Universidad Metropolitana

Asignatura: Sistemas Operativos
Profesor: Isaac Olivares
Sección 1

SIMULADOR VIRTUAL DE SISTEMA DE ARCHIVOS Y GESTIÓN DE RECURSOS

30.142.725 - Ricardo Baeta
28.155.240 - Aida Cárdenas
30.142.817 - René Chamorro

Caracas, 25 de Octubre de 2025

Índice

1. INTRODUCCIÓN

1.1 Contexto del Proyecto

El estudio de los Sistemas Operativos (SO) implica la comprensión de conceptos abstractos relacionados con la gestión de recursos de hardware y software. La administración del almacenamiento secundario (sistema de archivos), la planificación de peticiones a disco y la gestión de procesos son pilares fundamentales en la arquitectura de un computador. Sin embargo, su naturaleza interna y "oculta" al usuario final dificulta su visualización y comprensión práctica.

Este proyecto, titulado "Simulador Virtual de Sistema de Archivos con Gestión de Permisos y Asignación de Bloques", surge como una solución pedagógica y técnica para materializar estos conceptos. Se ha desarrollado una aplicación de escritorio en Java que emula el comportamiento de un núcleo de SO, permitiendo visualizar en tiempo real cómo se crean archivos, cómo se asignan bloques físicos en el disco, y cómo se planifican las operaciones de entrada/salida (I/O).

1.2 Objetivos

Objetivo General: Diseñar e implementar un simulador funcional de un Sistema Operativo centrado en la gestión de archivos, asignación de memoria secundaria y planificación de disco, utilizando una arquitectura modular y estructuras de datos propias.

Objetivos Específicos:

1. Implementar un Sistema de Archivos Jerárquico con asignación encadenada de bloques.
2. Simular un disco físico de 200 bloques con visualización de estado (ocupado/libre).
3. Desarrollar y comparar seis algoritmos de planificación de disco (FIFO, SSTF, SCAN, C-SCAN, LOOK, C-LOOK).
4. Gestionar el ciclo de vida de los procesos y sus solicitudes de I/O.
5. Implementar un Buffer de Memoria (Caché) con políticas de reemplazo para optimizar el acceso a disco.
6. Aplicar el patrón de arquitectura MVC (Modelo-Vista-Controlador) para desacoplar la lógica de negocio de la interfaz gráfica.

1.3 Alcance

El simulador abarca desde la creación lógica de archivos y directorios hasta la simulación física del movimiento del cabezal del disco. Incluye un sistema de permisos de usuario, persistencia de datos y una restricción técnica clave: el uso exclusivo de

estructuras de datos dinámicas personalizadas (`ListaEnlazada`, `Cola`, `Pila`) en lugar de las colecciones estándar de Java (`java.util`), para demostrar el dominio sobre la gestión de memoria y punteros.

2. MARCO TEÓRICO

2.1 Sistemas de Archivos y Asignación de Bloques

Un sistema de archivos es el componente del SO encargado de organizar y recuperar datos. Para este proyecto se seleccionó el método de **Asignación Encadenada (Linked Allocation)**.

- **Justificación:** A diferencia de la asignación contigua, la asignación encadenada resuelve el problema de la fragmentación externa. Cada archivo es una lista enlazada de bloques de disco; los bloques pueden estar dispersos en cualquier lugar del disco. El directorio solo necesita almacenar un puntero al primer bloque y (opcionalmente) al último. Esto permite que los archivos crezcan dinámicamente sin necesidad de compactación de disco, lo cual es ideal para un simulador educativo.

2.2 Algoritmos de Planificación de Disco

Dado que el acceso a disco es lento, el orden en que se atienden las solicitudes impacta el rendimiento. Se implementaron:

1. **FIFO (First-In, First-Out):** Atiende en orden de llegada. Justo pero ineficiente.
2. **SSTF (Shortest Seek Time First):** Selecciona la petición más cercana al cabezal actual. Minimiza el movimiento pero puede causar inanición (starvation).
3. **SCAN (Ascensor):** El cabezal se mueve de un extremo a otro atendiendo peticiones.
4. **C-SCAN (Circular SCAN):** Similar a SCAN, pero al llegar al final regresa al principio sin atender peticiones en el retorno. Ofrece un tiempo de espera más uniforme.
5. **LOOK / C-LOOK:** Variantes de SCAN/C-SCAN donde el cabezal solo llega hasta la última petición en esa dirección, sin necesidad de llegar al extremo físico del disco.

2.3 Buffering y Caché

El simulador implementa un **Buffer Cache** en memoria principal. Antes de acceder al disco lento, el sistema verifica si el bloque está en el buffer.

- **Políticas de Reemplazo:** Cuando el buffer (20 bloques) se llena, se debe decidir qué bloque expulsar:
    - **LRU (Least Recently Used):** Elimina el bloque menos usado recientemente.
    - **LFU (Least Frequently Used):** Elimina el bloque con menos accesos históricos.

    - **FIFO:** Elimina el bloque que entró primero.

2.4 Gestión de Procesos

Se utiliza el modelo de estados de 5 etapas:

- **Nuevo:** Proceso creado.
- **Listo:** Esperando CPU.
- **Ejecutando:** Usando CPU.
- **Bloqueado:** Esperando I/O (lectura/escritura en nuestro simulador).
- **Terminado:** Finalización.

3. DISEÑO E IMPLEMENTACIÓN

El sistema sigue estrictamente el patrón **MVC (Model-View-Controller)**. La lógica está separada de la representación visual, coordinada por gestores específicos. El proyecto consta de 48 clases organizadas en paquetes.

3.1 Paquete Controller (Lógica de Negocio)

`ControladorPrincipal`

- **Responsabilidad:** Punto central de orquestación. Inicializa el sistema y conecta las vistas con los gestores.
- **Interacciones:** Instancia a todos los demás `Gestor*` y a la `VentanaPrincipal`.
- **Decisión de Diseño:** Se usa como *Singleton* implícito para garantizar una única instancia de coordinación global.

`GestorArchivos`

- **Responsabilidad:** Lógica CRUD del sistema de archivos virtual.
- **Métodos Principales:**
    - `crearArchivo(String nombre, int tamaño, Usuario propietario)`: Calcula bloques necesarios y solicita asignación.
    - `eliminarNodo(NodoArchivo nodo)`: Maneja la recursividad para eliminar directorios y liberar bloques asociados.
- **Interacciones:** Solicita bloques libres al `GestorDisco` y actualiza el árbol en `SistemaArchivos`.

`GestorDisco`

- **Responsabilidad:** Administra el array de 200 bloques y la tabla de asignación.
- **Métodos Principales:**

    - `asignarBloques(int cantidad)`: Retorna una `ListaEnlazada<Integer>` con los índices de bloques libres.
    - `liberarBloques(ListaEnlazada<Integer> indices)`: Marca los bloques como libres en el mapa de bits o array de estado.
- **Decisión de Diseño:** Mantiene un mapa de estado en memoria para realizar búsquedas O(1) de bloques libres.

`GestorPlanificacion`

- **Responsabilidad:** Ejecuta los algoritmos de planificación de disco.
- **Métodos Principales:**
    - `ejecutarAlgoritmo(TipoAlgoritmo tipo, int cabezal, Cola<Peticion> peticiones)`: Recibe la cola de I/O y la reordena según la estrategia seleccionada.
- **Estructuras:** Utiliza una estrategia *Strategy Pattern* (implícita) donde cada algoritmo es una implementación intercambiable.

`GestorBuffer`

- **Responsabilidad:** Simula la memoria caché intermedia.
- **Métodos Principales:**
    - `solicitarBloque(int idBloque)`: Retorna el dato si es un HIT, o null si es un MISS.
    - `agregarAlBuffer(Bloque b)`: Si está lleno, invoca la política de reemplazo (LRU/LFU) para hacer espacio.
- **Decisión de Diseño:** Se implementa un HashMap auxiliar para búsquedas rápidas dentro del buffer, complementando la lista enlazada que mantiene el orden temporal.

3.2 Paquete Model (Entidades)

`SistemaArchivos`

- **Responsabilidad:** Clase raíz que serializa el estado completo del sistema.
- **Contenido:** Referencias al directorio raíz (`/`), lista de usuarios y configuración del disco.

`Archivo` (Extiende `NodoSistema`)

- **Responsabilidad:** Representa un inodo de archivo.
- **Atributos Clave:**
    - `ListaEnlazada<Integer> bloquesAsignados`: La implementación de la asignación encadenada. Contiene los IDs de los bloques físicos.

- `Permisos permisos`: Objeto que define r/w/x.

`Disco`

- **Responsabilidad:** Representación física.
- **Estructuras:** Array de objetos `Bloque[200]`.
- **Atributos:** Posición actual del cabezal (int).

`Proceso`

- **Responsabilidad:** Entidad activa que genera carga al sistema.
- **Métodos:** `generarSolicitudIO()`: Crea aleatoriamente una petición de lectura/escritura a un archivo existente.

3.3 Paquete View (Interfaz)

`VentanaPrincipal`

- **Responsabilidad:** Contenedor JFrame principal.
- **Diseño:** Layout `BorderLayout` y `GridLayout` dividiendo la pantalla en 4 cuadrantes funcionales (Árbol, Disco, Procesos, Simulación).

`PanelDisco`

- **Responsabilidad:** Visualización del estado del disco.
- **Implementación:** Matriz de 20x10 celdas. Se usa `Graphics2D` para pintar bloques: Verde (Libre), Rojo (Ocupado), Azul (En Buffer).
- **Patrón:** Actúa como *Observer* del `GestorDisco`. Cuando un bloque cambia de estado, se repinta solo la región afectada.

4. FLUJOS DE OPERACIÓN

4.1 Flujo de Creación de Archivo

1. **Solicitud:** El usuario selecciona un directorio en el `PanelArbolArchivos` y pulsa "Nuevo Archivo".
2. **Validación:** `GestorArchivos` verifica permisos de escritura en el directorio padre y nombre único.
3. **Cálculo:** Se calcula cuántos bloques se necesitan (Ej: Archivo de 4KB en bloques de 1KB = 4 bloques).
4. **Asignación:** `GestorDisco` busca 4 bloques libres. Si hay espacio, los marca como ocupados y retorna sus IDs (Ej: 10, 45, 12, 190).
5. **Enlazado:** Se crea el objeto `Archivo`. Su lista `bloquesAsignados` se llena: 10 -> 45

-> 12 -> 190.

6. **UI:** Se actualiza el `JTree` y el `PanelDisco` pinta los nuevos bloques en rojo.

4.2 Flujo de Procesamiento de Solicitudes I/O

1. **Generación:** Un `Proceso` en estado *Ejecutando* lanza una interrupción de I/O solicitando leer el bloque #45.
2. **Buffer Check:** El `GestorBuffer` verifica si el bloque #45 está en memoria.
    - **Caso HIT:** El dato se retorna inmediatamente. Tiempo simulado: 1ms.
    - **Caso MISS:** La petición se encola en la `Cola<SolicitudIO>` del disco. El proceso pasa a estado *Bloqueado*.
3. **Planificación:** El `GestorPlanificacion` observa la cola. Si usa SCAN, ordena las peticiones basándose en la dirección actual del cabezal.
4. **Movimiento:** El `SimuladorIO` mueve el cabezal visualmente hacia el bloque #45.
5. **Carga:** El bloque se lee y se coloca en el Buffer (posiblemente expulsando otro).
6. **Desbloqueo:** El proceso pasa a *Listo* y luego a *Ejecutando*.

5. INTERFAZ GRÁFICA

La interfaz se ha diseñado priorizando la visualización de datos.

- **Cuadrante Superior Izquierdo (Sistema de Archivos):** `JTree` personalizado. Los iconos distinguen entre carpetas y archivos. Menú contextual con clic derecho para operaciones CRUD.
- **Cuadrante Superior Derecho (Mapa de Disco):** Grid de celdas. Al pasar el mouse sobre un bloque ocupado (Tooltip), muestra a qué archivo pertenece, demostrando la asignación encadenada visualmente.
- **Cuadrante Inferior Izquierdo (Tabla de Asignación):** Un `JTable` que detalla: Nombre Archivo | Tamaño | Bloque Inicio | Bloque Final | Lista de Bloques. Es la representación textual de la FAT (File Allocation Table).
- **Cuadrante Inferior Derecho (Simulador y Procesos):** Controles de reproducción (Play/Pause) para la simulación de I/O. Gráficos de barras simples para estadísticas de Buffer (Hits vs Misses).

6. ESTRUCTURAS DE DATOS PROPIAS

Siguiendo los requerimientos del proyecto, se prescindió de `java.util` (ArrayList, LinkedList, etc.) para la lógica central, implementando estructuras genéricas en el paquete `util`.

6.1 `ListaEnlazada<T>`

Implementación de una lista simplemente enlazada.

- **Uso:** Fundamental para la **Asignación Encadenada** de bloques en la clase `Archivo`. Cada nodo apunta al siguiente ID de bloque físico.
- **Ventajas:** Crecimiento dinámico O(1) al agregar al final (manteniendo un puntero `tail`).
- **Métodos:** `add(T data)`, `remove(T data)`, `get(int index)`, `contains(T data)`.

6.2 `Cola<T>`

Implementación basada en nodos (FIFO).

- **Uso:** Gestión de la cola de procesos en estado *Listo* y la cola de solicitudes de I/O crudas antes de ser reordenadas por el planificador.
- **Métodos:** `enqueue(T data)`, `dequeue()`, `peek()`.

6.3 Diferencias con Java Collections

Al implementar nuestras propias estructuras, perdimos la optimización extrema de la JVM, pero ganamos control total sobre los nodos. Por ejemplo, nuestra `ListaEnlazada` expone métodos internos para facilitar la depuración visual del encadenamiento de bloques, algo que `java.util.LinkedList` encapsula completamente.

7. PRUEBAS Y VALIDACIÓN

Se realizaron pruebas unitarias e integrales (manuales) para validar el comportamiento:

1. **Prueba de Fragmentación:**
    - *Escenario:* Llenar el disco al 80%, borrar archivos alternos y crear un archivo grande.
    - *Resultado:* El sistema asignó correctamente bloques dispersos (encadenados), validando que la asignación encadenada soluciona la fragmentación externa (no se requirió compactación).
2. **Prueba de Planificación (SCAN vs FIFO):**
    - *Escenario:* Cola de peticiones: 10, 190, 20, 180. Cabezal en 50.
    - *Resultado FIFO:* Recorrido errático (total 340 saltos).
    - *Resultado SCAN:* Ordenado (50->20->10->180->190). Movimiento fluido del cabezal validado visualmente.
3. **Prueba de Buffer:**
    - *Escenario:* Lectura repetida del mismo archivo pequeño.

- *Resultado:* Primera lectura = MISS (lenta). Siguientes lecturas = HIT (instantáneas). Tasa de aciertos aumentó conforme a lo esperado.

8. CONCLUSIONES

El desarrollo del "Simulador Virtual de Sistema de Archivos" ha permitido consolidar los conocimientos teóricos de la materia.

1. **Asignación de Archivos:** Se demostró prácticamente que la **asignación encadenada** es excelente para gestionar el espacio en disco sin fragmentación externa, aunque penaliza el acceso directo aleatorio (solucionado parcialmente con el uso de la tabla de asignación en memoria).
2. **Planificación de Disco:** La implementación visual de algoritmos como **SCAN y C-LOOK** evidenció cómo el reordenamiento de peticiones reduce drásticamente el tiempo de búsqueda (seek time) y el desgaste mecánico simulado del disco.
3. **Complejidad del Kernel:** La coordinación entre el gestor de procesos, el gestor de memoria (buffer) y el gestor de I/O reveló la complejidad de sincronización necesaria en un SO real para evitar condiciones de carrera y asegurar la integridad de los datos.

Como mejora futura, se propone la implementación de **sistemas de archivos con indexación (i-nodes)** para comparar el rendimiento contra la asignación encadenada en archivos de gran tamaño.

9. BIBLIOGRAFÍA

1. **Silberschatz,** A., Galvin, P. **B., & Gagne, G.** (2018). *Operating System Concepts* (10th ed.). Wiley. (Capítulos: File-System Implementation, Mass-Storage Structure).
2. **Tanenbaum, A. S., & Bos, H.** (2014). *Modern Operating Systems* (4th ed.). Pearson.
3. **Oracle Documentation.** *Java Platform, Standard Edition & Java Swing Components*.