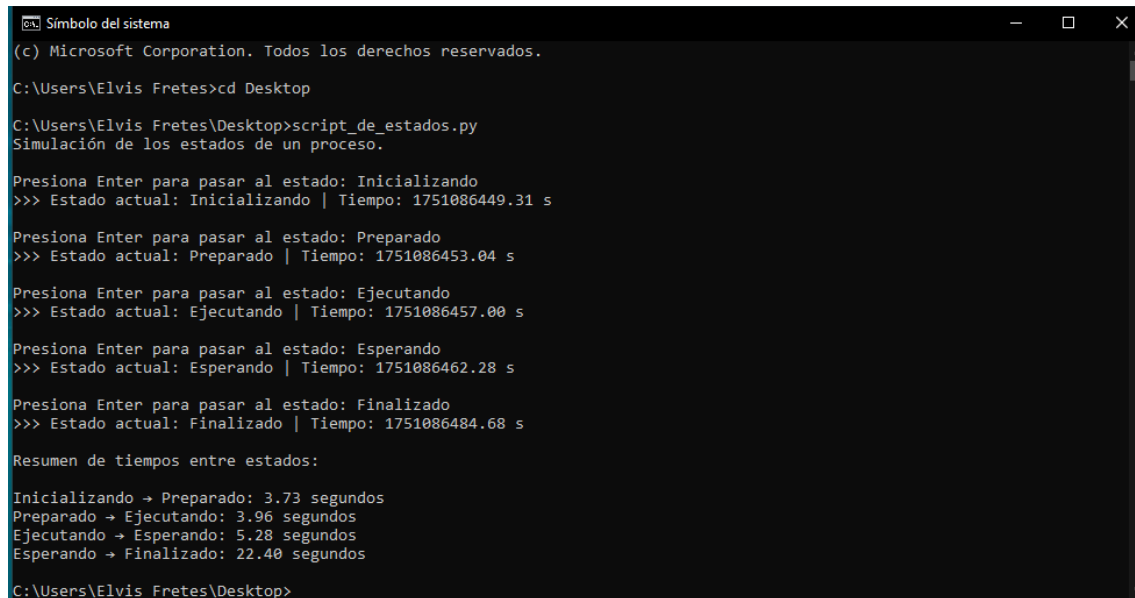


# Laboratorio 1: Gestión de Procesos

## Estados de Procesos

Para este apartado se desarrolló un pequeño script en Python llamado `script_de_estados.py`, cuya función principal fue simular el ciclo de vida de un proceso dentro del sistema operativo. La simulación incluyó los siguientes estados: Inicializando, Preparado, Ejecutando, Esperando y Finalizado.

Cada transición se activaba presionando la tecla Enter, y se medía el tiempo entre cada uno de los estados, tal como se muestra a continuación:



```
Símbolo del sistema
(c) Microsoft Corporation. Todos los derechos reservados.
C:\Users\Elvis Fretes>cd Desktop
C:\Users\Elvis Fretes\Desktop>script_de_estados.py
Simulación de los estados de un proceso.

Presiona Enter para pasar al estado: Inicializando
>>> Estado actual: Inicializando | Tiempo: 1751086449.31 s

Presiona Enter para pasar al estado: Preparado
>>> Estado actual: Preparado | Tiempo: 1751086453.04 s

Presiona Enter para pasar al estado: Ejecutando
>>> Estado actual: Ejecutando | Tiempo: 1751086457.00 s

Presiona Enter para pasar al estado: Esperando
>>> Estado actual: Esperando | Tiempo: 1751086462.28 s

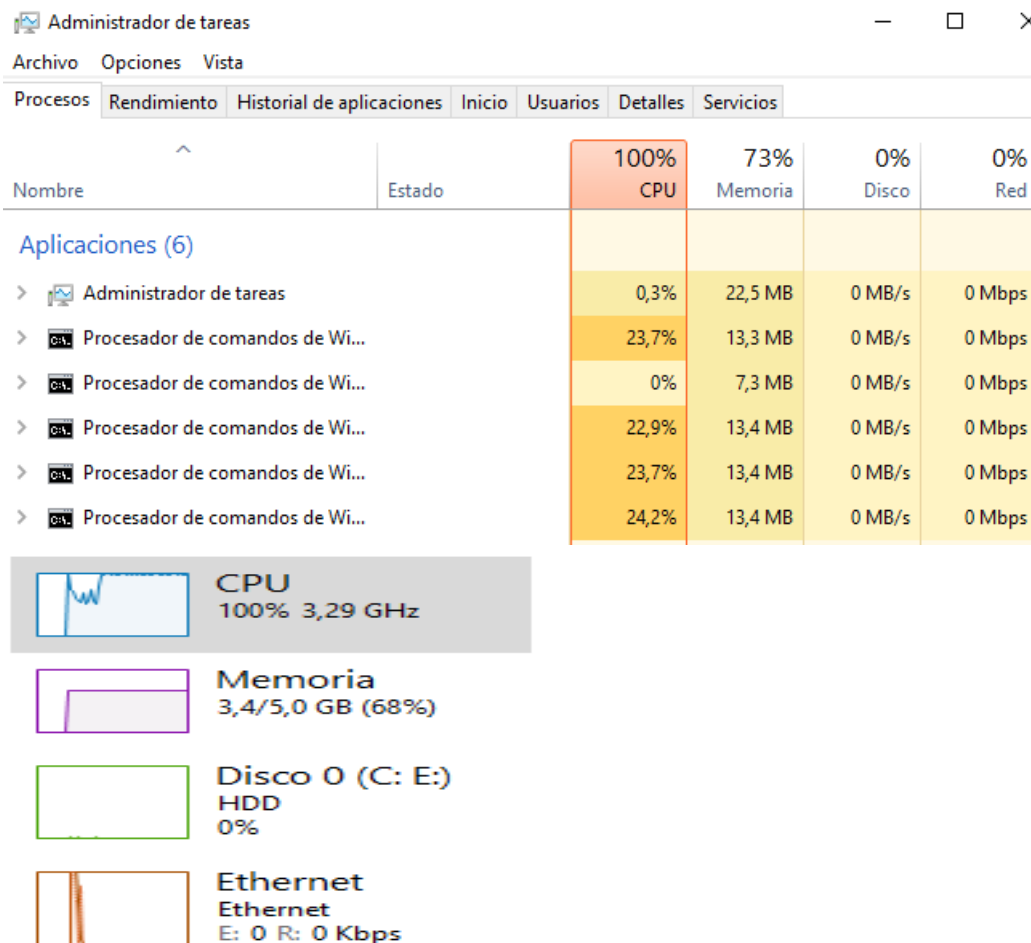
Presiona Enter para pasar al estado: Finalizado
>>> Estado actual: Finalizado | Tiempo: 1751086484.68 s

Resumen de tiempos entre estados:
Inicializando -> Preparado: 3.73 segundos
Preparado -> Ejecutando: 3.96 segundos
Ejecutando -> Esperando: 5.28 segundos
Esperando -> Finalizado: 22.40 segundos
C:\Users\Elvis Fretes\Desktop>
```

## Scheduling del Sistema Operativo

Se realizó una prueba para observar cómo Windows administra los procesos cuando hay múltiples cargas simultáneas en la CPU. Para esto, se ejecutó el script `carga_cpu.py` en **cinco consolas CMD diferentes** de forma paralela.

El resultado fue que el uso total de CPU subió al **100%**, y el administrador de tareas mostró una distribución bastante uniforme del consumo entre los procesos, lo cual refleja un reparto razonable de los recursos:



El comportamiento observado se asemeja al algoritmo de **Round Robin**, ya que el sistema parece repartir tiempo de CPU entre los procesos activos. El **Proceso 2** no alcanzó consumo significativo, probablemente por una condición inicial o error puntual.

### Simulación y Resolución de Deadlock

Para finalizar, se realizó una simulación de *deadlock* utilizando Python y múltiples *locks*. En el primer experimento, los hilos adquirirían los recursos en distinto orden, lo cual provocó que ambos quedaran esperando uno al otro, bloqueándose:

```
Proceso 1 adquirió lock_x
Proceso 2 adquirió lock_y
>> Bloqueo detectado: los hilos quedaron bloqueados.
```

Administrador de tareas			
Archivo Opciones Vista			
Procesos	Rendimiento	Historial de aplicaciones	Inicio Usuarios Detalles Servicios
Nombre	Estado	29% CPU	66% Memoria
Aplicaciones (2)			
> Administrador de tareas		0%	20,5 MB
> Python (32 bits) (3)		0%	11,2 MB

Para resolver este conflicto, se modificó el código para que ambos hilos adquieran los recursos en el **mismo orden**, eliminando así la posibilidad de espera circular. El resultado fue exitoso:

```
C:\Users\Elvis Fretes>cd Desktop
C:\Users\Elvis Fretes\Desktop>script_de_deadlock.py
Hilo 1 adquirió lock_a
Hilo 1 adquirió lock_b
Hilo 1 finalizó correctamente
Hilo 2 adquirió lock_a
Hilo 2 adquirió lock_b
Hilo 2 finalizó correctamente
>> Ambos hilos finalizaron sin deadlock.
C:\Users\Elvis Fretes\Desktop>_
```

### Conclusión

Este laboratorio permitió observar cómo un sistema operativo gestiona los procesos desde su creación hasta su finalización, pasando por diferentes estados controlados. Además, se analizó la forma en que Windows distribuye los recursos entre múltiples procesos activos, mostrando un comportamiento cercano a la planificación Round Robin.

También se pudo experimentar un escenario de *deadlock* y aplicar una solución práctica, entendiendo las condiciones que lo provocan. A pesar de su simpleza, el ejercicio refleja la importancia del orden de acceso a recursos compartidos en programación concurrente.