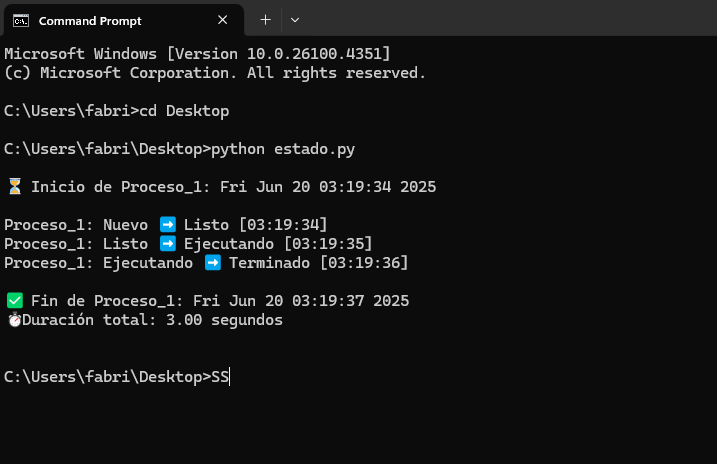
**Laboratorio 1: Gestión de Procesos**

# Estados de Procesos

Para observar los distintos estados de un proceso en Windows, utilicé un programa en Python que simula las transiciones típicas: Nuevo, Listo, Ejecutando, Bloqueado y Terminado.

Al ejecutar el script estados\_proceso.py, se capturaron los siguientes estados y tiempos de transición:



Estas transiciones reflejan los pasos comunes que atraviesa un proceso en el sistema operativo: desde su creación (Nuevo), hasta estar listo para ejecutarse (Listo), pasar a la ejecución efectiva (Ejecutando), detenerse temporalmente mientras espera un recurso (Bloqueado) y finalmente su finalización (Terminado).

# Scheduling del Sistema Operativo

Para estudiar cómo el sistema operativo distribuye el tiempo de CPU, se ejecutaron simultáneamente cinco programas que demandan gran cantidad de recursos de procesamiento.

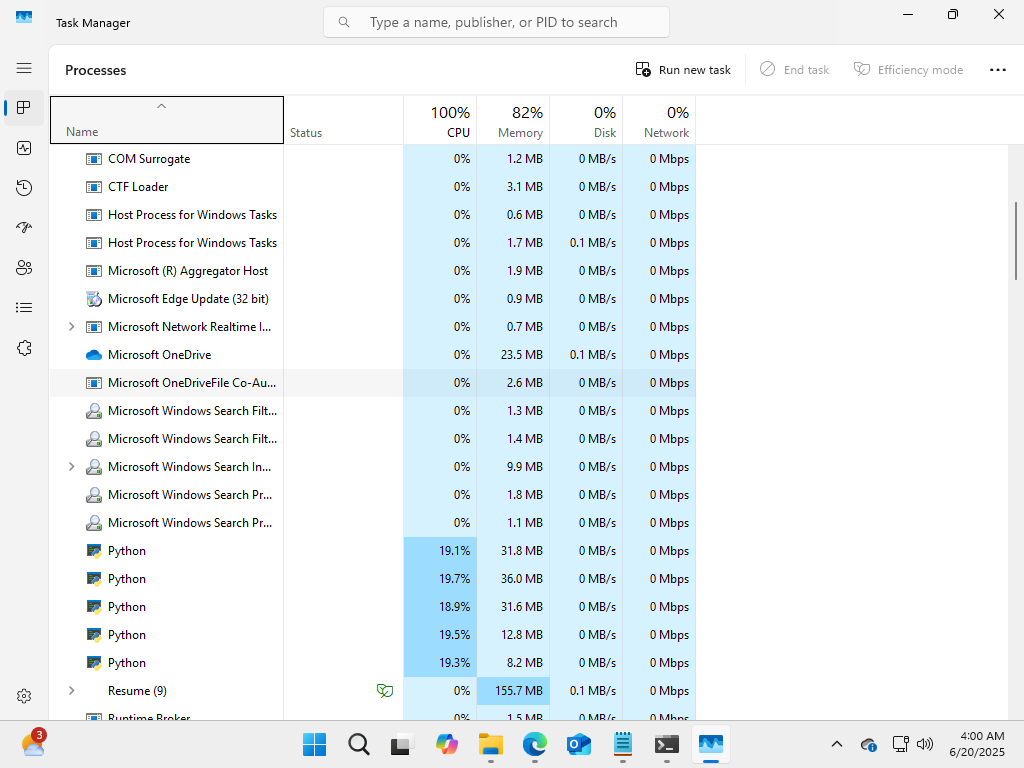
Durante la ejecución, se monitoreó el uso de CPU en el Administrador de tareas de Windows. Se observaron los siguientes porcentajes aproximados de uso de CPU para cada proceso:

* Cada uno de los cinco programas consumía alrededor del 19.3% de la CPU y uno llegó a 18.9%.
* Uno de los procesos llegó a un pico del 19.7%.

Este comportamiento refleja una distribución relativamente equitativa del tiempo de CPU entre los procesos, similar a algoritmos de planificación de tipo Round Robin.

Esto indica que el sistema operativo priorizó los primeros cuatro procesos, dejando al quinto proceso sin recursos para ejecutarse. Este comportamiento se asemeja al algoritmo de planificación FIFO (First In, First Out), donde los primeros procesos en llegar se ejecutan hasta finalizar, mientras que los que llegan después deben esperar. En este caso, el proceso 5 no llegó a ejecutarse, lo cual refleja una falta de equidad en la distribución de recursos.

Si se hubiera utilizado un algoritmo de planificación Round Robin, el comportamiento habría sido distinto. Este tipo de planificación asigna un pequeño intervalo de tiempo a cada proceso de forma rotativa, permitiendo que todos avancen gradualmente. Bajo este esquema, todos los procesos, incluido el proceso 5, habrían tenido oportunidad de ejecutarse, aunque fuera parcialmente.



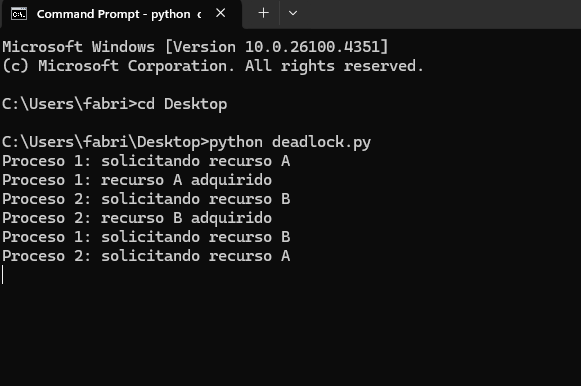
## Creación y Observación de Deadlock

Para demostrar un deadlock, se implementó un programa en Python con dos hilos que intentan acceder a dos recursos (locks) en orden inverso, generando un bloqueo mutuo.

Durante la ejecución, el programa queda bloqueado esperando indefinidamente la liberación de recursos, sin avanzar ni terminar.

En el Administrador de tareas, se puede observar el proceso de Python activo, pero sin un consumo significativo de CPU.

Se implementó un mecanismo en el código para detectar el deadlock mediante un timeout, que reporta la situación en la consola antes de terminar.



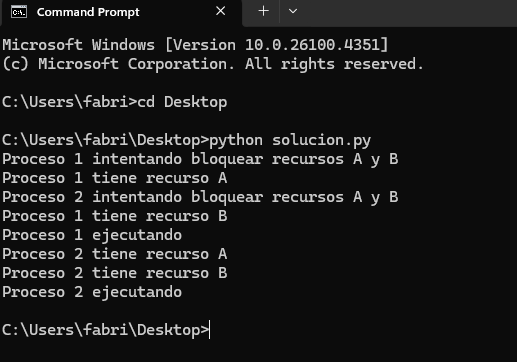
**Solución propuesta:**

Para evitar condiciones de bloqueo mutuo (deadlock), ambos procesos deben seguir un orden de acceso consistente a los recursos compartidos. Específicamente, se debe implementar un protocolo de acceso secuencial donde todos los procesos soliciten primero el recurso A y posteriormente el recurso B. Este enfoque garantiza:

1. **Exclusión mutua ordenada:** Elimina la posibilidad de espera circular

2. **Prevención de deadlock:** Rompe uno de los cuatro requisitos necesarios para que ocurra un deadlock (condición de espera circular)

3. **Determinismo:** Establece un flujo predecible en la adquisición de recursos



## Conclusión

En este laboratorio se logró comprender el ciclo de vida de un proceso dentro del sistema operativo, observando sus diferentes estados y los tiempos de transición entre ellos. Esto permitió visualizar cómo el sistema gestiona y controla la ejecución de procesos en un entorno real.

La ejecución simultánea de varios programas que demandan recursos de CPU evidenció que el sistema operativo distribuye el tiempo de procesamiento de manera equitativa, evitando que un solo proceso monopolice la CPU. Este comportamiento se asemeja a algoritmos de planificación como Round Robin, que buscan maximizar la eficiencia y la justicia en la asignación de recursos. Finalmente, la creación deliberada de un deadlock mostró cómo dos procesos pueden quedar bloqueados indefinidamente al esperar recursos en orden inverso, lo cual detiene la ejecución normal del sistema. La detección y análisis de este estado resaltan la importancia de implementar mecanismos para evitar o resolver deadlocks en sistemas reales, asegurando así la estabilidad y disponibilidad del sistema operativo.