UMAG Universidad de Magallanes | Facultad de Ingeniería

Trabajo número 2 de Estructura de datos

Pablo Oyarzo[†], **Benjamin Sanhueza**[†] **y Duvan Figueroa**[†]

[†] *Universidad de Magallanes*

Este manuscrito fue compilado el 27 de octubre de 2024

Resumen

This work presents the implementation of a memory management and process planning simulator, which emulates the basic principles of an operating system. The main goal is to apply fundamental data structure concepts, such as linked lists and arrays, to efficiently manage memory allocation and process scheduling. Process scheduling algorithms such as FIFO and Round Robin have been implemented, as well as memory allocation algorithms such as Best Fit, First Fit and Worst Fit. The simulator manages multiple processes and allows the simulation of memory fragmentation and context switching between processes. This provides a comprehensive view of the challenges associated with resource allocation and release in operating systems, such as the complexity involved in the optimal allocation of limited resources. The results obtained demonstrate the efficiency of the proposed algorithms for different process and memory configurations, while highlighting the trade-offs in memory use, execution time, and CPU utilization.

Keywords: *Simulador de sistema operativo, Gestión de memoria, Planificación de procesos, Estructuras de datos, Algoritmos de asignación de memoria, FIFO, Round Robin, Best Fit, First Fit, Worst Fit, Fragmentación, CPU, Contexto de procesos*

Índice

1. Introducción

La gestión eficiente de la memoria y la planificación de procesos son pilares fundamentales en el diseño de sistemas operativos. Estos aspectos permiten maximizar el uso de los recursos disponibles, asegurando que los procesos se ejecuten de manera ordenada y eficiente, minimizando los tiempos de espera y optimizando el uso de la CPU.

A lo largo de los años, el diseño de los sistemas operativos ha evolucionado significativamente, desarrollando técnicas avanzadas para optimizar el uso de los recursos limitados disponibles, como la CPU, la memoria y los dispositivos de almacenamiento. Este desarrollo ha permitido que los sistemas operativos modernos puedan gestionar eficientemente grandes volúmenes de datos y procesos, incluso en sistemas multitarea. Los recursos deben ser gestionados cuidadosamente para evitar cuellos de botella que afecten el rendimiento y la estabilidad del sistema, ya sea en entornos pequeños como dispositivos embebidos, o en sistemas grandes como supercomputadoras.

La asignación de memoria y la planificación de procesos son componentes cruciales que influyen directamente en el rendimiento global del sistema operativo. Una gestión inadecuada de estos aspectos puede conducir a la fragmentación de la memoria, tanto interna como externa, y a una utilización ineficiente de los recursos disponibles. Estos problemas impactan negativamente en el rendimiento del sistema, incrementan los tiempos de espera de los procesos y pueden provocar situaciones críticas como la inanición de procesos.

Por ello, la elección de los algoritmos de planificación y asignación de recursos debe hacerse cuidadosamente, dependiendo de las necesidades específicas del sistema. En algunos casos, la prioridad es la equidad entre procesos, asegurando que todos reciban una cantidad justa de tiempo de CPU, mientras que en otros, la minimización del tiempo de espera o la maximización del rendimiento global del sistema puede ser más importante. Este informe tiene como objetivo estudiar y simular el comportamiento de varios algoritmos de gestión de memoria y planificación de procesos, proporcionando una herramienta didáctica que permita comprender mejor los compromisos entre estos algoritmos en términos de rendimiento y uso de recursos.

2. Metodología

El desarrollo de este simulador de gestión de memoria y planificación de procesos sigue un enfoque estructurado, compuesto por varias etapas clave que garantizan la correcta implementación y funcionamiento del sistema. Cada una de estas etapas está diseñada para abordar de manera eficiente los desafíos asociados a la gestión de memoria y la planificación de procesos en un entorno simulado, permitiendo explorar en detalle el comportamiento de los distintos algoritmos involucrados. En este apartado, se describen en profundidad los pasos seguidos en la construcción del simulador, que incluye desde la implementación de estructuras de datos esenciales hasta la simulación de la asignación dinámica de memoria y la planificación efectiva de los procesos en función de diferentes estrategias.

El simulador tiene como objetivo emular un entorno de sistema operativo, donde tanto la memoria como los procesos se gestionan de manera eficiente mediante el uso de estructuras de datos y algoritmos específicos. Para ello, se han implementado diversas fases que, en conjunto, permiten simular con precisión el comportamiento de un sistema real.

2.1. Estructuras de Datos

Las estructuras de datos constituyen el núcleo fundamental del simulador, ya que permiten modelar tanto los procesos como los bloques de memoria, facilitando la gestión eficiente de los recursos del sistema. Estas estructuras están diseñadas para soportar tanto la asignación dinámica de memoria como el control de los estados de los procesos a lo largo de la simulación. A continuación, se describen las principales estructuras utilizadas:

Config: Esta estructura almacena toda la configuración del sistema. Incluye parámetros críticos como la cantidad total de memoria disponible, el tamaño de los bloques de memoria, el número de núcleos de CPU y los algoritmos de asignación de memoria y planificación de procesos seleccionados por el usuario. La correcta configuración de estos parámetros permite que el simulador emule con precisión un entorno realista. Esta estructura es esencial para garantizar que las decisiones relacionadas con la asignación de memoria y la planificación se realicen de acuerdo con los algoritmos predefinidos.

Process: Cada proceso que participa en la simulación está representado por una estructura que contiene toda la información relevante para su ejecución. Esta información incluye su ID único, el tiempo de llegada al sistema, el tiempo de ejecución (o ráfaga), y la cantidad de memoria requerida para su correcto funcionamiento. Adicionalmente, cada proceso tiene asociado un estado, que puede ser uno de los siguientes: `NUEVO`, `LISTO`, `EJECUTANDO`, `ESPERA`, `BLOQUEADO` o `FINALIZADO`. Estos estados permiten realizar un seguimiento detallado del progreso de cada proceso dentro del sistema. Además, la estructura de proceso incluye un puntero a una pila (`stack`) que se utiliza para gestionar la memoria temporal asociada a cada proceso, así como un puntero al siguiente proceso en la lista, lo que facilita la organización en una lista enlazada.

MemoryBlock: Los bloques de memoria disponibles en el sistema están modelados mediante la estructura `MemoryBlock`. Esta estructura almacena información sobre el tamaño de cada bloque, su estado (libre u ocupado), y el proceso que lo está utilizando (en caso de que esté asignado). Estos bloques de memoria juegan un papel crucial en la gestión dinámica de la memoria, ya que son utilizados por los diferentes algoritmos de asignación implementados en el simulador, tales como *First Fit*, *Best Fit* y *Worst Fit*.

Estas estructuras se complementan con una serie de funciones clave que permiten gestionar la interacción entre los procesos y los bloques de memoria. Estas funciones se encargan de tareas como la lectura de la configuración del sistema, la creación de procesos, la asignación de memoria a los procesos, y la liberación de memoria una vez que los procesos finalizan su ejecución. Además, se ha implementado un mecanismo que permite manejar la pila de cada proceso, facilitando la gestión de memoria temporal durante la ejecución.

```
1 typedef struct Process {
2     char id[3];
3     int arrival_time;
4     int burst_time;
5     int memory_required;
6     ProcessState state;  // Estado actual del proceso
```

```
    Stack* stack;  // Pila asociada al proceso
    struct Process* next;  // Puntero al siguiente proceso
} Process;
```

Código 1. Estructura para representar procesos

```
typedef struct Stack {
    int size;
    int *data;
    int top;
} Stack;
```

Código 2. Definición de la pila

Estas estructuras permiten modelar tanto los procesos como la memoria del sistema. En particular, cada proceso tiene una pila asociada que puede ser utilizada para almacenar información temporal durante su ejecución.

2.2. Lectura de Configuración y Carga de Procesos

La primera fase del simulador implica la lectura de un archivo de entrada que contiene tanto la configuración del sistema como la lista de procesos. Esta configuración incluye parámetros como la memoria total disponible, el tamaño de los bloques de memoria, el algoritmo de asignación de memoria seleccionado, el número de núcleos de CPU y el algoritmo de planificación de procesos.

```
int read_config(const char *filename, Config *config) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        perror("Error al abrir el archivo");
        return 0;
    }

    // Leer par metros del archivo
    fscanf(file, "Total_Memory, %d", &config->total_memory);
    fclose(file);
    return 1;
}
```

Código 3. Lectura de configuración desde un archivo

La función `read_config()` se encarga de leer los parámetros desde un archivo de entrada y almacenarlos en la estructura `Config`.

2.3. Asignación de Memoria

El simulador admite tres estrategias principales de asignación de memoria: `First Fit`, `Best Fit` y `Worst Fit`. Cada estrategia se implementa de la siguiente manera:

- **First Fit**: El algoritmo asigna el primer bloque libre que cumpla con el tamaño solicitado por el proceso.
- **Best Fit**: El algoritmo busca el bloque más pequeño disponible que sea lo suficientemente grande para acomodar el proceso.
- **Worst Fit**: Este algoritmo asigna el bloque más grande disponible, dejando bloques pequeños libres para futuras asignaciones.

A continuación, se muestra un fragmento de la implementación del algoritmo de asignación *First Fit*:

```
int first_fit(MemoryBlock* head, int memory_required) {
    MemoryBlock* current_block = head;
    int accumulated_memory = 0;

    while (current_block != NULL) {
        if (current_block->free) {
            accumulated_memory += current_block->size;
            if (accumulated_memory >= memory_required) {
                // Asignar los bloques
                return 1;  // Asignaci n exitosa
            }
        }
        current_block = current_block->next;
    }

    return 0;  // No se pudo asignar memoria
}
```

Código 4. Implementación de First Fit

2.4. Planificación de Procesos

El simulador implementa dos algoritmos principales de planificación de procesos: `FIFO` y `Round Robin`.

FIFO (First In, First Out) ejecuta los procesos en el orden en que llegan, sin interrupciones, hasta que finalizan. Este algoritmo es fácil de implementar y muy eficiente para sistemas en los que los tiempos de ráfaga de los procesos son similares. A continuación se presenta un fragmento del código que implementa `FIFO`:

```c
void fifo(Process* head, MemoryBlock* memory, int asignacion_opcion) {
    Process* actual = head;
    int tiempo = 0; // contador de tiempo

    // Inicializa los procesos como "NUEVO"
    while (actual != NULL) {
        actual->state = NUEVO;
        actual = actual->next;
    }

    actual = head;
    printf("Inicio de la simulacion FIFO\n");

    while (actual != NULL) {
        if (actual->state == NUEVO) {
            actual->state = LISTO;
            printf("Proceso %s cambia a estado: listo\n", actual->id);
        }

        if (actual->state == LISTO) {
            actual->state = EJECUTANDO;
            printf("Ejecutando %s\n", actual->id);

            // Simula la ejecuci n del proceso
            sleep(actual->burst_time);
            tiempo += actual->burst_time;

            // Marca el proceso como finalizado y libera memoria
            actual->state = FINALIZADO;
            free_memory(memory, actual->memory_required);
        }

        printf("Proceso %s termina en tiempo %d\n", actual->id, tiempo);
        actual = actual->next;
    }

    printf("Fin de la simulaci n FIFO\n");
}
```

Código 5. Fragmento de implementación de FIFO

Round Robin es un algoritmo que divide el tiempo de CPU en intervalos denominados *quantum*. Los procesos se ejecutan durante un período de tiempo limitado y, si no terminan, se les da otra oportunidad en el próximo ciclo. Este algoritmo es más equitativo, pero puede implicar más cambios de contexto. A continuación, se muestra un fragmento de la implementación de `Round Robin`:

```c
void round_robin(Process* head, int quantum, MemoryBlock* memory, int asignacion_opcion) {
    Process* actual = head;
    int tiempo = 0;
    int ciclos = 1;

    printf("Inicio de la simulaci n Round Robin\n");

    do {
        actual = head;
        while (actual != NULL) {
            if (actual->arrival_time <= tiempo && actual->burst_time > 0) {
                actual->state = EJECUTANDO;
                printf("Ciclo %d: procesando %s\n", ciclos, actual->id);

                if (actual->burst_time > quantum) {
                    actual->burst_time -= quantum;
                    tiempo += quantum;
                    actual->state = ESPERA;
                } else {
                    tiempo += actual->burst_time;
                    actual->burst_time = 0;
```

```
                    actual->state = FINALIZADO;
                    free_memory(memory, actual->memory_required);
                    printf("Proceso %s finalizado en tiempo %d\n", actual->id, tiempo);
                }
            }
            actual = actual->next;
        }
        ciclos++;
    } while (/* procesos pendientes */);

    printf("Fin de la simulaci n Round Robin\n");
}
```

Código 6. Fragmento de implementación de Round Robin

En el caso de `Round Robin`, se asegura que cada proceso reciba un tiempo equitativo en la CPU, permitiendo que los procesos más pequeños se completen rápidamente y evitando que un solo proceso monopolice la CPU por mucho tiempo.

2.5. Liberación de Memoria y Finalización de Procesos

Cuando un proceso finaliza su ejecución, los bloques de memoria que ocupaba deben liberarse para que otros procesos puedan hacer uso de esos recursos. Esto se realiza mediante la función `free_memory()`.

```
void free_memory(MemoryBlock* head, int memory_required) {
    MemoryBlock* current = head;
    int freed_memory = 0;

    while (current != NULL && freed_memory < memory_required) {
        if (!current->free) {
            current->free = 1;  // Liberar bloque
            freed_memory += current->size;
        }
        current = current->next;
    }
}
```

Código 7. Liberación de memoria asignada

Esta función recorre los bloques de memoria y libera aquellos que estaban ocupados por el proceso finalizado.

3. Resultados

Se realizaron diversas simulaciones utilizando los algoritmos de planificación y asignación mencionados previamente. Los resultados obtenidos muestran las diferencias en el comportamiento de los procesos y la memoria bajo diferentes configuraciones y algoritmos. Los siguientes apartados presentan un análisis detallado de las simulaciones realizadas.

3.1. Ejecución con FIFO

En la simulación con el algoritmo FIFO, los procesos se ejecutan en el orden en que llegan, sin interrupciones. El tiempo total de ejecución es la suma de los tiempos de ráfaga de todos los procesos. Este algoritmo es eficiente en situaciones donde los tiempos de ráfaga de los procesos son relativamente cortos y similares entre sí. Sin embargo, si un proceso con una ráfaga muy larga llega primero, los procesos que llegan después tendrán que esperar mucho tiempo, lo que puede llevar a tiempos de espera elevados.

```
        Seleccione el algoritmo de asignacion de memoria:
        1. First Fit
        2. Best Fit
        3. Worst Fit
        Ingrese su opcion: 2
        Seleccione el algoritmo de planificacion:
        1. FIFO
        2. Round Robin
        Ingrese su opcion: 1

        Iniciando planificador FIFO...

        ========== Inicio de la simulacion FIFO ==========
        Proceso P1 cambia a estado: LISTO
        ...
        Proceso P5 termina en tiempo 26
        Tiempo total de ejecucion: 26 segundos
        ========== Fin de la simulacion ==========

```

3.2. Ejecución con Round Robin

En la simulación con el algoritmo Round Robin, los procesos se ejecutan en fragmentos de tiempo (quantum) definidos por el usuario. El uso de un quantum de 5 unidades permitió alternar entre los procesos de manera equitativa, reduciendo el tiempo de espera para los procesos con ráfagas más cortas. Sin embargo, para los procesos con ráfagas más largas, la ejecución completa tomó más ciclos en comparación con FIFO, lo que incrementó el tiempo total de ejecución.

A continuación, se muestra un diagrama de Gantt que ilustra la ejecución de los procesos bajo el algoritmo Round Robin:

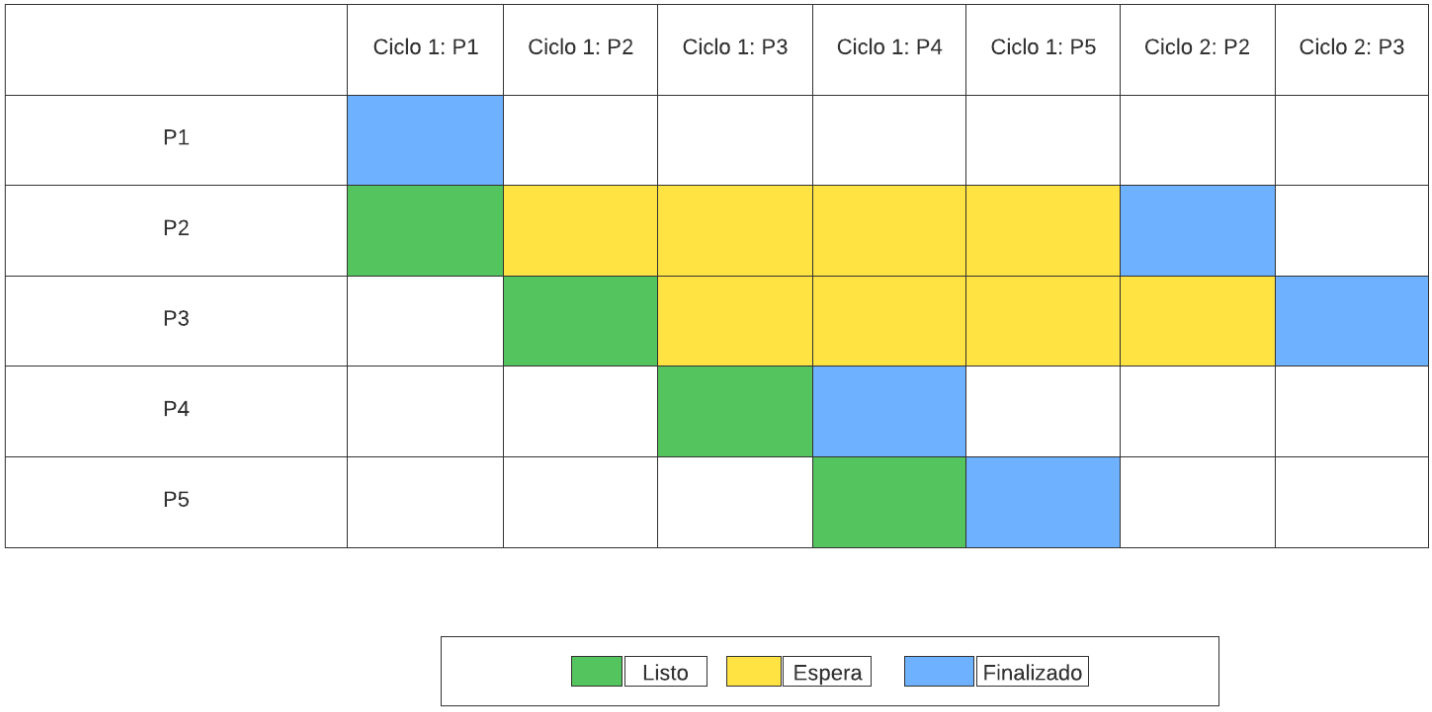

Figura 1. Diagrama de Gantt de la ejecución con Round Robin

3.3. Asignación de Memoria

La asignación de memoria es uno de los aspectos más importantes en la gestión de procesos dentro de un sistema operativo. Dependiendo del algoritmo de planificación utilizado, el comportamiento de la memoria puede variar significativamente. En este apartado, se presenta un análisis de cómo el uso de memoria cambia bajo diferentes algoritmos de planificación, en particular, *FIFO* y *Round Robin*. Ambos algoritmos presentan características diferentes en cuanto al uso de los recursos del sistema, lo cual es clave para comprender la eficiencia de la asignación de memoria en sistemas multitarea.

El algoritmo *FIFO* tiende a hacer un uso más constante y predecible de la memoria, ya que los procesos se ejecutan en el orden de llegada y no son interrumpidos hasta que finalizan. Por el contrario, el algoritmo *Round Robin* introduce variabilidad en el uso de memoria debido a las interrupciones frecuentes, lo que puede llevar a una mayor fragmentación en la memoria.

3.3.1. Uso de Memoria en FIFO

En la Figura 2, se muestra el uso de memoria en cada ciclo de ejecución cuando se utiliza el algoritmo *FIFO*. Como se puede observar, el uso de memoria es predecible y constante, lo que facilita una asignación eficiente y una liberación ordenada de los bloques de memoria. Dado que los procesos se ejecutan de forma secuencial sin interrupciones, la memoria se libera inmediatamente después de que un proceso termina, lo que minimiza la fragmentación interna.

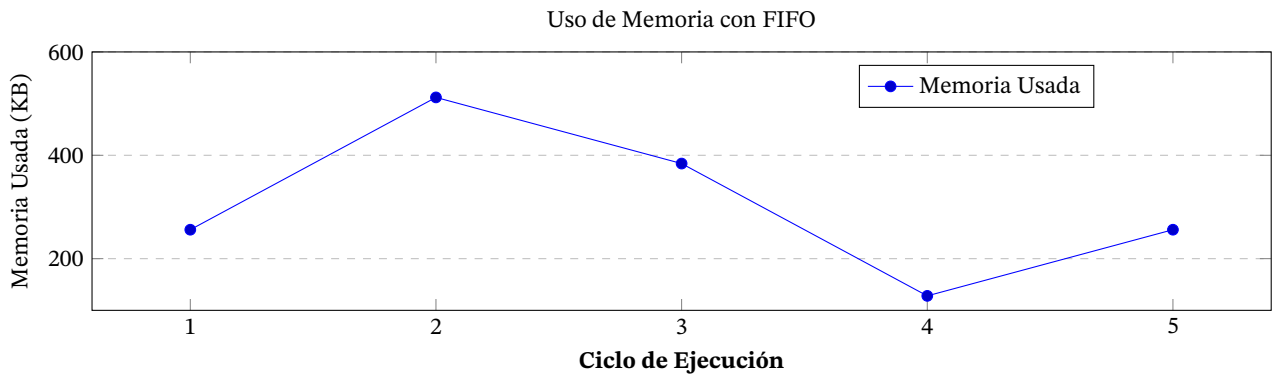

Figura 2. Uso de Memoria en cada ciclo de ejecución para FIFO

Este comportamiento es ideal en situaciones donde los tiempos de ráfaga de los procesos son similares, ya que minimiza el tiempo de espera de los procesos en cola. Sin embargo, cuando existen procesos con ráfagas de mayor duración, los procesos más pequeños pueden experimentar tiempos de espera más largos, lo que puede reducir la eficiencia global del sistema.

3.3.2. Uso de Memoria en Round Robin

Por otro lado, el algoritmo *Round Robin* asigna el tiempo de CPU de forma equitativa entre los procesos mediante el uso de intervalos de tiempo denominados *quantum*. A diferencia de *FIFO*, en *Round Robin* los procesos son interrumpidos y reprogramados al finalizar cada quantum, lo que provoca un uso más variable de la memoria. En la Figura 3, se observa cómo el uso de memoria fluctúa a medida que los procesos son interrumpidos y reprogramados, lo que incrementa la posibilidad de fragmentación interna y externa.

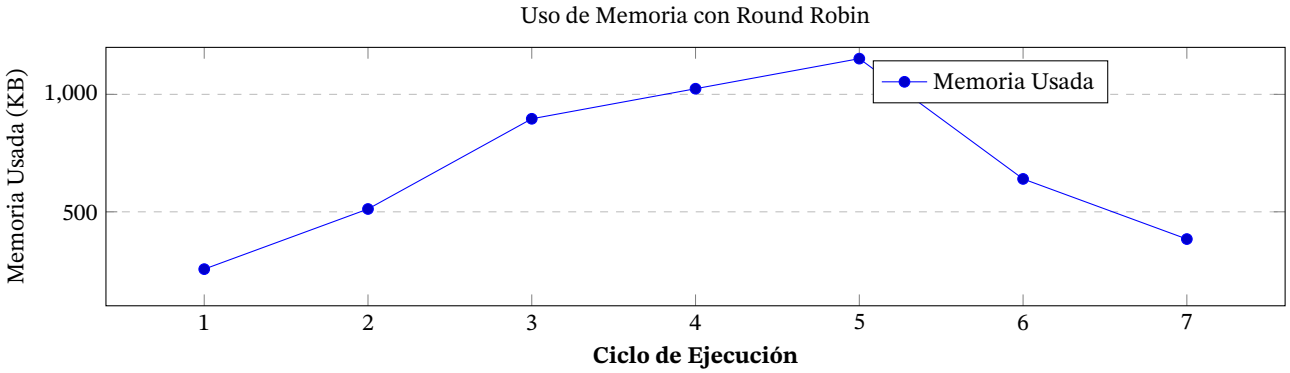

Figura 3. Uso de Memoria en cada ciclo de ejecución para Round Robin

Debido a la naturaleza de este algoritmo, el uso de memoria es menos predecible, lo que puede llevar a picos de consumo en momentos donde varios procesos requieren grandes cantidades de memoria de forma simultánea. Además, la constante reprogramación de procesos puede llevar a una mayor fragmentación de la memoria, especialmente en sistemas con muchas tareas que requieren diferentes cantidades de memoria.

3.3.3. Comparación de FIFO y Round Robin

Comparando ambos gráficos, podemos observar que el algoritmo *FIFO* tiende a utilizar la memoria de manera más predecible y constante, mientras que el algoritmo *Round Robin* genera picos de uso debido a la naturaleza de su planificación, especialmente en sistemas donde el quantum es menor que el tiempo de ráfaga de los procesos. Mientras que *FIFO* es ideal para sistemas en los que se prioriza la eficiencia de la memoria, *Round Robin* es más adecuado para sistemas donde se busca un reparto equitativo del tiempo de CPU entre todos los procesos, aunque esto pueda generar un mayor grado de fragmentación en la memoria.

En resumen, la elección entre *FIFO* y *Round Robin* depende del contexto específico en el que se apliquen. *FIFO* es útil en situaciones donde los procesos tienen tiempos de ráfaga similares y la prioridad es minimizar la fragmentación de la memoria, mientras que *Round Robin* es más adecuado en entornos donde es importante que todos los procesos reciban tiempo de CPU de manera justa, a expensas de un mayor uso de memoria y potencialmente más fragmentación.

4. Conclusión

Este trabajo ha implementado un simulador de gestión de memoria y planificación de procesos en un entorno controlado, permitiendo analizar el comportamiento de diversos algoritmos en diferentes escenarios. La simulación de estos algoritmos proporciona una plataforma útil para comprender cómo se gestionan los recursos limitados en un sistema operativo y los compromisos que deben hacerse al elegir diferentes métodos de asignación y planificación.

Los algoritmos de asignación de memoria First Fit, Best Fit y Worst Fit han sido probados con diversas configuraciones de memoria y procesos, demostrando que cada uno tiene ventajas y desventajas en función del contexto de uso. El algoritmo First Fit demostró ser el más rápido en términos de tiempo de asignación, debido a su simplicidad al seleccionar el primer bloque que cumple con los requisitos del proceso. Sin embargo, esta simplicidad puede causar fragmentación interna considerable cuando los tamaños de los procesos varían significativamente. Este fenómeno se debe a que los procesos más pequeños pueden ocupar bloques más grandes de memoria, lo que deja fragmentos de espacio que pueden ser difíciles de reutilizar eficientemente.

Por otro lado, el algoritmo Best Fit, aunque más eficiente en términos de reducción de fragmentación interna, puede causar mayor fragmentación externa a largo plazo, ya que deja bloques pequeños dispersos por todo el espacio de memoria. En sistemas con alta concurrencia de procesos, esto podría llevar a que la memoria libre se fragmente en pequeñas porciones que ya no pueden ser asignadas a nuevos procesos, incrementando así el número de fallos de asignación.

En el caso del algoritmo Worst Fit, se observó que tiende a generar un menor grado de fragmentación externa, dado que asigna los bloques más grandes disponibles primero. Este enfoque, si bien puede parecer contraintuitivo, ayuda a dejar grandes bloques disponibles para los procesos futuros, pero puede no ser eficiente cuando los procesos que llegan son predominantemente pequeños, lo que podría llevar a un desperdicio considerable de memoria en bloques de gran tamaño. Sin embargo, en entornos donde la memoria se solicita en tamaños variados y donde la disponibilidad de grandes bloques de memoria es crucial, Worst Fit puede ser una estrategia valiosa.

Respecto a la planificación de procesos, la simulación con el algoritmo FIFO (First In, First Out) mostró que este enfoque es ideal en situaciones donde los tiempos de ráfaga de los procesos son homogéneos o similares. Este algoritmo es fácil de implementar y tiene una baja sobrecarga computacional, ya que no requiere realizar cambios frecuentes en el estado de los procesos. Sin embargo, uno de los mayores inconvenientes observados en este enfoque es el potencial de inanición de procesos. Esto puede ocurrir si un proceso con una ráfaga larga ocupa la CPU durante mucho tiempo, lo que obliga a los procesos que llegan después a esperar hasta que el primero termine. Este comportamiento es particularmente problemático en sistemas donde se requiere una rápida respuesta para ciertos procesos críticos.