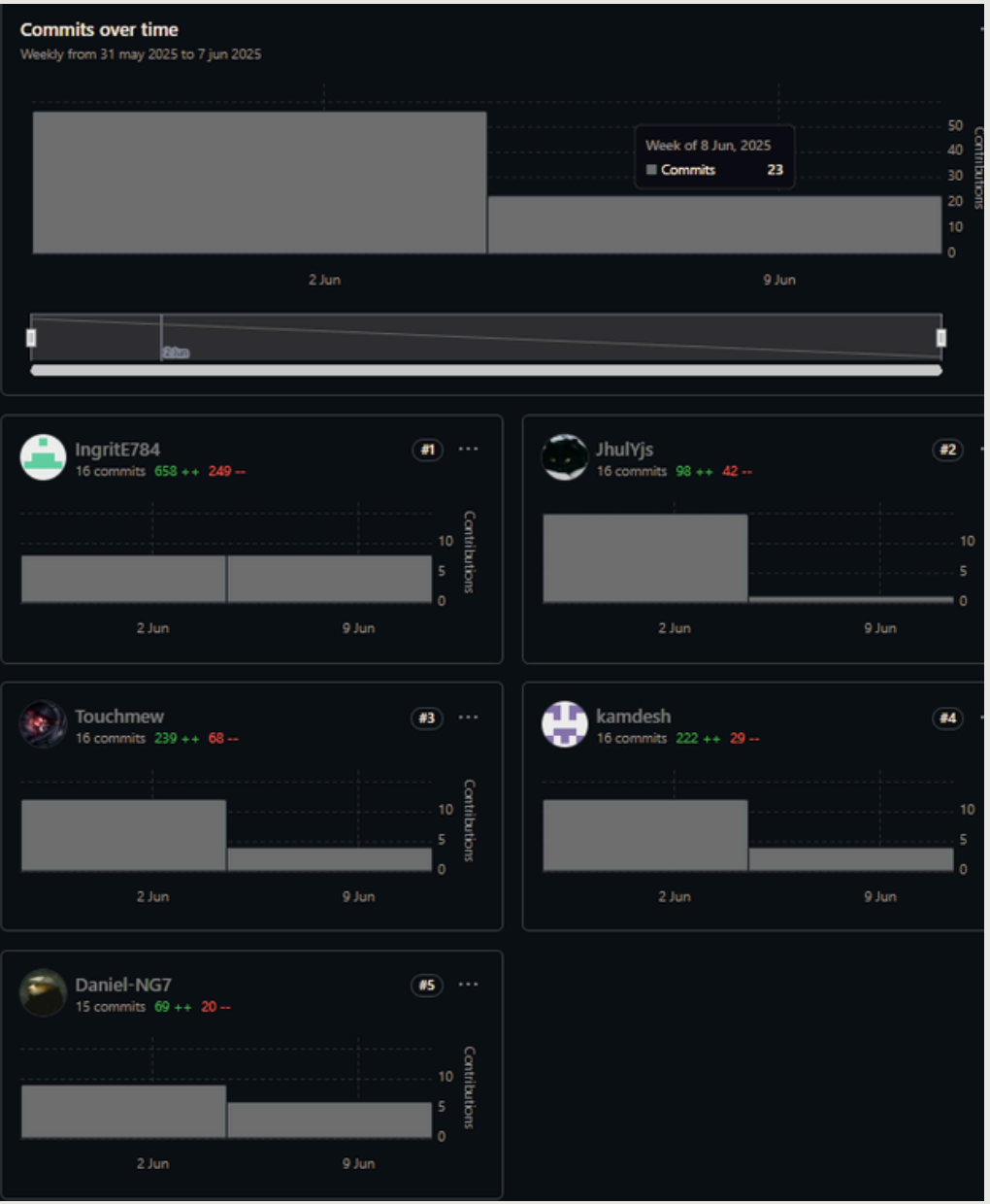


PRESENTACION: GESTOR DE TAREAS

En esta presentación, exploraremos el desarrollo de un sistema de gestión de tareas. Abordaremos desde el análisis del problema hasta la solución final, así como la evidencia del trabajo en equipo que hizo posible este proyecto. Nuestro objetivo es proporcionar una herramienta eficiente para organizar, controlar y dar seguimiento a las actividades del usuario.

Integrantes:

- Alvaro Gabriel Abril Abrill
- Jhul Dalens Gonzales
- Daniel Enoc NIna Gaurdapuclla
- Almir Aitor Ticona Sequeiros
- Ingrit Elida Huaman Villafuerte

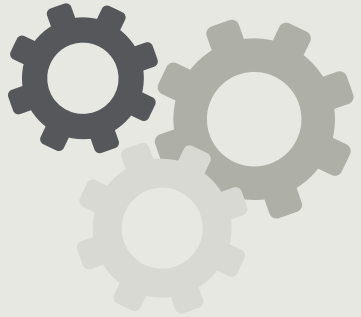
ANÁLISIS DEL PROBLEMA Y REQUERIMIENTOS

DESCRIPCIÓN DEL PROBLEMA

Un gestor de tareas es un software diseñado para organizar, controlar y dar seguimiento a las actividades. Su objetivo principal es ayudar a los usuarios a administrar sus tareas dentro de un dispositivo, considerando diversos aspectos relevantes para su eficiencia.

REQUERIMIENTOS DEL SISTEMA

- **Funcionales**: Gestión de aplicación (creación, modificación), gestión de registro (consulta, actualización, eliminación), manejo de datos (guardar información).
- **No funcionales (Calidad)**: Escalabilidad (manejo de datos sin perder eficiencia), manejo (rendimiento óptimo con estructuras dinámicas), estructuración del código (facilitar mejoras y nuevas funcionalidades).



ESTRUCTURAS DE DATOS PROPUESTAS

LISTA ENLAZADA

Utilizada en el Gestor de Procesos para almacenar y gestionar todos los procesos registrados en el sistema, permitiendo inserciones, eliminaciones y búsquedas eficientes.

COLA DE PRIORIDAD

Empleada en el Planificador de CPU para organizar y ejecutar procesos según su nivel de prioridad, replicando el comportamiento real de los sistemas operativos.

PILA

Utilizada en el Gestor de Memoria para simular la asignación y liberación de bloques de memoria de manera LIFO (Last-In, First-Out), facilitando un control eficiente.

JUSTIFICACIÓN DE LA ELECCIÓN DE ESTRUCTURAS

LISTA ENLAZADA (GESTOR DE PROCESOS)

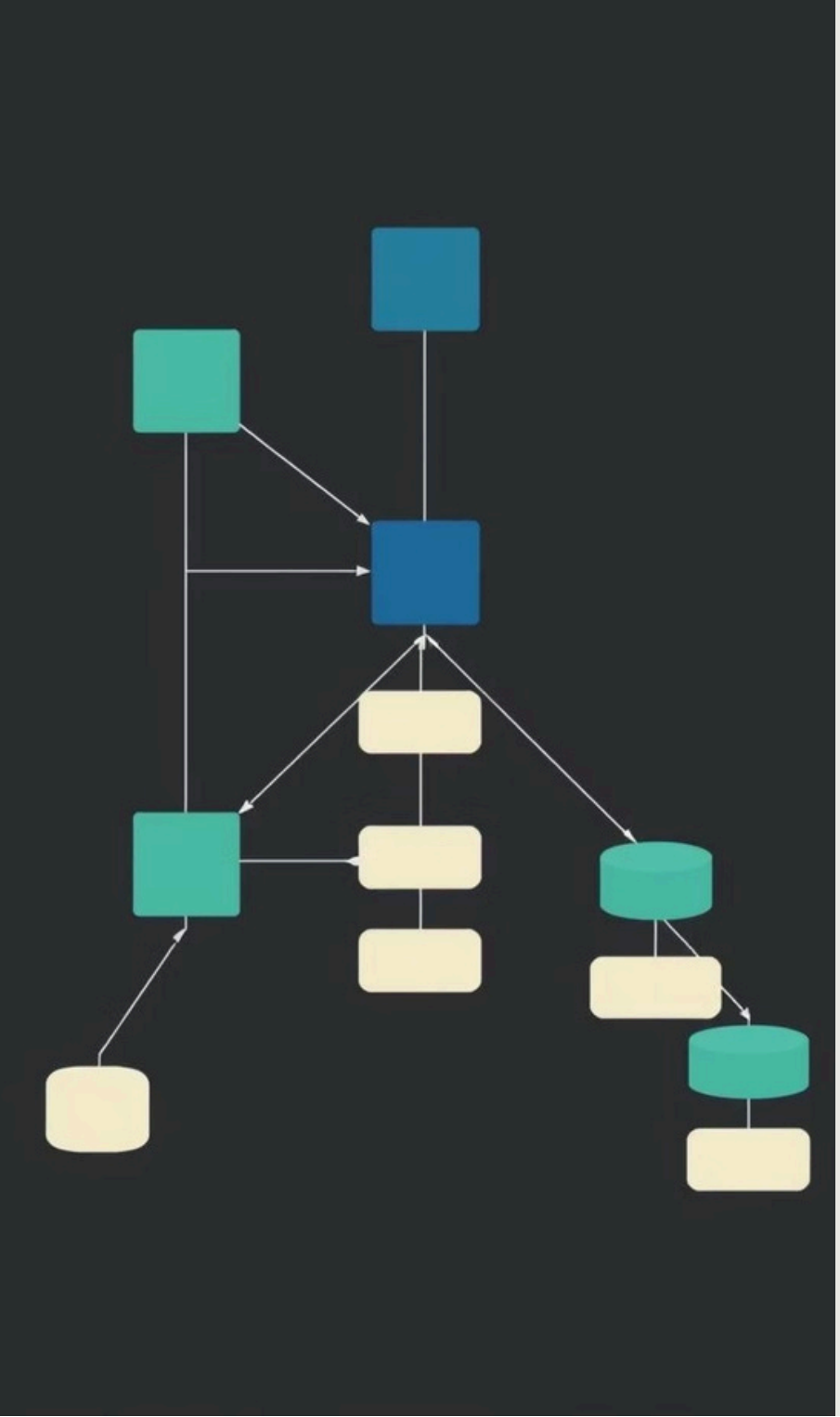
Elegida por su eficiencia para insertar, eliminar y buscar procesos en cualquier posición. Su naturaleza dinámica ajusta el tamaño según los procesos activos,
evitando el desperdicio de memoria y
facilitando la gestión de un número variable de tareas.

COLA DE PRIORIDAD (PLANIFICADOR DE CPU)

Ideal para organizar la ejecución de procesos según su importancia, replicando sistemas operativos reales. Asegura que los procesos prioritarios se atiendan primero y permite políticas
claras para resolver empates, como el
orden de llegada.

PILA (GESTOR DE MEMORIA)

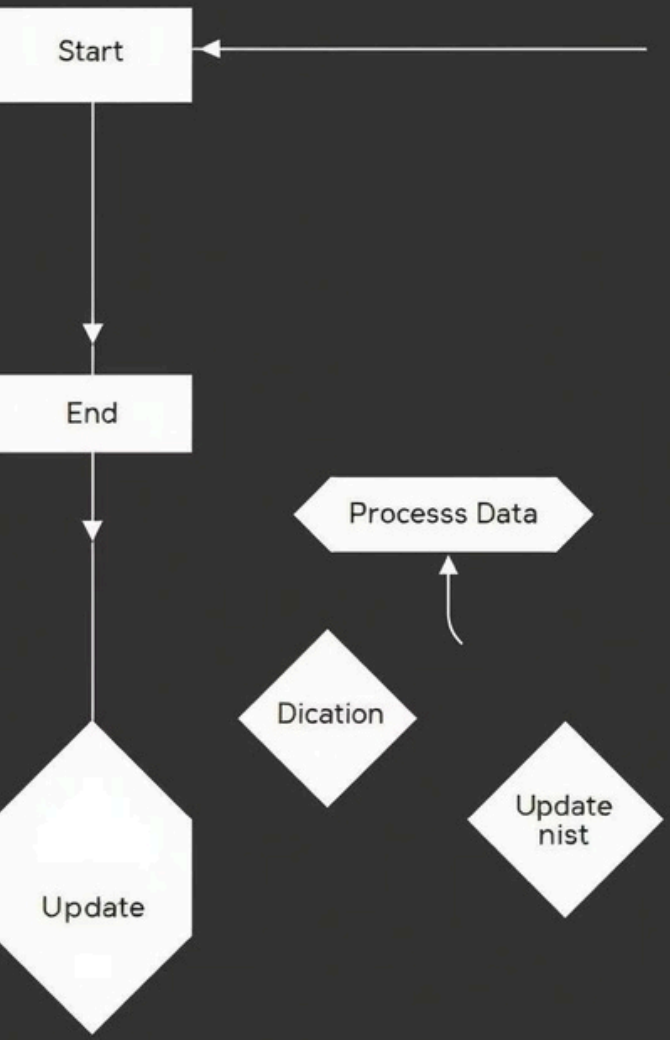
Simula la asignación y liberación de memoria bajo el modelo LIFO, común en muchos sistemas. Su simplicidad para manejar operaciones de inserción
y eliminación (push y pop) facilita un
control eficiente y ordenado del uso de
la memoria.



DISEÑO DE LA SOLUCIÓN: ESTRUCTURAS Y OPERACIONES

Este programa en C++ implementa un gestor de procesos utilizando estas tres
estructuras de datos principales, cada una con campos específicos y un uso definido
para optimizar la gestión de tareas, ejecución y memoria.

Lista Enlazada (Manejo de Procesos)	ID, Nombre, Estado, Prioridad, Fecha de creación ID, Nombre,	Almacenar y recorrer procesos creados
Cola con Prioridad (Ejecución de Procesos)	Prioridad, Tiempo de ejecución	Ordenar y ejecutar procesos según prioridad (alta, media, baja)
Pila (Gestión de Memoria)	ID de procesos, Nombre, Tamaño en MB	Simular asignación y liberación de memoria (LIFO)



ALGORITMOS PRINCIPALES

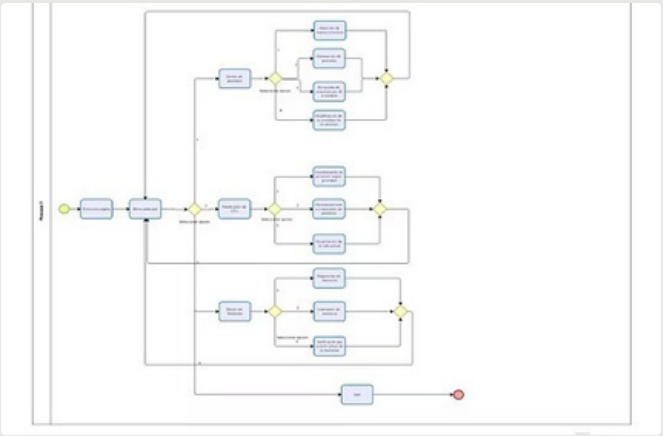
AGREGAR PROCESO

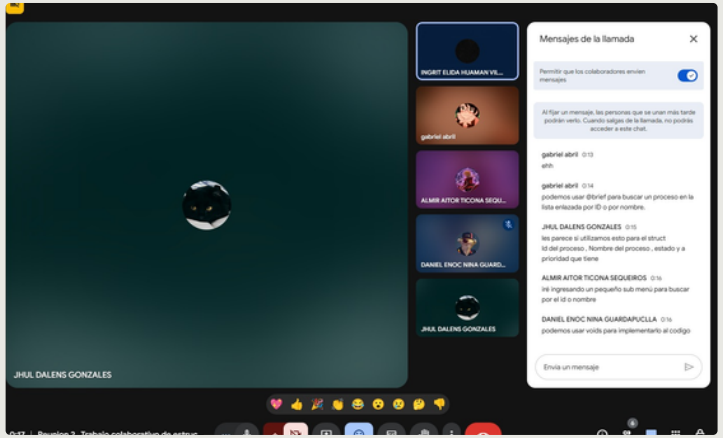
Crea un nuevo nodo con ID, Nombre, Prioridad, TamañoMemoria, Estado "Nuevo" y Fecha de Creación. Si la lista está vacía, el nuevo proceso es el primero; de lo contrario, se añade al final.

CAMBIAR ESTADO DEL PROCESO

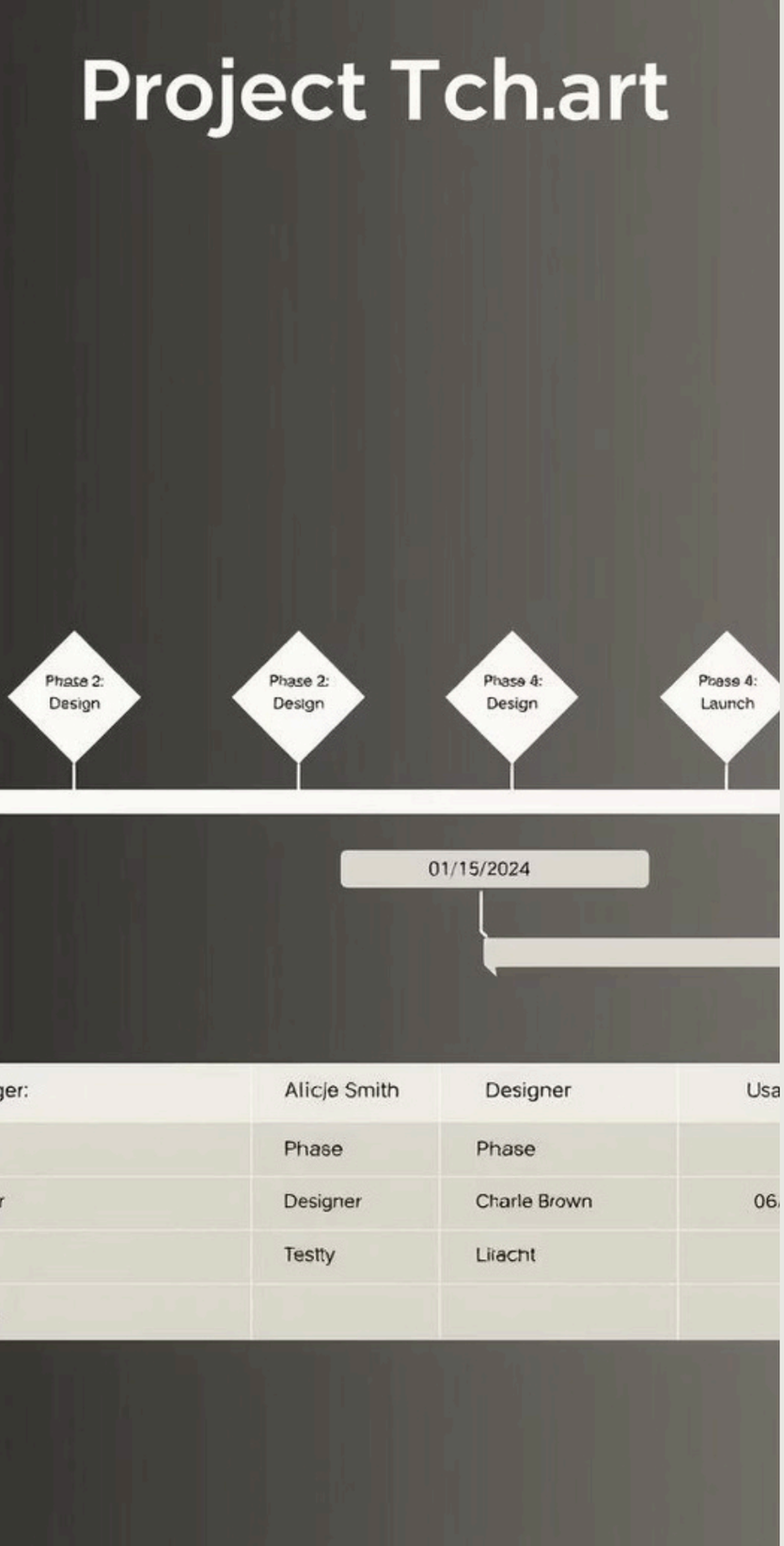
Recorre la lista de procesos buscando un ID específico. Si lo encuentra, actualiza el estado del proceso al NuevoEstado y muestra un mensaje de éxito. Si no se encuentra, informa al usuario.

EVIDENCIA DE TRABAJO EN EQUIPO





El proyecto se desarrolló con un fuerte enfoque en el trabajo colaborativo, utilizando un repositorio con control de versiones para registrar los aportes individuales. Se realizaron reuniones periódicas para la toma de decisiones y la revisión del código, asegurando la participación activa de cada miembro del equipo.



ROLES Y CRONOGRAMA DEL PROYECTO

1 — ANÁLISIS, DISEÑO Y DIAGRAMAS

Todos los integrantes leen y entienden el proyecto. Jhul crea diagramas de sistema y estructuras. Fecha límite: Lunes 2 de junio.

2 — IMPLEMENTACIÓN DE ESTRUCTURAS BASE

Implementación de listas enlazadas (Almir, Ingrit), colas de prioridad (Alva, Ingrit) y pilas (Daniel, Jhul). Fechas límite: Viernes-Sábado.

3 — INTEGRACIÓN Y PRUEBAS FINALES

Integrar módulos, desarrollar interfaz de consola, implementar persistencia y pruebas integrales. Fechas límite: Sábado-Domingo.

4 — DOCUMENTACIÓN Y ENTREGA FINAL

Documentación completa y entrega del sistema funcional. Fecha límite: Domingo 3:00 pm 3 9:00 pm.

El equipo se organizó con roles específicos:

- Líder (Ingrit),
- Programadores (Todos),
- Tester (Jhul), Integrador (Almir),
- Supervisor (Álvaro) y
- Documentador (Daniel),

siguiendo un cronograma detallado para cada entrega parcial.

ACTAS DE LAS REUNIONES

DECISIONES REGISTRADAS

Cada reunión tuvo actas detalladas para registrar acuerdos y tareas asignadas.

REVISIÓN CONTINUA

Facilitaron la revisión de código y la toma de decisiones estratégicas del proyecto.

SEGUIMIENTO DEL PROGRESO

Sirvieron como referencia clave para monitorear el avance y la culminación de tareas.

COMUNICACIÓN EFICAZ

Aseguraron la claridad y la participación activa de todos los miembros del equipo.

Evidencia del Sistema en Ejecución

•Menú principal

- Este programa simula las funciones básicas de un sistema operativo, gestionando:
- **Procesos** (creación, eliminación, búsqueda y modificación)
  - Crear: Añade procesos con ID único, nombre, estado (Activo/Inactivo/Terminado) y prioridad (Baja/Media/Alta).
  - Eliminar/Buscar: Por ID o nombre.
  - Modificar: Cambia prioridad o estado.
- **Planificación de CPU** (cola de ejecución priorizada)
  - Encolar: Ordena procesos por prioridad (Alta primero).
  - Ejecutar: Simula atención al siguiente proceso en cola.
  - Visualizar: Muestra lista de procesos pendientes.
- **Memoria** (asignación y liberación de bloques)
  - Asignar (Push): Registra procesos con su consumo de memoria (hasta 32 GB).
  - Liberar (Pop): Elimina la última asignación.
  - Memoria restante: Calcula y muestra porcentaje disponible.

```
//Menu principal
int main(){
    setlocale(LC_CTYPE, "Spanish");
    cargarProcesos();      // <- lee procesos.txt (si existe) y reconstruye la lista
    cargarPila();     // Cargar pila de memoria
    verificarArchivoCola();
    cargarCola();
    int opcion;
    do {
        cout << "\n=====================================\n";
        cout << "         MENU PRINCIPAL      \n";
        cout << "\n=====================================\n";
        cout << "|    1. Gestor de Proceso\n";
        cout << "|    2. Planificador de CPU \n";
        cout << "|    3. Gestor de Memoria\n";
        cout << "|    4. Salir\n";
        cout << "\n=====================================\n";
        cout << "Seleccione una opcion: ";
        cin >> opcion;
        cin.ignore();

        switch (opcion) {
            case 1: gestorDeProcesos(); break;
            case 2: planificadorCPU(); break;
            case 3: gestorDeMemoria(); break;
            case 4:
            guardarProcesos();
            guardarPila();
            guardarCola();
            cout << endl;
                cout << "|============================================================|" << endl;
                cout << "|                                                            |" << endl;
                cout << "|                        Saliendo...                         |" << endl;
                cout << "|                                                            |" << endl;
                cout << "|          Gracias por usar Nuestro Programa                 |" << endl;
                cout << "|                                                            |" << endl;
                cout << "|============================================================|" << endl;
                cout << endl;
                break;
            default:
                cout << "X Error: Ingrese una opción válida (1-4)" << endl;
        }

    } while (opcion != 4);

    return 0;
}
```

```
=====================================
         MENU PRINCIPAL

=====================================
|    1. Gestor de Proceso
|    2. Planificador de CPU
|    3. Gestor de Memoria
|    4. Salir

=====================================
Seleccione una opcion:
```

1. Gestor de procesos

El sistema implementa un gestor de Procesos, en el cual se utiliza un menu interactivo basado en do-while y switch-case para que el usuario pueda navegar de manera clara e intuitiva.

```
void gestorDeProcesos() {
    int opcion;
    do {
    cout << "\n=====================================\n";
        cout << "\n--- Gestor de Procesos ---\n";
        cout << "|    1. Insertar proceso\n";
        cout << "|    2. Eliminar proceso\n";
        cout << "|    3. Buscar proceso\n";
        cout << "|    4. Modificar prioridad\n";
        cout << "|    5. Mostrar procesos con fecha\n";
        cout << "|    6. Cambiar estado de un proceso\n";
        cout << "|    7. Volver al menu principal\n";
        cout << "|    Seleccione una opcion: ";
    cout << "\n=====================================\n";
        cin >> opcion;
        cin.ignore();

        switch (opcion) {
            case 1: insertarProceso(); break;
            case 2: eliminarProceso(); break;
            case 3: buscarProceso(); break;
            case 4: modificarPrioridad(); break;
        case 5: mostrarProcesosConFecha(); break;
        case 6: cambiarEstadoProcesoPorID();break;
        case 7 : cout << "Volviendo al menu principal...\n"; break;
        default: cout << "Opcion invalida.\n";
        }
    } while (opcion != 7);
}
```

```
--- Gestor de Procesos ---
|    1. Insertar proceso
|    2. Eliminar proceso
|    3. Buscar proceso
|    4. Modificar prioridad
|    5. Mostrar procesos con fecha
|    6. Cambiar estado de un proceso
|    7. Volver al menu principal
|    Seleccione una opcion:
=====================================
```

Insertar procesos

```cpp
void insertarProceso() {
    int id;
    string nombre, estado, prioridad;
    // Solicita al usuario que ingrese el ID del nuevo proceso
    cout << "Ingrese ID del proceso: ";
    cin >> id;
    cin.ignore(); // Limpia el buffer

    // Verificar si el ID ya existe
    Nodo* actual = inicio;
    while (actual != NULL) {
        if (actual->id_Proceso == id) {
            cout << "Error: Ya existe un proceso con ese ID.\n";
            return;
        }
        actual = actual->siguiente; // Avanza al siguiente nodo
    }

    cout << "Ingrese nombre del proceso: ";
    getline(cin, nombre);
    cout << "Ingrese estado del proceso(Activo/Inactivo/Terminado): ";
    getline(cin, estado);
    cout << "Ingrese prioridad del proceso(Baja/Media/Alta): ";
    getline(cin, prioridad);

   // Crea un nuevo nodo con los datos ingresados
    Nodo* nuevo = new Nodo(id, nombre, estado, prioridad);
// Inserta el nodo en la lista enlazada
    if (inicio == NULL) {
        inicio = nuevo;
    } else {
        actual = inicio;
        while (actual->siguiente != NULL) {
            actual = actual->siguiente; // Conecta el último nodo al nuevo
        }
        actual->siguiente = nuevo;
    }
    cout << "Proceso insertado correctamente.\n";
}
```

```
|    Seleccione una opcion: 1
Ingrese ID del proceso: 6
Ingrese nombre del proceso: Google
Ingrese estado del proceso(Activo/Inactivo/Terminado): activo
Ingrese prioridad del proceso(Baja/Media/Alta): media
Proceso insertado correctamente.
```

Esta función permite agregar un nuevo proceso con un identificador único, un nombre, un estado y una prioridad a una lista enlazada.

1. Solicitamos al usuario los datos del nuevo proceso:
   - ID(único)
   - Nombre del proceso
   - Estado (Activo, Inactivo, Terminado)
   - Prioridad (Baja, Media, Alta)

2. verifica si el ID ya existe en la lista enlazada
3. Si el ID es único, crea un nuevo nodo con los datos ingresados.
4. Inserta el nuevo proceso al final de la lista enlazada.
5. Muestra un mensaje de éxito: "Proceso insertado correctamente."

Eliminar proceso

```cpp
// Eliminar proceso por ID
void eliminarProceso() {
    int id;
    cout << "Ingrese el ID del proceso a eliminar: ";
    cin >> id;

    if (inicio == NULL) {
        cout << "La lista esta vacia.\n";
        return;
    }

    Nodo* actual = inicio;
    Nodo* anterior = NULL;

    while (actual != NULL && actual->id_Proceso != id) {
        anterior = actual;
        actual = actual->siguiente;
    }

    if (actual == NULL) {
        cout << "Proceso no encontrado.\n";
        return;
    }

    if (anterior == NULL) {
        inicio = actual->siguiente;
    } else {
        anterior->siguiente = actual->siguiente;
    }

    delete actual;
    cout << "Proceso eliminado correctamente.\n";
}
```

```
========================================
|    Seleccione una opcion: 2
Ingrese el ID del proceso a eliminar: 1
Proceso eliminado correctamente.
```

Esta función permite al usuario buscar y eliminar un proceso dentro de una lista enlazada. Se asegura que el ID existe en la lista antes de eliminar. También verifica si la lista esta vacía antes de proceder la búsqueda.

1. Solicita al usuario el ID del proceso a eliminar.
2.Verifica si la lista esta vacía (inicio == NULL)
3.Busca el proceso recorriendo la lista enlazada.
4. Si el proceso no se encuentra, se muestra un mensaje de error.
5. Si el proceso se encuentra, se elimina y se ajusta los punteros.
6. Se muestra un mensaje de éxito: “Proceso eliminado correctamente”.

Buscar proceso

```
void buscarProceso() {
    int opcion;
    cout << "Buscar por:\n1. ID\n2. Nombre\nSeleccione una opcion: ";
    //te da un menu pequenio para buscar por id o nombre
    cin >> opcion;
    cin.ignore();

    if (opcion == 1) {
        int id;
        cout << "Ingrese ID: ";
        cin >> id;

        Nodo* actual = inicio;
        while (actual != NULL) {
            if (actual->id_Proceso == id) {
            // Si encuentra el ID, muestra los detalles del proceso
                cout << "Proceso encontrado: " << actual->NombreProceso
                     << " | Estado: " << actual->Estado
                     << " | Prioridad: " << actual->Prioridad << endl;
                return;
            }
            actual = actual->siguiente; // Continúa con el siguiente nodo
        }

    } else if (opcion == 2) {
        string nombre;
        cout << "Ingrese nombre: ";
        getline(cin, nombre);

        Nodo* actual = inicio;
        while (actual != NULL) {
         // Si encuentra el nombre, muestra los detalles del proceso
            if (actual->NombreProceso == nombre) {
                cout << "Proceso encontrado: ID " << actual->id_Proceso
                     << " | Estado: " << actual->Estado
                     << " | Prioridad: " << actual->Prioridad << endl;
                return;
            }
            actual = actual->siguiente;
        }

    } else {
        cout << "Opcion invalida.\n";
        return;
    }
```

```
|    Seleccione una opcion: 3
Buscar por:
1. ID
2. Nombre
Seleccione una opcion: 1
Ingrese ID: 2
Proceso encontrado: set | Estado: alta | Prioridad: alta
```

Esta función permite al usuario que elija como buscar un proceso dentro de las listas enlazada. Se puede buscar por ID o Nombre del proceso.

1.Muestra un pequeño menú para que el seleccione el tipo de búsqueda.
2. Captura la opción del usuario (1para ID, 2 para nombre).
3. Realiza la búsqueda en la lista enlazada recorriendo los nodos:
   - si el usuario busca por ID, compara actual ->id_Proceso.
   - si el usuario busca por Nombre, compara actual → NombreProceso.
4. Si se encuentra el proceso, imprime los detalles (Estado, Prioridad).
5. si no se encuentra, muestra un mensaje de error “Procesos no encontrado”.
6. si el usuario ingresa una opción invalida, muestra “Opción invalida”.

- **Modificar prioridad**

```
// Modificar prioridad de un proceso por ID

void modificarPrioridad() {
    int id; // ingresamos un id
    cout << "Ingrese ID del proceso a modificar: ";
    cin >> id;
    cin.ignore();

    Nodo* actual = inicio;
    while (actual != NULL) {// miestras el nodo actual sea diferente de nulo
        if (actual->id_Proceso == id) { // si el id de algun nodo es igual al nodo ingresado se
            string nuevaPrioridad; //modificara la prioridad
            cout << "Ingrese nueva prioridad: ";
            getline(cin, nuevaPrioridad);
            actual->Prioridad = nuevaPrioridad;
            cout << "Prioridad actualizada correctamente.\n";
            return;
        }
        actual = actual->siguiente; // recorra toda la lsita
    }

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

Esta funcion permite modificar la prioridad de un proceso el cual existe buscando su ID, el usuario ingresa el ID del proceso que se desea actualizar y si se encuentra en la lista se le solicita la nueva prioridad. Siendo asi, una ves ingresada la prioridad del proceso se acrualiza y se muestra un mensaje de confirmacion, en caso no se encuentre el ID, se notifica al usuario que el proceso no existe

- **Mostrar Procesos con sus fechas**

En esta funcion muestra todos los procesos registrados junto con su fecha de creacion, primero se verifica si la lista esta vacia, si no lo esta recorre cada nodo desde el inicio e imprime en pantalla el ID , nombre, estado, prioridad y la fecha de creacion de cada proceso, Para culminar esto sirve para visualizar el historial y orden cronologico de los procesos

```
//Mostrar proceso con la fecha de registro
void mostrarProcesosConFecha() {
    if (inicio == NULL) {// confimamos si es que a lista esta vacia
        cout << "No hay procesos registrados.\n";
        return;
    }

    Nodo* actual = inicio; // el nodo actual es igual a inicio
    cout << "\n--- Lista de Procesos con Fecha de CreaciÃ³n ---\n";
    while (actual != NULL) { // si es distinto de nulo , se muestran los datos de los procesos registrados
        cout << "ID: " << actual->id_Proceso
             << " | Nombre: " << actual->NombreProceso
             << " | Estado: " << actual->Estado
             << " | Prioridad: " << actual->Prioridad
             << " | Fecha de creaciÃ³n: " << actual->fechaCreacion << endl;
        actual = actual->siguiente; // para que recorra toda la lista
    }
}
```

- **Cambiar estado de un proceso**

La función cambiarEstadoProcesoPorID permite al usuario cambiar el estado de un proceso buscando su ID. Solicita el ID y recorre la lista de procesos con un puntero actual. Si encuentra un nodo con el ID ingresado, muestra su estado actual y solicita un nuevo estado (Activo, Inactivo o Terminado). Luego, actualiza el estado del nodo, guarda los cambios en el archivo con guardarProcesos(), y confirma al usuario que la actualización fue exitosa. Si no se encuentra el proceso, informa al usuario que no se halló.

```
void cambiarEstadoProcesoPorID() {
    int id;
    cout << "Ingrese el ID del proceso que desea cambiar de estado: ";
    cin >> id;

    Nodo* actual = inicio;

    while (actual != NULL) {// si el nodo actual no esta vacio o no es igual a nulo
        if (actual->id_Proceso == id) { //confirmamos que el id del nodo actual se igual al que ingresamos
            cout << "Estado actual del proceso '" << actual->NombreProceso
                 << "' (ID: " << actual->id_Proceso << "): "
                 << actual->Estado << endl; // se imprimira el nodo actual con sus datos

            cout << "Ingrese el nuevo estado (Activo / Inactivo / Terminado): ";
            string nuevoEstado;
            cin.ignore(); // limpiar buffer
            getline(cin, nuevoEstado);// se ingresa un nuevo estado

            actual->Estado = nuevoEstado; // se le asigna el neuvo estao

            guardarProcesos(); // Se guarda el cambio en el archivo

            cout << "El estado del proceso ha sido actualizado a '" << nuevoEstado << "' correctamente." << endl;
            return;
        }
        actual = actual->siguiente; // el nodo actual en caso se nulo para a sel el siguente
    }

    cout << "No se encontró un proceso con el ID proporcionado." << endl;
}
```

- **Menú de Planificador de CPU**

Este módulo permite simular el orden en que la CPU ejecuta procesos. Se pueden encolar procesos según su prioridad (Alta, Media o Baja). Al ejecutarlos, se muestra su información y se elimina de la cola. También permite visualizar la cola completa. Así se imita cómo un sistema operativo organiza y da turno a los procesos.
Permite:

- Encolar proceso:
- Se ingresan los datos del proceso, incluida su prioridad. Según esta, se inserta en una posición específica dentro de la cola.
- Ejecutar siguiente proceso:
- Toma el proceso al frente de la cola, simula su ejecución y lo elimina.
- Visualizar cola:
- Muestra todos los procesos en espera con su ID, nombre, prioridad y tiempo de ejecución.

Así, se representa cómo un sistema operativo organiza y ejecuta procesos de forma ordenada.

```cpp
// Submenu de la opción 2. Planificador de CPU
void planificadorCPU() {
    int opcion;
    do {
    cout << "\n====================================\n";
        cout << "\n--- Planificador de CPU ---\n";
        cout << "|    1. Encolar proceso\n";
        cout << "|    2. Ejecutar siguiente proceso\n";
        cout << "|    3. Visualizar cola de procesos\n";
        cout << "|    4. Volver al menu principal\n";
        cout << "|    Seleccione una opcion: ";
    cout << "\n====================================\n";
        cin >> opcion;
        cin.ignore();

        switch (opcion) {
            case 1: encolarProceso(); break;
            case 2: desencolarProceso(); break;
            case 3: visualizarCola(); break;
            case 4: cout << "Volviendo al menu principal...\n"; break;
            default: cout << "Opcion invalida.\n";
        }
    } while (opcion != 4);
}
```

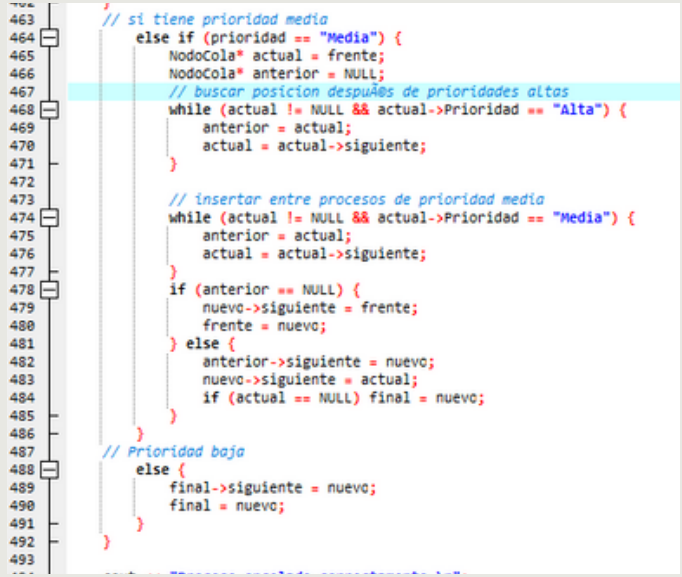
```
====================================

--- Planificador de CPU ---
|    1. Encolar proceso
|    2. Ejecutar siguiente proceso
|    3. Visualizar cola de procesos
|    4. Volver al menu principal
|    Seleccione una opcion:
====================================
3
ID: 1 | Nombre: word | Prioridad: media | Tiempo: 3 s
ID: 2 | Nombre: excel | Prioridad: baja | Tiempo: 2 s
```

- # Encolar procesos

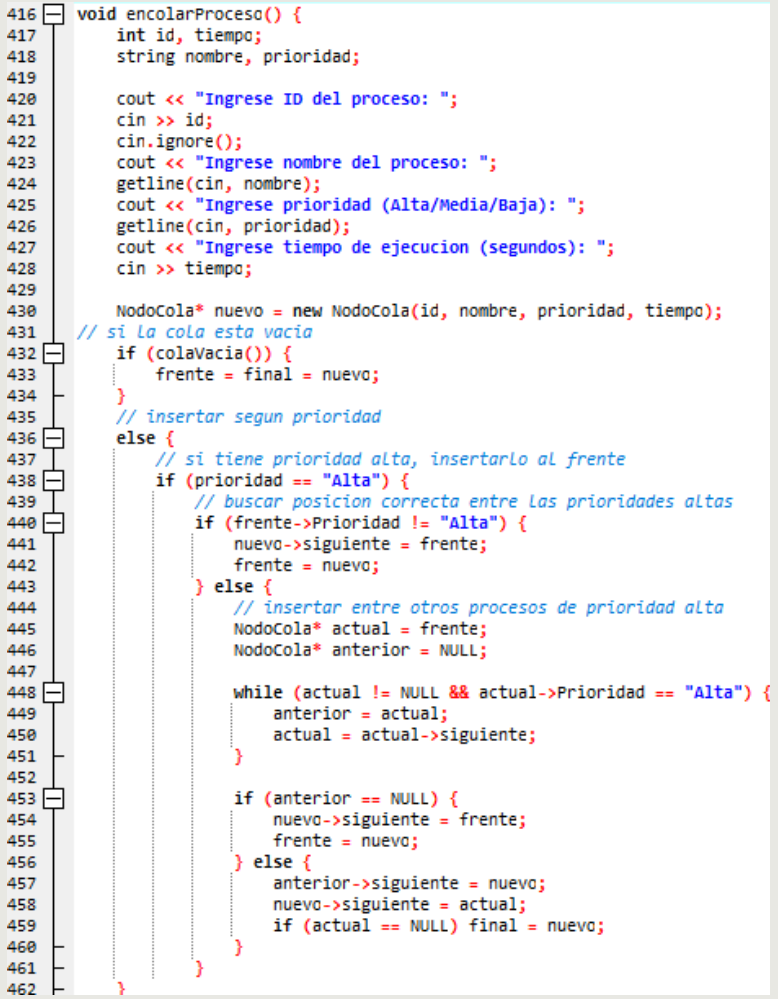
Función para enconlar proceso :

Se declaran las variables necesarias y se solicita al usuario que ingrese el ID, nombre, prioridad (Alta/Media/Baja) y tiempo de ejecución en segundos. Luego, se crea un nuevo nodo de tipo NodoCola con estos datos. La función verifica si la cola está vacía; si es así, el nuevo nodo se asigna como el frente y el final de la cola. La inserción del nodo se realiza según su prioridad: si es "Alta", se coloca al frente; si es "Media", se inserta después de los nodos de alta prioridad; y si es "Baja", se añade al final de la cola.



```
    // si tiene prioridad media
    else if (prioridad == "Media") {
        NodoCola* actual = frente;
        NodoCola* anterior = NULL;
        // buscar posicion despuÃ©s de prioridades altas
        while (actual != NULL && actual->Prioridad == "Alta") {
            anterior = actual;
            actual = actual->siguiente;
        }

        // insertar entre procesos de prioridad media
        while (actual != NULL && actual->Prioridad == "Media") {
            anterior = actual;
            actual = actual->siguiente;
        }
        if (anterior == NULL) {
            nuevo->siguiente = frente;
            frente = nuevo;
        } else {
            anterior->siguiente = nuevo;
            nuevo->siguiente = actual;
            if (actual == NULL) final = nuevo;
        }
    }
// Prioridad baja
    else {
        final->siguiente = nuevo;
        final = nuevo;
    }
}
```



```
void encolarProceso() {
    int id, tiempo;
    string nombre, prioridad;

    cout << "Ingrese ID del proceso: ";
    cin >> id;
    cin.ignore();
    cout << "Ingrese nombre del proceso: ";
    getline(cin, nombre);
    cout << "Ingrese prioridad (Alta/Media/Baja): ";
    getline(cin, prioridad);
    cout << "Ingrese tiempo de ejecucion (segundos): ";
    cin >> tiempo;

    NodoCola* nuevo = new NodoCola(id, nombre, prioridad, tiempo);
// si la cola esta vacia
    if (colaVacia()) {
        frente = final = nuevo;
    }
    // insertar segun prioridad
    else {
        // si tiene prioridad alta, insertarlo al frente
        if (prioridad == "Alta") {
            // buscar posicion correcta entre las prioridades altas
            if (frente->Prioridad != "Alta") {
                nuevo->siguiente = frente;
                frente = nuevo;
            } else {
                // insertar entre otros procesos de prioridad alta
                NodoCola* actual = frente;
                NodoCola* anterior = NULL;

                while (actual != NULL && actual->Prioridad == "Alta") {
                    anterior = actual;
                    actual = actual->siguiente;
                }

                if (anterior == NULL) {
                    nuevo->siguiente = frente;
                    frente = nuevo;
                } else {
                    anterior->siguiente = nuevo;
                    nuevo->siguiente = actual;
                    if (actual == NULL) final = nuevo;
                }
            }
    }
```

Ejecutar Procesos (Desencolar)

```
// Desencolar y ejecutar proceso

void desencolarProceso() {
    if (colaVacia()) {
        cout << "No hay procesos en la cola de ejecucion.\n";
        return;
    }

    NodoCola* procesoEjecutar = frente;

    cout << "\n--- Ejecutando Proceso ---\n";
    cout << "ID: " << procesoEjecutar->id_Proceso << endl;
    cout << "Nombre: " << procesoEjecutar->NombreProceso << endl;
    cout << "Prioridad: " << procesoEjecutar->Prioridad << endl;
    cout << "Tiempo de ejecucion: " << procesoEjecutar->tiempoEjecucion << " segundos\n";
    cout << "Proceso ejecutado exitosamente.\n";

    frente = frente->siguiente;
    if (frente == NULL) {
        final = NULL;
    }

    delete procesoEjecutar;
```

- Ejecuta el próximo proceso en la cola (siempre el de mayor prioridad)
- Libera los recursos del proceso después de su ejecución

Lógica de Operación

1. Verifica cola vacía → Muestra mensaje si no hay procesos
2. Toma el primer proceso (siempre el de mayor prioridad)
3. Muestra detalles:
   - ID, Nombre, Prioridad y Tiempo de ejecución
4. Actualiza punteros:
   - Avanza frente al siguiente proceso
   - Si era el último, limpia final
5. Libera memoria del proceso ejecutado

Visualizar Procesos

- Muestra todos los procesos pendientes en orden de ejecución

  Características:
- Formato claro: Muestra ID, Nombre, Prioridad y Tiempo estimado
- Orden natural: Desde el próximo a ejecutar (frente) hasta el último
- Manejo de casos:
  - Muestra mensaje especial si la cola está vacía
  - Recorre toda la lista enlazada para mostrar cada proceso

```
522  //Visualizacion de la cola actual
523  void visualizarCola() {
524      if (colaVacia()) {
525          cout << "No hay procesos en la cola.\n";
526          return;
527      }
528      NodoCola* actual = frente;
529      while (actual != NULL) {
530      cout << "ID: " << actual->id_Proceso
531           << " | Nombre: " << actual->NombreProceso
532           << " | Prioridad: " << actual->Prioridad
533           << " | Tiempo: " << actual->tiempoEjecucion << " s\n";
534      actual = actual->siguiente;
535      }
536  }
```

3.- Gestor de Memoria

En la opcion de gestor de memoria se imprimento un pequeño interfas utilizando do y switch para que sea mas intuitivo para el usuario.

```
void gestorDeMemoria(){
    int opcion;
    do {
        cout << "\n+-----------------------------------------------+\n";
    cout << "|              Gestor de Memoria                |\n";
    cout << "|-----------------------------------------------|\n";
    cout << "|  1. Asignación de memoria a procesos (push)|\n";
    cout << "|  2. Liberación de memoria (pop)            |\n";
    cout << "|  3. Ver estado actual de la memoria        |\n";
    cout << "|  4. Volver al menú principal               |\n";
    cout << "+-----------------------------------------------+\n";
    cout << "Seleccione una opción: ";
        cin >> opcion;
        cin.ignore();
        switch (opcion){
            case 1: AsignarMemoria(); break;
            case 2: LiberarMemoria(); break;
            case 3: MostrarMemoria(); break;
            case 4: cout << "Volviendo al menu principal...\n"; break;
            default: cout << "Opcion invalida.\n";
        }
    }while (opcion != 4);
}
```

```
Seleccione una opcion: 3

+-----------------------------------------------+
|              Gestor de Memoria                |
|-----------------------------------------------|
|  1. Asignación de memoria a procesos (push)|
|  2. Liberación de memoria (pop)            |
|  3. Ver estado actual de la memoria        |
|  4. Volver al menú principal               |
+-----------------------------------------------+
Seleccione una opción: 1
Ingrese el nombre del proceso: word
Ingrese tamaño de memoria (mg): 17000
>> Memoria asignada al proceso: word Id 1 (con un tamano de 17000 mg) | (17 Gb)
>> Quedan 46.875% de memoria disponible.
```

Partes implementadas en el Gestor de Memoria

1.-Asignacion de memoria a procesos (Push):

- se realiza mediante una función que solicita al usuario el nombre del proceso y el tamaño de memoria en miligramos. Se verifica si hay suficiente memoria (32,000 mg o 32 GB) y, si es así, se imprime el nombre del proceso, su ID, el tamaño asignado y el porcentaje de memoria disponible restante. Esto garantiza una gestión eficiente de los recursos del sistema.

```
====================================
1
Ingrese el nombre del proceso: Google
Ingrese tamaño de memoria (mg): 8000
>> Memoria asignada al proceso: Google Id 1 (con un tamano
 de 8000 mg) | (8 Gb)
>> Quedan 75% de memoria disponible.

====================================
```

```
void AsignarMemoria(){
    double tamano;
    string Nombre;
    int id = siguienteID++;
    cout << "Ingrese el nombre del proceso: ";
    cin >> Nombre;
    cout << "Ingrese tamaño de memoria (mg): ";
    cin >> tamano;
// Verifica si hay suficiente memoria disponible
    if (memoriaUtilizada + tamano > MEMORIA) {
        cout << ">> No hay suficiente memoria disponible para asignar " << tamano << " MB.\n";
        return;}
  // Crea un nuevo bloque de memoria
    BloqueMemoria* nuevo = new BloqueMemoria;
    nuevo->idProceso = id; // Asigna el ID al nuevo bloque
    nuevo->Nombre = Nombre;
    nuevo->tamano = tamano;
    nuevo->siguiente = cima;// Enlaza el nuevo bloque a la cima actual
    cima = nuevo; // Actualiza la cima

    memoriaUtilizada += tamano; // Actualizar memoria utilizada
    double memoriaRestante = MEMORIA - memoriaUtilizada;// Calcula la memoria restante
    double porcentajeRestante = (memoriaRestante / MEMORIA) * 100;// Porcentaje de memoria restante

    cout << ">> Memoria asignada al proceso: " << Nombre << " Id " << id
    << " (con un tamano de " << tamano << " mg) | (" << tamano / 1000 << " Gb)\n";
    cout << ">> Quedan " << porcentajeRestante << "% de memoria disponible.\n";
```

2. **Liberación de memoria (Pop):**
   - La Liberación de Memoria se implementa en una función que primero verifica si la memoria está vacía. Si no está vacía, se procede a eliminar el último proceso ingresado utilizando únicamente su ID.

```
void MostrarMemoria(){
    if (cima == NULL) { // Verifica si la pila está vacía
        cout << ">> La pila de memoria esta vacia.\n";
        return;
    }

    cout << ">> Estado actual de la memoria (de mas reciente a mas antigua):\n";
    BloqueMemoria* actual = cima;// Comienza desde la cima de la pila
    while (actual != NULL) {// Recorre todos los bloques de memoria
         cout << "   - Proceso ID: " << actual->idProceso << " Nombre:" << actual->Nombre
           <<", tamano: " << actual->tamano << " mg\n";
        actual = actual->siguiente;// Avanza al siguiente bloque
    }
    double memoriaRestante = MEMORIA - memoriaUtilizada;
    double porcentajeRestante = (memoriaRestante / MEMORIA) * 100;
    cout << ">> Quedan " << porcentajeRestante << "% de memoria disponible.\n";
}
```

```
=====================================
3
>> Estado actual de la memoria (de mas reciente a mas
 antigua):
   - Proceso ID: 1 Nombre:Google, tamano: 8000 mg
>> Quedan 75% de memoria disponible.

=====================================
```

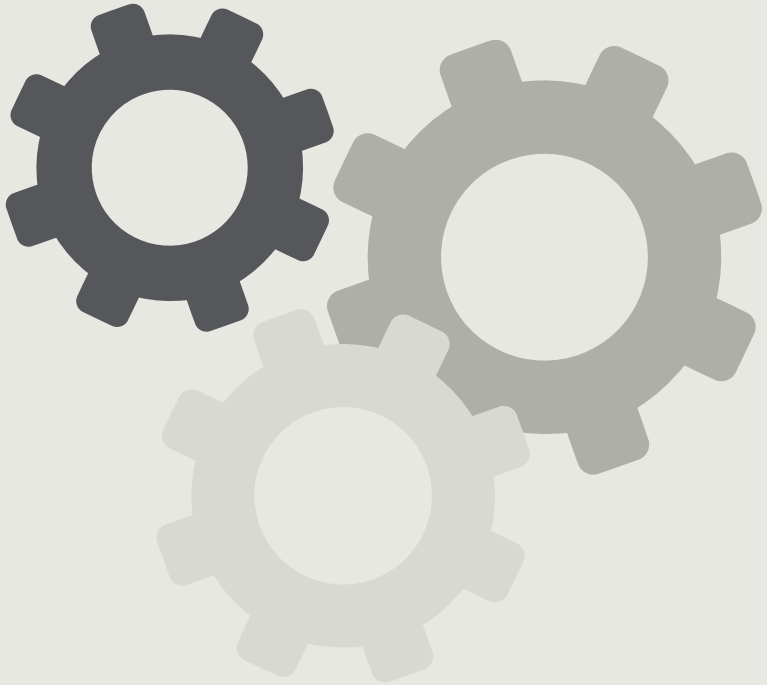
```
void LiberarMemoria(){
    if (cima == NULL) {// Verifica si la pila está vacía
        cout << ">> No hay bloques de memoria para liberar.\n";
        return;
    }

    BloqueMemoria* temp = cima;// Almacena el bloque a liberar
    cima = cima->siguiente;
    siguienteID--;// Decrementa el ID para la próxima asignación
    memoriaUtilizada -= temp->tamano;
    cout << ">> Memoria liberada del proceso " << temp->idProceso <<" "
    << temp->Nombre << " (" << temp->tamano << " mg)\n";
    delete temp;// Libera la memoria del bloque
}
```

```
=====================================
2
>> Memoria liberada del proceso 1 Google (8000 mg)

=====================================
```

3. **Mostrar Memoria:**
   - La función MostrarMemoria verifica si la pila de memoria está vacía. Si hay procesos, recorre los bloques de memoria, mostrando el ID del proceso, su nombre y el tamaño asignado. Al final, calcula y muestra el porcentaje de memoria disponible.



Codigos de Persistencia

Persistencia de Gestor de Procesos

```cpp
// --------------------- Funciones de Persistencia de Listas ---------------------

void guardarProcesos() {
    ofstream archivo("procesos.txt"); //crea un flijo de salida para escribir en un archivo txt
    if (!archivo) {  //verifica si el archivo se abre correctamente
        cerr << "Error al abrir procesos.txt.\n";
        return;
    }
    //recorre las listas enlazadas comenzando desde el nodo inicio
    for (Nodo* act = inicio; act; act = act->siguiente) {
        // Escribimos los datos de cada proceso en el archivo, separados por '|'
        archivo << act->id_Proceso        << '|'
                << act->NombreProceso     << '|'
                << act->Estado            << '|'
                << act->Prioridad         << '|'
                << act->fechaCreacion     << '\n';
    }
  // Cerramos el archivo una vez que todos los procesos han sido guardados
    archivo.close();
    cout << "Procesos guardados correctamente.\n";
}
```

- Guardar datos

Esta función guarda todos los procesos registrados en un archivo de texto llamado procesos.txt. Recorre la lista enlazada desde el nodo inicial y escribe los datos de cada proceso (ID, nombre, estado, prioridad y fecha de creación) en una línea del archivo, separándolos con el carácter '|'. Una vez finalizado el recorrido, cierra el archivo y notifica que los procesos fueron guardados correctamente.

- Cargar Datos

Esta función carga los procesos previamente guardados desde el archivo procesos.txt. Abre el archivo en modo lectura y procesa línea por línea, separando cada campo por el carácter '|' mediante un stringstream. Luego convierte los datos a los tipos apropiados y reconstruye los nodos de la lista enlazada, insertándolos al final. También conserva la fecha de creación si está presente en el archivo. Si el archivo no existe, se asume que es la primera ejecución del sistema.

```cpp
//carga los procesos guardados en el archivo txt
void cargarProcesos() {
    ifstream archivo("procesos.txt");//abre el archivo txt
    if (!archivo.good()) {
    cout << "(Primera ejecuciÃ³n) No se encontraron procesos guardados.\n";
    return; //sale de la funcion si no encuentra el archico a leer
    }
    string linea;
    //Lee el archivo linea por linea
    while (getline(archivo, linea)) {

        if (linea.empty()) continue;
    // Se utiliza un stringstream para dividir la LÃnea en campos separados por '|'
        stringstream ss(linea);
        string idStr, nombre, estado, prioridad, fecha;
     // Extraemos cada campo individualmente
        getline(ss, idStr,     '|');
        getline(ss, nombre,    '|');
        getline(ss, estado,    '|');
        getline(ss, prioridad, '|');
        getline(ss, fecha);                 // puede estar vacÃa en archivos antiguos
    // Convertimos el ID de string a entero
        int id;
        stringstream ss_id(idStr);
        ss_id >> id;

        Nodo* nuevo = new Nodo(id, nombre, estado, prioridad);
        if (!fecha.empty()) nuevo->fechaCreacion = fecha;   // conserva la fecha de crea

        /* inserta al final de la lista */
        if (!inicio) {
        inicio = nuevo;
    }else {
        // Si ya hay elementos, recorremos hasta el final para insertar
        Nodo* a = inicio;
        while (a->siguiente) a = a->siguiente;
        a->siguiente = nuevo;
        }
    }

    archivo.close();
    cout << "Procesos cargados correctamente.\n";
}
```

Persistencia de PLanificador del CPU

1.- Función para guardar el archivo de la cola:

.La función guardarCola() guarda los datos de una cola en "cola.txt". Si el archivo se abre correctamente, recorre los nodos desde el frente, escribiendo cada atributo (id, nombre, prioridad y tiempo de ejecución) separados por "|". Al finalizar, cierra el archivo y muestra un mensaje de éxito o error.

```
void guardarCola() {
    ofstream archivo("cola.txt");
    if (archivo.is_open()) { // si el archivo se abre
        NodoCola* actual = frente;
    // Recorre todos los nodos de la cola
        while (actual != NULL) { // comprovamos que el nodo actual es diferente de nulo
            archivo << actual->id_Proceso << "|"// guardan los datos  ingresados
                    << actual->NombreProceso << "|"
                    << actual->Prioridad << "|"
                    << actual->tiempoEjecucion << "\n";
            actual = actual->siguiente; // Avanza al siguiente nodo
        }
        archivo.close();/// se cierra el archivo
        cout << "Cola guardada  exitosamente.\n";
    } else { // si el archivo no abre te mandara un error
        cerr << "Error al abrir archivo para guardar cola.\n";
    }
}
```

```
void cargarCola() {
    ifstream archivo("cola.txt");
    if (archivo.is_open()) { // si el archivo esta abierto
        string linea;
    // Lee el archivo línea por línea
        while (getline(archivo, linea)){
        stringstream ss(linea);
        string idStr, nombre, prioridad, tiempoStr; // se consideran todos los datos que se han guardado
        getline(ss, idStr, '|');
        getline(ss, nombre, '|');
        getline(ss, prioridad, '|');
        getline(ss, tiempoStr, '|');
        // Convierte los campos numéricos de string a int
        int id;
        stringstream ssId(idStr);
        ssId >> id;

        int tiempo;
        stringstream ssTiempo(tiempoStr);
        ssTiempo >> tiempo;
        // Inserta el nodo en la cola
            NodoCola* nuevo = new NodoCola(id, nombre, prioridad, tiempo);
            nuevo->siguiente = NULL;

            if (frente == NULL) {
                frente = final = nuevo; // Si la cola está vacía, este nodo es el primero y el último
            } else {
                final->siguiente = nuevo; // Lo enlaza al final
                final = nuevo;      // Actualiza el final
            }
        }
        archivo.close();// el archivo se guarda
        cout << "Cola cargada exitosamente \n";
    } else { // caso contrario nos manda un error
        cerr << "No se pudo abrir para cargar.\n";
    }
}
```

2.- Función para cargar el archivo de la cola:

La función cargarCola() carga datos de "cola.txt" en una cola. Si el archivo se abre, lee cada línea, extrae id, nombre, prioridad y tiempo, convierte los campos numéricos y crea un nuevo nodo. Si la cola está vacía, el nodo se convierte en el primero y el último; de lo contrario, se añade al final. Finalmente, cierra el archivo y muestra un mensaje de éxito o error.

3.Persistencia de datos en Gestor de Memoria

1.- Función para guardar la pila en un archivo:

- Se realizo un codigo para que los datos registrado en el Gestor de Memoria se almacene en pila.txt para poder almacenar todos los datos registrados al cerrar el programa.

```
void guardarPila() {
    ofstream archivo("pila.txt");// Abre el archivo para escritura
    if (!archivo) { cerr << "Error al abrir pila.txt.\n"; return; }

    BloqueMemoria* actual = cima;// Comienza desde la cima de la pila
    while (actual) {
        archivo << actual->idProceso << '|'
                << actual->Nombre << '|'
                << actual->tamano << '\n';
        actual = actual->siguiente;
    }
    cout << "Pila guardada.\n";
}
```

```
Pila guardada.
```

```
void cargarPila() {
    ifstream archivo("pila.txt"); // Abre el archivo para lectura
    if (!archivo.good()) {
        cout << "Sin datos en la pila guardados.\n";
        return;
    }

    // Vaciar pila actual si hay
    while (cima) {
        BloqueMemoria* aux = cima;
        cima = cima->siguiente;
        delete aux;
    }

    BloqueMemoria* base = NULL; // último nodo de la pila cargada
    BloqueMemoria* ultimo = NULL; // puntero para insertar al final
```

2.- Función para cargar el archivo de la pila :

- La función cargarPila se encarga de cargar datos desde un archivo llamado "pila.txt" en una pila de memoria. Primero, verifica si el archivo se abre correctamente; si no, imprime un mensaje y termina. Luego, vacía la pila actual si existe.

2.1.- Función para cargar el archivo de la pila :

A continuación, lee el archivo línea por línea, ignorando las líneas vacías. Para cada línea, extrae el ID del proceso, el nombre y el tamaño. Si alguna de estas partes está vacía, se salta la línea. Después, crea un nuevo nodo de memoria con los datos leídos e inserta este nodo al final de una lista temporal.

```
BloqueMemoria* base = NULL; // último nodo de la pila cargada
BloqueMemoria* ultimo = NULL; // puntero para insertar al final
string linea;
while (getline(archivo, linea)) {
    if (linea.empty()) continue;
    stringstream ss(linea);
    string idStr, nombre, tamanoStr;
    getline(ss, idStr, '|');
    getline(ss, nombre, '|');
    getline(ss, tamanoStr);

    if (idStr.empty() || tamanoStr.empty()) {
            cout << "Línea malformada: " << linea << endl;
        continue;  // salta esta línea
    }
    int id;
    stringstream ssId(idStr);
    ssId >> id;
    double tamano;
    stringstream ssTam(tamanoStr);
    ssTam >> tamano;
    // Crea un nuevo nodo de memoria con los datos leídos
    BloqueMemoria* nodo = new BloqueMemoria;
    nodo->idProceso = id;
    nodo->Nombre = nombre;
    nodo->tamano = tamano;
    nodo->siguiente = NULL;
    // Inserta el nodo al final de la lista temporal
    if (base == NULL) {
        base = nodo;
            ultimo = nodo;
    } else {
            ultimo->siguiente = nodo;
            ultimo = nodo;
    }
}
```

2.2.- Función para cargar el archivo de la pila :

Finalmente, invierte la lista temporal para convertirla en una pila, actualizando el puntero cima para que apunte al nuevo tope de la pila. Al finalizar, cierra el archivo y muestra un mensaje indicando que la pila se ha cargado correctamente.

```
    // Revertir la lista para convertirla en pila
    BloqueMemoria* prev = NULL;
    BloqueMemoria* current = base;
    BloqueMemoria* next = NULL;
    // Invierte los punteros de la lista
    while (current != NULL) {
        next = current->siguiente;
        current->siguiente = prev;
        prev = current;
        current = next;
    }
    cima = prev; // ahora cima apunta a la cima real de la pila
        archivo.close();
    cout << "Pila cargada correctamente.\n";
}
```

GRACIAS

Por su Atencion