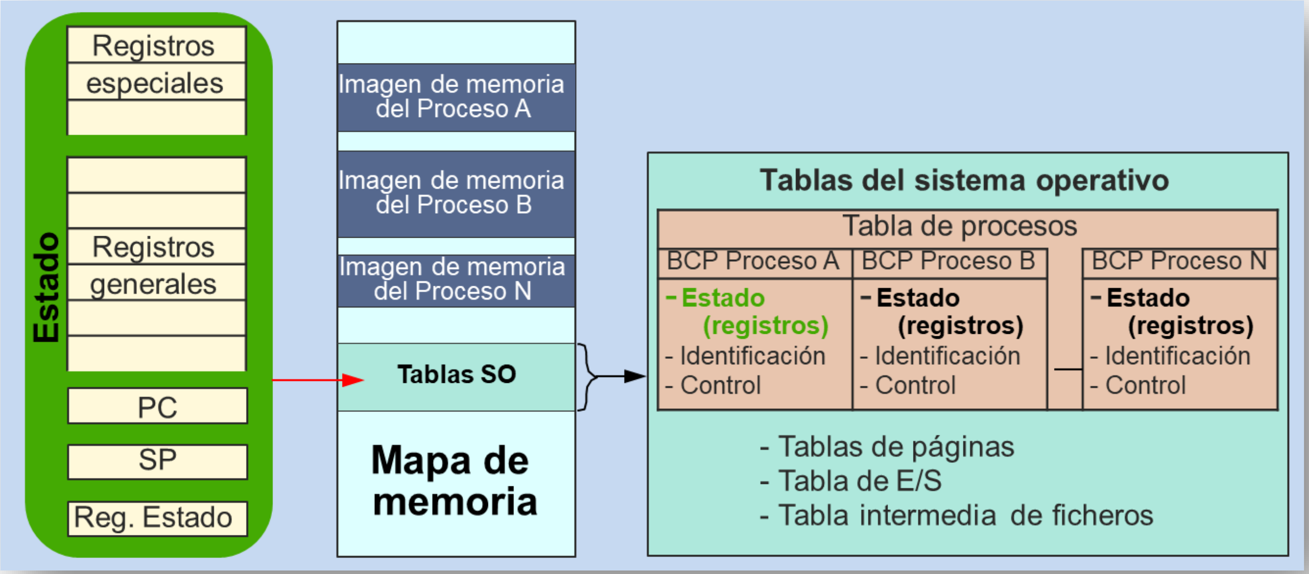
Process Information

The information is organized into three groups

Processor Registers

Memory Image

- OS Tables

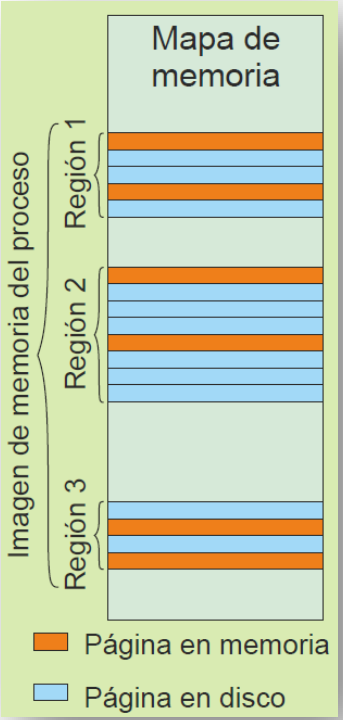


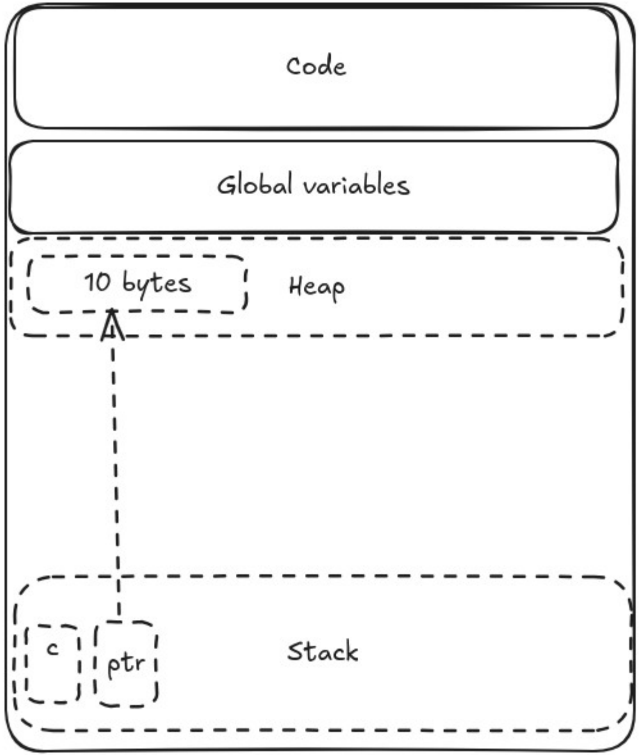
A.- Processor Status

- It consists of the contents of all processor registers
- Can reside in
- Processor registers, when the process is running
  - In the BCP, when the process is not running
- When blocking or preempting a process, the OS copies the processor state to its corresponding BCP
  - Performed by routine treatment interruptions
- Reminder: OS -> Event-driven execution
  - OS only executes when an event occurs

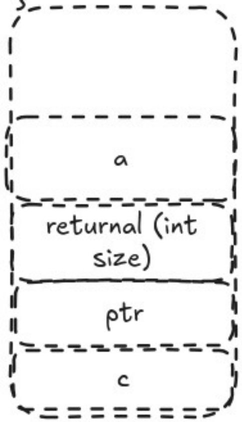
B.- Memory Map and Memory Image

- **Memory map**
  - Total Address Space of the Process
  - With Virtual Memory, these are virtual addresses
- **Memory Image**
  - Regions or segments that the process is authorized to use
  - With Virtual Memory, regions are divided into page sizes






```
int main() {
    unsigned char c;
    unsigned int *ptr = malloc(10);
    foo()
return 0;
int foo(){
    int a = 100;
    return a;
}
```



C.- Operating System Tables Table of Processes – BCP Information

BCP - Block Control Process

Identification Information

PID – Process Identifier

Father's PID

User ID (Real UID, Effective UID)

Group Identifier (Real GID, Effective GID)

- Process Group Identifier

Processor Status

- Contains the initial values of the processor state or its value at the instant the process was interrupted

Address Space Identifier Record

C.- Operating System Tables Table of Processes – BCP Information II

Process Control Information

- Planning & Status Information
- Process Status (Blocked, Ready, or Running)
- Event where you wait for the process when it's blocked
  - Planning Information: Priority and Standby Time
- Memory Image: Description of the memory regions allocated to the process
- Allocated resources, such as
- Open files (table of descriptors)
- Current Directory, Root Directory, (UNIX) Filemask
- Assigned Ports
  - Synchronization resources (semaphores, locks, etc.)

C.- Operating System Tables
Table of Processes – BCP Information III

Process Control Information

- Inter-process communication. Space to store signals and some message sent to the process
- Signals (UNIX)
  - Armed Signals
  - Signal Mask
- Timer
- Accounting Information (Resource Usage)
- Processor Time Consumed
- I/O Operations Performed
  - Limits on resource usage

C.- Operating System Tables Table of Processes – BCP Information III

Process Control Information

- Inter-process communication. Space to store signals and some message sent to the process
- Signals (UNIX)
  - Armed Signals
  - Signal Mask
- Timer
- Accounting Information (Resource Usage)
- Processor Time Consumed
- I/O Operations Performed
  - Limits on resource usage

C.- Operating System Tables Information outside the BCP

Justification

- For Implementation Reasons (Efficiency)
- To share it with other processes

Process Creation Operation

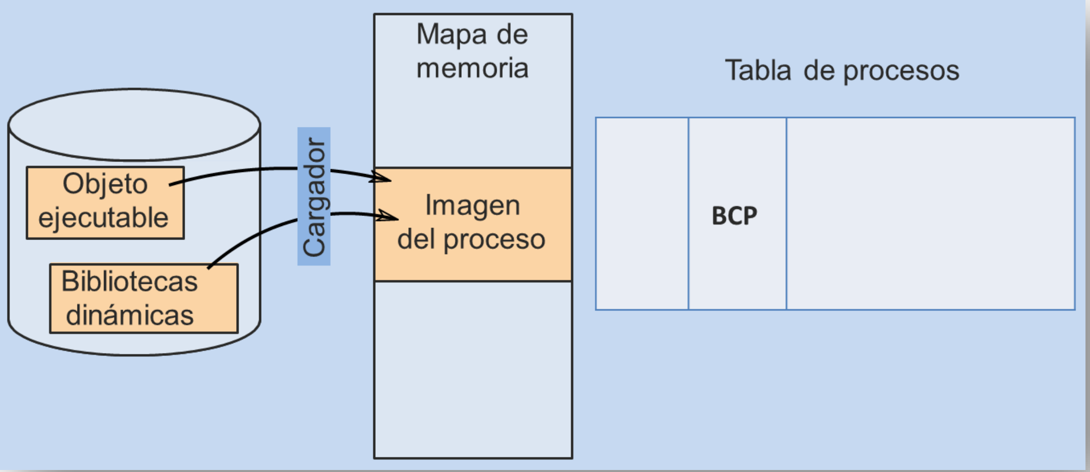
Create the Process Memory Image

Select Free BCP

Fill out the BCP

Load the Text Region (Code) and Data Region

- Create the Stack Region
  - The initial stack will have the process environment and program invocation paramet



UNIX Services
Process Management

UNIX Process Management Services
Target Domain

Understanding Process Identification Services
Understand process environment management services
Understanding the fork service
Understanding the exec service
Understanding the wait and waitpid services
Understanding the exit service

- Define Orphan and Zombie Processes

Process Management
Services Identification

- **getpid**
  - Returns the process ID
- **getppid**
  - Returns the identifier of the parent process
- **getuid** **getgid**
  - Return the real user ID and the real group
- **geteuid** **getegid**
  - Return the effective user and effective group ID

Process Management Services Identification II

```c
#include <stdio.h>
#include <unistd.h>
int main() {
    // Get current process ID
    pid_t pid = getpid();
    // Get parent process ID
    pid_t ppid = getppid();
    // Get user ID
    uid_t uid = getuid();
    // Get group ID
    gid_t gid = getgid();
        printf("Process ID: %d\n", pid);
        printf("Parent Process ID: %d\n", ppid);
    printf("User ID: %d\n", uid);
    printf("Group ID: %d\n", gid);
    return 0;
}
```

Process Management Services Environment Variables

- `getenv`
  - Returns the value of the environment variable
- `putenv`
  - Sets the value of environment variables

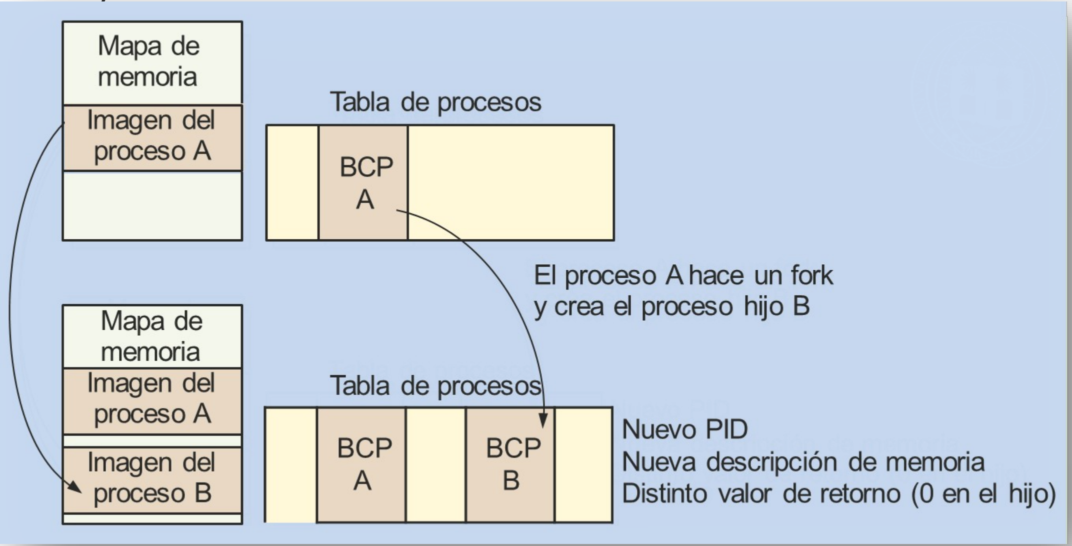
Environment Variable Process Management

```
import os

 // Set an environment variable
    setenv("MY_VAR", "Hello World", 1);
 // Retrieve the environment variable
    char *value = getenv("MY_VAR");
    printf("Value of MY_VAR: %s\n", value);
```

Creation
Process Management

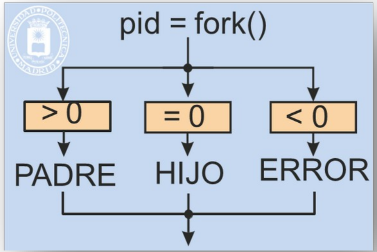
fork

- Create a child process. Returns 0 to the child process and the child pid to the parent process
- The son is a clone of the father, with three differences:
  - New pid
  - New Memory Image
  - Fork Return Value 0



Process Management Services Creation II

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
    pid_t pid = fork();
    if (pid > 0) {
        printf("Parent process with PID: %d, my child's PID is: %d\n", getpid(), pid);}
    else if (pid == 0) {
        printf("Child process with PID: %d, my parent's PID: %d\n", getpid(), getppid());}
    else {
        // An error occurred
        printf("Fork failed.\n");
        return 1;}
    printf("This line is executed by both processes\n");
    return 0;
}
```



fork - Paint the process hierarchy

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    // Attempt to fork the process
    pid_t pid = fork();

    if (pid == -1) {
        // Handle the error case
        fprintf(stderr, "Error fork\n");
        return 1;
    }
    else if (pid == 0) {
        // Child process code
        printf("Child process\n");

        pid_t child_pid = fork();

        if (child_pid == -1) {
            fprintf(stderr, "Error fork\n");
            return 1;
        }
        else if (child_pid == 0) {
            printf("hola\n");
```

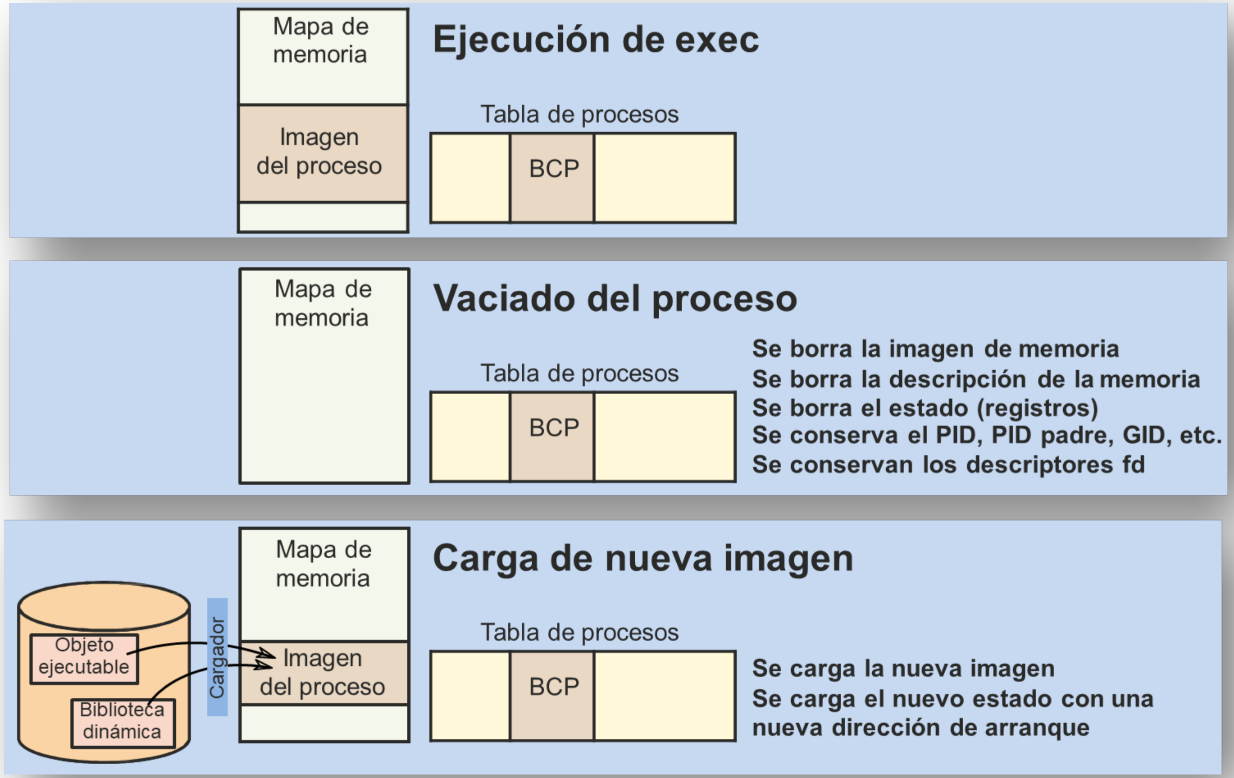
Fork Exercise II

What hierarchy of processes is left in this case?

What values are printed on the screen? ¿In what order?

```
a=1
os.fork()
a=a+1
os.fork()
a=a+1
print(a);
```

Process Management Services



Process Management Services

Exec

- Allows a process to go on to run another program (code). The pid doesn't change

Process Management Services

Función	pathname	filename	Arg List	argv[]	environ	envp[]
execl	X		X		X	
execlp		X	X		X	
execle	X		X			X
execv	X			X	X	
execvp		X		X	X	
execve	X			X		X
execvpe		X		X		X
Letra		p	l	v		e

Process Management Services

```
int main() {
    // Attempt to fork the process
    pid_t pid = fork();

    if (pid == -1) {
        // Handle fork failure
        fprintf(stderr, "Error fork\n");
        return 1;
    }
    else if (pid == 0) {
        // Child process
        printf("Child process\n");

        // Create a grandchild process
        pid_t child_pid = fork();

        if (child_pid == -1) {
            fprintf(stderr, "Error fork\n");
            return 1;
        }
        else if (child_pid == 0) {
```

Process Management Services Termination

- `wait`
  - Wait until a child process finishes (the first one to finish)

  `waitpid`
  - Wait until the process pid is finished
- `exit`
  - Completes the execution of a process by indicating the completion status of the process
  - Status value is sent

Process Management Services Termination II

The status variable in wait and waitpid

Status contains two interesting values

- **What happened to the child process:** it ended successfully or by receiving a signal
- **How the child process ended:** The process exit value or signal number that caused the end

- Macros Defined on the Status Variable
  - **WIFEXITED**(status): positive value if the child finished normally
  - **WEXITSTATUS**(status): value returned by the child process (exit or return) if finished normally
  - **WIFSIGNALED**(status): positive value if the process was terminated by the receipt of a signal
  - **WTERMSIG**(status): Number of Signal That Caused Process Termination

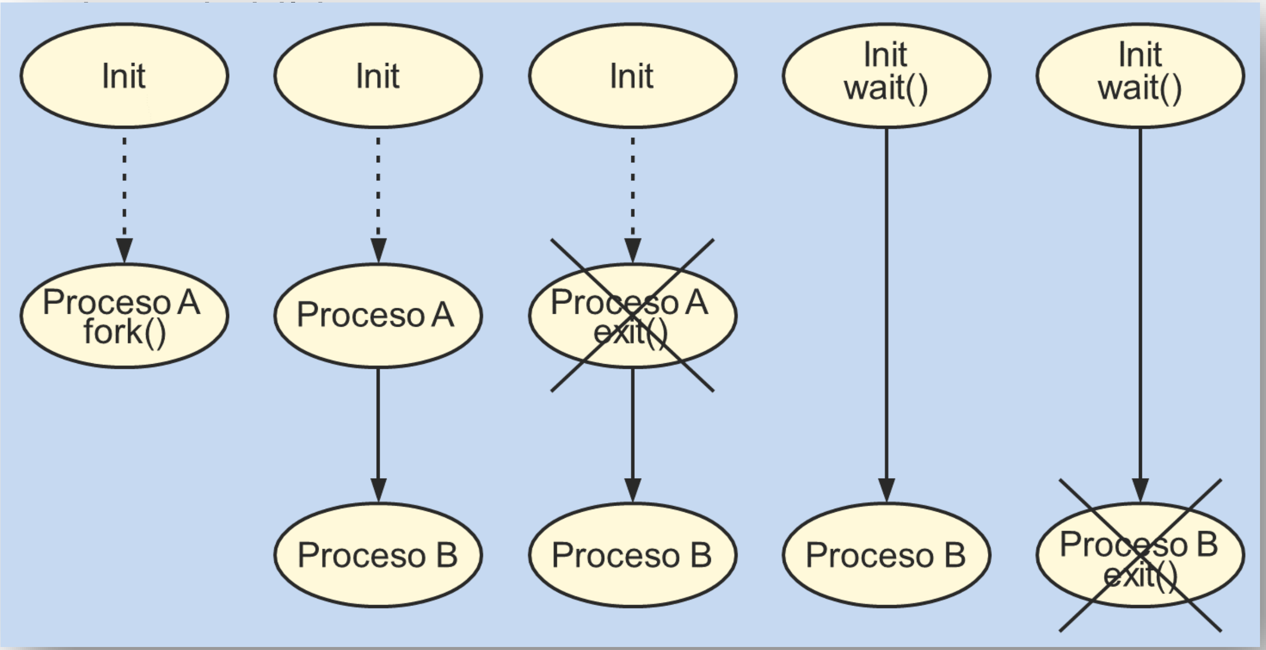
Process Management Services Termination III

```
pid = os.fork()

if pid == 0:  # Child process
    # Child process does something that could cause it to be killed
    print(f"In Child Process: PID= {os.getpid()}")
    # Note: Here we would have the child process do something that could
    # potentially cause it to be terminated by a signal, for demonstration.
else:  # Parent process
    pidChild, status = os.waitpid(pid, 0)  # Wait for the child process to finish

    if os.WIFSIGNALED(status):
        # If the child process was terminated by a signal, get which signal
        term_signal = os.WTERMSIG(status)
        print(f"Child process was terminated by signal: {term_signal}")
    elif os.WIFEXITED(status):
        # If the child process exited normally, get the exit status
        exit_status = os.WEXITSTATUS(status)
        print(f"Child process exited normally with status: {exit_status}")
    else:
        print("Child process did not exit normally or by a signal")

# This would be executed by the parent process after the child status has been handled
print(f"Back in Parent Process, PID= {os.getpid()}, after handling child process status")
```

Orphan Process

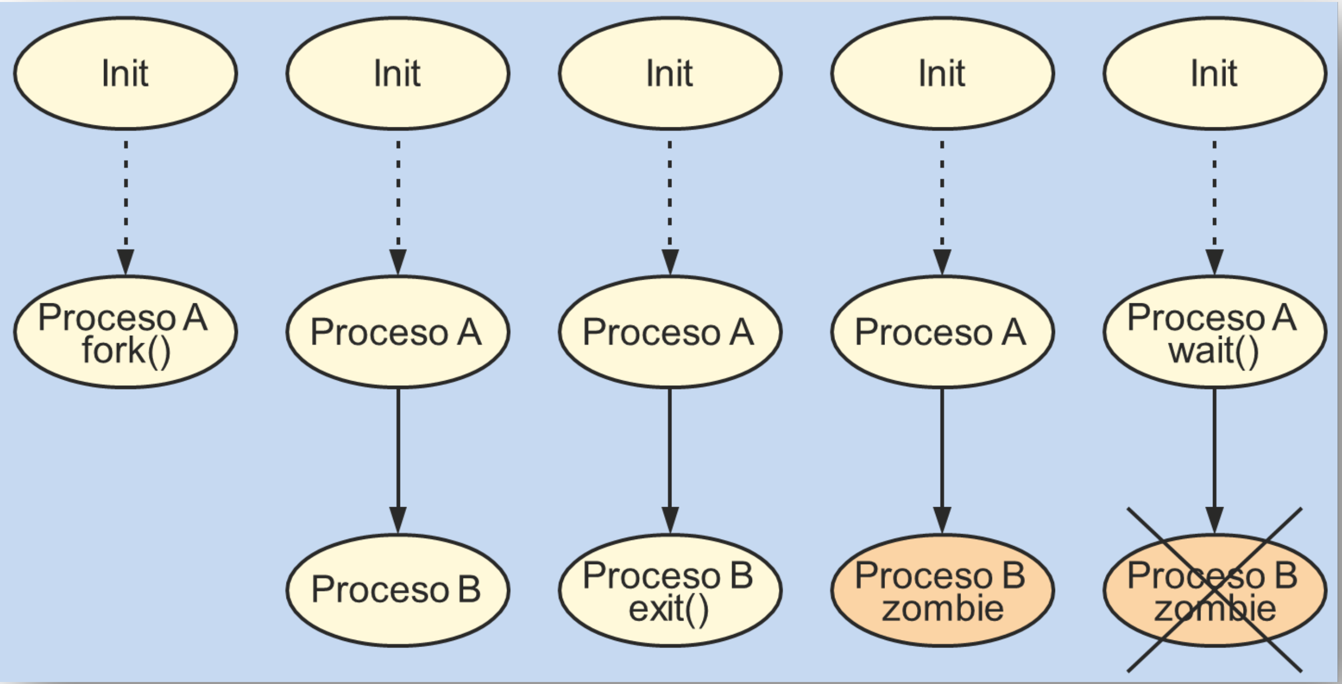
The parent process dies without waiting for the child process, which becomes an orphan

INIT accepts

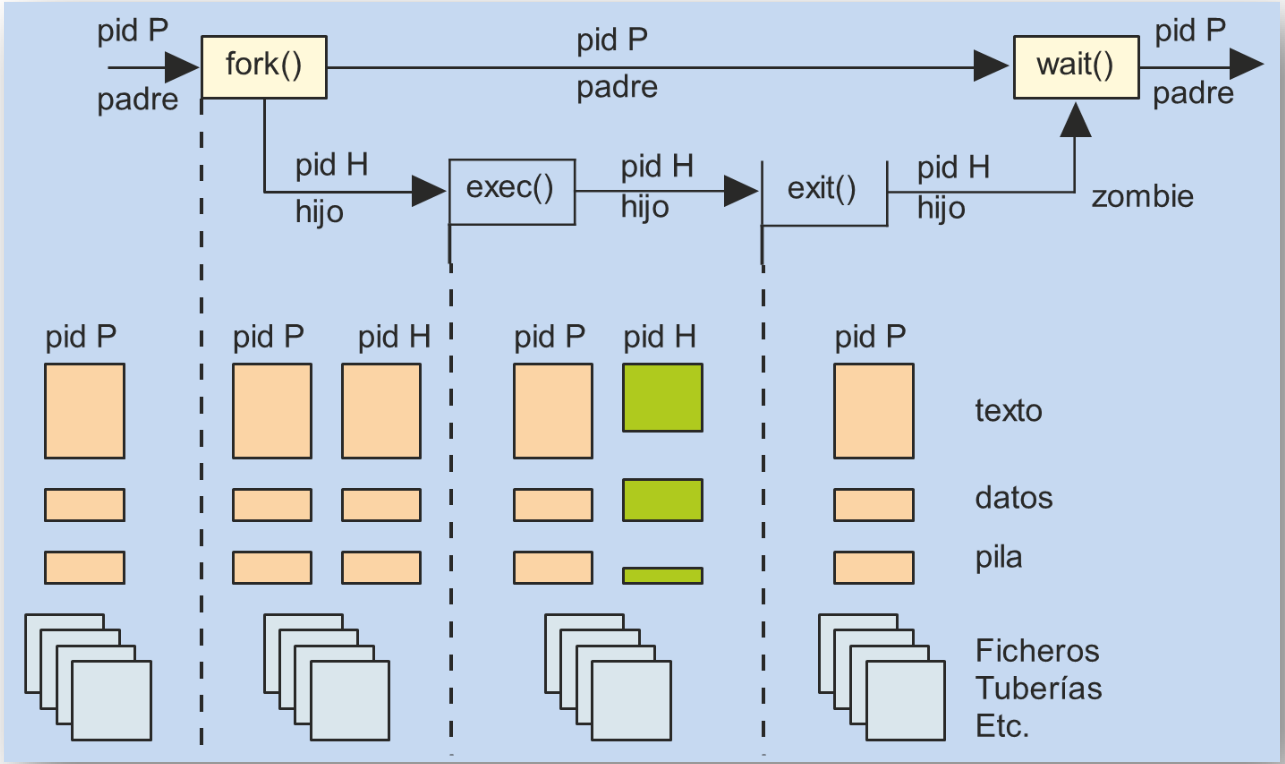


Zombie process

The child dies and the father doesn't wait



Zombie process



Exercise Orphans & Zombies

Can an orphaned process also be a zombie?

Can a zombie process also be an orphan?

Signals & Timers

Signals & Timers
Target Domain

Understanding the Signals
Understanding the Origin of Signals
Understand the actions of the process with respect to signals
Understanding Signal Blocking Through Masks
Understanding the Different Alarms and Timers

- Understanding the Different Types of Signals

Signals

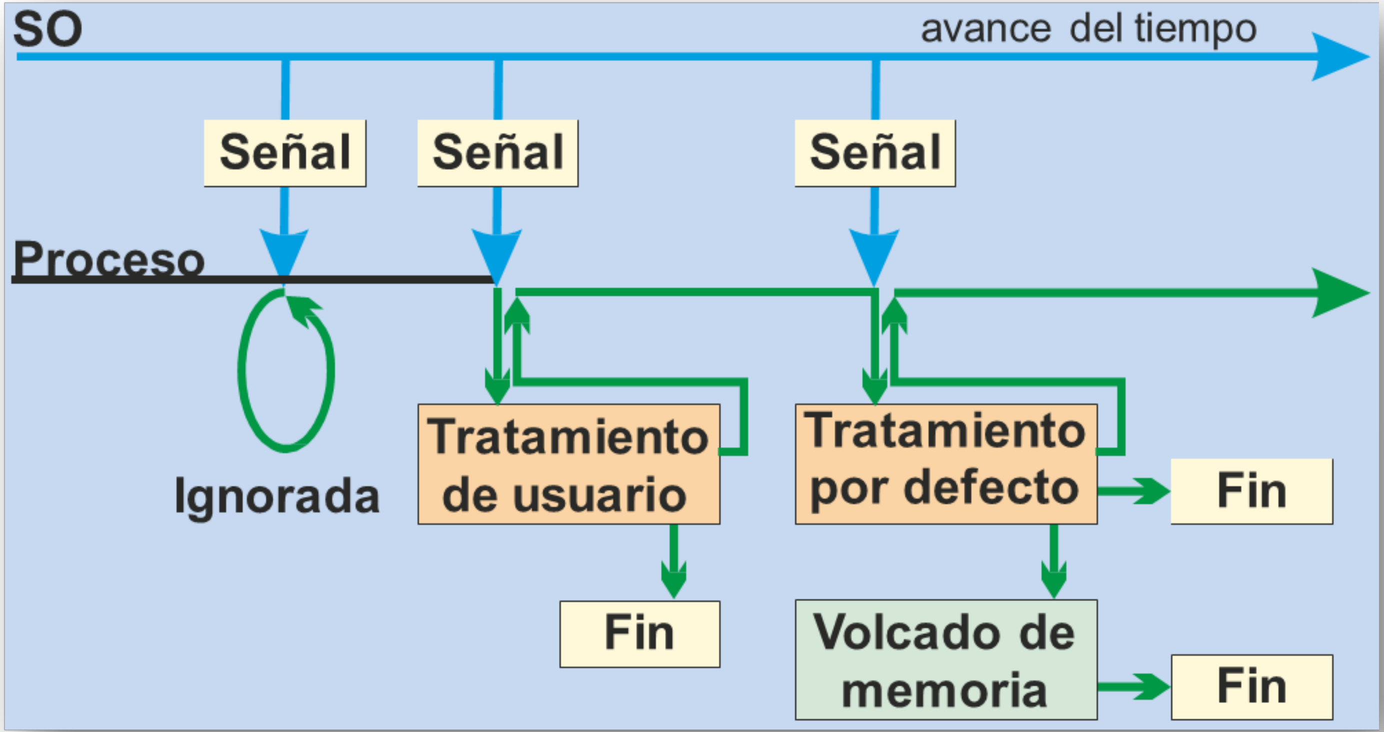
- The OS notifies a process of the occurrence of a certain event by means of the signal mechanism
- **From a process point of view, a signal**
- It's an event that receives (through the OS)
- Interrupt the process
- It transmits very limited information to you (a number, which identifies the type of signal)
  - A process can also send signals to other processes (in the same group), using the kill() service
- **From the OS point of view**
  - A signal is sent to a single process
  - Origin
  - SO → process
    - process → process

Origin of Signals

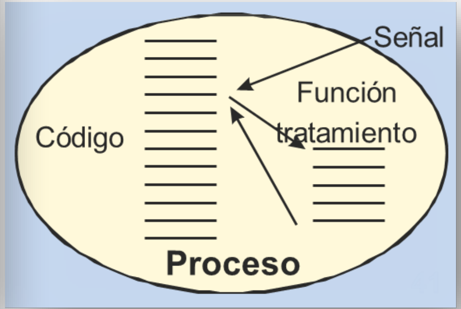
There are many types of signals, depending on their origin

- **Caused by HW exceptions, e.g.**
- Ilegal Instruction
- Memory Violation
  - Overflow in arithmetic operation
- **Caused by Interruptions, e.g.**
- Timer expires
  - Abort Process from Keyboard (ctrl + C)
- **Originated by another process, using the kill service**

Signals – Process Actions



Ignore
Handle
Default Action



Signals – Actions of the Process II

- **Ignore**
  - The process may have instructed the OS to ignore that type of signal (sigaction service / signal service)
- **Handling the Signal**
  - The process tells the OS the function to execute for that type of signal (sigaction service / signal service)
  - The OS emulates for the process an interruption whose treatment is that function
- If the process has not indicated anything, a **default action** occurs
    - The process, in general, dies
    - There are some signals that are ignored or have another effect

Alarms & Timers

- The OS maintains a timer per process (on the BCP)
  - The process activates the timer (alarm service)
- The OS sends a signal to the process when its timer expires (SIGALRM signal)

Types of Signals

Some examples are

SIGALRM, end-of-timing signal
SIGFPE, erroneous arithmetic operation (end the process by default)
SIGILL, invalid hardware statement (end process by default)
SIGINT, Interactive Attention Signal (ctrl + C) (end process by default)
SIGKILL, termination signal (can't be ignored or armed, kills the process) (kill -9)
SIGQUIT, Interactive Termination Signal (ctrl + \) – Generates core archive
SIGSEGV, invalid memory reference (terminates process by default)
SIGTERM, termination signal (default kill signal). It is the one sent during the OS shutdown
SIGUSR1, application-defined signal
SIGUSR2, application-defined signal
SIGCHLD, indicates the termination of the child process (ignored by default)
SIGCONT, continue if the process is locked

- **SIGSTOP**, lock signal (cannot be ignored or armed) (ctrl + Z)

Fork and exec: what is inherited

	FORK	EXEC
Handled	✓	✗
Ignored	✓	✓
Alarm	✗	✓
Pending Signals	✗	✓

Signal & Timer Services

Signal & Timer Services Target Domain

Understanding the Signal Sending Service
Understanding how to Handle and Ignore a signal

- Understanding Alarm and Timer Services

Signal Sending Service

UNIVERSIDAD POLITÉCNICA DE MADRID

`kill(pid, sig)`

Sends the sig signal to the pid process

Example

```
os.kill(pid, signal.SIGTERM)
```

Handler, Ignore and default action

```
signal.signal(signal, handler)
```

- **Handler**
  - Name of Handler Function
  - signal.SIG_IGN – Ignore the signal
  - signal.SIG_DFL – Default action

Handler, Ignore and default action

```
signal.signal(signal, handler)
```

- **Handler**
  - Name of Handler Function
  - signal.SIG_IGN – Ignore the signal
  - signal.SIG_DFL – Default action

Handle and Ignore examples

```
import os
import signal
def signal_handler(signum, frame):
    print(f"Signal {signum} Captured")

signal.signal(signal.SIGINT, signal_handler)
```

```
import signal
import time  # Import the time module
signal.signal(signal.SIGINT, signal.SIG_IGN)
print("Program is running. Try pressing Ctrl+C. This program should ignore the signal and keep running.")
while True:
      signal.pause()
```

Alarm and Timing Services

- `signal.pause()`

Blocks the process until a signal is received

- `signal.alarm(seconds)`

Generates SIGALRM signal reception after seconds

signal.alarm(0) Cancels a previously set alarm

If an alarm is called when a previous alarm already exists, the previous alarm will be cancelled and the new alarm will be set

- `time.sleep(seconds)`

The process wakes up when the set time has elapsed or when a signal is received

Print message every 10 seconds

```
import signal
import time

# This function will be called whenever a SIGALRM
signal is received
def signal_handler(signum, frame):
    print("Printing this message every 10
seconds.")
    # Reschedule the alarm
    signal.alarm(10)

# Set the signal handler for SIGALRM
signal.signal(signal.SIGALRM, signal_handler)

# Schedule the first SIGALRM after 10 seconds
signal.alarm(10)
while True:
        time.sleep(1)
```

Timing Child Process

```
import os
import signal
import time

def child_process():
    print("Child Process Begins")
    time.sleep(20)
    print("Process hijo appointment")

def alarm_handler(signum, frame):
    print("Alarm signal (SIGALRM) received. Finishing the
process son.")
    os.kill(child_pid, signal.SIGTERM)

timeout = 10

pid = os.fork()
if pid == 0:
    child_process()
else:
    print(f"Parent: Child Process with PID {pid}
Started")
    signal.signal(signal.SIGALRM, alarm_handler)
    signal.alarm(timeout)
          _, status = os.waitpid(pid, 0)
     print(f"child process ended with status {status >>
8}")
```

Special Processes

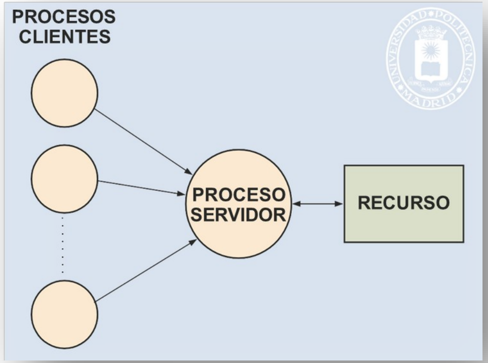
Special Processes
Target Domain

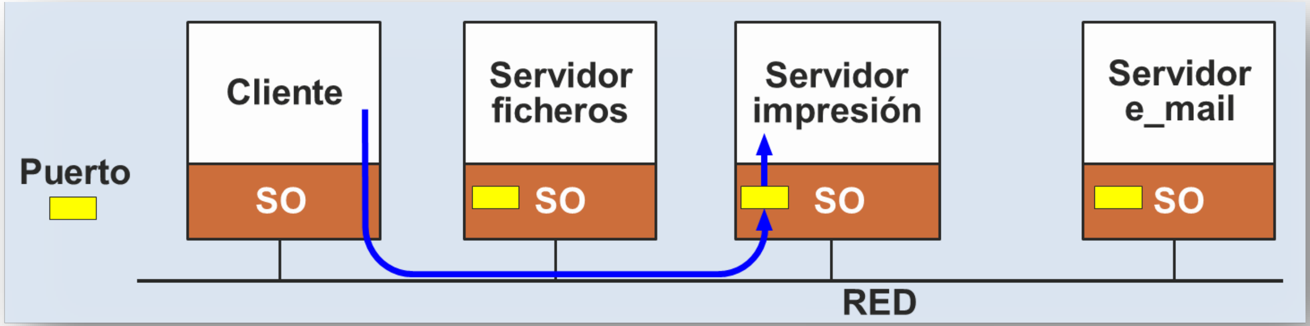
Understand server processes and their characteristics

- Understanding Demon Processes and Their Characteristics

Server Process

- Respond to requests
- Loop
- Order Reading
- Order Execution
    - Customer Response





Parallel Server Process

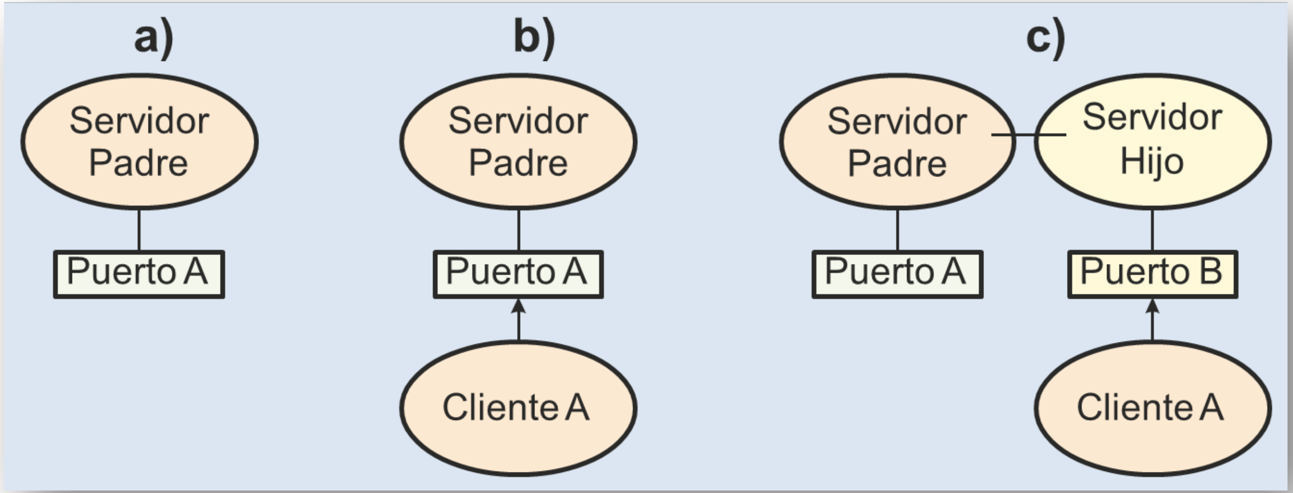
Loop

Order Reading

Assigning a new port and child launch

Child process that serves the customer

- Back to top



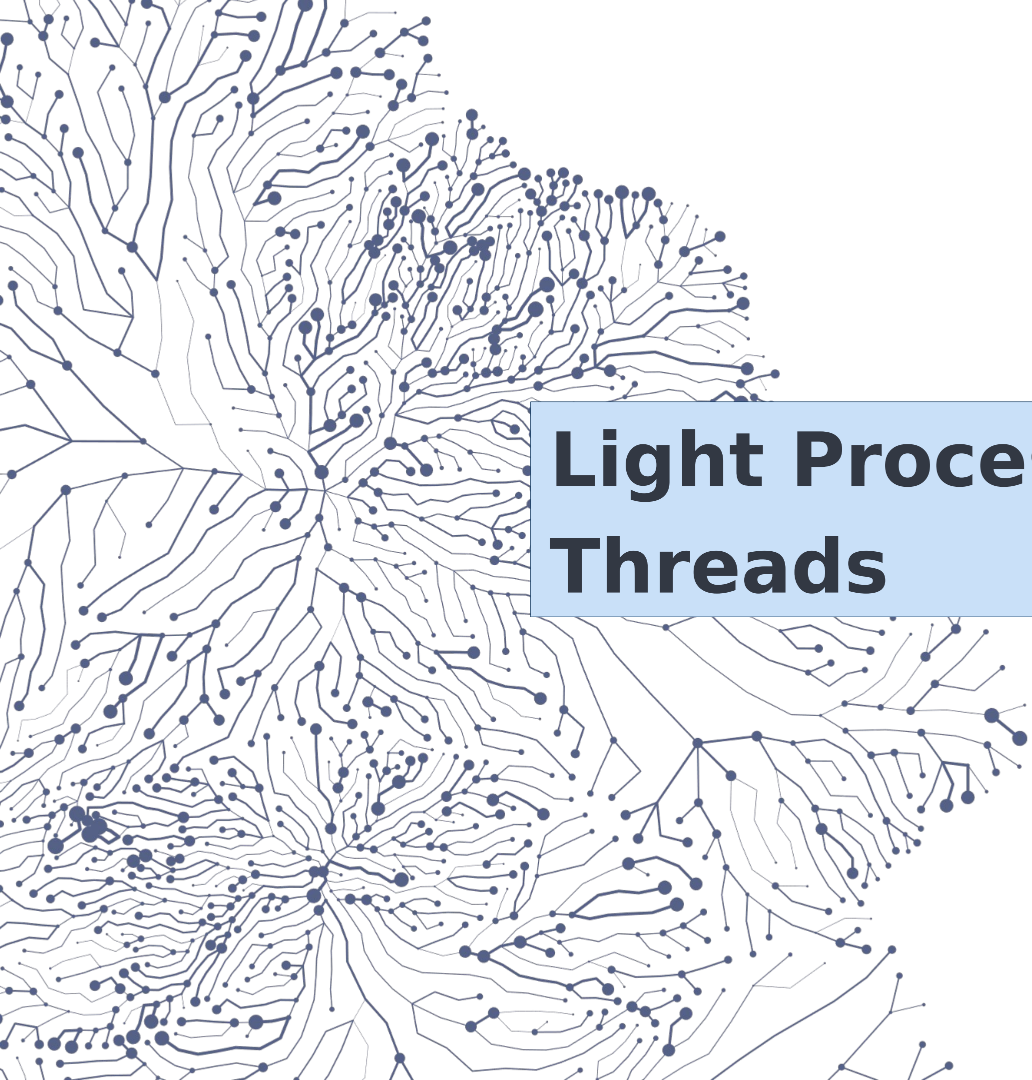
Demon Process

- A program that runs in the background on an operating system without direct user interaction
- Not associated with a terminal or login process
  - They are usually started automatically during the startup of the operating system
- They typically run as services, providing functionality that is used by other programs or services
  - Or, they perform a task on a regular basis
- They don't die
- They are normally waiting for an event
- They don't do the work directly, they launch other processes or light processes

Examples of Linux Servers and Demons

LPD - Line Printer Daemon
Inetd - boots network servers: ftp, telnet, http, finger, talk, etc.
SMBD - Samba Daemon
ATD - Execution of tasks at certain times
Crond - Execution of Periodic Tasks
nfsd - NFS server

- httpd - WEB server



Light Processes or Threads

Light Processes or Threads
Target Domain

Thread concept
Understand thread-level and process-level information
Analyze the advantages and disadvantages of using threads vs. processes

- Understanding the Different Software Architectures with Threads

Threads or Light Processes

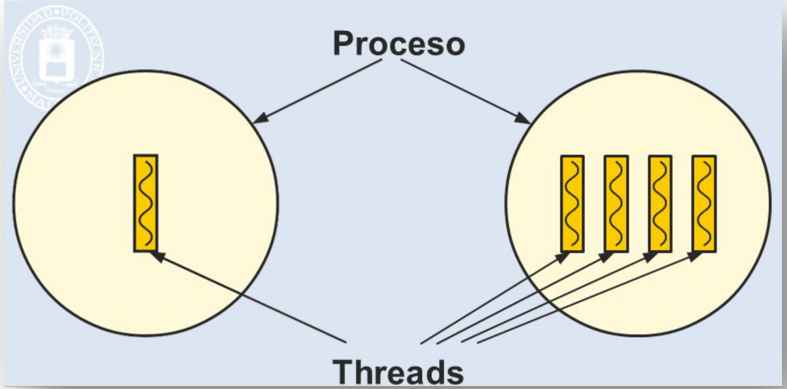
- Concurrent Application Performance with Processes
- Time consumption in the creation and completion of processes
- Time-consuming context switching
  - Resource sharing issues
- A process can have multiple threads, each of which can run simultaneously and independently
- The thread is the basic unit of CPU utilization (thread as a planning unit)
- They allow a program to perform multiple tasks at the same time. E.g. in a web browser
- A thread is responsible for loading and displaying content on the screen

Threads or Light Processes

- Most modern OSs provide processes with multiple threads inside
- The OS has an extended BCP structure to support the information of the different threads (BCT)

Each thread has
- Thread ID
- Registry Set (especially the program counter)
- Stack
- State (running, ready, or locked)

Share with the rest of the threads
- Memory Image
- Global Variables
- Signals, semaphores and timers
- Open Files



Pros and Cons

Pros

Responsiveness

- Increased interactivity by separating user interactions from processing tasks into different threads

Resource Sharing

Threads share most resources automatically

- Easier synchronization between lightweight processes

Resource Economy

- Creating a process is much more expensive than creating a thread (30 vs 1)

Use on multiprocessor architectures

- Higher concurrency by assigning different threads to different processors

Cons

Access to shared data needs to be synchronized

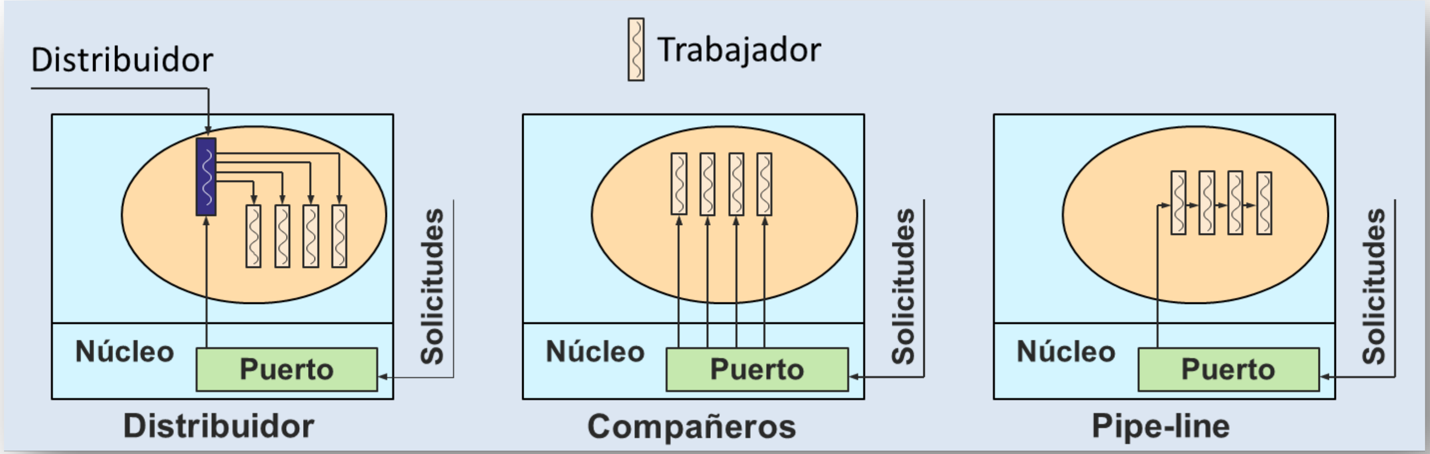
- Access to shared areas is organised by 'mutex'

If a thread fails, the entire process dies

sw architectures with Threads

Examples of thread-based software architectures

- Distributor with multiple worker threads (created for each service request or previously created)
- 'Companion' Threads
- Segmented scheme or pipe-line (type of manufacturing chain)



UNIX Threads Services

UNIX Services for Threads
Target Domain

Understand the difference between joinable and detached threads

Understand system calls to work with thread attributes

Understand System Calls for Creating and Terminating Theards

- Understand the development of programs where several threads collaborate to solve a problem

Threads in Python: Create and Join

- Python provides support for thread creation and management through the threading module.
- To create a new thread, you can use the Thread class of the threading module.
- To start running a thread, call the start() method of the Thread object.
- To wait for a thread to finish its execution, call the join() method of the Thread object.

Joinable and detached threads

Joinable (Non-Daemon) Threads

- **Definition**: Joinable threads, also known as non-daemon threads, are the default type of threads in Python. These threads must complete their execution before the main program can terminate.
- **Termination**: The main program waits for all joinable threads to finish their tasks before it terminates. This behavior ensures that all critical tasks being executed by the threads reach completion.
- **Usage**: They are typically used for tasks that must be completed before a program ends, such as finalizing resources, saving state, or completing any background computations that are essential to the program's operation.
- **Control**: Developers have explicit control over joinable threads by using the `join()` method, which blocks the calling thread (typically the main thread) until the thread whose `join()` method is called is terminated.

Detached (Daemon) Threads

- **Definition**: Detached threads, or daemon threads, are specifically marked as daemons. This means they do not need to complete for the main program to terminate. The program can exit when only daemon threads are left running.
- **Termination**: These threads are abruptly stopped when the main program exits. As a result, any cleanup or finalization logic in daemon threads may not execute if the main program terminates.
- **Usage**: Daemon threads are useful for tasks that run in the background and do not need to complete for the program to finish. Common examples include logging, monitoring, and background computations that can safely terminate at any point.
- **Control**: To create a daemon thread in Python, you set the thread's `daemon` attribute to `True` before starting the thread. Once the main program exits, any running daemon threads are automatically killed.

Thread joinable

```
import threading
import time

# Define a function for the thread
def print_numbers():
    for i in range(5):
        time.sleep(1)  # Simulate a time-consuming task
        print(f"Number {i}")

# Create a thread that will execute the print_numbers function
thread = threading.Thread(target=print_numbers)

# Start the thread
thread.start()

# Wait for the thread to complete
thread.join()

print("Thread has finished execution.")
```

Thread detached

```
import threading
import time

def background_task():
    print("Starting background task")
    time.sleep(2)  # Simulate a task taking some time
    print("Background task completed")

# Create a daemon thread
daemon_thread = threading.Thread(target=background_task)
daemon_thread.daemon = True  # Set the thread as a daemon

# Start the daemon thread
daemon_thread.start()

print("Main program continues to run in parallel.")
# Main program waits for a moment to see the output of the background
task
time.sleep(1)
print("Main program is exiting.")
```

Threads with arguments

```
import threading

# Define a function to run in a new thread
def print_numbers(a, b):
    for i in range(a, b + 1):
        print(i)

# Create a thread that runs the 'print_numbers' function with arguments
my_thread = threading.Thread(target=print_numbers, args=(1, 10))
my_thread2 = threading.Thread(target=print_numbers, args=(11, 21))

# Start the threads
my_thread.start()
my_thread2.start()

# Wait for the threads to complete before continuing with the rest of the
program
my_thread.join()
my_thread2.join()

print("Threads have finished execution.")
```