Simulador de Planificación de CPU

Este proyecto es una simulación de algoritmos clásicos de planificación de procesos en un sistema operativo. Permite visualizar cómo se comportan distintos algoritmos al asignar tiempo de CPU a procesos según distintas estrategias.

El proyecto cuenta con una interfaz gráfica desarrollada en Tkinter que permite:

Cargar archivos de recursos y acciones.

Visualizar errores mediante ventanas emergentes.

Simular el uso de recursos por procesos con sincronización usando semáforos o mutexes.

Cómo ejecutar Para iniciar la aplicación, ejecuta el siguiente comando en tu consola:

```
python maintkinter.py
```

Algoritmos Implementados FCFS (First Come First Served) Atiende los procesos en el orden en que llegan.

SJF (Shortest Job First) Selecciona el proceso más corto disponible. Es no-preemptivo.

Round Robin Asigna a cada proceso un quantum fijo y alterna entre ellos de forma cíclica. Ideal para sistemas de tiempo compartido.

Priority Scheduling Ejecuta procesos basados en su prioridad. A menor número, mayor prioridad. Es no-preemptivo.

SRTF (Shortest Remaining Time First) Variante preemptiva de SJF. Siempre ejecuta el proceso con el menor tiempo restante.

Estructura del Código Clase SimpleQueue Implementa una cola simple para Round Robin:

```
class SimpleQueue:
    def enqueue(self, item): ...
    def dequeue(self): ...
    def is_empty(self): ...
Interfaz esperada para los procesos
Cada proceso debe tener los siguientes atributos:

at: tiempo de llegada (arrival time)

bt: tiempo de ráfaga (burst time)

priority: prioridad (opcional, solo en Priority Scheduling)
```

remaining_time: tiempo restante (usado en RR y SRTF)

start_time: momento en que empieza a ejecutarse

completed_time: tiempo en que termina su ejecución

completed: bandera booleana

Funciones de planificación

Cada función toma una lista de procesos y devuelve:

La lista de procesos actualizada (con tiempos de inicio y finalización)

Un timeline o secuencia de ejecución, indicando qué proceso se ejecutó en cada unidad de tiempo (o None si el CPU estuvo inactivo).

```
def fcfs_scheduling(process_list) -> (list, list)
def sjf_scheduling(process_list) -> (list, list)
def round_robin_scheduling(process_list, quantum) -> (list, list)
def priority_scheduling(process_list) -> (list, list)
def srtf(process_list) -> list
```

Requisitos Python 3.x

Uso Debes definir una clase Process con los atributos necesarios. Por ejemplo:

```
class Process:
    def __init__(self, pid, at, bt, priority=0):
        self.pid = pid
        self.at = at
        self.bt = bt
        self.priority = priority
        self.remaining_time = bt
        self.start_time = None
        self.completed_time = None
        self.completed = False
```

Ejemplo de simulación:

```bash
```

```bash
processes = [
    Process("P1", at=0, bt=5, priority=2),
    Process("P2", at=2, bt=3, priority=1),
    Process("P3", at=4, bt=1, priority=3),
]
```

result, timeline = fcfs_scheduling(processes) Visualización El timeline devuelto por los algoritmos puede usarse para crear un diagrama de Gantt o para simular

la ejecución de procesos en una GUI, por ejemplo, usando Tkinter.

Este módulo simula la interacción de procesos con recursos del sistema usando semáforos o mutexes para la sincronización. Está diseñado para integrarse con una interfaz gráfica (GUI) y permite cargar recursos y acciones desde archivos.

Características Define y administra recursos con semáforos o mutexes.

Carga recursos desde archivo (resources.txt).

Carga acciones de procesos desde archivo (accion.txt).

Usa Tkinter para mostrar mensajes de error emergentes.

Valida que los procesos y recursos referenciados en las acciones existan.

Estructura de Archivos resources.txt Cada línea define un recurso con el formato:

<nombre_recurso>,<cantdad>

Ejemplo:

```
R1,3
R2,1
```

Cada línea define una acción de un proceso sobre un recurso:

```
<PID>,<OPERACIÓN>,<NOMBRE_RECURSO>,<CICLO>
Operaciones: READ, WRITE, RELEASE

Ejemplo:

P1,READ,R1,2
P2,WRITE,R2,4
P1,RELEASE,R1,5
```

Cómo Funciona

Clase Resource: Representa un recurso del sistema, que puede usar semáforo o mutex según configuración.

Función load_resources_resource(filepath, use_semaphore): Lee el archivo de recursos y devuelve un diccionario con objetos Resource.

Clase Action: Representa una acción que un proceso realiza sobre un recurso.

Función load_actions(filename, process_list, resource_list): Lee las acciones, valida que el PID y recurso existan, y genera mensajes de error con Tkinter en caso de inconsistencias.

Manejo de Errores Las líneas inválidas en los archivos se omiten mostrando un mensaje por consola.

Si un PID o recurso referenciado en accion.txt no existe, se muestra un error emergente usando Tkinter (messagebox.showerror).

Dependencias Python 3.x

Tkinter (incluido en la instalación estándar de Python)

Módulo threading (incluido en Python estándar)

Ejemplo de Uso

```
resources = load_resources_resource("resources.txt", use_semaphore=True)
actions = load_actions("accion.txt", process_list, list(resources.values()))
```

Puedes extender la clase Action para soportar más operaciones como LOCK, UNLOCK o esperas con tiempo.

Licencia MIT