UNIVERSIDAD AUTÓNOMA DE TAMAULIPAS

FACULTAD DE INGENIERÍA TAMPICO

Manual Técnico

SISTEMAS OPERATIVOS

11:00 am - 12:00 pm

ALUMNO:

García Hernández Jorge Eduardo
Gaytán Gonzalez Ángel Alejandro
Gonzalez Mercado Francisco Antonio
Guzmán Ortega José Emilio
Martínez Azuara Juan Francisco

SEMESTRE: 6° **GRUPO:** "J"

CARRERA:

ING. EN SISTEMAS COMPUTACIONALES

PROFESOR:

Muñoz Quintero Dante Adolfo

Entrega de Trabajo

Martes 02 de Diciembre del 2025

VERDAD, BELLEZA, PROBIDAD

Universidad Autónoma de TAMAULIPAS

Facultad de Ingeniería Tampico

Indice

Manual Técnico - Producto Integrador Final

Simulador Gestión de Procesos

1. Introducción

1.1 Propósito del Documento

Este manual técnico documenta de manera exhaustiva la arquitectura, diseño, implementación y operación del Simulador de Gestión de Procesos. Está dirigido a desarrolladores, ingenieros de software y estudiantes avanzados que necesiten comprender, mantener o extender el sistema.

1.2 Alcance del Sistema

El simulador implementa los siguientes componentes principales de un sistema operativo:

- Planificación de CPU mediante algoritmo FCFS (First Come, First Served)
- Gestión de memoria con paginación por demanda
- Memoria virtual con área de swap en disco
- Sincronización de procesos mediante semáforos
- Memoria compartida entre procesos
- Detección y resolución de deadlocks

1.3 Tecnologías y Herramientas

- Lenguaje: Python 3.8+ (compatible hasta Python 3.13)
- Paradigma: Programación Orientada a Objetos (POO)
- Estructuras de datos: deque, dict, list, set
- Librerías estándar: sys, time, random, os, configparser, logging, collections, enum
- Control de versiones: Git (recomendado)

1.4 Audiencia del Documento

Este manual está diseñado para:

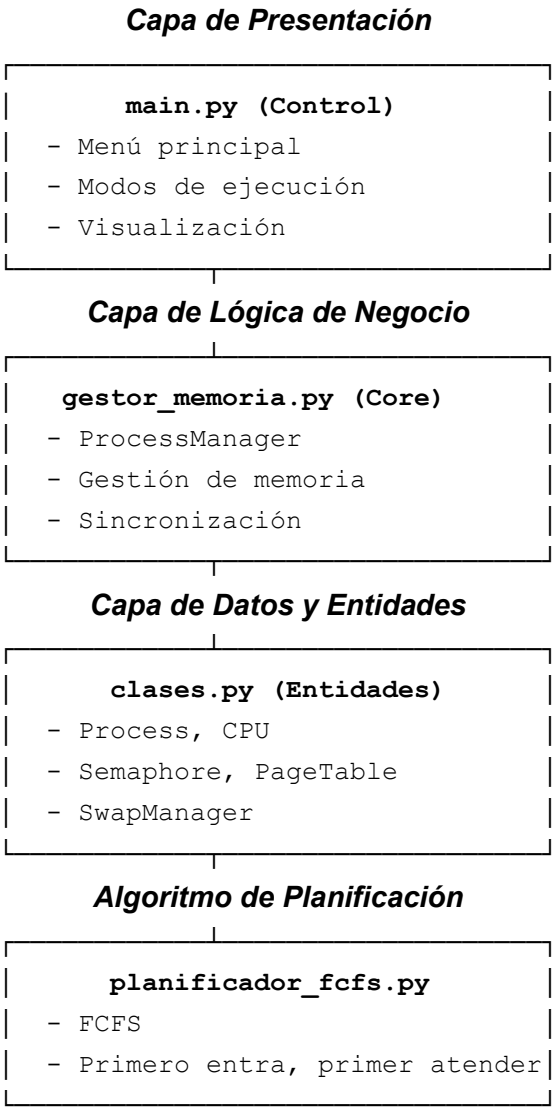
- Desarrolladores que necesiten modificar o extender el sistema
- Ingenieros de QA para realizar pruebas técnicas
- Estudiantes avanzados que estudien sistemas operativos
- Profesores que utilicen el sistema con fines educativos

2. Arquitectura del Sistema

2.1 Vista General de la Arquitectura

El sistema sigue una arquitectura modular de capas donde cada componente tiene responsabilidades bien definidas. La separación de preocupaciones permite modificar componentes individuales sin afectar al resto del sistema.

2.1.1 Diagrama de Componentes



2.2 Componentes Principales

Módulo	Responsabilidad	Componentes Clave
main.py	Control de flujo y UI	main(), run_automatic_mode(), run_interactive_mode()
clases.py	Definición de entidades	Process, CPU, Semaphore, PageTable, SwapManager
gestor_memoria.py	Gestión centralizada	ProcessManager (400+ líneas)
planificador_fcfs.py	Algoritmo de planificación	FCFSScheduler
config.ini	Configuración del sistema	Parámetros de memoria, simulación y logs

2.3 Patrones de Diseño Implementados

2.3.1 Manager Pattern

La clase ProcessManager actúa como punto central de coordinación para todas las operaciones del sistema. Este patrón simplifica la comunicación entre componentes y centraliza la lógica de negocio.

```
class ProcessManager:
    def __init__(self, config_file):
        # Inicializa todos los subsistemas
        self.cpu = CPU()
        self.scheduler = FCFSScheduler()
        self.ram = [None] * total_frames
        self.swap = SwapManager(...)
```

2.3.2 State Pattern

Los estados de los procesos se manejan mediante un Enum, lo que garantiza estados válidos y facilita las transiciones.

```
class ProcessState(Enum):
    NEW = 'Nuevo'
    READY = 'Listo'
    RUNNING = 'En Ejecucion'
    BLOCKED = 'Bloqueado'
    TERMINATED = 'Terminado'
```

2.3.3 Strategy Pattern

El sistema está diseñado para permitir diferentes algoritmos de planificación. Aunque actualmente solo FCFS está implementado, la arquitectura permite añadir Round Robin, Priority Scheduling, etc., sin modificar el código existente.

3. Clases y Componentes Detallados

3.1 Clase Process

3.1.1 Descripción

La clase Process representa un proceso en el sistema operativo. Contiene toda la información necesaria para gestionar el ciclo de vida del proceso, su uso de CPU, memoria y sincronización.

3.1.2 Atributos Principales

Atributo	Tipo	Descripción
pid	int	Identificador único del proceso (auto-incremental)
size_kb	int	Tamaño del proceso en kilobytes

Atributo	Tipo	Descripción
num_pages	int	Número total de páginas (calculado: size_kb / page_size_kb)
state	ProcessState	Estado actual del proceso (NEW, READY, RUNNING, BLOCKED, TERMINATED)
priority	int	Prioridad del proceso (1=mínima, 10=máxima)
cpu_burst	int	Tiempo total de CPU necesario (en ciclos)
remaining_cpu	int	Tiempo de CPU restante
pages_in_ram	set	Conjunto de números de páginas actualmente en RAM
pages_in_swap	set	Conjunto de números de páginas actualmente en swap
page_faults	int	Contador de fallos de página ocurridos
blocked_on	str	Nombre del recurso por el que está bloqueado (None si no bloqueado)

3.1.3 Métodos Importantes

__init__(size_kb, lifetime, priority, cpu_burst):

Constructor que inicializa un nuevo proceso. Asigna automáticamente un PID único mediante un contador de clase. Inicializa todos los atributos con valores por defecto.

is_active():

Retorna True si el proceso está en un estado activo (no NEW ni TERMINATED). Utilizado para determinar si el proceso debe recibir tiempo de CPU o acceder a memoria.

3.2 Clase CPU

3.2.1 Descripción

Simula el procesador del sistema. Mantiene registro del proceso actualmente en ejecución y estadísticas de utilización.

3.2.2 Métodos Detallados

assign(process):

Asigna la CPU a un proceso específico. Realiza las siguientes operaciones:

1. Incrementa el contador de context switches si había otro proceso ejecutándose
2. Cambia el estado del proceso a RUNNING
3. Registra el tiempo de inicio si es la primera vez que se ejecuta

execute_cycle():

Ejecuta un ciclo de CPU. Si hay un proceso asignado:

4. Incrementa busy_time
5. Decrementa remaining_cpu del proceso
6. Retorna True indicando trabajo realizado

Si no hay proceso, incrementa idle_time y retorna False.

get_utilization():

Calcula el porcentaje de utilización de la CPU como: (busy_time / total_time) * 100. Un valor cercano a 100% indica alta utilización, mientras que valores bajos sugieren que la CPU pasa mucho tiempo ociosa.

3.3 Clase Semaphore

3.3.1 Implementación de Operaciones P y V

Los semáforos implementan el mecanismo clásico de sincronización de Dijkstra. Cada semáforo mantiene un valor entero y una cola de procesos en espera.

wait(process) - Operación P:

```
def wait(self, process):
    self.value -= 1
    if self.value < 0:
        process.state = ProcessState.BLOCKED
        process.blocked_on = self.name
        self.waiting_queue.append(process)
        return False  # Bloqueado
    return True  # Puede continuar
```

signal(process) - Operación V:

```
def signal(self, process=None):
    self.value += 1
    if self.waiting_queue:
        blocked = self.waiting_queue.popleft()
        blocked.state = ProcessState.READY
        blocked.blocked_on = None
        return blocked
```

3.4 Clase SwapManager

3.4.1 Gestión del Área de Swap

El SwapManager simula el área de intercambio en disco. Utiliza una lista de tamaño fijo donde cada posición puede contener una página de un proceso.

allocate(pid, page_num):

Busca un frame libre en el swap y lo asigna a la página especificada. Proceso:

7. Recorre frames buscando posición None (libre)
8. Asigna tupla (pid, page_num) a esa posición
9. Incrementa contador used
10. Incrementa contador swaps_in

free_process(pid):

Libera todas las páginas de un proceso del swap. Útil cuando un proceso termina o se elimina del sistema. Recorre todos los frames y elimina aquellos que pertenezcan al PID especificado.

4. Algoritmos Implementados

4.1 Planificación FCFS (First Come, First Served)

4.1.1 Descripción del Algoritmo

FCFS es el algoritmo de planificación más simple. Los procesos se ejecutan en el orden en que llegan a la cola de listos. No hay apropiación (preemption): una vez que un proceso obtiene la CPU, la mantiene hasta completar su ejecución o bloquearse.

4.1.2 Ventajas y Desventajas

Ventajas:

- Simplicidad de implementación
- Bajo overhead (sin cálculos complejos)
- Predecible y justo en términos de orden de llegada

Desventajas:

- Efecto convoy: procesos cortos esperan a procesos largos
- Tiempo de espera promedio puede ser alto
- No considera prioridades ni tipos de procesos

4.1.3 Pseudocódigo Detallado

```
FUNCIÓN schedule_cpu():
    SI cpu.is_free():
        proceso = scheduler.get_next_process()
        SI proceso != None:
            cpu.assign(proceso)
            LOG('Proceso P' + proceso.pid + ' asignado a CPU')

    SI cpu.current_process != None:
        cpu.execute_cycle()
        SI cpu.current_process.remaining_cpu == 0:
            proceso_terminado = cpu.current_process
            cpu.release()
            terminate_process(proceso_terminado)
```

4.1.4 Análisis de Complejidad

- Tiempo: O(1) por ciclo de ejecución
- Espacio: O(n) donde n es el número de procesos en ready_queue
- Operaciones principales son accesos directos a deque (popleft es O(1))

4.2 Algoritmo de Reemplazo de Páginas LRU

4.2.1 Concepto LRU (Least Recently Used)

LRU reemplaza la página que no ha sido utilizada durante el período más largo de tiempo. Se basa en el principio de localidad: las páginas recientemente usadas probablemente se usarán pronto, mientras que las no usadas pueden descartarse.

4.2.2 Implementación Detallada

El sistema mantiene un timestamp (last_access_time) para cada página en RAM. Cuando se necesita reemplazar una página:

```
def page_replacement(self):
    oldest_time = infinito
    victim_frame = None

    PARA cada frame_idx en range(total_frames):
        SI ram[frame_idx] != None:
            pid, page_num = frame_map[frame_idx]
            entry = page_tables[pid].get(page_num)

            SI entry.last_access < oldest_time:
                oldest_time = entry.last_access
                victim_frame = frame_idx

    RETORNAR victim_frame
```

4.2.3 Ejemplo de Ejecución

Supongamos RAM con 3 frames y secuencia de accesos a páginas: 1, 2, 3, 1, 4

Acceso	RAM (frames)	Page Fault?	Acción
Pág 1	[1, -, -]	Sí	Cargar en frame 0

Acceso	RAM (frames)	Page Fault?	Acción
Pág 2	[1, 2, -]	Sí	Cargar en frame 1
Pág 3	[1, 2, 3]	Sí	Cargar en frame 2
Pág 1	[1, 2, 3]	No	Hit! Actualizar timestamp
Pág 4	[1, 4, 3]	Sí	Reemplazar pág 2 (LRU)

4.3 Detección de Deadlock

4.3.1 Concepto de Deadlock

Un deadlock (interbloqueo) ocurre cuando dos o más procesos están esperando indefinidamente por recursos que otros procesos del conjunto están reteniendo. Condiciones necesarias: exclusión mutua, hold and wait, no preemption, espera circular.

4.3.2 Algoritmo de Detección

El sistema construye un grafo de espera donde:

- Nodos = Procesos bloqueados
- Aristas = P1 → P2 si P1 espera recurso que P2 posee

```
def detect_deadlock():
    deadlocked_processes = []

    PARA cada proceso P en blocked_processes:
        SI P.blocked_on != None:
            recurso = P.blocked_on
            semaforo = semaphores[recurso]

            # Buscar quien tiene el recurso
            PARA cada proceso_holder:
                SI proceso_holder tiene el recurso:
                    SI proceso_holder en blocked_processes:
                        # Posible ciclo detectado
                        deadlocked_processes.append(P)

    RETORNAR deadlocked_processes
```

5. Configuración del Sistema

5.1 Archivo config.ini - Formato y Secciones

5.1.1 Sección [MEMORY]

Controla todos los parámetros relacionados con la gestión de memoria:

```
[MEMORY]
ram_size_kb = 2048    # Total RAM en KB (recomendado: 1024-8192)
page_size_kb = 4      # Tamaño de página (debe ser potencia de 2)
swap_size_kb = 4096   # Tamaño del swap (típicamente 2-4x RAM)
```

Consideraciones de configuración:

- ram_size_kb debe ser múltiplo de page_size_kb
- page_size_kb típicamente 4KB (igual que sistemas reales)
- swap_size_kb = 2-4 veces ram_size_kb es recomendado

5.1.2 Sección [SIMULATION]

```
[SIMULATION]
max_processes = 20          # Número máximo de procesos a generar
process_size_min = 256      # Tamaño mínimo de proceso en KB
process_size_max = 1024     # Tamaño máximo de proceso en KB
process_lifetime_min = 10   # Vida mínima en ciclos
process_lifetime_max = 50   # Vida máxima en ciclos
process_arrival_min = 1     # Ciclos mínimos entre llegadas
process_arrival_max = 3     # Ciclos máximos entre llegadas
```

5.1.3 Sección [LOGS]

```
[LOGS]
log_file = tests/logs/simulation.log
log_level = INFO    # DEBUG, INFO, WARNING, ERROR, CRITICAL
```

5.2 Guía de Tuning de Rendimiento

Parámetro	Valor Bajo	Valor Alto	Efecto
ram_size_kb	512-1024	4096-8192	↑ RAM = ↓ page faults, ↑ procesos
page_size_kb	2-4	16-64	↑ tamaño = ↓ páginas, ↑ fragmentación
max_processes	5-10	50-100	↑ procesos = ↑ competencia, ↑ swapping
process_arrival	min=1, max=2	min=5, max=10	↓ intervalo = ↑ carga, ↑ congestión

Conclusión

El sistema simula componentes clave de un sistema operativo, como la planificación de CPU (usando FCFS), la gestión de memoria con paginación por demanda y memoria virtual, la sincronización con semáforos, y la detección de deadlocks.

La arquitectura es modular y de capas, utilizando la Programación Orientada a Objetos (POO) en Python 3.8+. Los componentes clave son main.py (Control y UI), clases.py (entidades: Process, CPU, Semaphore, SwapManager) y gestor_memoria.py (lógica centralizada con ProcessManager).

Se implementan patrones de diseño como el Manager Pattern (en ProcessManager) y el State Pattern (para los estados del proceso). El sistema soporta la estrategia de planificación FCFS y utiliza el algoritmo LRU (Least Recently Used) para el reemplazo de páginas. La configuración del sistema se realiza mediante el archivo config.ini.