os-simulator

Generated by Doxygen 1.13.0

Chapter 1

Estructura de Datos: Simulador Sistema Operativo

Integrantes:

- Miguel Loaiza (miloaiza@umag.cl) [https://github.com/EhMigueh]
- Diego Sanhueza (disanhue@umag.cl) [https://github.com/Diego0119]
- Oscar Cifuentes (ocifuent@umag.cl) [https://github.com/iBluZiiZ]

1.1 Descripción

Este proyecto es un Simulador de Sistemas Operativos, diseñado para emular el comportamiento de un sistema operativo básico. A través de esta simulación, puedes explorar cómo se manejan procesos, memoria y otros recursos del sistema. Nuestro enfoque es educativo, para ayudar a estudiantes a entender los conceptos fundamentales detrás de los sistemas operativos.

1.2 Funcionamiento

Para poder ejecutar el proyecto de manera satisfactoria, realice los siguientes pasos.

1. En caso de que falten carpetas (src, obj, incs, docs o build), ejecute el comando 'make folders' en la terminal.
2. Asegurece de que los archivos '.c' estén ubicados correctamente en la carpeta 'src' y que el archivo '.h' esté en la carpeta 'incs'.
3. Se recomienda limpiar los archivos '.o' con el comando 'make clean' antes de compilar.
4. Ejecute el comando 'make' en su terminal para poder compilar los archivos '.c' y crear los archivos '.o'.
5. Ejecute el comando 'make run' en su terminal para poder comenzar con la prueba del programa Simulador Sistema Operativo.
6. En la carpera 'os-simulator' se encuentran dos archivos (.eps y .png) que contienen la carta gantt de los procesos ejecutados con el programa.
7. En caso de querer eliminar los archivos '.o', ejecute el comando 'make clean' en su terminal para limpiar las carpetas.

1.3 Documentación

Antes de generar la documentación, por favor vaya a la carpeta docs para verificar si ya existe la documentación. Para poder generar la documentación del proyecto, realice los siguientes pasos.

1. Asegurece de que esté el archivo 'config' en la carpeta 'os-simulator'.
2. Ejecute el comando 'doxygen config' en su terminal para poder ejecutar el archivo config y generar la documentación.
3. En la carpeta docs, se encuentra la documentación en html y pdf.

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

Class Documentation

4.1 BloqueMemoria Struct Reference

Estructura de un BLOQUE de MEMORIA.

`#include <memoria.h>`

Public Attributes

- int tamano
- int estado

4.1.1 Detailed Description

Estructura de un BLOQUE de MEMORIA.

```
typedef struct
{
    int tamano;
    int estado;
} BloqueMemoria;
```

En esta estructura se define un BLOQUE de MEMORIA con un tamaño y un estado (1 cuando está LIBRE y 0 cuando está OCUPADO).

Definition at line 30 of file memoria.h.

4.1.2 Member Data Documentation

4.1.2.1 estado

`int BloqueMemoria::estado`

Definition at line 33 of file memoria.h.

4.1.2.2 tamano

```
int BloqueMemoria::tamano
```

Definition at line 32 of file memoria.h.

The documentation for this struct was generated from the following file:

- incs/memoria.h

4.2 Cola Struct Reference

Estructura de una COLA.

```
#include <planificador.h>
```

Public Attributes

- Proceso * front
- Proceso * rear

4.2.1 Detailed Description

Estructura de una COLA.

```
typedef struct
{
    Proceso *front;
    Proceso *rear;
} Cola;
```

En esta estructura se define una COLA con un FRENTE y un FINAL.

Definition at line 56 of file planificador.h.

4.2.2 Member Data Documentation

4.2.2.1 front

```
Proceso* Cola::front
```

Definition at line 58 of file planificador.h.

4.2.2.2 rear

```
Proceso* Cola::rear
```

Definition at line 59 of file planificador.h.

The documentation for this struct was generated from the following file:

- incs/planificador.h

4.3 Gantt Struct Reference

Estructura de un DIAGRAMA de GANTT.

`#include <planificador.h>`

Public Attributes

- int pid
- int tiempo_inicio
- int tiempo_final

4.3.1 Detailed Description

Estructura de un DIAGRAMA de GANTT.

```
typedef struct
{
    int pid;
    int tiempo_inicio;
    int tiempo_final;
} Gantt;
```

En esta estructura se define un DIAGRAMA de GANTT con un ID de proceso, tiempo de inicio y tiempo final.

Definition at line 76 of file planificador.h.

4.3.2 Member Data Documentation

4.3.2.1 pid

`int Gantt::pid`

Definition at line 78 of file planificador.h.

4.3.2.2 tiempo_final

`int Gantt::tiempo_final`

Definition at line 80 of file planificador.h.

4.3.2.3 tiempo_inicio

`int Gantt::tiempo_inicio`

Definition at line 79 of file planificador.h.

The documentation for this struct was generated from the following file:

- incs/planificador.h

4.4 Proceso Struct Reference

Estructura de un PROCESO.

`#include <planificador.h>`

Public Attributes

- int pid
- int tiempo_llegada
- int tiempo_rafaga
- int memoria_solicitada
- struct Proceso * next

4.4.1 Detailed Description

Estructura de un PROCESO.

```
typedef struct Proceso
{
    int pid;
    int tiempo_llegada;
    int tiempo_rafaga;
    int memoria_solicitada;
    struct Proceso *next;
} Proceso;
```

En esta estructura se define un PROCESO con un ID, tiempo de llegada, tiempo de ráfaga, memoria solicitada y un puntero al siguiente PROCESO.

Definition at line 34 of file planificador.h.

4.4.2 Member Data Documentation

4.4.2.1 memoria_solicitada

`int Proceso::memoria_solicitada`

Definition at line 39 of file planificador.h.

4.4.2.2 next

`struct Proceso* Proceso::next`

Definition at line 40 of file planificador.h.

4.4.2.3 pid

`int Proceso::pid`

Definition at line 36 of file planificador.h.

4.4.2.4 tiempo_llegada

```
int Proceso::tiempo_llegada
```

Definition at line 37 of file planificador.h.

4.4.2.5 tiempo_rafaga

```
int Proceso::tiempo_rafaga
```

Definition at line 38 of file planificador.h.

The documentation for this struct was generated from the following file:

- incs/planificador.h

Chapter 5

File Documentation

5.1 incs/auxiliar.h File Reference

Prototipos de funciones auxiliares.

```
#include "memoria.h"
```

Functions

- void leer_entrada (const char *, int *, int *, char *, char *, Cola *)
  < *Librería que contiene las funciones de MEMORIA y la estructra del BLOQUE de MEMORIA.*
- void asignar_valores_procesos (int, int, int, int, Cola *)
  Crea un nuevo PROCESO y le asigna los VALORES.
- void imprimir_cola_procesos (Cola *)
  Imprime la COLA de PROCESOS.
- void imprimir_memoria (BloqueMemoria *, int)
  Imprime los BLOQUES de MEMORIA INICIALIZADOS.
- void registrar_tiempos (Gantt *, int, int, int, int *)
  Registra los TIEMPOS de los PROCESOS.
- void imprimir_gantt (Gantt *, int)
  Imprime el DIAGRAMA de GANTT.
- void generar_archivo_gantt (Gantt *, int, const char *)
  Genera el ARCHIVO EPS que muestra la CARTA GANTT.

5.1.1 Detailed Description

Prototipos de funciones auxiliares.

Date

24-10-2024

Authors

Miguel Loaiza, Diego Sanhueza y Oscar Cifuentes

Contiene los prototipos de las funciones auxiliares que son utilizadas para el correcto funcionamiento de los archivos de MEMORIA y PLANIFICACIÓN.

Definition in file auxiliar.h.

5.1.2 Function Documentation

5.1.2.1 asignar_valores_procesos()

```
void asignar_valores_procesos (
            int id,
            int tiempo_llegada,
            int tiempo_rafaga,
            int memoria_solicitada,
            Cola * cola_procesos)
```

Crea un nuevo PROCESO y le asigna los VALORES.

Parameters

id	ID del proceso.
tiempo_llegada	Tiempo de llegada del proceso.
tiempo_rafaga	Tiempo de ráfaga del proceso.
memoria_solicitada	Memoria solicitada por el proceso.
cola_procesos	Cola de PROCESOS.

Se CREA un NUEVO PROCESO y se ASIGNAN los VALORES necesarios, luego se llama a la función "enqueue" para colocar en la COLA el PROCESO.

```
Proceso *nuevo_proceso = (Proceso *)malloc(sizeof(Proceso));
nuevo_proceso->pid = id;
nuevo_proceso->tiempo_llegada = tiempo_llegada;
nuevo_proceso->tiempo_rafaga = tiempo_rafaga;
nuevo_proceso->memoria_solicitada = memoria_solicitada;
nuevo_proceso->next = NULL;
fprintf(stdout, "Proceso %d - T. Llegada %d - T. Ráfaga %d - M. Solicitada %d\n", id, tiempo_llegada,
      tiempo_rafaga, memoria_solicitada);
enqueue(cola_procesos, nuevo_proceso);
```

Definition at line 91 of file auxiliar.c.

5.1.2.2 generar_archivo_gantt()

```
void generar_archivo_gantt (
            Gantt * diagrama_gantt,
            int num_procesos,
            const char * carta_gantt)
```

Genera el ARCHIVO EPS que muestra la CARTA GANTT.

Genera un ARCHIVO que muestra la CARTA GANTT.

Parameters

diagrama_gantt	Diagrama de Gantt.
num_procesos	Número de procesos.
carta_gantt	Nombre del archivo de la carta Gantt.

Genera el ARCHIVO EPS que muestra la CARTA GANTT.

Genera un ARCHIVO que muestra la CARTA GANTT.

Parameters

diagrama_gantt	Diagrama de Gantt.
num_procesos	Número de procesos.
carta_gantt	Nombre del archivo de la carta Gantt.

Se abre el archivo de GANTT en modo ESCRITURA y se VERIFICA si se abrió correctamente.

```
FILE *archivo_gantt = fopen(carta_gantt, "w");
if (archivo_gantt == NULL)
{
    fprintf(stderr, "ERROR al abrir el archivo de Gantt, saliendo...\n");
    exit(EXIT_FAILURE);
}
```

VARIABLES para DIBUJAR la CARTA GANTT. Se DIBUJAN los RECTÁNGULOS de los PROCESOS y se COLOREAN, se PINTA el BORDE del RECTÁNGULO y se ESCRIBE el ID del PROCESO en el eje Y.

```
int altura_proceso = 50;
int margen_inferior = 100, margen_superior = 50, margen_lateral = 100;
int ancho_total = 800, altura_total = (num_procesos * altura_proceso) + margen_superior + margen_inferior;
fprintf(archivo_gantt, "%%!PS-Adobe-3.0 EPSF-3.0\n");
fprintf(archivo_gantt, "%%%%BoundingBox: 0 0 %d %d\n", ancho_total + margen_lateral * 2, altura_total);
for (int i = 0; i < num_procesos; i++)
{
    int tiempo_inicio = diagrama_gantt[i].tiempo_inicio;
    int tiempo_final = diagrama_gantt[i].tiempo_final;
    int pid = diagrama_gantt[i].pid;
    int x_inicio = margen_lateral + tiempo_inicio * 10;
    int x_final = margen_lateral + tiempo_final * 10;
    int y = margen_inferior + i * altura_proceso;
    fprintf(archivo_gantt, "newpath\n");
    fprintf(archivo_gantt, "%d %d moveto\n", x_inicio, y);
    fprintf(archivo_gantt, "%d %d lineto\n", x_final, y);
    fprintf(archivo_gantt, "%d %d lineto\n", x_final, y + altura_proceso - 10);
    fprintf(archivo_gantt, "%d %d lineto\n", x_inicio, y + altura_proceso - 10);
    fprintf(archivo_gantt, "closepath\n");
    fprintf(archivo_gantt, "0.207 0.706 0.941 setrgbcolor\n");
    fprintf(archivo_gantt, "fill\n");
    fprintf(archivo_gantt, "0 0 0 setrgbcolor\n");
    fprintf(archivo_gantt, "stroke\n");
    fprintf(archivo_gantt, "/Times-Roman findfont 12 scalefont setfont\n");
    fprintf(archivo_gantt, "%d %d moveto\n", margen_lateral - 50, y + altura_proceso / 2);
    fprintf(archivo_gantt, "(ID %d) show\n", pid);
}
```

DIBUJA el eje X con los TIEMPOS REALES, se recorre el DIAGRAMA de GANTT y se imprime el TIEMPO de INICIO y FINAL del PROCESO en el eje X.

```
fprintf(archivo_gantt, "/Times-Roman findfont 12 scalefont setfont\n");
for (int i = 0; i < num_procesos; i++)
{
    int tiempo_inicio = diagrama_gantt[i].tiempo_inicio;
    int tiempo_final = diagrama_gantt[i].tiempo_final;
    int x_inicio = margen_lateral + tiempo_inicio * 10;
    fprintf(archivo_gantt, "newpath\n");
    fprintf(archivo_gantt, "%d %d moveto\n", x_inicio, margen_inferior - 5);
    fprintf(archivo_gantt, "%d %d lineto\n", x_inicio, margen_inferior + 5);
    fprintf(archivo_gantt, "stroke\n");
    fprintf(archivo_gantt, "%d %d moveto\n", x_inicio - 5, margen_inferior - 20);
    fprintf(archivo_gantt, "(%d) show\n", tiempo_inicio);
    int x_final = margen_lateral + tiempo_final * 10;
    fprintf(archivo_gantt, "newpath\n");
    fprintf(archivo_gantt, "%d %d moveto\n", x_final, margen_inferior - 5);
    fprintf(archivo_gantt, "%d %d lineto\n", x_final, margen_inferior + 5);
    fprintf(archivo_gantt, "stroke\n");
    fprintf(archivo_gantt, "%d %d moveto\n", x_final - 5, margen_inferior - 20);
    fprintf(archivo_gantt, "(%d) show\n", tiempo_final);
}
fprintf(archivo_gantt, "showpage\n");
fclose(archivo_gantt);
fprintf(stdout, "\nCARTA GANTT generada EXITOSAMENTE en el ARCHIVO %s\n\n", carta_gantt);
```

Se CONVIERTE el ARCHIVO EPS a PNG con GHOSTSCRIPT.

```
char comando[256];
snprintf(comando, sizeof(comando), "gs -dSAFER -dBATCH -dNOPAUSE -dEPSCrop -sDEVICE=png16m -r300
      -sOutputFile=%s.png %s", carta_gantt, carta_gantt);
int resultado = system(comando);
if (resultado != 0)
{
    fprintf(stderr, "ERROR al convertir la CARTA GANTT a PNG, saliendo...\n");
```

```
    exit(EXIT_FAILURE);
}
fprintf(stdout, "\nCARTA GANTT convertida a PNG en el archivo %s.png\n\n", carta_gantt);
```

Definition at line 218 of file auxiliar.c.

5.1.2.3 imprimir_cola_procesos()

```
void imprimir_cola_procesos (
            Cola * cola)
```

Imprime la COLA de PROCESOS.

Parameters

cola	Cola de PROCESOS.

Se IMPRIME la COLA de PROCESOS, se recorre la COLA y se imprime PROCESO actual, luego se actualiza el PROCESO actual.

```
Proceso *actual = cola->front;
fprintf(stdout, "\nCola de procesos:\n");
long actual_next = (long)actual->next;
while (actual != NULL)
{
    fprintf(stdout, "[Pid: %d]\t[Memoria Solicitada: %d]\t[Next: %ld]->\n", actual->pid,
      actual->memoria_solicitada, actual_next);
    actual = actual->next;
}
```

Definition at line 120 of file auxiliar.c.

5.1.2.4 imprimir_gantt()

```
void imprimir_gantt (
            Gantt * diagrama_gantt,
            int num_procesos)
```

Imprime el DIAGRAMA de GANTT.

Parameters

diagrama_gantt	Diagrama de Gantt.
num_procesos	Número de procesos.

Se IMPRIME el DIAGRAMA de GANTT, se recorre el DIAGRAMA de GANTT y se imprime el PROCESO actual.

```
fprintf(stdout, "Diagrama de Gantt:\n\n");
for (int i = 0; i < num_procesos; i++)
    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);
```

Definition at line 197 of file auxiliar.c.

5.1.2.5 imprimir_memoria()

```
void imprimir_memoria (
            BloqueMemoria * memoria,
            int cantidad_bloques)
```

Imprime los BLOQUES de MEMORIA INICIALIZADOS.

Parameters

memoria	Arreglo de BLOQUES de MEMORIA.
cantidad_bloques	Cantidad de BLOQUES de MEMORIA.

Se IMPRIME los BLOQUES de MEMORIA INICIALIZADOS, se recorre el ARREGLO de MEMORIA y se imprime el BLOQUE actual.

```
fprintf(stdout, "Bloques de memoria del SISTEMA OPERATIVO:\n\n");
for (int i = 0; i < cantidad_bloques; i++)
    fprintf(stdout, "[Bloque %d]\t[Tamaño %d]\t[Estado %d][LIBRE]\n", i, memoria[i].tamano,
      memoria[i].estado);
fprintf(stdout, "\n");
```

Definition at line 150 of file auxiliar.c.

5.1.2.6 leer_entrada()

```
void leer_entrada (
            const char * nombre_archivo,
            int * tamano_memoria,
            int * tamano_bloque,
            char * algoritmo_memoria,
            char * algoritmo_planificacion,
            Cola * cola_procesos)
```

< Librería que contiene las funciones de MEMORIA y la estructra del BLOQUE de MEMORIA.

Lee la entrada del archivo de texto (entrada.txt).

Parameters

nombre_archivo	Nombre del archivo de texto.
tamano_memoria	Tamaño de la memoria TOTAL (2048 KB).
tamano_bloque	Tamaño de los BLOQUES de MEMORIA (128 KB).
algoritmo_memoria	Algoritmo de MEMORIA (FF).
algoritmo_planificacion	Algoritmo de PLANIFICACIÓN (FIFO).
cola_procesos	Cola de PROCESOS.

< Librería que contiene las funciones de MEMORIA y la estructra del BLOQUE de MEMORIA.

Lee la entrada del archivo de texto (entrada.txt).

Parameters

nombre_archivo	Nombre del archivo de texto.
tamano_memoria	Tamaño de la memoria TOTAL (2048 KB).
tamano_bloque	Tamaño de los BLOQUES de MEMORIA (128 KB).
algoritmo_memoria	Algoritmo de MEMORIA (FF).
algoritmo_planificacion	Algoritmo de PLANIFICACIÓN (FIFO).
cola_procesos	Cola de PROCESOS.

Se abre el archivo de entrada en modo LECTURA y se VERIFICA si se abrió correctamente.

```
FILE *archivo_entrada = fopen(nombre_archivo, "r");
if (archivo_entrada == NULL)
{
    fprintf(stderr, "ERROR al abrir el archivo, saliendo...\n");
    exit(EXIT_FAILURE);
}
```

Lee la cabecera del archivo de entrada (2048 128 ff) y VERIFICA si los valores son 3, también lee el algoritmo de planificación (FIFO) y verifica si el valor es 1.

```
if (fscanf(archivo_entrada, "%d %d %s", tamano_memoria, tamano_bloque, algoritmo_memoria) != 3)
{
    fprintf(stderr, "ERROR al leer los valores de la memoria o bloque, saliendo...\n");
    exit(EXIT_FAILURE);
}

if (fscanf(archivo_entrada, "%s", algoritmo_planificacion) != 1)
{
    fprintf(stderr, "Error al leer el algoritmo de planificación\n");
    exit(EXIT_FAILURE);
}
```

VARIABLES del PROCESO (vienen de entrada.txt). Lee los VALORES de los PROCESOS del archivo de entrada y llama la función "asignar_valores_procesos", luego cierra el archivo.

```
int id, tiempo_llegada, tiempo_rafaga, memoria_solicitada;
while (fscanf(archivo_entrada, "%d %d %d %d", &id, &tiempo_llegada, &tiempo_rafaga, &memoria_solicitada) !=
      EOF)
    asignar_valores_procesos(id, tiempo_llegada, tiempo_rafaga, memoria_solicitada, cola_procesos);
fclose(archivo_entrada);
```

Definition at line 21 of file auxiliar.c.

5.1.2.7 registrar_tiempos()

```
void registrar_tiempos (
            Gantt * diagrama_gantt,
            int pid,
            int tiempo_inicio,
            int tiempo_rafaga,
            int * indice)
```

Registra los TIEMPOS de los PROCESOS.

Parameters

diagrama_gantt	Diagrama de Gantt.
pid	ID del proceso.
tiempo_inicio	Tiempo de inicio del proceso.
tiempo_rafaga	Tiempo de ráfaga del proceso.
indice	Índice del diagrama de Gantt.

Se REGISTRAN los TIEMPOS de los PROCESOS, se asignan los valores al DIAGRAMA de GANTT y se actualiza el ÍNDICE del diagrama de Gantt.

```
diagrama_gantt[*indice].pid = pid;
diagrama_gantt[*indice].tiempo_inicio = tiempo_inicio;
diagrama_gantt[*indice].tiempo_final = tiempo_inicio + tiempo_rafaga;
(*indice)++;
```

Definition at line 175 of file auxiliar.c.

5.2 auxiliar.h

Go to the documentation of this file.

```
00001
00010 #ifndef AUXILIAR_H
00011 #define AUXILIAR_H
00012
00013 #include "memoria.h"
00014
00024 void leer_entrada(const char *, int *, int *, char *, char *, Cola *);
00025
00034 void asignar_valores_procesos(int, int, int, int, Cola *);
00035
00040 void imprimir_cola_procesos(Cola *);
00041
00047 void imprimir_memoria(BloqueMemoria *, int);
00048
00057 void registrar_tiempos(Gantt *, int, int, int, int *);
00058
00064 void imprimir_gantt(Gantt *, int);
00065
00072 void generar_archivo_gantt(Gantt *, int, const char *);
00073
00080 void generar_archivo_gantt(Gantt *, int, const char *);
00081
00082 #endif
```

5.3 incs/memoria.h File Reference

Prototipos de funciones dedicadas a la memoria y estructura de los BLOQUES de MEMORIA.

```
#include "planificador.h"
```

Classes

- struct BloqueMemoria

  Estructura de un BLOQUE de MEMORIA.

Macros

- #define MAX_PROCESOS 100

  Máximo de PROCESOS que se imprimen en el DIAGRAMA de GANTT.

Functions

- void inicializar_bloques_memoria (BloqueMemoria *, int, int)

  Inicializa los BLOQUES de MEMORIA.
- void asignar_memoria_procesos (Cola *, BloqueMemoria *, int)

  Asigna MEMORIA a los PROCESOS para la EJECUCIÓN.
- void asignar_bloque_proceso (BloqueMemoria *, Proceso *, int *, int)

  Asigna un BLOQUE de MEMORIA a un PROCESO.
- void verificar_fragmentacion (BloqueMemoria *, int)

  Verifica la FRAGMENTACIÓN INTERNA.
- void manejar_proceso (BloqueMemoria *, Proceso *, int, Gantt *, int *, int *)

  Maneja el PROCESO.
- void ejecutar_proceso (BloqueMemoria *, Proceso *, int)

  Simula la ejecución de un PROCESO.
- void liberar_memoria (BloqueMemoria *, int)

  Libera la MEMORIA de un PROCESO.

5.3.1 Detailed Description

Prototipos de funciones dedicadas a la memoria y estructura de los BLOQUES de MEMORIA.

Date

24-10-2024

Authors

Miguel Loaiza, Diego Sanhueza y Oscar Cifuentes

Contiene los prototipos de las funciones dedicadas a la asignación de memoria a los procesos (First Fit) y la estructura de los BLOQUES de MEMORIA.

Definition in file memoria.h.

5.3.2 Macro Definition Documentation

5.3.2.1 MAX_PROCESOS

```
#define MAX_PROCESOS 100
```

Máximo de PROCESOS que se imprimen en el DIAGRAMA de GANTT.

Librería que contiene las funciones de PLANIFICACIÓN y la estructura de un PROCESO.

Definition at line 13 of file memoria.h.

5.3.3 Function Documentation

5.3.3.1 asignar_bloque_proceso()

```
void asignar_bloque_proceso (
            BloqueMemoria * memoria,
            Proceso * proceso,
            int * tamano_proceso,
            int i)
```

Asigna un BLOQUE de MEMORIA a un PROCESO.

Parameters

memoria	Arreglo de BLOQUES de MEMORIA.
proceso	PROCESO a asignar.
tamano_proceso	Tamaño del PROCESO.
i	Índice del BLOQUE de MEMORIA.

Asigna un BLOQUE de MEMORIA a un PROCESO.

Parameters

memoria	Arreglo de BLOQUES de MEMORIA.
proceso	Proceso.
tamano_proceso	Tamaño del proceso.
i	Índice.

Si el TAMAÑO de MEMORIA del PROCESO alcanza en un solo BLOQUE hace la signación COMPLETA del BLOQUE, si no, toma más de uno y lo asigna PARCIALMENTE.

```
if (memoria[i].tamano >= *tamano_proceso)
{
    fprintf(stdout, "El proceso %d usó MEMORIA hasta el BLOQUE %d\n", proceso->pid, i);
    memoria[i].tamano -= *tamano_proceso;
    memoria[i].estado = 0;
    *tamano_proceso = 0;
    verificar_fragmentacion(memoria, i);
    ejecutar_proceso(memoria, proceso, i);
}
else
{
    *tamano_proceso -= memoria[i].tamano;
    memoria[i].tamano = 0;
    memoria[i].estado = 0;
    fprintf(stdout, "Proceso %d en EJECUCIÓN, bloque (%d)\n", proceso->pid, i);
    ejecutar_proceso(memoria, proceso, i);
    verificar_fragmentacion(memoria, i);
}
```

Definition at line 153 of file memoria.c.

5.3.3.2 asignar_memoria_procesos()

```
void asignar_memoria_procesos (
            Cola * cola,
            BloqueMemoria * memoria,
            int cantidad_bloques)
```

Asigna MEMORIA a los PROCESOS para la EJECUCIÓN.

Parameters

cola	Cola de PROCESOS.
memoria	Arreglo de BLOQUES de MEMORIA.
cantidad_bloques	Cantidad de BLOQUES de MEMORIA.

Asigna MEMORIA a los PROCESOS para la EJECUCIÓN.

Parameters

cola	Cola de PROCESOS.
memoria	Arreglo de BLOQUES de MEMORIA.
cantidad_bloques	Cantidad de BLOQUES de MEMORIA.

Se CREA una COPIA de la COLA de PROCESOS y se EXTRAE el PROCESO de la COLA. Se CREA un DIAGRAMA de GANTT y se INICIALIZAN las VARIABLES.

```
Cola *temp_cola = malloc(sizeof(Cola));
temp_cola = cola;
Proceso *proceso_extraido = malloc(sizeof(Proceso));
Gantt diagrama_gantt[MAX_PROCESOS];
```

```
int indice = 0, tiempo_global = 0;
```

MIENTRAS la COLA de PROCESOS NO esté VACÍA, se EXTRAE el PROCESO de la COLA y se MANEJA el PROCESO. Depués de manejar el proceso, se libera el PROCESO EXTRAIDO. Posteriormente se IMPRIME el DIAGRAMA de GANTT.

```
while (temp_cola->front != NULL)
{
    proceso_extraido = dequeue(temp_cola);
    if (proceso_extraido == NULL)
    {
        fprintf(stdout, "No hay procesos en la cola para asignar memoria.\n");
        return;
    }
    manejar_proceso(memoria, proceso_extraido, cantidad_bloques, diagrama_gantt, &tiempo_global, &indice);
    free(proceso_extraido);
}
imprimir_gantt(diagrama_gantt, indice);
generar_archivo_gantt(diagrama_gantt, indice, "gantt.eps");
```

Definition at line 43 of file memoria.c.

5.3.3.3 ejecutar_proceso()

```
void ejecutar_proceso (
            BloqueMemoria * memoria,
            Proceso * proceso,
            int posicion)
```

Simula la ejecución de un PROCESO.

Parameters

memoria	Arreglo de BLOQUES de MEMORIA.
proceso	PROCESO a ejecutar.
i	Índice del BLOQUE de MEMORIA.

Simula la ejecución de un PROCESO.

Parameters

memoria	Arreglo de BLOQUES de MEMORIA.
proceso	Proceso.
posicion	Posición.

SIMULA la ejecución del PROCESO y luego LIBERA la MEMORIA del PROCESO.

```
sleep(proceso->tiempo_rafaga);
liberar_memoria(memoria, posicion);
```

Definition at line 224 of file memoria.c.

5.3.3.4 inicializar_bloques_memoria()

```
void inicializar_bloques_memoria (
            BloqueMemoria * memoria,
            int cantidad_bloques,
            int tamano_bloque)
```

Inicializa los BLOQUES de MEMORIA.

Parameters

memoria	Arreglo de BLOQUES de MEMORIA.
cantidad_bloques	Cantidad de BLOQUES de MEMORIA.
tamano_bloque	Tamaño de los BLOQUES de MEMORIA.

Inicializa los BLOQUES de MEMORIA.

Inicializa los BLOQUES de MEMORIA.

Parameters

memoria	Arreglo de BLOQUES de MEMORIA.
cantidad_bloques	Cantidad de BLOQUES de MEMORIA.
tamano_bloque	Tamaño de los BLOQUES de MEMORIA.

Recorre el ARREGLO de MEMORIA y asigna el TAMAÑO del BLOQUE y el ESTADO del BLOQUE.

```
for (int i = 0; i < cantidad_bloques; i++)
{
    memoria[i].tamano = tamano_bloque;
    memoria[i].estado = 1;
}
```

Definition at line 18 of file memoria.c.

5.3.3.5 liberar_memoria()

```
void liberar_memoria (
            BloqueMemoria * memoria,
            int posicion)
```

Libera la MEMORIA de un PROCESO.

Parameters

memoria	Arreglo de BLOQUES de MEMORIA.
i	Índice del BLOQUE de MEMORIA.
memoria	Arreglo de BLOQUES de MEMORIA.
posicion	Posición.

LIBERA la MEMORIA de un PROCESO y la MARCA como LIBRE (1).

```
memoria[posicion].estado = 1;
memoria[posicion].tamano = 128;
```

Definition at line 242 of file memoria.c.

5.3.3.6 manejar_proceso()

```
void manejar_proceso (
            BloqueMemoria * memoria,
            Proceso * proceso_extraido,
            int cantidad_bloques,
            Gantt * diagrama_gantt,
            int * tiempo_global,
            int * indice)
```

Maneja el PROCESO.

Parameters

memoria	Arreglo de BLOQUES de MEMORIA.
proceso	PROCESO a manejar.
cantidad_bloques	Cantidad de BLOQUES de MEMORIA.
diagrama_gantt	Diagrama de Gantt.
tiempo_global	Tiempo global de ejecución.
indice	Índice del diagrama de Gantt.

Maneja el PROCESO.

Parameters

memoria	Arreglo de BLOQUES de MEMORIA.
proceso_extraido	Proceso extraído de la COLA.
cantidad_bloques	Cantidad de BLOQUES de MEMORIA.
diagrama_gantt	Diagrama de Gantt.
tiempo_global	Tiempo global.
indice	Índice del diagrama de Gantt.

Se GUARDA el TIEMPO de INICIO del PROCESO y el TAMAÑO del PROCESO. Si el TAMAÑO del PROCESO es mayor a 2048, se imprime un MENSAJE de ERROR. Se RECORRE los BLOQUES de MEMORIA y se ASIGNA la MEMORIA al PROCESO.

```
int tiempo_inicio = *tiempo_global;
int tamano_proceso = proceso_extraido->memoria_solicitada;
if (tamano_proceso > 2048)
{
    fprintf(stderr, "No hay suficiente memoria disponible para asignar al PROCESO %d, pasando al siguiente
      PROCESO.\n\n", proceso_extraido->pid);
    return;
}
for (int i = 0; i < cantidad_bloques && tamano_proceso > 0; i++)
{
    if (memoria[i].estado == 1)
    {
        asignar_bloque_proceso(memoria, proceso_extraido, &tamano_proceso, i);
    }
}
*tiempo_global += proceso_extraido->tiempo_rafaga;
registrar_tiempos(diagrama_gantt, proceso_extraido->pid, tiempo_inicio, proceso_extraido->tiempo_rafaga,
      indice);
fprintf(stdout, "Proceso %d EJECUTADO EXITOSAMENTE.\n\n", proceso_extraido->pid);
```

Definition at line 103 of file memoria.c.

5.3.3.7 verificar_fragmentacion()

```
void verificar_fragmentacion (
            BloqueMemoria * memoria,
            int i)
```

Verifica la FRAGMENTACIÓN INTERNA.

Parameters

memoria	Arreglo de BLOQUES de MEMORIA.
i	Índice del BLOQUE de MEMORIA.
memoria	Arreglo de BLOQUES de MEMORIA.

i	Índice.

Calcula la FRAGMENTACIÓN INTERNA. Si el tamaño de la memoria del PROCESO es mayor a 0, hay fragmentación interna.

```
int fragmentacion_interna = memoria[i].tamano;
if (fragmentacion_interna > 0)
    fprintf(stdout, "FRAGMENTACIÓN INTERNA: %d KB en el BLOQUE %d\n", fragmentacion_interna, i);
```

Definition at line 203 of file memoria.c.

5.4 memoria.h

Go to the documentation of this file.

```
00001
00010 #ifndef MEMORIA_H
00011 #define MEMORIA_H
00012
00013 #define MAX_PROCESOS 100
00014
00015 #include "planificador.h"
00016
00030 typedef struct
00031 {
00032     int tamano;
00033     int estado;
00034 } BloqueMemoria;
00035
00042 void inicializar_bloques_memoria(BloqueMemoria *, int, int);
00043
00050 void asignar_memoria_procesos(Cola *, BloqueMemoria *, int);
00051
00059 void asignar_bloque_proceso(BloqueMemoria *, Proceso *, int *, int);
00060
00066 void verificar_fragmentacion(BloqueMemoria *, int);
00067
00077 void manejar_proceso(BloqueMemoria *, Proceso *, int, Gantt *, int *, int *);
00078
00085 void ejecutar_proceso(BloqueMemoria *, Proceso *, int);
00086
00092 void liberar_memoria(BloqueMemoria *, int);
00093
00094 #endif
```

5.5 incs/planificador.h File Reference

Prototipos de funciones dedicadas a la planificación (FIFO) y estructuras.

```
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
```

Classes

- struct Proceso

  Estructura de un PROCESO.
- struct Cola

  Estructura de una COLA.
- struct Gantt

  Estructura de un DIAGRAMA de GANTT.

Typedefs

- typedef struct Proceso **Proceso**

Functions

- void enqueue (Cola ∗cola, Proceso ∗proceso)
  Coloca un PROCESO a la COLA.
- Proceso ∗ dequeue (Cola ∗cola)
  Extrae un PROCESO de la COLA.

5.5.1 Detailed Description

Prototipos de funciones dedicadas a la planificación (FIFO) y estructuras.

Date

24-10-2024

Authors

Miguel Loaiza, Diego Sanhueza y Oscar Cifuentes

Contiene los prototipos de las funciones dedicadas a la planificación de los procesos (FIFO) y las estructuras de un PROCESO, la COLA y la CARTA GANTT.

Definition in file planificador.h.

5.5.2 Function Documentation

5.5.2.1 dequeue()

```
Proceso * dequeue (
            Cola * cola)
```

Extrae un PROCESO de la COLA.

Parameters

cola	Cola de PROCESOS.

Returns

Proceso∗ PROCESO extraído.

Parameters

cola	Cola de PROCESOS.

Returns

Proceso∗ Proceso extraído.

Si el FRENTE de la COLA es NULL, la COLA está VACÍA. Se imprime un MENSAJE de ERROR y se SALE del programa. Se EXTRAE el PROCESO del FRENTE de la COLA y se actualiza el FRENTE de la COLA. Si el FRENTE de la COLA es NULL, el FINAL de la COLA también es NULL (COLA VACÍA).

```
if (cola->front == NULL)
{
    fprintf(stdout, "La cola está vacía.\n");
    exit(EXIT_FAILURE);
}
```

Definition at line 52 of file planificador.c.

5.5.2.2 enqueue()

```
void enqueue (
            Cola * cola,
            Proceso * proceso)
```

Coloca un PROCESO a la COLA.

Parameters

cola	Cola de PROCESOS.
proceso	PROCESO a encolar.

Coloca un PROCESO a la COLA.

Coloca un PROCESO en la COLA.

Parameters

id	ID del proceso.
tiempo_llegada	Tiempo de llegada del proceso.
tiempo_rafaga	Tiempo de ráfaga del proceso.
memoria_solicitada	Memoria solicitada por el proceso.
cola_procesos	Cola de PROCESOS.

Si la COLA está VACÍA, el FRENTE y el FINAL de la COLA es el PROCESO. Se actualiza el SIGUIENTE PROCESO con NULL porque NO hay otro proceso DETRÁS. Imprime la cola de procesos.

```
if (cola->rear == NULL)
    cola->front = proceso;
else
    cola->rear->next = proceso;
cola->rear = proceso;
proceso->next = NULL;
fprintf(stdout, "Proceso %d encolado CORRECTANENTE.\n", proceso->pid);
imprimir_cola_procesos(cola);
fprintf(stdout, "\n");
```

Definition at line 20 of file planificador.c.

5.6 planificador.h

Go to the documentation of this file.

```
00001
00010 #ifndef PLANIFICADOR_H
00011 #define PLANIFICADOR_H
00012
00013 #include <string.h>
00014 #include <stdlib.h>
00015 #include <unistd.h>
00016 #include <stdio.h>
00017
00034 typedef struct Proceso
00035 {
00036     int pid;
00037     int tiempo_llegada;
00038     int tiempo_rafaga;
00039     int memoria_solicitada;
00040     struct Proceso *next;
00041 } Proceso;
00042
00056 typedef struct
00057 {
```

```
00058     Proceso *front; // FRENTE de la cola.
00059     Proceso *rear;  // FINAL de la cola.
00060 } Cola;
00061
00076 typedef struct
00077 {
00078     int pid;           // ID del PROCESO.
00079     int tiempo_inicio; // Tiempo de INICIO.
00080     int tiempo_final;  // Tiempo de FINALIZACIÓN.
00081 } Gantt;
00082
00088 void enqueue(Cola *cola, Proceso *proceso);
00089
00095 Proceso *dequeue(Cola *cola);
00096
00097 #endif
```

5.7 src/auxiliar.c File Reference

Funciones auxiliares.

```
#include "auxiliar.h"
```

Functions

- void leer_entrada (const char *nombre_archivo, int *tamano_memoria, int *tamano_bloque, char *algoritmo_memoria, char *algoritmo_planificacion, Cola *cola_procesos)

  < Librería que contiene las funciones auxiliares.
- void asignar_valores_procesos (int id, int tiempo_llegada, int tiempo_rafaga, int memoria_solicitada, Cola *cola_procesos)

  Crea un nuevo PROCESO y le asigna los VALORES.
- void imprimir_cola_procesos (Cola *cola)

  Imprime la COLA de PROCESOS.
- void imprimir_memoria (BloqueMemoria *memoria, int cantidad_bloques)

  Imprime los BLOQUES de MEMORIA INICIALIZADOS.
- void registrar_tiempos (Gantt *diagrama_gantt, int pid, int tiempo_inicio, int tiempo_rafaga, int *indice)

  Registra los TIEMPOS de los PROCESOS.
- void imprimir_gantt (Gantt *diagrama_gantt, int num_procesos)

  Imprime el DIAGRAMA de GANTT.
- void generar_archivo_gantt (Gantt *diagrama_gantt, int num_procesos, const char *carta_gantt)

  Genera el ARCHIVO EPS que muestra la CARTA GANTT. Al final se convierte el archivo EPS a PNG.

5.7.1 Detailed Description

Funciones auxiliares.

Date

24-10-2024

Authors

Miguel Loaiza, Diego Sanhueza y Oscar Cifuentes

Contiene las funciones auxiliares que son utilizadas para el correcto funcionamiento de los archivos de MEMORIA y PLANIFICACIÓN.

Definition in file auxiliar.c.

5.7.2 Function Documentation

5.7.2.1 asignar_valores_procesos()

```
void asignar_valores_procesos (
            int id,
            int tiempo_llegada,
            int tiempo_rafaga,
            int memoria_solicitada,
            Cola * cola_procesos)
```

Crea un nuevo PROCESO y le asigna los VALORES.

Parameters

id	ID del proceso.
tiempo_llegada	Tiempo de llegada del proceso.
tiempo_rafaga	Tiempo de ráfaga del proceso.
memoria_solicitada	Memoria solicitada por el proceso.
cola_procesos	Cola de PROCESOS.

Se CREA un NUEVO PROCESO y se ASIGNAN los VALORES necesarios, luego se llama a la función "enqueue" para colocar en la COLA el PROCESO.

```
Proceso *nuevo_proceso = (Proceso *)malloc(sizeof(Proceso));
nuevo_proceso->pid = id;
nuevo_proceso->tiempo_llegada = tiempo_llegada;
nuevo_proceso->tiempo_rafaga = tiempo_rafaga;
nuevo_proceso->memoria_solicitada = memoria_solicitada;
nuevo_proceso->next = NULL;
fprintf(stdout, "Proceso %d - T. Llegada %d - T. Ráfaga %d - M. Solicitada %d\n", id, tiempo_llegada,
      tiempo_rafaga, memoria_solicitada);
enqueue(cola_procesos, nuevo_proceso);
```

Definition at line 91 of file auxiliar.c.

5.7.2.2 generar_archivo_gantt()

```
void generar_archivo_gantt (
            Gantt * diagrama_gantt,
            int num_procesos,
            const char * carta_gantt)
```

Genera el ARCHIVO EPS que muestra la CARTA GANTT. Al final se convierte el archivo EPS a PNG.

Genera el ARCHIVO EPS que muestra la CARTA GANTT.

Genera un ARCHIVO que muestra la CARTA GANTT.

Parameters

diagrama_gantt	Diagrama de Gantt.
num_procesos	Número de procesos.
carta_gantt	Nombre del archivo de la carta Gantt.

Se abre el archivo de GANTT en modo ESCRITURA y se VERIFICA si se abrió correctamente.

```
FILE *archivo_gantt = fopen(carta_gantt, "w");
if (archivo_gantt == NULL)
{
    fprintf(stderr, "ERROR al abrir el archivo de Gantt, saliendo...\n");
    exit(EXIT_FAILURE);
}
```

VARIABLES para DIBUJAR la CARTA GANTT. Se DIBUJAN los RECTÁNGULOS de los PROCESOS y se COLOREAN, se PINTA el BORDE del RECTÁNGULO y se ESCRIBE el ID del PROCESO en el eje Y.

```
int altura_proceso = 50;
int margen_inferior = 100, margen_superior = 50, margen_lateral = 100;
int ancho_total = 800, altura_total = (num_procesos * altura_proceso) + margen_superior + margen_inferior;
fprintf(archivo_gantt, "%%!PS-Adobe-3.0 EPSF-3.0\n");
fprintf(archivo_gantt, "%%%%BoundingBox: 0 0 %d %d\n", ancho_total + margen_lateral * 2, altura_total);
for (int i = 0; i < num_procesos; i++)
{
    int tiempo_inicio = diagrama_gantt[i].tiempo_inicio;
    int tiempo_final = diagrama_gantt[i].tiempo_final;
    int pid = diagrama_gantt[i].pid;
    int x_inicio = margen_lateral + tiempo_inicio * 10;
    int x_final = margen_lateral + tiempo_final * 10;
    int y = margen_inferior + i * altura_proceso;
    fprintf(archivo_gantt, "newpath\n");
    fprintf(archivo_gantt, "%d %d moveto\n", x_inicio, y);
    fprintf(archivo_gantt, "%d %d lineto\n", x_final, y);
    fprintf(archivo_gantt, "%d %d lineto\n", x_final, y + altura_proceso - 10);
    fprintf(archivo_gantt, "%d %d lineto\n", x_inicio, y + altura_proceso - 10);
    fprintf(archivo_gantt, "closepath\n");
    fprintf(archivo_gantt, "0.207 0.706 0.941 setrgbcolor\n");
    fprintf(archivo_gantt, "fill\n");
    fprintf(archivo_gantt, "0 0 0 setrgbcolor\n");
    fprintf(archivo_gantt, "stroke\n");
    fprintf(archivo_gantt, "/Times-Roman findfont 12 scalefont setfont\n");
    fprintf(archivo_gantt, "%d %d moveto\n", margen_lateral - 50, y + altura_proceso / 2);
    fprintf(archivo_gantt, "(ID %d) show\n", pid);
}
```

DIBUJA el eje X con los TIEMPOS REALES, se recorre el DIAGRAMA de GANTT y se imprime el TIEMPO de INICIO y FINAL del PROCESO en el eje X.

```
fprintf(archivo_gantt, "/Times-Roman findfont 12 scalefont setfont\n");
for (int i = 0; i < num_procesos; i++)
{
    int tiempo_inicio = diagrama_gantt[i].tiempo_inicio;
    int tiempo_final = diagrama_gantt[i].tiempo_final;
    int x_inicio = margen_lateral + tiempo_inicio * 10;
    fprintf(archivo_gantt, "newpath\n");
    fprintf(archivo_gantt, "%d %d moveto\n", x_inicio, margen_inferior - 5);
    fprintf(archivo_gantt, "%d %d lineto\n", x_inicio, margen_inferior + 5);
    fprintf(archivo_gantt, "stroke\n");
    fprintf(archivo_gantt, "%d %d moveto\n", x_inicio - 5, margen_inferior - 20);
    fprintf(archivo_gantt, "(%d) show\n", tiempo_inicio);
    int x_final = margen_lateral + tiempo_final * 10;
    fprintf(archivo_gantt, "newpath\n");
    fprintf(archivo_gantt, "%d %d moveto\n", x_final, margen_inferior - 5);
    fprintf(archivo_gantt, "%d %d lineto\n", x_final, margen_inferior + 5);
    fprintf(archivo_gantt, "stroke\n");
    fprintf(archivo_gantt, "%d %d moveto\n", x_final - 5, margen_inferior - 20);
    fprintf(archivo_gantt, "(%d) show\n", tiempo_final);
}
fprintf(archivo_gantt, "showpage\n");
fclose(archivo_gantt);
fprintf(stdout, "\nCARTA GANTT generada EXITOSAMENTE en el ARCHIVO %s\n\n", carta_gantt);
```

Se CONVIERTE el ARCHIVO EPS a PNG con GHOSTSCRIPT.

```
char comando[256];
snprintf(comando, sizeof(comando), "gs -dSAFER -dBATCH -dNOPAUSE -dEPSCrop -sDEVICE=png16m -r300
      -sOutputFile=%s.png %s", carta_gantt, carta_gantt);
int resultado = system(comando);
if (resultado != 0)
{
    fprintf(stderr, "ERROR al convertir la CARTA GANTT a PNG, saliendo...\n");
    exit(EXIT_FAILURE);
}
fprintf(stdout, "\nCARTA GANTT convertida a PNG en el archivo %s.png\n\n", carta_gantt);
```

Definition at line 218 of file auxiliar.c.

5.7.2.3 imprimir_cola_procesos()

```
void imprimir_cola_procesos (
            Cola * cola)
```

Imprime la COLA de PROCESOS.

Parameters

cola	Cola de PROCESOS.

Se IMPRIME la COLA de PROCESOS, se recorre la COLA y se imprime PROCESO actual, luego se actualiza el PROCESO actual.

```
Proceso *actual = cola->front;
fprintf(stdout, "\nCola de procesos:\n");
long actual_next = (long)actual->next;
while (actual != NULL)
{
    fprintf(stdout, "[Pid: %d]\t[Memoria Solicitada: %d]\t[Next: %ld]->\n", actual->pid,
      actual->memoria_solicitada, actual_next);
    actual = actual->next;
}
```

Definition at line 120 of file auxiliar.c.

5.7.2.4 imprimir_gantt()

```
void imprimir_gantt (
            Gantt * diagrama_gantt,
            int num_procesos)
```

Imprime el DIAGRAMA de GANTT.

Parameters

diagrama_gantt	Diagrama de Gantt.
num_procesos	Número de procesos.

Se IMPRIME el DIAGRAMA de GANTT, se recorre el DIAGRAMA de GANTT y se imprime el PROCESO actual.

```
fprintf(stdout, "Diagrama de Gantt:\n\n");
for (int i = 0; i < num_procesos; i++)
    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);
```

Definition at line 197 of file auxiliar.c.

5.7.2.5 imprimir_memoria()

```
void imprimir_memoria (
            BloqueMemoria * memoria,
            int cantidad_bloques)
```

Imprime los BLOQUES de MEMORIA INICIALIZADOS.

Parameters

memoria	Arreglo de BLOQUES de MEMORIA.
cantidad_bloques	Cantidad de BLOQUES de MEMORIA.

Se IMPRIME los BLOQUES de MEMORIA INICIALIZADOS, se recorre el ARREGLO de MEMORIA y se imprime el BLOQUE actual.

```
fprintf(stdout, "Bloques de memoria del SISTEMA OPERATIVO:\n\n");
for (int i = 0; i < cantidad_bloques; i++)
    fprintf(stdout, "[Bloque %d]\t[Tamaño %d]\t[Estado %d][LIBRE]\n", i, memoria[i].tamano,
      memoria[i].estado);
fprintf(stdout, "\n");
```

Definition at line 150 of file auxiliar.c.

5.7.2.6 leer_entrada()

```
void leer_entrada (
            const char * nombre_archivo,
            int * tamano_memoria,
            int * tamano_bloque,
            char * algoritmo_memoria,
            char * algoritmo_planificacion,
            Cola * cola_procesos)
```

< Librería que contiene las funciones auxiliares.

< Librería que contiene las funciones de MEMORIA y la estructra del BLOQUE de MEMORIA.

Lee la entrada del archivo de texto (entrada.txt).

Parameters

nombre_archivo	Nombre del archivo de texto.
tamano_memoria	Tamaño de la memoria TOTAL (2048 KB).
tamano_bloque	Tamaño de los BLOQUES de MEMORIA (128 KB).
algoritmo_memoria	Algoritmo de MEMORIA (FF).
algoritmo_planificacion	Algoritmo de PLANIFICACIÓN (FIFO).
cola_procesos	Cola de PROCESOS.

Se abre el archivo de entrada en modo LECTURA y se VERIFICA si se abrió correctamente.

```
FILE *archivo_entrada = fopen(nombre_archivo, "r");
if (archivo_entrada == NULL)
{
    fprintf(stderr, "ERROR al abrir el archivo, saliendo...\n");
    exit(EXIT_FAILURE);
}
```

Lee la cabecera del archivo de entrada (2048 128 ff) y VERIFICA si los valores son 3, también lee el algoritmo de planificación (FIFO) y verifica si el valor es 1.

```
if (fscanf(archivo_entrada, "%d %d %s", tamano_memoria, tamano_bloque, algoritmo_memoria) != 3)
{
    fprintf(stderr, "ERROR al leer los valores de la memoria o bloque, saliendo...\n");
    exit(EXIT_FAILURE);
}

if (fscanf(archivo_entrada, "%s", algoritmo_planificacion) != 1)
{
    fprintf(stderr, "Error al leer el algoritmo de planificación\n");
    exit(EXIT_FAILURE);
```

```
}
```

VARIABLES del PROCESO (vienen de entrada.txt). Lee los VALORES de los PROCESOS del archivo de entrada y llama la función "asignar_valores_procesos", luego cierra el archivo.

```
int id, tiempo_llegada, tiempo_rafaga, memoria_solicitada;
while (fscanf(archivo_entrada, "%d %d %d %d", &id, &tiempo_llegada, &tiempo_rafaga, &memoria_solicitada) !=
      EOF)
    asignar_valores_procesos(id, tiempo_llegada, tiempo_rafaga, memoria_solicitada, cola_procesos);
fclose(archivo_entrada);
```

Definition at line 21 of file auxiliar.c.

5.7.2.7 registrar_tiempos()

```
void registrar_tiempos (
            Gantt * diagrama_gantt,
            int pid,
            int tiempo_inicio,
            int tiempo_rafaga,
            int * indice)
```

Registra los TIEMPOS de los PROCESOS.

Parameters

diagrama_gantt	Diagrama de Gantt.
pid	ID del proceso.
tiempo_inicio	Tiempo de inicio del proceso.
tiempo_rafaga	Tiempo de ráfaga del proceso.
indice	Índice del diagrama de Gantt.

Se REGISTRAN los TIEMPOS de los PROCESOS, se asignan los valores al DIAGRAMA de GANTT y se actualiza el ÍNDICE del diagrama de Gantt.

```
diagrama_gantt[*indice].pid = pid;
diagrama_gantt[*indice].tiempo_inicio = tiempo_inicio;
diagrama_gantt[*indice].tiempo_final = tiempo_inicio + tiempo_rafaga;
(*indice)++;
```

Definition at line 175 of file auxiliar.c.

5.8 auxiliar.c

Go to the documentation of this file.

```
00001
00010 #include "auxiliar.h"
00011
00021 void leer_entrada(const char *nombre_archivo, int *tamano_memoria, int *tamano_bloque, char
      *algoritmo_memoria, char *algoritmo_planificacion, Cola *cola_procesos)
00022 {
00034     FILE *archivo_entrada = fopen(nombre_archivo, "r");
00035     if (archivo_entrada == NULL)
00036     {
00037         fprintf(stderr, "ERROR al abrir el archivo, saliendo...\n");
00038         exit(EXIT_FAILURE);
00039     }
00040
00057     if (fscanf(archivo_entrada, "%d %d %s", tamano_memoria, tamano_bloque, algoritmo_memoria) != 3)
00058     {
00059         fprintf(stderr, "ERROR al leer los valores de la memoria o bloque, saliendo...\n");
00060         exit(EXIT_FAILURE);
00061     }
```

```
00062     if (fscanf(archivo_entrada, "%s", algoritmo_planificacion) != 1)
00063     {
00064         fprintf(stderr, "Error al leer el algoritmo de planificación\n");
00065         exit(EXIT_FAILURE);
00066     }
00067
00077     int id, tiempo_llegada, tiempo_rafaga, memoria_solicitada;
00078     while (fscanf(archivo_entrada, "%d %d %d %d", &id, &tiempo_llegada, &tiempo_rafaga,
      &memoria_solicitada) != EOF)
00079         asignar_valores_procesos(id, tiempo_llegada, tiempo_rafaga, memoria_solicitada,
      cola_procesos);
00080     fclose(archivo_entrada);
00081 }
00082
00091 void asignar_valores_procesos(int id, int tiempo_llegada, int tiempo_rafaga, int memoria_solicitada,
      Cola *cola_procesos)
00092 {
00106     Proceso *nuevo_proceso = (Proceso *)malloc(sizeof(Proceso));
00107     nuevo_proceso->pid = id;
00108     nuevo_proceso->tiempo_llegada = tiempo_llegada;
00109     nuevo_proceso->tiempo_rafaga = tiempo_rafaga;
00110     nuevo_proceso->memoria_solicitada = memoria_solicitada;
00111     nuevo_proceso->next = NULL;
00112     fprintf(stdout, "Proceso %d - T. Llegada %d - T. Ráfaga %d - M. Solicitada %d\n", id,
      tiempo_llegada, tiempo_rafaga, memoria_solicitada);
00113     enqueue(cola_procesos, nuevo_proceso);
00114 }
00115
00120 void imprimir_cola_procesos(Cola *cola)
00121 {
00135     Proceso *actual = cola->front;
00136     fprintf(stdout, "\nCola de procesos:\n");
00137     long actual_next = (long)actual->next;
00138     while (actual != NULL)
00139     {
00140         fprintf(stdout, "[PID: %d]\t[Memoria Solicitada: %d]\t[Next: %ld]->\n", actual->pid,
      actual->memoria_solicitada, actual_next);
00141         actual = actual->next;
00142     }
00143 }
00144
00150 void imprimir_memoria(BloqueMemoria *memoria, int cantidad_bloques)
00151 {
00161     fprintf(stdout, "Bloques de memoria del SISTEMA OPERATIVO:\n\n");
00162     for (int i = 0; i < cantidad_bloques; i++)
00163         fprintf(stdout, "[Bloque %d]\t[Tamaño %d]\t[Estado %d][LIBRE]\n", i, memoria[i].tamano,
      memoria[i].estado);
00164     fprintf(stdout, "\n");
00165 }
00166
00175 void registrar_tiempos(Gantt *diagrama_gantt, int pid, int tiempo_inicio, int tiempo_rafaga, int
      *indice)
00176 {
00186     diagrama_gantt[*indice].pid = pid;
00187     diagrama_gantt[*indice].tiempo_inicio = tiempo_inicio;
00188     diagrama_gantt[*indice].tiempo_final = tiempo_inicio + tiempo_rafaga;
00189     (*indice)++;
00190 }
00191
00197 void imprimir_gantt(Gantt *diagrama_gantt, int num_procesos)
00198 {
00207     fprintf(stdout, "Diagrama de Gantt:\n\n");
00208     for (int i = 0; i < num_procesos; i++)
00209         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);
00210 }
00211
00218 void generar_archivo_gantt(Gantt *diagrama_gantt, int num_procesos, const char *carta_gantt)
00219 {
00231     FILE *archivo_gantt = fopen(carta_gantt, "w");
00232     if (archivo_gantt == NULL)
00233     {
00234         fprintf(stderr, "ERROR al abrir el archivo de Gantt, saliendo...\n");
00235         exit(EXIT_FAILURE);
00236     }
00237
00270     int altura_proceso = 50;
00271     int margen_inferior = 100, margen_superior = 50, margen_lateral = 100;
00272     int ancho_total = 800, altura_total = (num_procesos * altura_proceso) + margen_superior +
      margen_inferior;
00273     fprintf(archivo_gantt, "%%!PS-Adobe-3.0 EPSF-3.0\n");
00274     fprintf(archivo_gantt, "%%%%BoundingBox: 0 0 %d %d\n", ancho_total + margen_lateral * 2,
      altura_total);
00275     for (int i = 0; i < num_procesos; i++)
00276     {
00277         int tiempo_inicio = diagrama_gantt[i].tiempo_inicio;
00278         int tiempo_final = diagrama_gantt[i].tiempo_final;
```

```
00279         int pid = diagrama_gantt[i].pid;
00280         int x_inicio = margen_lateral + tiempo_inicio * 10;
00281         int x_final = margen_lateral + tiempo_final * 10;
00282         int y = margen_inferior + i * altura_proceso;
00283         fprintf(archivo_gantt, "newpath\n");
00284         fprintf(archivo_gantt, "%d %d moveto\n", x_inicio, y);
00285         fprintf(archivo_gantt, "%d %d lineto\n", x_final, y);
00286         fprintf(archivo_gantt, "%d %d lineto\n", x_final, y + altura_proceso - 10);
00287         fprintf(archivo_gantt, "%d %d lineto\n", x_inicio, y + altura_proceso - 10);
00288         fprintf(archivo_gantt, "closepath\n");
00289         fprintf(archivo_gantt, "0.207 0.706 0.941 setrgbcolor\n");
00290         fprintf(archivo_gantt, "fill\n");
00291         fprintf(archivo_gantt, "0 0 0 setrgbcolor\n");
00292         fprintf(archivo_gantt, "stroke\n");
00293         fprintf(archivo_gantt, "/Times-Roman findfont 12 scalefont setfont\n");
00294         fprintf(archivo_gantt, "%d %d moveto\n", margen_lateral - 50, y + altura_proceso / 2);
00295         fprintf(archivo_gantt, "(ID %d) show\n", pid);
00296     }
00297
00326     fprintf(archivo_gantt, "/Times-Roman findfont 12 scalefont setfont\n");
00327     for (int i = 0; i < num_procesos; i++)
00328     {
00329         int tiempo_inicio = diagrama_gantt[i].tiempo_inicio;
00330         int tiempo_final = diagrama_gantt[i].tiempo_final;
00331         int x_inicio = margen_lateral + tiempo_inicio * 10;
00332         fprintf(archivo_gantt, "newpath\n");
00333         fprintf(archivo_gantt, "%d %d moveto\n", x_inicio, margen_inferior - 5);
00334         fprintf(archivo_gantt, "%d %d lineto\n", x_inicio, margen_inferior + 5);
00335         fprintf(archivo_gantt, "stroke\n");
00336         fprintf(archivo_gantt, "%d %d moveto\n", x_inicio - 5, margen_inferior - 20);
00337         fprintf(archivo_gantt, "(%d) show\n", tiempo_inicio);
00338         int x_final = margen_lateral + tiempo_final * 10;
00339         fprintf(archivo_gantt, "newpath\n");
00340         fprintf(archivo_gantt, "%d %d moveto\n", x_final, margen_inferior - 5);
00341         fprintf(archivo_gantt, "%d %d lineto\n", x_final, margen_inferior + 5);
00342         fprintf(archivo_gantt, "stroke\n");
00343         fprintf(archivo_gantt, "%d %d moveto\n", x_final - 5, margen_inferior - 20);
00344         fprintf(archivo_gantt, "(%d) show\n", tiempo_final);
00345     }
00346     fprintf(archivo_gantt, "showpage\n");
00347     fclose(archivo_gantt);
00348     fprintf(stdout, "\nCARTA GANTT generada EXITOSAMENTE en el ARCHIVO %s\n\n", carta_gantt);
00349
00350     // CONVERTIR archivo EPS a PNG con GHOSTSCRIPT.
00351
00366     char comando[256];
00367     snprintf(comando, sizeof(comando), "gs -dSAFER -dBATCH -dNOPAUSE -dEPSCrop -sDEVICE=png16m -r300
      -sOutputFile=%s.png %s", carta_gantt, carta_gantt);
00368     int resultado = system(comando);
00369     if (resultado != 0)
00370     {
00371         fprintf(stderr, "ERROR al convertir la CARTA GANTT a PNG, saliendo...\n");
00372         exit(EXIT_FAILURE);
00373     }
00374     fprintf(stdout, "\nCARTA GANTT convertida a PNG en el archivo %s.png\n\n", carta_gantt);
00375 }
```

5.9 src/main.c File Reference

Función principal.

```
#include "auxiliar.h"
```

Functions

- int main (int argc, char ∗argv[ ])

  < *Librería que contiene las funciones auxiliares.*

5.9.1 Detailed Description

Función principal.

Date

24-10-2024

Authors

Miguel Loaiza, Diego Sanhueza y Oscar Cifuentes

Contiene la función principal del programa, donde se leen los argumentos de la terminal y se ejecutan las funciones de MEMORIA y PLANIFICACIÓN.

Definition in file main.c.

5.9.2 Function Documentation

5.9.2.1 main()

```
int main (
            int argc,
            char * argv[])
```

< Librería que contiene las funciones auxiliares.

Función principal del programa.

Parameters

argc	Cantidad de argumentos.
argv	Argumentos.

Returns

EXIT_SUCCESS si el programa termina correctamente.

Se LEEN los ARGUMENTOS de la TERMINAL y se GUARDAN en VARIABLES. Mientras se LEEN los ARGUMENTOS, se VALIDAN y se IMPRIMEN MENSAJES de AYUDA o ERROR. En caso de que sea correcto, se GUARDA el NOMBRE del ARCHIVO de ENTRADA.

```
int opt;
char *nombre_archivo = NULL;
while ((opt = getopt(argc, argv, "hf:")) != -1)
{
    switch (opt)
    {
        case 'h':
            fprintf(stdout, "\nPara leer sus procesos, porfavor coloque el parametro <-f>
      <nombre_archivo.txt>\n\n");
            break;
        case 'f':
            nombre_archivo = optarg;
            printf("\nArchivo: %s\n\n", nombre_archivo);
            break;
        case '?':
            fprintf(stderr, "Opción no reconocida: -%c\n", optopt);
            exit(EXIT_FAILURE);
```

```
        default:
            fprintf(stderr, "Uso: %s [-h] [-f filename]\n", argv[0]);
            exit(EXIT_FAILURE);
    }
}
```

Se LEEN los VALORES de los PROCESOS del ARCHIVO de ENTRADA y se GUARDAN en VARIABLES. Se CALCULA la CANTIDAD de BLOQUES de MEMORIA y se INICIALIZAN los BLOQUES de MEMORIA. Se IMPRIME la MEMORIA y se ASIGNA la MEMORIA a los PROCESOS.

```
int tamano_memoria, tamano_bloque;
char algoritmo_memoria[10], algoritmo_planificacion[10];
Cola cola_procesos = {NULL, NULL};
leer_entrada(nombre_archivo, &tamano_memoria, &tamano_bloque, algoritmo_memoria, algoritmo_planificacion,
      &cola_procesos);
int cantidad_bloques = tamano_memoria / tamano_bloque;
BloqueMemoria memoria[cantidad_bloques];
inicializar_bloques_memoria(memoria, cantidad_bloques, tamano_bloque);
imprimir_memoria(memoria, cantidad_bloques);
asignar_memoria_procesos(&cola_procesos, memoria, cantidad_bloques);
return EXIT_SUCCESS;
```

Definition at line 18 of file main.c.

5.10 main.c

Go to the documentation of this file.

```
00001
00010 #include "auxiliar.h"
00011
00018 int main(int argc, char *argv[])
00019 {
00046     int opt;
00047     char *nombre_archivo = NULL;
00048     while ((opt = getopt(argc, argv, "hf:")) != -1)
00049     {
00050         switch (opt)
00051         {
00052         case 'h':
00053             fprintf(stdout, "\nPara leer sus procesos, porfavor coloque el parametro <-f>
      <nombre_archivo.txt>\n\n");
00054             break;
00055         case 'f':
00056             nombre_archivo = optarg;
00057             printf("\nArchivo: %s\n\n", nombre_archivo);
00058             break;
00059         case '?':
00060             fprintf(stderr, "Opción no reconocida: -%c\n", optopt);
00061             exit(EXIT_FAILURE);
00062         default:
00063             fprintf(stderr, "Uso: %s [-h] [-f filename]\n", argv[0]);
00064             exit(EXIT_FAILURE);
00065         }
00066     }
00067
00083     int tamano_memoria, tamano_bloque;
00084     char algoritmo_memoria[10], algoritmo_planificacion[10];
00085     Cola cola_procesos = {NULL, NULL};
00086     leer_entrada(nombre_archivo, &tamano_memoria, &tamano_bloque, algoritmo_memoria,
      algoritmo_planificacion, &cola_procesos);
00087     int cantidad_bloques = tamano_memoria / tamano_bloque;
00088     BloqueMemoria memoria[cantidad_bloques];
00089     inicializar_bloques_memoria(memoria, cantidad_bloques, tamano_bloque);
00090     imprimir_memoria(memoria, cantidad_bloques);
00091     asignar_memoria_procesos(&cola_procesos, memoria, cantidad_bloques);
00092     return EXIT_SUCCESS;
00093 }
```

5.11 src/memoria.c File Reference

Funciones de memoria.

```
#include "auxiliar.h"
```

Functions

- void inicializar_bloques_memoria (BloqueMemoria ∗memoria, int cantidad_bloques, int tamano_bloque)

  < Librería que contiene las funciones auxiliares.
- void asignar_memoria_procesos (Cola ∗cola, BloqueMemoria ∗memoria, int cantidad_bloques)

  Asigna la MEMORIA a los PROCESOS para la EJECUCIÓN.
- void manejar_proceso (BloqueMemoria ∗memoria, Proceso ∗proceso_extraido, int cantidad_bloques, Gantt ∗diagrama_gantt, int ∗tiempo_global, int ∗indice)

  Maneja el PROCESO, asigna MEMORIA y REGISTRA los tiempos de los procesos.
- void asignar_bloque_proceso (BloqueMemoria ∗memoria, Proceso ∗proceso, int ∗tamano_proceso, int i)

  Asigna el BLOQUE de MEMORIA al PROCESO.
- void verificar_fragmentacion (BloqueMemoria ∗memoria, int i)

  Verifica la FRAGMENTACIÓN INTERNA.
- void ejecutar_proceso (BloqueMemoria ∗memoria, Proceso ∗proceso, int posicion)

  Ejecuta el PROCESO.
- void liberar_memoria (BloqueMemoria ∗memoria, int posicion)

  Libera la MEMORIA de un PROCESO.

5.11.1 Detailed Description

Funciones de memoria.

Date

24-10-2024

Authors

Miguel Loaiza, Diego Sanhueza y Oscar Cifuentes

Contiene las funciones qe son utilizadas para la CORRECTA asignación de MEMORIA a los PROCESOS.

Definition in file memoria.c.

5.11.2 Function Documentation

5.11.2.1 asignar_bloque_proceso()

```
void asignar_bloque_proceso (
            BloqueMemoria * memoria,
            Proceso * proceso,
            int * tamano_proceso,
            int i)
```

Asigna el BLOQUE de MEMORIA al PROCESO.

Asigna un BLOQUE de MEMORIA a un PROCESO.

Parameters

memoria	Arreglo de BLOQUES de MEMORIA.
proceso	Proceso.
tamano_proceso	Tamaño del proceso.
i	Índice.

Si el TAMAÑO de MEMORIA del PROCESO alcanza en un solo BLOQUE hace la signación COMPLETA del BLOQUE, si no, toma más de uno y lo asigna PARCIALMENTE.

```
if (memoria[i].tamano >= *tamano_proceso)
{
    fprintf(stdout, "El proceso %d usó MEMORIA hasta el BLOQUE %d\n", proceso->pid, i);
    memoria[i].tamano -= *tamano_proceso;
    memoria[i].estado = 0;
    *tamano_proceso = 0;
    verificar_fragmentacion(memoria, i);
    ejecutar_proceso(memoria, proceso, i);
}
else
{
    *tamano_proceso -= memoria[i].tamano;
    memoria[i].tamano = 0;
    memoria[i].estado = 0;
    fprintf(stdout, "Proceso %d en EJECUCIÓN, bloque (%d)\n", proceso->pid, i);
    ejecutar_proceso(memoria, proceso, i);
    verificar_fragmentacion(memoria, i);
}
```

Definition at line 153 of file memoria.c.

5.11.2.2 asignar_memoria_procesos()

```
void asignar_memoria_procesos (
            Cola * cola,
            BloqueMemoria * memoria,
            int cantidad_bloques)
```

Asigna la MEMORIA a los PROCESOS para la EJECUCIÓN.

Asigna MEMORIA a los PROCESOS para la EJECUCIÓN.

Parameters

cola	Cola de PROCESOS.
memoria	Arreglo de BLOQUES de MEMORIA.
cantidad_bloques	Cantidad de BLOQUES de MEMORIA.

Se CREA una COPIA de la COLA de PROCESOS y se EXTRAE el PROCESO de la COLA. Se CREA un DIAGRAMA de GANTT y se INICIALIZAN las VARIABLES.

```
Cola *temp_cola = malloc(sizeof(Cola));
temp_cola = cola;
Proceso *proceso_extraido = malloc(sizeof(Proceso));
Gantt diagrama_gantt[MAX_PROCESOS];
int indice = 0, tiempo_global = 0;
```

MIENTRAS la COLA de PROCESOS NO esté VACÍA, se EXTRAE el PROCESO de la COLA y se MANEJA el PROCESO. Depués de manejar el proceso, se libera el PROCESO EXTRAIDO. Posteriormente se IMPRIME el DIAGRAMA de GANTT.

```
while (temp_cola->front != NULL)
{
    proceso_extraido = dequeue(temp_cola);
    if (proceso_extraido == NULL)
```

```
    {
        fprintf(stdout, "No hay procesos en la cola para asignar memoria.\n");
        return;
    }
    manejar_proceso(memoria, proceso_extraido, cantidad_bloques, diagrama_gantt, &tiempo_global, &indice);
    free(proceso_extraido);
}
imprimir_gantt(diagrama_gantt, indice);
generar_archivo_gantt(diagrama_gantt, indice, "gantt.eps");
```

Definition at line 43 of file memoria.c.

5.11.2.3 ejecutar_proceso()

```
void ejecutar_proceso (
            BloqueMemoria * memoria,
            Proceso * proceso,
            int posicion)
```

Ejecuta el PROCESO.

Simula la ejecución de un PROCESO.

Parameters

memoria	Arreglo de BLOQUES de MEMORIA.
proceso	Proceso.
posicion	Posición.

SIMULA la ejecución del PROCESO y luego LIBERA la MEMORIA del PROCESO.

```
sleep(proceso->tiempo_rafaga);
liberar_memoria(memoria, posicion);
```

Definition at line 224 of file memoria.c.

5.11.2.4 inicializar_bloques_memoria()

```
void inicializar_bloques_memoria (
            BloqueMemoria * memoria,
            int cantidad_bloques,
            int tamano_bloque)
```

< Librería que contiene las funciones auxiliares.

Inicializa los BLOQUES de MEMORIA.

Inicializa los BLOQUES de MEMORIA.

Parameters

memoria	Arreglo de BLOQUES de MEMORIA.
cantidad_bloques	Cantidad de BLOQUES de MEMORIA.
tamano_bloque	Tamaño de los BLOQUES de MEMORIA.

Recorre el ARREGLO de MEMORIA y asigna el TAMAÑO del BLOQUE y el ESTADO del BLOQUE.

```
for (int i = 0; i < cantidad_bloques; i++)
{
    memoria[i].tamano = tamano_bloque;
    memoria[i].estado = 1;
}
```

Definition at line 18 of file memoria.c.

5.11.2.5 liberar_memoria()

```
void liberar_memoria (
            BloqueMemoria * memoria,
            int posicion)
```

Libera la MEMORIA de un PROCESO.

Parameters

memoria	Arreglo de BLOQUES de MEMORIA.
posicion	Posición.

LIBERA la MEMORIA de un PROCESO y la MARCA como LIBRE (1).

```
memoria[posicion].estado = 1;
memoria[posicion].tamano = 128;
```

Definition at line 242 of file memoria.c.

5.11.2.6 manejar_proceso()

```
void manejar_proceso (
            BloqueMemoria * memoria,
            Proceso * proceso_extraido,
            int cantidad_bloques,
            Gantt * diagrama_gantt,
            int * tiempo_global,
            int * indice)
```

Maneja el PROCESO, asigna MEMORIA y REGISTRA los tiempos de los procesos.

Maneja el PROCESO.

Parameters

memoria	Arreglo de BLOQUES de MEMORIA.
proceso_extraido	Proceso extraído de la COLA.
cantidad_bloques	Cantidad de BLOQUES de MEMORIA.
diagrama_gantt	Diagrama de Gantt.
tiempo_global	Tiempo global.
indice	Índice del diagrama de Gantt.

Se GUARDA el TIEMPO de INICIO del PROCESO y el TAMAÑO del PROCESO. Si el TAMAÑO del PROCESO es mayor a 2048, se imprime un MENSAJE de ERROR. Se RECORRE los BLOQUES de MEMORIA y se ASIGNA la MEMORIA al PROCESO.

```
int tiempo_inicio = *tiempo_global;
int tamano_proceso = proceso_extraido->memoria_solicitada;
if (tamano_proceso > 2048)
{
    fprintf(stderr, "No hay suficiente memoria disponible para asignar al PROCESO %d, pasando al siguiente
      PROCESO.\n\n", proceso_extraido->pid);
    return;
}
for (int i = 0; i < cantidad_bloques && tamano_proceso > 0; i++)
{
    if (memoria[i].estado == 1)
    {
        asignar_bloque_proceso(memoria, proceso_extraido, &tamano_proceso, i);
    }
}
*tiempo_global += proceso_extraido->tiempo_rafaga;
registrar_tiempos(diagrama_gantt, proceso_extraido->pid, tiempo_inicio, proceso_extraido->tiempo_rafaga,
      indice);
fprintf(stdout, "Proceso %d EJECUTADO EXITOSAMENTE.\n\n", proceso_extraido->pid);
```

Definition at line 103 of file memoria.c.

5.11.2.7 verificar_fragmentacion()

```
void verificar_fragmentacion (
            BloqueMemoria * memoria,
            int i)
```

Verifica la FRAGMENTACIÓN INTERNA.

Parameters

memoria	Arreglo de BLOQUES de MEMORIA.
i	Índice.

Calcula la FRAGMENTACIÓN INTERNA. Si el tamaño de la memoria del PROCESO es mayor a 0, hay fragmentación interna.

```
int fragmentacion_interna = memoria[i].tamano;
if (fragmentacion_interna > 0)
    fprintf(stdout, "FRAGMENTACIÓN INTERNA: %d KB en el BLOQUE %d\n", fragmentacion_interna, i);
```

Definition at line 203 of file memoria.c.

5.12 memoria.c

Go to the documentation of this file.

```
00001
00010 #include "auxiliar.h"
00011
00018 void inicializar_bloques_memoria(BloqueMemoria *memoria, int cantidad_bloques, int tamano_bloque)
00019 {
00030     for (int i = 0; i < cantidad_bloques; i++)
00031     {
00032         memoria[i].tamano = tamano_bloque;
00033         memoria[i].estado = 1;
00034     }
00035 }
00036
00043 void asignar_memoria_procesos(Cola *cola, BloqueMemoria *memoria, int cantidad_bloques)
00044 {
00055     Cola *temp_cola = malloc(sizeof(Cola));
00056     temp_cola = cola;
00057     Proceso *proceso_extraido = malloc(sizeof(Proceso));
00058     Gantt diagrama_gantt[MAX_PROCESOS];
00059     int indice = 0, tiempo_global = 0;
00060
00079     while (temp_cola->front != NULL)
00080     {
00081         proceso_extraido = dequeue(temp_cola);
00082         if (proceso_extraido == NULL)
00083         {
00084             fprintf(stdout, "No hay procesos en la cola para asignar memoria.\n");
00085             return;
00086         }
00087         manejar_proceso(memoria, proceso_extraido, cantidad_bloques, diagrama_gantt, &tiempo_global,
      &indice);
00088         free(proceso_extraido);
00089     }
00090     imprimir_gantt(diagrama_gantt, indice);
00091     generar_archivo_gantt(diagrama_gantt, indice, "gantt.eps");
00092 }
00093
00103 void manejar_proceso(BloqueMemoria *memoria, Proceso *proceso_extraido, int cantidad_bloques, Gantt
      *diagrama_gantt, int *tiempo_global, int *indice)
00104 {
00127     int tiempo_inicio = *tiempo_global;
00128     int tamano_proceso = proceso_extraido->memoria_solicitada;
00129     if (tamano_proceso > 2048)
00130     {
00131         fprintf(stderr, "No hay suficiente memoria disponible para asignar al PROCESO %d, pasando al
      siguiente PROCESO.\n\n", proceso_extraido->pid);
00132         return;
```

```
00133     }
00134     for (int i = 0; i < cantidad_bloques && tamano_proceso > 0; i++)
00135     {
00136         if (memoria[i].estado == 1)
00137         {
00138             asignar_bloque_proceso(memoria, proceso_extraido, &tamano_proceso, i);
00139         }
00140     }
00141     *tiempo_global += proceso_extraido->tiempo_rafaga;
00142     registrar_tiempos(diagrama_gantt, proceso_extraido->pid, tiempo_inicio,
      proceso_extraido->tiempo_rafaga, indice);
00143     fprintf(stdout, "Proceso %d EJECUTADO EXITOSAMENTE.\n\n", proceso_extraido->pid);
00144 }
00145
00153 void asignar_bloque_proceso(BloqueMemoria *memoria, Proceso *proceso, int *tamano_proceso, int i)
00154 {
00178     if (memoria[i].tamano >= *tamano_proceso)
00179     {
00180         fprintf(stdout, "El proceso %d usó MEMORIA hasta el BLOQUE %d\n", proceso->pid, i);
00181         memoria[i].tamano -= *tamano_proceso;
00182         memoria[i].estado = 0;
00183         *tamano_proceso = 0;
00184         verificar_fragmentacion(memoria, i);
00185         ejecutar_proceso(memoria, proceso, i);
00186     }
00187     else
00188     {
00189         *tamano_proceso -= memoria[i].tamano;
00190         memoria[i].tamano = 0;
00191         memoria[i].estado = 0;
00192         fprintf(stdout, "Proceso %d en EJECUCIÓN, bloque (%d)\n", proceso->pid, i);
00193         ejecutar_proceso(memoria, proceso, i);
00194         verificar_fragmentacion(memoria, i);
00195     }
00196 }
00197
00203 void verificar_fragmentacion(BloqueMemoria *memoria, int i)
00204 {
00213     int fragmentacion_interna = memoria[i].tamano;
00214     if (fragmentacion_interna > 0)
00215         fprintf(stdout, "FRAGMENTACIÓN INTERNA: %d KB en el BLOQUE %d\n", fragmentacion_interna, i);
00216 }
00217
00224 void ejecutar_proceso(BloqueMemoria *memoria, Proceso *proceso, int posicion)
00225 {
00233     sleep(proceso->tiempo_rafaga);
00234     liberar_memoria(memoria, posicion);
00235 }
00236
00242 void liberar_memoria(BloqueMemoria *memoria, int posicion)
00243 {
00251     memoria[posicion].estado = 1;
00252     memoria[posicion].tamano = 128;
00253 }
```

5.13 src/planificador.c File Reference

Funciones de planificación.

```
#include "auxiliar.h"
```

Functions

- void enqueue (Cola *cola, Proceso *proceso)

  < *Librería que contiene las funciones auxiliares.*
- Proceso * dequeue (Cola *cola)

  Extrae un PROCESO de la COLA.

5.13.1 Detailed Description

Funciones de planificación.

Date

24-10-2024

Authors

Miguel Loaiza, Diego Sanhueza y Oscar Cifuentes

Contiene las funciones que son utilizadas para la correcta PLANIFICACIÓN de los PROCESOS.

Definition in file planificador.c.

5.13.2 Function Documentation

5.13.2.1 dequeue()

```
Proceso * dequeue (
            Cola * cola)
```

Extrae un PROCESO de la COLA.

Parameters

cola	Cola de PROCESOS.

Returns

Proceso* Proceso extraído.

Si el FRENTE de la COLA es NULL, la COLA está VACÍA. Se imprime un MENSAJE de ERROR y se SALE del programa. Se EXTRAE el PROCESO del FRENTE de la COLA y se actualiza el FRENTE de la COLA. Si el FRENTE de la COLA es NULL, el FINAL de la COLA también es NULL (COLA VACÍA).

```
if (cola->front == NULL)
{
    fprintf(stdout, "La cola está vacía.\n");
    exit(EXIT_FAILURE);
}
```

Definition at line 52 of file planificador.c.

5.13.2.2 enqueue()

```
void enqueue (
            Cola * cola,
            Proceso * proceso)
```

< Librería que contiene las funciones auxiliares.

Coloca un PROCESO a la COLA.

Coloca un PROCESO en la COLA.

Parameters

id	ID del proceso.
tiempo_llegada	Tiempo de llegada del proceso.
tiempo_rafaga	Tiempo de ráfaga del proceso.
memoria_solicitada	Memoria solicitada por el proceso.
cola_procesos	Cola de PROCESOS.

Si la COLA está VACÍA, el FRENTE y el FINAL de la COLA es el PROCESO. Se actualiza el SIGUIENTE PROCESO con NULL porque NO hay otro proceso DETRÁS. Imprime la cola de procesos.

```
if (cola->rear == NULL)
    cola->front = proceso;
else
    cola->rear->next = proceso;
cola->rear = proceso;
proceso->next = NULL;
fprintf(stdout, "Proceso %d encolado CORRECTANENTE.\n", proceso->pid);
imprimir_cola_procesos(cola);
fprintf(stdout, "\n");
```

Definition at line 20 of file planificador.c.

5.14 planificador.c

Go to the documentation of this file.

```
00001
00010 #include "auxiliar.h"
00011
00020 void enqueue(Cola *cola, Proceso *proceso)
00021 {
00036     if (cola->rear == NULL)
00037         cola->front = proceso;
00038     else
00039         cola->rear->next = proceso;
00040     cola->rear = proceso;
00041     proceso->next = NULL;
00042     fprintf(stdout, "Proceso %d encolado CORRECTANENTE.\n", proceso->pid);
00043     imprimir_cola_procesos(cola);
00044     fprintf(stdout, "\n");
00045 }
00046
00052 Proceso *dequeue(Cola *cola)
00053 {
00064     if (cola->front == NULL)
00065     {
00066         fprintf(stdout, "La cola está vacía.\n");
00067         exit(EXIT_FAILURE);
00068     }
00069     Proceso *proceso_extraido = cola->front;
00070     cola->front = cola->front->next;
00071     if (cola->front == NULL)
00072         cola->rear = NULL;
00073     return proceso_extraido;
00074 }
```

Index