Gestión de Archivos y Memoria en Sistemas Operativos

Laboratorio de Sistemas Operativos
Universidad Distrital

7 de octubre de 2025

Resumen

Este informe presenta un análisis teórico y práctico de los principales esquemas de gestión de memoria en sistemas operativos, apoyado en la simulación implementada en el código `scriptLaboratorio.js`. Se explican los métodos de particiones fijas, variables, asignación dinámica y compactación, destacando sus ventajas, desventajas y aplicaciones. Asimismo, se incluyen diagramas y ejemplos para ilustrar el proceso de asignación y liberación de memoria.

Índice

1. Introducción

La memoria principal es un recurso limitado y esencial en la ejecución de programas dentro de un sistema operativo. Su correcta administración garantiza eficiencia en el uso de recursos, evitando desperdicio de espacio y mejorando la multitarea. El presente documento describe los distintos métodos de gestión de memoria simulados en el código `scriptLaboratorio.js`, los cuales reproducen situaciones reales de asignación de bloques de memoria a procesos.

2. Marco Teórico

2.1. Gestión de Memoria

La gestión de memoria es la disciplina encargada de controlar el uso de la memoria principal. Entre sus objetivos se encuentran:

- Proveer espacio a los procesos en ejecución.
- Evitar colisiones entre procesos.
- Reducir la fragmentación.
- Optimizar el tiempo de acceso.

2.2. Tipos de Fragmentación

- **Fragmentación interna:** ocurre cuando la memoria asignada a un proceso excede lo que realmente necesita, desperdiciando espacio dentro de una partición.
- **Fragmentación externa:** se presenta cuando existen múltiples bloques libres en memoria, pero están dispersos y no son contiguos.

2.3. Algoritmos de Asignación

Existen diferentes estrategias para decidir en qué bloque libre se colocará un proceso:

- **First Fit:** selecciona el primer bloque libre que cumpla con el tamaño requerido.
- **Best Fit:** busca el bloque libre más pequeño que sea suficiente, optimizando espacio.
- **Worst Fit:** selecciona el bloque libre más grande, buscando reducir el número de bloques pequeños dispersos.

2.4. Tipos de memoria

En el código, la memoria simulada tiene un tamaño de 16 MiB y se administra mediante diferentes técnicas de particionamiento.

Particiones Fijas

La memoria se divide en bloques de igual tamaño. En la simulación se crean 8 particiones de 2 MiB. Esto produce **fragmentación interna** cuando los programas no utilizan todo el bloque asignado.

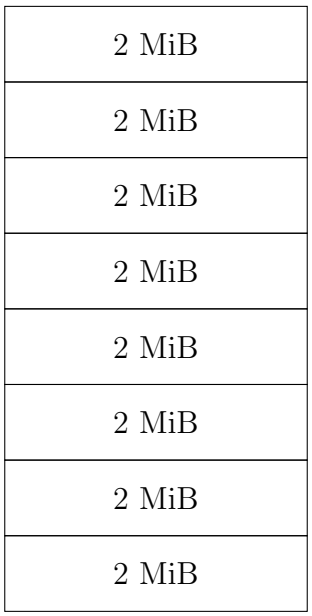

Figura 1. Memoria dividida en particiones fijas.

Particiones Variables

La memoria se divide en bloques de tamaños distintos, definidos de antemano (2, 4, 6 y 4 MiB). Esto reduce la fragmentación interna, aunque genera **fragmentación externa**.

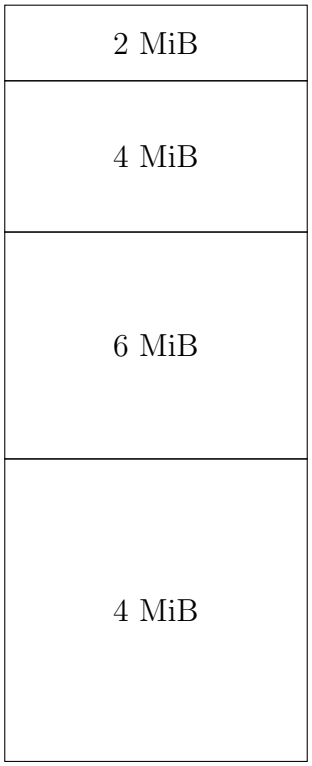

Figura 2. Particiones variables.

Asignación Dinámica

Se inicia con un único bloque de memoria libre y se divide conforme los programas se cargan. La simulación permite seleccionar entre **First Fit**, **Best Fit** y **Worst Fit**.

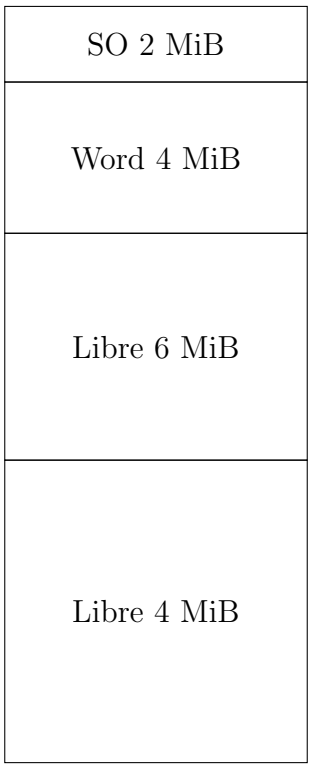

Figura 3. Asignación dinámica con bloques ocupados y libres.

Dinámica con Compactación

Cuando la memoria presenta bloques libres dispersos, la compactación mueve los procesos para unificar los espacios libres en un bloque único.

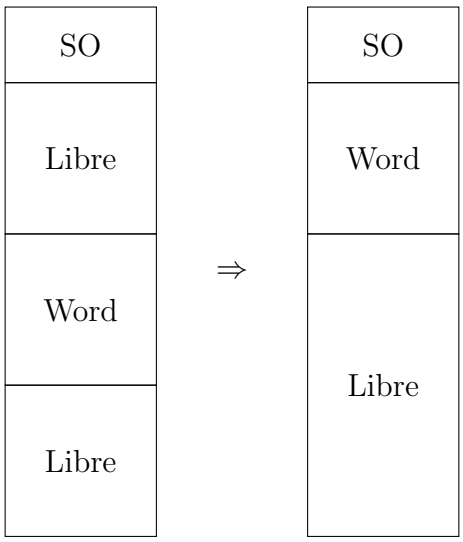

Figura 4. Proceso de compactación.

Dinámica sin Compactación

Cuando se liberan bloques de memoria, los huecos permanecen dispersos y no se reorganizan. Esto provoca que, aunque exista suficiente memoria libre en total, un programa grande no pueda cargarse debido a la **fragmentación externa**.

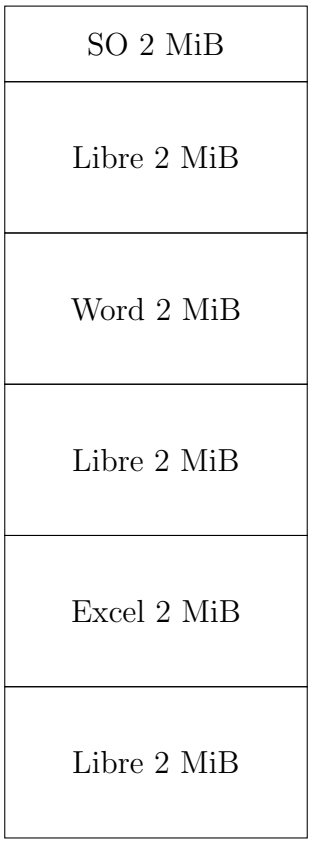

Figura X. Memoria con fragmentación externa sin compactación.

3. Análisis y Diseño

3.1. Objetivo

El propósito del simulador es representar distintos enfoques de gestión de memoria en sistemas operativos. Se busca que el usuario observe en forma visual cómo se comporta la memoria bajo diferentes técnicas de particionamiento, con y sin compactación, y con la aplicación de varios algoritmos de asignación.

3.2. Metodología de Desarrollo

Para la construcción del simulador se utilizó una metodología iterativa e incremental:

1. **Análisis del problema:** Se estudiaron los esquemas de administración de memoria (particiones fijas, variables, dinámicas y compactación), identificando los aspectos relevantes a implementar.

2. **Diseño modular:** Se definieron clases independientes para la memoria, los algoritmos de asignación y la interfaz. Esto permitió mantener un código organizado y fácil de mantener.

3. **Implementación incremental:** Se comenzó con la memoria estática (particiones fijas), luego se añadieron las particiones variables, y finalmente la asignación dinámica con y sin compactación.

4. **Pruebas continuas:** En cada iteración se realizaron pruebas de carga y liberación de programas para validar el funcionamiento de los algoritmos y la representación gráfica.

3.3. Requerimientos Funcionales

- Permitir instalar y liberar programas en memoria.
- Ofrecer la selección del algoritmo de asignación (First Fit, Best Fit, Worst Fit, Next Fit).
- Mostrar en pantalla el estado actual de la memoria con bloques ocupados y libres.
- Representar diferentes configuraciones de memoria:
  - Particiones fijas.
  - Particiones variables.
  - Asignación dinámica sin compactación.
  - Asignación dinámica con compactación.

3.4. Requerimientos No Funcionales

- Interfaz visual clara y entendible para los usuarios.
- Código fuente modular y reutilizable.
- Eficiencia en el manejo de memoria y en el tiempo de respuesta.
- Compatibilidad con navegadores modernos, ya que la simulación se implementó en **JavaScript**.

3.5. Diseño del Sistema

El diseño se basa en tres módulos principales:

- **Memoria:** Modela el bloque de RAM disponible y sus particiones.
- **Algoritmos de asignación:** Implementa First Fit, Best Fit, Worst Fit y Next Fit.
- **Interfaz gráfica:** Permite la interacción con el usuario y la visualización de la memoria.

A continuación, se presentan los diagramas de diseño que muestran la estructura general del simulador:

4. Implementación

4.1. Diagramas UML

A continuación, se presentan los diagramas UML que describen los principales aspectos estructurales y funcionales del sistema de gestión de memoria desarrollado en el laboratorio.

Diagrama de Casos de Uso

El siguiente diagrama muestra las interacciones entre el actor principal (usuario) y el sistema, destacando las funcionalidades disponibles como la carga, liberación y visualización de programas en memoria.

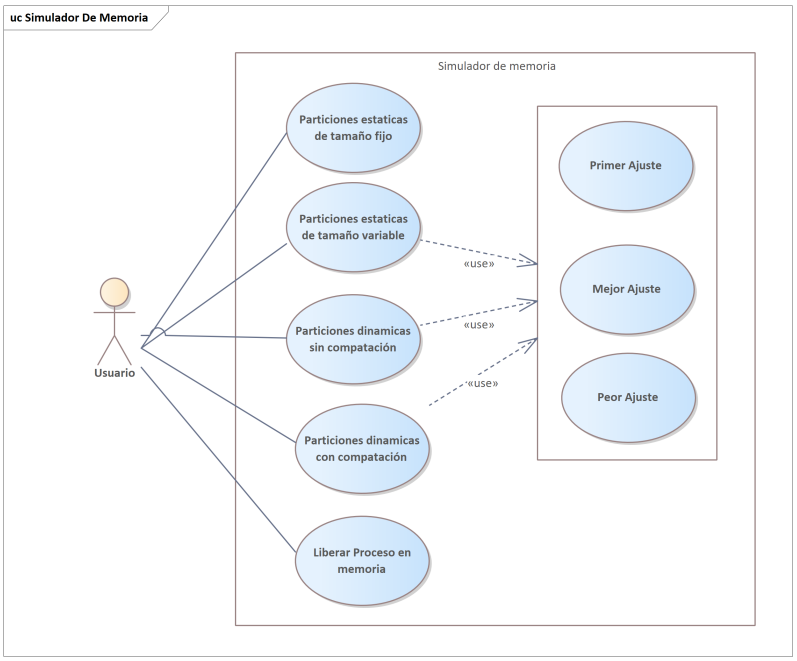

Figura 1: Diagrama de casos de uso del sistema de simulación de gestión de memoria.

Diagrama de Clases

En el diagrama de clases se representa la estructura lógica del sistema, mostrando las relaciones entre las clases principales como `Memoria`, `MemoriaDinamica`, y los módulos que implementan los algoritmos de asignación.

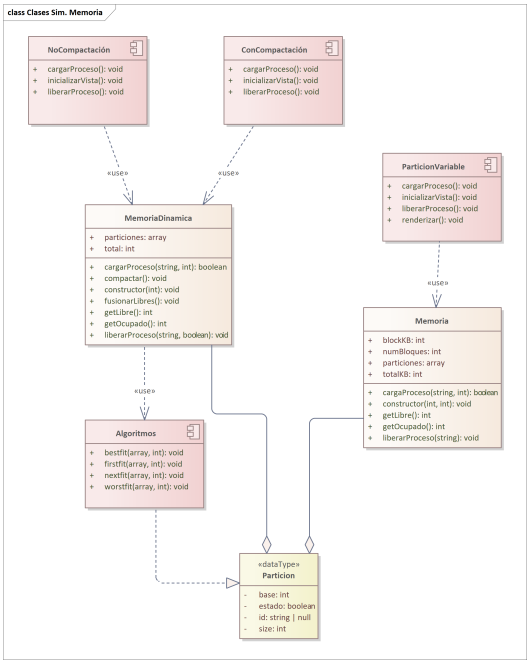

Figura 2: Diagrama de clases del sistema de gestión de memoria.

Diagrama de Secuencia

El diagrama de secuencia ilustra el flujo dinámico del sistema durante la ejecución del algoritmo de asignación de memoria, mostrando la interacción entre el usuario, el módulo de interfaz, la clase de memoria y las funciones de asignación.

4.2. Entorno de Desarrollo

La simulación fue implementada en JavaScript, utilizando módulos para separar la lógica de memoria, los algoritmos de asignación y la interfaz. El proyecto se ejecuta en cualquier navegador moderno y no requiere instalación adicional. Se empleó HTML para la estructura de la página web, y se le añadieron estilos con CSS, para que sea agradable visualmente.

4.3. Estructura del código

El proyecto sigue una arquitectura modular que separa la interfaz principal, la lógica de simulación y los estilos. El archivo `index.html` actúa como núcleo del sistema, car-

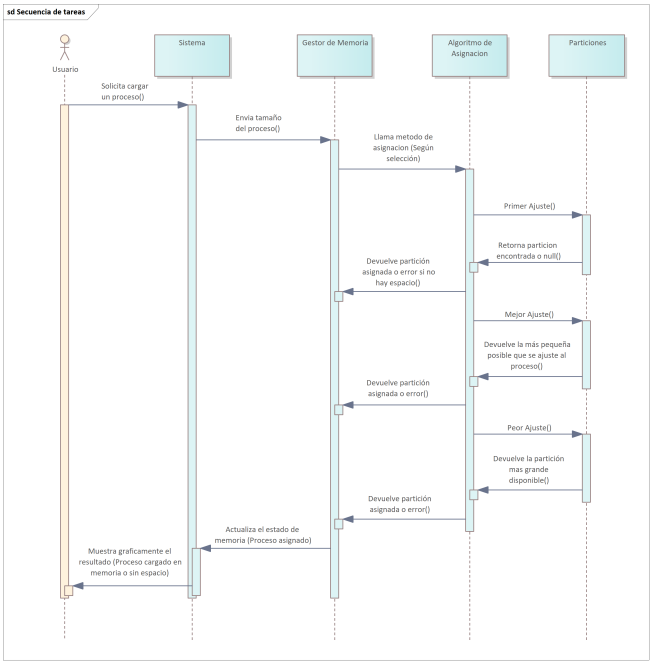

Figura 3: Diagrama de secuencia para el proceso de asignación de memoria.

gando dinámicamente las vistas implementadas en los distintos módulos JavaScript. En la figura 4 se muestra la organización general del proyecto.

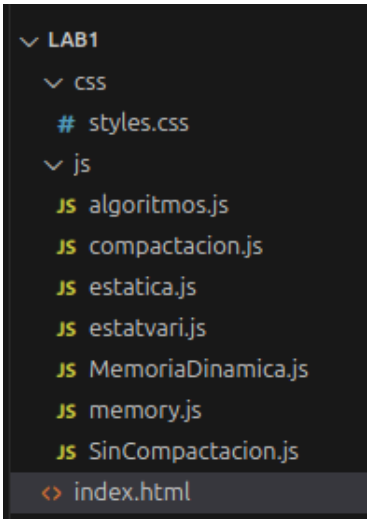

Figura 4: Estructura modular del código fuente.

- `index.html`: Archivo principal de la aplicación. Define la estructura base de la interfaz con un encabezado, un menú de navegación y un contenedor principal denominado `#contenedor`. Mediante la función `cargarVista()`, se importa de forma dinámica el módulo correspondiente (`estatica.js`, `estatvari.js`, `SinCompactacion.js` o `compactacion.js`) según el botón que el usuario seleccione. De esta forma, el contenido del contenedor se actualiza sin recargar la página, permitiendo una navegación fluida entre las diferentes simulaciones.

- `css/styles.css`: Contiene las reglas de estilo que definen la apariencia de la interfaz. Controla colores, bordes, espaciado y la representación visual de la memoria, resaltando los bloques ocupados, libres o compactados.

- `js/memory.js`: Implementa la clase base `Memoria`, que modela la estructura física de la RAM. Incluye atributos como el tamaño total, las particiones y los procesos cargados, así como métodos comunes para reservar, liberar y consultar el estado de los bloques.

- `js/algoritmos.js`: Contiene las funciones que implementan los algoritmos de asignación: **First Fit**, **Best Fit**, **Worst Fit** y **Next Fit**. Estos algoritmos son utilizados por los módulos dinámicos para determinar en qué posición se debe ubicar un nuevo programa.

- `js/estatica.js`: Simula el esquema de **particiones fijas**. Crea ocho bloques de 2 MiB cada uno y gestiona la instalación o liberación de programas dentro de cada partición. También genera dinámicamente la representación visual de la memoria en el contenedor principal.

- `js/estatvari.js`: Gestiona la simulación con **particiones variables**. En este caso, las particiones tienen tamaños diferentes (2, 4, 6 y 4 MiB). Además, coordina la tabla de estado de memoria, mostrando qué programa ocupa cada partición.

- `js/MemoriaDinamica.js`: Define la lógica de la **asignación dinámica de memoria**. Parte de un bloque libre total y lo subdivide conforme se instalan programas, aplicando el algoritmo de asignación seleccionado. Controla tanto la fragmentación como la actualización del espacio libre restante.

- `js/SinCompactacion.js`: Variante de la memoria dinámica que opera **sin aplicar compactación**. Los espacios liberados permanecen como huecos dispersos, permitiendo observar visualmente la **fragmentación externa**.

- `js/compactacion.js`: Implementa la funcionalidad de **compactación**. Al liberarse memoria, reubica los procesos ocupados hacia el inicio para unir los espacios libres en un bloque contiguo, de forma automática.

4.4. Interfaz gráfica del simulador

La interfaz gráfica fue diseñada para ofrecer una experiencia visual e intuitiva al usuario. Desde la vista principal, el encabezado presenta un conjunto de botones que permiten alternar entre los diferentes esquemas de administración de memoria: *particiones fijas, particiones variables, dinámica sin compactación y dinámica con compactación*. Cada uno de estos modos carga una vista independiente dentro del contenedor principal, como se observa en la figura 4.4.

Las vistas manejan una estructura similar, que cambia su contenido según el tipo de memoria seleccionada. Los componentes de la interfaz son:

1. **Tabla de Particiones:** muestra la información de cada bloque de memoria, incluyendo su dirección base (en formato decimal y hexadecimal), tamaño, estado (libre u ocupado) y el identificador del proceso asignado. Además, permite liberar memoria mediante un botón asociado a cada proceso.

2. **Programas Instalados:** contiene una tabla con las características de los programas cargables(como tamaño en disco, código, datos inicializados y no inicializados, entre otros). Desde esta tabla es posible cargar los procesos en memoria mediante el botón "Cargar".

3. **Mapa de Memoria:** representa visualmente el espacio de memoria mediante bloques de colores, distinguiendo los segmentos ocupados y libres. El bloque inicial corresponde al sistema operativo.

Complementariamente, se incluye un cuadro de **estado global** que muestra en tiempo real la cantidad de memoria ocupada y libre, junto con una tabla de **generalidades** donde se detallan parámetros como la RAM total instalada, el tamaño del montículo, la pila y el encabezado ejecutable.

El renderizado de los datos se realiza mediante funciones específicas:

- `renderizar()`: actualiza la visualización del mapa de memoria y las tablas cada vez que se ejecuta una acción de carga o liberación.

- `renderProgramDetails()`: genera dinámicamente la tabla de programas disponibles y asocia los eventos a los botones de carga.

- `renderResumen()`: presenta la información global del sistema de memoria, calculando las equivalencias entre bytes, KiB y MiB.

De esta forma, la interfaz proporciona un entorno interactivo que facilita la comprensión del funcionamiento de la memoria.

Memoria Estática Fija

Al momento de iniciar el simulador, por defecto, se visualiza la interfaz y componentes de la memoria estática fija. En la Figura 5 se observa la interfaz con las secciones mencionadas anteriormente.

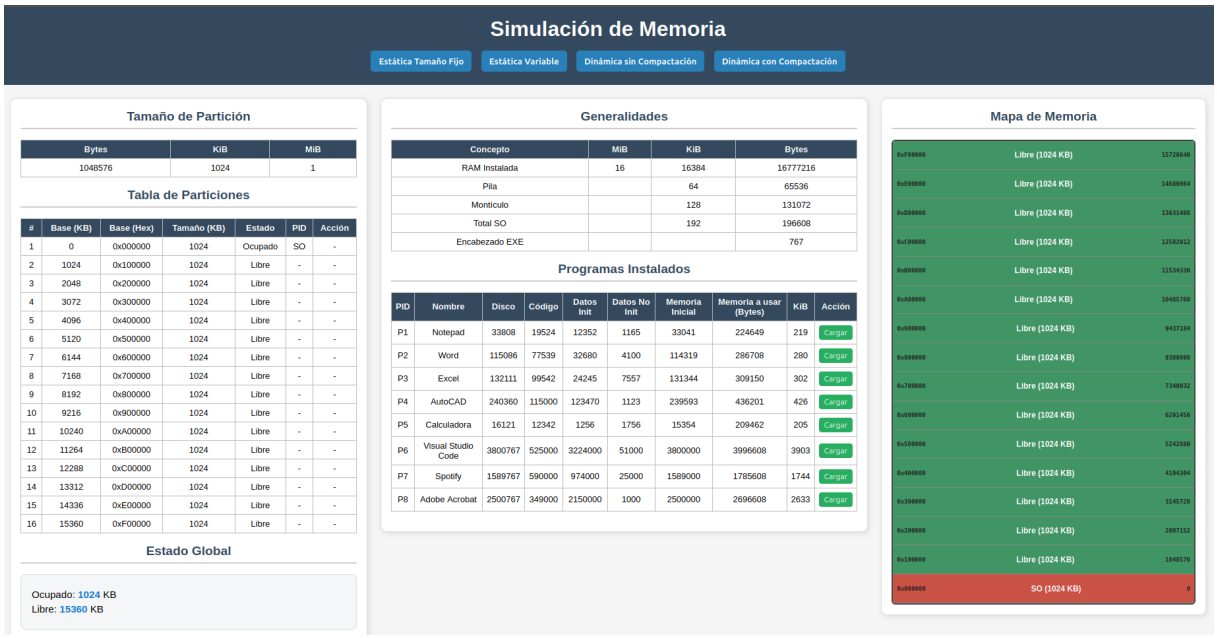

Figura 5: Interfaz estado inicial de la memoria estática con particiones fijas.

La simulación de memoria estática de tamaño fijo implementa un esquema de particiones predefinidas, donde cada bloque posee un tamaño fijo de 1 MiB. Este modelo es útil para comprender la gestión de memoria en sistemas que no permiten la reasignación dinámica de espacio una vez creado el esquema de particiones.

En esta vista, la memoria se inicializa mediante la clase `Memoria`, que asigna la primera partición al sistema operativo y deja las demás disponibles para la carga de programas. La Figura 5 muestra el estado inicial de la memoria, donde únicamente el bloque reservado para el sistema operativo aparece como ocupado, y los demás permanecen libres.

Cada programa disponible en el sistema se encuentra descrito en la tabla de procesos, donde se especifican sus requerimientos de memoria en bytes y kilobytes. Cuando el usuario selecciona la opción **Cargar**, el simulador ejecuta el método `cargarProceso()`, que busca la primera partición libre con espacio suficiente y marca su estado como ocupado. La actualización del mapa de memoria se realiza de manera automática a través de la función `renderizar()`, la cual refresca tanto el diagrama visual como la tabla de particiones.

En la Figura 6 se observa el sistema después de cargar varios procesos, donde los bloques asignados aparecen identificados con el PID correspondiente. El espacio ocupado y libre se recalcula dinámicamente y se refleja en el resumen general.

Finalmente, al ejecutar la acción **Liberar**, el método `liberarProceso()` identifica el bloque asociado al PID y lo marca nuevamente como libre, actualizando la vista general del sistema. Este proceso se ilustra en la Figura 7, donde se puede observar cómo el bloque correspondiente vuelve a quedar disponible tras liberar la memoria utilizada por un proceso.

Este comportamiento permite visualizar de manera clara la relación entre los procesos cargados, la cantidad de memoria disponible y el funcionamiento interno del esquema de

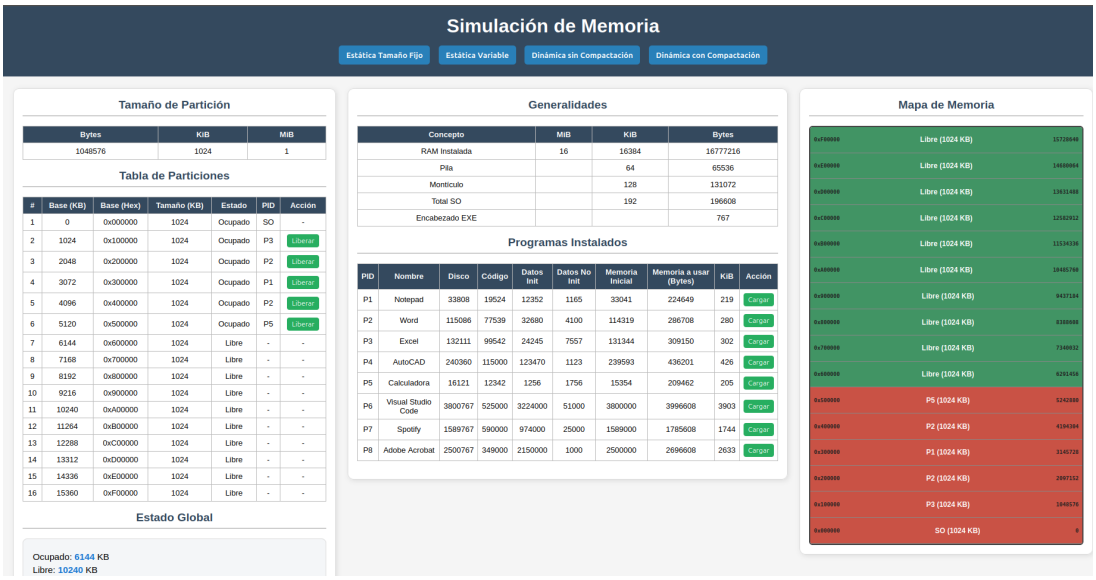

Figura 6: Memoria estática tras la carga de varios procesos.

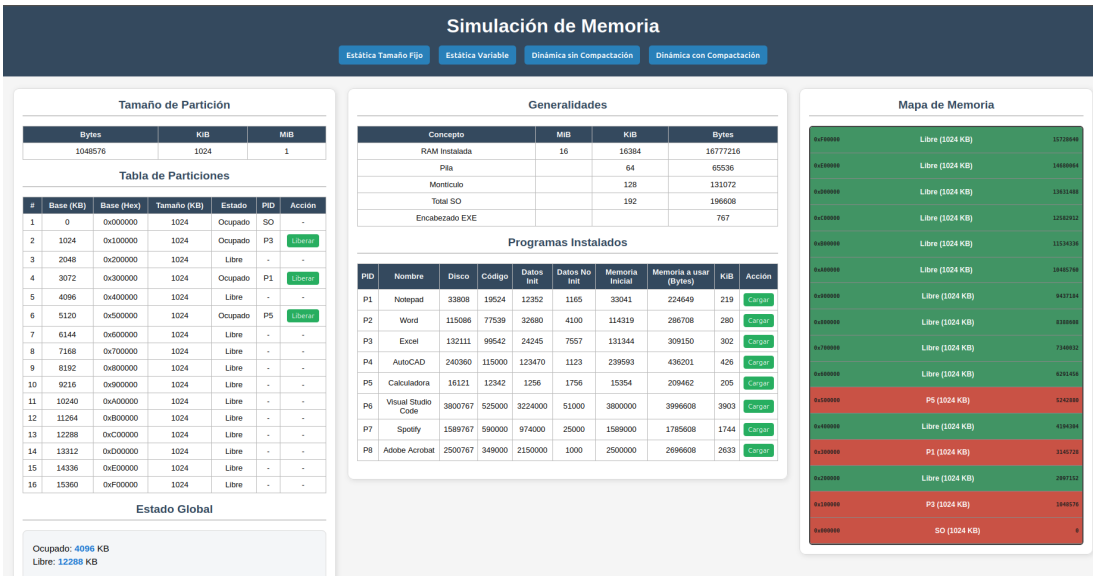

Figura 7: Estado de la memoria tras liberar un proceso.

particiones fijas, proporcionando una herramienta interactiva para analizar los principios de la gestión estática de memoria en sistemas operativos.

Memoria Estática Variable

La simulación de memoria estática variable amplía el modelo de particiones fijas al permitir la creación de bloques de diferentes tamaños desde la inicialización del sistema. En este esquema, las particiones no son uniformes y pueden adaptarse a los requerimientos de los procesos según el algoritmo de asignación seleccionado. La Figura 8 muestra la interfaz correspondiente a este modo de operación.

En esta modalidad, la clase `Memoria` crea un conjunto de particiones predefinidas de

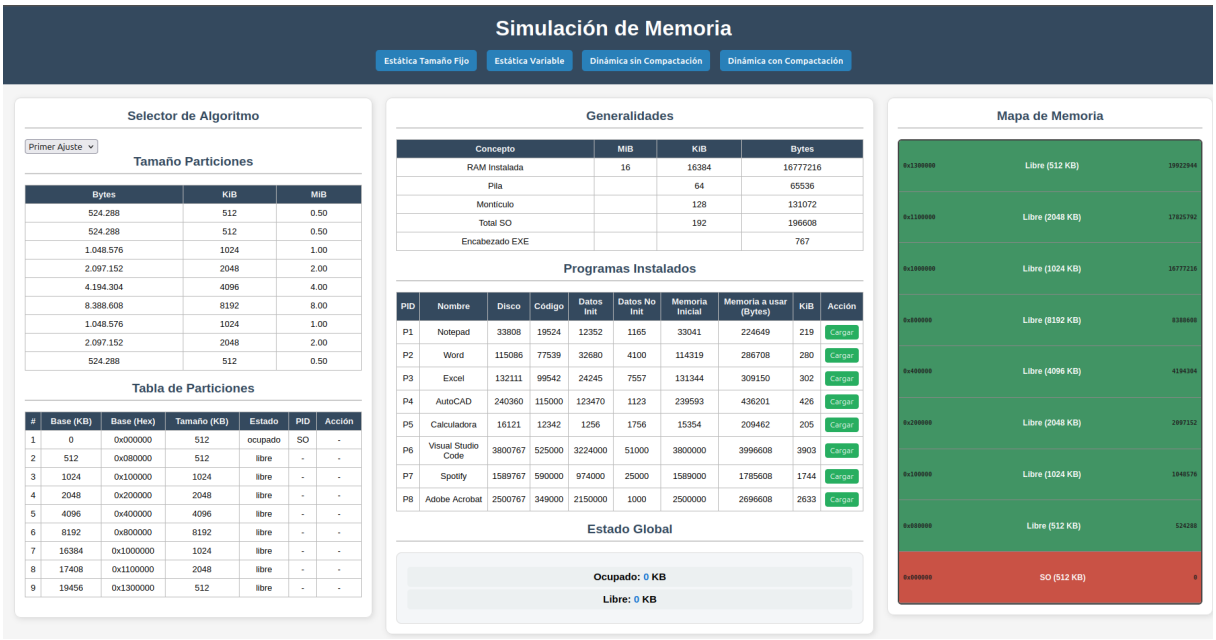

Figura 8: Interfaz inicial del simulador en modo de memoria estática variable.

tamaños variados, manteniendo una asignación estática (es decir, sin posibilidad de dividir ni fusionar bloques en tiempo de ejecución). Sin embargo, a diferencia del caso de tamaño fijo, el sistema permite elegir entre distintos **algoritmos de ajuste** para decidir en qué partición se cargará un nuevo proceso.

El usuario puede seleccionar cualquiera de estos algoritmos desde el menú desplegable **Selector de Algoritmo**. Cada vez que se presiona la opción **Cargar**, el simulador ejecuta el método `cargarProceso()`, que aplica el algoritmo seleccionado para encontrar la partición más adecuada. Una vez asignado el bloque, la función `renderizar()` actualiza tanto el mapa visual de memoria como las tablas de particiones y programas.

A continuación se presentan las visualizaciones resultantes al cargar múltiples procesos bajo cada estrategia de asignación. En ellas se aprecia cómo varía la distribución del espacio libre y la fragmentación según el algoritmo utilizado:

Finalmente, el botón **Liberar** ejecuta el método `liberarProceso()`, el cual identifica el bloque ocupado por el proceso correspondiente y lo marca nuevamente como libre. Este procedimiento no compacta la memoria ni combina particiones contiguas.

Este modelo resulta especialmente útil para estudiar el impacto de los diferentes algoritmos de asignación sobre el aprovechamiento del espacio disponible y el grado de fragmentación producido en sistemas que emplean particiones de tamaño desigual pero estáticas.

Memoria Dinámica sin Compactación

En este modo de simulación, la memoria se administra de manera dinámica, creando particiones ajustadas al tamaño de los procesos cargados. A diferencia del esquema

Figura 9: Distribución de memoria con el algoritmo de Primer Ajuste.

Figura 10: Distribución de memoria con el algoritmo de Mejor Ajuste.

estático, aquí no existen bloques predefinidos, sino que los espacios se asignan y liberan de forma flexible según las solicitudes del sistema. Sin embargo, al no realizar compactación, los espacios de memoria liberados no se fusionan, lo que puede provocar fragmentación externa a lo largo del tiempo.

La Figura 12 muestra la interfaz inicial del simulador, donde la memoria aparece completamente libre, a excepción del bloque reservado para el sistema operativo. En esta vista, la clase `MemoriaDinamica` inicializa la estructura de datos base con un único bloque libre que representa toda la memoria disponible del sistema.

Cuando el usuario selecciona un programa y presiona la opción **Cargar**, el sistema ejecuta el método `cargarProceso()`, el cual utiliza una estrategia de *Best-Fit*. Esta función busca el bloque libre más pequeño que pueda contener al proceso solicitado, evitando

Figura 11: Distribución de memoria con el algoritmo de Peor Ajuste.

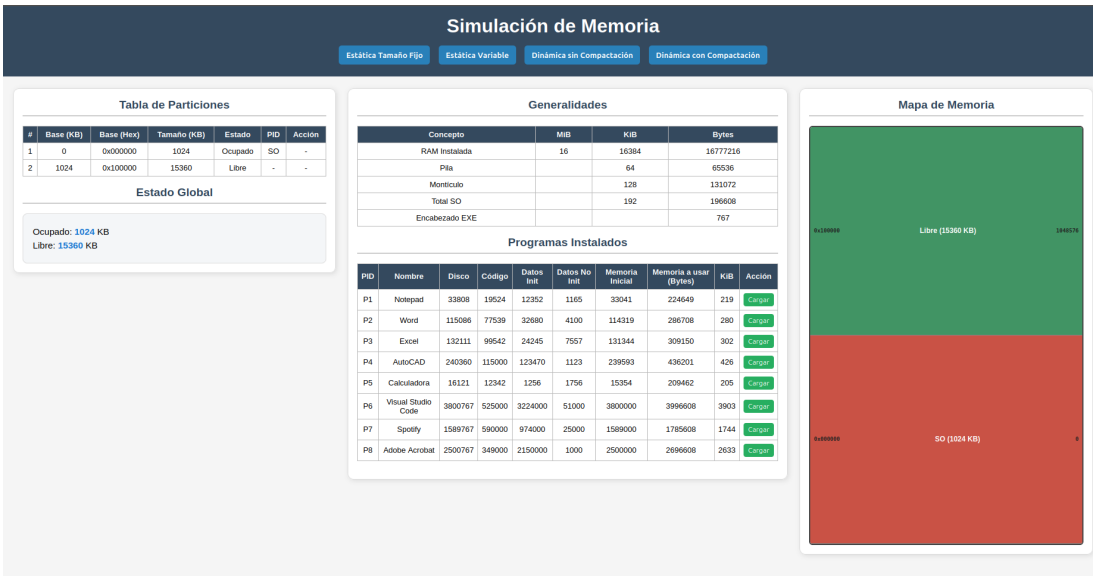

Figura 12: Interfaz inicial de la memoria dinámica sin compactación.

desperdiciar memoria. Si el bloque excede el tamaño necesario, se divide en dos: una nueva partición ocupada y otra libre. En la Figura 13 se aprecia este comportamiento, donde la memoria se fragmenta progresivamente con cada carga.

Posteriormente, al liberar un proceso mediante la opción **Liberar**, el método `liberarProceso()` marca la partición correspondiente como libre, sin realizar ningún tipo de reacomodo o fusión de espacios contiguos. Como resultado, pueden generarse múltiples espacios pequeños distribuidos entre los bloques ocupados, dificultando la carga de programas de mayor tamaño. Este efecto puede observarse en la Figura 14, donde se evidencia la fragmentación externa tras liberar algunos procesos.

Este módulo permite analizar visualmente los problemas de fragmentación asociados con la ausencia de compactación, proporcionando un entorno interactivo que refleja

Figura 13: Estado de la memoria dinámica después de cargar varios procesos.

Figura 14: Fragmentación de la memoria tras liberar procesos sin compactación.

fielmente el comportamiento de la gestión dinámica de memoria en sistemas operativos reales.

Memoria Dinámica con Compactación

En esta modalidad, el sistema implementa un esquema de gestión dinámica de memoria que incluye la **compactación automática** tras la liberación de procesos. Este enfoque permite reducir la fragmentación externa, reorganizando los bloques libres contiguos para optimizar el espacio disponible y mejorar la eficiencia de asignación.

La Figura 15 muestra el estado inicial de la simulación, donde únicamente el bloque correspondiente al sistema operativo aparece ocupado. La clase `MemoriaDinamica` es la encargada de gestionar la lista de particiones, y en este caso incorpora la lógica necesaria

para detectar espacios libres adyacentes y fusionarlos durante el proceso de liberación.

Figura 15: Interfaz inicial de la memoria dinámica con compactación.

Cuando el usuario selecciona un programa y hace clic en la opción **Cargar**, el sistema invoca el método `cargarProceso()`, que crea una nueva partición en el primer bloque libre con tamaño suficiente para contener el proceso. Si no existe espacio contiguo disponible, se muestra un mensaje de advertencia. En la Figura 16 se observa cómo los bloques de memoria comienzan a ocuparse de manera no uniforme, dejando fragmentos libres entre ellos.

Figura 16: Estado de la memoria después de cargar varios procesos.

Al liberar un proceso mediante la opción **Liberar**, el método `liberarProceso()` invoca de forma interna la función de compactación. Este procedimiento identifica todos los bloques libres y los reorganiza para consolidar un único espacio contiguo disponible al

final de la memoria. De esta manera, se eliminan los huecos intermedios y se incrementa la eficiencia del sistema al permitir la carga de procesos de mayor tamaño. La Figura 17 muestra el resultado posterior a la liberación de varios procesos, donde puede observarse la memoria completamente reacomodada.

Figura 17: Compactación automática tras liberar memoria.

Gracias a esta funcionalidad, el simulador ofrece una representación clara del impacto que tiene la compactación en la gestión de memoria, permitiendo observar cómo esta técnica reduce la fragmentación y mejora la utilización del espacio disponible. Este comportamiento refleja fielmente las estrategias empleadas por los sistemas operativos modernos para optimizar la asignación dinámica de memoria.

5. Conclusiones

El simulador `scriptLaboratorio.js` ilustra de forma didáctica los principales métodos de gestión de memoria:

- Las particiones fijas simplifican la gestión, pero desperdician memoria (**fragmentación interna**).
- Las particiones variables reducen el desperdicio, aunque pueden producir **fragmentación externa**.
- La asignación dinámica mejora la flexibilidad, dependiendo del algoritmo usado (**First Fit, Best Fit o Worst Fit**).
- La compactación es un mecanismo costoso pero necesario para recuperar memoria contigua.

6. Referencias

- Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Fundamentos de Sistemas Operativos*. Wiley.
- Stallings, W. (2014). *Operating Systems: Internals and Design Principles*. Pearson.
- Tanenbaum, A. S., & Bos, H. (2015). *Modern Operating Systems*. Pearson.