Universidad Continental

Trabajo Grupal

“SISTEMA DE GESTIÓN DE PROCESOS EN UN SISTEMA OPERATIVO”

CURSO:
Estructura de Datos

DOCENTE:
Ing. Ninoska Nataly Cohaila Bravo

Integrantes:

- Torres Huamani Aldair Alexis
- Laos Ayala Joseph Karl
- Silva Crispin, Yanpol
- Jeferson Eli Rios Ayala

Huancayo - 2025

Índice del informe:

CAPÍTULO 1: Análisis del Problema

1. Descripción del problema

En un sistema operativo, tareas como la gestión de procesos, la planificación de la CPU y la asignación de memoria son esenciales y deben ejecutarse de manera ordenada y eficiente. No obstante, representar este funcionamiento de forma didáctica y comprensible para los estudiantes supone un desafío, especialmente cuando se requiere aplicar conceptos de estructuras de datos dinámicas.

El reto consiste en cómo diseñar y manejar estos elementos procesos, colas de ejecución y memoria sin recurrir a estructuras de datos predefinidas, garantizando que el sistema permita realizar operaciones básicas como inserción, eliminación, búsqueda y modificación de datos. Asimismo, debe incorporar mecanismos que reproduzcan la planificación basada en prioridades y la gestión dinámica de memoria dentro de un entorno controlado.

Este proyecto busca ofrecer una herramienta educativa que facilite la simulación práctica y sencilla de la administración de procesos en un entorno similar a un sistema operativo, empleando listas enlazadas, colas y pilas desarrolladas manualmente.

2. Requerimientos del sistema

- Funcionales
    - Encolar y desencolar procesos en función de su prioridad.
    - Asignar y liberar bloques de memoria simulada a los procesos.
    - Validar entradas de usuario (evitar letras en campos numéricos).
    - Mostrar el estado actual de la memoria y la cola de procesos.

- No funcionales
    - El sistema debe ejecutarse en consola y ser compatible con C++.
    - Debe ser estable y manejar errores de ingreso de datos.
    - La estructura del código debe estar bien comentada y modularizada.
    - Debe ejecutarse correctamente en entornos como **Dev-C++**
    - La interacción debe ser sencilla e intuitiva para el usuario.

3. Estructuras de datos propuestas

3.1 Lista enlazada para el gestor de procesos

Para el módulo del gestor de procesos se implementó una lista enlazada simple, donde cada nodo representa un proceso almacenado dinámicamente.

Cada nodo (estructura Proceso) contiene la siguiente información:

- PID: identificador único del proceso.
- Nombre: nombre asignado por el usuario.
- Prioridad: nivel de prioridad del proceso (entero positivo).
- Puntero al siguiente proceso: que mantiene el enlace con el siguiente nodo en la lista.

El uso de una lista enlazada permite agregar, eliminar y recorrer procesos de manera flexible y eficiente, ya que no requiere un tamaño fijo y facilita la gestión dinámica de los procesos que están registrados en el sistema.

3.2 Cola por prioridad para el planificador de CPU

En el caso del planificador de CPU, se utilizó una cola con prioridad implementada mediante una lista enlazada ordenada (NodoCola).
Cada nodo apunta a un proceso existente en la lista general, y se inserta en la posición correspondiente según su nivel de prioridad.
Los procesos con menor valor numérico en prioridad (es decir, con mayor prioridad real) son ubicados al frente de la cola.

Esta estructura permite:

- Insertar procesos de forma ordenada según su prioridad.
- Desencolar y ejecutar el proceso más prioritario.
- Visualizar en cualquier momento el orden de ejecución de la cola.

Con este diseño se simula adecuadamente la planificación de la CPU basada en prioridades, similar a la forma en que los sistemas operativos reales gestionan los procesos listos para ejecutarse.

3.3 Pila para el gestor de memoria

Para la gestión de memoria se empleó una pila (estructura LIFO), implementada también mediante una lista enlazada (BloqueMemoria).
Cada nodo de la pila representa un bloque de memoria asignado a un proceso, e incluye:

- PID del proceso propietario.
- Tamaño del bloque de memoria asignado.
- Puntero al siguiente bloque en la pila.

Las operaciones principales que maneja son:

- Asignar memoria (push): agrega un nuevo bloque en la parte superior de la pila.
- Liberar memoria (pop): elimina el último bloque asignado.
- Visualizar el estado actual de la memoria: muestra los bloques activos y su tamaño.

Esta estructura permite simular una gestión simple de la memoria dinámica, donde los últimos bloques asignados son los primeros en liberarse, representando un comportamiento ordenado y controlado del uso de memoria.

4. Justificación de la elección

La elección de las estructuras de datos implementadas lista enlazada, cola por prioridad y pila responde directamente a las necesidades funcionales del sistema de gestión de procesos. Cada una de ellas permite simular con precisión el comportamiento de componentes reales de un sistema operativo, garantizando un manejo eficiente, ordenado y dinámico de los datos.

4.1 Lista enlazada para el gestor de procesos

Se seleccionó una **lista enlazada simple** para almacenar los procesos registrados dentro del sistema. Esta estructura resulta adecuada porque permite la inserción y eliminación de nodos de manera dinámica, sin requerir una cantidad fija de memoria. Además, facilita el recorrido lineal para mostrar o buscar procesos de forma sencilla.

Entre sus ventajas se destacan la posibilidad de agregar procesos al final sin necesidad de redimensionar un arreglo, eliminar nodos reajustando punteros en lugar de mover elementos, y utilizar la memoria de manera flexible.

Por estas razones, la lista enlazada es ideal para gestionar un conjunto cambiante de procesos, cumpliendo eficientemente con su rol dentro del **Gestor de Procesos**.

4.2 Cola por prioridad para el planificador de CPU

Para el **planificador de CPU**, se utilizó una **cola con prioridad** implementada como una lista enlazada ordenada de forma ascendente (donde un número menor representa una prioridad más alta). Esta estructura permite insertar procesos en la posición que les corresponde según su nivel de prioridad, garantizando que el proceso más importante se encuentre siempre al inicio. Además, su funcionamiento simula fielmente la planificación real de los sistemas operativos, donde los procesos más prioritarios son los primeros en ejecutarse.
Gracias a esta implementación, se obtiene una simulación ordenada, eficiente y comprensible del comportamiento de un planificador de CPU basado en prioridades, sin necesidad de estructuras más complejas como montículos o árboles.

4.3 Pila para el gestor de memoria

La **pila enlazada** fue seleccionada para la gestión de memoria, siguiendo el principio **LIFO (Last In, First Out)**. Este enfoque permite que la última porción de memoria asignada sea la primera en liberarse, simulando el funcionamiento de la memoria en entornos de programación reales.
Entre sus ventajas se encuentra la rapidez para asignar o liberar bloques (ya que solo se manipula el tope de la pila) y la simplicidad de su implementación, al no requerir recorridos ni ordenamientos.
Por estas características, la pila resulta ideal para representar una administración de memoria básica, especialmente útil en entornos educativos y de simulación como el de este proyecto.

4.4 Simplicidad en el recorrido y manejo

El uso combinado de estas estructuras lista, cola y pila ofrece un sistema claro, ordenado y fácil de mantener.
La **lista enlazada** permite insertar, eliminar y buscar procesos fácilmente;
la **cola con prioridad** organiza la ejecución de los procesos según su importancia;
y la **pila** gestiona la memoria de forma rápida y estructurada.
En conjunto, proporcionan una base sólida para la simulación del sistema operativo, manteniendo un código limpio, modular y eficiente.

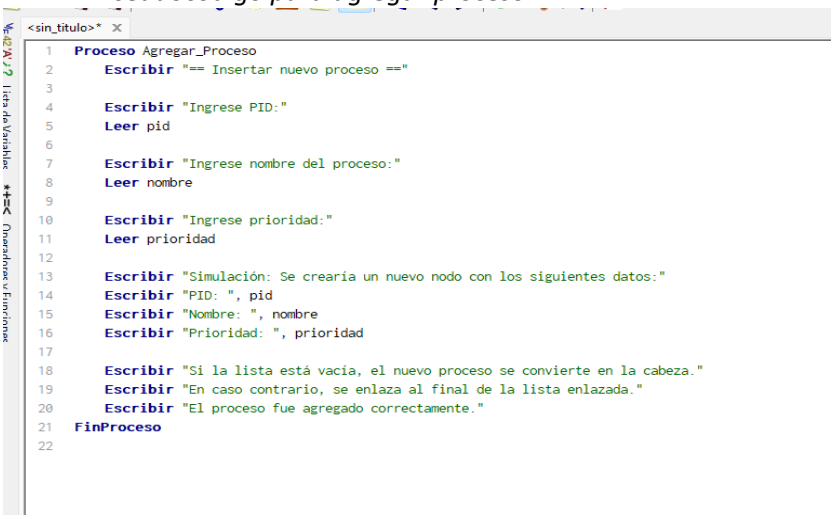
Capítulo 2: Diseño de la Solución

1. Descripción de estructuras de datos y operaciones:

El sistema fue diseñado utilizando tres estructuras de datos principales: una **lista enlazada**, una **cola con prioridad** y una **pila**, cada una cumpliendo una función específica dentro de la simulación.
La lista enlazada permite **almacenar, insertar y eliminar procesos** de forma dinámica, representando la lista general de procesos del sistema.
La cola con prioridad organiza los procesos **según su nivel de prioridad**, garantizando que los más importantes se ejecuten primero, como ocurre en la **planificación de la CPU**.
Finalmente, la pila se utiliza para **gestionar la memoria** de los procesos, aplicando el principio **LIFO (Last In, First Out)** para asignar y liberar bloques de manera ordenada.
En conjunto, estas estructuras y sus operaciones reproducen el comportamiento básico de un sistema operativo real, logrando una simulación funcional, eficiente y fácil de entender.

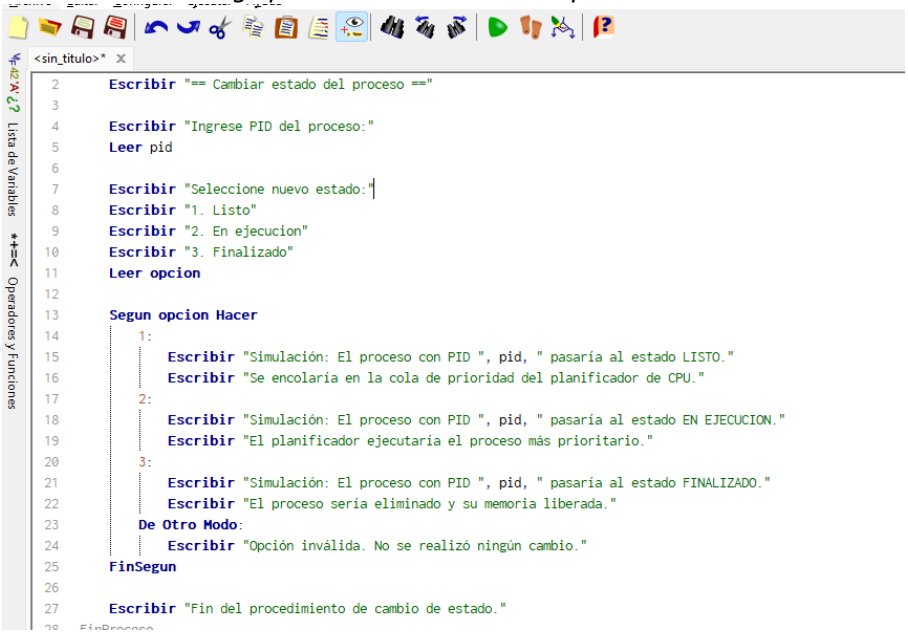
2. Algoritmos principales:

- *Pseudocódigo para agregar proceso.*



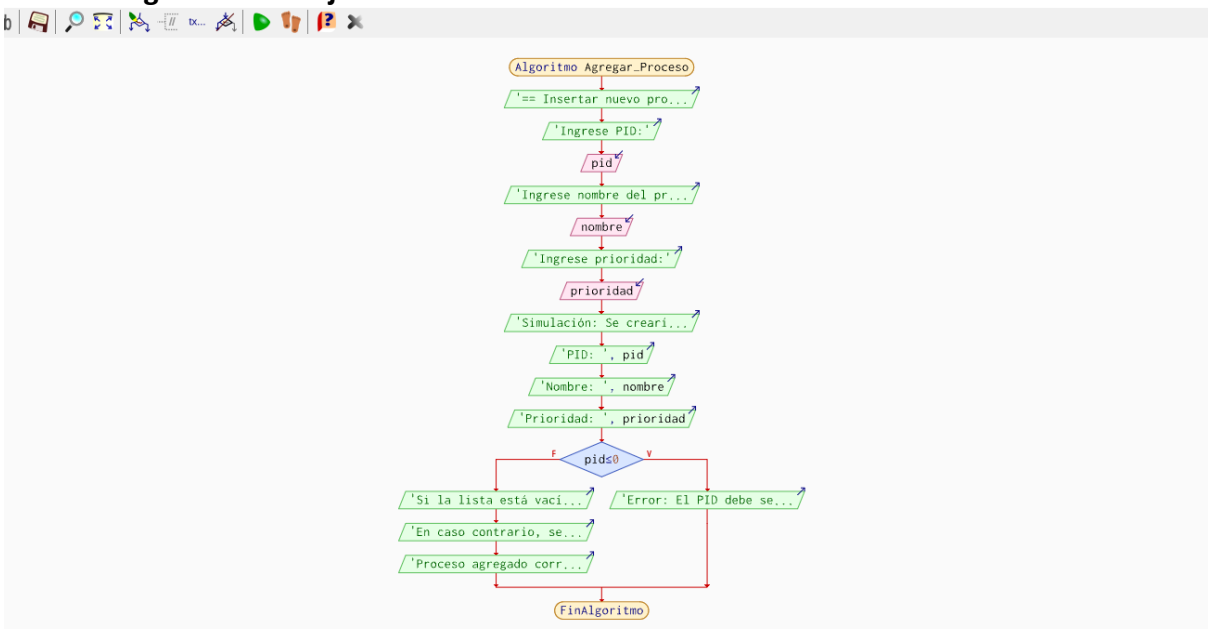
```
Proceso Agregar_Proceso
    Escribir "== Insertar nuevo proceso =="

    Escribir "Ingrese PID:"
    Leer pid

    Escribir "Ingrese nombre del proceso:"
    Leer nombre

    Escribir "Ingrese prioridad:"
    Leer prioridad

    Escribir "Simulación: Se crearía un nuevo nodo con los siguientes datos:"
    Escribir "PID: ", pid
    Escribir "Nombre: ", nombre
    Escribir "Prioridad: ", prioridad

    Escribir "Si la lista está vacía, el nuevo proceso se convierte en la cabeza."
    Escribir "En caso contrario, se enlaza al final de la lista enlazada."
    Escribir "El proceso fue agregado correctamente."
FinProceso
```

- *Pseudocódigo para cambiar estado del proceso.*

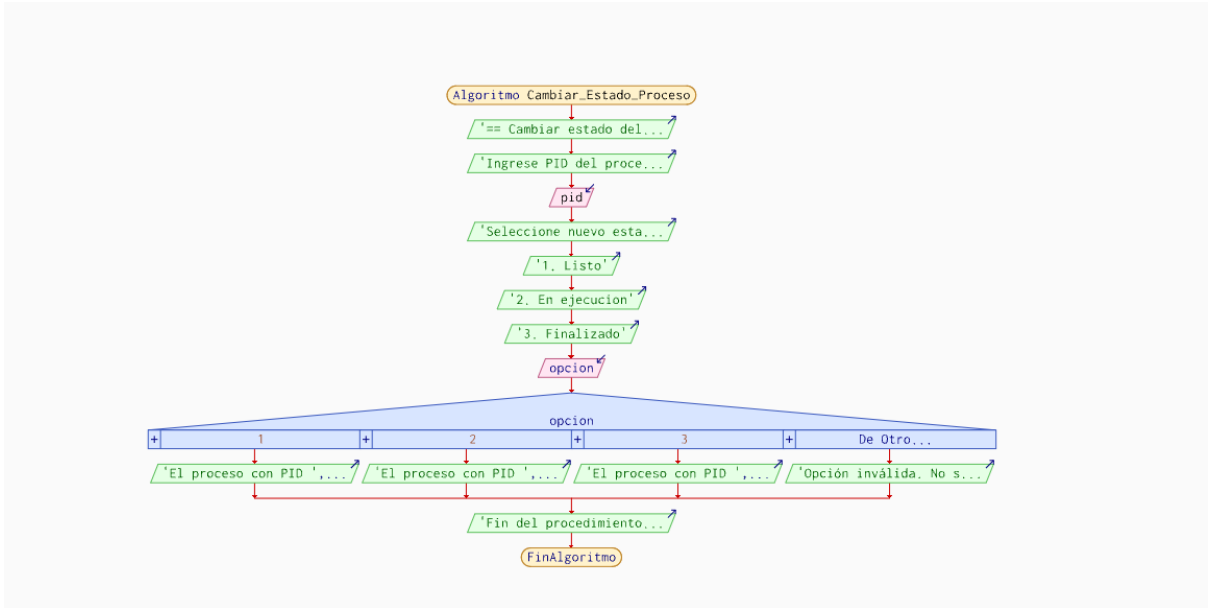


```
    Escribir "== Cambiar estado del proceso =="

    Escribir "Ingrese PID del proceso:"
    Leer pid

    Escribir "Seleccione nuevo estado:"
    Escribir "1. Listo"
    Escribir "2. En ejecucion"
    Escribir "3. Finalizado"
    Leer opcion

    Segun opcion Hacer
        1:
            Escribir "Simulación: El proceso con PID ", pid, " pasaría al estado LISTO."
            Escribir "Se encolaría en la cola de prioridad del planificador de CPU."
        2:
            Escribir "Simulación: El proceso con PID ", pid, " pasaría al estado EN EJECUCION."
            Escribir "El planificador ejecutaría el proceso más prioritario."
        3:
            Escribir "Simulación: El proceso con PID ", pid, " pasaría al estado FINALIZADO."
            Escribir "El proceso sería eliminado y su memoria liberada."
        De Otro Modo:
            Escribir "Opción inválida. No se realizó ningún cambio."
    FinSegun

    Escribir "Fin del procedimiento de cambio de estado."
```

3. Diagramas de Flujo



==========



4. Justificación del diseño:

Se eligió este diseño porque permite simular de forma clara y estructurada el funcionamiento básico de un sistema operativo, separando cada componente de gestión de procesos, planificación de CPU y gestión de memoria en módulos independientes dentro del menú principal.
Esta organización facilita la comprensión del sistema y mejora la interacción del usuario, ya que cada parte del programa cumple una función específica y puede probarse de manera individual.

El uso de **listas enlazadas** para el gestor de procesos permite almacenar, recorrer, insertar y eliminar elementos de forma dinámica, lo cual es ideal para manejar una cantidad variable de procesos sin ocupar memoria innecesaria.

Para el **planificador de CPU**, se empleó una **cola de prioridad**, que permite simular de manera realista cómo los sistemas operativos gestionan los procesos según su nivel de prioridad, asegurando que los más importantes sean atendidos primero. Por último, la **pila** utilizada en la **gestión de memoria** refleja el comportamiento LIFO (Last In, First Out), característico de la asignación y liberación de memoria en entornos reales, como las pilas de ejecución de programas. Gracias a esta combinación de estructuras, el sistema logra un diseño eficiente, modular y fácil de comprender, ideal para fines didácticos y de simulación.

Capítulo 3: Solución Final

1. Código limpio, bien comentado y estructurado.

```cpp
#include <iostream>     // Permite usar cout y cin
#include <string>       // Para manejar cadenas de texto
#include <cstdlib>      // Para usar system("cls") y system("pause")
#include <cctype>       // Para validar caracteres (isdigit, isspace)
using namespace std;    // Evita escribir std:: antes de cada función

//---------------- FUNCIONES AUXILIARES ----------------//

// Elimina los espacios en blanco al inicio y final de una cadena
string trim(const string &s) {
    size_t start = 0;  // Índice del primer carácter no vacío
    while (start < s.size() && isspace((unsigned char)s[start])) start++;
    if (start == s.size()) return "";  // Si toda la cadena está vacía
    size_t end = s.size() - 1;         // Índice del último carácter no vacío
    while (end > start && isspace((unsigned char)s[end])) end--;
    return s.substr(start, end - start + 1);  // Devuelve la parte sin espacios
}

// Convierte una cadena numérica en un número entero
int stringADecimal(const string &s) {
    int valor = 0;  // Variable acumuladora del número convertido
    for (size_t i = 0; i < s.size(); ++i)
        valor = valor * 10 + (s[i] - '0');  // Convierte carácter por carácter
    return valor;  // Retorna el valor convertido
}

// Lee un número entero positivo desde el teclado
int leerEnteroPositivo(const string &mensaje) {
    string entrada;  // Variable para almacenar la entrada del usuario
    while (true) {
        cout << mensaje;          // Muestra el mensaje
        getline(cin, entrada);    // Lee la entrada completa
        entrada = trim(entrada);  // Elimina espacios

        if (entrada.empty()) {    // Si está vacío
            cout << "Error: Ingrese un número entero positivo.\n";
            continue;
        }

        bool valido = true;       // Bandera para validar caracteres
        for (size_t i = 0; i < entrada.size(); ++i) {
            if (!isdigit((unsigned char)entrada[i])) {  // Si hay letras o símbolos
```

```
        bool valido = true;       // Bandera para validar caracteres
        for (size_t i = 0; i < entrada.size(); ++i) {
            if (!isdigit((unsigned char)entrada[i])) {  // Si hay letras o símbolos
                valido = false;
                break;
            }
        }

        if (!valido) {
            cout << "Error: Solo se permiten números.\n";
            continue;
        }

        int valor = stringADecimal(entrada);  // Convierte la cadena a número
        if (valor <= 0) {
            cout << "Error: El número debe ser mayor que 0.\n";
            continue;
        }

        return valor;  // Devuelve el número válido
    }
}

//---------------- ESTRUCTURAS PRINCIPALES ----------------//

// Cada proceso contiene su identificador (PID), nombre, prioridad y un puntero al siguiente
struct Proceso {
    int pid;                // Identificador único del proceso
    string nombre;          // Nombre descriptivo del proceso
    int prioridad;          // Nivel de prioridad (menor número = mayor prioridad)
    Proceso* siguiente;     // Enlace al siguiente proceso en la lista
};
Proceso* cabezaProcesos = NULL;   // Puntero al inicio de la lista de procesos

// Nodo para representar la cola de planificación de CPU
struct NodoCola {
    Proceso* proceso;       // Apunta al proceso asociado
    NodoCola* siguiente;    // Enlace al siguiente nodo en la cola
};
NodoCola* cabezaCola = NULL;      // Inicio de la cola (vacía al comienzo)

// Nodo para representar los bloques de memoria como una pila
struct BloqueMemoria {
    Proceso* proceso;        // Proceso que ocupa el bloque de memoria
    int tamanio;             // Tamaño asignado al bloque
    BloqueMemoria* siguiente;// Enlace al siguiente bloque
};
BloqueMemoria* topeMemoria = NULL; // Puntero al bloque superior de la pila

//---------------- FUNCIONES DE BÚSQUEDA ----------------//

// Busca un proceso por su PID en la lista enlazada
Proceso* buscarProcesoPorPID(int pid) {
    Proceso* actual = cabezaProcesos;  // Apunta al primer proceso
    while (actual != NULL) {
        if (actual->pid == pid) return actual;  // Si encuentra coincidencia
        actual = actual->siguiente;             // Avanza al siguiente
    }
    return NULL;  // Retorna NULL si no lo encuentra
}

// Verifica si un proceso ya está en la cola del planificador
bool estaEnCola(int pid) {
    NodoCola* actual = cabezaCola;  // Apunta al primer nodo
    while (actual != NULL) {
        if (actual->proceso->pid == pid) return true;  // Encontrado
        actual = actual->siguiente;                    // Avanza
    }
    return false;  // No encontrado
}

//---------------- GESTOR DE PROCESOS ----------------//

// Inserta un nuevo proceso al final de la lista enlazada
void insertarProceso() {
    cout << "\n--- Insertar nuevo proceso ---\n";

    int pid = leerEnteroPositivo("Ingrese el PID: ");  // PID único

    if (buscarProcesoPorPID(pid) != NULL) {            // Verifica duplicados
        cout << "Error: Ya existe un proceso con ese PID.\n";
        return;
    }
```

```
    string nombre;
    cout << "Ingrese el nombre del proceso: ";
    getline(cin, nombre);
    nombre = trim(nombre);

    while (nombre.empty()) {  // Evita nombres vacíos
        cout << "Error: El nombre no puede estar vacío.\n";
        cout << "Ingrese el nombre del proceso: ";
        getline(cin, nombre);
        nombre = trim(nombre);
    }

    int prioridad = leerEnteroPositivo("Ingrese la prioridad: ");  // Prioridad

    // Crea un nuevo nodo de proceso
    Proceso* nuevo = new Proceso;
    nuevo->pid = pid;
    nuevo->nombre = nombre;
    nuevo->prioridad = prioridad;
    nuevo->siguiente = NULL;

    // Inserta el proceso al final de la lista
    if (cabezaProcesos == NULL)
        cabezaProcesos = nuevo;
    else {
        Proceso* actual = cabezaProcesos;
        while (actual->siguiente != NULL) actual = actual->siguiente;
        actual->siguiente = nuevo;
    }

    cout << "Proceso agregado correctamente.\n";
}

// Elimina un proceso de la lista según su PID
void eliminarProceso() {
    cout << "\n--- Eliminar proceso ---\n";
    int pid = leerEnteroPositivo("Ingrese el PID a eliminar: ");

    if (cabezaProcesos == NULL) {  // Si no hay procesos
        cout << "No hay procesos registrados.\n";
        return;
    }
```

```
}

// Muestra todos los procesos de la lista
void mostrarProcesos() {
    cout << "\n--- Lista de procesos ---\n";
    if (cabezaProcesos == NULL) {
        cout << "No hay procesos registrados.\n";
        return;
    }
    Proceso* actual = cabezaProcesos;
    while (actual != NULL) {
        cout << "PID: " << actual->pid
             << " | Nombre: " << actual->nombre
             << " | Prioridad: " << actual->prioridad << endl;
        actual = actual->siguiente;
    }
}

//---------------- PLANIFICADOR DE CPU ----------------//

// Encola un proceso según su prioridad
void encolarProcesoEnPlanificador() {
    cout << "\n--- Encolar proceso ---\n";
    int pid = leerEnteroPositivo("Ingrese el PID: ");
    Proceso* p = buscarProcesoPorPID(pid);

    if (p == NULL) {
        cout << "Error: No existe un proceso con ese PID.\n";
        return;
    }
    if (estaEnCola(pid)) {
        cout << "Error: El proceso ya está en la cola.\n";
        return;
    }

    NodoCola* nuevo = new NodoCola;
    nuevo->proceso = p;
    nuevo->siguiente = NULL;

    // Inserta ordenado por prioridad
    if (cabezaCola == NULL || cabezaCola->proceso->prioridad > p->prioridad) {
        nuevo->siguiente = cabezaCola;
```

```
    if (cabezaCola == NULL) {
        cout << "La cola está vacía.\n";
        return;
    }

    NodoCola* temp = cabezaCola;
    cabezaCola = cabezaCola->siguiente;
    cout << "Ejecutando proceso PID: " << temp->proceso->pid
         << " | Nombre: " << temp->proceso->nombre
         << " | Prioridad: " << temp->proceso->prioridad << endl;
    delete temp;
}

// Muestra el contenido de la cola
void mostrarColaPlanificador() {
    cout << "\n--- Cola de planificación ---\n";
    if (cabezaCola == NULL) {
        cout << "La cola está vacía.\n";
        return;
    }
    NodoCola* actual = cabezaCola;
    while (actual != NULL) {
        cout << "PID: " << actual->proceso->pid
             << " | Nombre: " << actual->proceso->nombre
             << " | Prioridad: " << actual->proceso->prioridad << endl;
        actual = actual->siguiente;
    }
}

//---------------- GESTOR DE MEMORIA ----------------//

// Asigna memoria a un proceso (operación push)
void asignarMemoria() {
    cout << "\n--- Asignar memoria ---\n";
    int pid = leerEnteroPositivo("Ingrese el PID: ");
    Proceso* p = buscarProcesoPorPID(pid);
    if (p == NULL) {
        cout << "Error: No existe un proceso con ese PID.\n";
        return;
    }

    int tamanio = leerEnteroPositivo("Ingrese el tamaño de memoria: ");
```

```
    if (p == NULL) {
        cout << "Error: No existe un proceso con ese PID.\n";
        return;
    }

    int tamanio = leerEnteroPositivo("Ingrese el tamaño de memoria: ");
    BloqueMemoria* nuevo = new BloqueMemoria;
    nuevo->proceso = p;
    nuevo->tamanio = tamanio;
    nuevo->siguiente = topeMemoria;
    topeMemoria = nuevo;

    cout << "Memoria asignada correctamente.\n";
}

// Libera el último bloque asignado (operación pop)
void liberarMemoria() {
    cout << "\n--- Liberar memoria ---\n";
    if (topeMemoria == NULL) {
        cout << "No hay memoria para liberar.\n";
        return;
    }

    BloqueMemoria* temp = topeMemoria;
    topeMemoria = topeMemoria->siguiente;
    cout << "Memoria liberada del proceso PID: " << temp->proceso->pid
         << " | Tamaño: " << temp->tamanio << endl;
    delete temp;
}

// Muestra los bloques de memoria actuales
void estadoMemoria() {
    cout << "\n--- Estado de memoria ---\n";
    if (topeMemoria == NULL) {
        cout << "La memoria está libre.\n";
        return;
    }
    BloqueMemoria* actual = topeMemoria;
    while (actual != NULL) {
        cout << "PID: " << actual->proceso->pid
             << " | Nombre: " << actual->proceso->nombre
```

```
            do {
                cout << "\n--- PLANIFICADOR DE CPU ---\n";
                cout << "[1] Encolar proceso\n";
                cout << "[2] Ejecutar proceso\n";
                cout << "[3] Mostrar cola\n";
                cout << "[4] Volver\n";
                opcionSecundaria = leerEnteroPositivo("Opción: ");
                if (opcionSecundaria == 1) encolarProcesoEnPlanificador();
                else if (opcionSecundaria == 2) desencolarYEjecutarProceso();
                else if (opcionSecundaria == 3) mostrarColaPlanificador();
                system("pause");
                system("cls");
            } while (opcionSecundaria != 4);
            break;

        case 3:
            do {
                cout << "\n--- GESTOR DE MEMORIA ---\n";
                cout << "[1] Asignar memoria\n";
                cout << "[2] Liberar memoria\n";
                cout << "[3] Mostrar estado\n";
                cout << "[4] Volver\n";
                opcionSecundaria = leerEnteroPositivo("Opción: ");
                if (opcionSecundaria == 1) asignarMemoria();
                else if (opcionSecundaria == 2) liberarMemoria();
                else if (opcionSecundaria == 3) estadoMemoria();
                system("pause");
                system("cls");
            } while (opcionSecundaria != 4);
            break;
        }
    } while (opcionPrincipal != 4);
    cout << "Saliendo del programa...\n";
}

// Función principal: punto de inicio del programa
int main() {
    menu();  // Llama al menú principal
    return 0;
}
```

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

Validación de PID (Gestor de Procesos)

```
========== MENU PRINCIPAL ==========
[1] Gestor de Procesos
[2] Planificador de CPU
[3] Gestor de Memoria
[4] Salir
Seleccione una opci%n: 1

--- GESTOR DE PROCESOS ---
[1] Insertar proceso
[2] Eliminar proceso
[3] Mostrar procesos
[4] Volver
Opci%n: 1

--- Insertar nuevo proceso ---
Ingrese el PID: abc
Error: Solo se permiten n·meros.
Ingrese el PID: -5
Error: Solo se permiten n·meros.
Ingrese el PID:
```

Validación de nombre vacío

```
Ingrese el PID: 31412
Ingrese el nombre del proceso:
Error: El nombre no puede estar vacÝo.
Ingrese el nombre del proceso:
```

Proceso insertado correctamente

```
--- Insertar nuevo proceso ---
Ingrese el PID: 1
Ingrese el nombre del proceso: Chrome
Ingrese la prioridad: 3
Proceso agregado correctamente.
Presione una tecla para continuar . . .
```

Evitar PID repetido

```
--- GESTOR DE PROCESOS ---
[1] Insertar proceso
[2] Eliminar proceso
[3] Mostrar procesos
[4] Volver
Opci¾n: 1

--- Insertar nuevo proceso ---
Ingrese el PID: 1
Error: Ya existe un proceso con ese PID.
Presione una tecla para continuar . . .
```

Mostrar lista de procesos

```
--- GESTOR DE PROCESOS ---
[1] Insertar proceso
[2] Eliminar proceso
[3] Mostrar procesos
[4] Volver
Opci¾n: 3

--- Lista de procesos ---
PID: 1 | Nombre: Chrome | Prioridad: 3
Presione una tecla para continuar . . .
```

Encolar proceso por prioridad (Planificador de CPU)

```
========== MENU PRINCIPAL ==========
[1] Gestor de Procesos
[2] Planificador de CPU
[3] Gestor de Memoria
[4] Salir
Seleccione una opci%n: 2

--- PLANIFICADOR DE CPU ---
[1] Encolar proceso
[2] Ejecutar proceso
[3] Mostrar cola
[4] Volver
Opci%n: 3

--- Cola de planificaci%n ---
PID: 1 | Nombre: Chrome | Prioridad: 3
Presione una tecla para continuar
```

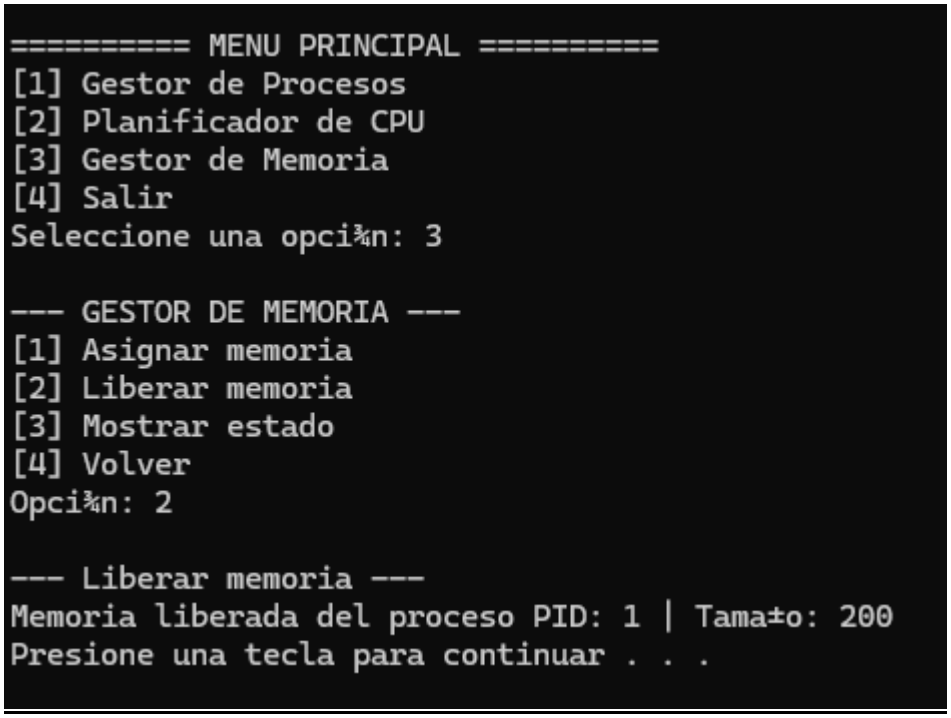
Desencolar y ejecutar proceso

```
--- PLANIFICADOR DE CPU ---
[1] Encolar proceso
[2] Ejecutar proceso
[3] Mostrar cola
[4] Volver
Opci%n: 2

--- Desencolar y ejecutar ---
Ejecutando proceso PID: 1 | Nombre: Chrome | Prioridad: 3
Presione una tecla para continuar . . .
```

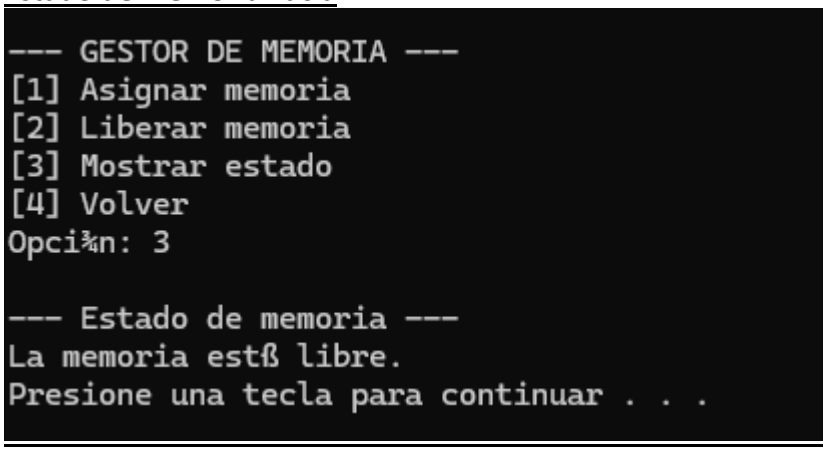
Asignar memoria

```
========== MENU PRINCIPAL ==========
[1] Gestor de Procesos
[2] Planificador de CPU
[3] Gestor de Memoria
[4] Salir
Seleccione una opci%n: 3

--- GESTOR DE MEMORIA ---
[1] Asignar memoria
[2] Liberar memoria
[3] Mostrar estado
[4] Volver
Opci%n: 1

--- Asignar memoria ---
Ingrese el PID: 1
Ingrese el tama±o de memoria: 200
Memoria asignada correctamente.
Presione una tecla para continuar . . .
```

Liberar memoria



Estado de memoria vacía



3. Manual de usuario

Descripción general

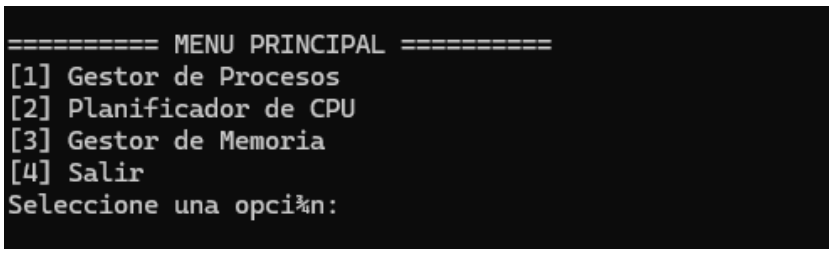
El programa simula la gestión de procesos, la planificación de CPU y la asignación de memoria de un sistema operativo. Permite registrar procesos, encolados según su prioridad y asignar o liberar bloques de memoria. La interfaz es totalmente en consola, sencilla y con menús interactivos.

Requisitos del sistema

- Sistema operativo: **Windows**
- Compilador: **Code::Blocks**, **Dev-C++** o cualquier IDE compatible con C++
- Consola habilitada para mostrar texto (CMD o terminal del IDE)

Instrucciones de ejecución

1. **Compila el archivo `GestionProcesos.cpp`**
2. **Ejecuta el programa desde tu IDE o la terminal.**
3. **En la pantalla principal aparecerá el siguiente menú:**



Descripción de menús y opciones

===MENÚ PRINCIPAL===

Contiene tres módulos principales:

1. **Gestor de Procesos – Permite crear, eliminar y visualizar procesos.**
2. **Planificador de CPU – Gestiona la cola de procesos por prioridad.**
3. **Gestor de Memoria – Simula la asignación y liberación de memoria.**
4. **Salir – Finaliza el programa.**

Gestor de Procesos

Permite administrar los procesos registrados.

Opciones:

- **[1] Insertar nuevo proceso**
  - **Pide un PID, un nombre y una prioridad.**
  - **Valida que el PID sea entero positivo y que no esté repetido.**
- **[2] Eliminar proceso**
  - **Pide el PID del proceso y lo elimina de la lista.**
- **[3] Mostrar todos los procesos**
  - **Muestra la lista completa con PID, nombre y prioridad.**
- **[4] Volver al menú principal**

Planificador de CPU

Simula la cola de procesos ordenada por prioridad.

Opciones:

- **[1] Encolar proceso**
  - **Agrega un proceso a la cola de planificación según su prioridad.**
- **[2] Desencolar y ejecutar proceso**
  - **Quita el proceso con mayor prioridad (menor número).**
- **[3] Mostrar cola actual**
  - **Muestra la cola completa ordenada por prioridad.**
- **[4] Volver al menú principal**

Gestor de Memoria

Simula la asignación y liberación tipo pila (LIFO).

Opciones:

- **[1] Asignar memoria (push)**
  - **Ingresa el PID y el tamaño de memoria a asignar.**
- **[2] Liberar memoria (pop)**
  - **Libera el último bloque asignado.**
- **[3] Ver estado actual de la memoria**
  - **Muestra los bloques asignados y sus tamaños.**
- **[4] Volver al menú principal**

Validaciones

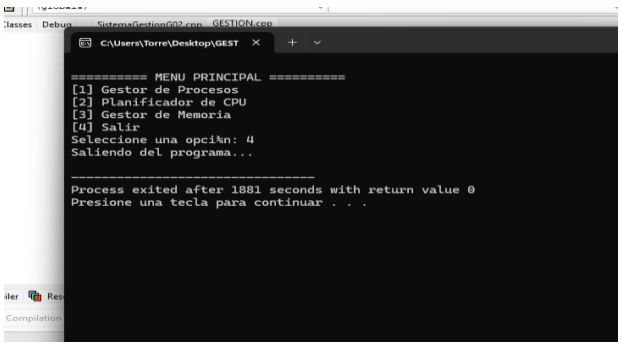
- **Solo se aceptan números positivos para PID, prioridad y tamaño de memoria.**
- **Si se ingresa texto o caracteres no válidos, el programa muestra un mensaje de error y vuelve a pedir los datos.**
- **No se permite agregar procesos con PID duplicado.**
- **No se permite asignar memoria a procesos inexistentes.**

Ejemplo de uso

1. **Crear tres procesos: Chrome, Word y Excel.**
2. **Encolar los procesos con prioridades distintas (1, 2 y 3).**
3. **Mostrar la cola para verificar el orden.**
4. **Asignar memoria a Chrome.**
5. **Liberar la memoria y observar el cambio.**

Salida del programa

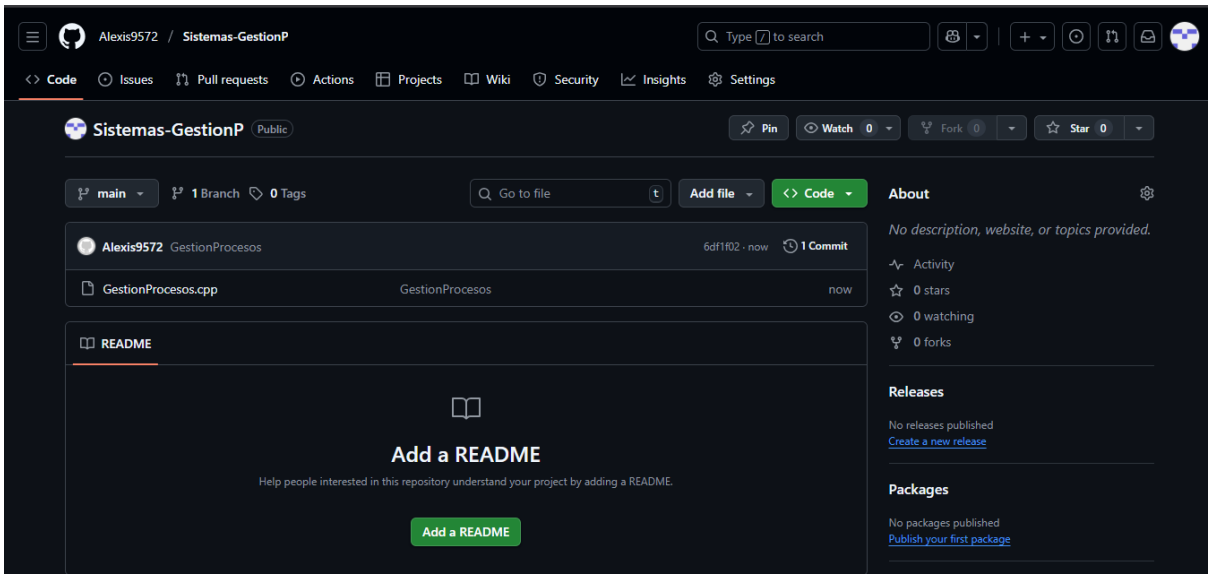
Selecciona la opción [4] Salir desde el menú principal.
El programa mostrará:



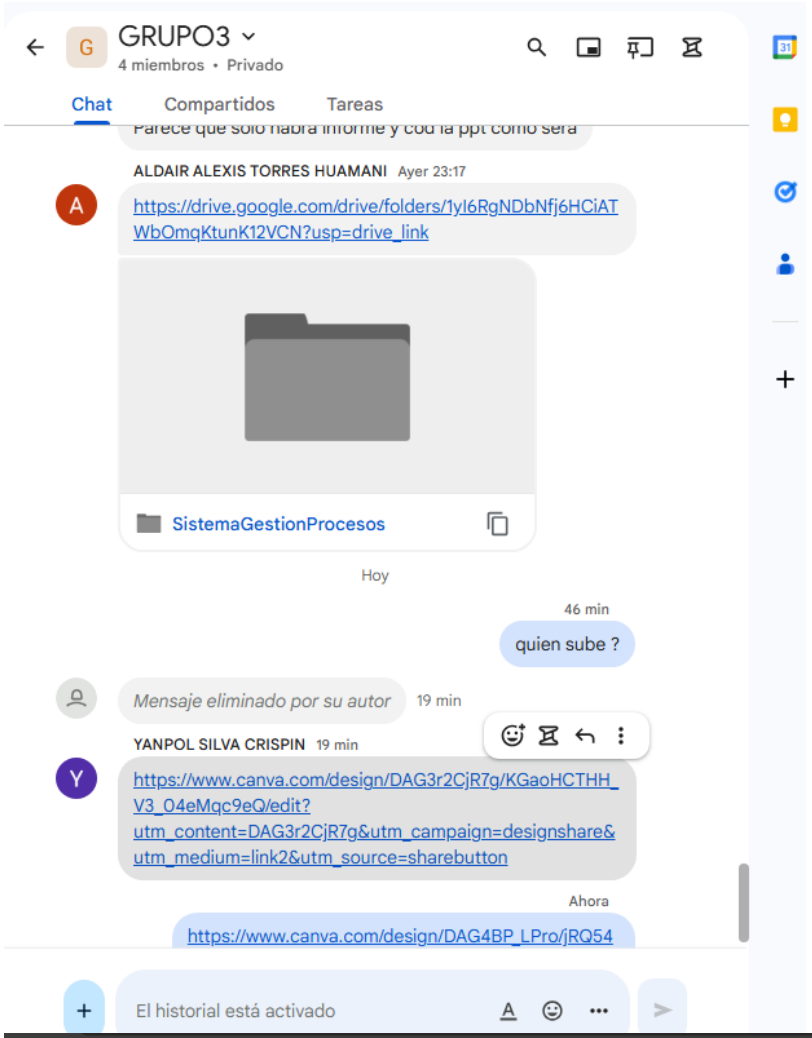
Capítulo 4: Evidencias de Trabajo en Equipo

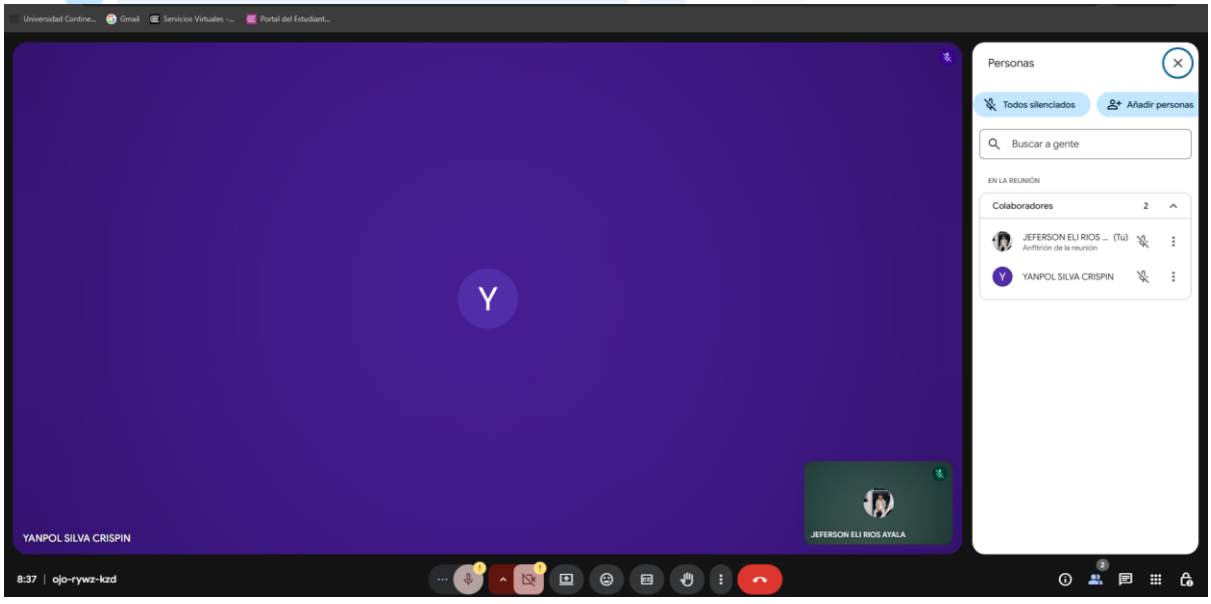
3. **Repositorio con Control de Versiones (Capturas de Pantalla)**

- **Torres Huamani Aldair Alexis**



https://github.com/Alexis9572/Sistemas-GestionP


GRUPO3
4 miembros • Privado
Chat
Compartidos
Tareas
ALDAIR ALEXIS TORRES HUAMANI Ayer 23:17
https://drive.google.com/drive/folders/1yI6RgNDbNfj6HCiATWbOmqKtunK12VCN?usp=drive_link
SistemaGestionProcesos
Hoy
46 min
quien sube ?
Mensaje eliminado por su autor
19 min
YANPOL SILVA CRISPIN 19 min
https://www.canva.com/design/DAG3r2CjR7g/KGaoHCTHH_V3_04eMqc9eQ/edit?utm_content=DAG3r2CjR7g&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton
Ahora
https://www.canva.com/design/DAG4BP_LPro/jRQ54
El historial está activado


Personas
Todos silenciados
Añadir personas
Buscar a gente
EN LA REUNIÓN
Colaboradores 2
JEFERSON ELI RIOS ... (Tú)
YANPOL SILVA CRISPIN
YANPOL SILVA CRISPIN
JEFERSON ELI RIOS AYALA
8:37 | ojo-rywz-kzd