**Estructura de Datos**

**Informe Técnico del Proyecto: *“Sistema de Gestión de Procesos”***

# CAPÍTULO 1: Análisis del Problema

1. **Descripción del problema**El sistema de gestión de procesos en C + + tiene como objetivo simular el manejo de tareas en un entorno. Este sistema permite insertar procesos en una lista enlazada, encolar en una cola estática para su atención y asignarles bloques de memoria utilizando una pila. Además, incorpora operaciones de búsqueda y liberación de memoria, facilitando así la comprensión y aplicación de estructuras dinámicas y estáticas en un contexto práctico. El problema principal que resuelve este sistema es la administración eficiente de procesos, su prioridad y uso de memoria, utilizando estructuras de datos fundamentales como listas enlazadas, colas y pilas. **(Lo modifican)**
2. **Requerimientos del sistema**

* Funcionales
* Inserta procesos con ID, nombre, prioridad y asignación de memoria
* Busca procesos por ID
* Encola procesos existentes para simular
* Desencola procesos
* Muestra los procesos actualmente en la cola.
* Muestra los bloques de memoria asignados.
* Libera el último bloque de memoria asignado (LIFO).
* Finaliza la ejecución del programa.
* No funcionales
* El sistema debe ejecutarse en consola con una interfaz clara y amigable
* El código debe ser modular, reutilizable y comprensible para futuros desarrollos
* El sistema debe utilizar estructuras de datos dinámicas para flexibilidad
* El número máximo de procesos en la cola de CPU está limitado a 5 (definido por #define MAX 5).

1. **Estructuras de datos propuestas**

| Estructura | Uso Principal |
| --- | --- |
| Lista | Gestionar procesos |
| Cola | Simular el ingreso y salida de procesos |
| Pila | Gestionar bloques de memoria asignados (LIFO) |

1. **Justificación de la elección**

* Lista enlazada: Permite insertar, buscar y recorrer procesos de forma dinámica sin necesidad de conocer el número de elementos previamente.
* La **cola de prioridad** es ideal para simular la planificación de la CPU, donde los procesos con mayor prioridad deben ejecutarse antes.
* Pila dinámica: Refleja la gestión de memoria con una política LIFO, donde el último bloque de memoria asignado es el primero en liberarse, imitando el comportamiento de una pila de activación o stack frame.

# CAPÍTULO 2: Diseño de la Solución

1. **Descripción de estructuras de datos y operaciones:**

### Lista Enlazada (Gestor de Procesos)

**Descripción:** Se utilizó una **lista enlazada simple** para almacenar todos los procesos activos dentro del sistema. Cada nodo de la lista representa un proceso y contiene información clave como:

* id: identificador único del proceso.
* nombre: nombre asignado al proceso.
* prioridad: valor numérico que determina su orden de ejecución en la cola de planificación.
* estado: texto que indica el estado actual del proceso (por ejemplo: “nuevo”, “listo”, “ejecutando”).

La lista enlazada permite agregar procesos de forma dinámica sin necesidad de conocer de antemano la cantidad total, ya que cada nodo se enlaza al siguiente mediante un puntero. Esta estructura fue elegida por su eficiencia en operaciones como inserción, búsqueda y eliminación, sin requerir desplazamiento de elementos como ocurriría en un arreglo.

**Operaciones implementadas:**

* insertarProceso(): añade un nuevo nodo al final de la lista. Se utiliza cuando el usuario registra un proceso desde la consola.
* eliminarProceso(id): recorre la lista y elimina el nodo cuyo ID coincide con el especificado. Si el proceso está en el medio o final, se ajustan los punteros para preservar la integridad de la lista.
* buscarProceso(id o nombre): permite localizar un proceso por su ID (entero) o su nombre (string). Se devuelve un puntero al nodo si se encuentra, o NULL en caso contrario.
* modificarPrioridad(id, nuevaPrioridad): busca el proceso con el ID dado y actualiza su prioridad con el nuevo valor ingresado por el usuario.

**Ventajas clave:**

* Memoria eficiente: solo se asigna memoria cuando se necesita (por cada proceso nuevo).
* Flexibilidad: fácil eliminación e inserción sin necesidad de reorganizar toda la estructura.
* Modularidad: se puede reutilizar la lista en otros contextos (como la persistencia o la cola de ejecución).

### Cola de Prioridad (Planificador de CPU)

**Descripción:** Para simular la ejecución de procesos por parte de la CPU, se implementó una **cola de prioridad**. En esta estructura, los procesos se ordenan automáticamente según su nivel de prioridad al ser encolados. El proceso con mayor prioridad (valor más alto) se ubica al frente de la cola y será el siguiente en ejecutarse.

A diferencia de una cola FIFO estándar, esta cola inserta cada nuevo proceso en la posición adecuada para mantener el orden de prioridades. Esto permite que la CPU ejecute siempre el proceso más importante en primer lugar.

**Operaciones implementadas:**

* encolarProceso(proceso): inserta un proceso en la cola según su prioridad. Se compara con los elementos ya presentes y se posiciona donde corresponda.
* desencolarProceso(): elimina y retorna el proceso con mayor prioridad (el primero de la cola).
* mostrarCola(): imprime el estado actual de la cola, mostrando todos los procesos pendientes de ejecución y su prioridad.

**Ventajas clave:**

* Simula correctamente una política de planificación por prioridad (común en sistemas operativos).
* Asegura que los procesos críticos no esperen detrás de procesos menos importantes.
* Permite una ejecución justa y controlada.

### Pila (Gestor de Memoria)

**Descripción:** El sistema de gestión de memoria fue implementado mediante una **pila**, una estructura con comportamiento LIFO (Last In, First Out). Cada vez que un proceso requiere memoria, se le asigna un bloque (push). Cuando se libera memoria, se elimina el bloque más recientemente asignado (pop).

Cada bloque de memoria se representa como un nodo que contiene:

* blockId: identificador único del bloque.
* processId: ID del proceso al que se le asignó.
* size: cantidad de memoria asignada en ese bloque.

Esta lógica imita la asignación dinámica de recursos en sistemas reales, como el uso de stacks en funciones recursivas o contextos de procesos.

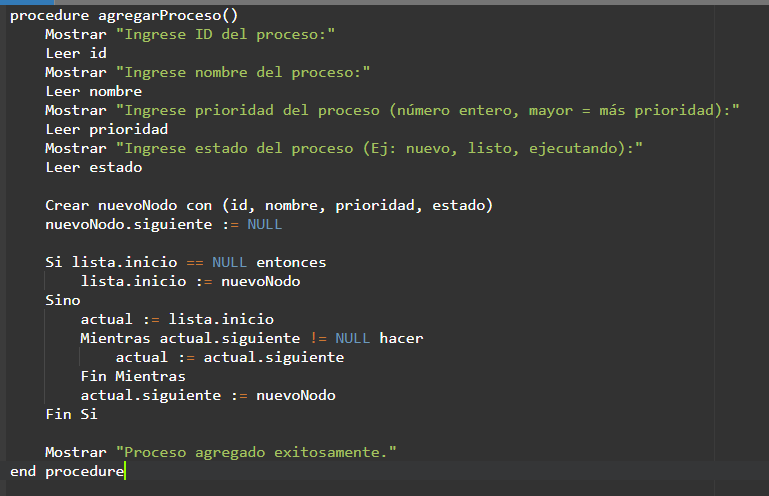
**Operaciones implementadas:**

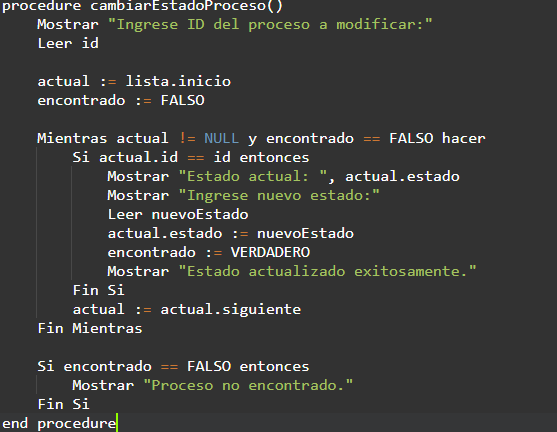
* asignarMemoria(procId, tamaño): crea un nuevo bloque de memoria y lo apila sobre la cima de la pila.
* liberarMemoria(): remueve el bloque superior, simulando la liberación del recurso más reciente.
* mostrarPila(): imprime todos los bloques de memoria actualmente asignados, desde el más reciente al más antiguo.

**Ventajas clave:**

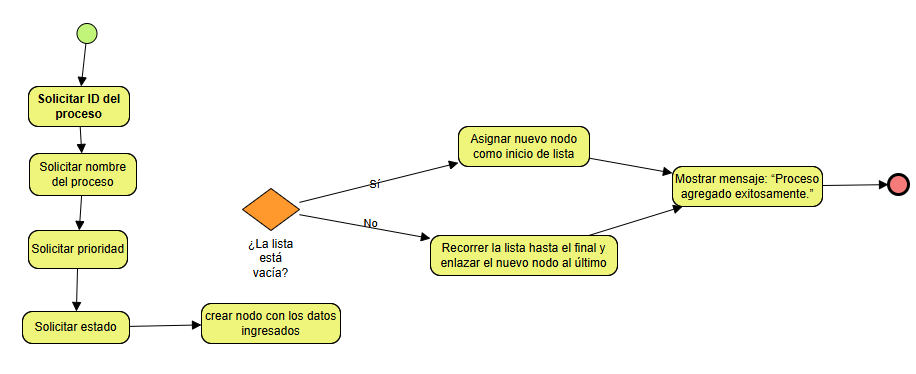
* Simplicidad de implementación y uso.
* Refleja un modelo natural de uso de memoria en muchos lenguajes y sistemas operativos.
* Permite liberar recursos de forma rápida y eficiente.

1. **Algoritmos principales:**

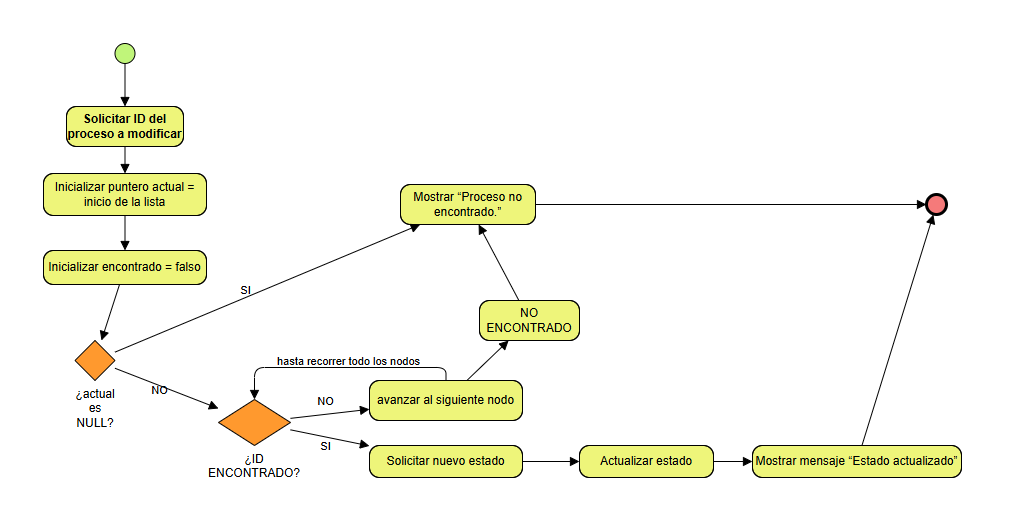
***Pseudocódigo para agregar un proceso a la lista enlazada:***

***Pseudocódigo para cambiar el estado de un proceso:***

1. **Diagramas de Flujo:**

**Diagrama de flujo: Agregar un proceso**

**Diagrama de flujo: Cambiar el estado de un proceso**



1. **Justificación del diseño:**

El diseño de este sistema fue cuidadosamente estructurado para responder a los desafíos de gestionar múltiples procesos de forma organizada, eficiente y coherente con el comportamiento de un sistema operativo. A continuación se exponen las justificaciones detalladas:

1. **Modularidad y separación de responsabilidades:**
   * Cada componente del sistema fue encapsulado en su propia clase (procesos, cola de planificación, pila de memoria), lo que permite desarrollar, probar y mantener cada parte de forma independiente. Esta estructura facilita también la colaboración en equipo, ya que distintos integrantes pueden trabajar en componentes distintos sin interferencias.
2. **Ajuste preciso entre estructura de datos y funcionalidad:**
   * El uso de una lista enlazada para almacenar procesos permite una gestión flexible, con inserciones y eliminaciones dinámicas, ideal para un entorno donde los procesos pueden ser creados y destruidos en cualquier momento.
   * La cola de prioridad garantiza que los procesos más importantes (según su nivel de prioridad) sean ejecutados antes. La implementación asegura que cada nuevo proceso se inserte en su posición correcta dentro de la cola, manteniendo el orden necesario sin necesidad de reordenar todo el conjunto.
   * La pila como gestor de memoria representa fielmente el principio LIFO usado en muchas arquitecturas reales, como en el manejo de la pila de llamadas o el uso de memoria temporal.
3. **Interacción clara con el usuario:**
   * Se implementó un menú en consola con navegación intuitiva que agrupa las funciones por subsistema. Esto permite que el usuario realice operaciones como registrar procesos, asignar memoria o planificar ejecuciones de forma secuencial y entendible.
   * Las entradas del usuario son solicitadas de forma explícita y se valida cada paso con mensajes de confirmación, mejorando la confiabilidad del sistema y la comprensión de las acciones realizadas.
4. **Persistencia entre sesiones:**
   * El sistema guarda automáticamente el estado de las estructuras dinámicas en archivos de texto plano (processes.txt, queue.txt, stack.txt). Esta funcionalidad garantiza que el trabajo realizado no se pierda tras cerrar el programa, lo cual es una mejora significativa frente a versiones completamente volátiles.
   * Esta característica también abre la puerta a funciones futuras como auditoría de acciones, recuperación de estados anteriores o importación/exportación de configuraciones.
5. **Código comprensible y extensible:**
   * Las funciones están claramente nombradas, y cada bloque del programa está acompañado por comentarios que explican su propósito.
   * La lógica fue estructurada en métodos simples y específicos, permitiendo que cualquier nuevo programador pueda entender rápidamente el sistema y modificarlo o extenderlo según las necesidades del curso o nuevos requerimientos.
6. **Facilidad para pruebas y validaciones:**
   * Gracias a la segmentación por módulos y a la estructura sencilla de las operaciones, cada módulo puede ser probado de forma aislada. Por ejemplo, la pila puede verificarse con una secuencia push/pop sin depender del sistema completo.
7. **Viabilidad académica y técnica:**
   * El diseño fue pensado para cumplir con los objetivos académicos de la asignatura: implementar estructuras dinámicas manualmente, resolver un problema realista y fomentar el trabajo colaborativo. A su vez, el sistema resultante tiene calidad suficiente para servir como base de proyectos más complejos o como ejemplo claro de aplicación práctica.

En conclusión, cada decisión de diseño está respaldada tanto por su adecuación técnica al problema como por su valor pedagógico dentro del contexto del curso. Se logró un equilibrio entre simplicidad, funcionalidad y escalabilidad, brindando un producto final robusto, entendible y ampliable.

# CAPÍTULO 3: Solución Final

1. **Código limpio, bien comentado y estructurado**

Código:

#include <iostream> // Biblioteca para entrada y salida estándar (cout, cin, etc.)

#include <fstream> // Biblioteca para manejo de archivos

#include <sstream> // Biblioteca para manipular strings como flujos (stringstream)

#include <string> // Biblioteca para usar el tipo de dato string

#include <limits> // Biblioteca para obtener límites de tipos de datos (como limpiar el buffer de entrada)

// Estructura que representa un proceso

struct Process {

int id; // Identificador único del proceso

std::string name; // Nombre del proceso

int priority; // Prioridad del proceso (entero)

std::string state; // Estado actual del proceso (ej. "Ejecutando", "Listo", etc.)

};

// Estructura de nodo para la lista enlazada simple de procesos

struct ProcessNode {

Process data; // Contiene la información del proceso

ProcessNode\* next; // Puntero al siguiente nodo de la lista

// Constructor que inicializa el nodo con los datos de un proceso

ProcessNode(const Process& p) : data(p), next(nullptr) {}

};

// Clase que gestiona una lista enlazada de procesos

class ProcessList {

private:

ProcessNode\* head; // Puntero al primer nodo (cabeza) de la lista

public:

// Constructor: al crear una nueva lista, la cabeza se inicializa como nula

ProcessList() : head(nullptr) {}

// Método para insertar un proceso al final de la lista

void insertProcess(const Process& p) {

ProcessNode\* newNode = new ProcessNode(p); // Se crea un nuevo nodo con el proceso recibido

if (!head) { // Si la lista está vacía

head = newNode; // El nuevo nodo se convierte en la cabeza

} else {

ProcessNode\* temp = head; // Se recorre la lista desde la cabeza

while (temp->next) temp = temp->next; // Se avanza hasta el último nodo

temp->next = newNode; // Se agrega el nuevo nodo al final

}

}

// Método para eliminar un proceso por su ID

bool removeProcess(int id) {

if (!head) return false; // Si la lista está vacía, no se puede eliminar

if (head->data.id == id) { // Si el proceso a eliminar está en la cabeza

ProcessNode\* toDelete = head; // Se guarda el nodo a eliminar

head = head->next; // Se actualiza la cabeza al siguiente nodo

delete toDelete; // Se libera la memoria del nodo eliminado

return true; // Retorna true indicando que se eliminó

}

ProcessNode\* prev = head; // Nodo anterior al actual

ProcessNode\* cur = head->next; // Nodo actual

while (cur) { // Se recorre la lista

if (cur->data.id == id) { // Si se encuentra el ID buscado

prev->next = cur->next; // Se desvincula el nodo actual

delete cur; // Se libera la memoria del nodo eliminado

return true; // Se indica que la eliminación fue exitosa

}

prev = cur; // Avanzar al siguiente nodo

cur = cur->next;

}

return false; // Si no se encontró el ID, retorna false

}

// Método para buscar un proceso por ID

ProcessNode\* searchById(int id) {

ProcessNode\* temp = head; // Comienza desde la cabeza de la lista

while (temp) { // Recorre toda la lista

if (temp->data.id == id) return temp; // Si encuentra el ID, retorna el nodo

temp = temp->next; // Avanza al siguiente nodo

}

return nullptr; // Si no se encuentra, retorna puntero nulo

}

// Método para buscar un proceso por nombre

ProcessNode\* searchByName(const std::string& name) {

ProcessNode\* temp = head; // Comienza desde la cabeza

while (temp) { // Recorre la lista

if (temp->data.name == name) return temp; // Si el nombre coincide, retorna el nodo

temp = temp->next; // Avanza al siguiente nodo

}

return nullptr; // Si no se encuentra, retorna puntero nulo

}

// Método para modificar la prioridad de un proceso dado su ID

bool modifyPriority(int id, int newPriority) {

ProcessNode\* node = searchById(id); // Busca el proceso por ID

if (!node) return false; // Si no se encuentra, retorna false

node->data.priority = newPriority; // Asigna la nueva prioridad

return true; // Retorna true indicando éxito

}

// Método para mostrar todos los procesos en la lista

void displayAll() {

if (!head) { // Si la lista está vacía

std::cout << "No hay procesos registrados.\n"; // Mensaje al usuario

return;

}

ProcessNode\* temp = head; // Puntero temporal para recorrer la lista

std::cout << "ID\tNombre\tPrioridad\tEstado\n"; // Encabezados de tabla

while (temp) { // Recorre toda la lista

std::cout << temp->data.id << "\t" // Imprime ID del proceso

<< temp->data.name << "\t" // Imprime nombre del proceso

<< temp->data.priority << "\t\t" // Imprime prioridad

<< temp->data.state << "\n"; // Imprime estado

temp = temp->next; // Avanza al siguiente nodo

}

}

// Método para guardar la lista de procesos en un archivo (persistencia de datos)

void saveToFile(const std::string& filename) {

std::ofstream file(filename); // Se abre un archivo de salida con el nombre especificado

ProcessNode\* temp = head; // Se usa un puntero temporal para recorrer la lista

while (temp) { // Mientras haya nodos en la lista

file << temp->data.id << "," // Escribe el ID del proceso seguido de una coma

<< temp->data.name << "," // Escribe el nombre del proceso

<< temp->data.priority << "," // Escribe la prioridad del proceso

<< temp->data.state << "\n"; // Escribe el estado del proceso y finaliza la línea

temp = temp->next; // Avanza al siguiente nodo

}

file.close(); // Se cierra el archivo

}

// Método para cargar los procesos desde un archivo (sobrescribe la lista actual)

void loadFromFile(const std::string& filename) {

// Primero libera la memoria de la lista actual

while (head) { // Mientras existan nodos en la lista

ProcessNode\* toDelete = head; // Se guarda el nodo actual para eliminarlo

head = head->next; // Se avanza al siguiente nodo

delete toDelete; // Se elimina el nodo anterior

}

std::ifstream file(filename); // Se abre el archivo en modo lectura

if (!file.is\_open()) return; // Si el archivo no se pudo abrir, se sale del método

std::string line; // Variable para guardar cada línea del archivo

while (std::getline(file, line)) { // Lee línea por línea hasta el final del archivo

std::stringstream ss(line); // Se crea un flujo de string para dividir los datos

Process p; // Se crea un objeto proceso temporal

std::string token; // Variable para almacenar cada campo leído

std::getline(ss, token, ','); // Lee el ID (hasta la coma)

p.id = std::stoi(token); // Convierte el ID a entero y lo asigna

std::getline(ss, p.name, ','); // Lee el nombre del proceso directamente

std::getline(ss, token, ','); // Lee la prioridad (hasta la coma)

p.priority = std::stoi(token); // Convierte la prioridad a entero y la asigna

std::getline(ss, p.state, ','); // Lee el estado del proceso directamente

insertProcess(p); // Inserta el proceso en la lista

}

file.close(); // Cierra el archivo al terminar

}

};

// Nodo para la cola de prioridad del planificador de CPU

struct QueueNode {

Process data; // Contiene el proceso

QueueNode\* next; // Puntero al siguiente nodo en la cola

// Constructor que inicializa el nodo con un proceso

QueueNode(const Process& p) : data(p), next(nullptr) {}

};

// Clase que implementa una cola de prioridad (mayor prioridad se atiende primero)

class PriorityQueue {

private:

QueueNode\* front; // Puntero al inicio de la cola (proceso con mayor prioridad)

public:

PriorityQueue() : front(nullptr) {} // Constructor: la cola comienza vacía

// Método para encolar un nuevo proceso según su prioridad

void enqueue(const Process& p) {

QueueNode\* newNode = new QueueNode(p); // Se crea un nuevo nodo con el proceso

// Si la cola está vacía o la prioridad del nuevo proceso es mayor a la del primero

if (!front || p.priority > front->data.priority) {

newNode->next = front; // El nuevo nodo apunta al actual frente

front = newNode; // Y se convierte en el nuevo frente de la cola

} else {

QueueNode\* temp = front; // Se recorre la cola desde el frente

// Se busca la posición donde insertar el nuevo proceso (orden descendente)

while (temp->next && temp->next->data.priority >= p.priority) {

temp = temp->next; // Avanza en la cola hasta encontrar lugar

}

newNode->next = temp->next; // Inserta el nodo entre temp y temp->next

temp->next = newNode; // Enlaza correctamente el nuevo nodo en la cola

}

}

// Método para desencolar (extraer) el proceso con mayor prioridad

bool dequeue(Process& p) {

if (!front) return false; // Si la cola está vacía, no hay proceso que atender

QueueNode\* toDelete = front; // Se guarda el nodo a eliminar (el primero)

p = front->data; // Se copia el proceso del frente al parámetro de salida

front = front->next; // Se actualiza el frente al siguiente nodo

delete toDelete; // Se elimina el nodo anterior (libera memoria)

return true; // Indica que el desencolado fue exitoso

}

// Método para mostrar la cola de prioridad actual en pantalla

void displayQueue() {

if (!front) { // Si el frente es nulo, la cola está vacía

std::cout << "La cola de prioridad está vacía.\n";

return; // Sale del método

}

std::cout << "ID\tNombre\tPrioridad\tEstado\n"; // Encabezado de la tabla

QueueNode\* temp = front; // Nodo temporal para recorrer la cola

while (temp) { // Mientras existan nodos

std::cout << temp->data.id << "\t" // Muestra el ID del proceso

<< temp->data.name << "\t" // Muestra el nombre del proceso

<< temp->data.priority << "\t\t" // Muestra la prioridad del proceso

<< temp->data.state << "\n"; // Muestra el estado del proceso

temp = temp->next; // Avanza al siguiente nodo

}

}

// Guardar la cola de prioridad en un archivo (persistencia)

void saveToFile(const std::string& filename) {

std::ofstream file(filename); // Abre un archivo en modo escritura

QueueNode\* temp = front; // Nodo temporal para recorrer la cola

while (temp) { // Mientras existan nodos

file << temp->data.id << "," // Escribe el ID

<< temp->data.name << "," // Escribe el nombre

<< temp->data.priority << "," // Escribe la prioridad

<< temp->data.state << "\n"; // Escribe el estado y termina la línea

temp = temp->next; // Avanza al siguiente nodo

}

file.close(); // Cierra el archivo

}

// Cargar cola de prioridad desde archivo (reemplaza contenido actual)

void loadFromFile(const std::string& filename) {

// Primero libera la cola actual

while (front) { // Mientras existan nodos

QueueNode\* toDelete = front; // Guarda el nodo actual

front = front->next; // Avanza al siguiente

delete toDelete; // Libera el nodo actual

}

std::ifstream file(filename); // Abre el archivo en modo lectura

if (!file.is\_open()) return; // Si no se puede abrir, se sale

std::string line; // Para almacenar cada línea

while (std::getline(file, line)) { // Lee línea por línea

std::stringstream ss(line); // Separa los campos usando stringstream

Process p; // Crea un proceso temporal

std::string token; // Almacena cada parte de la línea

std::getline(ss, token, ','); // Lee el ID

p.id = std::stoi(token); // Convierte a entero

std::getline(ss, p.name, ','); // Lee el nombre directamente

std::getline(ss, token, ','); // Lee la prioridad

p.priority = std::stoi(token); // Convierte a entero

std::getline(ss, p.state, ','); // Lee el estado directamente

enqueue(p); // Inserta el proceso en la cola según prioridad

}

file.close(); // Cierra el archivo

}

};

// Estructura para representar un bloque de memoria asignado a un proceso

struct MemoryBlock {

int blockId; // Identificador único del bloque de memoria

int processId; // ID del proceso al que se le asignó el bloque

int size; // Tamaño del bloque de memoria

};

// Nodo para la pila de bloques de memoria

struct StackNode {

MemoryBlock data; // Contiene el bloque de memoria

StackNode\* next; // Puntero al siguiente nodo en la pila

// Constructor que inicializa el nodo con un bloque de memoria

StackNode(const MemoryBlock& mb) : data(mb), next(nullptr) {}

};

// Clase que gestiona la memoria usando una pila (LIFO: último en entrar, primero en salir)

class MemoryStack {

private:

StackNode\* topNode; // Puntero al tope de la pila

int nextBlockId; // Contador para asignar IDs únicos a los bloques

public:

// Constructor: pila vacía y contador de bloques en 1

MemoryStack() : topNode(nullptr), nextBlockId(1) {}

// Push: asigna un nuevo bloque de memoria a un proceso

void push(int procId, int size) {

MemoryBlock mb; // Crea un nuevo bloque de memoria

mb.blockId = nextBlockId++; // Asigna un ID único y lo incrementa

mb.processId = procId; // Asigna el ID del proceso

mb.size = size; // Asigna el tamaño solicitado

StackNode\* newNode = new StackNode(mb); // Crea un nuevo nodo para la pila

newNode->next = topNode; // El nuevo nodo apunta al tope actual

topNode = newNode; // Se convierte en el nuevo tope de la pila

// Muestra mensaje de confirmación

std::cout << "Bloque " << mb.blockId << " asignado al proceso "

<< procId << " (tamaño " << size << ").\n";

}

// Pop: libera el último bloque de memoria asignado (el del tope)

bool pop() {

if (!topNode) return false; // Si la pila está vacía, no hay nada que liberar

StackNode\* toDelete = topNode; // Nodo a eliminar (el tope)

// Muestra información del bloque liberado

std::cout << "Bloque " << toDelete->data.blockId

<< " (proceso " << toDelete->data.processId

<< ", tamaño " << toDelete->data.size << ") liberado.\n";

topNode = topNode->next; // El nuevo tope es el siguiente nodo

delete toDelete; // Se libera el nodo anterior

return true; // Indica éxito

}

// Mostrar el estado actual de la pila de memoria

void displayStack() {

if (!topNode) { // Si la pila está vacía

std::cout << "La pila de memoria está vacía.\n";

return;

}

std::cout << "BlockID\tProcID\tTamaño\n"; // Encabezado

StackNode\* temp = topNode; // Nodo temporal para recorrer la pila

while (temp) { // Mientras haya nodos

std::cout << temp->data.blockId << "\t" // ID del bloque

<< temp->data.processId << "\t" // ID del proceso

<< temp->data.size << "\n"; // Tamaño del bloque

temp = temp->next; // Avanza al siguiente nodo

}

}

// Persistencia: guardar a archivo

void saveToFile(const std::string& filename) {

std::ofstream file(filename); // Abre archivo de salida

// Para guardar en orden de llegada, recolectamos en un arreglo temporal y luego invertimos

StackNode\* temp = topNode;

int count = 0;

while (temp) {

count++; // Contamos cuántos nodos hay

temp = temp->next;

}

if (count == 0) {

file.close(); // Si está vacía, solo cerramos el archivo

return;

}

MemoryBlock\* arr = new MemoryBlock[count]; // Arreglo temporal para almacenar bloques

temp = topNode;

for (int i = 0; i < count; i++) {

arr[i] = temp->data; // Copiamos cada nodo en el arreglo

temp = temp->next;

}

for (int i = count - 1; i >= 0; i--) {

file << arr[i].blockId << ","

<< arr[i].processId << ","

<< arr[i].size << "\n"; // Escribe desde el más antiguo al más reciente

}

delete[] arr; // Liberamos memoria del arreglo

file << nextBlockId << "\n"; // Guarda el siguiente ID disponible para bloques

file.close(); // Cierra el archivo

}

void loadFromFile(const std::string& filename) {

while (topNode) {

StackNode\* toDelete = topNode;

topNode = topNode->next;

delete toDelete;

}

std::ifstream file(filename);

if (!file.is\_open()) return; // Si el archivo no se abre, se cancela

std::string line;

MemoryBlock arr[1000]; // Arreglo temporal, límite 1000 bloques

int count = 0;

while (std::getline(file, line) && !line.empty()) {

std::stringstream ss(line);

MemoryBlock mb;

std::string token;

std::getline(ss, token, ',');

mb.blockId = std::stoi(token);

std::getline(ss, token, ',');

mb.processId = std::stoi(token);

std::getline(ss, token, ',');

mb.size = std::stoi(token);

arr[count++] = mb;

}

if (std::getline(file, line)) {

nextBlockId = std::stoi(line); // Recupera el valor del siguiente blockId

}

file.close();

for (int i = 0; i < count; i++) {

StackNode\* newNode = new StackNode(arr[i]);

newNode->next = topNode;

topNode = newNode;

}

}

void pause() {

std::cout << "Presione ENTER para continuar...";

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

}

void clearScreen() {

#ifdef \_WIN32

system("cls"); // Comando para limpiar pantalla en Windows

#else

system("clear"); // Comando para Linux/Mac

#endif

}

int main() {

ProcessList procList; // Instancia del gestor de procesos (lista enlazada)

PriorityQueue cpuQueue; // Instancia del planificador de CPU (cola de prioridad)

MemoryStack memStack; // Instancia del gestor de memoria (pila LIFO)

// Cargar datos guardados previamente desde archivos (persistencia)

procList.loadFromFile("processes.txt");

cpuQueue.loadFromFile("queue.txt");

memStack.loadFromFile("stack.txt");

int mainChoice; // Variable para la opción principal del menú

do {

clearScreen(); // Limpia la consola según el sistema operativo

std::cout << "==== SISTEMA DE GESTIÓN DE PROCESOS ====\n"; // Título principal

std::cout << "1. Gestor de Procesos\n"; // Opción 1: lista enlazada

std::cout << "2. Planificador de CPU (Cola de Prioridad)\n"; // Opción 2: cola

std::cout << "3. Gestor de Memoria (Pila)\n"; // Opción 3: pila

std::cout << "4. Salir y guardar\n"; // Opción 4: salir

std::cout << "Seleccione una opción: ";

std::cin >> mainChoice; // Leer opción elegida

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n'); // Limpiar buffer

// ====== MENÚ GESTOR DE PROCESOS ======

if (mainChoice == 1) {

int opt; // Subopción dentro del menú del gestor

do {

clearScreen();

std::cout << "-- Gestor de Procesos --\n";

std::cout << "1. Registrar nuevo proceso\n";

std::cout << "2. Eliminar proceso\n";

std::cout << "3. Buscar proceso\n";

std::cout << "4. Modificar prioridad\n";

std::cout << "5. Listar todos los procesos\n";

std::cout << "6. Volver al menú principal\n";

std::cout << "Seleccione: ";

std::cin >> opt;

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

// Subopción 1: registrar proceso nuevo

if (opt == 1) {

Process p; // Nueva estructura de proceso

std::cout << "Ingrese ID: ";

std::cin >> p.id;

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

std::cout << "Ingrese nombre: ";

std::getline(std::cin, p.name);

std::cout << "Ingrese prioridad (número entero, mayor = más prioridad): ";

std::cin >> p.priority;

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

std::cout << "Ingrese estado: ";

std::getline(std::cin, p.state);

procList.insertProcess(p); // Insertar en lista

std::cout << "Proceso registrado con éxito.\n";

pause(); // Esperar ENTER

}

// Subopción 2: eliminar proceso por ID

else if (opt == 2) {

int id;

std::cout << "Ingrese ID del proceso a eliminar: ";

std::cin >> id;

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

if (procList.removeProcess(id)) // Si se encontró

std::cout << "Proceso eliminado.\n";

else

std::cout << "Proceso no encontrado.\n";

pause();

}

// Subopción 3: búsqueda

else if (opt == 3) {

int subOpt;

std::cout << "Buscar por: 1) ID 2) Nombre: ";

std::cin >> subOpt;

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

if (subOpt == 1) {

int id;

std::cout << "Ingrese ID: ";

std::cin >> id;

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

ProcessNode\* node = procList.searchById(id); // Buscar por ID

if (node) {

// Mostrar datos

std::cout << "ID: " << node->data.id << "\n";

std::cout << "Nombre: " << node->data.name << "\n";

std::cout << "Prioridad: " << node->data.priority << "\n";

std::cout << "Estado: " << node->data.state << "\n";

} else {

std::cout << "Proceso no encontrado.\n";

}

} else if (subOpt == 2) {

std::string name;

std::cout << "Ingrese nombre: ";

std::getline(std::cin, name);

ProcessNode\* node = procList.searchByName(name); // Buscar por nombre

if (node) {

std::cout << "ID: " << node->data.id << "\n";

std::cout << "Nombre: " << node->data.name << "\n";

std::cout << "Prioridad: " << node->data.priority << "\n";

std::cout << "Estado: " << node->data.state << "\n";

} else {

std::cout << "Proceso no encontrado.\n";

}

} else {

std::cout << "Opción inválida.\n";

}

pause();

}

// Subopción 4: cambiar prioridad de un proceso

else if (opt == 4) {

int id, newPrio;

std::cout << "Ingrese ID del proceso: ";

std::cin >> id;

std::cout << "Ingrese nueva prioridad: ";

std::cin >> newPrio;

if (procList.modifyPriority(id, newPrio))

std::cout << "Prioridad actualizada.\n";

else

std::cout << "Proceso no encontrado.\n";

pause();

}

// Subopción 5: mostrar todos los procesos

else if (opt == 5) {

procList.displayAll(); // Mostrar lista completa

pause();

}

} while (opt != 6); // Salir del submenú con opción 6

}

// ====== MENÚ PLANIFICADOR DE CPU (COLA) ======

else if (mainChoice == 2) {

int opt;

do {

clearScreen();

std::cout << "-- Planificador de CPU (Cola de Prioridad) --\n";

std::cout << "1. Encolar proceso (según prioridad)\n";

std::cout << "2. Desencolar (Ejecutar)\n";

std::cout << "3. Ver cola actual\n";

std::cout << "4. Volver al menú principal\n";

std::cout << "Seleccione: ";

std::cin >> opt;

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

// Encolar proceso desde el gestor (por ID)

if (opt == 1) {

ProcessNode\* node;

int id;

std::cout << "Ingrese ID del proceso a encolar: ";

std::cin >> id;

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

node = procList.searchById(id); // Buscar en el gestor

if (node) {

cpuQueue.enqueue(node->data); // Insertar según prioridad

std::cout << "Proceso encolado.\n";

} else {

std::cout << "Proceso no encontrado en el gestor de procesos.\n";

}

pause();

}

// Desencolar (ejecutar proceso)

else if (opt == 2) {

Process executed;

if (cpuQueue.dequeue(executed)) {

// Mostrar el proceso ejecutado

std::cout << "Ejecutando proceso ID " << executed.id

<< " (" << executed.name << ").\n";

} else {

std::cout << "La cola está vacía.\n";

}

pause();

}

// Ver cola completa

else if (opt == 3) {

cpuQueue.displayQueue();

pause();

}

} while (opt != 4); // Salir del submenú con opción 4

}

// ====== MENÚ GESTOR DE MEMORIA (PILA) ======

else if (mainChoice == 3) {

int opt;

do {

clearScreen();

std::cout << "-- Gestor de Memoria (Pila) --\n";

std::cout << "1. Asignar bloque (Push)\n";

std::cout << "2. Liberar bloque (Pop)\n";

std::cout << "3. Ver pila de bloques\n";

std::cout << "4. Volver al menú principal\n";

std::cout << "Seleccione: ";

std::cin >> opt;

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

// Push: asignar bloque de memoria

if (opt == 1) {

int pid, size;

std::cout << "Ingrese ID de proceso que recibe memoria: ";

std::cin >> pid;

std::cout << "Ingrese tamaño del bloque: ";

std::cin >> size;

memStack.push(pid, size); // Apilar

pause();

}

// Pop: liberar bloque

else if (opt == 2) {

if (!memStack.pop()) {

std::cout << "La pila está vacía.\n";

}

pause();

}

// Mostrar pila completa

else if (opt == 3) {

memStack.displayStack();

pause();

}

} while (opt != 4); // Salir del submenú con opción 4

}

// ====== SALIDA Y GUARDADO FINAL ======

else if (mainChoice == 4) {

// Guardar todos los datos en archivos para persistencia

procList.saveToFile("processes.txt");

cpuQueue.saveToFile("queue.txt");

memStack.saveToFile("stack.txt");

std::cout << "Datos guardados. Saliendo...\n";

}

} while (mainChoice != 4); // Repetir el menú hasta que el usuario elija salir

return 0; // Fin del programa

}

1. **Capturas de pantalla de las ventanas de ejecución con las diversas pruebas de validación de datos**

****

1. **Manual de usuario**

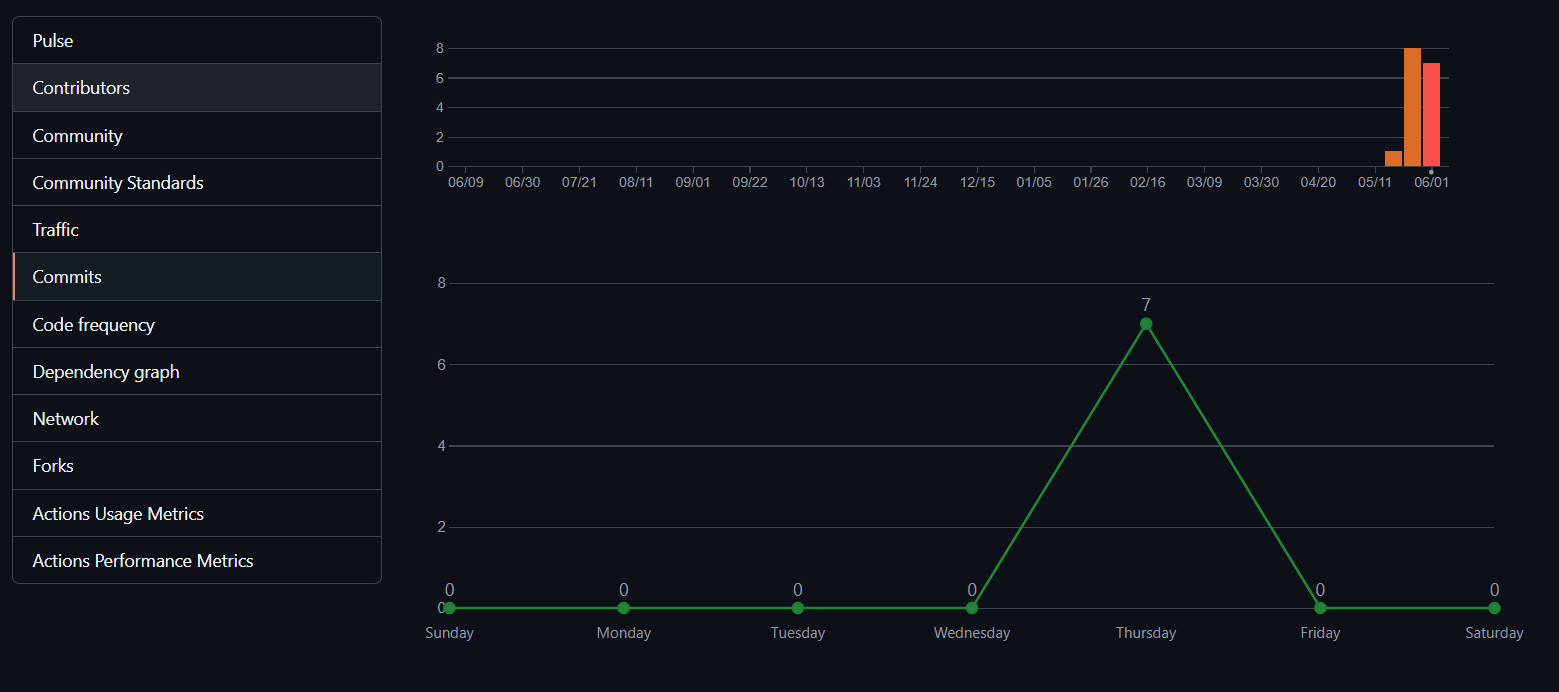
* Opción 1: Agrega un proceso y le asigna memoria.
* Opción 2: Mueve un proceso de la lista a la cola
* Opción 3: Atiende (elimina) al primer proceso de la cola.
* Opción 4: Muestra los procesos
* Opción 6: Lista todos los bloques de memoria activos.
* Opción 7: Libera la memoria del proceso más reciente.

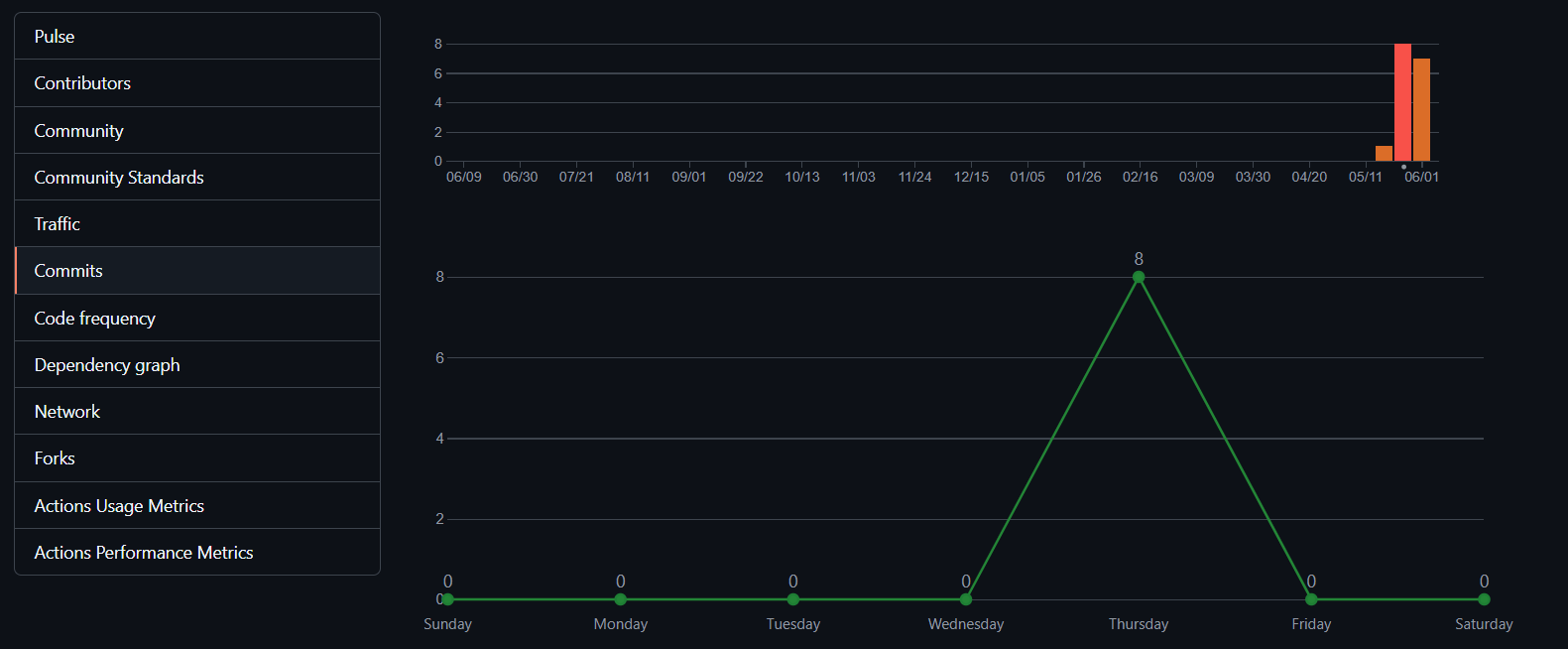
# CAPÍTULO 4: Evidencias de Trabajo en Equipo

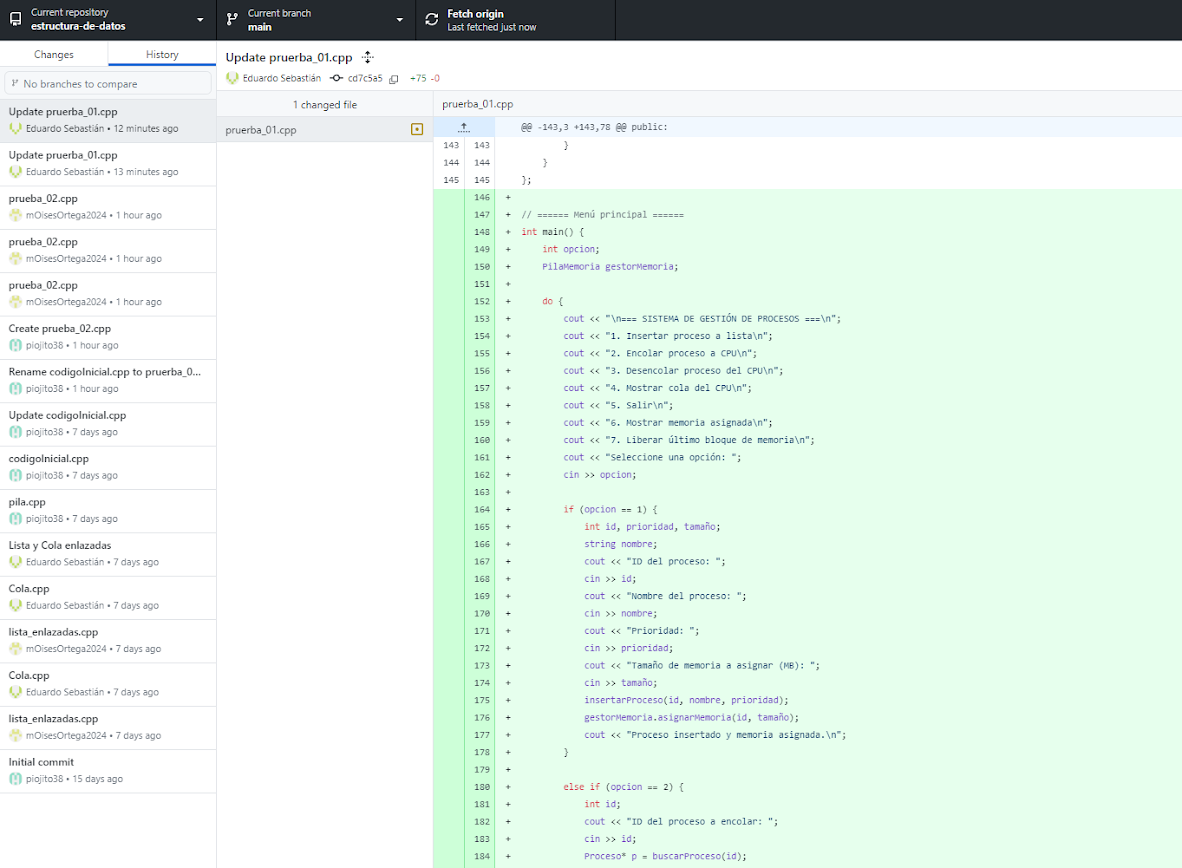
1. **Repositorio con Control de Versiones (Capturas de Pantalla)**

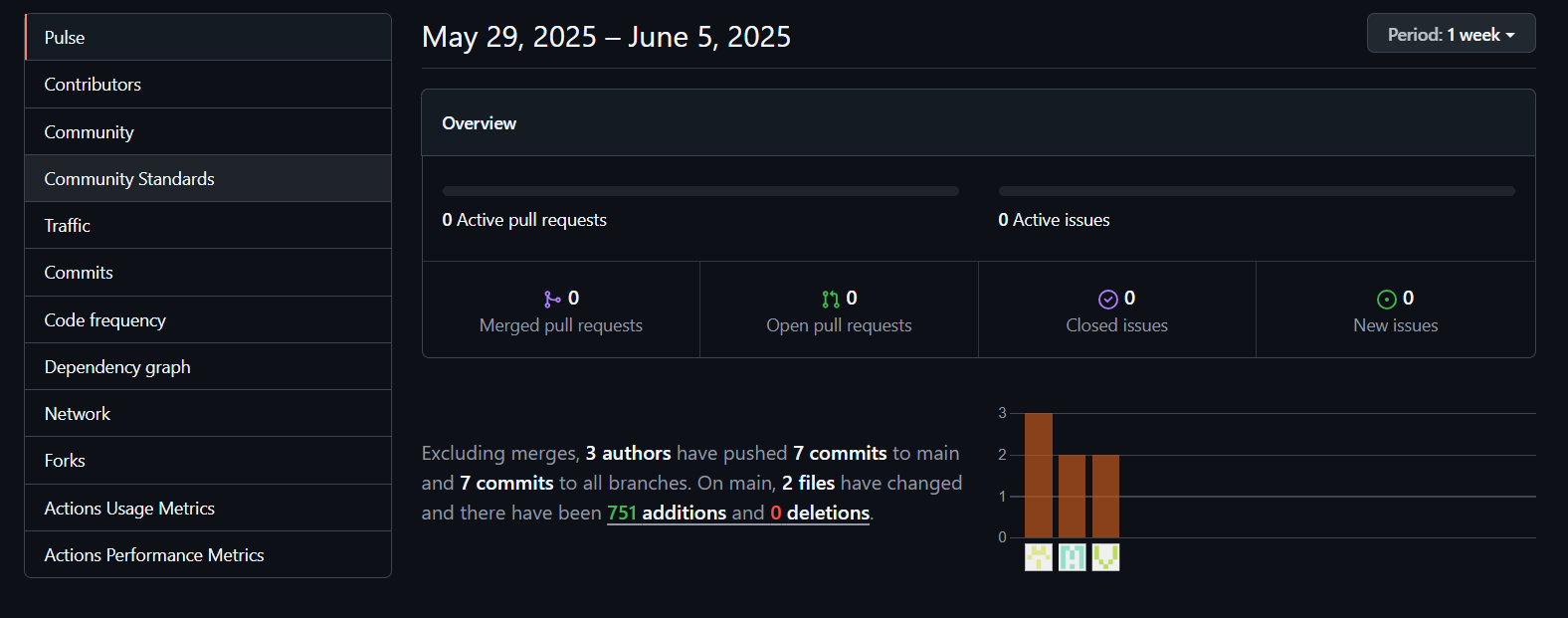
Comints:











Enlace a la herramienta colaborativa: <https://github.com/piojito38/estructura-de-datos.git>

Drive:

[estructura de datos](https://drive.google.com/drive/folders/19UYOdr8ws10g5XR7-JauFeQ5vw1NEk-x)

PPT:

<https://www.canva.com/design/DAGpitVGE08/nCKVxUAZdLpPPvHK__961w/edit?utm_content=DAGpitVGE08&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton>

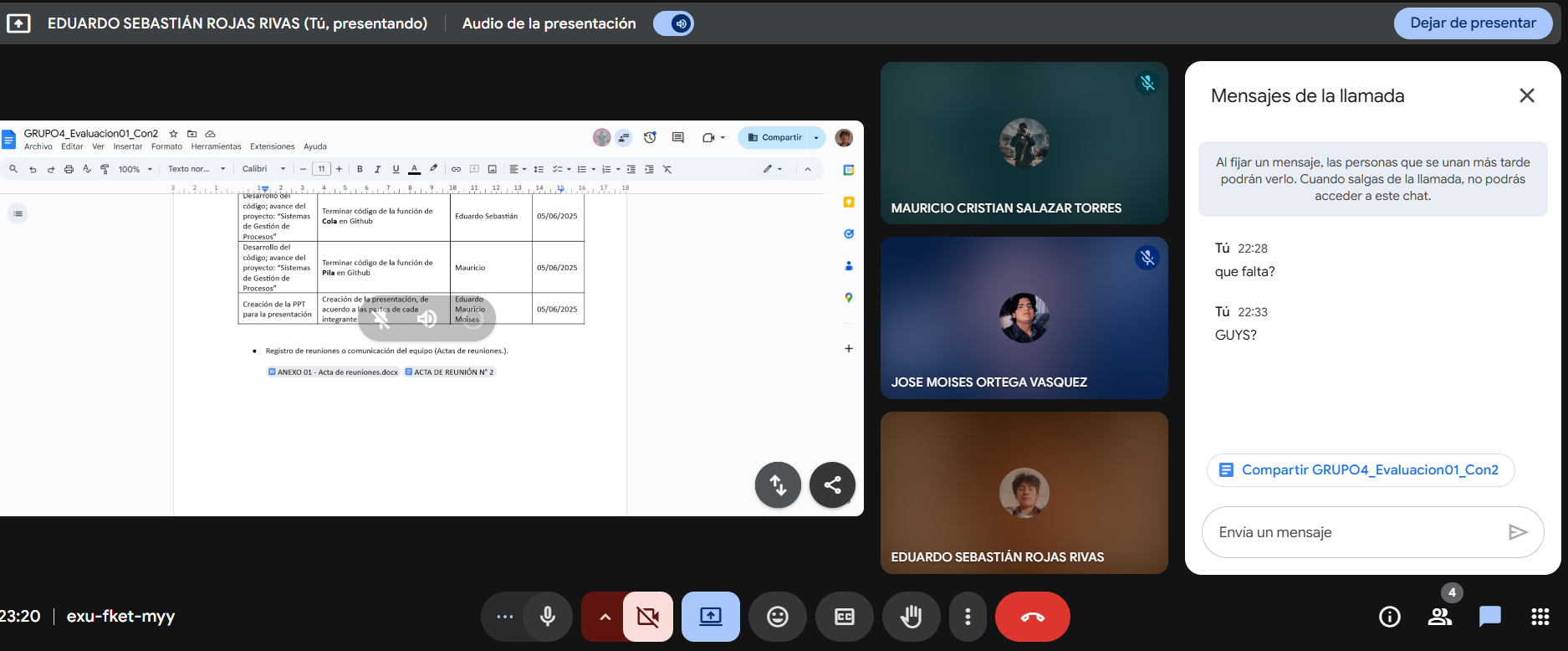
1. **Plan de Trabajo y Roles Asignados**

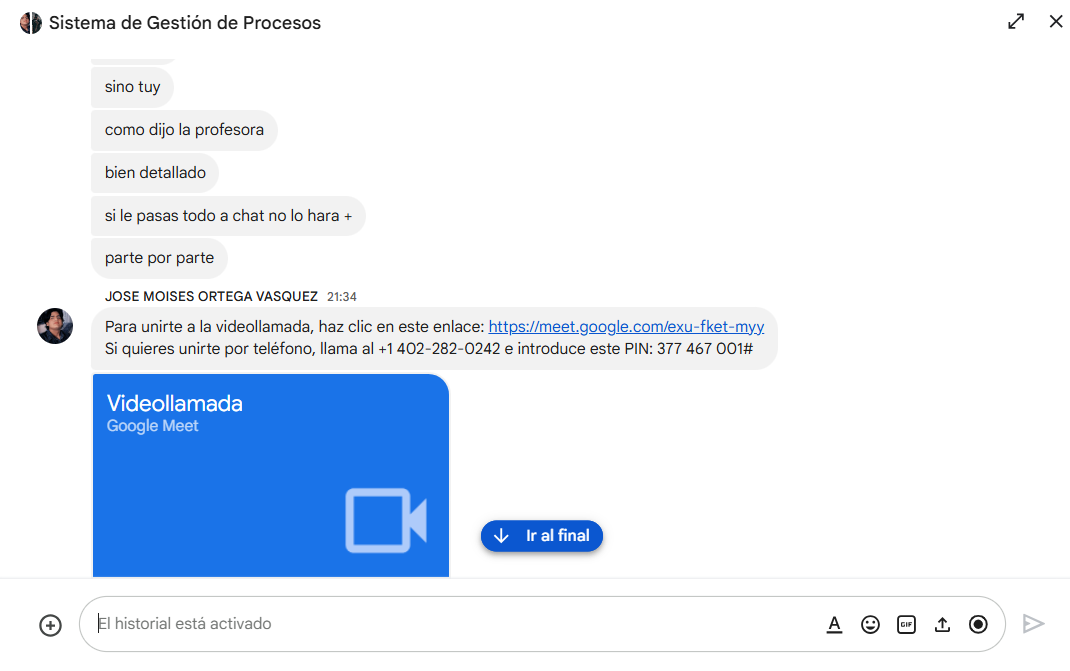
* **Integrantes:**

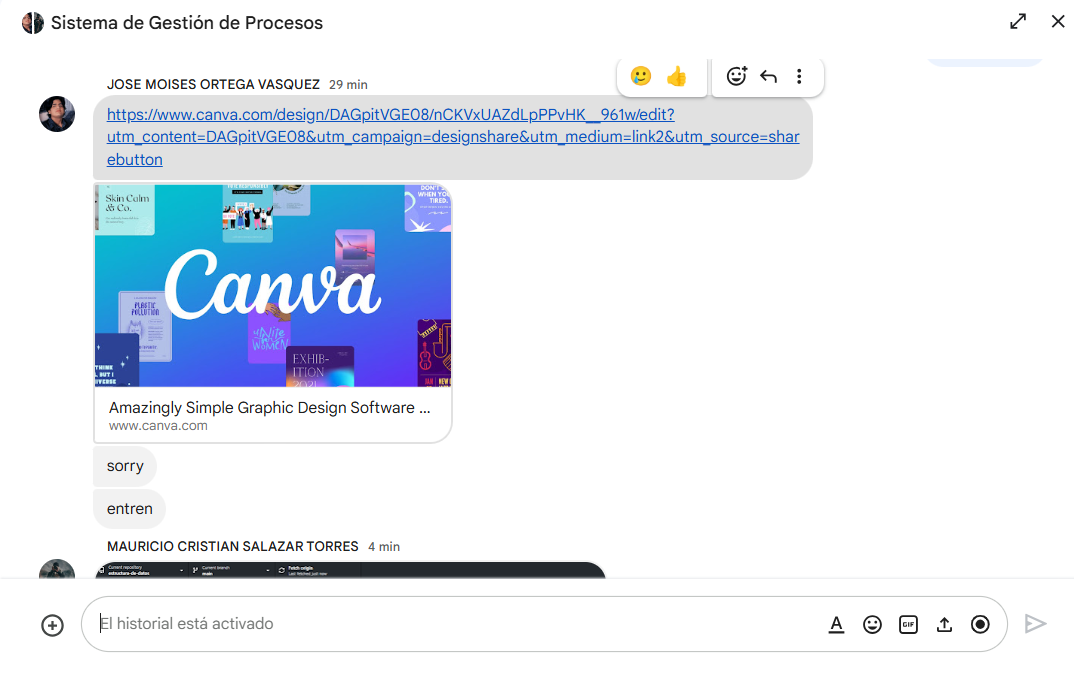
| **Apellidos y nombres** | **Asistió (Si/No)** | **% Participación** | **Firma** |
| --- | --- | --- | --- |
| 1. Rojas Rivas, Eduardo Sebastián | **si** | **100%** |  |
| 1. Salazar Torres, Mauricio | **si** | **100%** |  |
| 1. Ortega Vásquez, José Moisés | **si** | **100%** |  |
| 1. Marcelo Sanchez, Steven Smith | **no** | **0%** | - |

| **Temas tratados** | **Acuerdos** | **Responsables** | **Fecha de entrega** |
| --- | --- | --- | --- |
| Desarrollamos el avance del proyecto: “Sistemas de Gestión de Procesos” | Terminar las dimensiones de : **Lista** , trabajando en Github | Moises Ortega | 05/06/2025 |
| Desarrollo del código; avance del proyecto: “Sistemas de Gestión de Procesos” | Terminar código de la función de **Cola** en Github | Eduardo Sebastián | 05/06/2025 |
| Desarrollo del código; avance del proyecto: “Sistemas de Gestión de Procesos” | Terminar código de la función de **Pila** en Github | Mauricio | 05/06/2025 |
| Creación de la PPT para la presentación | Creación de la presentación, de acuerdo a las partes de cada integrante | Eduardo  Mauricio  Moises | 05/06/2025 |

* Registro de reuniones o comunicación del equipo (Actas de reuniones.).







[ANEXO 01 - Acta de reuniones.docx](https://docs.google.com/document/d/1d0Qs7OkypBPMmOlqz_ft0nP-TCASJxVO/edit?usp=sharing&ouid=107372243622481147885&rtpof=true&sd=true) [ACTA DE REUNIÓN N° 2](https://docs.google.com/document/d/1331YeYHP0Nnzkgsd0Fw3bROgrGnW2qdxXIdYzBbzzrM/edit?usp=sharing)