

Asignatura:

**ESTRUCTURA DE DATOS**

**NRC: 29902**

Proyecto Grupal:

**Sistema de control de tareas y gestión de software**

INTEGRANTES:

* Collazos Gracía Jose
* Escobar Ruiz Kevin
* Solis Hinostroza Cynthia Rossy
* Oroya Molina Luis Fernando

**PERÚ**

**2025**

CAPÍTULO 1: Análisis del problema

**1. Descripción del problema**

Un sistema de gestión de tareas y control de software en un sistema operativo utiliza estructuras dinámicas para manejar eficientemente los procesos y recursos: una lista enlazada registra y mantiene actualizados todos los procesos con su estado y datos, permitiendo inserciones y eliminaciones dinámicas; una cola gestiona la planificación de la CPU, encolando procesos listos para ejecución y desencolado según políticas como FIFO o prioridades, asegurando un acceso justo y ordenado a la CPU; y una pila se emplea para la gestión de memoria, asignando y liberando bloques en orden LIFO para evitar fugas y mantener el control sobre el uso de recursos temporales. Este enfoque integrado permite monitorear, controlar y optimizar la ejecución de software, garantizando que los procesos se registren, planifiquen y liberen correctamente, facilitando la eficiencia y estabilidad del sistema operativo.

El problema plantea la simulación de la gestión de procesos y recursos en un sistema operativo mediante tres estructuras dinámicas clave: lista enlazada, cola y pila, cada una con funciones específicas para optimizar el control y uso de la CPU y la memoria.

El desafío principal radica en mantener la sincronía y consistencia entre estas estructuras: los cambios en el estado de los procesos en la lista deben reflejarse inmediatamente en la cola de planificación, y las operaciones de memoria deben garantizar que no haya pérdidas. Además, el sistema debe ser capaz de manejar dinámicamente inserciones, eliminaciones y cambios de estado, simulando un entorno realista de gestión de procesos y memoria.

**2. Requerimientos del sistema**

* **Requerimientos Funcionales:**
* Registrar, modificar y eliminar procesos en una lista enlazada.
* Planificar la ejecución de procesos mediante una cola FIFO.
* Gestionar la memoria temporal de procesos utilizando una pila (LIFO).
* Simular el cambio de estado de los procesos: nuevo, listo, ejecutando, bloqueado, terminado.
* Sincronizar automáticamente los cambios entre las estructuras.
* **Requerimientos No Funcionales:**
* El sistema debe ser interactivo y fácil de usar.
* Debe manejar hasta 100 procesos y operaciones sin pérdida de datos.
* Código claro, estructurado y con comentarios.
* Ejecución eficiente, sin retardos perceptibles.

**3. Estructuras de datos propuestas**

1. **Lista enlazada de procesos:** Esta estructura permite registrar todos los procesos activos con sus atributos (ID, estado y prioridad). La ventaja principal es la flexibilidad para insertar y eliminar procesos en cualquier posición, lo cual es esencial para reflejar cambios en la ejecución, como creación o finalización de procesos. Además, facilita la actualización constante del estado de cada proceso, un requisito básico en la gestión real de sistemas operativos.
2. **Cola FIFO para planificación CPU:** La cola garantiza un acceso justo y ordenado a la CPU, siguiendo la política FIFO, que es simple y efectiva para muchos sistemas. Los procesos en estado “listo” se encolan según su llegada y se desencolan para asignar CPU. Esto simula cómo el sistema operativo programa procesos, evitando favoritismos y asegurando que todos reciban tiempo de CPU en orden. La cola también debe actualizarse eliminando procesos que terminan o se bloquean, manteniendo así un flujo coherente.
3. **Pila para gestión de memoria:** La pila simula la asignación y liberación de bloques de memoria en orden LIFO, reflejando cómo los procesos pueden requerir memoria temporal que debe liberarse en orden inverso a su asignación para evitar fugas. Esta estructura permite controlar el uso eficiente de recursos, asegurando que la memoria se libere correctamente cuando un proceso termina o libera bloques.
4. **Justificación de la elección:**

Las estructuras propuestas permiten simular de manera realista el funcionamiento de un sistema operativo. La lista enlazada otorga flexibilidad para agregar y eliminar procesos en cualquier posición. La cola FIFO asegura una planificación justa de la CPU según orden de llegada. La pila refleja adecuadamente la gestión de memoria temporal, donde los últimos recursos asignados deben ser los primeros en liberarse. Estas estructuras son las más adecuadas debido a su naturaleza dinámica y eficiencia operativa.

# 

# Capítulo 2: Diseño de la Solución

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

El sistema se basa en tres estructuras dinámicas fundamentales: lista enlazada, cola FIFO y pila LIFO, cada una seleccionada por su idoneidad para simular una parte específica del ciclo de vida de los procesos en un sistema operativo. A continuación, se describen en detalle estas estructuras y las operaciones implementadas sobre ellas:

**1.1 Lista Enlazada: Gestor de Procesos**

La lista enlazada es utilizada como base para representar todos los procesos activos del sistema. Cada nodo de la lista representa un proceso y contiene al menos los siguientes atributos:

* ID: identificador único del proceso.
* Estado: puede ser “Nuevo”, “Listo”, “Ejecutando”, “Bloqueado” o “Terminado”.
* Prioridad: nivel que puede influir en una futura planificación más avanzada.
* Enlace: puntero al siguiente nodo de la lista.

**Operaciones asociadas a la lista:**

* InsertarProceso(ID, estado, prioridad): Inserta un nuevo proceso al final de la lista.
* EliminarProceso(ID): Busca y elimina un nodo correspondiente a un proceso terminado.
* ActualizarEstado(ID, nuevoEstado): Cambia el estado de un proceso existente y sincroniza con otras estructuras si es necesario.
* BuscarProceso(ID): Recorre la lista para localizar un proceso específico.
* MostrarLista(): Visualiza todos los procesos activos con sus atributos.

Esta estructura permite modificar dinámicamente la cantidad de procesos en el sistema, simular la creación y terminación de procesos, y mantener una vista organizada del sistema.

**1.2 Cola FIFO: Planificación de la CPU**

La cola implementa el mecanismo de planificación de procesos. Es decir, controla el orden en el que los procesos “listos” acceden a la CPU. La política de planificación aplicada es FIFO (First In, First Out), lo que significa que el primer proceso en llegar será el primero en ejecutarse.

**Operaciones asociadas a la cola:**

* Encolar(ID): Inserta un proceso en estado “Listo” al final de la cola, esperando ejecución.
* Desencolar(): Retira el proceso del frente de la cola para asignarle CPU.
* VerCola(): Muestra los procesos actualmente en cola, en orden de espera.
* ActualizarCola(): Elimina procesos que ya no estén en estado “Listo”, asegurando consistencia.

La cola reproduce el comportamiento del planificador del sistema operativo, asegurando orden y equidad en el acceso a la CPU. Puede ampliarse para simular prioridades o técnicas más avanzadas como Round Robin.

**1.3 Pila: Gestión de Memoria Temporal**

La pila simula el manejo de memoria dinámica, en especial memoria de tipo stack utilizada para variables locales, llamadas a funciones y contexto de ejecución. La naturaleza LIFO (Last In, First Out) garantiza que el último bloque asignado sea el primero en liberarse, tal como ocurre en los entornos reales.

**Operaciones asociadas a la pila:**

* Apilar(ID, tamaño): Reserva un bloque de memoria para un proceso activo.
* Desapilar(): Libera el último bloque asignado.
* DesapilarMemoria(ID): Elimina bloques asignados a un proceso cuando este finaliza.
* VerPila(): Muestra el estado actual de la memoria temporal.

Esta pila permite modelar el uso eficiente de memoria, evitando pérdidas (fugas) y replicando cómo se limpian automáticamente los recursos al finalizar una función o proceso.

# **Algoritmos principales**

# **Pseudocódigo - Lista Enlazada (Gestor de Procesos)**

# **Cola Basica:**

// =====================

// ESTRUCTURA: COLA

// =====================

Proceso ColaBasica

Dimension cola[100]

Definir frente, ultimo, elemento Como Entero

frente <- 1

ultimo <- 0

// Encolar (Agregar al final)

Escribir "Encolar elemento"

Leer elemento

Si ultimo < 100 Entonces

ultimo <- ultimo + 1

cola[ultimo] <- elemento

Escribir "Elemento encolado: ", elemento

Sino

Escribir "La cola está llena"

Finsi

// Desencolar (Eliminar del frente)

Si frente <= ultimo Entonces

Escribir "Elemento desencolado: ", cola[frente]

frente <- frente + 1

Sino

Escribir "La cola está vacía"

FinSi

// Mostrar cola (del frente al último)

Si frente <= ultimo Entonces

Escribir "Elementos en la cola:"

Para i <- frente Hasta ultimo

Escribir cola[i]

FinPara

Sino

Escribir "La cola está vacía"

FinSi

FinProceso

# 

# **Pila Basica:**

// =====================

// ESTRUCTURA: PILA

// =====================

Algoritmo PilaBasica

Dimensionar pila(100)

Definir tope, elemento Como Entero

tope <- 0

// Apilar (Push)

Escribir "Apilar elemento"

Leer elemento

Si tope < 100 Entonces

tope <- tope + 1

pila[tope] <- elemento

Escribir "Elemento apilado: ", elemento

Sino

Escribir "La pila está llena"

FinSi

// Desapilar (Pop)

Si tope > 0 Entonces

Escribir "Elemento desapilado: ", pila[tope]

tope <- tope - 1

Sino

Escribir "La pila está vacía"

FinSi

// Mostrar pila (de arriba hacia abajo)

Si tope > 0 Entonces

Escribir "Elementos en la pila (de arriba hacia abajo):"

Para i <- tope Hasta 1 Con Paso -1 Hacer

Escribir pila[i]

FinPara

Sino

Escribir "La pila está vacía"

FinSi

FinAlgoritmo

# **Lista Basica:**

// =====================

// ESTRUCTURA: LISTA

// =====================

Proceso ListaBasica

Dimension lista[100]

Definir tam, i, pos, elemento Como Entero

tam <- 0

// Insertar al final

Escribir "Insertar elemento al final"

Leer elemento

Si tam < 100 Entonces

tam <- tam + 1

lista[tam] <- elemento

Escribir "Elemento insertado: ", elemento

Sino

Escribir "La lista está llena, no se puede insertar"

FinSi

// Eliminar en una posición

Escribir "Eliminar elemento en posición"

Leer pos

Si pos >= 1 Y pos <= tam Entonces

Para i <- pos Hasta tam - 1 Con Paso 1

lista[i] <- lista[i + 1]

FinPara

tam <- tam - 1

Escribir "Elemento en posición ", pos, " eliminado"

Sino

Escribir "Posición inválida"

FinSi

// Mostrar lista

Si tam > 0 Entonces

Escribir "Elementos en la lista:"

Para i <- 1 Hasta tam

Escribir lista[i]

FinPara

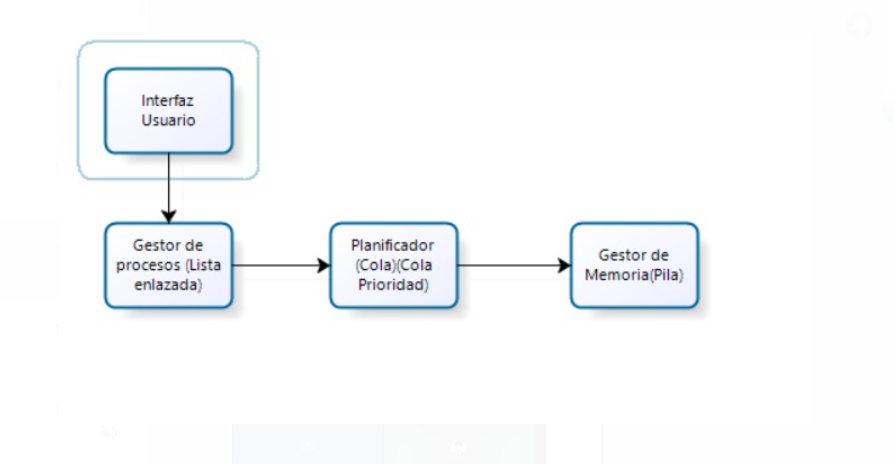
Sino

Escribir "La lista está vacía"

FinSi

FinProceso

# **3. Diagrama de flujo**

Diagrama lógico del sistema:

# **4. Justificación de diseño**

El diseño del sistema está basado en tres estructuras principales: una lista, una cola y una pila. Cada una cumple una función específica y se eligieron porque trabajan bien juntas y permiten simular de forma clara cómo un sistema operativo maneja sus procesos y recursos.

* La lista se usa para llevar un control de todos los procesos activos. Es útil porque permite agregar, eliminar o modificar información de los procesos de manera flexible. Así podemos actualizar su estado fácilmente cuando cambian, por ejemplo, de “listo” a “ejecutando”.
* La cola se encarga de organizar los procesos que están esperando usar la CPU. Los acomoda en el orden en que llegan, lo que asegura que todos tengan su turno. Esta forma de organización es simple pero efectiva para distribuir el uso del procesador de forma justa.
* La pila se utiliza para simular la memoria que los procesos usan de manera temporal. Esta memoria debe liberarse en el orden inverso al que se reservó, y la pila permite hacerlo de forma ordenada, evitando errores o pérdidas de memoria.

El uso de estas estructuras nos da varias ventajas:

* El sistema es fácil de entender porque cada parte cumple una función clara.
* Es rápido y eficiente, porque las operaciones que se realizan (agregar, quitar, buscar) se hacen de forma directa y sin complicaciones.
* Además, es fácil de modificar o mejorar, por si más adelante se quiere agregar más funciones.

En resumen, este diseño nos permite construir un sistema ordenado, funcional y sencillo, que cumple con lo necesario para simular cómo se controlan los procesos y recursos en un sistema real.

# Capítulo 3: Diseño de la Solución

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

#include <iostream> // Incluye la biblioteca para entrada/salida estándar (como cout y cin)

using namespace std; // Permite usar los objetos de la biblioteca estándar sin anteponer 'std::'

// Definición de constantes y variables globales

const int MEMORIA\_TOTAL = 5; // Define la memoria total del sistema (cantidad máxima de procesos)

int memoriaDisponible = MEMORIA\_TOTAL; // Memoria actualmente disponible

int procesosCreados = 0; // Contador de procesos creados

int siguienteID = 1; // ID incremental para nuevos procesos

// Estructura que representa un proceso

struct Proceso {

int id; // Identificador único del proceso

string estado; // Estado del proceso (listo, ejecutando, bloqueado)

string prioridad; // Prioridad del proceso (Alta, Media, Baja)

int memoria[100]; // Pila de memoria utilizada por el proceso

int topeMemoria; // Índice del tope de la pila de memoria

Proceso\* siguiente; // Puntero al siguiente proceso en la lista enlazada

};

// Puntero al primer proceso de la lista enlazada

Proceso\* cabeza = NULL;

// Función que convierte una opción numérica en un estado de proceso

string obtenerEstado(int opcion) {

switch (opcion) {

case 1: return "listo";

case 2: return "ejecutando";

case 3: return "bloqueado";

default: return "invalido";

}

}

// Función que convierte una opción numérica en una prioridad

string obtenerPrioridad(int opcion) {

switch (opcion) {

case 1: return "Alta";

case 2: return "Media";

case 3: return "Baja";

default: return "invalida";

}

}

// Verifica si un proceso con un ID dado ya existe

bool idExiste(int id) {

Proceso\* aux = cabeza; // Comienza desde el inicio de la lista

while (aux != NULL) { // Recorre la lista

if (aux->id == id)

return true; // Si encuentra el ID, retorna true

aux = aux->siguiente; // Pasa al siguiente proceso

}

return false; // Si no lo encuentra, retorna false

}

// Crea un nuevo proceso con estado y prioridad dados

void crearProceso(string estado, string prioridad) {

if (estado == "invalido" || prioridad == "invalida") {

cout << "Estado o prioridad inválida.\n";

return;

}

Proceso\* nuevo = new Proceso; // Reserva memoria para un nuevo proceso

nuevo->id = siguienteID++; // Asigna un ID único y lo incrementa

nuevo->estado = estado;

nuevo->prioridad = prioridad;

nuevo->topeMemoria = -1; // Inicializa la pila como vacía

nuevo->siguiente = NULL;

// Inserta el proceso al final de la lista enlazada

if (cabeza == NULL) {

cabeza = nuevo;

} else {

Proceso\* aux = cabeza;

while (aux->siguiente != NULL)

aux = aux->siguiente;

aux->siguiente = nuevo;

}

procesosCreados++; // Incrementa el contador de procesos

cout << "Proceso creado con ID: " << nuevo->id << ", Estado: " << estado << ", Prioridad: " << prioridad << endl;

// Mensajes de advertencia sobre uso de memoria

if (procesosCreados == 4)

cout << " Memoria casi llena (por cantidad de procesos).\n";

else if (procesosCreados == 5)

cout << " Memoria llena (por cantidad de procesos).\n";

}

// Muestra todos los procesos registrados y su información

void mostrarProcesos() {

Proceso\* aux = cabeza;

cout << "\n--- Procesos registrados ---\n";

while (aux != NULL) {

cout << "ID: " << aux->id

<< ", Estado: " << aux->estado

<< ", Prioridad: " << aux->prioridad

<< ", Memoria usada (pila): " << aux->topeMemoria + 1;

if (aux->topeMemoria + 1 >= 90)

cout << " (¡Memoria casi llena!)";

cout << endl;

aux = aux->siguiente;

}

cout << "Memoria disponible del sistema: " << memoriaDisponible << " unidades\n";

if (memoriaDisponible == 0)

cout << " Memoria del sistema llena.\n";

else if (memoriaDisponible <= 1)

cout << " Advertencia: Memoria del sistema casi llena.\n";

}

// Cambia el estado de un proceso dado su ID

void cambiarEstado(int id, string nuevoEstado) {

if (nuevoEstado == "invalido") {

cout << "Estado inválido.\n";

return;

}

Proceso\* aux = cabeza;

while (aux != NULL) {

if (aux->id == id) {

aux->estado = nuevoEstado;

cout << "Estado del proceso " << id << " cambiado a " << nuevoEstado << endl;

return;

}

aux = aux->siguiente;

}

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

}

// Asigna (push) memoria al proceso en forma de pila

void usarMemoria(int id, int cantidad) {

Proceso\* aux = cabeza;

while (aux != NULL) {

if (aux->id == id) {

if (aux->topeMemoria + cantidad >= 100) {

cout << "Memoria del proceso llena (máximo 100).\n";

return;

}

if (memoriaDisponible < cantidad) {

cout << "Error: No hay suficiente memoria del sistema.\n";

return;

}

for (int i = 0; i < cantidad; i++)

aux->memoria[++aux->topeMemoria] = 1; // Apila memoria

memoriaDisponible -= cantidad;

cout << "Memoria apilada al proceso " << id << ": " << cantidad << " unidades.\n";

return;

}

aux = aux->siguiente;

}

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

}

// Libera (pop) memoria del proceso

void quitarMemoria(int id, int cantidad) {

Proceso\* aux = cabeza;

while (aux != NULL) {

if (aux->id == id) {

if (aux->topeMemoria < cantidad - 1) {

cout << "No hay suficiente memoria en la pila del proceso.\n";

return;

}

for (int i = 0; i < cantidad; i++)

aux->topeMemoria--; // Quita memoria

memoriaDisponible += cantidad;

cout << "Memoria desapilada del proceso " << id << ": " << cantidad << " unidades.\n";

return;

}

aux = aux->siguiente;

}

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

}

// Elimina completamente un proceso y libera su memoria

void eliminarProceso(int id) {

Proceso\* actual = cabeza;

Proceso\* anterior = NULL;

while (actual != NULL) {

if (actual->id == id) {

int memoriaLiberada = actual->topeMemoria + 1; // Memoria usada por el proceso

memoriaDisponible += memoriaLiberada;

if (anterior == NULL) {

cabeza = actual->siguiente;

} else {

anterior->siguiente = actual->siguiente;

}

delete actual;

cout << "Proceso " << id << " eliminado. Memoria liberada: " << memoriaLiberada << " unidades.\n";

return;

}

anterior = actual;

actual = actual->siguiente;

}

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

}

// Permite eliminar procesos de uno en uno desde el inicio de la lista

void liberarEspacioMemoria() {

while (cabeza != NULL) {

mostrarProcesos();

cout << "\n¿Desea eliminar el primer proceso? (s/n): ";

char opcion;

cin >> opcion;

if (opcion == 's' || opcion == 'S') {

int idEliminar = cabeza->id;

eliminarProceso(idEliminar);

} else {

cout << "Proceso no eliminado. Fin de la limpieza.\n";

break;

}

}

if (cabeza == NULL) {

cout << "Todos los procesos han sido eliminados.\n";

}

}

// Menú de opciones principal del sistema

void menu() {

int opcion;

do {

cout << "\n===== MENÚ DE GESTIÓN DE PROCESOS =====\n";

cout << "1. Crear proceso\n";

cout << "2. Cambiar estado de proceso\n";

cout << "3. Agregar memoria a proceso (Push)\n";

cout << "4. Mostrar todos los procesos\n";

cout << "5. Salir\n";

cout << "6. Eliminar proceso completamente\n";

cout << "7. Liberar espacio de memoria\n";

cout << "8. Quitar memoria a proceso (Pop)\n";

cout << "Seleccione una opción: ";

cin >> opcion;

int id, cantidad, estadoNum, prioridadNum;

string estado, prioridad;

switch (opcion) {

case 1:

// Solicita estado y prioridad, valida y crea proceso

while (true) {

cout << "Estado:\n 1. listo\n 2. ejecutando\n 3. bloqueado\nSeleccione: ";

cin >> estadoNum;

estado = obtenerEstado(estadoNum);

if (estado == "invalido") {

cout << "ERROR, la opción que marcó no existe.\n";

} else {

break;

}

}

while (true) {

cout << "Prioridad:\n 1. Alta\n 2. Media\n 3. Baja\nSeleccione: ";

cin >> prioridadNum;

prioridad = obtenerPrioridad(prioridadNum);

if (prioridad == "invalida") {

cout << "ERROR, la opción que marcó no existe.\n";

} else {

break;

}

}

crearProceso(estado, prioridad);

break;

case 2:

// Cambia el estado de un proceso

cout << "ID del proceso: ";

cin >> id;

if (!idExiste(id)) {

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

break;

}

while (true) {

cout << "Nuevo estado:\n 1. listo\n 2. ejecutando\n 3. bloqueado\nSeleccione: ";

cin >> estadoNum;

estado = obtenerEstado(estadoNum);

if (estado == "invalido") {

cout << "ERROR, la opción que marcó no existe.\n";

} else {

break;

}

}

cambiarEstado(id, estado);

break;

case 3:

// Apila memoria

cout << "ID del proceso: ";

cin >> id;

if (!idExiste(id)) {

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

break;

}

cout << "Cantidad de memoria a apilar: ";

cin >> cantidad;

usarMemoria(id, cantidad);

break;

case 4:

// Muestra los procesos

mostrarProcesos();

break;

case 5:

// Sale del programa

cout << "Saliendo del sistema...\n";

break;

case 6:

// Elimina un proceso

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

cin >> id;

eliminarProceso(id);

break;

case 7:

// Limpieza de memoria (elimina desde el principio)

liberarEspacioMemoria();

break;

case 8:

// Desapila memoria

cout << "ID del proceso: ";

cin >> id;

if (!idExiste(id)) {

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

break;

}

cout << "Cantidad de memoria a desapilar: ";

cin >> cantidad;

quitarMemoria(id, cantidad);

break;

default:

cout << "Opción inválida.\n";

}

} while (opcion != 5); // Repite hasta que se seleccione la opción 5 (salir)

}

// Función principal

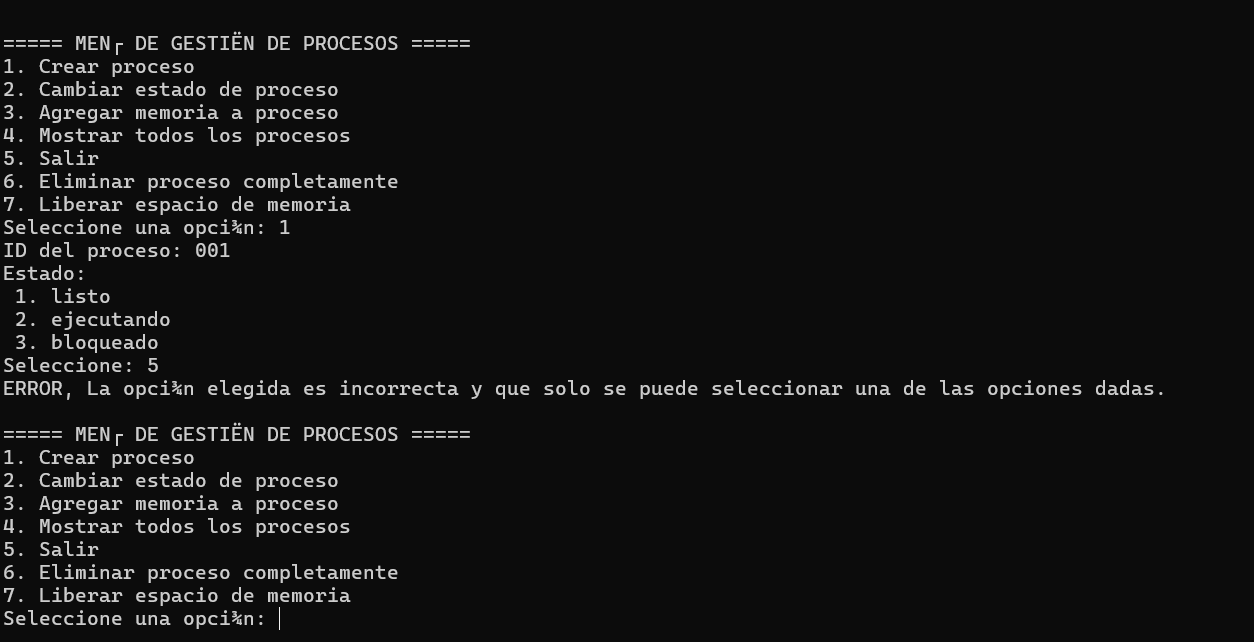
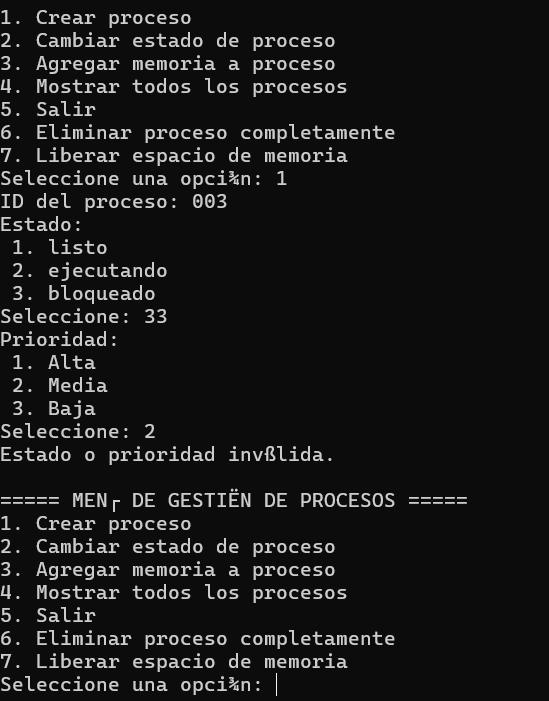
int main() {

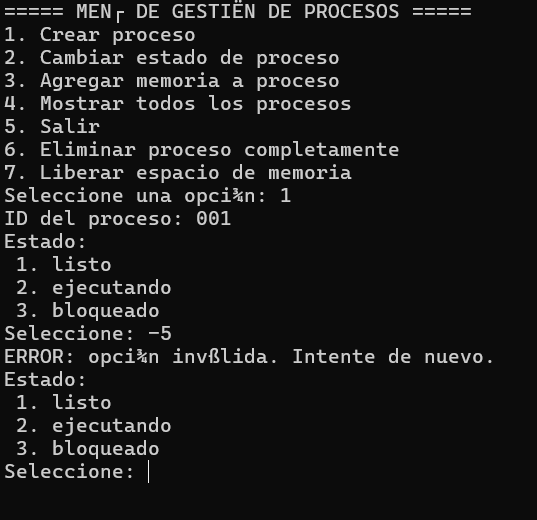
menu(); // Ejecuta el menú de gestión de procesos

return 0;

}

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

****

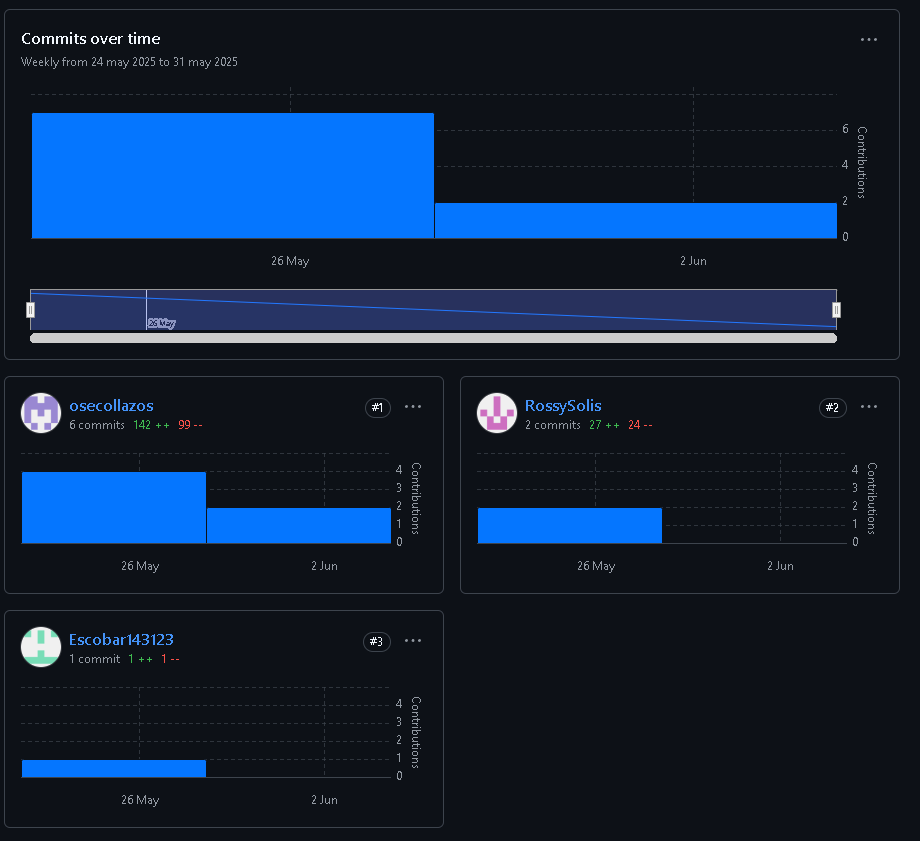
****

**3. Manual de usuario**

[Manual\_Gestion\_Procesos\_C++.docx](https://docs.google.com/document/d/16Aoegq-Pt5hEdH-w19wUO4KFK9kF9t5s/edit?usp=drive_link&ouid=116659874449891242141&rtpof=true&sd=true)

# Capítulo 4: Evidencias de Trabajo en Equipo

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

****

1. **Plan de trabajo y roles asignados**

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

[**\_ANEXO 01 - Acta de reuniones (1).docx**](https://docs.google.com/document/d/1wxxH6ZZL9Z1NhqRh9CTY4YD3Xz0o90LI/edit)

[**ANEXO 02 - Acta de reuniones (2)**](https://docs.google.com/document/d/1kq9C9axBX8f8llb5zk4MSOzAof2mBhbwQnWCfo8Onb8/edit?usp=drive_link)

[**ANEXO 02 - Acta de reuniones (2)**](https://docs.google.com/document/d/1kq9C9axBX8f8llb5zk4MSOzAof2mBhbwQnWCfo8Onb8/edit?usp=drive_link)

