UT 1 - Multiproceso

Programación de Servicios y Procesos

Curso 2025-26

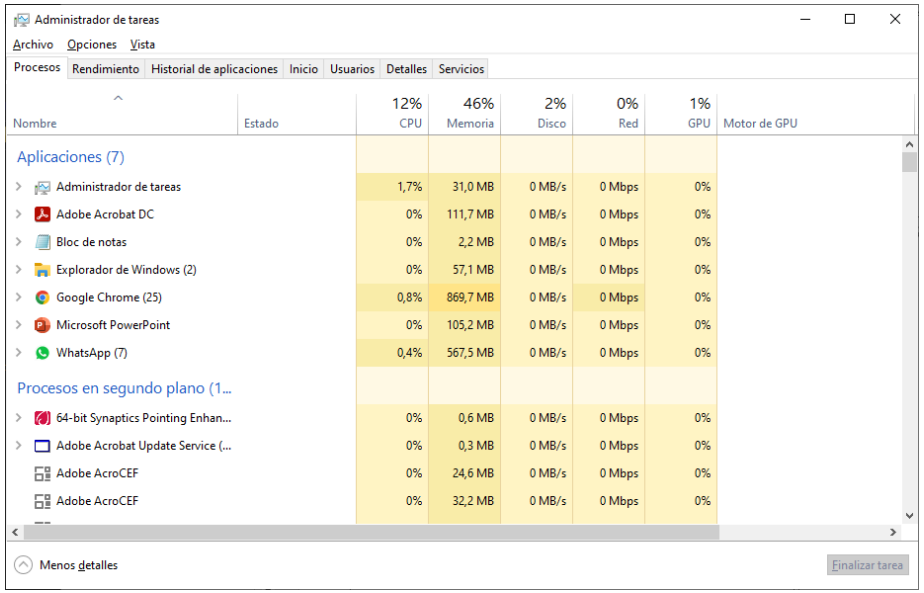
Profesor: Agustín González-Quel

Objetivos de la unidad

- Comenzar a hacer programas que no tienen porque ejecutarse secuencialmente, es decir, una sentencia detrás de otra.
- Comenzar a trabajar con el lenguaje Python
- Conocer librerías específicas para trabajar multiproceso en Python
- Conocer mecanismos de comunicación entre procesos

- ¿Para qué sirve una implementación multiproceso?
  - Varias pestañas de un navegador Web
  - Envío de trabajos a impresión
  - Descarga de archivos (aunque hay mejores maneras de hacerlo)

Programas y Procesos



Programa / Aplicación / App:

- Conjunto de instrucciones que resuelve una tarea específica en un ordenador.

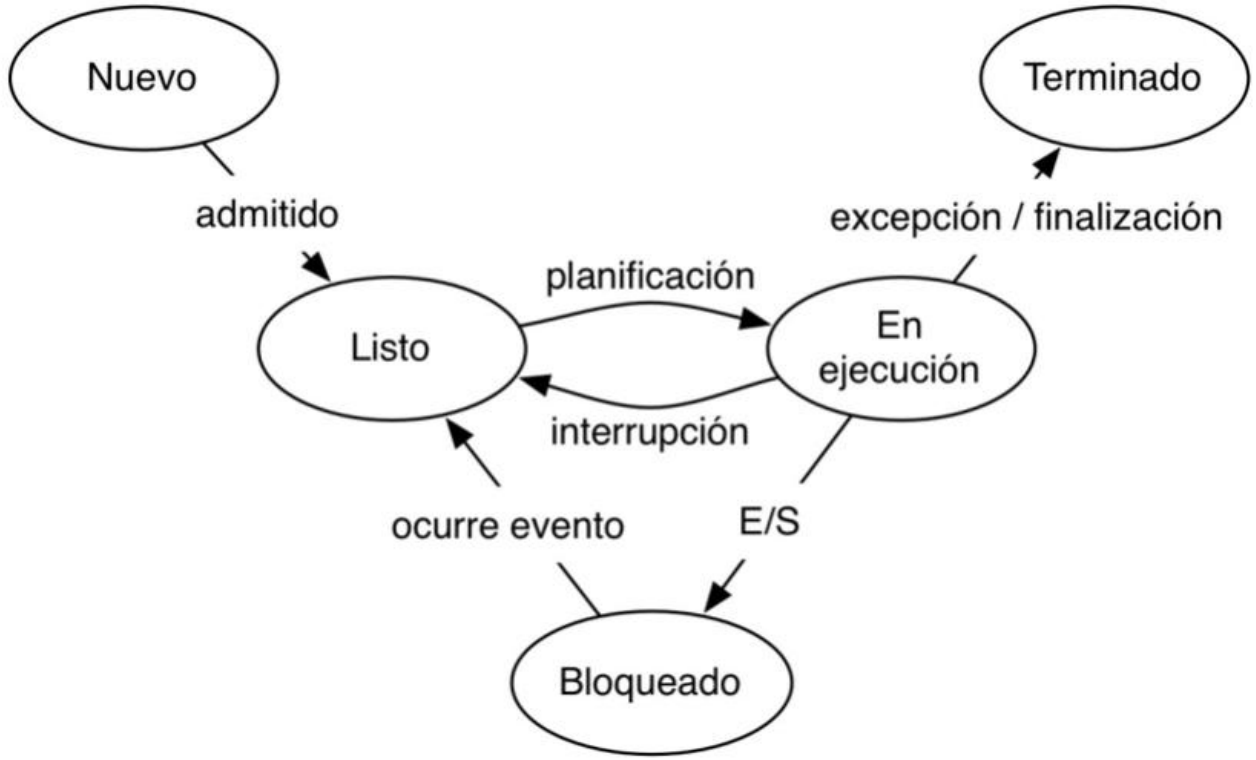
Proceso:

- Programa en ejecución.
- Se compone de
  - Instrucciones
  - Datos (variables, etc.)
  - Contexto / Registros del microprocesador (contador de programa, etc.)
  - Otros datos: prioridad, etc

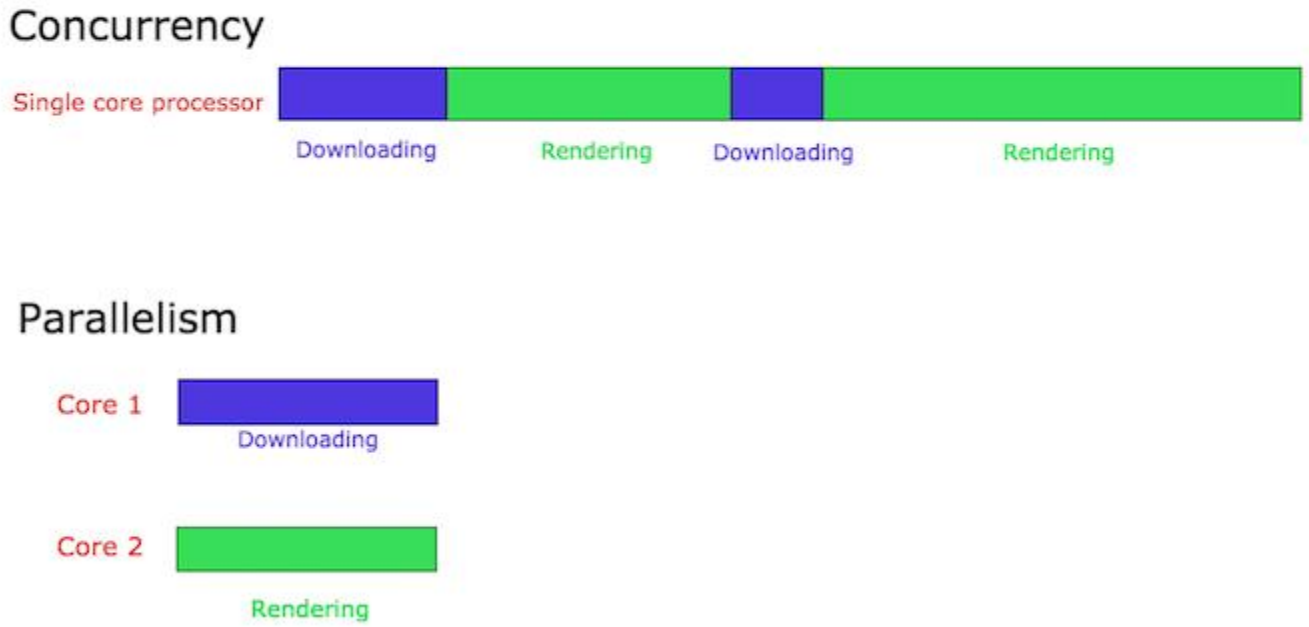
```
ubuntu@ip-172-31-46-157: ~
ubuntu@ip-172-31-46-157:~$ ps -ax | grep python
    570 ?        Ss     0:00 /usr/bin/python3 /usr/bin/networkd-dispatcher --run-startup-triggers
    729 ?        Ssl    0:00 /usr/bin/python3 /usr/share/unattended-upgrades/unattended-upgrade-shutdown --wait-for-signal
    779 ?        S      0:00 -bash -c cd /home/ubuntu/tracker && python3 main.py &
    780 ?        Sl    97:28 python3 main.py
   2144 pts/0    S+     0:00 grep --color=auto python
ubuntu@ip-172-31-46-157:~$
```

Estados de un proceso

- **Nuevo**: el proceso está siendo creado a partir del fichero ejecutable.
- **Listo**: el proceso no se encuentra en ejecución aunque está preparado para hacerlo. El sistema operativo no le ha asignado todavía un procesador para ejecutarse.
- **En ejecución**: el proceso se está ejecutando. El sistema operativo utiliza el mecanismo de interrupciones para controlar su ejecución.
- **Bloqueado**: el proceso está bloqueado esperando que ocurra algún suceso. Cuando ocurre el evento que lo desbloquea, el proceso no pasa directamente a ejecución sino que tiene que ser planificado de nuevo por el sistema.
- **Terminado**: el proceso ha finalizado su ejecuci y libera su imagen de memoria.



Gráficamente

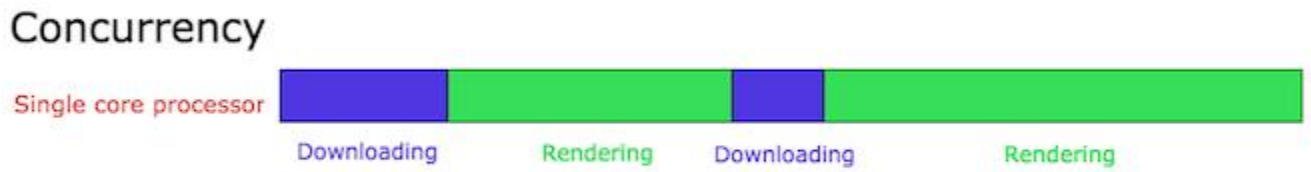


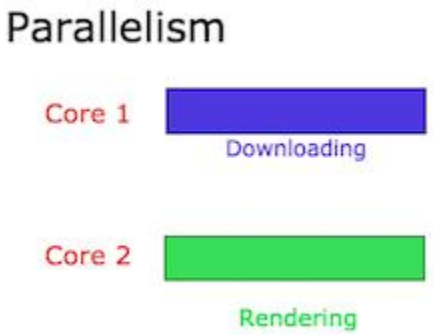
Programación concurrente

- Hasta ahora los programas que habéis implementado se ejecutaban de forma lineal.
- En este módulo veremos mecanismos para la programación concurrente

  Se conoce por programación concurrente a las técnicas de programación que se usan para trabajar con **tareas que se ejecutan a la vez.**

  Esta actividad simultánea puede provocar problemas que hay que resolver, especialmente **problemas de comunicación y sincronización** entre las tareas (sean procesos o hilos).





Cómo se consigue concurrencia / paralelismo

Procesador

- Número de cores (núcleo): Cada core ejecuta una instrucción en cada momento.

Sistema Operativo

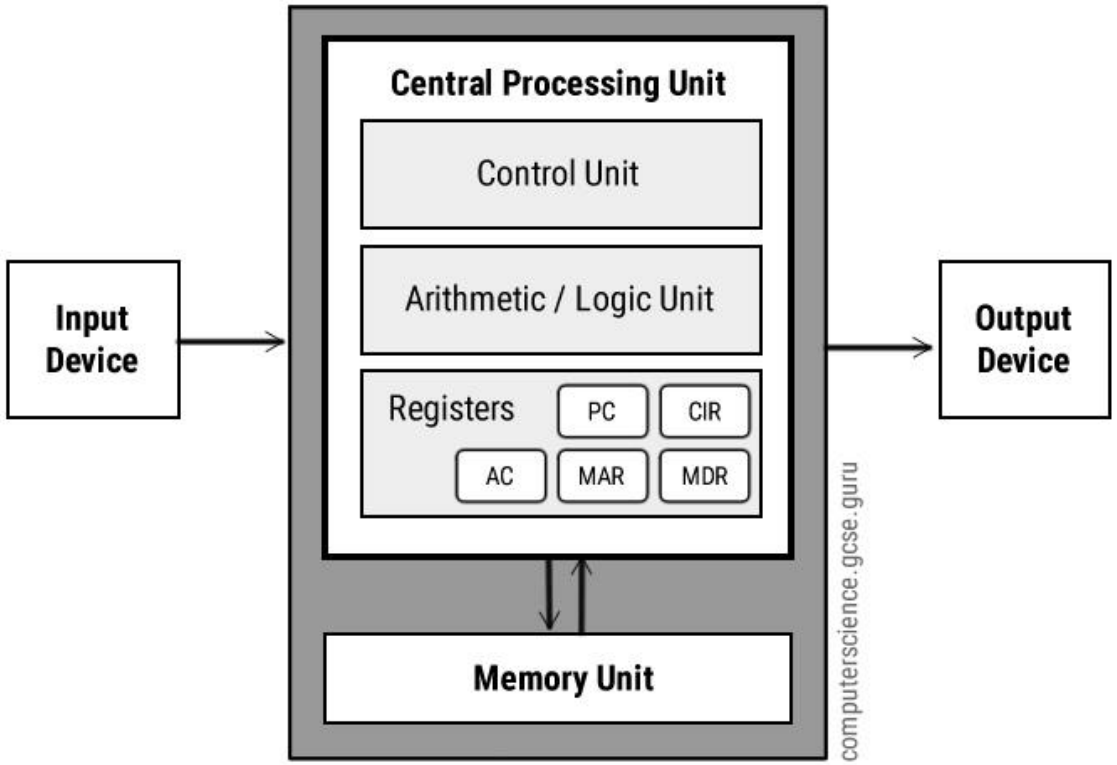
- Los S.O. modernos son multitarea.

Lenguaje de programación

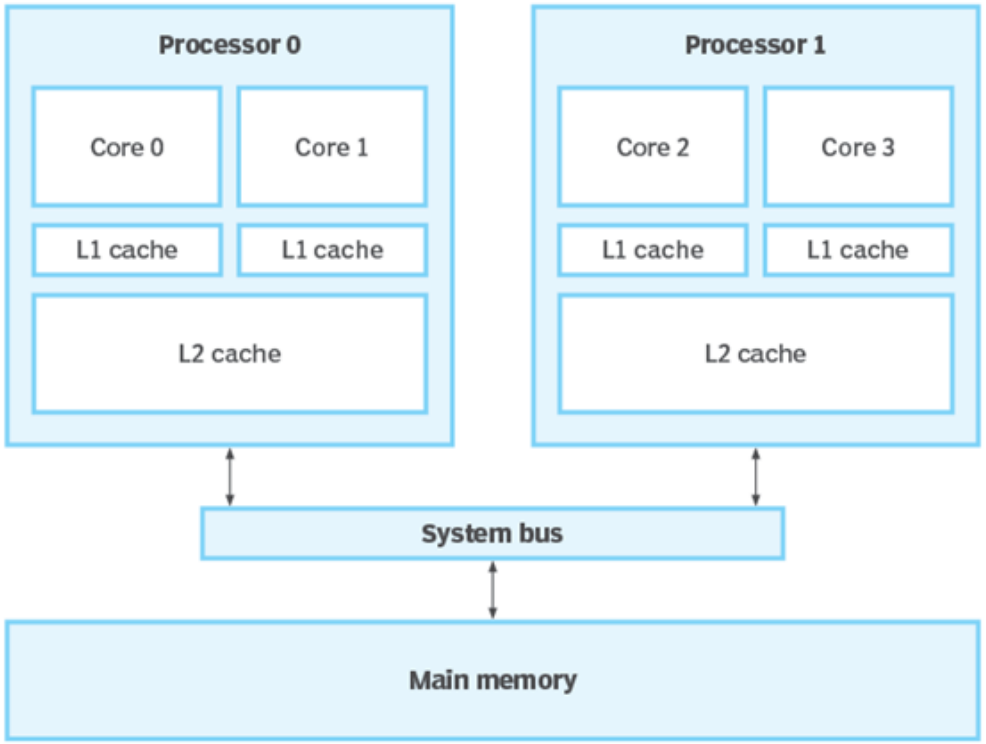
- Dependerá de cómo se construya cada programa para aprovechar las capacidades de la máquina
- Existen mecanismos en lenguajes como C, Java, Python, etc. Para aprovechar los recursos de la máquina.

Estructura de un procesador

Arquitectura Von Neumann



Procesador con varios cores



Programación concurrente vs paralela

Concurrente

- El sistema **sólo ejecuta un proceso en cada momento.**
- El S.O. selecciona en cada momento uno de los procesos que deben ejecutarse.
- La percepción del usuario es que varios procesos se ejecutan a la vez
- Cada cambio de proceso requiere un movimiento de toda la información de contexto: datos, posición del ejecutable, etc.

Paralela

- El sistema ejecuta **varios procesos en un mismo instante de tiempo.**
- Varios núcleos en un mismo procesador o varios procesadores.
  - Cada uno de ellos ejecuta un proceso en el mismo instante de tiempo.
  - Cada núcleo tiene registros para almacenar la información de contexto.
- El S.O. planifica los procesos de cada núcleo.

Sistemas distribuidos: varios ordenadores en red.

- Cada uno de los ordenadores tendrá sus propios procesadores y su propia memoria.
- La programación distribuida permite usar dispositivos de forma paralela, lo que lleva a alcanzar elevadas mejoras en el rendimiento.
- Como cada ordenador posee su propia memoria, impide que los procesos puedan comunicarse entre si fácilmente, teniendo que utilizar otros esquemas de comunicación más complejos y costosos a través de la red que los interconecte (sockets, websockets, APIs, etc.)

Gestión de procesos

- El Sistema Operativo esta continuamente cambiando los procesos que se ejecutan en un procesador
  - Cuando un proceso cambia de modo (p.e. ejecución → listo) ha de guardarse en algún lado cómo estaba al pararse, para que cuando vuelva a estar en ejecución, sepa volver dónde estaba y como estaba.
- Esta información se llama comúnmente el **contexto del proceso**, y se guarda en un registro del sistema operativo llamado Bloque de control del proceso (BCP)
  - El BCP es pues un registro especial donde se agrupa toda la información que el S.O. necesita conocer respecto a un proceso particular.
  - Cada vez que se crea un proceso el sistema operativo crea el BCP correspondiente para que sirva como descripción en tiempo de ejecución durante toda la vida del proceso.
  - El BCP contiene (entre otros) una serie de datos como los siguientes:
    - Identificador del proceso (Process Identificator -**PID**-, de sus siglas en inglés). El número que le identifica
    - Estado del proceso por ejemplo, listo, en espera, bloqueado).
    - Contador de programa: dirección de la próxima instrucción a ejecutar.
    - Valores de registro de CPU. Se utilizan también en el cambio de contexto.
    - Espacio de direcciones de memoria (direcciones de memoria que usa el proceso, que es por ejemplo donde están sus variables).
- El sistema operativo gestiona los procesos mediante colas, prioridades, etc.

Situaciones asociadas a una ejecución paralela

Ejecución Simultánea

- Existencia de varios procesos interactuando a la vez

Indeterminismo

- dependiendo de cada posible ejecución se podrán obtener resultados diferentes

Interacción

- Los procesos pueden interactuar entre si pasándose información entre ellos

Comunicación

- Permite a la ejecución de un proceso influenciar en la ejecución del otro
- Por el uso de variables compartidas por todos los procesos
- Paso de mensajes entre los procesos

Sincronización

- Permite establecer cierto orden de ejecución de diferentes partes del programa
- Estableciendo mecanismos de Exclusión mutua
- Estableciendo mecanismos de Sincronización por condición

Temas a considerar

- Tiempo asignado a cada segmento de proceso, ni muy grande ni muy pequeño.
- Tiempo de cambio de contexto
- Bloqueo – dependencia entre procesos
- Planificación (Scheduler del SO): Se utilizan diferentes estrategias
  - FCFS (First Come First Served)
  - Prioridad
  - Short Job First. Problema, no se suele saber a ciencia cierta.
  - Round Robin
  - Combinación de los anteriores.
- Por qué entra o sale un proceso de la CPU
  - Se agota su tiempo asignado
  - Termina
  - Se suspende por espera en E/S o interrupción del sistema
  - Queda a la espera de un evento (sincronización)

Lenguaje Python

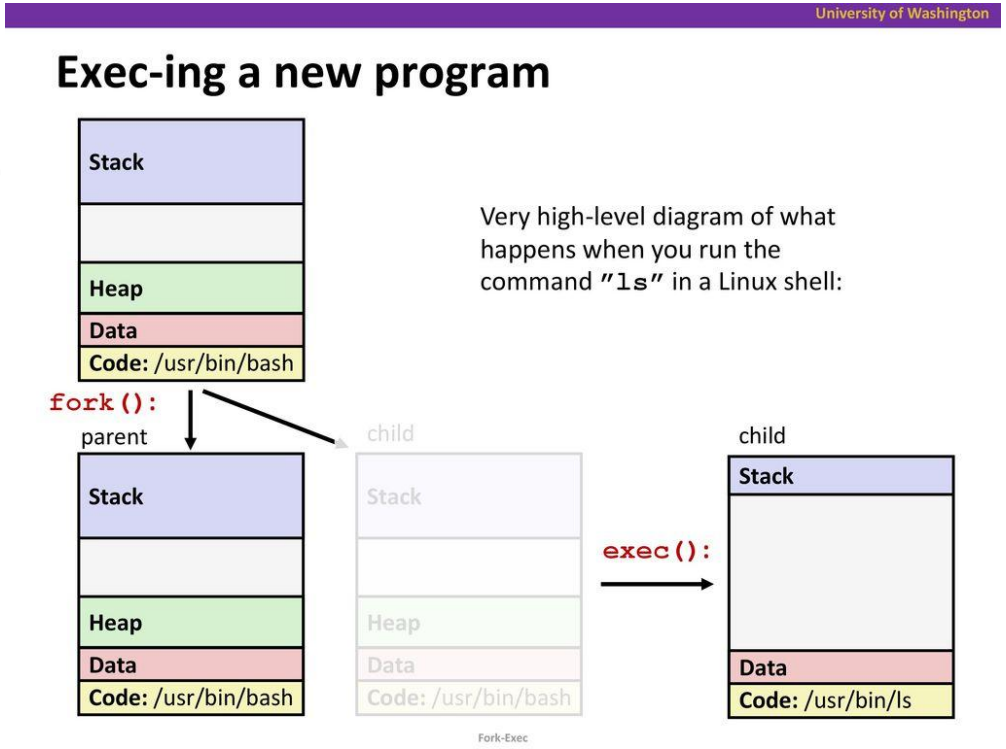
Fundamentos del lenguaje para trabajar con multiprocesos

Operaciones con procesos

1. Clonar un proceso: lanzar un proceso terminando el actual
2. Crear un proceso a partir de una función
   - Esperar a que acabe para continuar el flujo
   - Seguir con el programa de forma independiente
3. Crear un proceso desde un ejecutable
   - Esperar a que acabe para continuar.
   - Seguir con la ejecución independiente
4. Operaciones con procesos del sistema
   - Ver los procesos
   - Comprobar sus propiedades
   - Matar un proceso
5. Comunicación entre procesos

Independencia de procesos

- Cada proceso es una entidad independiente del Sistema Operativo
- Tiene por tanto su propio área de memoria a la que no podemos acceder desde otro proceso
- Si necesitamos compartir información será preciso un elemento externo: bases de datos, ficheros, colas de mensajes, sockets/websockets,



Fuente: https://github.com/Ixeoz/Timer-Old-Implementation/blob/main/Procesos/Procesos.md

1 - Clonar un proceso terminando el actual

- Puede hacerse con la librería os.
- Lanza un proceso que sustituye al actual que muere.

os

https://docs.python.org/3/library/os.html

- Muy buena para manejar temas del SO como ficheros, carpetas y demás.
- Por ejemplo, el directorio actual → `cwd = os.path.dirname(os.path.realpath(__file__))`
- Menos recursos para multiproceso. Permite arrancar procesos

Funciones

```
os.getpid() # Id de proceso
os.getppid() # Id del proceso padre
os.execvp()  # Ejecuta un nuevo proceso (hay varias versiones)
os.exit()   # Termina el proceso y devuelve un código al SO
```

Ejemplo: 01 - OS-lanzoExterno.py

```
os.execvp('notepad.exe', ['notepad.exe', 'C:/Users/profesor/Documents/mifichero.txt'])
```

Fork de procesos

- En los sistemas operativos multitarea, los procesos necesitan una forma de crear nuevos procesos, por ejemplo, para ejecutar otros programas.
- La forma más extendida de hacerlo en sistemas tipo Unix (Debian, Ubuntu, Suse, Red Hat, etc.) es mediante el comando fork
  - Fork crea un espacio de direcciones independiente para el proceso hijo.
  - El proceso hijo arranca con una copia exacta de todos los segmentos de memoria del proceso padre.

```
import os

newpid = os.fork()
```

En el proceso hijo, fork devuelve 0.

Ejemplo: 01 - OS-fork.py

Importante: en MS Windows, no funciona.

2 - Crear un proceso a partir de una función

Multiprocess

multiprocessing: Más compleja y completa

https://docs.python.org/3/library/multiprocessing.html

Tiene varias clases, pero usaremos solamente Process

Funciones

```
p = Process(target=hijo)  # crea un nuevo proceso que ejecutará la func hijo
p.start()                 # Arranca el proceso
p.join()                  # Detiene el flujo hasta que vuelve el proceso.
```

Ejemplo:

```
p = multiprocessing.Process(target=fun, args=…)
p.start()
```

Nota: args espera una tupla con los parámetros de entrada de la función hijo. Si solo tiene un parámetro, la tupla terminará en , → `(4,)`

2 – Creación de proceso a partir de función – ejemplo

Se arranca un proceso que ejecuta una función de nuestro programa

- Si queremos detener el proceso padre hasta que acabe el proceso hijo tenemos el método método join()

En el momento de crear el proceso se hace una copia de memoria.

- Hay dos formas de hacerlo:
  - Spawn: crea un espacio de memoria limpio para el nuevo proceso de forma independiente. Es la opción por defecto en en Linux y MaxOS y la única disponible en Windows.
  - Fork: Copia el espacio de memoria en ese momento del proceso padre. Es la opción por defecto en Linux y no está disponible en Windows.

https://docs.python.org/3/library/multiprocessing.html#contexts-and-start-methods

```
import time
import multiprocessing as mp
import os

def hijo(num):
    c1 = mp.get_context()
    global valor
    print("Hijo antes de modificar: ", valor)
    valor = num
    print("Hijo: ", valor)
    time.sleep(num)
    print("Salgo del proceso hijo {}".format(os.getpid()))
    os._exit(0)

valor = 0
def main():
    global valor
    valor = -5
    p1 = mp.Process(target=hijo, args=(3,))
    p1.start()
    print ("Proceso creado {}".format(p1.pid))
    #p1.join()
    print("Padre: ", valor)
    p2 = mp.Process(target=hijo, args=(1,))
    p2.start()
    print ("Proceso creado {}".format(p2.pid))
    #p2.join()
    print("Padre: ", valor)

    print("Salgo del proceso padre {}".format(os.getpid()))

if __name__ == '__main__':
  mp.set_start_method('spawn')
  main()
```

3 - Crear un proceso desde un ejecutable del sistema

subprocess:

https://docs.python.org/3/library/subprocess.html

Lanzamiento de procesos con bloqueo: función run

```
subprocess.run()
```

- El proceso padre queda bloqueado hasta que el hijo acaba
- Hay que protegerse con try-except en caso de error

```
import subprocess
import time

def CrearProceso():
  try:
    proc = subprocess.Popen('notepad.exe')
    return proc
  except subprocess.CalledProcessError as e:
    print(e.output)


p = CrearProceso()
time.sleep(5)
```

3 - Crear un proceso desde un ejecutable del sistema

subprocess:

https://docs.python.org/3/library/subprocess.html

Lanzamiento de procesos sin bloqueo: Clase Popen y communicate

```
proc = subprocess.Popen('notepad.exe’)
subprocess.communicate()   # Recoge la salida de un proceso.
```

- communicate bloquea hasta que no acaba el proceso llamado
- En ocasiones interesa protegerse con try-except.

3 - Crear un proceso desde un ejecutable del sistema: Recogiendo salida

- Recoger stdout, stderr → subprocess (Ejemplo)

```
import subprocess
import time

# Vigilo cada 1 segundos si un servicio Web está arriba
# codecs: https://docs.python.org/3/library/codecs.html#standard-encodings

servicio = "www.madrid.org"
error = False

while not error:
    try:
        subproc = subprocess.Popen(["ping", servicio,"-n","1"], stdout=subprocess.PIPE, stderr=subprocess.PIPE)
        (standardout, standarderr) = subproc.communicate()
        salidaStr = standardout.decode("cp858")
        salidaErr = standarderr.decode("cp858")
    except:
        error = True
        salidaStr = "Error en comando"
    print(salidaStr)
```

Popen.**communicate**(*input=None*, *timeout=None*)

Interact with process: Send data to stdin. Read data from stdout and stderr, until end-of-file is reached. Wait for process to terminate and set the `returncode` attribute. The optional *input* argument should be data to be sent to the child process, or `None`, if no data should be sent to the child. If streams were opened in text mode, *input* must be a string. Otherwise, it must be bytes.

`communicate()` returns a tuple `(stdout_data, stderr_data)`. The data will be strings if streams were opened in text mode; otherwise, bytes.

Note that if you want to send data to the process's stdin, you need to create the Popen object with `stdin=PIPE`. Similarly, to get anything other than `None` in the result tuple, you need to give `stdout=PIPE` and/or `stderr=PIPE` too.

https://docs.python.org/3.10/library/subprocess.html

4 – Operaciones con procesos del sistema

Psutil: Información sobre procesos en ejecución y otros parámetros del sistema en funcionamiento.
https://psutil.readthedocs.io/en/latest/

- Ver procesos en ejecución
- Detener procesos
- Temperaturas del hardware
- Duración de la batería (`battery = psutil.sensors_battery()`)

Funciones

```
p.process_iter()
p.is_running()
p.kill() / p.suspend() / p.resume()
p.name()/ p.exe() / p.nice()
p.openfiles()
p.parent()
```

4 – Operaciones con procesos del sistema, ejemplo

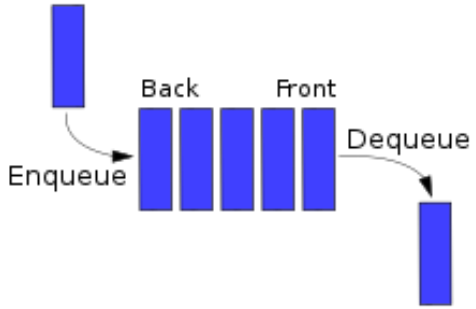
Identificar procesos del sistema y hacer operaciones con ellos.

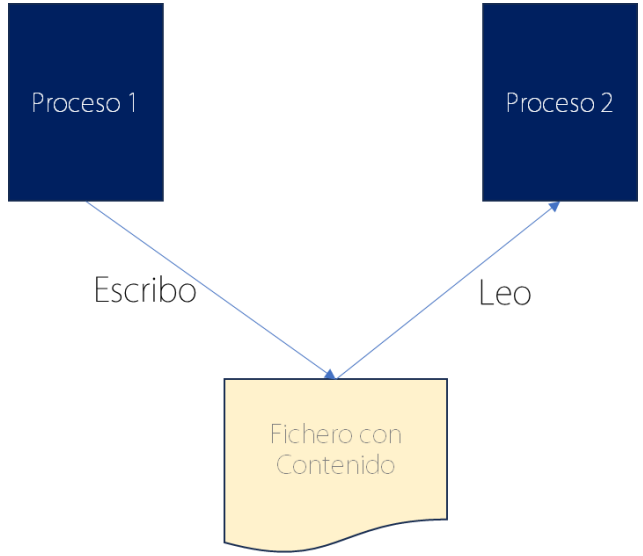
```
import psutil
import time

time.sleep(1)
for p in psutil.process_iter():
          if p.is_running():
                    if p.name() == "Notepad.exe":
                              print("Encontrado Notepad")
                              while p.is_running():
                                        p.suspend()
                                        print("Suspendo")
                                        time.sleep(10)
                                        p.resume()
                                        print("Reactivo")
                                        time.sleep(5)
```

5 – Comunicación entre procesos

Existen diversos mecanismos de comunicación entre procesos

- E/S estándar: stdout/stderr
  - Clase subprocess
- Colas de mensajes
  - Clase Queue de la librería multiprocessing
- Ficheros
  - Visto en ejemplos en clase
- Sockets / websockets
  - Los veremos en temas posteriores





Multiprocessing - Colas

Funciones para el uso de colas entre procesos

```
From multiprocessing import Queue
```

- Creación de la cola

  ```
  miCola = Queue(maxsize= Tamaño máximo )
  ```

- Comprobar si está vacía/llena: devuelve True-False

  ```
  miCola.empty()
  miCola.full()
  ```

- Añadir elemento: añade un elemento a la cola

  ```
  miCola.put(elemento)
  ```

- Obtener elemento: Devuelve el primer elemento de la cola y lo elimina de la misma

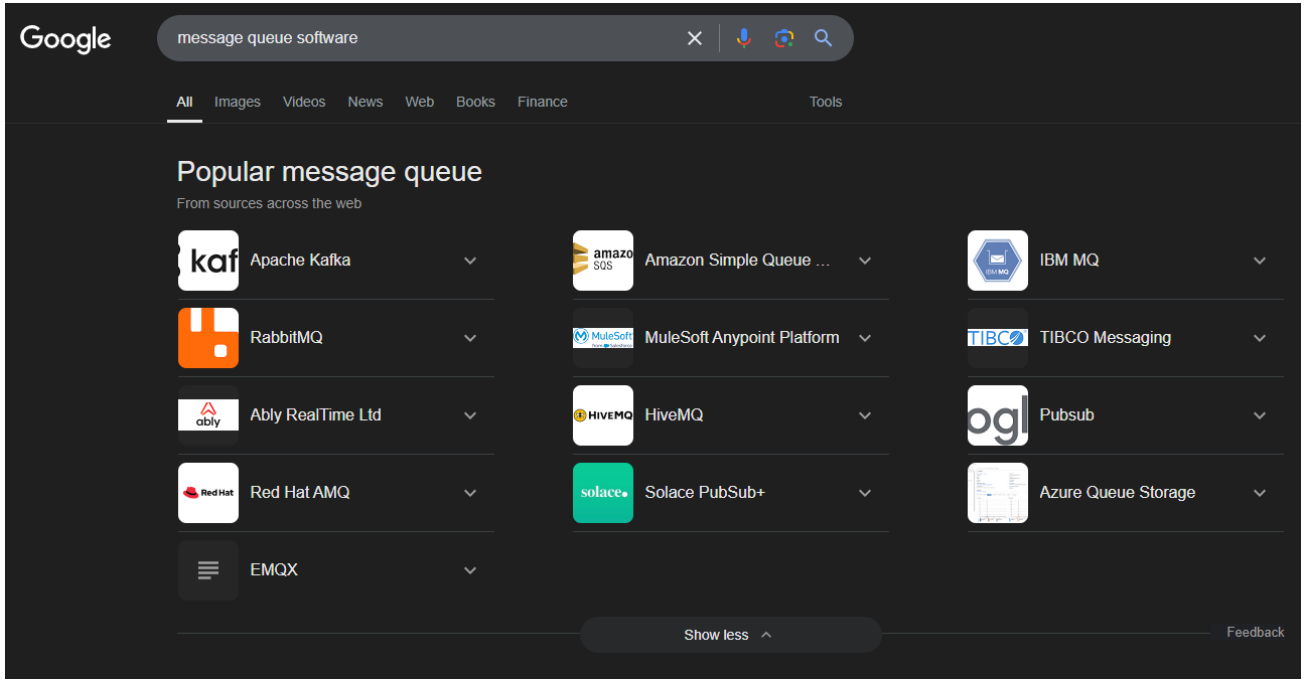
  ```
  miCola.get()
  ```

Ejemplo de uso de una Cola

```
from multiprocessing import Process, Queue
import os
import time

def leo(q):
    end = False
    cadena = ''
    while not end:
        texto = q.get()
        if texto == '--':
            end = True
        else:
            cadena += texto
            print (cadena)
    print("Salgo del proceso hijo {}".format(os.getpid()))
    os._exit(0)
```

```
def main():
    myQ = Queue(maxsize=1000)
    end = False

    rp = Process(target=leo, args=(myQ, ))
    rp.start()
    #rp.join()

    while not end:
        line = input("Introduce mensaje: ")
        if line == '--':
            end = True
        myQ.put(line)
        time.sleep(3)
    print("Salgo del proceso padre {}".format(os.getpid()))

if __name__ == '__main__':
  main()
```

Sistemas comerciales de colas de mensajes



- https://en.wikipedia.org/wiki/Message_queue
- https://icepanel.medium.com/top-6-message-queues-for-distributed-architectures-a3cbabf08993
- https://www.rabbitmq.com/
- https://kafka.apache.org/

Fin UD1 Multiprocesos