

Gestor de Memoria - Práctica 3 OS

1.0

Generado por Doxygen 1.15.0

1 Gestmemoria - Simulador de Gestión de Memoria	1
1.1 Descripción	1
1.2 Inicio Rápido	1
1.3 Documentación	1
1.4 Algoritmos Implementados	1
1.5 Requisitos	2
1.6 Compilación	2
1.7 Uso	2
1.7.1 Controles (GUI)	2
1.8 Formato del Archivo de Entrada	2
1.8.1 Ejemplo	2
1.9 Arquitectura del Proyecto	3
1.10 Constantes de Configuración	3
1.11 Estadísticas Futuras	4
1.12 Autor	4
1.13 Licencia	4
1.14 Licencia	4
2 Guía de Instalación	5
2.1 macOS	5
2.1.1 Instalar dependencias con Homebrew	5
2.1.2 Compilar el proyecto	5
2.1.3 Ejecutar	5
2.2 Linux (Ubuntu/Debian)	5
2.3 Problemas comunes	5
2.3.1 Error: "raylib not found"	5
2.3.2 Error: "fork() failed"	6
2.3.3 Error de compilación con C17	6
3 Arquitectura del Sistema	7
3.1 Diagrama de Módulos	7
3.2 Flujo de Ejecución	7
3.3 Estructuras de Datos Principales	8
3.3.1 Memoria	8
3.3.2 Proceso	8
3.3.3 Partición	8
3.4 Algoritmos de Asignación	8
3.4.1 First Fit (Primer Hueco)	8
3.4.2 Next Fit (Siguiente Hueco)	8
3.5 Gestión de Memoria	9
3.5.1 Alineación	9
3.5.2 Compactación	9
3.6 Uso de fork()	9

4 Changelog	11
4.1 [1.0.0] - 2025-12-19	11
4.1.1 Añadido	11
4.1.2 Características Técnicas	12
4.2 Pendiente (TODO)	12
4.2.1 Algoritmos	12
4.2.2 Estadísticas	12
4.2.3 Interfaz	12
4.2.4 Testing	12
4.3 Notas de Desarrollo	12
4.3.1 Estructura del proyecto	12
4.3.2 Generar documentación	12
5 Topic Index	13
5.1 Topics	13
6 Jerarquía de directorios	15
6.1 Directorios	15
7 Índice de estructuras de datos	17
7.1 Estructuras de datos	17
8 Índice de archivos	19
8.1 Lista de archivos	19
9 Topic Documentation	21
9.1 Constantes de Configuración	21
9.1.1 Descripción detallada	21
9.1.2 Documentación de «define»	21
9.1.2.1 MAX_PARTICIONES	21
9.1.2.2 MAX_PROCESOS	21
9.1.2.3 MEMORIA_TOTAL	21
9.1.2.4 UNIDAD_MINIMA	22
10 Documentación de directorios	23
10.1 Referencia del directorio docs	23
10.2 Referencia del directorio src	23
11 Documentación de estructuras de datos	25
11.1 Referencia de la estructura Memoria	25
11.1.1 Descripción detallada	25
11.1.2 Documentación de campos	25
11.1.2.1 cant_particiones	25
11.1.2.2 particiones	26
11.1.2.3 ultimo_indice_asignado	26
11.2 Referencia de la estructura Particion	26

11.2.1 Descripción detallada	26
11.2.2 Documentación de campos	26
11.2.2.1 dir_inicio	26
11.2.2.2 estado	27
11.2.2.3 nombre_proceso	27
11.2.2.4 tamano	27
11.3 Referencia de la estructura Proceso	27
11.3.1 Descripción detallada	27
11.3.2 Documentación de campos	28
11.3.2.1 en_memoria	28
11.3.2.2 finalizado	28
11.3.2.3 mem_requerida	28
11.3.2.4 nombre	28
11.3.2.5 t_ejecucion	28
11.3.2.6 t_final	28
11.3.2.7 t_llegada	28
11.3.2.8 t_restante	28
12 Documentación de archivos	29
12.1 Referencia del archivo docs/ARCHITECTURE.md	29
12.2 Referencia del archivo docs/CHANGELOG.md	29
12.3 Referencia del archivo docs/INSTALL.md	29
12.4 Referencia del archivo README.md	29
12.5 Referencia del archivo src/ficheros.c	29
12.5.1 Documentación de «define»	30
12.5.1.1 SIZE_BUFFER_LECTURA	30
12.5.2 Documentación de funciones	30
12.5.2.1 cargar_procesos()	30
12.5.2.2 guardar_estado()	31
12.5.2.3 limpiar_log()	32
12.6 ficheros.c	32
12.7 Referencia del archivo src/ficheros.h	33
12.7.1 Descripción detallada	34
12.7.2 Documentación de funciones	35
12.7.2.1 cargar_procesos()	35
12.7.2.2 guardar_estado()	36
12.7.2.3 limpiar_log()	37
12.8 ficheros.h	37
12.9 Referencia del archivo src/main.c	37
12.9.1 Descripción detallada	38
12.9.2 Documentación de «define»	39
12.9.2.1 ALTO_BARRA	39
12.9.2.2 MARGEN_DER	39

12.9.2.3 MARGEN_IZQ	39
12.9.2.4 WIN_HEIGHT	39
12.9.2.5 WIN_WIDTH	39
12.9.2.6 Y_BARRA	39
12.9.3 Documentación de funciones	39
12.9.3.1 main()	39
12.9.3.2 run_gui()	40
12.9.3.3 test_sim()	41
12.10 main.c	42
12.11 Referencia del archivo src/sim_engine.c	46
12.11.1 Documentación de funciones	47
12.11.1.1 alinear_size()	47
12.11.1.2 asignar_proceso()	48
12.11.1.3 avanzar_tiempo()	49
12.11.1.4 buscar_hueco()	50
12.11.1.5 compactar()	51
12.11.1.6 inicializar_memoria()	52
12.11.1.7 liberar_proceso()	52
12.11.1.8 mostrar_estado()	53
12.11.1.9 ocupar_memoria()	54
12.12 sim_engine.c	55
12.13 Referencia del archivo src/sim_engine.h	58
12.13.1 Descripción detallada	59
12.13.2 Documentación de enumeraciones	60
12.13.2.1 TipoAlgo	60
12.13.3 Documentación de funciones	60
12.13.3.1 alinear_size()	60
12.13.3.2 asignar_proceso()	61
12.13.3.3 avanzar_tiempo()	62
12.13.3.4 buscar_hueco()	63
12.13.3.5 compactar()	64
12.13.3.6 inicializar_memoria()	65
12.13.3.7 liberar_proceso()	65
12.13.3.8 mostrar_estado()	66
12.13.3.9 ocupar_memoria()	67
12.14 sim_engine.h	68
13 Ejemplos	71
13.1 /Users/julianhinojosagil/Documents/Dev/noob-code/UA/year-2/OS/practica3/src/sim_engine.h	71
Índice alfabético	73

Capítulo 1

GestOMEMORIA - Simulador de Gestión de Memoria

1.1. Descripción

Simulador visual de algoritmos de gestión de memoria dinámica con particiones variables. Proyecto de la asignatura Sistemas Operativos - Universidad de Alicante.

1.2. Inicio Rápido

```
# Compilar  
mkdir build && cd build  
cmake .. && make  
  
# Ejecutar  
./gestOMEMORIA
```

1.3. Documentación

- [Guía de Instalación](#)
- [Arquitectura del Sistema](#)
- [Changelog](#)

1.4. Algoritmos Implementados

Algoritmo	Descripción
First Fit	Asigna el primer hueco que sea suficientemente grande
Next Fit	Como First Fit, pero empieza desde la última posición asignada

1.5. Requisitos

- CMake 3.11+
- [Raylib](#) (para la interfaz gráfica)
- Compilador C con soporte C17

1.6. Compilación

```
mkdir build && cd build
cmake ..
make
```

1.7. Uso

```
./gestomemoria
```

El programa ejecuta dos procesos en paralelo:

- [Proceso](#) hijo: Interfaz gráfica (GUI) con Raylib
- [Proceso](#) padre: Interfaz de terminal (TUI) para depuración

1.7.1. Controles (GUI)

Tecla	Acción
ESPACIO	Avanzar un tick de simulación
P	Activar/desactivar reproducción automática
R	Reiniciar simulación
ESC	Salir

1.8. Formato del Archivo de Entrada

El archivo entrada.txt debe seguir este formato:

```
<nombre_proceso> <instante_llegada> <memoria_requerida> <tiempo_ejecucion>
```

1.8.1. Ejemplo

```
P1 0 243 5
P2 1 150 3
P3 2 500 4
```

1.9. Arquitectura del Proyecto

```
practica3/
src/
    main.c      # Punto de entrada, GUI (hijo) y TUI (padre)
    sim_engine.c/h # Motor de simulación
    ficheros.c/h # Entrada/Salida de archivos
docs/
    INSTALL.md   # Guía de instalación
    ARCHITECTURE.md # Arquitectura del sistema
    CHANGELOG.md   # Historial de cambios
    entrada.txt    # Archivo de procesos de ejemplo
    CMakeLists.txt # Configuración de CMake
    Doxyfile       # Configuración de Doxygen
    README.md      # Este archivo
```

1.10. Constantes de Configuración

Constante	Valor	Descripción
MEMORIA_TOTAL	2000	Tamaño total de memoria simulada

Constante	Valor	Descripción
UNIDAD_MINIMA	100	Alineación mínima de asignación
MAX_PARTICIONES	50	Máximo de particiones simultáneas
MAX_PROCESOS	100	Máximo de procesos en simulación

1.11. Estadísticas Futuras

- Fragmentación externa
- Tiempo medio de espera
- Uso de memoria a lo largo del tiempo

1.12. Autor

Julian Hinojosa Gil - 2025

1.13. Licencia

Proyecto académico - Universidad de Alicante

1.14. Licencia

Proyecto académico - Universidad de Alicante

Capítulo 2

Guía de Instalación

2.1. macOS

2.1.1. Instalar dependencias con Homebrew

```
# Instalar Raylib
brew install raylib

# Instalar CMake (si no lo tienes)
brew install cmake

# Opcional: Graphviz para generar documentación con diagramas
brew install graphviz
```

2.1.2. Compilar el proyecto

```
cd /ruta/al/proyecto
mkdir build && cd build
cmake ..
make
```

2.1.3. Ejecutar

```
./gestomemoria
```

2.2. Linux (Ubuntu/Debian)

```
# Instalar dependencias
sudo apt update
sudo apt install cmake libraylib-dev

# Compilar
mkdir build && cd build
cmake ..
make
```

2.3. Problemas comunes

2.3.1. Error: "raylib not found"

Asegúrate de que Raylib está instalado correctamente:

```
# macOS
brew reinstall raylib

# Linux
sudo apt install libraylib-dev
```

2.3.2. Error: "fork() failed"

El sistema no tiene recursos suficientes. Cierra otras aplicaciones.

2.3.3. Error de compilación con C17

Asegúrate de tener un compilador compatible:

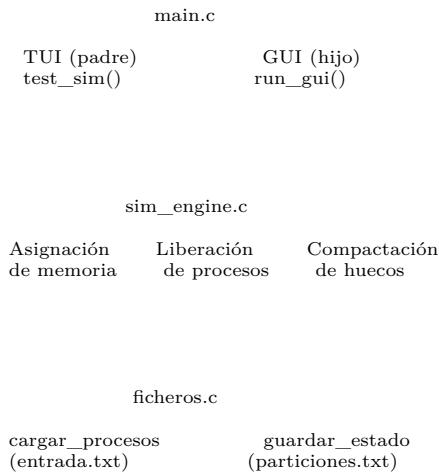
```
# macOS (Xcode Command Line Tools)  
xcode-select --install
```

```
# Linux  
sudo apt install build-essential
```

Capítulo 3

Arquitectura del Sistema

3.1. Diagrama de Módulos



3.2. Flujo de Ejecución

1. Inicio: `main()` ejecuta `fork()`
2. **Proceso** hijo (`pid == 0`): Ejecuta la GUI con Raylib
3. **Proceso** padre (`pid > 0`): Ejecuta la TUI en terminal y espera al hijo
4. Cada tick de simulación:
 - Se envejecen los procesos en memoria (decrementan `t_restante`)
 - Se liberan procesos que han terminado (`t_restante == 0`)
 - Se intentan asignar procesos de la cola de espera
 - Se guarda el estado en el log

3.3. Estructuras de Datos Principales

3.3.1. Memoria

La estructura central que gestiona las particiones:

```
typedef struct {
    Particion particiones[MAX_PARTICIONES]; // Array de particiones
    int cant_particiones;                  // Número actual
    int ultimo_indice_asignado;           // Para Next Fit
} Memoria;
```

3.3.2. Proceso

Representa un proceso en la simulación:

```
typedef struct {
    char nombre[10];          // Identificador único (ej: "P1")
    int t_llegada;            // Instante de llegada
    int mem_requerida;        // Memoria solicitada
    int t_ejecucion;           // Tiempo total de ejecución
    int t_restante;            // Ticks restantes
    bool en_memoria;           // ¿Está cargado?
    bool finalizado;           // ¿Ha terminado?
} Proceso;
```

3.3.3. Partición

Representa un bloque de memoria (hueco o proceso):

```
typedef struct {
    int dir_inicio;           // Dirección base
    int tamano;                // Tamaño del bloque
    int estado;                 // 0=HUECO, 1=OCUPADO
    char nombre_proceso[10];   // Nombre o "HUECO"
} Particion;
```

3.4. Algoritmos de Asignación

3.4.1. First Fit (Primer Hueco)

Busca secuencialmente desde el inicio de la memoria:

```
Para i = 0 hasta cant_particiones:
    Si particiones[i].estado == HUECO Y
        particiones[i].tamano >= mem_requerida:
            Retornar i
    Retornar -1 (no encontrado)
```

Ventajas: Simple, rápido para memorias pequeñas Desventajas: Fragmentación al inicio de la memoria

3.4.2. Next Fit (Siguiente Hueco)

Busca desde la última posición asignada (circular):

```
inicio = ultimo_indice_asignado
Para j = 0 hasta cant_particiones:
    i = (inicio + j) % cant_particiones
    Si particiones[i].estado == HUECO Y
        particiones[i].tamano >= mem_requerida:
            ultimo_indice_asignado = i
            Retornar i
    Retornar -1 (no encontrado)
```

Ventajas: Distribuye mejor la fragmentación Desventajas: Puede ser más lento en el peor caso

3.5. Gestión de Memoria

3.5.1. Alineación

Toda memoria se alinea a múltiplos de **UNIDAD_MINIMA** (100):

```
// Ejemplo: solicitar 243 → se asignan 300
int alinear_size(int size) {
    int bloques = size / UNIDAD_MINIMA;
    if (size % UNIDAD_MINIMA != 0)
        bloques++;
    return bloques * UNIDAD_MINIMA;
}
```

3.5.2. Compactación

Cuando se libera un proceso, se fusionan huecos adyacentes:

Para cada par de particiones consecutivas:
Si ambas son HUECO:

- Fusionar (sumar tamaños)
- Eliminar la segunda
- Repetir desde el inicio

3.6. Uso de fork()

El programa usa fork() para ejecutar GUI y TUI en paralelo:

```
pid_t pid = fork();

if (pid == 0) {
    // Hijo: GUI con Raylib
    run_gui(&m, procesos, num_procesos);
    _exit(0);
} else {
    // Padre: TUI en terminal
    test_sim();
    waitpid(pid, NULL, 0); // Esperar al hijo
}
```

Esto permite depurar en terminal mientras se visualiza en la GUI.

Capítulo 4

Changelog

Todos los cambios notables de este proyecto se documentan aquí.

El formato está basado en [Keep a Changelog](#).

4.1. [1.0.0] - 2025-12-19

4.1.1. Añadido

- Simulador de memoria con particiones variables
- Algoritmo First Fit (Primer Hueco)
- Algoritmo Next Fit (Siguiente Hueco)
- Interfaz gráfica con Raylib
 - Visualización de barra de memoria con colores
 - Lista de procesos y su estado
 - Controles interactivos (ESPACIO, P, R, ESC)
 - Auto-play con velocidad configurable
- Interfaz de terminal (TUI) para depuración
- Sistema de logging a archivo (particiones.txt)
- Compactación automática de huecos adyacentes
- Alineación de memoria a múltiplos de 100
- Ejecución paralela con fork() (GUI hijo, TUI padre)
- Documentación completa con Doxygen
 - Grafos de llamadas (call graphs)
 - Navegador de código fuente
 - Página principal con README

4.1.2. Características Técnicas

- Uso de llamadas POSIX (open, read, write, fork)
 - [Memoria](#) máxima: 2000 unidades
 - Hasta 50 particiones simultáneas
 - Hasta 100 procesos por simulación
-

4.2. Pendiente (TODO)

4.2.1. Algoritmos

- Best Fit (Mejor Hueco)
- Worst Fit (Peor Hueco)

4.2.2. Estadísticas

- Cálculo de fragmentación externa
- Tiempo medio de espera
- Tiempo medio de retorno
- Uso de memoria a lo largo del tiempo

4.2.3. Interfaz

- Cargar archivo de entrada desde GUI (file picker)
- Selector de algoritmo en GUI
- Modo paso a paso con retroceso (undo)
- Zoom en la barra de memoria
- Exportar estadísticas a CSV

4.2.4. Testing

- Tests unitarios con CUnit o similar
 - Tests de regresión para algoritmos
-

4.3. Notas de Desarrollo

4.3.1. Estructura del proyecto

```
practica3/
src/          # Código fuente
docs/         # Documentación
build/        # Archivos compilados
entrada.txt   # Archivo de prueba
```

4.3.2. Generar documentación

```
doxygen Doxyfile
open docs/html/index.html
```

Capítulo 5

Topic Index

5.1. Topics

Here is a list of all topics with brief descriptions:

Constantes de Configuración	21
---------------------------------------	----

Capítulo 6

Jerarquía de directorios

6.1. Directorios

docs	23
src	23
ficheros.c	29
ficheros.h	33
main.c	37
sim_engine.c	46
sim_engine.h	58

Capítulo 7

Índice de estructuras de datos

7.1. Estructuras de datos

Lista de estructuras con breves descripciones:

Memoria	Estructura principal que representa la memoria del sistema	25
Particion	Estructura que representa una partición de memoria	26
Proceso	Estructura que representa un proceso en el simulador	27

Capítulo 8

Índice de archivos

8.1. Lista de archivos

Lista de todos los archivos con breves descripciones:

src/ ficheros.c	29
src/ ficheros.h	
Módulo de entrada/salida para la simulación de memoria	33
src/ main.c	
Punto de entrada del simulador de gestión de memoria	37
src/ sim_engine.c	46
src/ sim_engine.h	
Motor de simulación de gestión de memoria con particiones variables	58

Capítulo 9

Topic Documentation

9.1. Constantes de Configuración

defines

- `#define MEMORIA_TOTAL 2000`
Tamaño total de la memoria simulada (en unidades).
- `#define UNIDAD_MINIMA 100`
Unidad mínima de asignación. Toda memoria se alinea a múltiplos de este valor.
- `#define MAX_PARTICIONES 50`
Máximo número de particiones simultáneas en memoria.
- `#define MAX_PROCESOS 100`
Máximo número de procesos que puede manejar la simulación.

9.1.1. Descripción detallada

Valores que determinan los límites del simulador.

9.1.2. Documentación de «define»

9.1.2.1. MAX_PARTICIONES

`#define MAX_PARTICIONES 50`
Máximo número de particiones simultáneas en memoria.

Ejemplos

`/Users/julianhinojosagil/Documents/Dev/noob-code/UA/year-2/OS/practica3/src/sim_engine.h.`

Definición en la línea 42 del archivo `sim_engine.h`.

Referenciado por `ocupar_memoria()`.

9.1.2.2. MAX_PROCESOS

`#define MAX_PROCESOS 100`
Máximo número de procesos que puede manejar la simulación.
Definición en la línea 45 del archivo `sim_engine.h`.
Referenciado por `cargar_procesos()`, `main()` y `test_sim()`.

9.1.2.3. MEMORIA_TOTAL

`#define MEMORIA_TOTAL 2000`
Tamaño total de la memoria simulada (en unidades).
Definición en la línea 36 del archivo `sim_engine.h`.
Referenciado por `inicializar_memoria()`, `run_gui()` y `test_sim()`.

9.1.2.4. UNIDAD_MINIMA

```
#define UNIDAD_MINIMA 100
```

Unidad mínima de asignación. Toda memoria se alinea a múltiplos de este valor.

Definición en la línea [39](#) del archivo [sim_engine.h](#).

Referenciado por [alinear_size\(\)](#).

Capítulo 10

Documentación de directorios

10.1. Referencia del directorio docs

10.2. Referencia del directorio src

Archivos

- archivo [ficheros.c](#)
- archivo [ficheros.h](#)

Módulo de entrada/salida para la simulación de memoria.

- archivo [main.c](#)

Punto de entrada del simulador de gestión de memoria.

- archivo [sim_engine.c](#)
- archivo [sim_engine.h](#)

Motor de simulación de gestión de memoria con particiones variables.

Capítulo 11

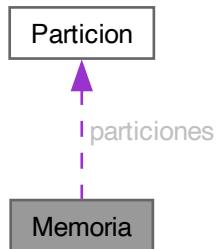
Documentación de estructuras de datos

11.1. Referencia de la estructura Memoria

Estructura principal que representa la memoria del sistema.

```
#include <sim_engine.h>
```

Diagrama de colaboración de Memoria:



Campos de datos

- `Particion particiones [MAX_PARTICIONES]`
- `int cant_particiones`
- `int ultimo_indice_asignado`

11.1.1. Descripción detallada

Estructura principal que representa la memoria del sistema.

Gestiona un array de particiones que cubren todo el espacio de memoria. Inicialmente contiene una única partición (hueco) de tamaño MEMORIA_TOTAL.

Ejemplos

```
/Users/julianhinojosagil/Documents/Dev/noob-code/UA/year-2/OS/practica3/src/sim_engine.h.
```

Definición en la línea 90 del archivo `sim_engine.h`.

11.1.2. Documentación de campos

11.1.2.1. `cant_particiones`

```
int Memoria::cant_particiones
```

Número actual de particiones (1 a MAX_PARTICIONES)

Definición en la línea 92 del archivo `sim_engine.h`.

Referenciado por `buscar_hueco()`, `compactar()`, `guardar_estado()`, `inicializar_memoria()`, `liberar_proceso()`, `mostrar_estado()`, `ocupar_memoria()` y `run_gui()`.

11.1.2.2. particiones

`Particion Memoria::particiones[MAX_PARTICIONES]`

Array de particiones (huecos y procesos)

Definición en la línea 91 del archivo `sim_engine.h`.

Referenciado por `buscar_hueco()`, `compactar()`, `guardar_estado()`, `inicializar_memoria()`, `liberar_proceso()`, `mostrar_estado()`, `ocupar_memoria()` y `run_gui()`.

11.1.2.3. ultimo_index_asignado

`int Memoria::ultimo_index_asignado`

Último índice usado (para Next Fit)

Definición en la línea 93 del archivo `sim_engine.h`.

Referenciado por `buscar_hueco()`, `inicializar_memoria()` y `ocupar_memoria()`.

La documentación de esta estructura está generada del siguiente archivo:

- `src/sim_engine.h`

11.2. Referencia de la estructura Particion

Estructura que representa una partición de memoria.

```
#include <sim_engine.h>
```

Campos de datos

- `int dir_inicio`
- `int tamano`
- `int estado`
- `char nombre_proceso [10]`

11.2.1. Descripción detallada

Estructura que representa una partición de memoria.

Una partición puede ser:

- Un hueco libre (estado=0, nombre="HUECO")
- Un bloque ocupado por un proceso (estado=1)

Ejemplos

```
/Users/julianhinojosagil/Documents/Dev/noob-code/UA/year-2/OS/practica3/src/sim_engine.h.
```

Definición en la línea 77 del archivo `sim_engine.h`.

11.2.2. Documentación de campos

11.2.2.1. dir_inicio

`int Particion::dir_inicio`

Dirección de inicio en memoria (0 a MEMORIA_TOTAL-1)

Definición en la línea 78 del archivo `sim_engine.h`.

Referenciado por `guardar_estado()`, `inicializar_memoria()`, `liberar_proceso()`, `mostrar_estado()`, `ocupar_memoria()` y `run_gui()`.

11.2.2.2. estado

```
int Particion::estado
Estado: 0=HUECO (libre), 1=PROCESO (ocupado)
Definición en la línea 80 del archivo sim\_engine.h.
Referenciado por buscar\_hueco\(\), compactar\(\), inicializar\_memoria\(\), liberar\_proceso\(\), ocupar\_memoria\(\) y run\_gui\(\).
```

11.2.2.3. nombre_proceso

```
char Particion::nombre_proceso[10]
Nombre del proceso o "HUECO" si está libre
Definición en la línea 81 del archivo sim\_engine.h.
Referenciado por compactar\(\), guardar\_estado\(\), inicializar\_memoria\(\), liberar\_proceso\(\), mostrar\_estado\(\), ocupar\_memoria\(\) y run\_gui\(\).
```

11.2.2.4. tamano

```
int Particion::tamano
Tamaño de la partición (múltiplo de UNIDAD_MINIMA)
Definición en la línea 79 del archivo sim\_engine.h.
Referenciado por buscar\_hueco\(\), compactar\(\), guardar\_estado\(\), inicializar\_memoria\(\), mostrar\_estado\(\), ocupar\_memoria\(\) y run\_gui\(\).
La documentación de esta estructura está generada del siguiente archivo:
```

- [src/sim_engine.h](#)

11.3. Referencia de la estructura Proceso

Estructura que representa un proceso en el simulador.

```
#include <sim_engine.h>
```

Campos de datos

- [char nombre \[10\]](#)
- [int t_llegada](#)
- [int mem_requerida](#)
- [int t_ejecucion](#)
- [int t_final](#)
- [int t_restante](#)
- [bool en_memoria](#)
- [bool finalizado](#)

11.3.1. Descripción detallada

Estructura que representa un proceso en el simulador.

Contiene toda la información necesaria para gestionar un proceso:

- Identificación y tiempos
- Requisitos de memoria
- Estado actual en la simulación

Ejemplos

```
/Users/julianhinojosagil/Documents/Dev/noob-code/UA/year-2/OS/practica3/src/sim\_engine.h.
```

Definición en la línea 57 del archivo [sim_engine.h](#).

11.3.2. Documentación de campos

11.3.2.1. en_memoria

bool Proceso::en_memoria
 true si está cargado en memoria
 Definición en la línea 66 del archivo [sim_engine.h](#).
 Referenciado por [avanzar_tiempo\(\)](#) y [cargar_procesos\(\)](#).

11.3.2.2. finalizado

bool Proceso::finalizado
 true si ya completó su ejecución
 Definición en la línea 67 del archivo [sim_engine.h](#).
 Referenciado por [avanzar_tiempo\(\)](#) y [cargar_procesos\(\)](#).

11.3.2.3. mem_requerida

int Proceso::mem_requerida
Memoria solicitada (se alineará a UNIDAD_MINIMA)
 Definición en la línea 60 del archivo [sim_engine.h](#).
 Referenciado por [asignar_proceso\(\)](#), [cargar_procesos\(\)](#) y [ocupar_memoria\(\)](#).

11.3.2.4. nombre

char Proceso::nombre[10]
 Nombre/ID único del proceso (ej: "P1")
 Definición en la línea 58 del archivo [sim_engine.h](#).
 Referenciado por [asignar_proceso\(\)](#) y [ocupar_memoria\(\)](#).

11.3.2.5. t_ejecucion

int Proceso::t_ejecucion
 Tiempo total de ejecución requerido
 Definición en la línea 61 del archivo [sim_engine.h](#).
 Referenciado por [avanzar_tiempo\(\)](#), [cargar_procesos\(\)](#) y [test_sim\(\)](#).

11.3.2.6. t_final

int Proceso::t_final
 Instante en que finalizó (-1 si no ha terminado)
 Definición en la línea 64 del archivo [sim_engine.h](#).

11.3.2.7. t_llegada

int Proceso::t_llegada
 Instante de llegada a la cola de procesos
 Definición en la línea 59 del archivo [sim_engine.h](#).
 Referenciado por [cargar_procesos\(\)](#).

11.3.2.8. t_restante

int Proceso::t_restante
 Ticks restantes para completar ejecución
 Definición en la línea 65 del archivo [sim_engine.h](#).
 Referenciado por [avanzar_tiempo\(\)](#) y [cargar_procesos\(\)](#).
 La documentación de esta estructura está generada del siguiente archivo:

- src/[sim_engine.h](#)

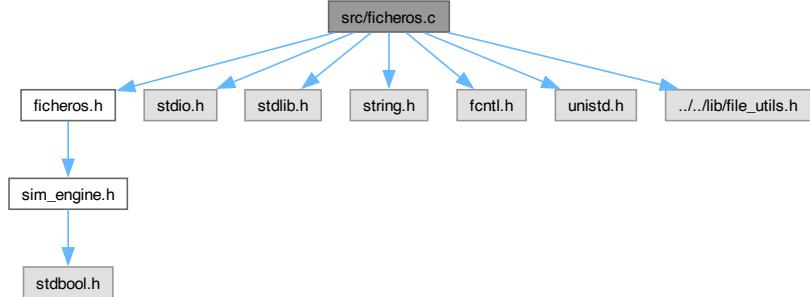
Capítulo 12

Documentación de archivos

- 12.1. Referencia del archivo docs/ARCHITECTURE.md
- 12.2. Referencia del archivo docs/CHANGELOG.md
- 12.3. Referencia del archivo docs/INSTALL.md
- 12.4. Referencia del archivo README.md
- 12.5. Referencia del archivo src/ficheros.c

```
#include "ficheros.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include "../lib/file_utils.h"
```

Gráfico de dependencias incluidas en ficheros.c:



defines

- `#define SIZE_BUFFER_LECTURA 4096`

Funciones

- `int cargar_procesos (const char *ruta, Proceso procesos[])`
Carga los procesos desde un archivo de texto.

- void [guardar_estado](#) (const char *ruta, [Memoria](#) *m, int instante)

Guarda el estado actual de la memoria en un archivo de log.
- void [limpiar_log](#) (const char *ruta)

Limpia el contenido de un archivo de log.

12.5.1. Documentación de «define»

12.5.1.1. SIZE_BUFFER_LECTURA

```
#define SIZE_BUFFER_LECTURA 4096
Definición en la línea 9 del archivo ficheros.c.
Referenciado por cargar\_procesos\(\).
```

12.5.2. Documentación de funciones

12.5.2.1. cargar_procesos()

```
int cargar_procesos (
    const char * ruta,
    Proceso procesos[])
```

Carga los procesos desde un archivo de texto.

Lee el archivo línea por línea y parsea cada proceso con el formato: <nombre> <t_llegada> <mem_requerida> <t_ejecucion>

Parámetros

in	ruta	Ruta del archivo de entrada
out	procesos	Array donde se almacenarán los procesos cargados

Devuelve

Cantidad de procesos cargados exitosamente

0 si hubo error al abrir/leer el archivo

Precondición

El archivo debe existir y ser legible

procesos debe tener espacio para MAX_PROCESOS elementos

Nota

Usa llamadas POSIX (open, read) según requerimientos de la práctica

Atención

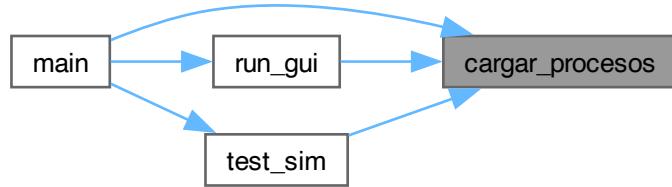
El buffer interno es de 4KB, archivos más grandes se truncan

Definición en la línea 11 del archivo [ficheros.c](#).

Hace referencia a [Proceso::en_memoria](#), [Proceso::finalizado](#), [MAX_PROCESOS](#), [Proceso::mem_requerida](#), [SIZE_BUFFER_LECTURA](#), [Proceso::t_ejecucion](#), [Proceso::t_llegada](#) y [Proceso::t_restante](#).

Referenciado por [main\(\)](#), [run_gui\(\)](#) y [test_sim\(\)](#).

Gráfico de llamadas a esta función:



12.5.2.2. guardar_estado()

```
void guardar_estado (
    const char * ruta,
    Memoria * m,
    int instante)
```

Guarda el estado actual de la memoria en un archivo de log.

Escribe una línea con formato: <instante>[dir1, nombre1, tam1] [dir2, nombre2, tam2] ...

Parámetros

in	ruta	Ruta del archivo de salida (se crea si no existe)
in	m	Puntero a la estructura de memoria a guardar
in	instante	Instante de tiempo actual de la simulación

Nota

Usa modo append, no sobrescribe contenido previo

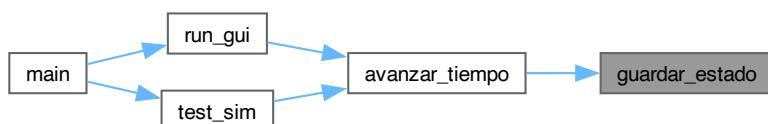
Permisos del archivo: 0644 (rw-r-r-)

Definición en la línea 64 del archivo [ficheros.c](#).

Hace referencia a [Memoria::cant_particiones](#), [Particion::dir_inicio](#), [Particion::nombre_proceso](#), [Memoria::particiones](#) y [Particion::tamano](#).

Referenciado por [avanzar_tiempo\(\)](#).

Gráfico de llamadas a esta función:



12.5.2.3. limpiar_log()

```
void limpiar_log (
    const char * ruta)
```

Limpia el contenido de un archivo de log.

Abre el archivo con O_TRUNC para borrar su contenido. Si no existe, lo crea vacío.

Parámetros

in	ruta	Ruta del archivo a limpiar
----	------	----------------------------

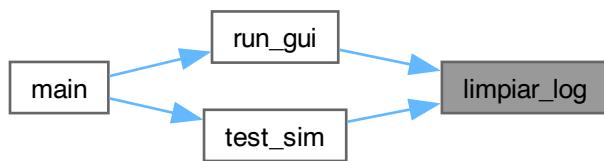
Nota

Llamar al inicio de cada simulación para empezar con log limpio

Definición en la línea 94 del archivo [ficheros.c](#).

Referenciado por [run_gui\(\)](#) y [test_sim\(\)](#).

Gráfico de llamadas a esta función:



12.6. ficheros.c

[Ir a la documentación de este archivo.](#)

```

00001 #include "ficheros.h"
00002 #include <stdio.h>
00003 #include <stdlib.h>
00004 #include <string.h>
00005 #include <fcntl.h>
00006 #include <unistd.h>
00007 #include "../lib/file_utils.h"
00008
00009 #define SIZE_BUFFER_LECTURA 4096 // Asumimos que el archivo de entrada no pesa más de 4KB
00010
00011 int cargar_procesos(const char* ruta, Proceso procesos[]) {
00012     // 1. Abrir con open(Unix standard)
00013     int fd = open(ruta, O_RDONLY);
00014     if (fd == -1) {
00015         perror("Error abriendo fichero de entrada");
00016         return 0;
00017     }
00018
00019     // 2. Leer TODO el archivo a un buffer usando file_utils.h
00020     char buffer_archivo[SIZE_BUFFER_LECTURA];
00021     memset(buffer_archivo, 0, SIZE_BUFFER_LECTURA); // Limpiamos el buffer para evitar basura
00022
00023     ssize_t leidos = read_all(fd, buffer_archivo, SIZE_BUFFER_LECTURA - 1);
00024     close(fd); // Ya tenemos los datos y por tanto cerramos el descriptor
00025     if (leidos == -1) {
00026         perror("Error al leer el fichero");
00027         return 0;
00028     }
00029
00030     // 3. Parsear el texto (linea por linea)
00031     int count = 0;
00032     char *linea = strtok(buffer_archivo, "\n"); // Primera linea
00033
  
```

```

00034     while(linea != NULL && count < MAX_PROCESOS) {
00035         char nombre[10];
00036         int llegada, mem, ejec;
00037         // Formato fuente: <Proceso> <Instante_llegada> <Memoria_requerida> <Tiempo_de_ejecucion>
00038         if (sscanf(linea, "%s %d %d %d", nombre, &llegada, &mem, &ejec) == 4) {
00039             // Copiamos al struct Proceso
00040             strcpy(procesos[count].nombre, nombre);
00041             procesos[count].t_llegada = llegada;
00042
00043             // Opcion 1: Aplicamos alinear_size para cumplir con la asignacion de 100 en 100
00044             // procesos[count].mem_requerida = alinear_size(mem);
00045             // Opcion 2: La memoria ya se aliena a la hora de asignar el proceso
00046             procesos[count].mem_requerida = mem;
00047
00048             procesos[count].t_ejecucion = ejec;
00049
00050             // Inicializacion del estado del proceso
00051             procesos[count].t_restante = ejec;
00052             procesos[count].en_memoria = false;
00053             procesos[count].finalizado = false;
00054
00055             count++;
00056         }
00057         // Siguiente linea
00058         linea = strtok(NULL, "\n");
00059     }
00060
00061     return count;
00062 }
00063
00064 void guardar_estado(const char* ruta, Memoria *m, int instante) {
00065     // Abrimos el archivo en modo append y CREAT si no existe
00066     int fd = open(ruta, O_WRONLY | O_CREAT | O_APPEND, 0644);
00067     if (fd == -1) {
00068         perror("Error al abrir o crear el archivo");
00069         return;
00070     }
00071
00072     char buffer_linea[1024]; // Buffer temporal
00073
00074     // Formato fuente: <instante_tiempo> <estado_memoria> \n
00075     // 1. Escribimos el instante de tiempo
00076     int len = sprintf(buffer_linea, "%d", instante);
00077     write_all(fd, buffer_linea, len);
00078
00079     // 2. Recorremos las particiones
00080     for (int i = 0; i < m->cant_particiones; i++) {
00081         Particion p = m->particiones[i];
00082
00083         // Formateamos una particion al formato requerido
00084         len = sprintf(buffer_linea, "[ %d, %s, %d ] ", p.dir_inicio, p.nombre_proceso, p.tamano);
00085         write_all(fd, buffer_linea, len);
00086     }
00087
00088     // 3. Salto de linea final
00089     write_all(fd, "\n", 1);
00090
00091     close(fd);
00092 }
00093
00094 void limpiar_log(const char* ruta) {
00095     // O_TRUNC borra el contenido al abrir
00096     int fd = open(ruta, O_WRONLY | O_CREAT | O_TRUNC, 0644);
00097     if (fd != -1) close(fd);
00098 }

```

12.7. Referencia del archivo src/ficheros.h

Módulo de entrada/salida para la simulación de memoria.

```
#include "sim_engine.h"
```

Gráfico de dependencias incluidas en ficheros.h:

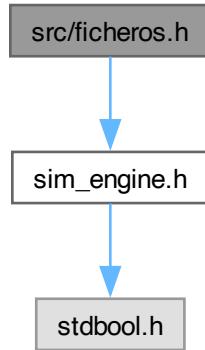
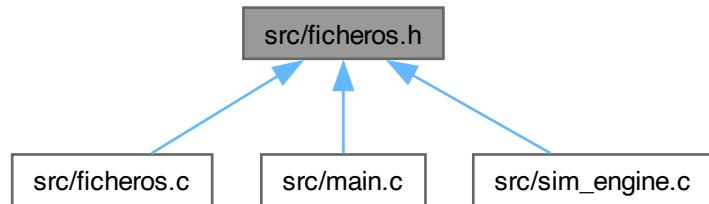


Gráfico de los archivos que directa o indirectamente incluyen a este archivo:



Funciones

- int [cargar_procesos](#) (const char *ruta, [Proceso](#) procesos[])
 - Carga los procesos desde un archivo de texto.
- void [guardar_estado](#) (const char *ruta, [Memoria](#) *m, int instante)
 - Guarda el estado actual de la memoria en un archivo de log.
- void [limpiar_log](#) (const char *ruta)
 - Limpia el contenido de un archivo de log.

12.7.1. Descripción detallada

Módulo de entrada/salida para la simulación de memoria.

Proporciona funciones para:

- Cargar procesos desde archivos de texto
- Guardar el estado de la memoria (logging)
- Gestión de archivos de log

Autor

Julian Hinojosa Gil

Fecha

2025

Versión

1.0

Definición en el archivo [ficheros.h](#).

12.7.2. Documentación de funciones

12.7.2.1. cargar_procesos()

```
int cargar_procesos (
    const char * ruta,
    Proceso procesos[])
```

Carga los procesos desde un archivo de texto.

Lee el archivo línea por línea y parsea cada proceso con el formato: <nombre> <t_llegada> <mem_requerida> <t_ejecucion>

Parámetros

in	ruta	Ruta del archivo de entrada
out	procesos	Array donde se almacenarán los procesos cargados

Devuelve

Cantidad de procesos cargados exitosamente

0 si hubo error al abrir/leer el archivo

Precondición

El archivo debe existir y ser legible

procesos debe tener espacio para MAX_PROCESOS elementos

Nota

Usa llamadas POSIX (open, read) según requerimientos de la práctica

Atención

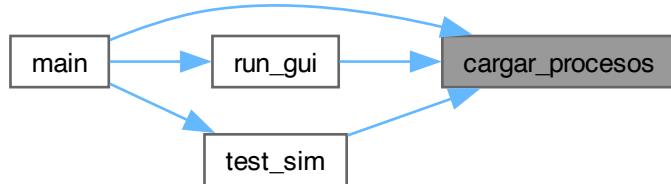
El buffer interno es de 4KB, archivos más grandes se truncan

Definición en la línea 11 del archivo [ficheros.c](#).

Hace referencia a [Proceso::en_memoria](#), [Proceso::finalizado](#), [MAX_PROCESOS](#), [Proceso::mem_requerida](#), [SIZE_BUFFER_LECTURA](#), [Proceso::t_ejecucion](#), [Proceso::t_llegada](#) y [Proceso::t_restante](#).

Referenciado por [main\(\)](#), [run_gui\(\)](#) y [test_sim\(\)](#).

Gráfico de llamadas a esta función:



12.7.2.2. guardar_estado()

```
void guardar_estado (
    const char * ruta,
    Memoria * m,
    int instante)
```

Guarda el estado actual de la memoria en un archivo de log.

Escribe una línea con formato: <instante>[dir1, nombre1, tam1] [dir2, nombre2, tam2] ...

Parámetros

in	ruta	Ruta del archivo de salida (se crea si no existe)
in	m	Puntero a la estructura de memoria a guardar
in	instante	Instante de tiempo actual de la simulación

Nota

Usa modo append, no sobrescribe contenido previo

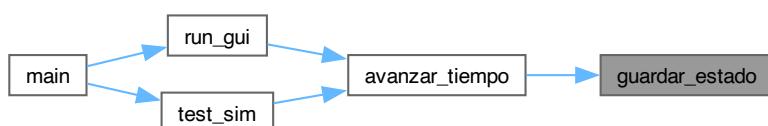
Permisos del archivo: 0644 (rw-r-r-)

Definición en la línea 64 del archivo [ficheros.c](#).

Hace referencia a [Memoria::cant_particiones](#), [Particion::dir_inicio](#), [Particion::nombre_proceso](#), [Memoria::particiones](#) y [Particion::tamano](#).

Referenciado por [avanzar_tiempo\(\)](#).

Gráfico de llamadas a esta función:



12.7.2.3. limpiar_log()

```
void limpiar_log (
    const char * ruta)
```

Limpia el contenido de un archivo de log.

Abre el archivo con O_TRUNC para borrar su contenido. Si no existe, lo crea vacío.

Parámetros

in	ruta	Ruta del archivo a limpiar
----	------	----------------------------

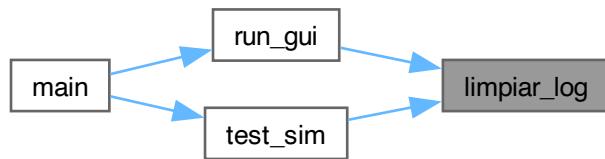
Nota

Llamar al inicio de cada simulación para empezar con log limpio

Definición en la línea 94 del archivo [ficheros.c](#).

Referenciado por [run_gui\(\)](#) y [test_sim\(\)](#).

Gráfico de llamadas a esta función:



12.8. ficheros.h

[Ir a la documentación de este archivo.](#)

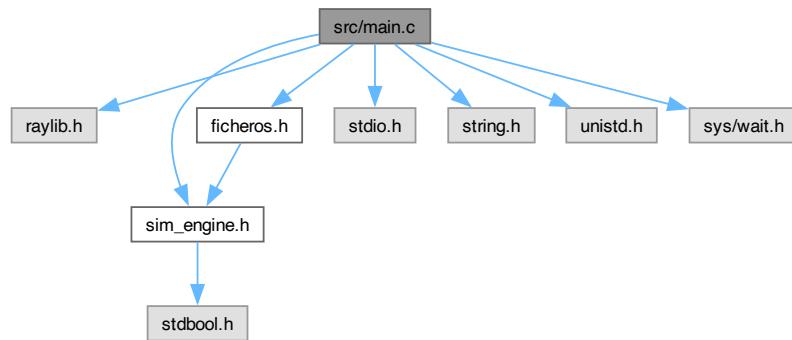
```
00001 #ifndef FICHEROS_H
00002 #define FICHEROS_H
00003
00004 #include "sim_engine.h"
00005
00019
00038 int cargar_procesos(const char* ruta, Proceso procesos[]);
00039
00053 void guardar_estado(const char* ruta, Memoria *m, int instante);
00054
00065 void limpiar_log(const char* ruta);
00066
00067 #endif // FICHEROS_H
```

12.9. Referencia del archivo src/main.c

Punto de entrada del simulador de gestión de memoria.

```
#include <raylib.h>
#include "sim_engine.h"
#include "ficheros.h"
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
```

Gráfico de dependencias incluidas en main.c:



defines

- `#define WIN_WIDTH 1200`
Ancho de la ventana gráfica en píxeles.
- `#define WIN_HEIGHT 750`
Alto de la ventana gráfica en píxeles.
- `#define Y_BARRA 250`
Posición Y de la barra de memoria.
- `#define ALTO_BARRA 100`
Alto de la barra de memoria.
- `#define MARGEN_IZQ 50`
Margen izquierdo de la interfaz.
- `#define MARGEN_DER 50`
Margen derecho de la interfaz.

Funciones

- `void test_sim ()`
Ejecuta la simulación en modo terminal (TUI).
- `void run_gui (Memoria *m, Proceso *procesos, int num_procesos)`
Ejecuta la interfaz gráfica con Raylib.
- `int main (int argc, char const *argv[])`
Punto de entrada principal del programa.

12.9.1. Descripción detallada

Punto de entrada del simulador de gestión de memoria.

Implementa dos interfaces de usuario:

- GUI: Interfaz gráfica con Raylib (proceso hijo)
- TUI: Interfaz de terminal para depuración (proceso padre)

Usa fork() para ejecutar ambas interfaces en paralelo.

Autor

Julian Hinojosa Gil

Fecha

2025

Versión

1.0

Definición en el archivo [main.c](#).

12.9.2. Documentación de «define»

12.9.2.1. ALTO_BARRA

```
#define ALTO_BARRA 100
```

Alto de la barra de memoria.

Definición en la línea [33](#) del archivo [main.c](#).

Referenciado por [run_gui\(\)](#).

12.9.2.2. MARGEN_DER

```
#define MARGEN_DER 50
```

Margen derecho de la interfaz.

Definición en la línea [37](#) del archivo [main.c](#).

Referenciado por [run_gui\(\)](#).

12.9.2.3. MARGEN_IZQ

```
#define MARGEN_IZQ 50
```

Margen izquierdo de la interfaz.

Definición en la línea [35](#) del archivo [main.c](#).

Referenciado por [run_gui\(\)](#).

12.9.2.4. WIN_HEIGHT

```
#define WIN_HEIGHT 750
```

Alto de la ventana gráfica en píxeles.

Definición en la línea [29](#) del archivo [main.c](#).

Referenciado por [run_gui\(\)](#).

12.9.2.5. WIN_WIDTH

```
#define WIN_WIDTH 1200
```

Ancho de la ventana gráfica en píxeles.

Definición en la línea [27](#) del archivo [main.c](#).

Referenciado por [run_gui\(\)](#).

12.9.2.6. Y_BARRA

```
#define Y_BARRA 250
```

Posición Y de la barra de memoria.

Definición en la línea [31](#) del archivo [main.c](#).

Referenciado por [run_gui\(\)](#).

12.9.3. Documentación de funciones

12.9.3.1. main()

```
int main (
```

```
    int argc,  
    char const * argv[])
```

Punto de entrada principal del programa.

Crea dos procesos mediante fork():

- **Proceso** hijo ($pid == 0$): Ejecuta la interfaz gráfica (GUI)
- **Proceso** padre ($pid > 0$): Ejecuta la interfaz de terminal (TUI) y espera al hijo

Parámetros

argc	Número de argumentos (no utilizado)
argv	Array de argumentos (no utilizado)

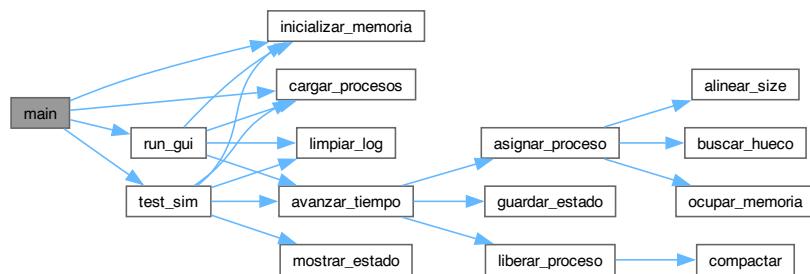
Devuelve

0 si la ejecución fue exitosa

-1 si hubo error en fork()

Definición en la línea 56 del archivo [main.c](#).

Hace referencia a [cargar_procesos\(\)](#), [inicializar_memoria\(\)](#), [MAX_PROCESOS](#), [run_gui\(\)](#) y [test_sim\(\)](#). Gráfico de llamadas de esta función:



12.9.3.2. run_gui()

```
void run_gui (
    Memoria * m,
    Proceso * procesos,
    int num_procesos)
```

Ejecuta la interfaz gráfica con Raylib.

Crea una ventana de [WIN_WIDTH](#) x [WIN_HEIGHT](#) píxeles y muestra:

- Barra de memoria con particiones coloreadas
- Lista de procesos y su estado
- Controles interactivos

Parámetros

in,out	m	Puntero a la estructura de memoria del simulador
in,out	procesos	Array de procesos a simular
in	num_procesos	Cantidad de procesos en el array

Controles:

- ESPACIO: Avanzar un tick
- P: Activar/desactivar auto-play
- R: Reiniciar simulación
- ESC: Salir

Nota

Genera log en "particiones.txt"

Velocidad auto-play: 1 tick/segundo

Definición en la línea 173 del archivo [main.c](#).

Hace referencia a [ALGO_PRIMER_HUECO](#), [ALTO_BARRA](#), [avanzar_tiempo\(\)](#), [Memoria::cant_particiones](#), [cargar_procesos\(\)](#), [Particion::dir_inicio](#), [Particion::estado](#), [inicializar_memoria\(\)](#), [limpiar_log\(\)](#), [MARGEN_DER](#), [MARGEN_IZQ](#), [MEMORIA_TOTAL](#), [Particion::nombre_proceso](#), [Memoria::particiones](#), [Particion::tamano](#), [WIN_HEIGHT](#), [WIN_WIDTH](#) y [Y_BARRA](#).

Referenciado por [main\(\)](#).

Gráfico de llamadas de esta función:

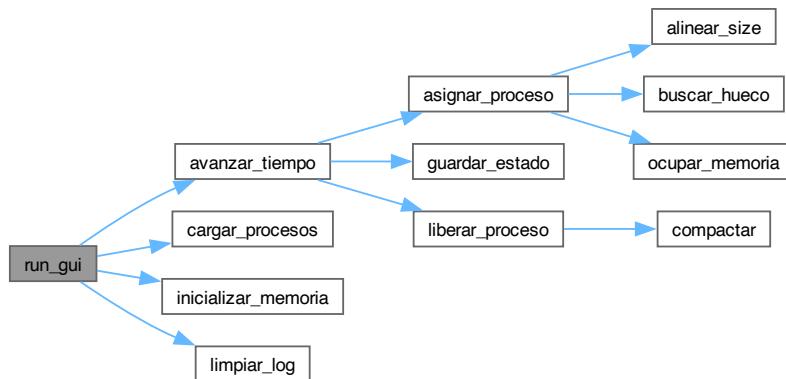


Gráfico de llamadas a esta función:



12.9.3.3. test_sim()

```
void test_sim ()
```

Ejecuta la simulación en modo terminal (TUI).

Carga procesos desde "entrada.txt" y ejecuta la simulación paso a paso, mostrando el estado en consola.

Nota

Usa getchar() para avanzar manualmente entre ticks

Genera log en "particiones_tui.txt"

Usa algoritmo First Fit por defecto

Definición en la línea 95 del archivo [main.c](#).

Hace referencia a [ALGO_PRIMER_HUECO](#), [avanzar_tiempo\(\)](#), [cargar_procesos\(\)](#), [inicializar_memoria\(\)](#), [limpiar_log\(\)](#), [MAX_PROCESOS](#), [MEMORIA_TOTAL](#), [mostrar_estado\(\)](#) y [Proceso::t_ejecucion](#).

Referenciado por [main\(\)](#).

Gráfico de llamadas de esta función:

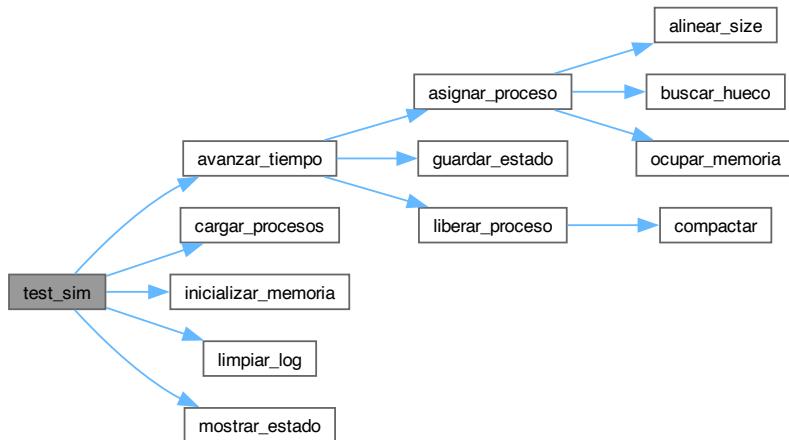


Gráfico de llamadas a esta función:



12.10. main.c

[Ir a la documentación de este archivo.](#)

```

00001
00015
00016 #include <raylib.h>
00017
00018 #include "sim_engine.h"
00019 #include "ficheros.h"
00020
00021 #include <stdio.h>
00022 #include <string.h>
00023 #include <unistd.h>
00024 #include <sys/wait.h>
00025
00027 #define WIN_WIDTH 1200
00029 #define WIN_HEIGHT 750
00031 #define Y_BARRA 250
00033 #define ALTO_BARRA 100
  
```

```

00035 #define MARGEN_IZQ 50
00037 #define MARGEN_DER 50
00038
00039 /* Declaraciones adelantadas */
00040 void test_sim();
00041 void run_gui(Memoria *m, Proceso *procesos, int num_procesos);
00042
00056 int main(int argc, char const* argv[])
00057 {
00058     pid_t pid = fork();
00059
00060     if (pid < 0) {
00061         perror("Error en fork");
00062         return -1;
00063     }
00064     else if (pid == 0) {
00065         // Proceso Hijo: GUI (Raylib)
00066         Memoria m;
00067         inicializar_memoria(&m);
00068
00069         Proceso procesos[MAX_PROCESOS];
00070         int num_procesos = cargar_procesos("entrada.txt", procesos);
00071
00072         run_gui(&m, procesos, num_procesos);
00073         _exit(0); // El hijo termina después de cerrar la GUI
00074     } else {
00075         // Proceso Padre: TUI (Terminal)
00076         test_sim();
00077
00078         // Esperar a que el hijo (GUI) termine
00079         waitpid(pid, NULL, 0);
00080     }
00081
00082     return 0;
00083 }
00084
00095 void test_sim() {
00096     Memoria m;
00097     inicializar_memoria(&m);
00098
00099     TipoAlgo algoritmo_actual = ALGO_PRIMER_HUECO;
00100
00101     limpiar_log("particiones_tui.txt");
00102
00103
00104     // // Simulamos procesos falsos para probar
00105     // Proceso procesos[2] = {
00106     //     {"P1", 0, 243, 5, false, false}, // ID, Llegada, Tam, Duracion, Restante...
00107     //     {"P2", 1, 45, 2, 2, false, false}
00108     //     // {"P3", 2, 500, 5, 5, false, false},
00109     //     // {"P4", 3, 800, 3, 3, false, false}
00110     // };
00111
00112     // Ahora con argumento de fichero
00113     Proceso procesos[MAX_PROCESOS]; // Maximo 100 procesos
00114     int num_procesos = cargar_procesos("entrada.txt", procesos);
00115
00116     int reloj_sim = 0;
00117
00118     int tiempo_total; // Simulamos 20 instantes
00119
00120     for (int i = 0; i < num_procesos; i++) {
00121         tiempo_total += procesos[i].t_ejecucion;
00122     }
00123
00124     // --- BUCLE DE SIMULACIÓN (10 Instantes) ---
00125     printf("==== INICIO DE LA SIMULACIÓN COMPLEJA ====\n");
00126     printf("Memoria Total: %d\n", MEMORIA_TOTAL);
00127
00128     for (int i = 0; i < tiempo_total; i++) {
00129         // Llamamos al motor de tiempo
00130         avanzar_tiempo(&m, procesos, num_procesos, &reloj_sim, algoritmo_actual, "particiones_tui.txt");
00131
00132         // Visualizamos la memoria
00133         printf("MEMORIA: ");
00134         mostrar_estado(&m); // Mostramos el estado de la memoria con el formato adecuado [0 P1 500]...
00135
00136         // Visualizamos quién está esperando (Cola de espera)
00137         printf("COLA ESPERA: ");
00138         int esperando = 0;
00139         for(int j=0; j<4; j++) {
00140             if (!procesos[j].en_memoria && !procesos[j].finalizado && procesos[j].t_llegada <= reloj_sim-1) {
00141                 printf("[%s req: %d]", procesos[j].nombre, procesos[j].mem_requerida);
00142                 esperando++;
00143             }
00144         }
00145         if(esperando == 0) printf("(Nadie)");

```

```

00146     printf("\n-----\n");
00147
00148     getchar(); // Descomenta para ir paso a paso con ENTER
00149 }
00150 }
00151
00173 void run_gui(Memoria *m, Proceso *procesos, int num_procesos) {
00174     InitWindow(WIN_WIDTH, WIN_HEIGHT, "Gestmemoria - Memory Simulator");
00175     SetTargetFPS(60);           // Set our game to run at 60 frames-per-second
00176
00177     limpiar_log("particiones.txt");
00178
00179     int reloj_sim = 0;
00180     TipoAlgo algoritmo_actual = ALGO_PRIMER_HUECO;
00181
00182     // Variables para repro automatica
00183     bool auto_play = false;
00184     float tiempo_acumulado = 0.0f;
00185     float velocidad = 1.0f; // 1 segundo por tick
00186
00187     // Variables para los colores de los procesos
00188     Color colores_prcesos[] = {
00189         SKYBLUE,
00190         PINK,
00191         ORANGE,
00192         LIME,
00193         GOLD,
00194         GREEN,
00195         VIOLET,
00196         MAROON,
00197         BLUE,
00198         RED,
00199         PURPLE
00200     };
00201     int num_colores = sizeof(colores_prcesos) / sizeof(colores_prcesos[0]);
00202
00203     // Main app loop
00204     while (!WindowShouldClose()) // Detect window close button or ESC key
00205     {
00206
00207         // Opcion A: Paso Manual(Tecla espacio)
00208         if (IsKeyPressed(KEY_SPACE))
00209             avanzar_tiempo(m, procesos, num_procesos, &reloj_sim, algoritmo_actual, "particiones.txt");
00210
00211         // Opcion B: Auto Play(Tecla P)
00212         if (IsKeyPressed(KEY_P))
00213             auto_play = !auto_play;
00214
00215         // Logica del auto play
00216         if (auto_play)
00217             tiempo_acumulado += GetFrameTime();
00218             if (tiempo_acumulado >= velocidad) {
00219                 avanzar_tiempo(m, procesos, num_procesos, &reloj_sim, algoritmo_actual, "particiones.txt");
00220                 tiempo_acumulado = 0.0f;
00221             }
00222
00223
00224         // Opcion C: Resetear (Tecla R)
00225         if (IsKeyPressed(KEY_R)) {
00226             inicializar_memoria(m);
00227             num_procesos = cargar_procesos("entrada.txt", procesos);
00228             reloj_sim = 0;
00229             limpiar_log("particiones.txt");
00230             auto_play = false;
00231         }
00232
00233         // Draw
00234         //-----
00235     BeginDrawing();
00236
00237     ClearBackground(RAYWHITE);
00238
00239         // 1. Textos de informacion
00240     DrawText("Simulador de Gestión de Memoria", 20, 20, 30, DARKGRAY);
00241     DrawText(TextFormat("Instante Actual: %d", reloj_sim), 20, 60, 20, BLACK);
00242
00243     DrawText("Pulsa ESPACIO para avanzar, P para auto play, R para resetear", 20, 90, 20, DARKGRAY);
00244     DrawText("Pulsa ESC para salir", 20, 120, 20, DARKGRAY);
00245
00246         if (auto_play) DrawText("Auto Play: ON (P para Desactivar)", 20, 150, 20, DARKGREEN);
00247         else DrawText("Auto Play: OFF (P para Activar)", 20, 150, 20, RED);
00248
00249     DrawText(TextFormat("Procesos Cargados: %d", num_procesos), 20, 180, 15, DARKGRAY);
00250
00251         // 2. Dibujar la barra de memoria
00252         // Escala: Pantalla / Memoria -> WIN_WIDTH / MEMORIA_TOTAL
00253     float ancho_util = WIN_WIDTH - MARGEN_IZQ - MARGEN_DER;

```

```

00254     float escala = ancho_util / MEMORIA_TOTAL;
00255
00256     for (int i = 0; i < m->cant_particiones; i++) {
00257         Particion p = m->particiones[i];
00258
00259         // Coordenadas
00260         float x = MARGEN_IZQ + (p.dir_inicio * escala);
00261         float ancho = p.tamano * escala;
00262
00263         // Colores de los bloques
00264         Color colorBloque;
00265         Color colorHueco = (Color){227, 217, 132, 255};
00266
00267         // Determinar color segun estado
00268         // Si es hueco, colorHueco, si es proceso, color segun indice de color_procesos
00269         if (p.estado == 0) {
00270             colorBloque = colorHueco;
00271         } else {
00272             // Buscar indice del proceso por nombre para asignar color
00273             int indice_color = 0;
00274             for (int j = 0; j < num_procesos; j++) {
00275                 if (strcmp(procesos[j].nombre, p.nombre_proceso) == 0) {
00276                     // Si hay mas procesos que colores, reutilizamos por ejemplo, con 6 colores en 15 procesos, el 15 usará
00277                     el color 3
00278                     indice_color = j % num_colores;
00279                     break;
00280                 }
00281             }
00282             colorBloque = colores_prcesos[indice_color];
00283         }
00284
00285         // Dibujar el rectangulo
00286         DrawRectangle(x, Y_BARRA, ancho, ALTO_BARRA, colorBloque);
00287         DrawRectangleLines(x, Y_BARRA, ancho, ALTO_BARRA, Fade(BLACK, 0.5f));
00288
00289         // Informacion del bloque (si cabe)
00290         if (ancho > 40) {
00291             DrawText(p.nombre_proceso, x + 5, Y_BARRA + 35, 15, BLACK);
00292             DrawText(TextFormat("%d", p.tamano), x + 5, Y_BARRA + 50, 15, DARKGRAY);
00293         }
00294
00295         // Direccion de memoria (arriba del bloque)
00296         DrawText(TextFormat("%d", p.dir_inicio), x, Y_BARRA - 15, 15, BLACK);
00297
00298         // Marca final (2000)
00299         DrawText("2000", MARGEN_IZQ + ancho_util - 30, Y_BARRA - 15, 15, BLACK);
00300
00301         // Tablas de procesos
00302         // -----
00303         // 3. DIBUJAR TABLA DE PROCESOS (Debajo de la barra de memoria)
00304
00305         int y_inicio_tabla = Y_BARRA + ALTO_BARRA + 40; // Empezamos 40px debajo de la barra
00306         int x_columna = 20;
00307         int ancho_columna = 120; // Espacio entre columnas
00308
00309         DrawText("ESTADO DE PROCESOS:", x_columna, y_inicio_tabla - 25, 20, DARKGRAY);
00310
00311         // Cabecera simple
00312         // DrawText("ID  Mem  Vida  Estado", x_columna, y_inicio_tabla, 10, BLACK);
00313
00314         for (int i = 0; i < num_procesos; i++) {
00315             // Solo dibujamos si cabe en pantalla, si hay muchos habría que hacer paginación
00316             if (y_inicio_tabla + (i * 20) > WIN_HEIGHT - 20) break;
00317
00318             // Determinar Color y Estado Texto
00319             Color colorEstado = LIGHTGRAY;
00320             const char* textoEstado = "FUTURO";
00321
00322             if (procesos[i].finalizado) {
00323                 colorEstado = GRAY; // Gris oscuro para acabados
00324                 textoEstado = "FIN";
00325             }
00326             else if (procesos[i].en_memoria) {
00327                 colorEstado = GREEN; // Verde para activos
00328                 textoEstado = "ACTIVO";
00329             }
00330             else if (procesos[i].t_llegada <= reloj_sim) {
00331                 colorEstado = RED; // Rojo para cola de espera
00332                 textoEstado = "COLA DE ESPERA";
00333             }
00334             else {
00335                 colorEstado = LIGHTGRAY; // Gris claro para los que aun no llegan
00336                 textoEstado = TextFormat("T=%d", procesos[i].t_llegada);
00337             }
00338
00339             // Dibujamos la fila del proceso

```

```

00340     // Calculamos posición (hacemos varias columnas si hay muchos, opcional)
00341     int fila_y = y_inicio_tabla + (i * 20);
00342
00343     // 1. ID y Tamaño
00344     DrawText(TextFormat("%s (%d)", procesos[i].nombre, procesos[i].mem_requerida),
00345             x_columna, fila_y, 20, BLACK);
00346
00347     // 2. Barra de vida (Pequeña barrita visual)
00348     if (procesos[i].en_memoria) {
00349         // Vida restante vs Total
00350         float porcentaje = (float)procesos[i].t_restante / procesos[i].t_ejecucion;
00351         DrawRectangle(x_columna + 100, fila_y + 5, 50 * porcentaje, 10, BLUE);
00352         DrawRectangleLines(x_columna + 100, fila_y + 5, 50, 10, BLACK);
00353     }
00354
00355     // 3. Estado (Texto coloreado)
00356     DrawText(textoEstado, x_columna + 160, fila_y, 20, colorEstado);
00357 }
00358
00359     EndDrawing();
00360 //-----
00361 }
00362
00363 // De-Initialization
00364 //-----
00365 CloseWindow(); // Close window and OpenGL context
00366 //-----
00367 }

```

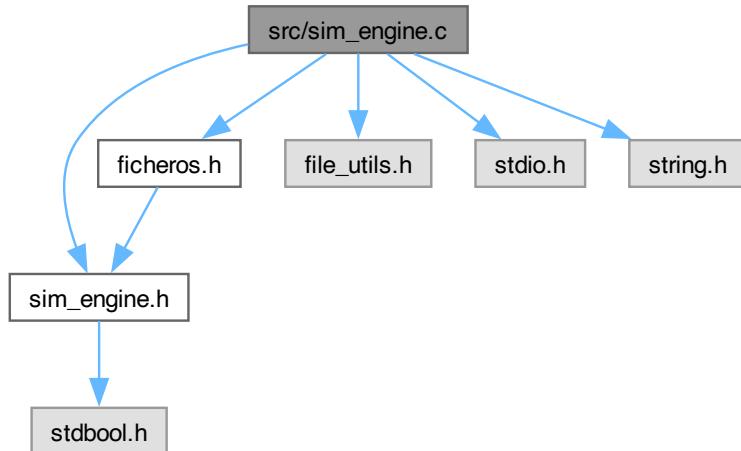
12.11. Referencia del archivo src/sim_engine.c

```

#include "sim_engine.h"
#include "ficheros.h"
#include "file_utils.h"
#include <stdio.h>
#include <string.h>

```

Gráfico de dependencias incluidas en sim_engine.c:



Funciones

- void **inicializar_memoria** (**Memoria** *m)
Inicializa la memoria con un único hueco libre.
- void **mostrar_estado** (**Memoria** *m)
Muestra el estado actual de la memoria en consola.

- int **ocupar_memoria** (**Memoria** *m, int indice_hueco, **Proceso** p)
Ocupa un hueco de memoria con un proceso.
- void **compactar** (**Memoria** *m)
Compacta la memoria uniendo huecos adyacentes.
- bool **liberar_proceso** (**Memoria** *m, char *nombre_proceso)
Libera un proceso de la memoria.
- int **buscar_hueco** (**Memoria** *m, int mem_requerida, **TipoAlgo** tipo_algo)
Busca un hueco adecuado según el algoritmo especificado.
- int **alinear_size** (int size)
Alinea un tamaño a múltiplos de UNIDAD_MINIMA.
- bool **asignar_proceso** (**Memoria** *m, **Proceso** p, **TipoAlgo** tipo_algo)
Asigna un proceso a la memoria.
- void **avanzar_tiempo** (**Memoria** *m, **Proceso** procesos[], int num_procesos, int *reloj_actual, **TipoAlgo** algo, const char *ruta_log)
Avanza un tick en la simulación.

12.11.1. Documentación de funciones

12.11.1.1. alinear_size()

```
int alinear_size (
    int size)
```

Alinea un tamaño a múltiplos de UNIDAD_MINIMA.

Redondea hacia arriba para garantizar que la memoria asignada sea siempre múltiplo de 100.

Parámetros

in	size	Tamaño solicitado original
----	------	----------------------------

Devuelve

Tamaño alineado (múltiplo de UNIDAD_MINIMA)

Ejemplos:

- alinear_size(50) → 100
- alinear_size(100) → 100
- alinear_size(243) → 300

Ejemplos

[/Users/julianhinojosagil/Documents/Dev/noob-code/UA/year-2/OS/practica3/src/sim_engine.h](#).

Definición en la línea 151 del archivo **sim_engine.c**.

Hace referencia a **UNIDAD_MINIMA**.

Referenciado por **asignar_proceso()**.

Gráfico de llamadas a esta función:



12.11.1.2. asignar_proceso()

```
bool asignar_proceso (
    Memoria * m,
    Proceso p,
    TipoAlgo tipo_algo)
```

Asigna un proceso a la memoria.

Realiza el flujo completo de asignación:

1. Alinea la memoria requerida
2. Busca un hueco según el algoritmo
3. Ocupa el hueco encontrado

Parámetros

in,out	m	Puntero a la estructura de memoria
in	p	Proceso a asignar
in	tipo_algo	Algoritmo de búsqueda a usar

Devuelve

true si se asignó correctamente
false si no hay espacio suficiente

Ver también

[alinear_size\(\)](#), [buscar_hueco\(\)](#), [ocupar_memoria\(\)](#)

Ejemplos

[/Users/julianhinojosagil/Documents/Dev/noob-code/UA/year-2/OS/practica3/src/sim_engine.h](#).

Definición en la línea 167 del archivo [sim_engine.c](#).

Hace referencia a [alinear_size\(\)](#), [buscar_hueco\(\)](#), [Proceso::mem_requerida](#), [Proceso::nombre](#) y [ocupar_memoria\(\)](#).

Referenciado por [avanzar_tiempo\(\)](#).

Gráfico de llamadas de esta función:

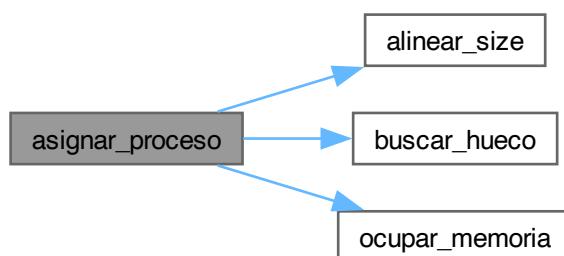
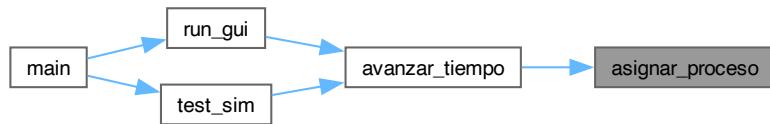


Gráfico de llamadas a esta función:



12.11.1.3. avanzar_tiempo()

```
void avanzar_tiempo (
    Memoria * m,
    Proceso procesos[],
    int num_procesos,
    int * reloj_actual,
    TipoAlgo algo,
    const char * ruta_log)
```

Avanza un tick en la simulación.

Ejecuta la lógica de un instante de tiempo:

1. Envejece procesos en memoria (decrementa t_restante)
2. Finaliza procesos cuyo tiempo restante llegó a 0
3. Intenta cargar nuevos procesos de la cola
4. Guarda el estado en el log
5. Incrementa el reloj

Parámetros

in,out	m	Puntero a la estructura de memoria
in,out	procesos	Array de procesos a gestionar
in	num_procesos	Número de procesos en el array
in,out	reloj_actual	Puntero al reloj de simulación
in	algo	Algoritmo de asignación a utilizar
in	ruta_log	Ruta del archivo de log para guardar estado

Postcondición

(*reloj_actual) se incrementa en 1

Ejemplos

[/Users/julianhinojosagil/Documents/Dev/noob-code/UA/year-2/OS/practica3/src/sim_engine.h](file:///Users/julianhinojosagil/Documents/Dev/noob-code/UA/year-2/OS/practica3/src/sim_engine.h).

Definición en la línea 189 del archivo `sim_engine.c`.

Hace referencia a `asignar_proceso()`, `Proceso::en_memoria`, `Proceso::finalizado`, `guardar_estado()`, `liberar_proceso()`, `Proceso::t_ejecucion` y `Proceso::t_restante`.

Referenciado por `run_gui()` y `test_sim()`.

Gráfico de llamadas de esta función:

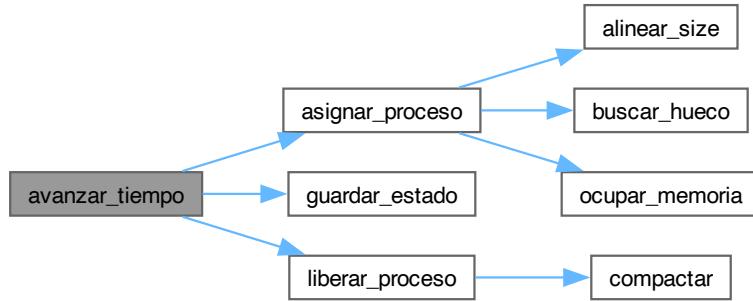
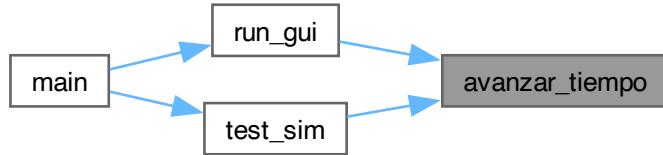


Gráfico de llamadas a esta función:



12.11.1.4. buscar_hueco()

```
int buscar_hueco (
    Memoria * m,
    int mem_requerida,
    TipoAlgo tipo_algo)
```

Busca un hueco adecuado según el algoritmo especificado.

Implementa los algoritmos:

- First Fit: Busca desde el inicio ($O(n)$ peor caso)
- Next Fit: Busca desde última posición (búsqueda circular)

Parámetros

in	m	Puntero a la estructura de memoria
in	mem_↔ requerida	Memoria requerida (ya alineada)
in	tipo_algo	Algoritmo a utilizar

Devuelve

- Índice del hueco encontrado (0 a cant_particiones-1)
- 1 si no hay hueco suficiente

Ejemplos

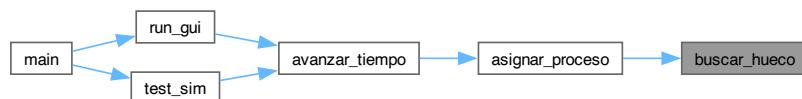
[/Users/julianhinojosagil/Documents/Dev/noob-code/UA/year-2/OS/practica3/src/sim_engine.h](#).

Definición en la línea 118 del archivo `sim_engine.c`.

Hace referencia a `ALGO_PRIMER_HUECO`, `ALGO_SIGUIENTE_HUECO`, `Memoria::cant_particiones`, `Particion::estado`, `Memoria::particiones`, `Particion::tamano` y `Memoria::ultimo_indice_asignado`.

Referenciado por `asignar_proceso()`.

Gráfico de llamadas a esta función:



12.11.1.5. compactar()

```
void compactar (
    Memoria * m)
```

Compacta la memoria uniendo huecos adyacentes.

Recorre las particiones y fusiona huecos consecutivos en uno solo. Se llama automáticamente después de liberar un proceso.

Parámetros

in,out	m	Puntero a la estructura de memoria
--------	---	------------------------------------

Postcondición

- No hay dos huecos consecutivos en el array
- `cant_particiones` puede disminuir

Ejemplos

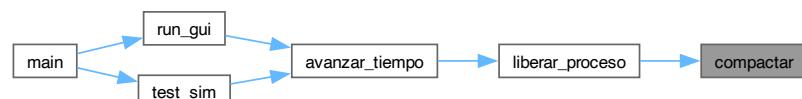
[/Users/julianhinojosagil/Documents/Dev/noob-code/UA/year-2/OS/practica3/src/sim_engine.h](#).

Definición en la línea 79 del archivo `sim_engine.c`.

Hace referencia a `Memoria::cant_particiones`, `Particion::estado`, `Particion::nombre_proceso`, `Memoria::particiones` y `Particion::tamano`.

Referenciado por `liberar_proceso()`.

Gráfico de llamadas a esta función:



12.11.1.6. inicializar_memoria()

```
void inicializar_memoria (
    Memoria * m)
```

Inicializa la memoria con un único hueco libre.

Configura la memoria con una sola partición de tipo hueco que ocupa todo el espacio disponible (MEMORIA_TOTAL).

Parámetros

out	m	Puntero a la estructura de memoria a inicializar
-----	---	--

Precondición

`m != NULL`

Postcondición

```
m->cant_particiones == 1
m->particiones[0].tamano == MEMORIA_TOTAL
```

Ejemplos

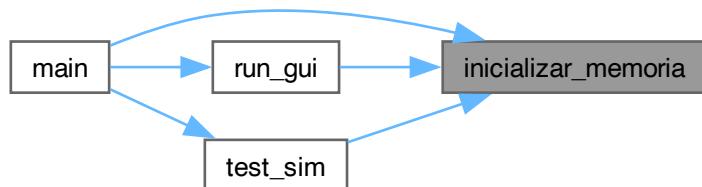
[/Users/julianhinojosagil/Documents/Dev/noob-code/UA/year-2/OS/practica3/src/sim_engine.h](#).

Definición en la línea 7 del archivo `sim_engine.c`.

Hace referencia a `Memoria::cant_particiones`, `Particion::dir_inicio`, `Particion::estado`, `MEMORIA_TOTAL`, `Particion::nombre_proceso`, `Memoria::particiones`, `Particion::tamano` y `Memoria::ultimo_indice_asignado`.

Referenciado por `main()`, `run_gui()` y `test_sim()`.

Gráfico de llamadas a esta función:



12.11.1.7. liberar_proceso()

```
bool liberar_proceso (
    Memoria * m,
    char * nombre_proceso)
```

Libera un proceso de la memoria.

Busca el proceso por nombre, lo convierte en hueco y compacta.

Parámetros

in,out	m	Puntero a la estructura de memoria
--------	---	------------------------------------

in	nombre_proceso	Nombre del proceso a liberar
----	----------------	------------------------------

Devuelve

true si se encontró y liberó el proceso
false si el proceso no estaba en memoria

Nota

Llama automáticamente a [compactar\(\)](#) tras liberar

Ejemplos

[/Users/julianhinojosagil/Documents/Dev/noob-code/UA/year-2/OS/practica3/src/sim_engine.h](#).

Definición en la línea 100 del archivo [sim_engine.c](#).

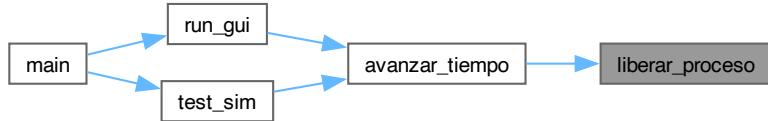
Hace referencia a [Memoria::cant_particiones](#), [compactar\(\)](#), [Particion::dir_inicio](#), [Particion::estado](#), [Particion::nombre_proceso](#) y [Memoria::particiones](#).

Referenciado por [avanzar_tiempo\(\)](#).

Gráfico de llamadas de esta función:



Gráfico de llamadas a esta función:



12.11.1.8. mostrar_estado()

void mostrar_estado (

[Memoria * m](#))

Muestra el estado actual de la memoria en consola.

Imprime cada partición en formato: [dir_inicio nombre tamaño]

Parámetros

in	m	Puntero a la estructura de memoria
----	---	------------------------------------

Nota

Solo para depuración/TUI, no afecta el estado

Ejemplos

[/Users/julianhinojosagil/Documents/Dev/noob-code/UA/year-2/OS/practica3/src/sim_engine.h](#)

Definición en la línea 23 del archivo `sim_engine.c`.

Hace referencia a `Memoria::cant_particiones`, `Particion::dir_inicio`, `Particion::nombre_proceso`, `Memoria::particiones` y `Particion::tamano`.

Referenciado por `test_sim()`.

Gráfico de llamadas a esta función:



12.11.1.9. ocupar_memoria()

```
int ocupar_memoria (
    Memoria * m,
    int indice_hueco,
    Proceso p)
```

Ocupa un hueco de memoria con un proceso.

Maneja dos casos:

- Ajuste exacto: El proceso ocupa todo el hueco
- División: Se crea una nueva partición con el espacio sobrante

Parámetros

in,out	m	Puntero a la estructura de memoria
in	indice_hueco	Índice del hueco en el array de particiones
in	p	Proceso a asignar (se usa nombre y mem_requerida)

Devuelve

- 1 si se asignó correctamente
- 0 si hubo error (hueco ocupado, tamaño insuficiente o array lleno)

Precondición

```
0 <= indice_hueco < m->cant_particiones
m->particiones[indice_hueco].estado == 0 (es un hueco)
```

Atención

No verifica si el proceso ya existe en memoria

Ejemplos

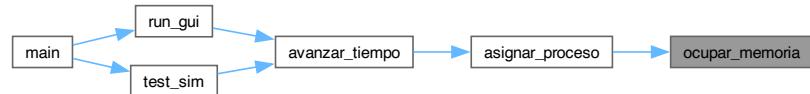
[/Users/julianhinojosagil/Documents/Dev/noob-code/UA/year-2/OS/practica3/src/sim_engine.h](file:///Users/julianhinojosagil/Documents/Dev/noob-code/UA/year-2/OS/practica3/src/sim_engine.h).

Definición en la línea 31 del archivo `sim_engine.c`.

Hace referencia a `Memoria::cant_particiones`, `Particion::dir_inicio`, `Particion::estado`, `MAX_PARTICIONES`, `Proceso::mem_requerida`, `Proceso::nombre`, `Particion::nombre_proceso`, `Memoria::particiones`, `Particion::tamano` y `Memoria::ultimo_indice_asignado`.

Referenciado por `asignar_proceso()`.

Gráfico de llamadas a esta función:



12.12. sim_engine.c

Ir a la documentación de este archivo.

```

00001 #include "sim_engine.h"
00002 #include "ficheros.h"
00003 #include "file_utils.h"
00004 #include <stdio.h>
00005 #include <string.h>
00006
00007 void inicializar_memoria(Memoria *m) {
00008     // Inicializamos la memoria con un único hueco que ocupa toda la memoria disponible
00009     m->cant_particiones = 1;
0010
0011     // El último índice asignado empieza en 0
0012     m->ultimo_indice_asignado = 0;
0013
0014     // Única partición: Hueco de tamaño MEMORIA_TOTAL al inicio
0015     m->particiones[0].dir_inicio = 0;
0016     m->particiones[0].tamano = MEMORIA_TOTAL;
0017     // Estado 0 indica que es un hueco libre
0018     m->particiones[0].estado = 0;
0019     // Nombre del proceso "HUECO" para indicar que está libre
0020     strcpy(m->particiones[0].nombre_proceso, "HUECO");
0021 }
0022
0023 void mostrar_estado(Memoria *m) {
0024     // Recorremos todas las particiones y mostramos su estado en formato [dir_inicio nombre_proceso tamano]
0025     for (int i = 0; i < m->cant_particiones; i++) {
0026         printf("[%d %s %d]", m->particiones[i].dir_inicio, m->particiones[i].nombre_proceso, m->particiones[i].tamano);
0027     }
0028     printf("\n");
0029 }
0030
0031 int ocupar_memoria(Memoria *m, int indice_hueco, Proceso p) {
0032     // Obtenemos un puntero a la partición hueco que vamos a ocupar
0033     Particion *hueco = &m->particiones[indice_hueco];
0034
0035     // Verificamos que el hueco es libre y que cabe el proceso si no, error
0036     if (hueco->estado != 0 || hueco->tamano < p.mem_requerida) {
0037         printf("ERROR: no se puede asignar el proceso \n");
0038         return 0;
0039     }
0040
0041     // CASO A: Ajuste Exacto (El proceso ocupa todo el hueco)
0042     if (hueco->tamano == p.mem_requerida) {
0043         hueco->estado = 1;
0044         strcpy(hueco->nombre_proceso, p.nombre);
0045     } else {
0046         // CASO B: Hay que dividir (El hueco es más grande)
0047         // 1. Verificamos que no desbordamos el array
0048         if (m->cant_particiones >= MAX_PARTICIONES) return 1;
0049     }
  
```

```

00050 // 2. DESPLAZAMIENTO: Movemos todo una posición a la derecha
00051 // desde el final hasta el siguiente al hueco actual
00052 for (int i = m->cant_particiones; i > indice_hueco + 1; i--) {
00053     m->particiones[i] = m->particiones[i - 1];
00054 }
00055
00056 int tamano_restante = hueco->tamano - p.mem_requerida;
00057 int dir_nueva = hueco->dir_inicio + p.mem_requerida;
00058
00059 // 3. Crear el NUEVO hueco restante en la posición siguiente (i+1)
00060 m->particiones[indice_hueco + 1].dir_inicio = dir_nueva;
00061 m->particiones[indice_hueco + 1].tamano = tamano_restante;
00062 m->particiones[indice_hueco + 1].estado = 0; // LIBRE
00063 strcpy(m->particiones[indice_hueco + 1].nombre_proceso, "HUECO");
00064
00065 // 4. Actualizar el hueco actual para convertirlo en PROCESO
00066 hueco->tamano = p.mem_requerida;
00067 hueco->estado = 1; // OCUPADO
00068 strcpy(hueco->nombre_proceso, p.nombre);
00069
00070 // 5. Actualizamos contadores
00071 m->cant_particiones++;
00072 }
00073 // Actualizamos el puntero para Next Fit (para cuando lo implementemos)
00074 m->ultimo_indice_asignado = indice_hueco;
00075
00076 return 1; // Éxito
00077 }
00078
00079 void compactar(Memoria *m) {
00080     // Recorremos hasta el penúltimo elemento
00081     for (int i = 0; i < m->cant_particiones - 1; i++) {
00082         // Si la partición actual y la siguiente son huecos, las unimos
00083         if (m->particiones[i].estado == 0 && m->particiones[i+1].estado == 0) {
00084             // 1. Sumamos tamaños de las particiones
00085             m->particiones[i].tamano += m->particiones[i+1].tamano;
00086             // 2. Movemos todas las particiones siguientes una posición a la izquierda
00087             for (int j = i + 1; j < m->cant_particiones - 1; j++) {
00088                 m->particiones[j] = m->particiones[j+1];
00089             }
00090             // 3. Actualizamos el contador de particiones
00091             m->cant_particiones--;
00092             // 4. Retrocedemos el índice para revisar la nueva partición fusionada
00093             i--;
00094
00095             printf("[DEBUG] Compactacion realizada del proceso %s\n", m->particiones[i].nombre_proceso);
00096         }
00097     }
00098 }
00099
00100 bool liberar_proceso(Memoria *m, char *nombre_proceso) {
00101     for (int i = 0; i < m->cant_particiones; i++) {
00102         // Si encontramos el proceso y esta ocupado, lo liberamos
00103         if (m->particiones[i].estado == 1 && strcmp(m->particiones[i].nombre_proceso, nombre_proceso) == 0) {
00104             // 1. Lo convertimos en hueco pasando a estado 0
00105             m->particiones[i].estado = 0;
00106             strcpy(m->particiones[i].nombre_proceso, "HUECO");
00107             printf("[DEBUG] Proceso %s liberado en dir %d\n", nombre_proceso, m->particiones[i].dir_inicio);
00108
00109             // 2. Compactamos la memoria para unir huecos adyacentes
00110             compactar(m);
00111
00112             return true;
00113         }
00114     }
00115     return false;
00116 }
00117
00118 int buscar_hueco(Memoria *m, int mem_requerida, TipoAlgo tipo_algo) {
00119     // Algoritmo Primer Hueco
00120     if (tipo_algo == ALGO_PRIMER_HUECO) {
00121         // Recorremos todas las particiones buscando el primer hueco que quepa
00122         for (int i = 0; i < m->cant_particiones; i++) {
00123             // Si es un hueco libre y cabe el proceso
00124             if (m->particiones[i].estado == 0 && m->particiones[i].tamano >= mem_requerida) {
00125                 return i; // Devolvemos el índice del hueco encontrado
00126             }
00127         }
00128     }
00129
00130     // Algoritmo Siguiente Hueco
00131     if (tipo_algo == ALGO_SIGUIENTE_HUECO) {
00132         // El punto de partida es el último índice asignado
00133         int inicio = m->ultimo_indice_asignado;
00134         // Sacamos el total de particiones para poder hacer la búsqueda circular
00135         int total = m->cant_particiones;
00136

```

```

00137     // Recorremos todas las particiones empezando desde 'inicio' de forma circular
00138     for (int j = 0; j < total; j++) {
00139         // Calculamos el índice real con módulo para la circularidad
00140         int i = (inicio + j) % total;
00141
00142         // Si es un hueco libre y cabe el proceso devolvemos su índice en la memoria
00143         if (m->particiones[i].estado == 0 && m->particiones[i].tamano >= mem_requerida)
00144             return i;
00145     }
00146
00147     return -1;
00148 }
00149
00150
00151 int alinear_size(int size) {
00152     int tam_final; // Variable para el tamaño final alineado
00153
00154     // Si el tamaño es mayor que la unidad mínima, lo alineamos
00155     if (size > UNIDAD_MINIMA) {
00156         int bloques = size / 100; // Cantidad de bloques completos de 100
00157         if (size % 100 != 0) // Si hay resto
00158             tam_final = UNIDAD_MINIMA * bloques + 100; // Sumamos un bloque más
00159         else
00160             tam_final = bloques * 100; // Exacto, no hay resto
00161     } else
00162         tam_final = UNIDAD_MINIMA; // Si es menor o igual a 100, lo ajustamos a 100
00163
00164     return tam_final;
00165 }
00166
00167 bool asignar_proceso(Memoria *m, Proceso p, TipoAlgo tipo_algo) {
00168     // Calculamos el tamaño real alineado
00169     int tam_real = alinear_size(p.mem_requerida);
00170
00171     // [Opcional] Debug para ver el cambio
00172     if (tam_real != p.mem_requerida)
00173         printf("[DEBUG Alineacion] Proceso %s pide %d pero ocupara %d\n", p.nombre, p.mem_requerida, tam_real);
00174
00175     // Buscamos un hueco adecuado con el tamaño real y el algoritmo indicado
00176     int pos_mem = buscar_hueco(m, tam_real, tipo_algo);
00177
00178     // Si no se encontró hueco, devolvemos false
00179     if (pos_mem == -1)
00180         return false;
00181
00182     // Actualizamos el requerimiento de memoria del proceso al tamaño real
00183     p.mem_requerida = tam_real;
00184
00185     // Intentamos ocupar el hueco encontrado
00186     return ocupar_memoria(m, pos_mem, p);
00187 }
00188
00189 void avanzar_tiempo(Memoria *m, Proceso procesos[], int num_procesos, int *reloj_actual, TipoAlgo algo, const char*
ruta_log) {
00190     printf("\n-----INSTANTE %d-----\n", *reloj_actual);
00191
00192     // Paso 1 y 2 -> Envejecimiento y finalización de procesos
00193     for (int i = 0; i < num_procesos; i++) {
00194         // Solo procesamos los que están en memoria y no han finalizado
00195         if (procesos[i].en_memoria && !procesos[i].finalizado) {
00196             // DEBUG: Ver cuánto le queda antes de restar ...
00197             printf("[DEBUG Vida] %s tiene %d ticks restantes.\n", procesos[i].nombre, procesos[i].t_restante);
00198             procesos[i].t_restante--; // Disminuimos su tiempo restante
00199
00200             // Si ya no le queda tiempo, lo finalizamos
00201             if (procesos[i].t_restante <= 0) {
00202                 printf("-> FIN: El proceso %s ha terminado. Liberando memoria...\n", procesos[i].nombre);
00203
00204                 // Liberamos su memoria
00205                 liberar_proceso(m, procesos[i].nombre);
00206
00207                 // Marcamos el proceso como finalizado
00208                 procesos[i].en_memoria = false;
00209                 procesos[i].finalizado = true;
00210             }
00211         }
00212     }
00213
00214     // Paso 3 -> Llegada de nuevos procesos
00215     for (int i = 0; i < num_procesos; i++) {
00216         // Se busca proceso que no esté en memoria, no finalizado y que haya llegado o su momento ya haya pasado
00217         if (!procesos[i].en_memoria && !procesos[i].finalizado && procesos[i].t_llegada <= *reloj_actual) {
00218             printf("-> LLEGADA: %s intenta entrar (Requiere %d)\n", procesos[i].nombre, procesos[i].mem_requerida);
00219
00220             // Intentamos asignarlo en memoria
00221             if (asignar_proceso(m, procesos[i], algo)) {
00222                 printf("EXITO: %s asignado en memoria \n", procesos[i].nombre);
00223             }
00224         }
00225     }
00226
00227     // Limpiamos la memoria temporal
00228     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00229
00230     // Actualizamos el reloj
00231     *reloj_actual++;
00232
00233     // Guardamos el log
00234     FILE *f = fopen(ruta_log, "a");
00235     if (f) {
00236         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00237         for (int i = 0; i < num_procesos; i++) {
00238             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00239         }
00240         fclose(f);
00241     }
00242
00243     // Limpiamos la memoria temporal
00244     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00245
00246     // Actualizamos el reloj
00247     *reloj_actual++;
00248
00249     // Guardamos el log
00250     FILE *f = fopen(ruta_log, "a");
00251     if (f) {
00252         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00253         for (int i = 0; i < num_procesos; i++) {
00254             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00255         }
00256         fclose(f);
00257     }
00258
00259     // Limpiamos la memoria temporal
00260     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00261
00262     // Actualizamos el reloj
00263     *reloj_actual++;
00264
00265     // Guardamos el log
00266     FILE *f = fopen(ruta_log, "a");
00267     if (f) {
00268         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00269         for (int i = 0; i < num_procesos; i++) {
00270             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00271         }
00272         fclose(f);
00273     }
00274
00275     // Limpiamos la memoria temporal
00276     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00277
00278     // Actualizamos el reloj
00279     *reloj_actual++;
00280
00281     // Guardamos el log
00282     FILE *f = fopen(ruta_log, "a");
00283     if (f) {
00284         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00285         for (int i = 0; i < num_procesos; i++) {
00286             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00287         }
00288         fclose(f);
00289     }
00290
00291     // Limpiamos la memoria temporal
00292     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00293
00294     // Actualizamos el reloj
00295     *reloj_actual++;
00296
00297     // Guardamos el log
00298     FILE *f = fopen(ruta_log, "a");
00299     if (f) {
00300         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00301         for (int i = 0; i < num_procesos; i++) {
00302             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00303         }
00304         fclose(f);
00305     }
00306
00307     // Limpiamos la memoria temporal
00308     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00309
00310     // Actualizamos el reloj
00311     *reloj_actual++;
00312
00313     // Guardamos el log
00314     FILE *f = fopen(ruta_log, "a");
00315     if (f) {
00316         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00317         for (int i = 0; i < num_procesos; i++) {
00318             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00319         }
00320         fclose(f);
00321     }
00322
00323     // Limpiamos la memoria temporal
00324     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00325
00326     // Actualizamos el reloj
00327     *reloj_actual++;
00328
00329     // Guardamos el log
00330     FILE *f = fopen(ruta_log, "a");
00331     if (f) {
00332         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00333         for (int i = 0; i < num_procesos; i++) {
00334             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00335         }
00336         fclose(f);
00337     }
00338
00339     // Limpiamos la memoria temporal
00340     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00341
00342     // Actualizamos el reloj
00343     *reloj_actual++;
00344
00345     // Guardamos el log
00346     FILE *f = fopen(ruta_log, "a");
00347     if (f) {
00348         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00349         for (int i = 0; i < num_procesos; i++) {
00350             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00351         }
00352         fclose(f);
00353     }
00354
00355     // Limpiamos la memoria temporal
00356     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00357
00358     // Actualizamos el reloj
00359     *reloj_actual++;
00360
00361     // Guardamos el log
00362     FILE *f = fopen(ruta_log, "a");
00363     if (f) {
00364         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00365         for (int i = 0; i < num_procesos; i++) {
00366             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00367         }
00368         fclose(f);
00369     }
00370
00371     // Limpiamos la memoria temporal
00372     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00373
00374     // Actualizamos el reloj
00375     *reloj_actual++;
00376
00377     // Guardamos el log
00378     FILE *f = fopen(ruta_log, "a");
00379     if (f) {
00380         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00381         for (int i = 0; i < num_procesos; i++) {
00382             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00383         }
00384         fclose(f);
00385     }
00386
00387     // Limpiamos la memoria temporal
00388     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00389
00390     // Actualizamos el reloj
00391     *reloj_actual++;
00392
00393     // Guardamos el log
00394     FILE *f = fopen(ruta_log, "a");
00395     if (f) {
00396         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00397         for (int i = 0; i < num_procesos; i++) {
00398             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00399         }
00400         fclose(f);
00401     }
00402
00403     // Limpiamos la memoria temporal
00404     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00405
00406     // Actualizamos el reloj
00407     *reloj_actual++;
00408
00409     // Guardamos el log
00410     FILE *f = fopen(ruta_log, "a");
00411     if (f) {
00412         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00413         for (int i = 0; i < num_procesos; i++) {
00414             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00415         }
00416         fclose(f);
00417     }
00418
00419     // Limpiamos la memoria temporal
00420     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00421
00422     // Actualizamos el reloj
00423     *reloj_actual++;
00424
00425     // Guardamos el log
00426     FILE *f = fopen(ruta_log, "a");
00427     if (f) {
00428         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00429         for (int i = 0; i < num_procesos; i++) {
00430             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00431         }
00432         fclose(f);
00433     }
00434
00435     // Limpiamos la memoria temporal
00436     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00437
00438     // Actualizamos el reloj
00439     *reloj_actual++;
00440
00441     // Guardamos el log
00442     FILE *f = fopen(ruta_log, "a");
00443     if (f) {
00444         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00445         for (int i = 0; i < num_procesos; i++) {
00446             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00447         }
00448         fclose(f);
00449     }
00450
00451     // Limpiamos la memoria temporal
00452     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00453
00454     // Actualizamos el reloj
00455     *reloj_actual++;
00456
00457     // Guardamos el log
00458     FILE *f = fopen(ruta_log, "a");
00459     if (f) {
00460         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00461         for (int i = 0; i < num_procesos; i++) {
00462             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00463         }
00464         fclose(f);
00465     }
00466
00467     // Limpiamos la memoria temporal
00468     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00469
00470     // Actualizamos el reloj
00471     *reloj_actual++;
00472
00473     // Guardamos el log
00474     FILE *f = fopen(ruta_log, "a");
00475     if (f) {
00476         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00477         for (int i = 0; i < num_procesos; i++) {
00478             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00479         }
00480         fclose(f);
00481     }
00482
00483     // Limpiamos la memoria temporal
00484     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00485
00486     // Actualizamos el reloj
00487     *reloj_actual++;
00488
00489     // Guardamos el log
00490     FILE *f = fopen(ruta_log, "a");
00491     if (f) {
00492         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00493         for (int i = 0; i < num_procesos; i++) {
00494             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00495         }
00496         fclose(f);
00497     }
00498
00499     // Limpiamos la memoria temporal
00500     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00501
00502     // Actualizamos el reloj
00503     *reloj_actual++;
00504
00505     // Guardamos el log
00506     FILE *f = fopen(ruta_log, "a");
00507     if (f) {
00508         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00509         for (int i = 0; i < num_procesos; i++) {
00510             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00511         }
00512         fclose(f);
00513     }
00514
00515     // Limpiamos la memoria temporal
00516     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00517
00518     // Actualizamos el reloj
00519     *reloj_actual++;
00520
00521     // Guardamos el log
00522     FILE *f = fopen(ruta_log, "a");
00523     if (f) {
00524         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00525         for (int i = 0; i < num_procesos; i++) {
00526             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00527         }
00528         fclose(f);
00529     }
00530
00531     // Limpiamos la memoria temporal
00532     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00533
00534     // Actualizamos el reloj
00535     *reloj_actual++;
00536
00537     // Guardamos el log
00538     FILE *f = fopen(ruta_log, "a");
00539     if (f) {
00540         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00541         for (int i = 0; i < num_procesos; i++) {
00542             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00543         }
00544         fclose(f);
00545     }
00546
00547     // Limpiamos la memoria temporal
00548     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00549
00550     // Actualizamos el reloj
00551     *reloj_actual++;
00552
00553     // Guardamos el log
00554     FILE *f = fopen(ruta_log, "a");
00555     if (f) {
00556         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00557         for (int i = 0; i < num_procesos; i++) {
00558             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00559         }
00560         fclose(f);
00561     }
00562
00563     // Limpiamos la memoria temporal
00564     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00565
00566     // Actualizamos el reloj
00567     *reloj_actual++;
00568
00569     // Guardamos el log
00570     FILE *f = fopen(ruta_log, "a");
00571     if (f) {
00572         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00573         for (int i = 0; i < num_procesos; i++) {
00574             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00575         }
00576         fclose(f);
00577     }
00578
00579     // Limpiamos la memoria temporal
00580     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00581
00582     // Actualizamos el reloj
00583     *reloj_actual++;
00584
00585     // Guardamos el log
00586     FILE *f = fopen(ruta_log, "a");
00587     if (f) {
00588         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00589         for (int i = 0; i < num_procesos; i++) {
00590             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00591         }
00592         fclose(f);
00593     }
00594
00595     // Limpiamos la memoria temporal
00596     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00597
00598     // Actualizamos el reloj
00599     *reloj_actual++;
00600
00601     // Guardamos el log
00602     FILE *f = fopen(ruta_log, "a");
00603     if (f) {
00604         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00605         for (int i = 0; i < num_procesos; i++) {
00606             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00607         }
00608         fclose(f);
00609     }
00610
00611     // Limpiamos la memoria temporal
00612     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00613
00614     // Actualizamos el reloj
00615     *reloj_actual++;
00616
00617     // Guardamos el log
00618     FILE *f = fopen(ruta_log, "a");
00619     if (f) {
00620         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00621         for (int i = 0; i < num_procesos; i++) {
00622             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00623         }
00624         fclose(f);
00625     }
00626
00627     // Limpiamos la memoria temporal
00628     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00629
00630     // Actualizamos el reloj
00631     *reloj_actual++;
00632
00633     // Guardamos el log
00634     FILE *f = fopen(ruta_log, "a");
00635     if (f) {
00636         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00637         for (int i = 0; i < num_procesos; i++) {
00638             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00639         }
00640         fclose(f);
00641     }
00642
00643     // Limpiamos la memoria temporal
00644     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00645
00646     // Actualizamos el reloj
00647     *reloj_actual++;
00648
00649     // Guardamos el log
00650     FILE *f = fopen(ruta_log, "a");
00651     if (f) {
00652         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00653         for (int i = 0; i < num_procesos; i++) {
00654             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00655         }
00656         fclose(f);
00657     }
00658
00659     // Limpiamos la memoria temporal
00660     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00661
00662     // Actualizamos el reloj
00663     *reloj_actual++;
00664
00665     // Guardamos el log
00666     FILE *f = fopen(ruta_log, "a");
00667     if (f) {
00668         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00669         for (int i = 0; i < num_procesos; i++) {
00670             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00671         }
00672         fclose(f);
00673     }
00674
00675     // Limpiamos la memoria temporal
00676     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00677
00678     // Actualizamos el reloj
00679     *reloj_actual++;
00680
00681     // Guardamos el log
00682     FILE *f = fopen(ruta_log, "a");
00683     if (f) {
00684         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00685         for (int i = 0; i < num_procesos; i++) {
00686             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00687         }
00688         fclose(f);
00689     }
00690
00691     // Limpiamos la memoria temporal
00692     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00693
00694     // Actualizamos el reloj
00695     *reloj_actual++;
00696
00697     // Guardamos el log
00698     FILE *f = fopen(ruta_log, "a");
00699     if (f) {
00700         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00701         for (int i = 0; i < num_procesos; i++) {
00702             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00703         }
00704         fclose(f);
00705     }
00706
00707     // Limpiamos la memoria temporal
00708     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00709
00710     // Actualizamos el reloj
00711     *reloj_actual++;
00712
00713     // Guardamos el log
00714     FILE *f = fopen(ruta_log, "a");
00715     if (f) {
00716         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00717         for (int i = 0; i < num_procesos; i++) {
00718             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00719         }
00720         fclose(f);
00721     }
00722
00723     // Limpiamos la memoria temporal
00724     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00725
00726     // Actualizamos el reloj
00727     *reloj_actual++;
00728
00729     // Guardamos el log
00730     FILE *f = fopen(ruta_log, "a");
00731     if (f) {
00732         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00733         for (int i = 0; i < num_procesos; i++) {
00734             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00735         }
00736         fclose(f);
00737     }
00738
00739     // Limpiamos la memoria temporal
00740     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00741
00742     // Actualizamos el reloj
00743     *reloj_actual++;
00744
00745     // Guardamos el log
00746     FILE *f = fopen(ruta_log, "a");
00747     if (f) {
00748         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00749         for (int i = 0; i < num_procesos; i++) {
00750             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00751         }
00752         fclose(f);
00753     }
00754
00755     // Limpiamos la memoria temporal
00756     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00757
00758     // Actualizamos el reloj
00759     *reloj_actual++;
00760
00761     // Guardamos el log
00762     FILE *f = fopen(ruta_log, "a");
00763     if (f) {
00764         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00765         for (int i = 0; i < num_procesos; i++) {
00766             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00767         }
00768         fclose(f);
00769     }
00770
00771     // Limpiamos la memoria temporal
00772     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00773
00774     // Actualizamos el reloj
00775     *reloj_actual++;
00776
00777     // Guardamos el log
00778     FILE *f = fopen(ruta_log, "a");
00779     if (f) {
00780         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00781         for (int i = 0; i < num_procesos; i++) {
00782             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00783         }
00784         fclose(f);
00785     }
00786
00787     // Limpiamos la memoria temporal
00788     memset(procesos, 0, sizeof(Proceso) * num_procesos);
00789
00790     // Actualizamos el reloj
00791     *reloj_actual++;
00792
00793     // Guardamos el log
00794     FILE *f = fopen(ruta_log, "a");
00795     if (f) {
00796         fprintf(f, "\n-----INSTANTE %d-----\n", *reloj_actual);
00797         for (int i = 0; i < num_procesos; i++) {
00798             fprintf(f, "%s (%d, %d, %d, %d)\n", procesos[i].nombre, procesos[i].t_restante, procesos[i].mem_requerida, procesos[i].finalizado, procesos[i].en_memoria);
00799         }
00800         fclose(f);
00801     }
00802
00803     // Limpiamos la memoria temporal
00804     memset(proces
```

```

00223     procesos[i].en_memoria = true;
00224     // Inicializamos su tiempo restante
00225     procesos[i].t_restante = procesos[i].t_ejecucion;
00226 } else {
00227     // No se pudo asignar, se queda esperando, al ser t_llegada <= reloj_actual, volverá a intentar en el siguiente
00228     tick
00229     printf("ESPERA: No hay hueco para %s. Esperara al siguiente tick \n", procesos[i].nombre);
00230 }
00231 }
00232
00233 guardar_estado(ruta_log, m, *reloj_actual);
00234
00235 (*reloj_actual)++;
00236 }
```

12.13. Referencia del archivo src/sim_engine.h

Motor de simulación de gestión de memoria con particiones variables.

#include <stdbool.h>

Gráfico de dependencias incluidas en sim_engine.h:

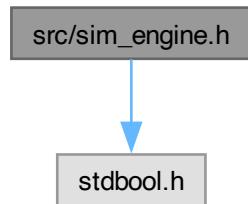
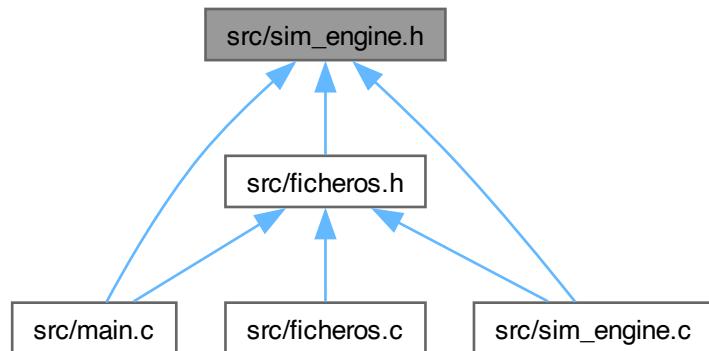


Gráfico de los archivos que directa o indirectamente incluyen a este archivo:



Estructuras de datos

- struct Proceso

Estructura que representa un proceso en el simulador.

- struct **Particion**
Estructura que representa una partición de memoria.
- struct **Memoria**
Estructura principal que representa la memoria del sistema.

defines

- #define **MEMORIA_TOTAL** 2000
Tamaño total de la memoria simulada (en unidades).
- #define **UNIDAD_MINIMA** 100
Unidad mínima de asignación. Toda memoria se alinea a múltiplos de este valor.
- #define **MAX_PARTICIONES** 50
Máximo número de particiones simultáneas en memoria.
- #define **MAX_PROCESOS** 100
Máximo número de procesos que puede manejar la simulación.

Enumeraciones

- enum **TipoAlgo** { **ALGO_PRIMER_HUECO** , **ALGO_SIGUIENTE_HUECO** }
Algoritmos de asignación de memoria disponibles.

Funciones

- void **inicializar_memoria** (**Memoria** *m)
Inicializa la memoria con un único hueco libre.
- void **mostrar_estado** (**Memoria** *m)
Muestra el estado actual de la memoria en consola.
- int **ocupar_memoria** (**Memoria** *m, int indice_hueco, **Proceso** p)
Ocupa un hueco de memoria con un proceso.
- void **compactar** (**Memoria** *m)
Compacta la memoria uniendo huecos adyacentes.
- bool **liberar_proceso** (**Memoria** *m, char *nombre_proceso)
Libera un proceso de la memoria.
- int **buscar_hueco** (**Memoria** *m, int mem_requerida, **TipoAlgo** tipo_algo)
Busca un hueco adecuado según el algoritmo especificado.
- int **alinear_size** (int size)
Alinea un tamaño a múltiplos de UNIDAD_MINIMA.
- bool **asignar_proceso** (**Memoria** *m, **Proceso** p, **TipoAlgo** tipo_algo)
Asigna un proceso a la memoria.
- void **avanzar_tiempo** (**Memoria** *m, **Proceso** procesos[], int num_procesos, int *reloj_actual, **TipoAlgo** algo, const char *ruta_log)
Avanza un tick en la simulación.

12.13.1. Descripción detallada

Motor de simulación de gestión de memoria con particiones variables.
Este módulo implementa la lógica central del simulador, incluyendo:

- Gestión de particiones de memoria (asignación/liberación)
- Algoritmos First Fit y Next Fit
- Compactación de huecos adyacentes
- Alineación de memoria a múltiplos de UNIDAD_MINIMA

Autor

Julian Hinojosa Gil

Fecha

2025

Versión

1.0

Definición en el archivo [sim_engine.h](#).

12.13.2. Documentación de enumeraciones

12.13.2.1. TipoAlgo

enum **TipoAlgo**

Algoritmos de asignación de memoria disponibles.

Ver también

[buscar_hueco\(\)](#)

[asignar_proceso\(\)](#)

Valores de enumeraciones

ALGO_PRIMER_HUECO	First Fit: busca desde el inicio de memoria
ALGO_SIGUIENTE_HUECO	Next Fit: busca desde la última posición asignada

Ejemplos

[/Users/julianhinojosagil/Documents/Dev/noob-code/UA/year-2/OS/practica3/src/sim_engine.h](#).

Definición en la línea 102 del archivo [sim_engine.h](#).

12.13.3. Documentación de funciones

12.13.3.1. alinear_size()

```
int alinear_size (  
    int size)
```

Alinea un tamaño a múltiplos de UNIDAD_MINIMA.

Redondea hacia arriba para garantizar que la memoria asignada sea siempre múltiplo de 100.

Parámetros

in	size	Tamaño solicitado original
----	------	----------------------------

Devuelve

Tamaño alineado (múltiplo de UNIDAD_MINIMA)

Ejemplos:

- `alinear_size(50) → 100`
- `alinear_size(100) → 100`
- `alinear_size(243) → 300`

Definición en la línea 151 del archivo `sim_engine.c`.

Hace referencia a `UNIDAD_MINIMA`.

Referenciado por `asignar_proceso()`.

Gráfico de llamadas a esta función:



12.13.3.2. `asignar_proceso()`

```
bool asignar_proceso (
    Memoria * m,
    Proceso p,
    TipoAlgo tipo_algo)
```

Asigna un proceso a la memoria.

Realiza el flujo completo de asignación:

1. Alinea la memoria requerida
2. Busca un hueco según el algoritmo
3. Ocupa el hueco encontrado

Parámetros

in,out	m	Puntero a la estructura de memoria
in	p	Proceso a asignar
in	tipo_algo	Algoritmo de búsqueda a usar

Devuelve

`true` si se asignó correctamente
`false` si no hay espacio suficiente

Ver también

`alinear_size()`, `buscar_hueco()`, `ocupar_memoria()`

Definición en la línea 167 del archivo `sim_engine.c`.

Hace referencia a `alinear_size()`, `buscar_hueco()`, `Proceso::mem_requerida`, `Proceso::nombre` y `ocupar_memoria()`.

Referenciado por `avanzar_tiempo()`.

Gráfico de llamadas de esta función:

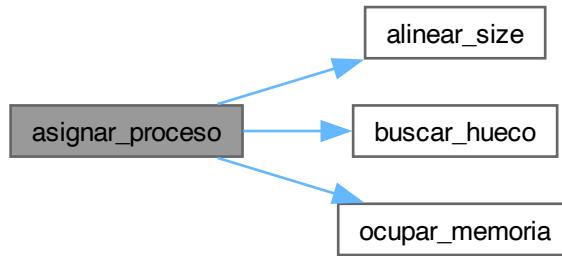
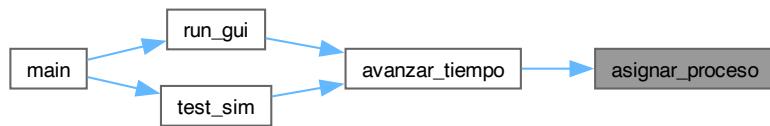


Gráfico de llamadas a esta función:



12.13.3.3. avanzar_tiempo()

```
void avanzar_tiempo (
    Memoria * m,
    Proceso procesos[],
    int num_procesos,
    int * reloj_actual,
    TipoAlgo algo,
    const char * ruta_log)
```

Avanza un tick en la simulación.

Ejecuta la lógica de un instante de tiempo:

1. Envejece procesos en memoria (decrementa t_restante)
2. Finaliza procesos cuyo tiempo restante llegó a 0
3. Intenta cargar nuevos procesos de la cola
4. Guarda el estado en el log
5. Incrementa el reloj

Parámetros

in,out	m	Puntero a la estructura de memoria
in,out	procesos	Array de procesos a gestionar

in	num_procesos	Número de procesos en el array
in,out	reloj_actual	Puntero al reloj de simulación
in	algo	Algoritmo de asignación a utilizar
in	ruta_log	Ruta del archivo de log para guardar estado

Postcondición

(*reloj_actual) se incrementa en 1

Definición en la línea 189 del archivo [sim_engine.c](#).

Hace referencia a [asignar_proceso\(\)](#), [Proceso::en_memoria](#), [Proceso::finalizado](#), [guardar_estado\(\)](#), [liberar_proceso\(\)](#), [Proceso::t_ejecucion](#) y [Proceso::t_restante](#).

Referenciado por [run_gui\(\)](#) y [test_sim\(\)](#).

Gráfico de llamadas de esta función:

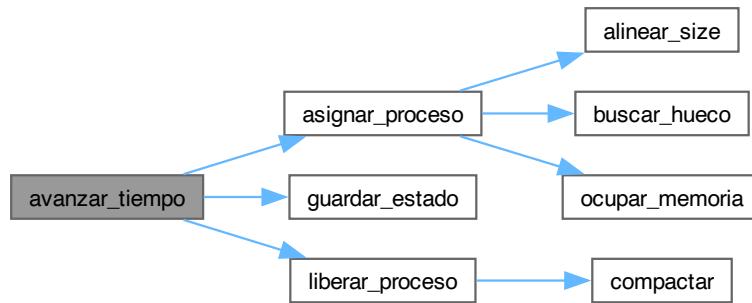
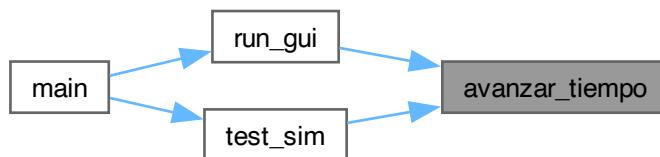


Gráfico de llamadas a esta función:



12.13.3.4. buscar_hueco()

```
int buscar_hueco (
    Memoria * m,
    int mem_requerida,
    TipoAlgo tipo_algo)
```

Busca un hueco adecuado según el algoritmo especificado.

Implementa los algoritmos:

- First Fit: Busca desde el inicio ($O(n)$ peor caso)

- Next Fit: Busca desde última posición (búsqueda circular)

Parámetros

in	m	Puntero a la estructura de memoria
in	mem_← requerida	Memoria requerida (ya alineada)
in	tipo_algo	Algoritmo a utilizar

Devuelve

Índice del hueco encontrado (0 a cant_particiones-1)

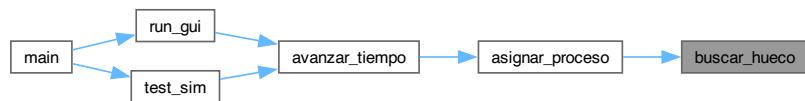
-1 si no hay hueco suficiente

Definición en la línea 118 del archivo [sim_engine.c](#).

Hace referencia a [ALGO_PRIMER_HUECO](#), [ALGO_SIGUIENTE_HUECO](#), [Memoria::cant_particiones](#), [Particion::estado](#), [Memoria::particiones](#), [Particion::tamano](#) y [Memoria::ultimo_indice_asignado](#).

Referenciado por [asignar_proceso\(\)](#).

Gráfico de llamadas a esta función:



12.13.3.5. compactar()

```
void compactar (
    Memoria * m)
```

Compacta la memoria uniendo huecos adyacentes.

Recorre las particiones y fusiona huecos consecutivos en uno solo. Se llama automáticamente después de liberar un proceso.

Parámetros

in,out	m	Puntero a la estructura de memoria
--------	---	------------------------------------

Postcondición

No hay dos huecos consecutivos en el array

cant_particiones puede disminuir

Definición en la línea 79 del archivo [sim_engine.c](#).

Hace referencia a [Memoria::cant_particiones](#), [Particion::estado](#), [Particion::nombre_proceso](#), [Memoria::particiones](#) y [Particion::tamano](#).

Referenciado por [liberar_proceso\(\)](#).

Gráfico de llamadas a esta función:



12.13.3.6. inicializar_memoria()

```
void inicializar_memoria (
    Memoria * m)
```

Inicializa la memoria con un único hueco libre.

Configura la memoria con una sola partición de tipo hueco que ocupa todo el espacio disponible (MEMORIA_TOTAL).

Parámetros

out	m	Puntero a la estructura de memoria a inicializar
-----	---	--

Precondición

`m != NULL`

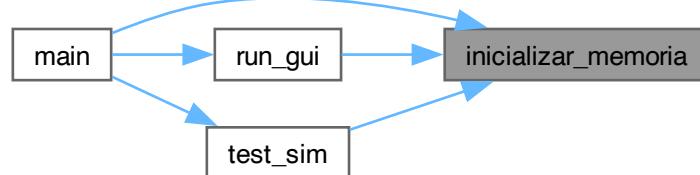
Postcondición

```
m->cant_particiones == 1
m->particiones[0].tamano == MEMORIA_TOTAL
```

Definición en la línea 7 del archivo `sim_engine.c`.

Hace referencia a `Memoria::cant_particiones`, `Particion::dir_inicio`, `Particion::estado`, `MEMORIA_TOTAL`, `Particion::nombre_proceso`, `Memoria::particiones`, `Particion::tamano` y `Memoria::ultimo_indice_asignado`. Referenciado por `main()`, `run_gui()` y `test_sim()`.

Gráfico de llamadas a esta función:



12.13.3.7. liberar_proceso()

```
bool liberar_proceso (
    Memoria * m,
    char * nombre_proceso)
```

Libera un proceso de la memoria.

Busca el proceso por nombre, lo convierte en hueco y compacta.

Parámetros

in,out	m	Puntero a la estructura de memoria
in	nombre_proceso	Nombre del proceso a liberar

Devuelve

true si se encontró y liberó el proceso

false si el proceso no estaba en memoria

Nota

Llama automáticamente a [compactar\(\)](#) tras liberar

Definición en la línea 100 del archivo [sim_engine.c](#).

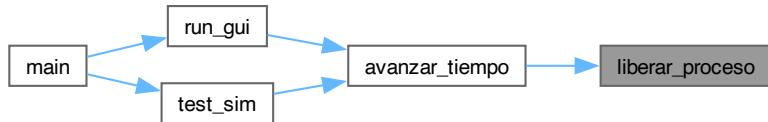
Hace referencia a [Memoria::cant_particiones](#), [compactar\(\)](#), [Particion::dir_inicio](#), [Particion::estado](#), [Particion::nombre_proceso](#) y [Memoria::particiones](#).

Referenciado por [avanzar_tiempo\(\)](#).

Gráfico de llamadas de esta función:



Gráfico de llamadas a esta función:



12.13.3.8. mostrar_estado()

```
void mostrar_estado (
```

```
    Memoria * m)
```

Muestra el estado actual de la memoria en consola.

Imprime cada partición en formato: [dir_inicio nombre tamaño]

Parámetros

in	m	Puntero a la estructura de memoria
----	---	------------------------------------

Nota

Solo para depuración/TUI, no afecta el estado

Definición en la línea 23 del archivo [sim_engine.c](#).

Hace referencia a [Memoria::cant_particiones](#), [Particion::dir_inicio](#), [Particion::nombre_proceso](#), [Memoria::particiones](#) y [Particion::tamano](#).

Referenciado por [test_sim\(\)](#).

Gráfico de llamadas a esta función:



12.13.3.9. ocupar_memoria()

```
int ocupar_memoria (
    Memoria * m,
    int indice_hueco,
    Proceso p)
```

Ocupa un hueco de memoria con un proceso.

Maneja dos casos:

- Ajuste exacto: El proceso ocupa todo el hueco
- División: Se crea una nueva partición con el espacio sobrante

Parámetros

in,out	m	Puntero a la estructura de memoria
in	indice_hueco	Índice del hueco en el array de particiones
in	p	Proceso a asignar (se usa nombre y mem_requerida)

Devuelve

- 1 si se asignó correctamente
 0 si hubo error (hueco ocupado, tamaño insuficiente o array lleno)

Precondición

$0 \leq \text{indice_hueco} < m->\text{cant_particiones}$
 $m->\text{particiones}[\text{indice_hueco}].\text{estado} == 0$ (es un hueco)

Atención

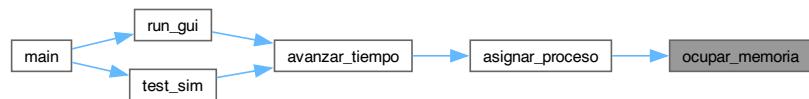
No verifica si el proceso ya existe en memoria

Definición en la línea 31 del archivo `sim_engine.c`.

Hace referencia a `Memoria::cant_particiones`, `Particion::dir_inicio`, `Particion::estado`, `MAX_PARTICIONES`, `Proceso::mem_requerida`, `Proceso::nombre`, `Particion::nombre_proceso`, `Memoria::particiones`, `Particion::tamano` y `Memoria::ultimo_indice_asignado`.

Referenciado por `asignar_proceso()`.

Gráfico de llamadas a esta función:



12.14. sim_engine.h

[Ir a la documentación de este archivo.](#)

```

00001 #ifndef SIM_ENGINE_H
00002 #define SIM_ENGINE_H
00003
00004 #include <stdbool.h>
00005
00029
00034
00036 #define MEMORIA_TOTAL 2000
00037
00039 #define UNIDAD_MINIMA 100
00040
00042 #define MAX_PARTICIONES 50
00043
00045 #define MAX_PROCESOS 100
00046
00048
00057 typedef struct {
00058     char nombre[10];
00059     int t_llegada;
00060     int mem_requerida;
00061     int t_ejecucion;
00062
00063     // Variables de control de estado
00064     int t_final;
00065     int t_restante;
00066     bool en_memoria;
00067     bool finalizado;
00068 } Proceso;
00069
00077 typedef struct {
00078     int dir_inicio;
00079     int tamano;
00080     int estado;
00081     char nombre_proceso[10];
00082 } Particion;
00083
00090 typedef struct {
00091     Particion particiones[MAX_PARTICIONES];
00092     int cant_particiones;
00093     int ultimo_indice_asignado;
00094 } Memoria;
00095
00102 typedef enum {
00103     ALGO_PRIMER_HUECO,
00104     ALGO_SIGUIENTE_HUECO
00105 } TipoAlgo;
00106
00119 void inicializar_memoria(Memoria *m);
00120
00130 void mostrar_estado(Memoria *m);
00131
00151 int ocupar_memoria(Memoria *m, int indice_hueco, Proceso p);
00152
00164 void compactar(Memoria *m);
  
```

```
00165
00179 bool liberar_proceso(Memoria *m, char *nombre_proceso);
00180
00195 int buscar_hueco(Memoria *m, int mem_requerida, TipoAlgo tipo_algo);
00196
00212 int alinear_size(int size);
00213
00231 bool asignar_proceso(Memoria *m, Proceso p, TipoAlgo tipo_algo);
00232
00252 void avanzar_tiempo(Memoria *m, Proceso procesos[], int num_procesos, int *reloj_actual, TipoAlgo algo, const char*
    ruta_log);
00253
00254 #endif // ENGINE_H
```


Capítulo 13

Ejemplos

13.1. /Users/julianhinojosagil/Documents/Dev/noob-code/UA/year-2/OS/practica3/src/sim_engine.h

```
Memoria m;
inicializar_memoria(&m);

Proceso p = {"P1", 0, 200, 5, 5, false, false};
asignar_proceso(&m, p, ALGO_PRIMER_HUECO);

#ifndef SIM_ENGINE_H
#define SIM_ENGINE_H

#include <stdbool.h>

#define MEMORIA_TOTAL 2000
#define UNIDAD_MINIMA 100
#define MAX_PARTICIONES 50
#define MAX_PROCESOS 100

typedef struct {
    char nombre[10];
    int t_llegada;
    int mem_requerida;
    int t_ejecucion;

    // Variables de control de estado
    int t_final;
    int t_restante;
    bool en_memoria;
    bool finalizado;
} Proceso;

typedef struct {
    int dir_inicio;
    int tamano;
    int estado;
    char nombre_proceso[10];
} Particion;

typedef struct {
    Particion particiones[MAX_PARTICIONES];
    int cant_particiones;
    int ultimo_indice_asignado;
} Memoria;

typedef enum {
    ALGO_PRIMER_HUECO,
    ALGO_SIGUIENTE_HUECO
}
```

```
} TipoAlgo;

void inicializar_memoria(Memoria *m);
void mostrar_estado(Memoria *m);
int ocupar_memoria(Memoria *m, int indice_hueco, Proceso p);
void compactar(Memoria *m);
bool liberar_proceso(Memoria *m, char *nombre_proceso);
int buscar_hueco(Memoria *m, int mem_requerida, TipoAlgo tipo_algo);
int alinear_size(int size);
bool asignar_proceso(Memoria *m, Proceso p, TipoAlgo tipo_algo);
void avanzar_tiempo(Memoria *m, Proceso procesos[], int num_procesos, int *reloj_actual, TipoAlgo algo, const char* ruta_log);
#endif // ENGINE_H
```

Índice alfabético

ALGO_PRIMER_HUECO
 sim_engine.h, 60

ALGO_SIGUIENTE_HUECO
 sim_engine.h, 60

alinear_size
 sim_engine.c, 47
 sim_engine.h, 60

ALTO_BARRA
 main.c, 39

Arquitectura del Sistema, 7

asignar_proceso
 sim_engine.c, 47
 sim_engine.h, 61

avanzar_tiempo
 sim_engine.c, 49
 sim_engine.h, 62

buscar_hueco
 sim_engine.c, 50
 sim_engine.h, 63

cant_particiones
 Memoria, 25

cargar_procesos
 ficheros.c, 30
 ficheros.h, 35

Changelog, 11

compactar
 sim_engine.c, 51
 sim_engine.h, 64

Constantes de Configuración, 21
 MAX_PARTICIONES, 21
 MAX_PROCESOS, 21
 MEMORIA_TOTAL, 21
 UNIDAD_MINIMA, 21

dir_inicio
 Particion, 26

docs/ARCHITECTURE.md, 29

docs/CHANGELOG.md, 29

docs/INSTALL.md, 29

en_memoria
 Proceso, 28

estado
 Particion, 26

ficheros.c
 cargar_procesos, 30
 guardar_estado, 31

 limpiar_log, 31
 SIZE_BUFFER_LECTURA, 30

ficheros.h
 cargar_procesos, 35
 guardar_estado, 36
 limpiar_log, 36

finalizado
 Proceso, 28

GestOMEMORIA - Simulador de Gestión de Memoria,
 1

guardar_estado
 ficheros.c, 31
 ficheros.h, 36

Guía de Instalación, 5

inicializar_memoria
 sim_engine.c, 51
 sim_engine.h, 65

liberar_proceso
 sim_engine.c, 52
 sim_engine.h, 65

limpiar_log
 ficheros.c, 31
 ficheros.h, 36

main
 main.c, 39

main.c
 ALTO_BARRA, 39
 main, 39
 MARGEN_DER, 39
 MARGEN_IZQ, 39
 run_gui, 40
 test_sim, 41
 WIN_HEIGHT, 39
 WIN_WIDTH, 39
 Y_BARRA, 39

MARGEN_DER
 main.c, 39

MARGEN_IZQ
 main.c, 39

MAX_PARTICIONES
 Constantes de Configuración, 21

MAX_PROCESOS
 Constantes de Configuración, 21

mem_requerida
 Proceso, 28

Memoria, 25

cant_particiones, 25
 particiones, 26
 ultimo_indice_asignado, 26
MEMORIA_TOTAL
 Constantes de Configuración, 21
mostrar_estado
 sim_engine.c, 53
 sim_engine.h, 66
nombre
 Proceso, 28
nombre_proceso
 Particion, 27
ocupar_memoria
 sim_engine.c, 54
 sim_engine.h, 67
Particion, 26
 dir_inicio, 26
 estado, 26
 nombre_proceso, 27
 tamano, 27
particiones
 Memoria, 26
Proceso, 27
 en_memoria, 28
 finalizado, 28
 mem_requerida, 28
 nombre, 28
 t_ejecucion, 28
 t_final, 28
 t_llegada, 28
 t_restante, 28
README.md, 29
Referencia del directorio docs, 23
Referencia del directorio src, 23
run_gui
main.c, 40
sim_engine.c
 alinear_size, 47
 asignar_proceso, 47
 avanzar_tiempo, 49
 buscar_hueco, 50
 compactar, 51
 inicializar_memoria, 51
 liberar_proceso, 52
 mostrar_estado, 53
 ocupar_memoria, 54
sim_engine.h
 ALGO_PRIMER_HUECO, 60
 ALGO_SIGUIENTE_HUECO, 60
 alinear_size, 60
 asignar_proceso, 61
 avanzar_tiempo, 62
 buscar_hueco, 63
 compactar, 64
 inicializar_memoria, 65
 liberar_proceso, 65
 mostrar_estado, 66
 ocupar_memoria, 67
 TipoAlgo, 60
SIZE_BUFFER_LECTURA
 ficheros.c, 30
src/ficheros.c, 29, 32
src/ficheros.h, 33, 37
src/main.c, 37, 42
src/sim_engine.c, 46, 55
src/sim_engine.h, 58, 68
t_ejecucion
 Proceso, 28
t_final
 Proceso, 28
t_llegada
 Proceso, 28
t_restante
 Proceso, 28
tamano
 Particion, 27
test_sim
 main.c, 41
TipoAlgo
 sim_engine.h, 60
ultimo_indice_asignado
 Memoria, 26
UNIDAD_MINIMA
 Constantes de Configuración, 21
WIN_HEIGHT
 main.c, 39
WIN_WIDTH
 main.c, 39
Y_BARRA
 main.c, 39