MANUAL TÉCNICO DEL SIMULADOR DE PROCESOS

AUTOR:

JUAN DAVID BELTRÁN BARACALDO 20221578059

PROYECTO PARA LA ASIGNATURA SISTEMAS OPERACIONALES-(578-303)

DOCENTE:

DARIN JAIRO MOSQUERA PALACIOS

UNIVERSIDAD DISTRITAL FRANCISCO JOSE DE CALDAS

FACULTAD TECNOLÓGICA

TECNOLOGÍA EN SISTEMATIZACIÓN DE DATOS

BOGOTA DC

2024-2025

Contenido

1. Introducción

El **Simulador de Procesos** es una aplicación web diseñada para emular el comportamiento de un sistema operativo en lo que respecta a la creación, asignación y ejecución de procesos, así como la gestión de la memoria tanto en la RAM como en un espacio virtual.
El objetivo del simulador es proporcionar una herramienta didáctica que permita visualizar de forma dinámica cómo se distribuyen los recursos (memoria y recursos específicos) entre procesos y cómo se lleva a cabo la ejecución cíclica de estos procesos. Además, se implementa una lógica de preeminencia que permite que ciertos procesos tengan prioridad sobre los demás.

Este manual se enfoca en explicar cada aspecto del proyecto, desde la estructura del código hasta la interacción entre el backend y el frontend, pasando por el análisis de cada función y variable crítica del sistema.

---

2. Arquitectura General del Proyecto

El proyecto se divide en dos grandes componentes:

Backend (Servidor)

- **Lenguaje:** Python
- **Framework:** Flask
- **Responsabilidades:**
    - Gestión de procesos: creación, asignación de memoria, actualización de estados y ejecución de ciclos.
    - Manejo de la memoria: simulación de la memoria RAM y virtual mediante matrices.
    - Asignación y verificación de recursos requeridos por cada proceso.
    - Generación de respuestas en formato JSON para que el frontend actualice la interfaz de manera dinámica.
- **Archivo Principal:** app.py

Frontend (Interfaz de Usuario)

- **Lenguajes y Tecnologías:** HTML, CSS, JavaScript (con jQuery)
- **Responsabilidades:**
  - Mostrar visualmente la información de memoria (matrices de RAM y virtual).
  - Permitir la interacción del usuario para agregar procesos y ejecutar ciclos.
  - Actualizar las tablas y visualizaciones en tiempo real mediante peticiones AJAX.
- **Archivos Principales:**
  - index.html: Página principal que contiene la estructura de la interfaz.
  - script.js: Lógica de inicialización y actualización de las matrices y tablas.
  - style.css: Definición de estilos para lograr una apariencia limpia y organizada.

La comunicación entre ambos componentes se efectúa mediante solicitudes HTTP (POST) que permiten la actualización sin necesidad de recargar la página.

---

3. Descripción del Backend (app.py)

3.1 Configuración Inicial e Importaciones

El archivo app.py inicia importando las librerías necesarias:

```
from flask import Flask, request, jsonify, render_template
import random
```

- **Flask:** Se utiliza para crear la aplicación web, definir rutas y manejar peticiones.
- **random:** Se emplea para seleccionar colores aleatorios y asignar celdas de memoria de forma no secuencial.

Luego, se inicializa la aplicación:

```
app = Flask(__name__)
```

Esta línea crea la instancia de Flask que gestionará las rutas y la comunicación con el cliente.

3.2 Definición y Funcionamiento de la Clase Proceso

La clase Proceso es el núcleo para representar cada proceso en el simulador. Cada instancia de Proceso contiene los siguientes atributos:

- **nombre:** Identificador único del proceso (cadena de texto).
- **memoria:** Cantidad de memoria que le queda por usar; esta disminuye en cada ciclo de ejecución correspondiente.
- **memoria_inicial:** Valor original de la memoria asignada, usado para cálculos y para liberar memoria al finalizar.
- **recursos:** Lista de recursos requeridos, los cuales se verifican antes de la ejecución de un ciclo.
- **estado:** Estado actual del proceso. Puede ser 'Nuevo', 'Ejecutando', 'Listo', 'Bloqueado' o 'Terminado'.
- **color:** Color único asignado al proceso, facilitando su identificación visual en las matrices.
- **preminencia:** Indica si el proceso tiene prioridad (valor 'con' para procesos con preeminencia).
- **celdas_ram / celdas_virtual:** Listas que almacenan los índices de las celdas asignadas en la RAM y en la memoria virtual.
- **grupo_actual:** Contador que ayuda a identificar qué grupo de celdas (en este caso en particular fue, por ejemplo, de 3 en 3) se está utilizando en el ciclo actual.
- **páginas:** Número de páginas calculado con la fórmula (memoria + 2) // 3, simulando la paginación en sistemas operativos.

La clase se define de la siguiente manera:

```
class Proceso:
    def __init__(self, nombre, memoria, recursos, color, preminencia):
        self.nombre = nombre
        self.memoria = memoria
        self.memoria_inicial = memoria
        self.recursos = recursos
        self.estado = 'Nuevo'
        self.color = color
        self.preminencia = preminencia
```

```
self.celdas_ram = []
self.celdas_virtual = []
self.grupo_actual = 0
self.paginas = (memoria + 2) // 3
```

Cada atributo está pensado para simular un aspecto real del manejo de procesos y su relación con la memoria.

3.3 Variables Globales y Gestión de Memoria

Se definen variables globales que controlan el total de memoria, la parte reservada para el sistema y la memoria que queda para los procesos:

```
procesos = []
memoria_total = 50
memoria_sistema = 5
memoria_disponible = memoria_total - memoria_sistema
historial_ciclos = []
indice_proceso_actual = 0
```

- **procesos:** Lista que almacenará todas las instancias de la clase Proceso.
- **memoria_total:** Representa el total de celdas disponibles en la RAM simulada.
- **memoria_sistema:** Cantidad de memoria reservada para el sistema operativo (o funciones críticas).
- **memoria_disponible:** Calculada restando memoria_sistema del total, se usa para validar la asignación de nuevos procesos.
- **historial_ciclos:** Registro de cada ciclo de ejecución, almacenando el estado de todos los procesos en cada iteración.
- **indice_proceso_actual:** Índice para llevar un recorrido cíclico entre los procesos al ejecutar los ciclos.

Las matrices que representan la memoria se inicializan de la siguiente forma:

```
matriz_ram = [{'color': None, 'proceso': None, 'nombre': None} for _
in range(memoria_total)]
matriz_virtual = [{'color': None, 'proceso': None, 'nombre': None} for
_ in range(100)]
```

- **matriz_ram:** Lista de diccionarios que simula la RAM. Cada elemento representa una celda.

- **matriz_virtual:** Lista de 100 elementos para simular la memoria virtual.

Se reserva la memoria del sistema en las primeras celdas de la RAM:

```
for i in range(memoria_sistema):
    matriz_ram[i] = {'color': '#006400', 'proceso': 'Sistema', 'nombre':
'Sistema'}
```

Este bloque asigna las primeras 5 celdas (en este caso) para el sistema, usando un color fijo (verde oscuro) y la etiqueta "Sistema".

3.4 Funciones Auxiliares

El código incluye varias funciones que facilitan la administración de procesos y la asignación de memoria.

3.4.1 Función generar_color_unico()

Esta función se encarga de asignar un color único para cada proceso, evitando que se repitan o que sean demasiado similares. El proceso es el siguiente:

- Se define una lista base de colores (preestablecidos).
- Se verifica cuáles de estos colores no han sido usados (usando el conjunto colores_usados).
- Si no hay colores disponibles en la lista base, se generan colores aleatorios asegurando que la diferencia entre el color nuevo y los ya usados sea suficientemente alta (más de 150 en la suma de las diferencias RGB).
- El color elegido se agrega al conjunto de colores usados y se retorna.

Código relevante:

```
def generar_color_unico():
    colores_base = [
        '#FF0000', '#00FF00', '#0000FF', '#FF00FF', '#00FFFF',
        '#FFA500', '#800080', '#008000', '#800000', '#000080',
        '#FFD700', '#4B0082', '#FF4500', '#2E8B57', '#DC143C',
        '#9400D3', '#696969', '#FF69B4', '#CD853F', '#006400',
        '#8B008B', '#556B2F', '#8B4513', '#4682B4', '#D2691E'
    ]
```

```
    colores_disponibles = [color for color in colores_base if color not in
colores_usados]

    if not colores_disponibles:
        while True:
            r = random.randint(0, 255)
            g = random.randint(0, 255)
            b = random.randint(0, 255)
            nuevo_color = f'#{r:02x}{g:02x}{b:02x}'
            if all(es_color_diferente(nuevo_color, color_usado) for
color_usado in colores_usados):
                break
    else:
        nuevo_color = random.choice(colores_disponibles)

    colores_usados.add(nuevo_color)
    return nuevo_color
```

3.4.2 Función es_color_diferente(color1, color2)

Esta función compara dos colores en formato hexadecimal y verifica si son suficientemente diferentes.
El criterio de diferencia se basa en la suma de las diferencias absolutas de los componentes RGB, la cual debe superar el valor 150.

```
def es_color_diferente(color1, color2):
    r1 = int(color1[1:3], 16)
    g1 = int(color1[3:5], 16)
    b1 = int(color1[5:7], 16)
    r2 = int(color2[1:3], 16)
    g2 = int(color2[3:5], 16)
    b2 = int(color2[5:7], 16)
    return abs(r1 - r2) + abs(g1 - g2) + abs(b1 - b2) > 150
```

3.4.3 Función encontrar_celdas_disponibles(memoria_requerida, es_virtual=False)

Esta función se encarga de buscar en la matriz (RAM o virtual) las celdas que están libres (donde el valor 'proceso' es None).

- Se usa una lista de índices para aquellas celdas libres.
- Si la cantidad de celdas libres es suficiente para la cantidad requerida, se seleccionan aleatoriamente mediante random.sample.
- Si no hay suficientes celdas, se retorna None para indicar la imposibilidad de asignar memoria al proceso.

```
def encontrar_celdas_disponibles(memoria_requerida, es_virtual=False):
    matriz = matriz_virtual if es_virtual else matriz_ram
    celdas_libres = [i for i in range(len(matriz)) if matriz[i]['proceso'] is None]
    return random.sample(celdas_libres, memoria_requerida) if len(celdas_libres) >= memoria_requerida else None
```

3.4.4 Función obtener_proceso_a_ejecutar()

Esta función define la política de selección de procesos para la ejecución en cada ciclo.

- **Prioridad de Preeminencia:** Se busca si existe algún proceso con preeminencia (valor 'con') que no esté en estado 'Terminado'. Si se encuentra, se retorna ese proceso.
- **Recorrido Cíclico:** Si no hay procesos con preeminencia, se recorre la lista de procesos usando el índice global indice_proceso_actual, garantizando que se elija un proceso cuyo estado no sea 'Terminado'.
- Se realiza un ciclo de intentos para evitar bloqueos en caso de que todos los procesos se hayan terminado.

```
def obtener_proceso_a_ejecutar():
    global indice_proceso_actual

    proceso_preminente = next((p for p in procesos if p.preminencia == 'con' and p.estado != 'Terminado'), None)
    if proceso_preminente:
        return proceso_preminente

    intentos = 0
    while intentos < len(procesos):
        if indice_proceso_actual >= len(procesos):
            indice_proceso_actual = 0
```

```
        proceso = procesos[indice_proceso_actual]

        if proceso.estado != 'Terminado':
            return proceso

        indice_proceso_actual = (indice_proceso_actual + 1) %
    len(procesos)
        intentos += 1

    return None
```

3.4.5 Función actualizar_matrices(proceso, grupo_inicio, grupo_fin)

Esta función es crucial para actualizar la representación gráfica de la memoria en ambas matrices (RAM y virtual) según el estado y la posición del grupo de celdas que el proceso está utilizando en el ciclo actual.

Pasos que realiza la función:

1. **Limpieza de Celdas Asignadas:**
   Recorre toda la matriz de RAM y la virtual, y para cada celda que esté asignada al proceso (comparando el objeto del proceso), se la limpia, asignando valores None.

2. **Asignación de Grupo Activo en RAM:**
   Utilizando los índices grupo_inicio y grupo_fin, se actualizan las celdas correspondientes en la matriz de RAM, asignando:
   - El color del proceso.
   - El objeto proceso (para mantener la referencia).
   - Un nombre que concatena el nombre del proceso y el número de celda (por ejemplo, "Proceso1-2").

3. **Asignación de Celdas en la Memoria Virtual:**
   Se recorre la cantidad total de celdas asignadas al proceso (memoria_inicial) y, excluyendo las que ya están en el grupo activo, se asignan en la matriz virtual, siguiendo el mismo formato.

Código relevante:

```
def actualizar_matrices(proceso, grupo_inicio, grupo_fin):
    for i in range(len(matriz_ram)):
        if matriz_ram[i]['proceso'] == proceso:
            matriz_ram[i] = {'color': None, 'proceso': None, 'nombre':
None}

    for i in range(len(matriz_virtual)):
        if matriz_virtual[i]['proceso'] == proceso:
            matriz_virtual[i] = {'color': None, 'proceso': None, 'nombre':
None}

    for i in range(grupo_inicio, grupo_fin):
        if i < len(proceso.celdas_ram):
            celda_ram = proceso.celdas_ram[i]
            matriz_ram[celda_ram] = {
                'color': proceso.color,
                'proceso': proceso,
                'nombre': f"{proceso.nombre}-{i+1}"
            }

    virtual_index = 0
    for i in range(proceso.memoria_inicial):
        if i < grupo_inicio or i >= grupo_fin:
            if virtual_index < len(proceso.celdas_virtual):
                celda_virtual = proceso.celdas_virtual[virtual_index]
                matriz_virtual[celda_virtual] = {
                    'color': proceso.color,
                    'proceso': proceso,
                    'nombre': f"{proceso.nombre}-{i+1}"
                }
                virtual_index += 1
```

3.5 Gestión de Ciclos y Ejecución de Procesos

Existen dos caminos principales para la ejecución de procesos: uno cuando se agrega un proceso (especialmente si tiene preeminencia) y otro cuando se ejecuta un ciclo mediante la ruta /ciclo.

3.5.1 Función ejecutar_ciclo_completo(proceso)

Esta función se invoca en el caso en que se quiera ejecutar de inmediato un proceso, particularmente cuando tiene preeminencia.
Pasos que realiza:

1. **Verificación de Recursos:**
   Antes de proceder, se comprueba que todos los recursos requeridos por el proceso estén disponibles en el diccionario recursos_disponibles.
2. **Cambio de Estado:**
   Si los recursos están disponibles, se cambia el estado del proceso a 'Ejecutando'.
3. **Cálculo del Grupo Activo:**
   Se define el tamaño del grupo (3 celdas) y se calcula el número total de grupos que se pueden formar con la memoria asignada. Luego se determina cuál es el grupo activo en base al contador grupo_actual.
4. **Actualización de la Matriz de Memoria:**
   Se llama a actualizar_matrices con el rango de celdas correspondiente al grupo activo.
5. **Actualización de Variables Internas:**
   Se incrementa el contador grupo_actual de forma cíclica y se decrementa la memoria restante del proceso.
6. **Verificación de Finalización:**
   Si la memoria restante del proceso es menor o igual a cero, se marca el proceso como 'Terminado' y se liberan las celdas asignadas tanto en la matriz de RAM como en la virtual. Además, se incrementa la memoria disponible global y se elimina el color del proceso del conjunto de colores usados.
7. **Registro en Historial:**
   Se guarda un registro del estado actual de todos los procesos en la lista historial_ciclos.

Código relevante:

```
def ejecutar_ciclo_completo(proceso):
    global memoria_disponible

    if all(recursos_disponibles[r] for r in proceso.recursos):
        proceso.estado = 'Ejecutando'

        grupo_size = 3
        total_grupos = (proceso.memoria_inicial + grupo_size - 1) //
grupo_size
        grupo_inicio = (proceso.grupo_actual % total_grupos) *
grupo_size
        grupo_fin = min(grupo_inicio + grupo_size,
proceso.memoria_inicial)

        actualizar_matrices(proceso, grupo_inicio, grupo_fin)

        proceso.grupo_actual = (proceso.grupo_actual + 1) %
total_grupos
        proceso.memoria -= 1

        if proceso.memoria <= 0:
            proceso.estado = 'Terminado'
            for i in range(len(matriz_ram)):
                if matriz_ram[i]['proceso'] == proceso:
                    matriz_ram[i] = {'color': None, 'proceso': None, 'nombre':
None}
            for i in range(len(matriz_virtual)):
                if matriz_virtual[i]['proceso'] == proceso:
                    matriz_virtual[i] = {'color': None, 'proceso': None,
'nombre': None}
            memoria_disponible += proceso.memoria_inicial
            colores_usados.discard(proceso.color)

        historial_ciclos.append([(
            p.nombre, p.memoria, p.memoria_inicial, p.recursos, p.estado,
p.preminencia, p.paginas
        ) for p in procesos])
    else:
```

```
        proceso.estado = 'Bloqueado'
        historial_ciclos.append([(
            p.nombre, p.memoria, p.memoria_inicial, p.recursos, p.estado,
    p.preminencia, p.paginas
        ) for p in procesos])
```

3.5.2 Ruta /ciclo y su Lógica

Esta ruta permite ejecutar un ciclo de procesamiento cuando el usuario lo solicita desde el frontend.
Pasos importantes:

1. **Selección del Proceso a Ejecutar:**
   Se llama a obtener_proceso_a_ejecutar() para determinar cuál proceso debe ejecutar el ciclo.
   Si no hay ningún proceso válido (por ejemplo, si todos están terminados), se retorna un error.
2. **Verificación de Recursos:**
   Se comprueba que todos los recursos del proceso actual estén disponibles. Si lo están, se actualiza el estado a 'Ejecutando'.
3. **Cálculo y Actualización del Grupo Activo:**
   Al igual que en ejecutar_ciclo_completo, se determina el grupo activo de celdas y se actualiza la matriz de memoria.
4. **Decremento de la Memoria y Verificación de Finalización:**
   Se decrementa la memoria del proceso y, si se agota, se marca como 'Terminado' y se liberan las celdas.
5. **Actualización del Índice de Proceso Actual:**
   Si el proceso no tiene preeminencia o ya terminó, se actualiza el índice para la siguiente iteración.
6. **Registro en el Historial y Respuesta:**
   Se añade el estado del ciclo al historial y se retorna un JSON con la información actualizada de la memoria, matrices, procesos y el historial.

Código relevante:

```
@app.route('/ciclo', methods=['POST'])
def ciclo():
    global procesos, indice_proceso_actual, memoria_disponible

    if not procesos:
        return jsonify({'error': 'No hay procesos para ejecutar'})

    proceso_actual = obtener_proceso_a_ejecutar()
    if not proceso_actual:
        return jsonify({'error': 'No hay procesos válidos para ejecutar'})

    proceso_preminente = next((p for p in procesos if p.preminencia ==
'con' and p.estado != 'Terminado'), None)
    if not proceso_preminente:
        for p in procesos:
            if p.estado != 'Terminado':
                p.estado = 'Listo'

    if all(recursos_disponibles[r] for r in proceso_actual.recursos):
        proceso_actual.estado = 'Ejecutando'

        grupo_size = 3
        total_grupos = (proceso_actual.memoria_inicial + grupo_size -
1) // grupo_size
        grupo_inicio = (proceso_actual.grupo_actual % total_grupos) *
grupo_size
        grupo_fin = min(grupo_inicio + grupo_size,
proceso_actual.memoria_inicial)

        actualizar_matrices(proceso_actual, grupo_inicio, grupo_fin)

        proceso_actual.grupo_actual = (proceso_actual.grupo_actual +
1) % total_grupos
        proceso_actual.memoria -= 1

        if proceso_actual.memoria <= 0:
            proceso_actual.estado = 'Terminado'
            memoria_disponible += proceso_actual.memoria_inicial
```

```
            for i in range(len(matriz_ram)):
                if matriz_ram[i]['proceso'] == proceso_actual:
                    matriz_ram[i] = {'color': None, 'proceso': None, 'nombre':
None}
            for i in range(len(matriz_virtual)):
                if matriz_virtual[i]['proceso'] == proceso_actual:
                    matriz_virtual[i] = {'color': None, 'proceso': None,
'nombre': None}
            colores_usados.discard(proceso_actual.color)
        else:
            proceso_actual.estado = 'Bloqueado'

        if not proceso_actual.preminencia or proceso_actual.estado ==
'Terminado':
            indice_proceso_actual = (indice_proceso_actual + 1) %
len(procesos)

        historial_ciclos.append([(
            p.nombre, p.memoria, p.memoria_inicial, p.recursos, p.estado,
p.preminencia, p.paginas
        ) for p in procesos])

        return jsonify({
            'historial': historial_ciclos,
            'memoria_total': memoria_total,
            'memoria_sistema': memoria_sistema,
            'memoria_disponible': memoria_disponible,
            'memoria_usada': memoria_total - memoria_sistema -
memoria_disponible,
            'matriz_ram': [{'color': c['color'], 'nombre': c['nombre']} for c in
matriz_ram],
            'matriz_virtual': [{'color': c['color'], 'nombre': c['nombre']} for c
in matriz_virtual],
            'procesos_actuales': [(
                p.nombre, p.memoria, p.memoria_inicial, p.recursos, p.estado,
p.preminencia, p.paginas
            ) for p in procesos]
        })
```

3.6 Rutas y Endpoints de la Aplicación

El backend define tres rutas principales:

1. Ruta raíz (/):
   Renderiza la plantilla index.html, iniciando la interfaz del simulador.
2. @app.route('/')
3. def index():
4. return render_template('index.html')
5. Ruta /agregar_proceso:
   Método POST que recibe los datos del formulario para agregar un nuevo proceso.
   - Se extraen los datos (nombre, memoria, recursos y preeminencia).
   - Se realizan validaciones: disponibilidad de memoria, existencia de nombre duplicado y espacio en la RAM.
   - Se buscan celdas libres en la RAM y en la memoria virtual.
   - Se asigna un color único al proceso.
   - Se crea la instancia de Proceso y se actualizan las matrices mediante actualizar_matrices.
   - Se actualiza la variable global memoria_disponible y se ejecuta el proceso de inmediato si tiene preeminencia.
   - Se retorna un JSON con el estado actualizado de la memoria, la lista de procesos y el historial de ciclos.
6. **Ruta /ciclo:**
   Método POST que ejecuta un ciclo de procesamiento.
   - Se selecciona el proceso a ejecutar.
   - Se actualiza el estado del proceso según la disponibilidad de recursos.
   - Se actualizan las matrices y se decrementa la memoria asignada.
   - Se actualiza el índice del proceso actual.
   - Se retorna un JSON con la información actualizada.

Finalmente, se lanza el servidor en modo debug:

```
if __name__ == '__main__':
    app.run(debug=True)
```

4. Descripción del Frontend

El frontend está compuesto por tres archivos principales: index.html, script.js y style.css. Cada uno cumple funciones específicas para la visualización y la interacción del usuario.

4.1 Estructura de index.html

Este archivo define la estructura HTML de la interfaz. Se destacan los siguientes elementos:

- **Metadatos y Enlaces:**
  Se especifica la codificación UTF-8, se define el viewport y se enlaza la hoja de estilos style.css.
  Además, se carga jQuery desde una CDN para facilitar la manipulación del DOM.
- **Título y Encabezado:**
  Un <h1> principal indica que se trata del "Simulador de Procesos".
- **Navegación por Pestañas:**
  Se definen tres pestañas:
  - **Principal:** Contiene información sobre la memoria disponible y el formulario para agregar procesos.
  - **Memoria:** Muestra la representación gráfica de la memoria (dos matrices: RAM y virtual).
  - **Detalle:** Presenta la tabla de procesos y el historial de ciclos, además de un botón para ejecutar un ciclo.
- **Formulario para Agregar Proceso:**
  Se incluyen campos para:
  - Nombre del proceso (campo de texto).
  - Memoria requerida (campo numérico).
  - Selección de recursos mediante checkboxes.
  - Opción para marcar el proceso como preeminente. Al presionar el botón "Agregar Proceso", se envían los datos al servidor.
- **Visualización de Matrices de Memoria:**
  Se crean dos contenedores para mostrar la matriz de RAM (5 filas x 10 columnas) y la matriz virtual (10x10), cada uno con un que será llenado dinámicamente.
- **Tablas de Estado e Historial:**
  Dos tablas permiten ver en detalle:

    - El estado actual de cada proceso (nombre, memoria inicial, memoria pendiente, recursos, estado, preeminencia y páginas).
    - El historial de ciclos, donde cada ciclo muestra una lista detallada del estado de cada proceso.

- **Código JavaScript Inline:**
  Dentro del HTML se incluye código jQuery que:
    - Inicializa las matrices al cargar la página.
    - Gestiona la navegación entre pestañas.
    - Realiza peticiones AJAX para agregar procesos y ejecutar ciclos.
    - Actualiza dinámicamente el contenido de las matrices y tablas según la respuesta del servidor.

4.2 Lógica JavaScript y Comunicación AJAX

La interacción principal se basa en solicitudes AJAX que permiten:

- **Agregar Proceso:**
  Al presionar el botón, se captura la información del formulario, se validan los datos y se realiza una petición POST a la ruta /agregar_proceso. La respuesta se usa para actualizar:
    - La memoria disponible (en ambos lugares de la interfaz).
    - Las matrices de memoria (actualizando cada celda con el color y el nombre asignado).
    - La tabla de procesos y el historial.
- **Ejecutar Ciclo:**
  Cuando se presiona el botón "Ejecutar Ciclo", se envía una petición POST a la ruta /ciclo. La respuesta actualiza:
    - Las matrices de RAM y virtual.
    - La tabla de procesos.
    - El historial de ciclos.
    - Los contadores de memoria disponible y memoria usada.

4.3 Archivo script.js y su Funcionalidad

Aunque parte de la lógica JavaScript se encuentra en index.html, el archivo script.js centraliza varias funciones:

- **Inicialización de las Matrices:**
  Se crean las matrices de memoria para RAM y virtual, asignando a cada celda un índice único.
  - La RAM se estructura en 5 filas y 10 columnas, y se reserva de forma fija las primeras celdas para el sistema.
  - La memoria virtual se organiza en una matriz de 10x10.
- **Actualización de Tablas:**
  Funciones que reciben datos en formato JSON del servidor y actualizan el DOM para reflejar el estado actual de los procesos y el historial de ciclos.
- **Manejo de Eventos:**
  Se gestionan los clics en los botones de "Agregar Proceso" y "Ejecutar Ciclo", y se actualizan las pestañas de navegación.

4.4 Diseño y Estilos con style.css

La hoja de estilos style.css define el aspecto visual de la aplicación, enfatizando:

- **Estilos Generales:**
  Se define la fuente (Arial, sans-serif), márgenes y un fondo claro para toda la aplicación.
- **Pestañas de Navegación:**
  Se dan estilos diferenciados a las pestañas activas y a las inactivas, utilizando colores y bordes redondeados para una apariencia moderna.
- **Matrices de Memoria:**
  Se especifican los tamaños, márgenes, bordes y alineación de las celdas que representan la RAM y la memoria virtual.
  - Cada celda tiene un tamaño fijo, y se definen estilos para mostrar el texto (nombre del proceso) de forma legible.
- **Tablas:**
  Se aplican estilos a las tablas de estado e historial para que sean fácilmente legibles, con bordes y fondos diferenciados en el encabezado.

5. Flujo Completo de Ejecución del Simulador

A continuación, se describe paso a paso el flujo de ejecución del simulador:

1. **Carga Inicial:**
    - Al acceder a la URL principal, se ejecuta la función de renderizado de index.html.
    - Se inicializan las matrices de memoria, reservando las primeras celdas para el sistema.
2. **Interacción con el Usuario:**
    - El usuario ve la memoria disponible, las matrices y el formulario para agregar un proceso.
    - Al llenar el formulario y presionar "Agregar Proceso", se envía la información al backend.
3. **Validación y Creación del Proceso:**
    - El servidor valida que exista memoria suficiente y que el nombre del proceso sea único.
    - Se asignan celdas libres en la RAM (y en la memoria virtual si es posible).
    - Se genera un color único para el proceso.
    - Se instancia la clase Proceso y se actualizan las matrices con actualizar_matrices.
    - Si el proceso tiene preeminencia, se ejecuta de inmediato; de lo contrario, se queda en estado "Nuevo" o "Listo".
    - Se actualiza la variable global memoria_disponible.
4. **Actualización de la Interfaz:**
    - La respuesta en formato JSON incluye la nueva distribución de memoria, el estado de los procesos y el historial actualizado.
    - El frontend actualiza las matrices de RAM y virtual, la tabla de procesos y el historial de ciclos.
5. **Ejecución de Ciclos:**
    - Al presionar "Ejecutar Ciclo", se selecciona el proceso a ejecutar según la política (preeminencia o recorrido cíclico).
    - Se comprueba la disponibilidad de recursos. Si están disponibles, se ejecuta el ciclo:
        - Se actualiza el grupo de celdas activo y se disminuye la memoria asignada al proceso.
        - Se actualizan las matrices para reflejar los cambios.
        - Si la memoria del proceso llega a cero, se marca como "Terminado" y se liberan las celdas.

- Se actualiza el índice del proceso actual para la siguiente iteración.
- Se registra el estado del ciclo en el historial.
- El frontend recibe la respuesta y actualiza nuevamente las matrices, tablas y contadores.

6. **Historial y Seguimiento:**
   - Cada ciclo se registra en historial_ciclos, lo que permite al usuario ver la evolución de cada proceso y de la memoria a lo largo del tiempo.
   - Este historial se muestra en una tabla en la pestaña "Detalle".

---

6. Consideraciones Técnicas y Aspectos Relevantes

A lo largo del desarrollo del simulador se tuvieron en cuenta varios aspectos técnicos:

- **Asignación de Memoria y “Paginación”:**
  La división de la memoria en grupos (con tamaño de 3 celdas) simula de forma sencilla la paginación que se usa en sistemas operativos.
  La variable *paginas* en cada proceso facilita entender cómo se distribuye la memoria.
- **Manejo de Recursos:**
  Cada proceso requiere de ciertos recursos (por ejemplo, "Recurso 1", "Recurso 2" y "Recurso 3").
  Antes de ejecutar un ciclo, se verifica la disponibilidad de estos recursos, lo que puede llevar a que el proceso se bloquee si alguno de ellos no está disponible.
- **Preeminencia:**
  La implementación de la preeminencia permite que un proceso pueda interrumpir la secuencia normal de ejecución, obligando a que el proceso en curso se ponga en estado "Listo" y permitiendo que el proceso preeminente se ejecute de inmediato.
- **Actualización en Tiempo Real:**
  La comunicación AJAX entre el frontend y el backend permite que la interfaz se actualice sin necesidad de recargar la página, ofreciendo una experiencia interactiva en tiempo real.

- **Registro Detallado del Historial:**
  Cada ciclo se guarda en un historial que contiene el estado de todos los procesos, lo que facilita el análisis del comportamiento del simulador a lo largo del tiempo.

---

7. Posibles Extensiones y Mejoras Futuras

El diseño modular del simulador permite incorporar futuras mejoras, tales como:

- **Implementación de Algoritmos de Planificación Más Complejos:**
  Por ejemplo, algoritmos de planificación por prioridad, round-robin, o planificación basada en tiempo real.
- **Visualización Mejorada:**
  Se podría mejorar la interfaz gráfica utilizando frameworks modernos (como React o Vue) para una experiencia de usuario más fluida y dinámica.
- **Simulación de Fallas y Recuperación:**
  Incluir escenarios donde se simulen errores en la asignación de recursos o fallas en la ejecución de procesos, mostrando estrategias de recuperación.
- **Extensión en la Gestión de Memoria:**
  Implementar técnicas de compactación de memoria o algoritmos de reemplazo de páginas para la memoria virtual.
- **Persistencia de Datos:**
  Guardar el historial de ciclos y el estado de los procesos en una base de datos para análisis posterior.
- **Correcta implementación de hilos y paginación:**
  Debido a faltas de recursos mayormente temporales, no fue posible la implementación adecuada de los hilos, los “multiprocesadores” y la paginación.

---

8. Conclusión

El **Simulador de Procesos** es una aplicación didáctica que logra simular de forma visual y operativa la asignación y ejecución de procesos en un entorno controlado, utilizando conceptos fundamentales de gestión de memoria y

planificación de procesos.
Este manual técnico ha detallado minuciosamente la estructura del código, la función de cada módulo y la interacción entre el backend y el frontend. Se han explicado los mecanismos de asignación de celdas, la generación de colores únicos, la verificación de recursos y la ejecución cíclica, ofreciendo un panorama completo del funcionamiento interno del simulador.

Con este nivel de detalle, se espera que cualquier desarrollador, docente o estudiante pueda:

- Comprender la lógica del proyecto.
- Realizar modificaciones o extensiones de manera informada.
- Utilizar el simulador como base para aprender conceptos de sistemas operativos y gestión de memoria.

Fin del Manual Técnico del Simulador de Procesos.

Profesor, muchas gracias por tomarse el tiempo de evaluar este proyecto, le pido disculpas por no cumplir con las expectativas ni las solicitudes que se hicieron a lo largo del semestre, fue un proceso muy educativo, aunque por falta de tiempo no fue posible lograr el objetivo que usted esperaba, sin embargo, le doy las gracias por este semestre, y espero volverlo a ver en otra aula de clase, juan David Beltrán Baracaldo.