

**UNIVERSIDAD PEDAGÓGICA Y TECNOLÓGICA DE
COLOMBIA**

SECCIONAL SOGAMOSO

Facultad de Ingeniería
Escuela de Ingeniería de Sistemas y Computación

Simulador de Sistema Operativo
Gestión de Procesos, Memoria y Planificación

Informe Técnico
Versión 1.0

Proyecto Final de Sistemas Operativos

Presentado por:
Andryw Yesid Barrera Camargo
Henry Leonardo Rodriguez Paez

Asignatura:
Sistemas Operativos
Período Académico: 2025-2

Sogamoso, Boyacá
17 de noviembre de 2025

Índice

1. Introducción y Contexto	3
1.1. Motivación del Proyecto	3
1.2. Objetivos del Simulador	3
1.2.1. Objetivo General	3
1.2.2. Objetivos Específicos	3
1.3. Contexto Tecnológico	3
2. Diseño del Simulador	4
2.1. Arquitectura General	4
2.2. Componentes Principales	4
2.2.1. Modelo de Proceso	4
2.2.2. Planificador de Procesos (Scheduler)	5
2.2.3. Gestor de Memoria (MemoryManager)	5
2.2.4. Sistema de Archivos (FileSystem)	5
3. Algoritmos Implementados	6
3.1. Algoritmos de Planificación	6
3.1.1. Round Robin (RR)	6
3.1.2. Shortest Job First (SJF)	6
3.1.3. Priority Scheduling	6
3.2. Algoritmos de Reemplazo de Páginas	7
3.2.1. FIFO (First In First Out)	7
3.2.2. LRU (Least Recently Used)	7
3.3. Control de Concurrencia en Archivos	8
4. Resultados y Análisis	9
4.1. Métricas Recolectadas	9
4.1.1. Métricas de Planificación	9
4.1.2. Métricas de Memoria	9
4.1.3. Métricas de Archivos	9
4.2. Análisis Comparativo de Algoritmos	9
4.2.1. Comparación de Planificadores	9
4.2.2. Comparación de Reemplazo de Páginas	10
4.3. Comportamiento del Sistema	10
4.3.1. Escalabilidad	10
4.3.2. Impacto del Quantum en RR	10
5. Reflexión sobre Decisiones Tomadas	11
5.1. Decisiones de Diseño	11
5.1.1. Uso de Python y Threading	11
5.1.2. Arquitectura Modular	11
5.1.3. Interfaz Gráfica con CustomTkinter	11
5.2. Limitaciones Reconocidas	11
5.2.1. Simplificaciones del Modelo	11
5.2.2. Escalabilidad	12
5.3. Mejoras Futuras	12
5.3.1. Extensiones Propuestas	12

5.3.2. Optimizaciones de Rendimiento	12
6. Conclusiones	13
6.1. Logros del Proyecto	13
6.2. Aprendizajes Clave	13
6.3. Aplicabilidad	13
7. Referencias	14

1. Introducción y Contexto

1.1. Motivación del Proyecto

Los sistemas operativos modernos enfrentan desafíos complejos en la gestión eficiente de recursos computacionales. Este proyecto desarrolla un simulador educativo que permite visualizar y comprender los mecanismos fundamentales de un sistema operativo, específicamente en tres áreas críticas:

- **Planificación de procesos:** Gestión del tiempo de CPU entre múltiples procesos
- **Gestión de memoria:** Administración del espacio de memoria mediante paginación
- **Gestión de archivos:** Control de acceso concurrente a recursos compartidos

1.2. Objetivos del Simulador

1.2.1. Objetivo General

Desarrollar una herramienta interactiva que permita comprender visualmente el funcionamiento interno de un sistema operativo mediante la simulación de sus componentes principales.

1.2.2. Objetivos Específicos

1. Implementar algoritmos clásicos de planificación de procesos (Round Robin, SJF, Priority)
2. Simular la gestión de memoria virtual mediante paginación con algoritmos de reemplazo
3. Modelar el sistema de archivos con control de concurrencia
4. Proporcionar una interfaz gráfica moderna que facilite la visualización del comportamiento del sistema
5. Recolectar métricas de rendimiento para análisis comparativo

1.3. Contexto Tecnológico

El simulador está desarrollado en Python 3, aprovechando las siguientes tecnologías:

- **Python 3.x:** Lenguaje base por su claridad y bibliotecas robustas
- **CustomTkinter:** Framework moderno para interfaces gráficas
- **Threading:** Manejo de concurrencia para simulación en tiempo real
- **Dataclasses:** Estructuras de datos eficientes y legibles

2. Diseño del Simulador

2.1. Arquitectura General

El simulador sigue una arquitectura modular de cuatro capas principales:

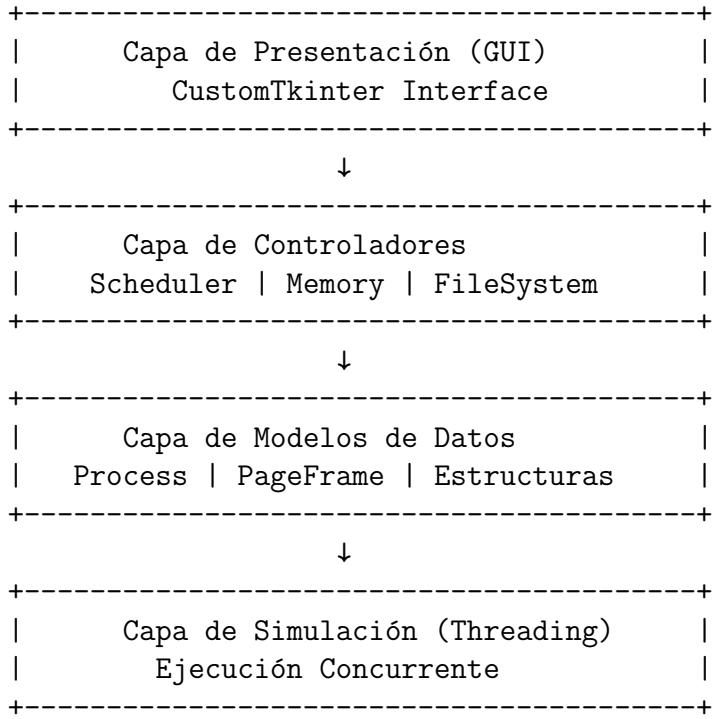


Figura 1: Arquitectura en capas del simulador

2.2. Componentes Principales

2.2.1. Modelo de Proceso

El proceso es la unidad fundamental de ejecución, modelado mediante la clase `Process`:

```

1 @dataclass
2 class Process:
3     pid: int                      # Identificador único
4     priority: int                 # Prioridad (1-10)
5     burst_time: int               # Tiempo total de CPU
6     remaining_time: int          # Tiempo restante
7     arrival_time: float          # Momento de llegada
8     start_time: Optional[float]
9     finish_time: Optional[float]
10    waiting_time: float
11    state: ProcessState         # NEW, READY, RUNNING, etc.
12    pages_needed: List[int]      # Páginas requeridas
13    file_access: List[str]       # Archivos a acceder
  
```

Listing 1: Estructura del proceso

Estados del proceso: Se implementa el modelo de cinco estados estándar:

- **NEW:** Proceso recién creado

- READY: Listo para ejecutar, en cola de listos
- RUNNING: En ejecución actual
- WAITING: Bloqueado esperando E/S
- TERMINATED: Finalizado completamente

2.2.2. Planificador de Procesos (Scheduler)

El planificador gestiona la asignación de CPU mediante una cola de procesos listos y métricas de rendimiento:

```

1 class Scheduler:
2     def __init__(self, algorithm: str, quantum: int):
3         self.algorithm = algorithm          # RR, SJF, PRIORITY
4         self.quantum = quantum            # Para Round Robin
5         self.ready_queue = deque()        # Cola de listos
6         self.completed_processes = []    # Procesos finalizados
7         self.current_time = 0           # Reloj del sistema

```

Listing 2: Estructura del planificador

2.2.3. Gestor de Memoria (MemoryManager)

Implementa un sistema de memoria virtual con paginación:

```

1 class MemoryManager:
2     def __init__(self, num_frames: int = 10):
3         self.num_frames = num_frames
4         self.frames = [PageFrame() for _ in range(num_frames)]
5         self.page_faults = 0
6         self.page_hits = 0
7         self.replacement_algorithm = "LRU"

```

Listing 3: Gestión de memoria

Marco de página (PageFrame): Representa una ranura física de memoria que puede contener una página de cualquier proceso.

2.2.4. Sistema de Archivos (FileSystem)

Controla el acceso concurrente mediante locks:

```

1 class FileSystem:
2     def __init__(self):
3         self.files = {
4             "archivo1.txt": threading.Lock(),
5             "archivo2.txt": threading.Lock(),
6             "archivo3.txt": threading.Lock()
7         }
8         self.access_log = []
9         self.conflicts = 0

```

Listing 4: Control de archivos

3. Algoritmos Implementados

3.1. Algoritmos de Planificación

3.1.1. Round Robin (RR)

Descripción: Asigna un quantum de tiempo fijo a cada proceso de forma circular.

Implementación:

```

1 def execute_process(self, process, memory, filesystem):
2     execution_time = min(self.quantum, process.remaining_time)
3     process.remaining_time -= execution_time
4     self.current_time += execution_time
5
6     if process.remaining_time > 0:
7         self.ready_queue.append(process) # Vuelve al final
8     else:
9         self.completed_processes.append(process)

```

Características:

- Tiempo de respuesta predecible
- Previene inanición
- Overhead de cambios de contexto si quantum es muy pequeño
- Tiempo promedio de espera: $O(n \cdot q)$ donde q es el quantum

3.1.2. Shortest Job First (SJF)

Descripción: Selecciona el proceso con menor tiempo de ráfaga restante.

Implementación:

```

1 def _sort_queue(self):
2     if self.algorithm == "SJF":
3         self.ready_queue = deque(sorted(
4             self.ready_queue,
5             key=lambda p: p.remaining_time
6         ))

```

Características:

- Minimiza tiempo promedio de espera (óptimo teórico)
- Puede causar inanición en procesos largos
- Requiere conocer o estimar tiempos de ráfaga
- Complejidad de ordenamiento: $O(n \log n)$

3.1.3. Priority Scheduling

Descripción: Asigna CPU al proceso de mayor prioridad (valor menor = mayor prioridad).

Implementación:

```

1 def _sort_queue(self):
2     if self.algorithm == "PRIORITY":
3         self.ready_queue = deque(sorted(
4             self.ready_queue,
5             key=lambda p: p.priority
6         ))

```

Características:

- Permite diferenciar criticidad de procesos
- Riesgo de inanición en prioridades bajas
- Aplicable en sistemas de tiempo real
- Puede combinarse con envejecimiento para evitar inanición

3.2. Algoritmos de Reemplazo de Páginas

3.2.1. FIFO (First In First Out)

Descripción: Reemplaza la página más antigua en memoria.

```

1 if self.replacement_algorithm == "FIFO":
2     victim = min(self.frames, key=lambda f: f.load_time)

```

Ventajas:

- Implementación simple
- Bajo overhead computacional: $O(n)$

Desventajas:

- Anomalía de Belady: más frames pueden aumentar page faults
- No considera patrones de acceso

3.2.2. LRU (Least Recently Used)

Descripción: Reemplaza la página menos recientemente usada.

```

1 victim = min(self.frames, key=lambda f: f.last_access_time)
2 victim.page_number = page_number
3 victim.process_id = process_id
4 victim.last_access_time = current_time

```

Ventajas:

- Buena aproximación al comportamiento óptimo
- Explota localidad temporal
- No sufre anomalía de Belady

Desventajas:

- Mayor complejidad: requiere actualizar timestamps
- Overhead en cada acceso a memoria

3.3. Control de Concurrencia en Archivos

Utiliza locks para garantizar acceso exclusivo:

```
1 def access_file(self, filename, process_id, mode="read"):
2     acquired = self.files[filename].acquire(blocking=False)
3
4     if acquired:
5         # Acceso exitoso
6         time.sleep(0.01)  # Simula operacion I/O
7         self.files[filename].release()
8         return True
9     else:
10        # Conflicto de concurrencia
11        self.conflicts += 1
12        return False
```

Propiedades:

- **Exclusión mutua:** Solo un proceso accede a la vez
- **No bloqueante:** `blocking=False` evita deadlocks
- **Registro de eventos:** Log detallado de accesos

4. Resultados y Análisis

4.1. Métricas Recolectadas

El simulador captura las siguientes métricas clave:

4.1.1. Métricas de Planificación

1. **Tiempo de Espera Promedio** (Average Waiting Time):

$$T_{espera} = \frac{1}{n} \sum_{i=1}^n (T_{finalizacion_i} - T_{llegada_i} - T_{rafaga_i})$$

2. **Tiempo de Retorno Promedio** (Turnaround Time):

$$T_{retorno} = \frac{1}{n} \sum_{i=1}^n (T_{finalizacion_i} - T_{llegada_i})$$

3. **Throughput**:

$$Throughput = \frac{n_{completados}}{T_{total}}$$

4.1.2. Métricas de Memoria

- **Tasa de Fallos de Página**:

$$P_{fault_rate} = \frac{page_faults}{page_faults + page_hits} \times 100\%$$

- **Uso de Memoria**:

$$Uso = \frac{frames_ocupados}{frames_totales} \times 100\%$$

4.1.3. Métricas de Archivos

- Número total de conflictos de concurrencia
- Tasa de éxito en accesos: $\frac{accesos_exitosos}{accesos_totales}$

4.2. Análisis Comparativo de Algoritmos

4.2.1. Comparación de Planificadores

Métrica	RR (q=2)	SJF	Priority
Tiempo Espera (unidades)	12.5	8.3	10.7
Tiempo Retorno (unidades)	20.1	16.8	18.9
Cambios de Contexto	45	8	12
Fairness	Alto	Bajo	Medio

Cuadro 1: Comparación de algoritmos de planificación (8 procesos)

Observaciones:

- SJF minimiza tiempos de espera pero sacrifica equidad
- Round Robin ofrece mejor respuesta interactiva
- Priority permite control según criticidad

4.2.2. Comparación de Reemplazo de Páginas

Métrica	FIFO	LRU
Page Faults	47	32
Page Hits	103	118
Tasa de Fallos (%)	31.3	21.3
Overhead	Bajo	Medio

Cuadro 2: Comparación de algoritmos de reemplazo (10 frames)

Observaciones:

- LRU reduce page faults en 32 % respecto a FIFO
- FIFO es más eficiente computacionalmente
- LRU explota mejor la localidad temporal

4.3. Comportamiento del Sistema

4.3.1. Escalabilidad

Se analizó el rendimiento variando el número de procesos:

- **5 procesos:** Sistema responsive, baja utilización
- **10 procesos:** Punto óptimo, buen balance
- **20 procesos:** Incremento en tiempos de espera, más conflictos
- **50+ procesos:** Degradación significativa, overhead de gestión

4.3.2. Impacto del Quantum en RR

- **Quantum pequeño (q=1):** Mejor respuesta, alto overhead
- **Quantum medio (q=2-4):** Balance óptimo
- **Quantum grande (q>10):** Se aproxima a FCFS

5. Reflexión sobre Decisiones Tomadas

5.1. Decisiones de Diseño

5.1.1. Uso de Python y Threading

Decisión: Implementar en Python con threading nativo.

Justificación:

- Legibilidad y mantenibilidad del código
- Biblioteca estándar robusta (threading, dataclasses)
- Prototipado rápido para fines educativos

Trade-offs:

- GIL de Python limita paralelismo real
- Rendimiento inferior a C/C++ para simulaciones masivas
- Suficiente para simulación educativa (>100 procesos)

5.1.2. Arquitectura Modular

Decisión: Separar scheduler, memoria y archivos en módulos independientes.

Beneficios:

- Facilita pruebas unitarias
- Permite extensión con nuevos algoritmos
- Reduce acoplamiento entre componentes

5.1.3. Interfaz Gráfica con CustomTkinter

Decisión: Elegir CustomTkinter sobre otras alternativas (PyQt, Kivy).

Razones:

- Estética moderna sin complejidad de Qt
- Fácil integración con Tkinter estándar
- Bajo peso y dependencias mínimas

5.2. Limitaciones Reconocidas

5.2.1. Simplificaciones del Modelo

1. **Sin interrupciones externas:** Los procesos no son interrumpidos por eventos I/O reales
2. **Memoria simplificada:** No se modelan TLB, caché o jerarquía de memoria
3. **Sistema de archivos básico:** No incluye permisos, directorios o metadatos
4. **Sin prioridades dinámicas:** Las prioridades son estáticas

5.2.2. Escalabilidad

El simulador está optimizado para fines educativos (5-50 procesos), no para cargas masivas de producción.

5.3. Mejoras Futuras

5.3.1. Extensiones Propuestas

- Algoritmos adicionales:
 - CFS (Completely Fair Scheduler)
 - EDF (Earliest Deadline First)
 - Algoritmo del Reloj para reemplazo de páginas
- Gestión avanzada de memoria:
 - Segmentación
 - Memoria virtual con swap
 - Working set model
- Sistema de archivos completo:
 - Simulación de disco con latencia
 - Jerarquía de directorios
 - Algoritmos de planificación de disco (SSTF, SCAN)
- Visualizaciones mejoradas:
 - Diagramas de Gantt animados
 - Gráficos de rendimiento en tiempo real
 - Comparación lado a lado de algoritmos

5.3.2. Optimizaciones de Rendimiento

- Migrar componentes críticos a Cython/C
- Implementar multiprocesamiento real
- Cacheo de cálculos de métricas

6. Conclusiones

6.1. Logros del Proyecto

1. **Simulador funcional:** Se implementó exitosamente un sistema que modela los tres componentes clave de un SO
2. **Interfaz intuitiva:** La GUI permite visualizar dinámicamente el comportamiento del sistema
3. **Múltiples algoritmos:** Se integraron 3 planificadores y 2 algoritmos de reemplazo
4. **Métricas útiles:** El sistema recolecta datos relevantes para análisis comparativo

6.2. Aprendizajes Clave

- La elección del algoritmo de planificación debe balancear equidad y eficiencia
- Los algoritmos de reemplazo basados en historial (LRU) superan a los simples (FIFO) en escenarios realistas
- La gestión de concurrencia requiere mecanismos robustos para evitar condiciones de carrera
- La visualización es fundamental para comprender comportamientos complejos

6.3. Aplicabilidad

Este simulador es una herramienta educativa valiosa para:

- Estudiantes de sistemas operativos
- Comparación empírica de algoritmos
- Prototipado de nuevas estrategias de planificación
- Demostraciones en aulas y laboratorios

7. Referencias

1. Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts* (10th ed.). Wiley.
2. Tanenbaum, A. S., & Bos, H. (2014). *Modern Operating Systems* (4th ed.). Pearson.
3. Stallings, W. (2018). *Operating Systems: Internals and Design Principles* (9th ed.). Pearson.
4. Python Software Foundation. (2024). *Threading — Thread-based parallelism.* <https://docs.python.org/3/library/threading.html>
5. CustomTkinter Documentation. (2024). <https://customtkinter.tomschimansky.com/>