Simulador de Procesos de Procesador (Documentacion)

April 14, 2025

1 Simulador de Procesos de Procesador

1.1 Descripción General del Programa

El programa tiene como propósito simular la ejecución de procesos en un procesador utilizando hilos y una interfaz gráfica basada en tkinter. Permite visualizar el progreso de los procesos en tiempo real, gestionar estados y analizar tiempos estimados y reales de ejecución.

Esta simulación implementa un modelo simplificado pero funcional de un sistema operativo multitarea, donde varios procesos compiten por recursos del procesador. El sistema utiliza una arquitectura basada en eventos para representar la transición de los procesos a través de diferentes estados, desde su creación hasta su terminación.

La simulación se fundamenta en los conceptos teóricos de gestión de procesos en sistemas operativos modernos, implementando: - Ciclo de vida completo de procesos - Asignación de recursos (núcleos, hilos, memoria) - Priorización de procesos - Concurrencia mediante hilos de ejecución - Visualización en tiempo real del rendimiento y estados - Capacidad de interacción mediante pausas y reanudaciones

El enfoque del programa es tanto educativo como demostrativo, permitiendo observar de manera visual conceptos abstractos de sistemas operativos como la planificación de procesos, estados y transiciones.

1.2 Estructura de Clases

El programa está construido sobre un diseño orientado a objetos que separa claramente las responsabilidades entre la representación de los procesos y la gestión de la simulación.

1.2.1 1. Clase `Proceso`

Representa una unidad fundamental de ejecución en el sistema, modelando un proceso individual del sistema operativo.

Atributos principales:

- `id` (int): Identificador único que distingue cada proceso en el sistema
- `prioridad` (int): Valor numérico que determina la precedencia del proceso (menor valor = mayor prioridad)
- `tiempo_ejecucion` (float): Duración total estimada del proceso en segundos
- `ciclo_ejecucion` (int): Número que representa la frecuencia de ejecución del proceso

- `estados` (dict): Diccionario que mapea los cinco estados posibles (`Nuevo`, `Listo`, `Ejecutando`, `Bloqueado`, `Terminado`) a valores booleanos
- `nucleo` (int): Número del núcleo asignado al proceso (1-4)
- `hilo` (int): Número del hilo asignado dentro del núcleo (1-8)
- `memoria_asignada` (int): Cantidad de memoria en MB asignada al proceso (100-1000)
- `ciclos_realizados` (int): Contador de ciclos de procesamiento completados

Métodos detallados:

[ ]:
```python
def __init__(self, id, prioridad, tiempo_ejecucion, ciclo_ejecucion):
    """Inicializa un proceso con parámetros específicos y estados por defecto"""
    # Inicialización de atributos básicos
    self.id = id
    self.prioridad = prioridad
    self.tiempo_ejecucion = tiempo_ejecucion
    self.ciclo_ejecucion = ciclo_ejecucion

    # Inicialización del diccionario de estados (todos falsos inicialmente)
    self.estados = {
        'Nuevo': False,
        'Listo': False,
        'Ejecutando': False,
        'Bloqueado': False,
        'Terminado': False
    }

    # Inicialización de recursos y contadores
    self.nucleo = None
    self.hilo = None
    self.memoria_asignada = 0
    self.ciclos_realizados = 0
```

[ ]:
```python
def simular_proceso(self):
    """
    Simula la transición secuencial del proceso por todos sus estados posibles.
    Este método no se utiliza directamente en la implementación actual,
    pero muestra el concepto teórico del ciclo de vida completo.
    """
    # Secuencia de estados con retardos simulados entre transiciones
    self.estados['Nuevo'] = True
    time.sleep(random.uniform(0.5, 2))

    self.estados['Nuevo'] = False
    self.estados['Listo'] = True
    time.sleep(random.uniform(0.5, 2))

    self.estados['Listo'] = False
    self.estados['Ejecutando'] = True
```

```
    time.sleep(self.tiempo_ejecucion)

    self.estados['Ejecutando'] = False
    self.estados['Bloqueado'] = True
    time.sleep(random.uniform(0.5, 2))

    self.estados['Bloqueado'] = False
    self.estados['Terminado'] = True
```

1.2.2 2. Clase `SimuladorProcesador`

Esta clase es el núcleo de la aplicación y gestiona la simulación completa, incluyendo la interfaz gráfica, la concurrencia y la visualización.

Atributos fundamentales:

- `procesos` (list): Colección de objetos de tipo `Proceso` que participan en la simulación
- `lock_ejecucion` (threading.Lock): Mecanismo de sincronización para evitar condiciones de carrera
- `pausa_event` (threading.Event): Mecanismo para controlar la pausa/reanudación de hilos
- `root` (tk.Tk): Ventana principal de la interfaz gráfica
- `labels_estados` (list): Lista de etiquetas tkinter que muestran los estados de los procesos
- `progress_bars` (list): Lista de barras de progreso que visualizan el avance
- `info_labels` (list): Lista de etiquetas con información detallada de cada proceso
- `fig`, `ax`, `canvas`: Elementos para la generación y visualización de gráficas
- `procesos_terminados` (int): Contador de procesos que han completado su ejecución
- `treeview` (ttk.Treeview): Tabla para mostrar resultados finales

Métodos críticos analizados:

1. **Método `crear_procesos`**:

[ ]:
```
def crear_procesos(self, num_procesos):
    """
    Genera procesos con características aleatorias y asigna prioridades únicas

    La asignación de prioridades se realiza mediante un algoritmo de␣
 ↪aleatorización
    que garantiza que cada proceso tenga una prioridad distinta, lo que permite
    modelar un sistema de planificación basado en prioridades no preemptivo.
    """
    # Generar lista de prioridades únicas y mezclarlas aleatoriamente
    prioridades = list(range(1, num_procesos + 1))
    random.shuffle(prioridades)

    # Crear cada proceso con atributos aleatorios
    for i in range(num_procesos):
        proceso = Proceso(
```

```
            id=i + 1,
            prioridad=prioridades[i],  # Prioridad única del 1 al num_procesos
            tiempo_ejecucion=random.uniform(1, 5),  # Tiempo aleatorio entre␣
↪1-5 segundos
            ciclo_ejecucion=random.randint(1, 10)   # Ciclo aleatorio entre 1-10
        )
        # Asignación de recursos de hardware simulados
        proceso.nucleo = random.randint(1, 4)
        proceso.hilo = random.randint(1, 8)
        proceso.memoria_asignada = random.randint(100, 1000)

        # Agregar el proceso a la colección
        self.procesos.append(proceso)
```

2. **Método `simular_proceso_con_actualizacion`**: Este método es el corazón de la simulación del ciclo de vida de cada proceso:

[ ]:
```
def simular_proceso_con_actualizacion(self, proceso, index):
    """
    Ejecuta la simulación de un proceso individual con visualización en tiempo␣
↪real

    Implementa un modelo de ciclos alternados entre estados Ejecutando y␣
↪Bloqueado,
    simulando la ejecución realista de un proceso en un sistema operativo␣
↪moderno.
    Cada ciclo consume tiempo proporcional a la prioridad del proceso.
    """
    # Inicializar tiempo real si no existe
    if not hasattr(proceso, 'tiempo_ejecucion_real'):
        proceso.tiempo_ejecucion_real = 0

    # Cálculo de ciclos y tiempo por ciclo
    tiempo_estimado_total = proceso.tiempo_ejecucion
    num_ciclos = random.randint(20, 30)  # Entre 20-30 ciclos por proceso
    tiempo_por_ciclo = tiempo_estimado_total / num_ciclos

    # Ejecución de cada ciclo
    for ciclo in range(num_ciclos):
        # Control de pausa mediante eventos
        self.pausa_event.wait()

        # Transición a estado Ejecutando
        proceso.estados['Ejecutando'] = True
        self.labels_estados[index].config(
            text=f"Proceso {proceso.id}: Ejecutando (Ciclo {ciclo+1}/
↪{num_ciclos})"
```

```
        )
        self.progress_bars[index]['value'] = 3
        self.root.update_idletasks()

        # Simulación del tiempo de ejecución con factor de prioridad
        tiempo_actual = (tiempo_por_ciclo / proceso.prioridad) * 1.3
        start_time = time.time()
        time.sleep(tiempo_actual)
        tiempo_ciclo = time.time() - start_time
        proceso.tiempo_ejecucion_real += tiempo_ciclo

        # Incrementar contador de ciclos
        proceso.ciclos_realizados += 1

        # Transición a estado Bloqueado (excepto en el último ciclo)
        if ciclo < num_ciclos - 1:
            self.pausa_event.wait()
            proceso.estados['Bloqueado'] = True
            self.labels_estados[index].config(
                text=f"Proceso {proceso.id}: Bloqueado (Después del ciclo␣
↪{ciclo+1})"
            )
            self.progress_bars[index]['value'] = 4
            self.root.update_idletasks()
            time.sleep(random.uniform(0.3, 1) * 1.3)

    # Transición final a estado Terminado
    proceso.estados['Terminado'] = True
    self.labels_estados[index].config(
        text=f"Proceso {proceso.id}: Terminado (Ciclos realizados: {proceso.
↪ciclos_realizados})"
    )
    self.progress_bars[index]['value'] = 5

    # Actualización de contadores y visualizaciones
    self.procesos_terminados += 1
    if self.procesos_terminados == len(self.procesos):
        self.actualizar_tabla()
    self.actualizar_grafica()
```

1.3 Fundamentos de la Implementación Multihilo

La simulación utiliza el módulo `threading` de Python para proporcionar ejecución concurrente de procesos. Cada proceso se ejecuta en su propio hilo, permitiendo una simulación más realista del comportamiento de un sistema operativo.

1.3.1 Aspectos clave de la concurrencia:

1. **Creación de hilos**:

```
[ ]: thread = threading.Thread(
         target=self.simular_proceso_con_actualizacion,
         args=(proceso, i)
     )
     thread.start()
```

2. **Sincronización mediante eventos**:

```
[ ]: self.pausa_event = threading.Event()
     self.pausa_event.set()  # Inicialmente no está en pausa
```

3. **Protección contra condiciones de carrera**:

```
[ ]: self.lock_ejecucion = threading.Lock()
```

4. **Gestión de estados concurrentes**:

```
[ ]: # En pausar_simulacion
     self.pausa_event.clear()  # Detiene los hilos

     # En continuar_simulacion
     self.pausa_event.set()  # Permite que los hilos continúen
```

5. **Actualización segura de componentes gráficos**:

```
[ ]: self.root.update_idletasks()  # Actualización segura en contexto multihilo
```

1.4 Interfaz Gráfica

El simulador implementa una interfaz gráfica robusta utilizando la biblioteca `tkinter`, el estándar de facto para interfaces gráficas en Python.

1.4.1 Componentes de la interfaz:

1. **Ventana principal**: Contenedor raíz de toda la interfaz.

```
self.root = tk.Tk()
self.root.title("Simulador de Procesos de Procesador")
```

2. **Frame de estados**: Organiza los elementos visuales para cada proceso.

```
self.frame_estados = tk.Frame(self.root)
self.frame_estados.pack(padx=10, pady=10)
```

3. **Controles de simulación**:

- Botón de inicio: `self.boton_iniciar`
- Botón de pausa: `self.boton_pausa`
- Botón de continuación: `self.boton_continuar`

4. **Indicadores visuales por proceso**:
   - `labels_estados`: Muestran el estado actual (Ejecutando, Bloqueado, etc.)
   - `progress_bars`: Barras de progreso que indican la fase actual del proceso
   - `info_labels`: Información detallada sobre los recursos asignados
5. **Gráficas integradas**:

```
self.fig = Figure(figsize=(5, 3), dpi=100)
self.ax = self.fig.add_subplot(111)
self.canvas = FigureCanvasTkAgg(self.fig, master=self.root)
self.canvas.get_tk_widget().pack(pady=10)
```

6. **Tabla de resultados**:

```
self.treeview = ttk.Treeview(
    self.root,
    columns=("ID", "Ciclos", "Tiempo Estimado", "Tiempo Real"),
    show="headings"
)
```

1.4.2 Flujo de actualización de interfaz:

El simulador implementa un sistema de actualización de interfaz basado en eventos que permite visualizar en tiempo real los cambios en los procesos:

1. Al cambiar el estado de un proceso, se actualiza su etiqueta correspondiente.
2. Las barras de progreso muestran el avance en cada fase del ciclo de vida.
3. La gráfica se actualiza dinámicamente para reflejar los tiempos acumulados.
4. La tabla final muestra un resumen comparativo de tiempos estimados y reales.

1.5 Simulación de Procesos: Algoritmos y Matemáticas

La simulación de procesos se basa en un modelo matemático que replica el comportamiento de un sistema operativo con planificación de procesos basada en prioridades. A continuación se detallan los algoritmos y principios matemáticos implementados:

1.5.1 1. Algoritmo de planificación:

La simulación implementa un algoritmo de planificación no preemptivo basado en prioridades, donde: - Cada proceso tiene una prioridad única (valor menor = prioridad mayor) - El tiempo de ejecución por ciclo se ajusta según la prioridad: `python tiempo_actual = (tiempo_por_ciclo / proceso.prioridad) * 1.3`

1.5.2 2. Modelo probabilístico:

La simulación incorpora elementos aleatorios para simular la variabilidad de un sistema real: - Tiempos de espera: `random.uniform(0.3, 1) * 1.3` - Número de ciclos: `random.randint(20, 30)` - Tiempos estimados: `random.uniform(1, 5)`

1.5.3 3. Cálculo de tiempos reales:

La simulación mide el tiempo real transcurrido durante la ejecución:

```
[ ]: start_time = time.time()
time.sleep(tiempo_actual)
tiempo_ciclo = time.time() - start_time
proceso.tiempo_ejecucion_real += tiempo_ciclo
```

1.5.4 4. Distribución de ciclos:

El tiempo total estimado se distribuye entre varios ciclos:

```
[ ]: tiempo_estimado_total = proceso.tiempo_ejecucion
num_ciclos = random.randint(20, 30)
tiempo_por_ciclo = tiempo_estimado_total / num_ciclos
```

1.6 Gráficas y Visualización: Técnicas Avanzadas

El simulador utiliza la biblioteca `matplotlib` integrada con `tkinter` para proporcionar visualizaciones dinámicas del estado de los procesos.

1.6.1 Configuración de la gráfica:

```
[ ]: def actualizar_grafica(self):
    # Limpiar gráfica anterior
    self.ax.clear()
    self.ax.set_title("Progreso de Ejecución de Procesos")
    self.ax.set_xlabel("ID del Proceso")
    self.ax.set_ylabel("Tiempo (s)")

    ids = [proceso.id for proceso in self.procesos]
    tiempos_estimados = [proceso.tiempo_ejecucion for proceso in self.procesos]
    tiempos_reales = [proceso.tiempo_ejecucion_real for proceso in self.
 ↪procesos]

    # Dibujar barras para tiempos estimados
    self.ax.bar(ids, tiempos_estimados, alpha=0.4, color='lightblue',␣
 ↪label='Tiempo Estimado')

    # Dibujar barras para tiempos reales
    self.ax.bar(ids, tiempos_reales, alpha=0.7, color='blue', label='Tiempo␣
 ↪Ejecutado')
```

1.6.2 Técnicas avanzadas de visualización:

1. **Anotaciones dinámicas**:

```
for i, (est, real) in enumerate(zip(tiempos_estimados, tiempos_reales)):
    self.ax.text(ids[i], est + 0.1, f"{est:.2f}s", ha='center', va='bottom', fontsize=8)
```

```
        if real > 0:
            self.ax.text(ids[i], real / 2, f"{real:.2f}s", ha='center', va='center',
                         fontsize=8, color='white', fontweight='bold')
```

2. **Ajuste automático de escala**:

```
max_tiempo = max(max(tiempos_estimados), max(tiempos_reales)) * 1.2
self.ax.set_ylim(0, max_tiempo)
```

3. **Personalización de ejes**:

```
self.ax.set_xticks(ids)
```

4. **Mejoras de legibilidad**:

```
self.ax.legend(loc='upper right')
self.ax.grid(axis='y', linestyle='--', alpha=0.7)
```

1.7 Modelo de Estados y Transiciones

El simulador implementa un modelo completo de máquina de estados para representar el ciclo de vida de los procesos.

1.7.1 Diagrama de estados:

[ ]:
```
   [Nuevo] --> [Listo] --> [Ejecutando] <--> [Bloqueado]
                                |
                                v
                           [Terminado]
```

1.7.2 Implementación del modelo de estados:

Cada proceso mantiene un diccionario que representa su estado actual:

[ ]:
```
self.estados = {
    'Nuevo': False,
    'Listo': False,
    'Ejecutando': False,
    'Bloqueado': False,
    'Terminado': False
}
```

1.7.3 Transiciones de estado en la simulación:

1. **Transición a Ejecutando**:

```
proceso.estados['Ejecutando'] = True
self.labels_estados[index].config(
    text=f"Proceso {proceso.id}: Ejecutando (Ciclo {ciclo+1}/{num_ciclos})"
)
self.progress_bars[index]['value'] = 3
```

2. **Transición a Bloqueado**:

```
proceso.estados['Bloqueado'] = True
self.labels_estados[index].config(
    text=f"Proceso {proceso.id}: Bloqueado (Después del ciclo {ciclo+1})"
)
self.progress_bars[index]['value'] = 4
```

3. **Transición a Terminado**:

```
proceso.estados['Terminado'] = True
self.labels_estados[index].config(
    text=f"Proceso {proceso.id}: Terminado (Ciclos realizados: {proceso.ciclos_realizados}
)
self.progress_bars[index]['value'] = 5
```

1.8 Mecanismos de Sincronización y Control de Flujo

La implementación utiliza mecanismos avanzados para controlar el flujo de la simulación y sincronizar los hilos de ejecución.

1.8.1 1. Control de pausa mediante eventos:

[ ]:
```
def pausar_simulacion(self):
    self.pausa_event.clear()  # Detiene los hilos
    self.boton_pausa.pack_forget()
    self.boton_continuar.pack(pady=5)

    # Actualizar la interfaz para reflejar la pausa
    for i, proceso in enumerate(self.procesos):
        if proceso.estados['Ejecutando'] or proceso.estados['Bloqueado']:
            self.labels_estados[i].config(
                text=f"Proceso {proceso.id}: Pausado (Ciclos realizados:␣
 ↪{proceso.ciclos_realizados})"
            )
```

1.8.2 2. Reanudación de la simulación:

[ ]:
```
def continuar_simulacion(self):
    self.pausa_event.set()  # Permite que los hilos continúen
    self.boton_continuar.pack_forget()
    self.boton_pausa.pack(pady=5)
```

1.8.3 3. Espera activa en hilos:

[ ]:
```
# Dentro de simular_proceso_con_actualizacion
self.pausa_event.wait()  # Espera si la simulación está pausada
```

1.9 Análisis de Rendimiento y Resultados

El simulador realiza un seguimiento detallado del rendimiento de cada proceso, comparando tiempos estimados con tiempos reales.

1.9.1 Componentes de la tabla de resultados:

[ ]:
```python
def actualizar_tabla(self):
    # Limpiar la tabla antes de llenarla
    for row in self.treeview.get_children():
        self.treeview.delete(row)

    # Llenar la tabla con los datos actuales
    for proceso in self.procesos:
        tiempo_real = getattr(proceso, 'tiempo_ejecucion_real', 0)
        self.treeview.insert(
            "", "end", values=(
                proceso.id,
                proceso.ciclos_realizados,
                f"{proceso.tiempo_ejecucion:.2f}",
                f"{tiempo_real:.2f}"
            )
        )

    # Mostrar la tabla solo si todos los procesos han terminado
    if self.procesos_terminados == len(self.procesos):
        self.treeview.pack(pady=10)
```

1.10 Modelo de Recursos y Asignación

El simulador implementa un modelo de recursos simplificado que simula la asignación de hardware a los procesos:

1. **Asignación de núcleos**:

   ```
   proceso.nucleo = random.randint(1, 4)  # Simula un procesador de 4 núcleos
   ```

2. **Asignación de hilos**:

   ```
   proceso.hilo = random.randint(1, 8)  # Simula 8 hilos por núcleo
   ```

3. **Asignación de memoria**:

   ```
   proceso.memoria_asignada = random.randint(100, 1000)  # Memoria en MB
   ```

1.11 Consideraciones de Diseño y Arquitectura

El simulador sigue varios principios de diseño de software importantes:

1. **Separación de responsabilidades**:
   - Clase `Proceso` para el modelo de datos
   - Clase `SimuladorProcesador` para la lógica de simulación y UI

2. **Programación orientada a eventos**:
   - Uso de callbacks para botones
   - Sistema de eventos para sincronización
3. **Arquitectura MVC simplificada**:
   - Modelo: Clases de datos (`Proceso`)
   - Vista: Elementos de interfaz tkinter
   - Controlador: Métodos de `SimuladorProcesador`
4. **Diseño reactivo**:
   - Actualizaciones en tiempo real de la interfaz
   - Visualización dinámica de estados

1.12 Posibles Extensiones y Mejoras

El simulador actual podría extenderse con las siguientes funcionalidades:

1. **Algoritmos de planificación adicionales**:
   - Round Robin
   - Shortest Job First
   - Planificación multinivel
2. **Simulación de interrupciones**:
   - Eventos aleatorios que afecten la ejecución
   - Prioridades dinámicas
3. **Métricas avanzadas de rendimiento**:
   - Tiempo de espera promedio
   - Tiempo de respuesta
   - Throughput del sistema
4. **Visualizaciones adicionales**:
   - Diagrama de Gantt
   - Gráficos de utilización de recursos
   - Análisis estadístico de rendimiento
5. **Persistencia de datos**:
   - Guardar y cargar configuraciones de simulación
   - Exportar resultados a formatos estándar (CSV, JSON)

1.13 Comparación con Sistemas Reales

Este simulador modela de forma simplificada varios conceptos fundamentales de sistemas operativos reales:

Concepto	Implementación en el simulador	Sistema operativo real
Planificación	Basada en prioridades no preemptiva	Algoritmos complejos con quantum de tiempo
Estados	5 estados básicos	Modelos más complejos con subestados
Recursos	Asignación aleatoria	Gestión dinámica basada en demanda
Sincronización	Eventos simples	Semáforos, mutexes, condiciones

Concepto	Implementación en el simulador	Sistema operativo real
Medición de tiempo	Tiempo real del sistema	Ciclos de CPU y contadores de hardware

1.14 Conclusiones

El Simulador de Procesos de Procesador ofrece una representación visual e interactiva de conceptos fundamentales de sistemas operativos. A través de su interfaz gráfica, permite observar:

1. El ciclo de vida completo de los procesos
2. La ejecución concurrente mediante hilos
3. La comparación entre tiempos estimados y reales
4. La transición entre diferentes estados de proceso
5. El efecto de las prioridades en la planificación

Esta herramienta tiene un valor educativo significativo para comprender los principios de funcionamiento de sistemas operativos modernos y los desafíos asociados con la planificación de procesos y administración de recursos.