

Simulación de un Sistema Operativo en Lenguaje C \LaTeX

Miguel Loaiza[†], Oscar Cifuentes[†] y Diego Sanhueza[†]

[†]Universidad de Magallanes

Este manuscrito fue compilado el: 27 de octubre de 2024

Resumen

Este informe presenta el desarrollo de una simulación de un sistema operativo básico implementado en lenguaje C, utilizando estructuras de datos como listas simplemente enlazadas y colas. El algoritmo de planificación empleado en este trabajo es FIFO (First In, First Out). Se describe la arquitectura del sistema, destacando la implementación de la gestión de procesos y la asignación de memoria. A través de simulaciones, se evalúa el rendimiento del sistema en diversas cargas de trabajo, analizando cómo la planificación FIFO y el algoritmo de asignación de memoria (First Fit) afectan en la eficiencia en la ejecución de procesos.

Keywords: Listas, Colas, Estructuras de datos

■ Índice

| | | |
|----------|---|-----------|
| 1 | Introducción | 2 |
| 2 | Descripción del problema | 2 |
| 2.1 | Explicación de la entrada | 2 |
| 3 | Extractos de código de las funciones | 3 |
| 3.1 | Archivo 'auxiliar.h' | 3 |
| 3.2 | Archivo 'memoria.h' | 4 |
| 3.3 | Archivo 'planificador.h' | 5 |
| 3.4 | Archivo 'main.c' | 6 |
| 3.5 | Archivo 'auxiliar.c' | 7 |
| 3.6 | Archivo 'memoria.c' | 8 |
| 3.7 | Archivo 'planificador.c' | 8 |
| 4 | Seguimiento del programa | 10 |
| 5 | Pruebas y resultados | 13 |
| 5.1 | ¿Como nuestro programa maneja la salida de los datos? | 13 |
| 5.2 | Salida del programa | 13 |
| 6 | Dificultades y soluciones | 15 |
| 6.1 | ¿Qué problemas enfrentamos? | 15 |
| 6.2 | Problemas con el compilador | 15 |
| 6.3 | Soluciones a los problemas | 15 |

1. Introducción

Los *sistemas operativos* son herramientas que utilizamos cotidianamente. Generalmente, pasamos por alto sus funciones más esenciales, las cuales permiten la ejecución de diversos procesos. Muchas de estas funciones existen gracias a las *estructuras de datos*, como las que estudiamos en este curso. En este trabajo, implementaremos las funciones básicas de un *sistema operativo* empleando *arreglos*, *colas*, *listas simplemente enlazadas* y *memoria dinámica*.

Este proyecto tiene un repositorio en GitHub, el cual está disponible en: [os-simulator.git](https://github.com/os-simulator), donde se encuentra la implementación.

2. Descripción del problema

Se debe implementar una simulación de *Sistema Operativo* que logre gestionar la memoria de los procesos que se entreguen del archivo de texto (entrada.txt), en este archivo de texto deben estar los algoritmos utilizados (FIFO y FF), el tamaño de la memoria total, el tamaño de cada bloque de memoria y los procesos (id del proceso, tiempo de llegada, tiempo de ráfaga y memoria solicitada).

```

1    2048 128 ff
2    fifo
3    0 0 5 250
4    1 2 3 128
5    2 1 1 122
6    3 2 2 19
7    4 4 3 98
8    5 6 6 459
9    6 7 7 2049
10   7 8 8 128

```

Figura 1. Archivo de entrada .txt con los procesos, algoritmos utilizados, bloques de memoria y memoria total.

La entrada contempla varios campos que se leerán desde el programa para poder realizar las acciones correspondientes, las cuales se detallarán más adelante.

2.1. Explicación de la entrada

Esta entrada contempla en la línea 1 la memoria total del sistema operativo, tamaño de cada bloque (ambos en KB) y el algoritmo de asignación de memoria (First Fit).

En la línea 2, se agrega el algoritmo de planificación empleado para la ejecución de los procesos (FIFO).

En la línea 3 se tiene, de izquierda a derecha, el PID del proceso, el tiempo de llegada, el tiempo de ráfaga, y la memoria solicitada al sistema operativo.

3. Extractos de código de las funciones

A continuación se presenta un breve análisis de los archivos que conforman el programa desarrollado. Cada uno de ellos tienen un papel importante en el funcionamiento general, teniendo funciones específicas que ayudan a gestionar los procesos y la memoria de manera eficiente.

3.1. Archivo 'auxiliar.h'

Este archivo de cabecera define las funciones auxiliares necesarias para la gestión de procesos y memoria en el proyecto. Su objetivo es proporcionar declaraciones para varias funciones que facilitan la gestión de procesos y la planificación de memoria. Estas funciones son importantes para operar con estructuras de datos relacionadas con procesos y memoria en el sistema.

```

1  #ifndef AUXILIAR_H
2  #define AUXILIAR_H
3
4  #include "memoria.h"
5
6  // Funciones AUXILIARES para las funciones de MEMORIA y PLANIFICACIÓN.
7  void leer_entrada(const char *, int *, int *, char *, char *, Cola *); // LEER la entrada del ARCHIVO.
8  void asignar_valores_procesos(int, int, int, int, Cola *); // Asignar VALORES a los PROCESOS
9
10 void imprimirCola_procesos(Cola *); // Imprimir la COLA de PROCESOS.
11 void imprimir_memoria(BloqueMemoria *, int); // Imprimir los BLOQUES de
12 // MEMORIA INICIALIZADOS.
13 void registrar_tiempos(Gantt *, int, int, int, int *); // Registrar los TIEMPOS de los
14 // PROCESOS.
15 void imprimir_gantt(Gantt *, int); // Imprimir el DIAGRAMA de GANTT.
16 void generar_archivo_gantt(Gantt *, int, const char *); // Generar el ARCHIVO EPS que
17 // muestra la CARTA GANTT.
18 void generar_archivo_gantt(Gantt *, int, const char *); // Generar el ARCHIVO EPS que
19 // muestra la CARTA GANTT.
20 #endif // AUXILIAR_H

```

Código 1. auxiliar.h.

La función *leer_entrada* se encarga de obtener los valores almacenados en un archivo .txt y asignarlos a las variables correspondientes. La función *asignar_valores_procesos* permite crear nuevos procesos e inicializarlos con los valores obtenidos de la entrada. La función *imprimirCola_procesos* recorre la cola de procesos actual y la muestra en pantalla. Por su parte, la función *imprimir_memoria* permite visualizar el estado de la memoria, incluyendo los bloques ocupados y el tamaño de cada uno.

Las funciones *registrar_tiempos*, *imprimir_gantt* y *generar_archivo_gantt* están relacionadas con la salida final del programa. Estas funciones generan la carta Gantt, registran los tiempos y crean un archivo .EPS al finalizar la ejecución, mostrando el resultado del proceso de planificación.

3.2. Archivo 'memoria.h'

Este archivo de cabecera define la estructura *BloqueMemoria*, que es fundamental para la gestión de memoria en el sistema operativo. Esta estructura incluye dos campos: *tamano*, que representa el tamaño del bloque de memoria, y *estado*, que indica si el bloque está **LIBRE (1)** o **OCUPADO (0)**, es decir, si tiene un proceso asignado.

Además, se declaran varias funciones relacionadas con la gestión de la memoria, como *inicializar_bloques_memoria*, que inicializa los bloques de memoria; *asignar_memoria_procesos*, que asigna memoria a los procesos para su ejecución; y *asignar_bloques_proceso*, que asigna un bloque de memoria específico a un proceso. También se incluye *verificar_fragmentacion*, que permite comprobar la fragmentación interna de los bloques, y *manejar_proceso*, que gestiona un proceso en función del bloque de memoria asignado. Finalmente, las funciones *ejecutar_proceso* y *liberar_memoria* simulan la ejecución de un proceso en un bloque de memoria y liberan la memoria ocupada por un proceso, respectivamente.

```

1  #ifndef MEMORIA_H
2  #define MEMORIA_H
3
4  #define MAX_PROCESOS 100
5
6  #include "planificador.h"
7
8  // Estructura de un BLOQUE de MEMORIA.
9  typedef struct
10 {
11     int tamano; // Tamaño o BLOQUE de memoria.
12     int estado; // 1 es bloque LIBRE, 0 es bloque OCUPADO.
13 } BloqueMemoria;
14
15 // Funciones de MEMORIA FF (First Fit).
16 void inicializar_bloques_memoria(BloqueMemoria *, int, int);           // Inicializar los BLOQUES
    de MEMORIA.
17 void asignar_memoria_procesos(Cola *, BloqueMemoria *, int);           // Asignar MEMORIA a los
    PROCESOS para la EJECUCIÓN.
18 void asignar_bloque_proceso(BloqueMemoria *, Proceso *, int *, int);   // Asignar un BLOQUE de
    MEMORIA a un PROCESO.
19 void verificar_fragmentacion(BloqueMemoria *, int);                   // Verificar la
    FRAGMENTACIÓN INTERNA.
20 void manejar_proceso(BloqueMemoria *, Proceso *, int, Gantt *, int *, int *); // Manejar el PROCESO.
21 void ejecutar_proceso(BloqueMemoria *, Proceso *, int);               // Simula la ejecución de
    un proceso de un BLOQUE de MEMORIA.
22 void liberar_memoria(BloqueMemoria *, int);                           // Liberar la MEMORIA de
    un PROCESO.
23
24 #endif // MEMORIA_H

```

Código 2. memoria.h.

En conjunto, esta estructura y estas funciones son esenciales para implementar un esquema de gestión de memoria eficiente en el sistema operativo.

3.3. Archivo 'planificador.h'

Este archivo de cabecera define las estructuras *Proceso*, *Cola* y *Gantt*, las cuales son importantes para la planificación de procesos en el sistema operativo. La estructura *Proceso* incluye campos como el ID del proceso (*pid*), el tiempo de llegada (*tiempo_llegada*), el tiempo de ejecución (*tiempo_rafaga*) y la memoria solicitada (*memoria_solicitada*), además de un puntero al siguiente proceso en la lista. La estructura *Cola* permite gestionar los procesos en espera, manteniendo punteros al frente y al final de la cola. Por su parte, la estructura *Gantt* se utiliza para representar el tiempo de ejecución de los procesos, incluyendo el ID del proceso, el tiempo de inicio y el tiempo de finalización. Además, se declaran las funciones *enqueue* y *dequeue*, que permiten agregar y quitar los procesos de la cola, respectivamente.

Finalmente, el archivo incluye las bibliotecas estándar de C como lo son; *string.h*, *stdlib.h*, *unistd.h* y *stdio.h*, que son necesarias para la funcionalidad del programa.

```

1  #ifndef PLANIFICADOR_H
2  #define PLANIFICADOR_H
3
4  #include <string.h>
5  #include <stdlib.h>
6  #include <unistd.h>
7  #include <stdio.h>
8
9  // Estructura de un PROCESO. (PASA A SER UNA LISTA PORQUE LA MEMORIA SER VISTA COMO ARREGLO).
10 typedef struct Proceso
11 {
12     int pid;                // ID del proceso.
13     int tiempo_llegada;     // Tiempo de LLEGADA en segundos.
14     int tiempo_rafaga;     // Tiempo de EJECUCI N en segundos.
15     int memoria_solicitada; // MEMORIA solicitada en KB.
16     struct Proceso *next;   // Siguiente PROCESO.
17 } Proceso;
18
19 // Estructura de una COLA.
20 typedef struct
21 {
22     Proceso *front; // FRENTE de la cola.
23     Proceso *rear;  // FINAL de la cola.
24 } Cola;
25
26 // Estructura de la carta GANTT
27 typedef struct
28 {
29     int pid;                // ID del PROCESO.
30     int tiempo_inicio;     // Tiempo de INICIO.
31     int tiempo_final;      // Tiempo de FINALIZACI N.
32 } Gantt;
33
34 // Prototipos de funciones de PLANIFICACI N
35 void enqueue(Cola *cola, Proceso *proceso);
36 Proceso *dequeue(Cola *cola);
37
38 #endif // PLANIFICADOR_H

```

Código 3. planificador.h.

3.4. Archivo 'main.c'

Este archivo contiene la función principal del programa, donde se manejan los argumentos de línea de comandos, se leen los parámetros de entrada y se gestionan los procesos y la memoria. Se encarga de inicializar las estructuras necesarias y coordinar la ejecución de las funciones definidas en otros archivos.

```

1  #include "auxiliar.h"
2
3  int main(int argc, char *argv[])
4  {
5      int opt; // VARIABLE que GUARDA el ARGUMENTO que se pasa por la TERMINAL.
6      char *nombre_archivo = NULL; // VARIABLE que GUARDA el NOMBRE del ARCHIVO de ENTRADA.
7
8      // getopt devuelve -1 cuando ya LEY los ARGUMENTOS
9      while ((opt = getopt(argc, argv, "hf:")) != -1)
10     {
11         switch (opt)
12         {
13             case 'h': // Si se pasa el ARGUMENTO <-h>, se imprime un MENSAJE de AYUDA.
14                 fprintf(stdout, "\nPara leer sus procesos, porfavor coloque el parametro <-f> <
nombre_archivo.txt>\n\n");
15                 break;
16             case 'f': // 'optarg' contiene el ARGUMENTO que viene despu s de -f, es una VARIABLE GLOBAL
dada por la LIBRER A.
17                 nombre_archivo = optarg;
18                 printf("\nArchivo: %s\n\n", nombre_archivo);
19                 break;
20             case '?': // Si el ARGUMENTO no es reconocido, se imprime un MENSAJE de ERROR.
21                 fprintf(stderr, "Opci n no reconocida: -%c\n", optopt);
22                 exit(EXIT_FAILURE);
23             default: // Si no se pasan ARGUMENTOS, se imprime un MENSAJE de ERROR.
24                 fprintf(stderr, "Uso: %s [-h] [-f filename]\n", argv[0]);
25                 exit(EXIT_FAILURE);
26         }
27     }
28
29     int tamano_memoria, tamano_bloque; // VARIABLE del tama o de la memoria total
(2048 KB) y tama o del bloque de memoria (128 KB).
30     char algoritmo_memoria[10], algoritmo_planificacion[10]; // VARIABLE que GUARDA el algoritmo de
memoria first fit y VARIABLE que GUARDA el algoritmo de planificaci n FIFO.
31     Cola cola_procesos = {NULL, NULL}; // VARIABLE que GUARDA la COLA de PROCESOS.
32
33     leer_entrada(nombre_archivo, &tamano_memoria, &tamano_bloque, algoritmo_memoria,
algoritmo_planificacion, &cola_procesos);
34
35     int cantidad_bloques = tamano_memoria / tamano_bloque; // VARIABLE que GUARDA la cantidad de bloques
de memoria.
36     BloqueMemoria memoria[cantidad_bloques]; // ARREGLO de BLOQUES de MEMORIA.
37
38     inicializar_bloques_memoria(memoria, cantidad_bloques, tamano_bloque);
39
40     imprimir_memoria(memoria, cantidad_bloques);
41
42     asignar_memoria_procesos(&cola_procesos, memoria, cantidad_bloques);
43
44     return EXIT_SUCCESS;
45 }

```

Código 4. main.c.

En este archivo se implementa la función main, que es el punto de entrada del programa. Se utilizan argumentos de línea de comandos para permitir al usuario especificar un archivo de entrada mediante el parámetro -f. Si el usuario pasa -h, se imprime un mensaje de ayuda. Luego, se inicializan variables para almacenar el tamaño de la memoria, el tamaño del bloque y los algoritmos a utilizar. A continuación, se llama a la función leer_entrada para cargar la configuración desde el archivo. Se calcula la cantidad de bloques de memoria y se inicializan. Finalmente, se imprimen los bloques de memoria y se asigna memoria a los procesos utilizando las funciones previamente definidas. Este archivo es crucial para la orquestación del programa y la gestión de sus componentes.

3.5. Archivo 'auxiliar.c'

Este archivo contiene varias funciones auxiliares que facilitan la lectura de datos de entrada, la gestión de procesos en una cola, la impresión del estado de la memoria y la generación del diagrama de Gantt. Estas funciones son esenciales para el funcionamiento del programa, ya que permiten una mejor organización y visualización de los datos.

Este archivo proporciona funciones esenciales para el manejo de la entrada del programa y la gestión de procesos. La función `leer_entrada` abre un archivo y extrae parámetros de configuración, como el tamaño de la memoria y los algoritmos de planificación. Luego, lee los procesos definidos en el archivo y los añade a una cola a través de la función `asignar_valores_procesos`. Además, incluye funciones para imprimir la cola de procesos y el estado de los bloques de memoria. Por último, las funciones `registrar_tiempos` e `imprimir_gantt` permiten gestionar y visualizar el diagrama de Gantt, que es crucial para entender el orden de ejecución de los procesos. Este conjunto de funciones mejora la modularidad y la legibilidad del código.

```

1 void leer_entrada(const char *nombre_archivo, int *tamano_memoria, int *tamano_bloque, char *
   algoritmo_memoria, char *algoritmo_planificacion, Cola *cola_procesos)
2 {
3     FILE *archivo_entrada = fopen(nombre_archivo, "r");
4     if (archivo_entrada == NULL)
5     {
6         fprintf(stderr, "ERROR al abrir el archivo, saliendo...\n");
7         exit(EXIT_FAILURE);
8     }
9     if (fscanf(archivo_entrada, "%d %d %s", tamano_memoria, tamano_bloque, algoritmo_memoria) != 3)
10    {
11        fprintf(stderr, "ERROR al leer los valores de la memoria o bloque, saliendo...\n");
12        exit(EXIT_FAILURE);
13    }
14    if (fscanf(archivo_entrada, "%s", algoritmo_planificacion) != 1)
15    {
16        fprintf(stderr, "Error al leer el algoritmo de planificaci n\n");
17        exit(EXIT_FAILURE);
18    }
19    int id, tiempo_llegada, tiempo_rafaga, memoria_solicitada;
20    while (fscanf(archivo_entrada, "%d %d %d %d", &id, &tiempo_llegada, &tiempo_rafaga, &
   memoria_solicitada) != EOF)
21        asignar_valores_procesos(id, tiempo_llegada, tiempo_rafaga, memoria_solicitada, cola_procesos);
22    fclose(archivo_entrada);
23 }
24
25 void asignar_valores_procesos(int id, int tiempo_llegada, int tiempo_rafaga, int memoria_solicitada,
   Cola *cola_procesos)
26 {
27     Proceso *nuevo_proceso = (Proceso *)malloc(sizeof(Proceso));
28     nuevo_proceso->pid = id;
29     nuevo_proceso->tiempo_llegada = tiempo_llegada;
30     nuevo_proceso->tiempo_rafaga = tiempo_rafaga;
31     nuevo_proceso->memoria_solicitada = memoria_solicitada;
32     nuevo_proceso->next = NULL;
33     fprintf(stdout, "Proceso %d - T. Llegada %d - T. R faga %d - M. Solicitada %d\n", id,
   tiempo_llegada, tiempo_rafaga, memoria_solicitada);
34     enqueue(cola_procesos, nuevo_proceso);
35 }
36
37 void registrar_tiempos(Gantt *diagrama_gantt, int pid, int tiempo_inicio, int tiempo_rafaga, int *indice
   )
38 {
39     diagrama_gantt[*indice].pid = pid;
40     diagrama_gantt[*indice].tiempo_inicio = tiempo_inicio;
41     diagrama_gantt[*indice].tiempo_final = tiempo_inicio + tiempo_rafaga;
42     (*indice)++;
43 }
44
45 void imprimir_gantt(Gantt *diagrama_gantt, int num_procesos)
46 {
47     fprintf(stdout, "Diagrama de Gantt:\n\n");
48     for (int i = 0; i < num_procesos; i++)
49         fprintf(stdout, "[PID: %d]\t[Tiempo Inicio: %d]\t[Tiempo Final: %d]\n", diagrama_gantt[i].pid,
   diagrama_gantt[i].tiempo_inicio, diagrama_gantt[i].tiempo_final);
50 }

```

Código 5. auxiliar.c.

3.6. Archivo 'memoria.c'

Este archivo implementa funciones relacionadas con la gestión de la memoria del sistema. Incluye la inicialización de bloques de memoria, la asignación de memoria a los procesos en función de sus requerimientos, la simulación de la ejecución de estos procesos y la liberación de la memoria una vez que han finalizado. Estas funciones son cruciales para el manejo eficiente de los recursos de memoria del sistema.

En este archivo se definen funciones críticas para la gestión de la memoria del sistema. La función `inicializar_bloques_memoria` se encarga de configurar cada bloque de memoria, estableciendo su tamaño y estado inicial como libre. La función `asignar_memoria_procesos` gestiona la asignación de memoria a los procesos extraídos de la cola, verificando si hay suficiente memoria disponible y manejando la fragmentación interna. Si el proceso puede ser ejecutado, la función `ejecutar_proceso` simula su ejecución, mientras que `liberar_memoria` se utiliza para devolver el bloque de memoria a su estado libre después de completar la ejecución. Estas funciones permiten un manejo eficiente de los recursos de memoria, asegurando que los procesos se ejecuten de manera ordenada y con el uso óptimo del espacio de memoria disponible.

```

1 void inicializar_bloques_memoria(BloqueMemoria *memoria, int cantidad_bloques, int tamano_bloque)
2 {
3     for (int i = 0; i < cantidad_bloques; i++)
4     {
5         memoria[i].tamano = tamano_bloque;
6         memoria[i].estado = 1;
7     }
8 }
9
10 void asignar_memoria_procesos(Cola *cola, BloqueMemoria *memoria, int cantidad_bloques)
11 {
12     Cola *temp_cola = malloc(sizeof(Cola));
13     temp_cola = cola;
14     Proceso *proceso_extraido = malloc(sizeof(Proceso));
15     Gantt diagrama_gantt[MAX_PROCESOS];
16     int indice = 0, tiempo_global = 0;
17
18     while (temp_cola->front != NULL)
19     {
20         proceso_extraido = dequeue(temp_cola);
21         if (proceso_extraido == NULL)
22         {
23             fprintf(stdout, "No hay procesos en la cola para asignar memoria.\n");
24             return;
25         }
26         manejar_proceso(memoria, proceso_extraido, cantidad_bloques, diagrama_gantt, &tiempo_global, &
27         indice);
28         free(proceso_extraido);
29     }
30     imprimir_gantt(diagrama_gantt, indice);
31     generar_archivo_gantt(diagrama_gantt, indice, "gantt.eps");
32 }
33
34 void ejecutar_proceso(BloqueMemoria *memoria, Proceso *proceso, int posicion)
35 {
36     sleep(proceso->tiempo_rafaga);
37     liberar_memoria(memoria, posicion);
38 }
39
40 void liberar_memoria(BloqueMemoria *memoria, int posicion)
41 {
42     memoria[posicion].estado = 1;
43     memoria[posicion].tamano = 128;
44 }

```

Código 6. memoria.c.

3.7. Archivo 'planificador.c'

Este archivo implementa funciones para gestionar la cola de procesos en el sistema. Incluye las funciones `enqueue` y `dequeue`, que permiten agregar y extraer procesos de la cola, respectivamente. Estas funciones son fundamentales para el manejo de la planificación de procesos, asegurando que los procesos se gestionen de manera adecuada y en el orden correcto.

En este archivo se definen las funciones esenciales para manejar la cola de procesos. La función `enqueue` añade un nuevo proceso al final de la cola, actualizando el puntero del frente y del final de la cola según sea necesario. Si la cola está vacía, el nuevo proceso se convierte en el primer elemento. Por otro lado, la función `dequeue` extrae el proceso del frente de la cola y actualiza el puntero del frente, garantizando que el siguiente proceso en la cola esté disponible para su ejecución. Si la cola queda vacía tras la extracción, también se actualiza el puntero del final. Estas funciones son clave para implementar una política de planificación de procesos, como el algoritmo FIFO, asegurando que los procesos se gestionen de manera ordenada y eficiente.


```

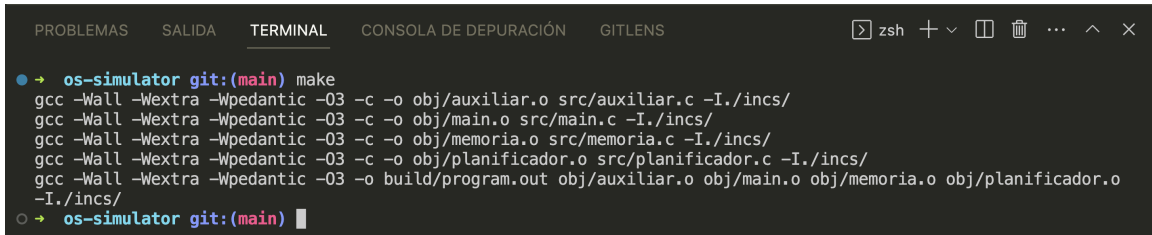
1 void enqueue(Cola *cola, Proceso *proceso)
2 {
3     if (cola->rear == NULL)
4         cola->front = proceso;
5     else
6         cola->rear->next = proceso;
7     cola->rear = proceso;
8     proceso->next = NULL;
9     fprintf(stdout, "Proceso %d encolado CORRECTAMENTE.\n", proceso->pid);
10    imprimir_cola_procesos(cola);
11    fprintf(stdout, "\n");
12 }
13
14 Proceso *dequeue(Cola *cola)
15 {
16     if (cola->front == NULL)
17     {
18         fprintf(stdout, "La cola est vac a.\n");
19         exit(EXIT_FAILURE);
20     }
21     Proceso *proceso_extraido = cola->front;
22     cola->front = cola->front->next;
23     if (cola->front == NULL)
24         cola->rear = NULL;
25     return proceso_extraido;
26 }

```

Código 7. planificador.c.

4. Seguimiento del programa

Para poder ver el funcionamiento del sistema operativo, se deben compilar los archivos, por lo que se hace un *make*, que compilará todos los archivos que usa el sistema operativo.



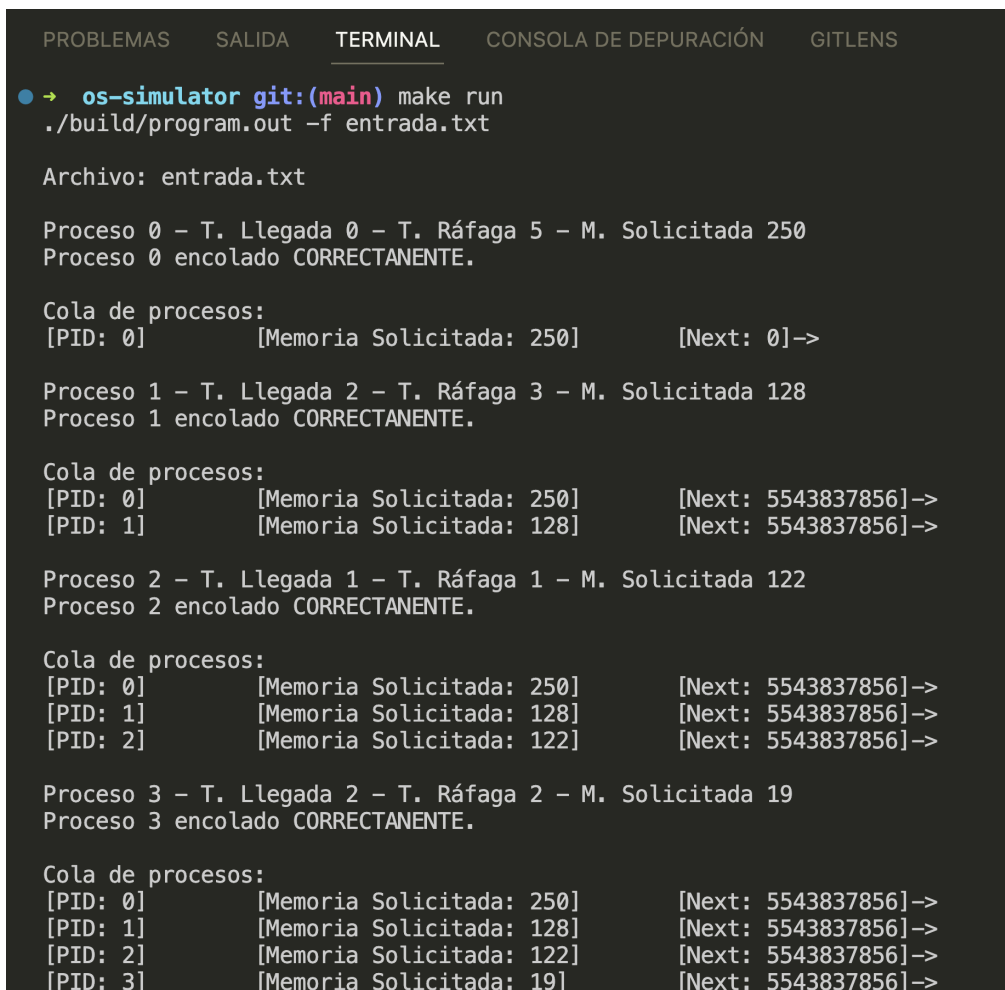
```

PROBLEMAS  SALIDA  TERMINAL  CONSOLA DE DEPURACIÓN  GITLENS
• → os-simulator git:(main) make
gcc -Wall -Wextra -Wpedantic -O3 -c -o obj/auxiliar.o src/auxiliar.c -I./incs/
gcc -Wall -Wextra -Wpedantic -O3 -c -o obj/main.o src/main.c -I./incs/
gcc -Wall -Wextra -Wpedantic -O3 -c -o obj/memoria.o src/memoria.c -I./incs/
gcc -Wall -Wextra -Wpedantic -O3 -c -o obj/planificador.o src/planificador.c -I./incs/
gcc -Wall -Wextra -Wpedantic -O3 -o build/program.out obj/auxiliar.o obj/main.o obj/memoria.o obj/planificador.o
-I./incs/
○ → os-simulator git:(main)

```

Figura 2. Compilación de los archivos.

Luego se ejecuta el programa usando el comando *make run*, lo que permite leer la entrada, los procesos se encolan en el orden de llegada (*esto es fundamental para luego poder implementar FIFO*).



```

PROBLEMAS  SALIDA  TERMINAL  CONSOLA DE DEPURACIÓN  GITLENS
• → os-simulator git:(main) make run
./build/program.out -f entrada.txt

Archivo: entrada.txt

Proceso 0 - T. Llegada 0 - T. Ráfaga 5 - M. Solicitada 250
Proceso 0 encolado CORRECTAMENTE.

Cola de procesos:
[PID: 0]      [Memoria Solicitada: 250]      [Next: 0]→

Proceso 1 - T. Llegada 2 - T. Ráfaga 3 - M. Solicitada 128
Proceso 1 encolado CORRECTAMENTE.

Cola de procesos:
[PID: 0]      [Memoria Solicitada: 250]      [Next: 5543837856]→
[PID: 1]      [Memoria Solicitada: 128]      [Next: 5543837856]→

Proceso 2 - T. Llegada 1 - T. Ráfaga 1 - M. Solicitada 122
Proceso 2 encolado CORRECTAMENTE.

Cola de procesos:
[PID: 0]      [Memoria Solicitada: 250]      [Next: 5543837856]→
[PID: 1]      [Memoria Solicitada: 128]      [Next: 5543837856]→
[PID: 2]      [Memoria Solicitada: 122]      [Next: 5543837856]→

Proceso 3 - T. Llegada 2 - T. Ráfaga 2 - M. Solicitada 19
Proceso 3 encolado CORRECTAMENTE.

Cola de procesos:
[PID: 0]      [Memoria Solicitada: 250]      [Next: 5543837856]→
[PID: 1]      [Memoria Solicitada: 128]      [Next: 5543837856]→
[PID: 2]      [Memoria Solicitada: 122]      [Next: 5543837856]→
[PID: 3]      [Memoria Solicitada: 19]       [Next: 5543837856]→

```

Figura 3. Leída de procesos.

A cada proceso de la cola, se le asigna memoria dependiendo de la memoria necesaria, si necesita mas de 1 bloque, se piden los bloques necesarios para que el proceso pueda ser ejecutado, en el caso de que no exista memoria disponible, se indicara por consola.

```

PROBLEMAS  SALIDA  TERMINAL  CONSOLA DE DEPURACIÓN  GITLENS

Bloques de memoria del SISTEMA OPERATIVO:

[Bloque 0]      [Tamaño 128]    [Estado 1] [LIBRE]
[Bloque 1]      [Tamaño 128]    [Estado 1] [LIBRE]
[Bloque 2]      [Tamaño 128]    [Estado 1] [LIBRE]
[Bloque 3]      [Tamaño 128]    [Estado 1] [LIBRE]
[Bloque 4]      [Tamaño 128]    [Estado 1] [LIBRE]
[Bloque 5]      [Tamaño 128]    [Estado 1] [LIBRE]
[Bloque 6]      [Tamaño 128]    [Estado 1] [LIBRE]
[Bloque 7]      [Tamaño 128]    [Estado 1] [LIBRE]
[Bloque 8]      [Tamaño 128]    [Estado 1] [LIBRE]
[Bloque 9]      [Tamaño 128]    [Estado 1] [LIBRE]
[Bloque 10]     [Tamaño 128]    [Estado 1] [LIBRE]
[Bloque 11]     [Tamaño 128]    [Estado 1] [LIBRE]
[Bloque 12]     [Tamaño 128]    [Estado 1] [LIBRE]
[Bloque 13]     [Tamaño 128]    [Estado 1] [LIBRE]
[Bloque 14]     [Tamaño 128]    [Estado 1] [LIBRE]
[Bloque 15]     [Tamaño 128]    [Estado 1] [LIBRE]

Proceso 0 en EJECUCIÓN, bloque (0)
Proceso 0 EJECUTADO.
El proceso 0 hizo uso de MEMORIA hasta el BLOQUE 1
FRAGMENTACIÓN INTERNA: 6 KB en el BLOQUE 1
Proceso 0 EJECUTADO EXITOSAMENTE.

El proceso 1 hizo uso de MEMORIA hasta el BLOQUE 0
Proceso 1 EJECUTADO EXITOSAMENTE.

El proceso 2 hizo uso de MEMORIA hasta el BLOQUE 0
FRAGMENTACIÓN INTERNA: 6 KB en el BLOQUE 0
Proceso 2 EJECUTADO EXITOSAMENTE.

El proceso 3 hizo uso de MEMORIA hasta el BLOQUE 0
FRAGMENTACIÓN INTERNA: 109 KB en el BLOQUE 0

```

Figura 4. Asignación de memoria del sistema operativo.

Luego los procesos comienzan a ser ejecutados, liberando así el bloque que anteriormente se estaba usando, dejando el espacio libre para otro proceso, estos procesos se ejecutan en el orden de llegada.

```

PROBLEMAS  SALIDA  TERMINAL  CONSOLA DE DEPURACIÓN  GITLENS

Proceso 0 en EJECUCIÓN, bloque (0)
Proceso 0 EJECUTADO.
El proceso 0 hizo uso de MEMORIA hasta el BLOQUE 1
FRAGMENTACIÓN INTERNA: 6 KB en el BLOQUE 1
Proceso 0 EJECUTADO EXITOSAMENTE.

El proceso 1 hizo uso de MEMORIA hasta el BLOQUE 0
Proceso 1 EJECUTADO EXITOSAMENTE.

El proceso 2 hizo uso de MEMORIA hasta el BLOQUE 0
FRAGMENTACIÓN INTERNA: 6 KB en el BLOQUE 0
Proceso 2 EJECUTADO EXITOSAMENTE.

El proceso 3 hizo uso de MEMORIA hasta el BLOQUE 0
FRAGMENTACIÓN INTERNA: 109 KB en el BLOQUE 0
Proceso 3 EJECUTADO EXITOSAMENTE.

El proceso 4 hizo uso de MEMORIA hasta el BLOQUE 0
FRAGMENTACIÓN INTERNA: 30 KB en el BLOQUE 0
Proceso 4 EJECUTADO EXITOSAMENTE.

Proceso 5 en EJECUCIÓN, bloque (0)
Proceso 5 EJECUTADO.
Proceso 5 en EJECUCIÓN, bloque (1)
Proceso 5 EJECUTADO.
Proceso 5 en EJECUCIÓN, bloque (2)
Proceso 5 EJECUTADO.
El proceso 5 hizo uso de MEMORIA hasta el BLOQUE 3
FRAGMENTACIÓN INTERNA: 53 KB en el BLOQUE 3
Proceso 5 EJECUTADO EXITOSAMENTE.

No hay suficiente memoria disponible para asignar al proceso 6, pasando al siguiente PROCESO.

El proceso 7 hizo uso de MEMORIA hasta el BLOQUE 0
Proceso 7 EJECUTADO EXITOSAMENTE.

```

Figura 5. Se ejecutan los procesos utilizando FIFO.

Cuando todos los procesos han sido ejecutados, se crea una carta gantt de salida, con toda la información acerca de la planificación.

```

Diagrama de Gantt:

[PID: 0]      [Tiempo Inicio: 0]      [Tiempo Final: 5]
[PID: 1]      [Tiempo Inicio: 5]      [Tiempo Final: 8]
[PID: 2]      [Tiempo Inicio: 8]      [Tiempo Final: 9]
[PID: 3]      [Tiempo Inicio: 9]      [Tiempo Final: 11]
[PID: 4]      [Tiempo Inicio: 11]     [Tiempo Final: 14]
[PID: 5]      [Tiempo Inicio: 14]     [Tiempo Final: 20]
[PID: 7]      [Tiempo Inicio: 20]     [Tiempo Final: 28]

CARTA GANTT generada EXITOSAMENTE en el ARCHIVO gantt.eps

→ os-simulator git:(main)

```

Figura 6. Se crea una carta gantt de salida.

Ademas, se crea un .EPS que tiene una representación grafica de la salida del sistema operativo.

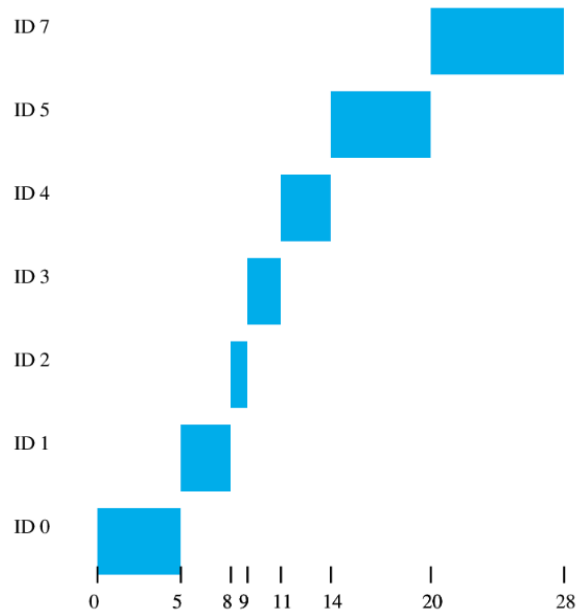


Figura 7. Vista del archivo .EPS.

En todo este flujo de asignación y ejecución, los procesos se ejecutan utilizando el método FIFO. Se valida el tamaño de los procesos para asegurarse de que no excedan el limite del bloque. Si un proceso requiere más memoria de la que el bloque puede proporcionar, se solicita memoria al bloque contiguo. Si no hay más memoria disponible, el proceso no puede ejecutarse. Además, se maneja la fragmentación mostrando cuántos KB de memoria quedan entre bloques.

5. Pruebas y resultados

5.1. ¿Como nuestro programa maneja la salida de los datos?

Nuestro programa toma el archivo de entrada con detalles de los procesos, como tiempos de llegada, ráfagas de CPU y memoria solicitada, y los encola siguiendo el algoritmo FIFO. Cada proceso se coloca en la cola de procesos en función de su tiempo de llegada, y se asegura que el primer proceso en llegar sea el primero en ejecutarse.

Posteriormente, el programa asigna la memoria mediante bloques de memoria de tamaño fijo y, en caso de que un proceso no use completamente un bloque, se genera fragmentación interna. A lo largo de la ejecución, el programa calcula y reporta dicha fragmentación interna, optimizando el uso de los recursos.

En el caso de no haber suficiente memoria disponible para un proceso, este es saltado, y el programa continúa con el siguiente proceso en la cola. Esta estrategia asegura que el sistema siga funcionando de manera eficiente, incluso cuando los recursos de memoria son limitados. Adicionalmente, el programa reporta las condiciones de fallo de memoria, lo que permite al usuario entender el por qué ciertos procesos no pudieron ejecutarse.

Finalmente, se genera un **Diagrama de Gantt** que ilustra visualmente el tiempo de ejecución de cada proceso, lo que facilita el análisis de rendimiento del sistema.

5.2. Salida del programa

Al ejecutar el programa con el archivo *entrada.txt*, la salida indica el estado de los procesos encolados y los bloques de memoria asignados. A continuación, se muestra cuando ingresa el primer proceso a la cola:

```
Proceso 0 - T. Llegada 0 - T. Ráfaga 5 - M. Solicitada 250
Proceso 0 encolado CORRECTAMENTE.
Cola de procesos:
[PID: 0] [Memoria Solicitada: 250] [Next: 0]->
```

Una vez que se ingrese el último proceso a la cola, se muestra de la siguiente manera:

```
Proceso 7 - T. Llegada 8 - T. Ráfaga 8 - M. Solicitada 128
Proceso 7 encolado CORRECTAMENTE.
Cola de procesos:
[PID: 0] [Memoria Solicitada: 250] [Next: 7017888]->
[PID: 1] [Memoria Solicitada: 128] [Next: 7017888]->
[PID: 2] [Memoria Solicitada: 122] [Next: 7017888]->
[PID: 3] [Memoria Solicitada: 19] [Next: 7017888]->
[PID: 4] [Memoria Solicitada: 98] [Next: 7017888]->
[PID: 5] [Memoria Solicitada: 459] [Next: 7017888]->
[PID: 6] [Memoria Solicitada: 2049] [Next: 7017888]->
[PID: 7] [Memoria Solicitada: 128] [Next: 7017888]->
```

Además, el programa gestiona la ejecución de cada proceso, indicando la fragmentación interna y si el proceso fue ejecutado exitosamente. A continuación se muestra el manejo de memoria y fragmentación del proceso 0:

```
El proceso 0 usó MEMORIA hasta el BLOQUE 1
FRAGMENTACIÓN INTERNA: 6 KB en el BLOQUE 1
Proceso 0 EJECUTADO EXITOSAMENTE
```

El manejo de la memoria es un aspecto clave del programa, ya que utiliza bloques de tamaño fijo para asignar memoria a cada proceso. Cuando un proceso no llena completamente un bloque de memoria, se genera fragmentación interna, como se muestra en el proceso 0, donde 6 KB no fueron utilizados. A medida que más procesos solicitan memoria, la fragmentación puede aumentar, reduciendo la disponibilidad de memoria para nuevos procesos y, a largo plazo, afectando la eficiencia general del sistema.

Cuando no hay suficiente memoria disponible para un proceso, como ocurrió en el caso del proceso 6, el programa lo omite, manteniendo la integridad del sistema y evitando problemas de sobrecarga. Esto es crucial en sistemas con recursos limitados, ya que permite que otros procesos continúen ejecutándose sin interrupciones. A continuación se muestra la salida cuando se intenta asignar memoria al proceso 6:

No hay suficiente memoria disponible para asignar al proceso 6, pasando al siguiente PROCESO.

Los procesos que no pueden ser ejecutados por falta de recursos no interrumpen la ejecución de otros procesos que tienen suficientes recursos disponibles.

En cuanto al rendimiento, el programa es eficiente en su manejo de la cola de proceso gracias al uso del algoritmo FIFO. Los procesos son ejecutados en el orden en que llegan, asegurando que no haya procesos ignorados o ejecutados fuera de turno. La estructura FIFO garantiza que los procesos se encolen y desencolen correctamente, como se observa con el proceso 0, que es el primero en ejecutarse.

Finalmente, se muestra un Diagrama de Gantt que detalla el tiempo de inicio y finalización de cada proceso, proporcionando un análisis del rendimiento del sistema. Este diagrama refleja que el programa gestiona de manera eficiente los tiempos de ejecución de procesos y asegura que los tiempos sean respetados para cada proceso en la cola:

Diagrama de Gantt:

```
[PID: 0] [Tiempo Inicio: 0] [Tiempo Final: 5]
[PID: 1] [Tiempo Inicio: 5] [Tiempo Final: 8]
[PID: 2] [Tiempo Inicio: 8] [Tiempo Final: 9]
[PID: 3] [Tiempo Inicio: 9] [Tiempo Final: 11]
[PID: 4] [Tiempo Inicio: 11] [Tiempo Final: 14]
[PID: 5] [Tiempo Inicio: 14] [Tiempo Final: 20]
[PID: 7] [Tiempo Inicio: 20] [Tiempo Final: 28]
```

Lo anterior muestra que la duración total de ejecución es de 28 unidades de tiempo, y los tiempos de inicio y finalización de cada proceso se respetan, lo que asegura un manejo eficaz del tiempo de CPU.

El sistema ejecuta los procesos sin mayor dificultad, lo que indica una gestión adecuada de los recursos disponibles.

PD: Revisa esto please, a lo que está dentro de *"beginrhoenv"* mi intención era colocarlo como foto pero a mi me aparece muy feo, lo podrías poner tú y así seguir el mismo formato de fotos.

6. Dificultades y soluciones

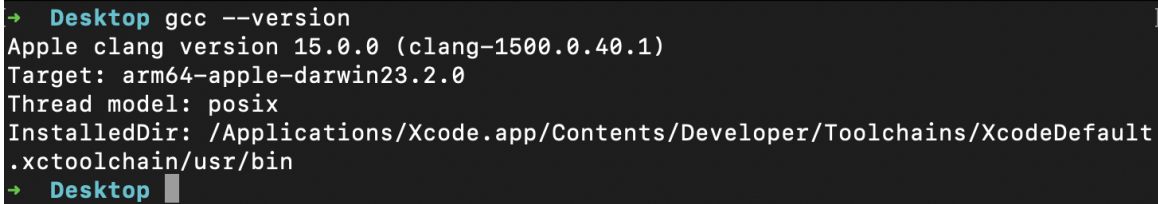
6.1. ¿Qué problemas enfrentamos?

Durante el desarrollo del simulador, nos encontramos con diversos problemas que generaron bloqueos temporales. Algunos de estos estuvieron relacionados con accesos fuera del heap asignado, lo que provocó errores de memoria, mientras que otros problemas surgieron por el manejo incorrecto del stack, causando situaciones de corrupción de pila (*stack smashing*).

6.2. Problemas con el compilador

Cada miembro del equipo trabajaba en su propio computador personal, utilizando diferentes sistemas operativos: macOS, Windows y Linux. Un problema notable surgió en macOS, donde, al instalar *gcc* a través del gestor de paquetes *Homebrew*, se instaló *Clang* en lugar de *GCC*.

Esta diferencia se hizo evidente durante las sesiones de *pair programming*, ya que se mostraban advertencias (*warnings*) diferentes dependiendo del compilador usado. El problema fue identificado cuando se verificó la versión del compilador de uno de los miembros, lo que permitió descubrir que uno de los sistemas estaba utilizando *Clang* en lugar de *GCC*.



```
→ Desktop gcc --version
Apple clang version 15.0.0 (clang-1500.0.40.1)
Target: arm64-apple-darwin23.2.0
Thread model: posix
InstalledDir: /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin
→ Desktop █
```

Figura 8. Uso de clang.

6.3. Soluciones a los problemas

Para resolver los problemas, tomamos las siguientes acciones:

1. Se revisaron cuidadosamente los accesos a memoria y el manejo del stack para evitar errores como los mencionados.
2. En cuanto a las diferencias de compiladores, se ajustaron los entornos de desarrollo para que todos los miembros del equipo utilizaran el mismo compilador.