TECNOLÓGICO NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE TLAXIACO

Simulación de administración de memoria en la CPU

CARRERA:

INGENIERÍA EN SISTEMAS COMPUTACIONALES

ASIGNATURA:

Graficación

SEMESTRE: 5BS

ALUMNO:

Diego Cruz Cruz	22620104
Irvin Jiménez Sánchez	22620075
Marco Uriel Rosas García	22620119

DOCENTE:

OSORIO SALINAS EDWARD

Tlaxiaco Oax.

05 diciembre de 2024

OBJETIVO

El objetivo de esta práctica es implementar mediante código Python un algoritmo que permite simular el funcionamiento de un sistema operativo el cual ejecute programas y su comportamiento en la administración de memoria de la CPU, cada programa tendrá diferente valor de espacio de memoria.

Se utilizará el software Visual Studio Code para la implementación del algoritmo en Python. Se pretende entender el comportamiento de memoria en de la CPU cuando se ejecuta un programa en un sistema operativo.

MATERIAL UTILIZADO

Equipo de cómputo.

Internet.

Software (Visual Studio Code con extensión de Python)

RESULTADOS

1. Se ejecuta el programa en Visual Studio, seguido de eso aparecerá una interfaz en la cual representa la CPU de un equipo de cómputo.

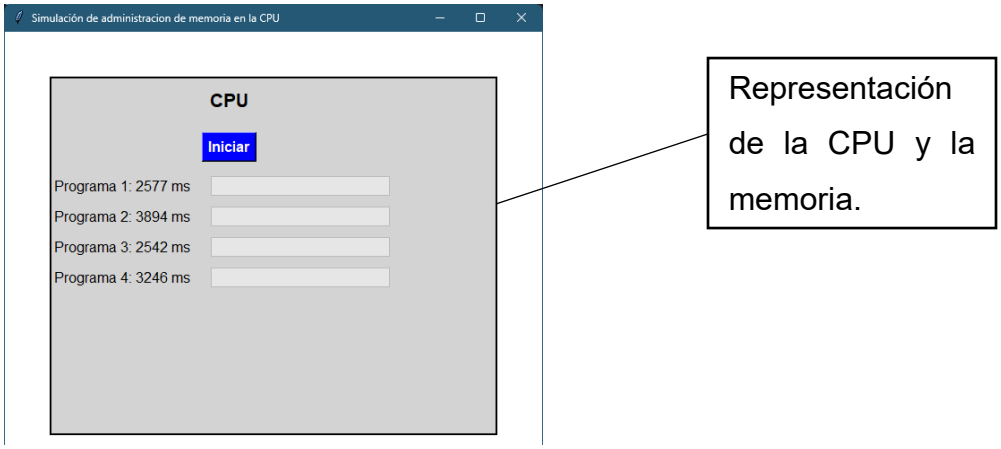


Ilustración 1 Representación

2. Dentro del algoritmo de Python se establecieron 4 programas los cuales simularan que se cargaran en memoria.

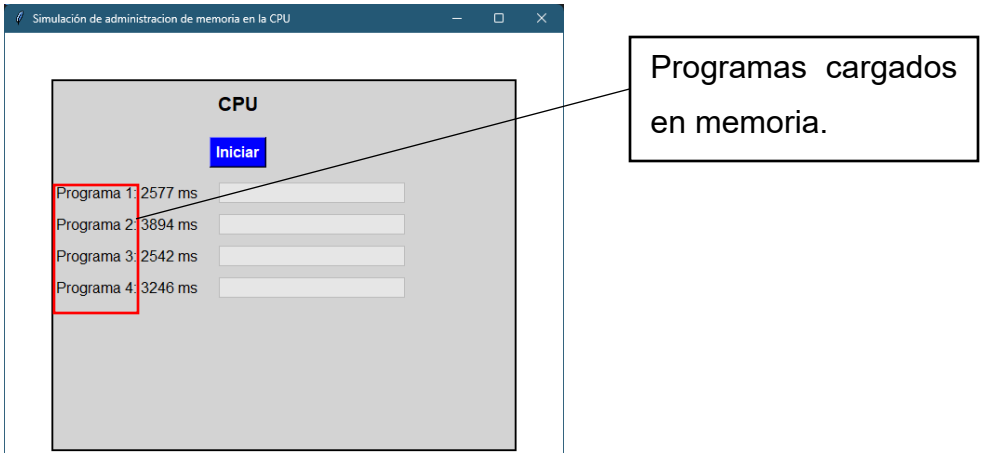


Ilustración 2 Programas cargados.

3. Se estableció dentro del algoritmo de Python que los valores de espacio de memoria de las variables, que en este caso son los programas, **fueran aleatorios**. Dichos valores estarán representados por **milisegundos.**

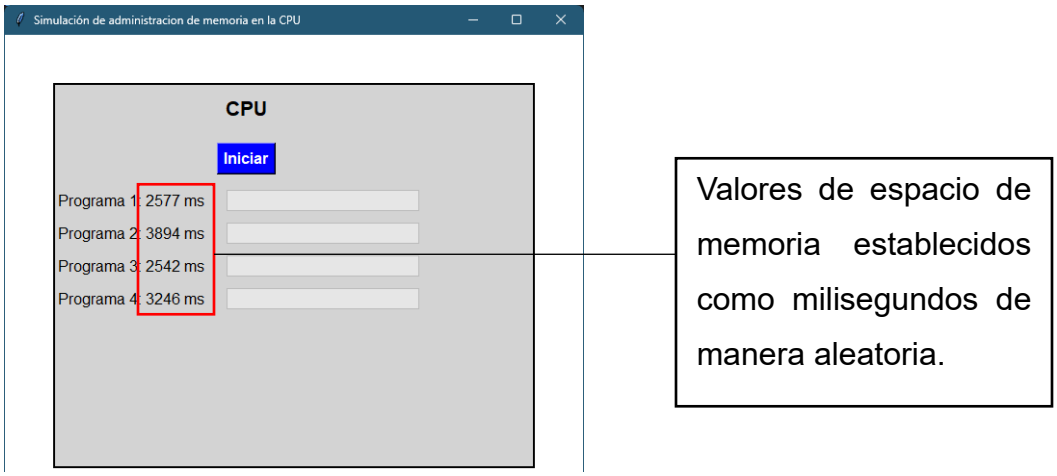


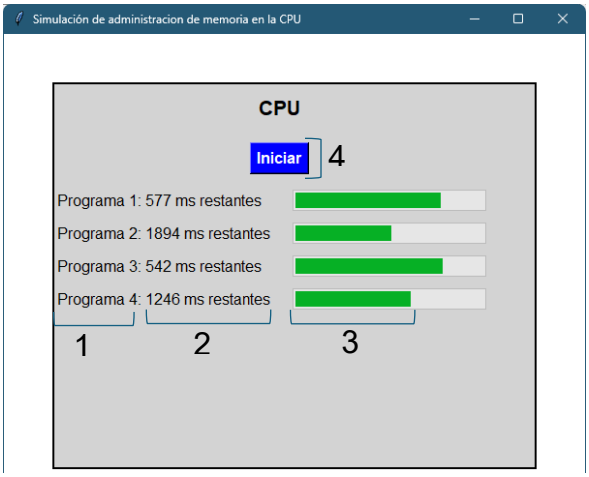
Ilustración 3 Valores de espacio de memoria.

4. El espacio de memoria de la CPU en total es de 16 000 ms, ese espacio se reparte en los 4 programas cargados. Cada barra de carga tiene el espacio de 4 000 ms para cada programa.

   Al ejecutar la simulación muestra que, los valores de espacio de memoria de las instrucciones ocuparan una parte de la memoria asignada.

- El programa 1 su instrucción solo ocupara el espacio de 2577 de los 4 000 asignados.
- El programa 2 su instrucción solo ocupara el espacio de 3894 de los 4 000 asignados.
- El programa 3 su instrucción solo ocupara el espacio de 2542 de los 4 000 asignados.
- El programa 4 su instrucción solo ocupara el espacio de 3246 de los 4 000 asignados.

5. Ejecución del programa:



1.- Cargar el programa en memoria.

2.- Asignar espacio de memoria a las variables.

3.- Asignar espacio de memoria a las instrucciones.

4.- Ejecutar el programa.

Ilustración 4 Ejecución

6. Como la simulación representa el funcionamiento real de una computadora. Al asignar espacio de memoria a las instrucciones, la memoria atenderá a todos los programas que no se han terminado de ejecutar o de cargar esto con el fin del liberar el espacio de la memoria y darles paso a más programas. Se Podría interpretar que realizar otro paso para terminar su proceso.

Ilustración 5 Ejecución terminada.

7. Código implementado en Python.

```
import tkinter as tk
from tkinter import ttk
import random
import time
from threading import Thread

class RoundRobinAnimation(tk.Tk):
    TIME_SLICE = 2000  # Tiempo de procesamiento por segmento en milisegundos

    def __init__(self):
        super().__init__()
        self.title("Simulación de administración de memoria en la CPU")
        self.geometry("600x600")
        self.configure(bg="white")

        self.process_durations = [random.randint(500, 4000) for _ in range(4)]  # Duraciones aleatorias
        self.remaining_times = self.process_durations[:]
        self.process_completed = [False] * 4
        self.current_process_index = 0

        self.create_widgets()
        self.animation_running = False

    def create_widgets(self):
        # Contenedor CPU
        self.cpu_frame = tk.Frame(self, bg="lightgray", bd=2, relief="solid")
        self.cpu_frame.place(x=50, y=50, width=500, height=400)

        # Título para el contenedor CPU
        self.cpu_title = tk.Label(self.cpu_frame, text="CPU", bg="lightgray", font=("Arial", 16, "bold"))
        self.cpu_title.grid(row=0, column=0, columnspan=2, pady=10)

        # Botón CPU
        self.cpu_button = tk.Button(self.cpu_frame, text="Iniciar", bg="blue", fg="white", font=("Arial", 12, "bold"), command=self.start_animation)
        self.cpu_button.grid(row=1, column=0, columnspan=2, pady=10)

        # Panel de procesos dentro del contenedor CPU
        self.process_labels = []
        self.progress_bars = []
        for i, duration in enumerate(self.process_durations):
```

```
            label = tk.Label(self.cpu_frame, text=f"Programa {i+1}: {duration} ms", bg="lightgray", font=("Arial", 12))
            label.grid(row=2 + i, column=0, pady=5, sticky="w")
            self.process_labels.append(label)

            # Modificado para mover las barras a la derecha
            progress = ttk.Progressbar(self.cpu_frame, maximum=100, length=200)
            progress.grid(row=2 + i, column=1, pady=5, padx=20)  # Se añadió el padx para mover las barras a la derecha
            self.progress_bars.append(progress)

    def update_ui(self):
        # Actualizar el estado de los procesos
        for i, remaining in enumerate(self.remaining_times):
            if self.process_completed[i]:
                self.process_labels[i].config(text=f"Programa {i+1}: Completado")
                self.progress_bars[i]["value"] = 100
            else:
                percentage = 100 * (1 - remaining / self.process_durations[i])
                self.process_labels[i].config(text=f"Programa {i+1}: {remaining} ms restantes")
                self.progress_bars[i]["value"] = percentage

    def round_robin_step(self):
        if all(self.process_completed):
            self.animation_running = False
            return

        # Seleccionar el proceso actual
        i = self.current_process_index
        if self.remaining_times[i] > self.TIME_SLICE:
            self.remaining_times[i] -= self.TIME_SLICE  # Reducir el tiempo restante del proceso
        else:
            self.remaining_times[i] = 0
            self.process_completed[i] = True  # Marcar el proceso como completado

        # Avanzar al siguiente proceso de manera cíclica
        self.current_process_index = (self.current_process_index + 1) % len(self.process_durations)

    def animate(self):
        while self.animation_running:
            self.round_robin_step()
            self.update_ui()
```

```
            time.sleep(2)  # Simula la rebanada de tiempo en segundos

    def start_animation(self):
        if not self.animation_running:
            self.remaining_times = self.process_durations[:]  # Copiar las duraciones iniciales
            self.process_completed = [False] * len(self.process_durations)  # Reiniciar los estados de los procesos
            self.current_process_index = 0  # Empezar desde el primer proceso
            self.animation_running = True
            animation_thread = Thread(target=self.animate)
            animation_thread.daemon = True  # Asegura que el hilo termine cuando se cierre la ventana
            animation_thread.start()

if __name__ == "__main__":
        app = RoundRobinAnimation()  # Crear instancia de la simulación
        app.mainloop()  # Ejecutar la ventana de la interfaz gráfica

```

CONCLUSIÓN

En conclusión, este proyecto nos ayudó a comprender el funcionamiento de un sistema operativo, especialmente en lo que respecta a la administración de memoria en la CPU. Utilizando implementación de métodos y algoritmos, logramos representar de manera más intuitiva los procesos clave involucrados en la gestión de memoria. Esto nos ayudó a visualizar de forma práctica cómo se manejan los recursos en un sistema operativo.

ÍNDICE DE IMÁGENES