Simulador de Asignación de Memoria: First Fit, Best Fit y Worst Fit

Parcial SO-P2-2025-II

16 de octubre de 2025

1. Introducción

Este proyecto implementa un **simulador de asignación de memoria dinámica** en C++ que permite evaluar el comportamiento de tres algoritmos clásicos de administración de memoria:

- **First Fit**: asigna el primer bloque libre que sea lo suficientemente grande.
- **Best Fit**: asigna el bloque libre más pequeño que pueda contener el proceso.
- **Worst Fit**: asigna el bloque libre más grande disponible.

El programa permite leer una secuencia de operaciones desde archivo o consola, simulando la gestión de memoria de un sistema paginado. Entre las funcionalidades principales están: la asignación, liberación, visualización del estado de la memoria y el cálculo de fragmentación interna y externa.

2. Arquitectura del sistema

El código se estructura de forma modular y orientada a objetos, con una separación clara entre la lógica de cada componente:

- **BloqueMemoria**: representa un bloque físico de memoria, almacenando procesos y tamaño disponible.
- **Memoria**: agrupa los bloques y gestiona las operaciones de asignación, liberación y visualización.
- **AlgoritmoAsignacion**: clase base abstracta para las políticas de asignación.
- **FirstFit, BestFit, WorstFit**: clases concretas que implementan las estrategias respectivas.

- **Operacion** y **OperacionManager**: encargadas de leer, limpiar y estructurar las operaciones ingresadas.
- **ResultadoEjecucion**: almacena las métricas de fragmentación para cada ejecución.
- **main.cpp**: punto de entrada; permite seleccionar algoritmo, modo de ingreso y mostrar los resultados comparativos.

3. Flujo de ejecución

El flujo general del simulador es el siguiente:

1. El usuario elige el modo de ingreso de operaciones (archivo o manual).
2. El programa permite configurar la memoria (por bloques personalizados o por defecto).
3. Se selecciona el algoritmo a ejecutar (`First Fit`, `Best Fit`, `Worst Fit`).
4. Se procesan las operaciones:
   - **A <proceso><tamaño>:** Asigna memoria al proceso.
   - **L <proceso>:** Libera la memoria asociada.
   - **M:** Muestra el estado actual de la memoria.
5. Se calcula la fragmentación interna y externa y se genera un resumen comparativo.

4. Descripción de archivos y funciones

4.1. Algoritmos de asignación

- **AlgoritmoAsignacion.h** / **.cpp**: clase base abstracta con el método virtual `seleccionarHueco`.
- **FirstFit** / **BestFit** / **WorstFit**: cada clase redefine el método, implementando su estrategia de búsqueda:

Listing 1: Ejemplo de implementación de Best Fit

```
int BestFit::seleccionarHueco(const vector<BloqueMemoria>& memoria,
    int tama o) {
    int mejorIdx = -1;
    int mejorTam = numeric_limits<int>::max();
    for (int i = 0; i < memoria.size(); ++i) {
        if (memoria[i].estaLibre()) {
            int tam = memoria[i].getTama o();
```

```
            if (tam >= tama o && tam < mejorTam) {
                mejorTam = tam;
                mejorIdx = i;
            }
        }
    }
    return mejorIdx;
}
```

4.2. Memoria y Bloques

- **BloqueMemoria**: almacena información del bloque (inicio, tamaño, procesos y espacio disponible).
- **Memoria**: gestiona el conjunto de bloques. Posee los métodos:
  - `asignar()`: ejecuta el algoritmo de selección y actualiza la memoria.
  - `liberar()`: elimina el proceso y libera el espacio correspondiente.
  - `mostrar()`: imprime el estado de la memoria.
  - `calcularFragmentacion()`: calcula fragmentación interna y externa.

4.3. Gestión de operaciones

- **OperacionManager**: procesa cada línea de entrada y la convierte en un objeto `Operacion`.
- **ResultadoEjecucion**: guarda los resultados de cada algoritmo para el resumen comparativo.

5. Cálculo de fragmentación

El simulador implementa el cálculo de:

- **Fragmentación externa:** suma de todos los bloques totalmente libres.
- **Fragmentación interna:** suma del espacio libre dentro de los bloques parcialmente ocupados.
- **Huecos libres:** cantidad de bloques libres.
- **Mayor hueco:** tamaño del bloque libre más grande.

6. Casos de prueba

Cada caso de prueba se ejecutó configurando una memoria de 100 unidades dividida en bloques. Por defecto, la configuración base fue:

$$Bloques = [25, 25, 25, 25]$$

Los resultados se obtuvieron para cada uno de los tres algoritmos implementados.

6.1. Caso 1: Flujo básico

Configuración de bloques: [25, 25, 25, 25] **Algoritmo usado:** `First Fit`

Entrada

```
A P1 10
A P2 25
L P1
A P3 8
M
```

Salida esperada

```
1 --- [Libre: 25]
2 --- [P2:25][Libre: 0 ]
3 --- [P3: 8][Libre: 17]
4 --- [Libre: 25]
Fragmentaci n externa: 50
Huecos libres: 2
Bloque libre m s grande: 25
Fragmentaci n interna: 17
```

Análisis: El algoritmo `First Fit` reutiliza el hueco de `P1` para `P3`, mostrando un uso eficiente de la memoria.

6.2. Caso 2: Fragmentación interna

Configuración de bloques: [20, 20, 20, 20, 20] **Algoritmo usado:** `Best Fit`

Entrada

```
A A1 20
A A2 15
A A3 5
```

```
L A2
A A4 10
A A5 4
M
```

Salida esperada

```
1 --- [A1:20][Libre:0 ]
2 --- [A3: 5][A4:10][A5: 4][Libre: 1]
3 --- [Libre: 20]
4 --- [Libre: 20]
5 --- [Libre: 20]
Fragmentaci n externa: 60
Huecos libres: 3
Mayor hueco: 20
Fragmentaci n interna: 1
```

Análisis: `Best Fit` asigna el espacio exacto posible, pero deja pequeños espacios dentro de los bloques ocupados, generando fragmentación interna.

6.3. Caso 3: Fragmentación externa

Configuración de bloques: [30, 20, 25, 25] **Algoritmo usado:** `Worst Fit`

Entrada

```
A X1 30
A X2 20
A X3 15
L X1
A X4 25
M
```

Salida esperada

```
1 --- [X4:25][Libre: 5]
2 --- [Libre: 20]
3 --- [X2:20][Libre: 5]
4 --- [X3:15][Libre: 10]

Fragmentaci n externa: 20
Huecos libres: 1
Mayor hueco: 20
Fragmentaci n interna: 20
```

Análisis: Al liberar bloques no contiguos, se generan huecos dispersos que `Worst Fit` intenta aprovechar usando los más grandes.

6.4. Caso 4: Carga y liberación dinámica

Configuración de bloques: [10, 40, 20, 5, 25] **Algoritmo usado:** `Best Fit`

Entrada

```
A P1 10
A P2 15
A P3 8
L P2
A P4 5
A P5 12
L P3
A P6 10
L P1
A P7 6
M
```

Salida esperada

```
1 --- [P7:6][Libre: 4]
2 --- [Libre:40]
3 --- [Libre:20]
4 --- [P4: 5][Libre: 0]
5 --- [P5:12][P6:10][Libre: 3]
Fragmentaci n externa: 60
Huecos libres: 2
Bloque libre m s grande: 40
Fragmentaci n interna: 7
```

Análisis: Con varias asignaciones y liberaciones, `Best Fit` distribuye la memoria óptimamente al inicio, pero termina con espacios pequeños inservibles.

6.5. Caso 5: Memoria casi llena

Configuración de bloques: [10, 35, 5, 10, 50] **Algoritmo usado:** `Worst Fit`

Entrada

```
A T1 25
A T2 25
A T3 2
```

```
A T4 8
A T5 5
M
L T2
A T6 15
A T7 10
M
```

Salida esperada

```
1 --- [Libre 10]
2 --- [T2:25][Libre:10]
3 --- [Libre: 5]
4 --- [T1:25][T3: 2][T4: 8][T5: 5][Libre:10]
Fragmentaci n externa: 15
Huecos libres: 2
Bloque libre m s grande: 10
Fragmentaci n interna: 20
```

`Worst Fit` aprovecha el hueco más grande disponible.

7. Conclusiones

Los resultados muestran que:

- **First Fit** es rápido y eficiente en cargas simples.
- **Best Fit** optimiza espacio inicialmente, pero tiende a fragmentar internamente.
- **Worst Fit** es ideal para mantener huecos grandes disponibles, pero puede desaprovechar memoria.

Cada estrategia tiene ventajas según el patrón de solicitudes. La modularidad del código permite extender fácilmente el simulador con nuevos algoritmos o configuraciones.

8. Referencias

Referencias

[1] A. S. Tanenbaum y H. Bos, *Modern Operating Systems*, 4ta ed., Pearson, 2015.

[2] A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10ma ed., Wiley, 2018.