UNIVERSIDAD CONTINENTAL

FACULTAD DE INGENIERÍA

INFORME DE ESTRUCTURA DE DATOS

NRC: 29901

Asignatura: Estructura de datos
Docente: Osorio Contreras Rosario Delia

Proyecto Grupal

"SISTEMA DE GESTIÓN DE PROCESOS EN UN SISTEMA OPERATIVO"

INTEGRANTES:

- García Betancourt Israel Jedidias
- Leon Armas Luis Aram
- Ramírez Quillatupa Juan Diego
- Veliz Durand Vieri Del Piero

2025 - 10
Huancayo - Perú

ÍNDICE

Sistema de gestión de procesos en un sistema operativo

CAPÍTULO 1: Análisis del Problema

1. Descripción del problema

En un sistema operativo, la gestión de procesos, la planificación de la CPU y la asignación de memoria son tareas fundamentales que deben realizarse de manera eficiente y organizada. Sin embargo, simular este comportamiento de manera didáctica y comprensible para estudiantes representa un reto importante, especialmente cuando se deben aplicar conceptos de estructuras de datos dinámicas.

El problema radica en cómo modelar y gestionar estos componentes (procesos, cola de ejecución y memoria) sin el uso de estructuras de datos predefinidas, asegurando al mismo tiempo que el sistema permita realizar operaciones básicas como insertar, eliminar, buscar y modificar elementos. Además, debe incluir mecanismos que simulen la planificación por prioridad y el uso dinámico de memoria en un entorno controlado.

Este sistema busca resolver la necesidad de contar con una herramienta educativa que permita simular de forma práctica y sencilla la gestión de procesos en un entorno tipo sistema operativo, utilizando listas enlazadas, colas y pilas implementadas manualmente.

2. Requerimientos del sistema

2.1. Funcionales

- El sistema debe permitir la creación de nuevos procesos con datos como nombre, prioridad, tiempo de llegada y tamaño en memoria.
- El sistema debe manejar la inserción, eliminación, búsqueda y modificación de procesos utilizando listas enlazadas.
- Debe simular la planificación de procesos en una cola según su prioridad.
- Debe permitir mostrar el estado actual de la cola de ejecución y de la memoria.
- El sistema debe simular la asignación y liberación dinámica de memoria utilizando una estructura tipo lista enlazada.
- El sistema debe permitir ejecutar procesos en orden de prioridad hasta que la cola esté vacía.

2.2. No funcionales

- El sistema debe estar desarrollado en lenguaje C++.
- No se deben utilizar estructuras de datos predefinidas de la STL (Standard Template Library).
- La interfaz será por consola, permitiendo al usuario interactuar mediante menús.
- El código debe estar modularizado y documentado para facilitar su mantenimiento.
- La simulación debe ejecutarse de manera clara y comprensible, con tiempos de respuesta adecuados.

3. Estructuras de datos propuestos

3.1. Lista enlazada para el gestor de proceso

Se implementó una lista enlazada simple para almacenar los procesos registrados. Cada nodo (estructura `Proceso`) contiene:

- **PID**: Identificador único del proceso
- **Nombre**: Nombre descriptivo del proceso
- **Prioridad**: Nivel de prioridad asignado (entero positivo)
- **Puntero al siguiente proceso**: Para mantener el orden en la lista

Esta estructura permite añadir, eliminar y recorrer procesos de forma dinámica y eficiente, ideal para el manejo flexible de la lista general de procesos.

3.2. Cola por prioridad para el planificador de CPU

Para simular la planificación por prioridad, se utiliza una cola con prioridad implementada mediante una lista enlazada ordenada (`NodoCola`). Cada nodo apunta a un proceso y se inserta en la cola de manera que los procesos con menor valor numérico en prioridad (mayor prioridad) están al frente.

Esta estructura permite:

- Insertar procesos ordenados según su prioridad
- Desencolar y ejecutar el proceso de mayor prioridad
- Visualizar la cola en cualquier momento

Con esta cola se simula el comportamiento de un planificador de CPU basado en prioridades.

3.3. Pila para el gestor de memoria

El gestor de memoria está basado en una pila (estructura LIFO) implementada con una lista enlazada (BloqueMemoria). Cada nodo representa un bloque de memoria asignado a un proceso y contiene:

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

Esta estructura permite:

- Asignar memoria (operación push) agregando bloques al tope de la pila
- Liberar memoria (operación pop) retirando bloques desde el tope
- Visualizar el estado actual de la memoria asignada

La pila simula una gestión básica de memoria dinámica donde los últimos bloques asignados son los primeros en liberarse.

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, simulando con precisión el comportamiento de componentes reales de un sistema operativo.

4.1. Lista enlazada para el gestor de procesos

Se eligió una lista enlazada simple para almacenar los procesos registrados en el sistema. Esta estructura es especialmente adecuada porque:

- **Inserción dinámica:** Permite agregar procesos al final sin necesidad de redimensionar una estructura, como ocurriría en un arreglo.
- **Eliminación eficiente:** Un proceso puede eliminarse sin necesidad de mover elementos, simplemente reajustando punteros.
- **Recorrido sencillo:** Puede recorrerse de forma lineal para mostrar todos los procesos activos.
- **Memoria eficiente:** No requiere reservar espacio fijo, lo que la hace adecuada para situaciones donde la cantidad de procesos puede variar.

Esto hace que la lista enlazada sea ideal para gestionar un conjunto dinámico y cambiante de procesos, cumpliendo perfectamente su rol en el Gestor de Procesos.

4.2. Cola por prioridad para el Planificador de CPU

Para el planificador, se utilizó una cola de prioridad, implementada como una lista enlazada ordenada por prioridad ascendente (menor número = mayor prioridad). Esta estructura es apropiada porque:

- **Inserción ordenada**: Al encolar un proceso, se ubica automáticamente en la posición que le corresponde según su prioridad.
- **Desencolado eficiente**: El proceso con mayor prioridad siempre se encuentra al principio, facilitando su ejecución inmediata.
- **Simula algoritmos reales**: Este comportamiento imita a muchos planificadores de sistemas operativos reales, donde se ejecuta el proceso más prioritario disponible.

Esto proporciona una simulación realista y ordenada del planificador de CPU por prioridad, manteniendo la eficiencia sin estructuras complejas como montículos (heaps).

4.3. Pila para el gestor de memoria

La gestión de memoria se realiza mediante una pila, que sigue el principio de Last-In, First-Out (LIFO). Esta decisión se justifica por:

- **Asignación y liberación rápida**: Asignar y liberar memoria es muy eficiente, ya que solo se manipula el tope de la pila.
- **Simula comportamiento real**: En muchas arquitecturas y lenguajes de programación, el uso de memoria en la pila sigue un patrón LIFO (por ejemplo, llamadas a funciones).
- **Simplicidad en la gestión**: No requiere recorrer ni ordenar, lo que facilita la implementación y reduce errores.

Esto hace que la pila enlazada sea ideal para simular una administración de memoria básica, especialmente útil en un entorno educativo o de simulación como el tuyo.

4.4. Simplicidad en el recorrido

El uso de listas, colas y pilas permite recorrer y manejar los procesos de forma simple y ordenada.

- **Listas**: permiten insertar, buscar y eliminar procesos fácilmente.
- **Colas**: gestionan la planificación en orden, sin estructuras adicionales.
- **Pilas**: simulan la memoria con acceso rápido al último bloque asignado.

Esto simplifica la simulación y mantiene el código claro y eficiente.

CAPÍTULO 2: Diseño de la Solución

1. Descripción de estructuras de datos y operaciones

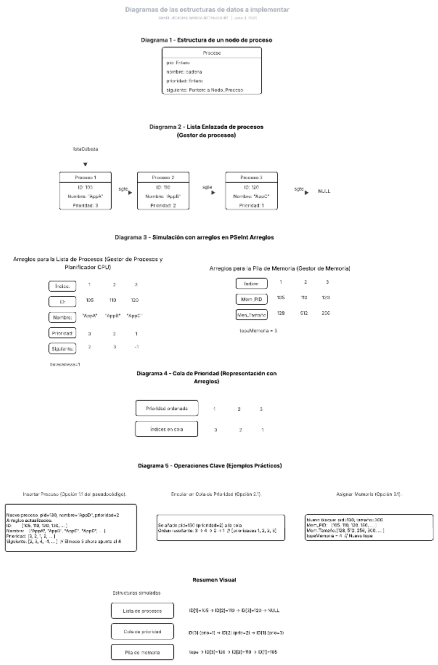
Se utilizaron tres estructuras de datos principales:

- **Lista enlazada**: para almacenar los procesos. Permite insertar, eliminar, buscar y recorrer procesos dinámicamente sin tamaño fijo, ideal para mantener una colección modificable.
- **Cola de prioridad**: para el planificador de CPU. Los procesos se insertan ordenados por prioridad (menor número = mayor prioridad) y se atienden en ese orden, simulando una planificación justa y eficiente.
- **Pila**: para la gestión de memoria. Sigue el esquema LIFO, donde el último bloque asignado es el primero en liberarse, tal como ocurre con las pilas de activación en memoria real.

Las operaciones realizadas incluyen inserción, eliminación, búsqueda, modificación y recorrido en cada estructura, simulando cómo un sistema operativo maneja procesos y recursos.

2. Algoritmos principales

2.1. Diagrama de las estructuras a implementar



Enlace para ver el diagrama completo:
https://lucid.app/lucidchart/457c69ba-9151-4eca-a34e-6c8fec21cab6/edit?viewport_loc=-1564%2C-455%2C5539%2C2722%2C0_0&invitationId=inv_95398363-859b-4f55-9687-98137b853153

2.2. Pseudocódigo para agregar proceso

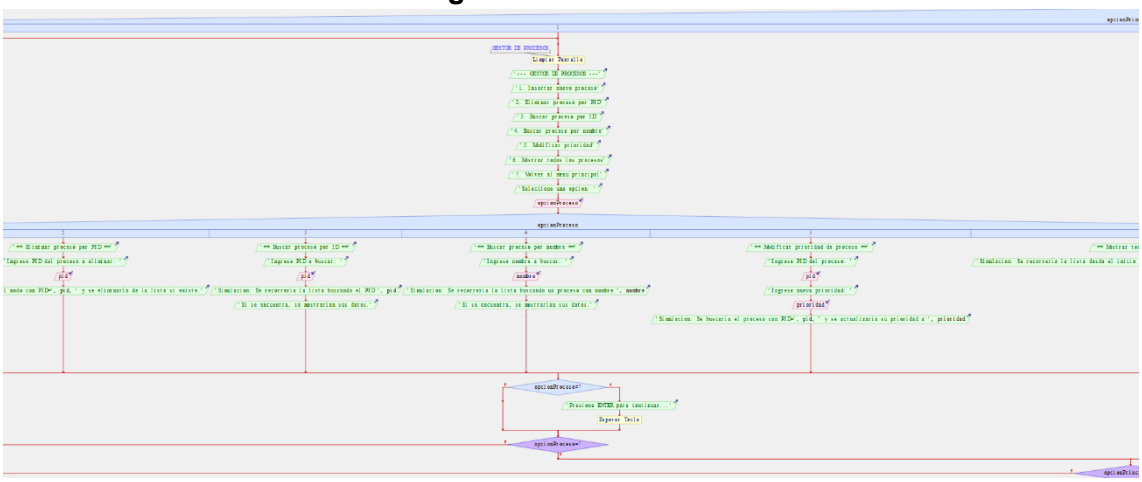
```
Escribir '== Insertar nuevo proceso =='
Escribir 'Ingrese PID: '
Leer pid
Escribir 'Ingrese nombre del proceso: '
Leer nombre
Escribir 'Ingrese prioridad: '
Leer prioridad
Escribir 'Simulacion: Se crearia un nodo con PID=', pid, ', nombre=', nombre, ', prioridad=', prioridad
Escribir 'Este nodo se agregaría al final de la lista enlazada de procesos.'
```

2.3. Pseudocódigo para cambiar estado del proceso

```
1:
    Escribir '== Asignar memoria a proceso =='
    Escribir 'Ingrese PID del proceso: '
    Leer pid
    Escribir 'Ingrese tamaño de memoria: '
    Leer tamanio
    Escribir 'Simulacion: Se crearía un nuevo bloque de memoria de tamaño ', tamanio
    Escribir 'y se agregaría al tope de la pila de memoria.'
2:
    Escribir '== Liberar memoria (POP) =='
    Escribir 'Simulacion: Se eliminaría el bloque de memoria que está en el tope de la pila'
    Escribir 'y se liberaría el espacio asignado al proceso correspondiente.'
```

3. Diagramas de flujo

Parte del diagrama Gestor de Procesos



Enlace para ver el diagrama de flujo completo:
https://sistemagestionprocesosg2.netlify.app/

4. Justificación del diseño

Elegimos este diseño porque permite simular de forma clara y estructurada el funcionamiento básico de un sistema operativo, separando cada componente (gestión de procesos, planificación de CPU y memoria) en menús independientes. Esta

organización facilita la comprensión del sistema y mejora la experiencia del usuario al permitirle interactuar de manera sencilla con cada módulo.

Se utilizaron listas para gestionar procesos porque permiten almacenar y recorrer múltiples elementos dinámicamente, ideal para buscar, insertar o eliminar procesos. Para la planificación de CPU se eligió una cola de prioridad, ya que refleja cómo los sistemas reales ejecutan primero los procesos más importantes (mayor prioridad). Finalmente, se usó una pila en la gestión de memoria para simular el comportamiento tipo LIFO (Last In, First Out), típico de la asignación y liberación de memoria en sistemas reales.

CAPÍTULO 3: Solución Final

1. Código limpio, bien comentado y estructurado

```cpp
#include <iostream>     // Librería para operaciones de entrada y salida (cin, cout)
#include <string>       // Librería para usar la clase string
#include <cstdlib>      // Librería para usar system("cls") y system("pause")

using namespace std;    // Permite usar nombres estándar sin anteponer "std::"

// Definición de estructura para representar un proceso
struct Proceso {
    int pid;                // Identificador único del proceso
    string nombre;          // Nombre del proceso
    int prioridad;          // Nivel de prioridad del proceso
    Proceso* siguiente;     // Puntero al siguiente proceso (lista enlazada)
};

// Puntero global al inicio de la lista de procesos
Proceso* cabezaProcesos = NULL;

// Estructura para nodo de la cola de planificación
struct NodoCola {
    Proceso* proceso;       // Puntero al proceso contenido
    NodoCola* siguiente;    // Puntero al siguiente nodo en la cola
};

// Puntero global al inicio de la cola
NodoCola* cabezaCola = NULL;

// Estructura para representar un bloque de memoria asignada (pila)
struct BloqueMemoria {
    int pid;                    // PID del proceso que usa la memoria
    int tamanio;                // Tamaño de la memoria asignada
    BloqueMemoria* siguiente;   // Puntero al siguiente bloque (pila)
};

// Puntero al tope de la pila de memoria
BloqueMemoria* topeMemoria = NULL;
```

```cpp
//---------------- FUNCIONES AUXILIARES ----------------//

// Busca un proceso por su PID en la lista enlazada
Proceso* buscarProcesoPorPID(int pid) {
    Proceso* actual = cabezaProcesos; // Comienza desde la cabeza de la lista
    while (actual != NULL) {
        if (actual->pid == pid)      // Si se encuentra el PID
            return actual;           // Retorna el puntero al proceso
        actual = actual->siguiente;  // Avanza al siguiente nodo
    }
    return NULL; // No se encontró el proceso
}

// Verifica si un proceso con el PID ya está en la cola de planificación
bool estaEnCola(int pid) {
    NodoCola* actual = cabezaCola;   // Comienza desde la cabeza de la cola
    while (actual != NULL) {
        if (actual->proceso->pid == pid) // Si encuentra el PID en algún nodo
            return true;                 // Ya está en la cola
        actual = actual->siguiente;      // Avanza al siguiente nodo
    }
    return false; // No se encontró el proceso en la cola
}
```

```cpp
//---------------- GESTOR DE PROCESOS ----------------//

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

    int pid;
    do {
        cout << "Ingrese PID (entero positivo): ";
        cin >> pid;                          // Solicita el PID
        if (pid <= 0) {
            cout << "Error: El PID debe ser un número entero positivo.\n";
        } else if (buscarProcesoPorPID(pid) != NULL) {
            cout << "Error: Ya existe un proceso con ese PID.\n";
            pid = -1;                        // Repetir entrada si ya existe
        }
        cin.clear();                         // Limpia el estado de cin
    } while (pid <= 0);                      // Repite hasta que el PID sea válido

    string nombre;
    cin.ignore();                            // Limpia el buffer del salto de línea
    do {
        cout << "Ingrese nombre del proceso: ";
        getline(cin, nombre);                // Lee el nombre completo con espacios
        if (nombre.empty()) {
            cout << "Error: La cadena no puede estar vacía.\n";
        }
    } while (nombre.empty());                // Repite hasta que el nombre no esté vacío

    int prioridad;
    do {
        cout << "Ingrese prioridad (entero positivo): ";
        cin >> prioridad;
        if (prioridad <= 0) {
            cout << "Error: La prioridad debe ser un número entero positivo.\n";
        }
        cin.clear();
    } while (prioridad <= 0);                // Repite hasta que la prioridad sea válida

    Proceso* nuevo = new Proceso;            // Crea un nuevo proceso dinámicamente
    nuevo->pid = pid;                        // Asigna los valores
    nuevo->nombre = nombre;
    nuevo->prioridad = prioridad;
    nuevo->siguiente = NULL;

    if (cabezaProcesos == NULL) {            // Si la lista está vacía
        cabezaProcesos = nuevo;              // El nuevo proceso es el primero
    } else {
        Proceso* actual = cabezaProcesos;
        while (actual->siguiente != NULL) actual = actual->siguiente;
        actual->siguiente = nuevo;           // Agrega el proceso al final de la lista
    }

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

// Elimina un proceso de la lista enlazada por PID
void eliminarProceso() {
    cout << "--- Eliminar proceso ---\n";
    int pid;
    do {
        cout << "Ingrese PID a eliminar: ";
        cin >> pid;
        if (pid <= 0) {
            cout << "Error: Debe ingresar un número entero positivo.\n";
        }
        cin.clear();
    } while (pid <= 0);                      // Repite hasta que el PID sea válido

    if (cabezaProcesos == NULL) {            // Lista vacía
        cout << "No hay procesos para eliminar.\n";
        return;
    }

    if (cabezaProcesos->pid == pid) {        // El proceso a eliminar es el primero
        Proceso* temp = cabezaProcesos;
        cabezaProcesos = cabezaProcesos->siguiente;
        delete temp;                         // Libera la memoria
        cout << "Proceso eliminado.\n";
        return;
    }

    Proceso* actual = cabezaProcesos;
    while (actual->siguiente != NULL && actual->siguiente->pid != pid)
        actual = actual->siguiente;

    if (actual->siguiente != NULL) {         // Se encontró el proceso
        Proceso* temp = actual->siguiente;
        actual->siguiente = temp->siguiente; // Lo elimina de la lista
        delete temp;                         // Libera la memoria
        cout << "Proceso eliminado.\n";
    } else {
        cout << "Proceso no encontrado.\n";  // No se encontró el PID
    }
}
```

```cpp
// Muestra todos los procesos registrados
void mostrarProcesos() {
    cout << "--- Lista general de procesos ---\n";

    // Verifica si no hay procesos en la lista general
    if (cabezaProcesos == NULL) {
        cout << "No hay procesos registrados.\n";
        return;
    }

    // Recorre la lista de procesos e imprime sus datos
    Proceso* actual = cabezaProcesos;
    while (actual != NULL) {
        cout << "PID: " << actual->pid << ", Nombre: " << actual->nombre << ", Prioridad: " << actual->prioridad << "\n";
        actual = actual->siguiente; // Avanza al siguiente nodo
    }
}
```

```cpp
//---------------- PLANIFICADOR DE CPU ----------------//

// Inserta un proceso en la cola ordenado por prioridad (menor número = mayor prioridad)
void encolarProcesoEnPlanificador() {
    cout << "--- Encolar proceso en planificador ---\n";

    int pid;
    // Solicita el PID del proceso a encolar, validando que sea positivo
    do {
        cout << "Ingrese PID del proceso a encolar: ";
        cin >> pid;
        if (pid <= 0) {
            cout << "Error: Debe ingresar un número entero positivo.\n";
        }
        cin.clear(); // Limpia posibles errores de entrada
    } while (pid <= 0);

    Proceso* p = buscarProcesoPorPID(pid);      // Busca el proceso en la lista general

    // Si no se encuentra, muestra error
    if (p == NULL) {
        cout << "Error: No existe un proceso con ese PID en la lista general.\n";
        return;
    }

    // Verifica si ya está en la cola
    if (estaEnCola(pid)) {
        cout << "Error: El proceso ya está en la cola de planificación.\n";
        return;
    }

    // Crea un nuevo nodo de la cola
    NodoCola* nuevo = new NodoCola;      // Crea un nuevo nodo
    nuevo->proceso = p;                  // Apunta al proceso encontrado
    nuevo->siguiente = NULL;

    // Inserta el nodo ordenado por prioridad (menor valor = más prioritario)
    if (cabezaCola == NULL || cabezaCola->proceso->prioridad > p->prioridad) {
        // Si la cola está vacía o el nuevo proceso tiene mayor prioridad
        nuevo->siguiente = cabezaCola;
        cabezaCola = nuevo;      // Se convierte en el nuevo primero
    } else {
        NodoCola* actual = cabezaCola;
        while (actual->siguiente != NULL && actual->siguiente->proceso->prioridad <= p->prioridad) {
            actual = actual->siguiente;      // Avanza mientras encuentre igual o menor prioridad
        }
        // Inserta el nuevo nodo en la posición adecuada
        nuevo->siguiente = actual->siguiente;
        actual->siguiente = nuevo;
    }

    cout << "Proceso encolado en planificador correctamente.\n";
}
```

```cpp
// Desencola y "ejecuta" el primer proceso (mayor prioridad)
void desencolarYEjecutarProceso() {

    cout << "--- Desencolar y ejecutar proceso ---\n";

    // Verifica si la cola está vacía
    if (cabezaCola == NULL) {
        cout << "La cola de planificación está vacía.\n";
        return;
    }

    NodoCola* temp = cabezaCola;              // Toma el primer nodo (mayor prioridad)
    cabezaCola = cabezaCola->siguiente;       // Avanza el inicio de la cola

    // Muestra información del proceso a ejecutar
    cout << "Ejecutando proceso:\n";
    cout << "PID: " << temp->proceso->pid << ", Nombre: " << temp->proceso->nombre << ", Prioridad: " << temp->proceso->prioridad << "\n";

    delete temp;    // Libera el nodo ejecutado
}
```

```cpp
// Muestra la cola de planificación
void mostrarColaPlanificador() {

    cout << "--- Cola de planificación (ordenada por prioridad) ---\n";

    // Si está vacía, lo indica
    if (cabezaCola == NULL) {
        cout << "La cola de planificación está vacía.\n";
        return;
    }

    // Recorre la cola e imprime cada proceso
    NodoCola* actual = cabezaCola;
    while (actual != NULL) {
        cout << "PID: " << actual->proceso->pid << ", Nombre: " << actual->proceso->nombre << ", Prioridad: " << actual->proceso->prioridad << "\n";
        actual = actual->siguiente;
    }
}
```

```cpp
//---------------- GESTOR DE MEMORIA ----------------//

// Asigna un bloque de memoria a un proceso (pila push)
void asignarMemoria() {
    cout << "--- Asignar memoria a proceso ---\n";

    int pid;
    // Solicita PID válido
    do {
        cout << "Ingrese PID del proceso: ";
        cin >> pid;
        if (pid <= 0) {
            cout << "Error: Debe ingresar un número entero positivo.\n";
        }
        cin.clear();
    } while (pid <= 0);

    int tamanio;
    // Solicita tamaño de memoria válido
    do {
        cout << "Ingrese tamaño de memoria a asignar: ";
        cin >> tamanio;
        if (tamanio <= 0) {
            cout << "Error: Debe ingresar un número entero positivo.\n";
        }
        cin.clear();
    } while (tamanio <= 0);

    // Crea un nuevo bloque de memoria (nodo de pila)
    BloqueMemoria* nuevo = new BloqueMemoria;
    nuevo->pid = pid;
    nuevo->tamanio = tamanio;
    nuevo->siguiente = topeMemoria;         // Lo enlaza con el anterior tope
    topeMemoria = nuevo;                    // Se convierte en el nuevo tope

    cout << "Memoria asignada al proceso " << pid << " (tamaño: " << tamanio << ")\n";
}
// Libera el último bloque de memoria (pila pop)
void liberarMemoria() {
    cout << "--- Liberar memoria ---\n";

    // Si no hay bloques, lo indica
    if (topeMemoria == NULL) {
        cout << "No hay memoria para liberar.\n";
        return;
    }

    // Elimina el bloque del tope de la pila
    BloqueMemoria* temp = topeMemoria;
    topeMemoria = topeMemoria->siguiente;       // Avanza el tope
    cout << "Memoria liberada del proceso " << temp->pid << " (tamaño: " << temp->tamanio << ")\n";
    delete temp;                                // Libera el bloque de memoria
}

// Muestra el estado actual de la pila de memoria
void estadoMemoria() {
    cout << "--- Estado actual de la memoria ---\n";

    // Si la pila está vacía
    if (topeMemoria == NULL) {
        cout << "La memoria está completamente libre.\n";
        return;
    }

    // Recorre la pila mostrando los bloques
    BloqueMemoria* actual = topeMemoria;
    while (actual != NULL) {
        cout << "Proceso " << actual->pid << " - Tamaño: " << actual->tamanio << "\n";
        actual = actual->siguiente;
    }
}
```

```
342  //---------------- MENÚ PRINCIPAL ----------------//
343
344  void menu() {
345      int opcionPrincipal;
346
347      do {
348          do {
349              cout << "\n=== Menu Principal ===\n";
350              cout << "1. Gestor de Procesos\n";
351              cout << "2. Planificador de CPU (Cola de Prioridad)\n";
352              cout << "3. Gestor de Memoria\n";
353              cout << "4. Salir\n";
354              cout << "Seleccione una opcion (1-4): ";
355              cin >> opcionPrincipal;
356
357              if (opcionPrincipal < 1 || opcionPrincipal > 4) {
358                  cout << "Opcion invalida. Por favor ingrese un número entre 1 y 4.\n";
359              }
360              cin.clear();
361          } while (opcionPrincipal < 1 || opcionPrincipal > 4);
362
363          switch (opcionPrincipal) {
364              case 1: {
365                  int opcionProceso;
366                  do {
367                      cout << "\n--- Gestor de Procesos ---\n";
368                      cout << "1. Insertar nuevo proceso\n";
369                      cout << "2. Eliminar proceso\n";
370                      cout << "3. Mostrar todos los procesos\n";
371                      cout << "4. Volver al menu principal\n";
372                      cout << "Seleccione una opcion (1-4): ";
373                      cin >> opcionProceso;
374
375                      switch (opcionProceso) {
376                          case 1: insertarProceso(); break;
377                          case 2: eliminarProceso(); break;
378                          case 3: mostrarProcesos(); break;
379                          case 4: cout << "Volviendo al menu principal...\n"; break;
380                          default: cout << "Opcion invalida.\n"; break;
381                      }
384                      system("cls");   // Limpia pantalla
385                  } while (opcionProceso != 4);
386                  break;
387              }
388              case 2: {
389                  int opcionPlanificador;
390                  do {
391                      cout << "\n=== Planificador de CPU (Cola de Prioridad) ===\n";
392                      cout << "1. Encolar proceso por PID\n";
393                      cout << "2. Desencolar y ejecutar proceso\n";
394                      cout << "3. Mostrar cola actual\n";
395                      cout << "4. Volver al menu principal\n";
396                      cout << "Seleccione una opcion (1-4): ";
397                      cin >> opcionPlanificador;
398
399                      switch (opcionPlanificador) {
400                          case 1: encolarProcesoEnPlanificador(); break;
401                          case 2: desencolarYEjecutarProceso(); break;
402                          case 3: mostrarColaPlanificador(); break;
403                          case 4: cout << "Volviendo al menu principal...\n"; break;
404                          default: cout << "Opcion invalida.\n"; break;
405                      }
406
407                      system("pause");
408                      system("cls");
409                  } while (opcionPlanificador != 4);
410                  break;
411              }
412              case 3: {
413                  int opcionMemoria;
414                  do {
415                      cout << "\n--- Gestor de Memoria ---\n";
416                      cout << "1. Asignar memoria a proceso (push)\n";
417                      cout << "2. Liberar memoria (pop)\n";
418                      cout << "3. Ver estado actual de la memoria\n";
419                      cout << "4. Volver al menu principal\n";
420                      cout << "Seleccione una opcion (1-4): ";
421                      cin >> opcionMemoria;
```

```
422
423                     switch (opcionMemoria) {
424                         case 1: asignarMemoria(); break;
425                         case 2: liberarMemoria(); break;
426                         case 3: estadoMemoria(); break;
427                         case 4: cout << "Volviendo al menu principal...\n"; break;
428                         default: cout << "Opcion invalida.\n"; break;
429                     }
430
431                     system("pause");
432                     system("cls");
433                 } while (opcionMemoria != 4);
434                 break;
435             }
436             case 4:
437                 cout << "Saliendo del programa...\n";
438                 break;
439         }
440
441     } while (opcionPrincipal != 4);
442 }
443
444 // Función principal del programa
445 int main() {
446     menu(); // Llama al menú principal
447     return 0;
448 }
```

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

Validación al seleccionar opciones

```
=== Menu Principal ===
1. Gestor de Procesos
2. Planificador de CPU (Cola de Prioridad)
3. Gestor de Memoria
4. Salir
Seleccione una opcion (1-4): 5
Opcion invalida. Por favor ingrese un n·mero entre 1 y 4.
```

Validación en todos los ingresos de PID

```
--- Gestor de Procesos ---
1. Insertar nuevo proceso
2. Eliminar proceso
3. Mostrar todos los procesos
4. Volver al menu principal
Seleccione una opcion (1-4): 1
--- Insertar nuevo proceso ---
Ingrese PID (entero positivo): -1
Error: El PID debe ser un n·mero entero positivo.
Ingrese PID (entero positivo):
```

Verificación en todas las partes donde se muestran los procesos

```
=== Planificador de CPU (Cola de Prioridad) ===
1. Encolar proceso por PID
2. Desencolar y ejecutar proceso
3. Mostrar cola actual
4. Volver al menu principal
Seleccione una opcion (1-4): 3
--- Cola de planificaci¾n (ordenada por prioridad) ---
La cola de planificaci¾n estß vacÝa.
Presione una tecla para continuar . . .
```

3. Manual de usuario

3.1. Descripción general

Este programa simula la gestión de procesos en un sistema operativo simple, clasificándolos según su prioridad: alta, media y baja. Utiliza listas enlazadas en C++ para almacenar y manejar los procesos. El sistema permite al usuario agregar nuevos procesos, visualizar la lista de procesos activos y atenderlos en orden de prioridad, todo mediante una interfaz de consola intuitiva.

3.2. Requisitos del sistema

- Sistema operativo: Windows, Linux o macOS
- Compilador de C++ (como g++ o Dev-C++)
- Terminal o consola de comandos

3.3. Compilación del Programa

Pasos a seguir:

Paso 1: Abre la terminal o línea de comandos.

Paso 2: Navega hasta la carpeta donde esté tu archivo, por ejemplo simulador.cpp.

Paso 3: Presiona la siguiente tecla para ejecutar (F10)

Paso 4: El programa mostrará un menú principal desde donde podrás acceder a sus funcionalidades.

3.4. Menú del programa

1. Gestor de procesos
2. Planificador de CPU (Cola de prioridad)
3. Gestor de memoria
4. Salir

Seleccione una opción (1-4):

3.5. Explicación de cada parte del menú

- **Gestor de procesos:** Aquí puedes crear nuevos procesos asignándoles un nombre y prioridad (alta, media, baja). Los procesos se almacenan en una lista enlazada según su prioridad.

Opciones disponibles:
- Insertar nuevo proceso
- Eliminar proceso
- Mostrar todos los procesos registrados

- **Planificador de CPU (Cola de prioridad):** Esta opción permite atender los procesos en orden de prioridad. Primero se atienden los procesos de prioridad alta, luego media, y después baja. Finalmente, se muestra qué proceso está siendo atendido y se elimina de la lista.
  Opciones disponibles:
  - Encolar un proceso
  - Desencolar y ejecutar un proceso
  - Mostrar la cola de planificación
- **Gestor de memoria:** Si está implementado, este módulo permite simular la asignación y liberación de memoria para los procesos. Además, consulta el menú interno para usar las funciones relacionadas a la memoria.
  Opciones disponibles:
  - Asignar memoria a proceso
  - Liberar memoria
  - Ver estado actual de la memoria
- **Salir:** Finaliza el programa.

3.5. Consejos de uso

- Antes de planificar procesos, asegúrate de haber creado algunos desde el Gestor de Procesos.
- En el planificador, puedes priorizar tareas críticas en la cola de ejecución
- El gestor de memoria es útil para simular operaciones de push y pop en la administración de bloques dinámicos.
- Usa correctamente los menús y evita ingresar valores no válidos (el sistema está preparado para detectar errores comunes).
- El programa es ideal para estudiantes o usuarios que estén aprendiendo sobre estructuras de datos y funcionamiento interno de un sistema operativo.

3.5. Problemas comunes

Problema	Causa Posible	Solución
No se muestran procesos	Aun no has creado procesos	Ve al gestor de proceso y crea uno
No se puedes ejecutar ningún proceso	No has colocado ningún proceso	Usa el Planificador de CPU para encolar primero
Memoria no se asigna	No has ingresado un PID valido	Asegúrate de ingresar un numero entero positivo

Error en selección menú	Entrada valida/letras símbolos, números fuera del rango permitido	Ingresar solamente opciones del 1 al 4

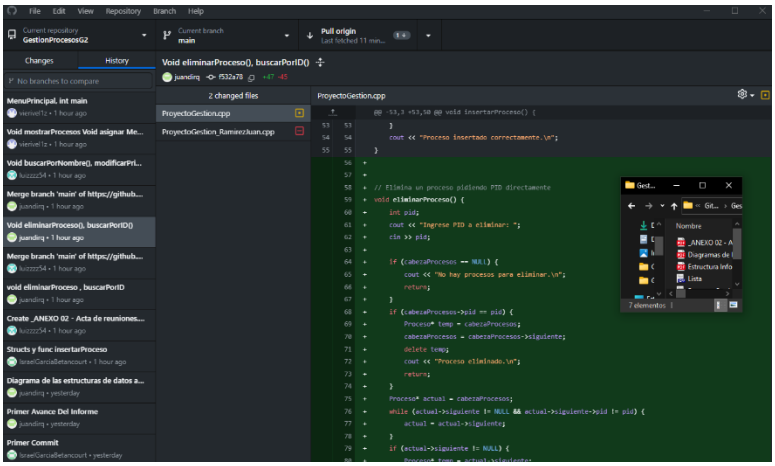
3.5. Recomendaciones finales

Este programa es útil para estudiantes que desean reforzar conocimientos sobre estructuras como listas, colas y pilas, así como conceptos básicos de planificación de procesos y gestión de memoria. Se recomienda usarlo en sesiones prácticas acompañadas de una guía docente o durante tareas de laboratorio. Puede adaptarse o ampliarse para incluir otras funcionalidades como multitarea, simulación de interrupciones o gestión de recursos del sistema.

CAPÍTULO 4: Evidencias de trabajo en equipo

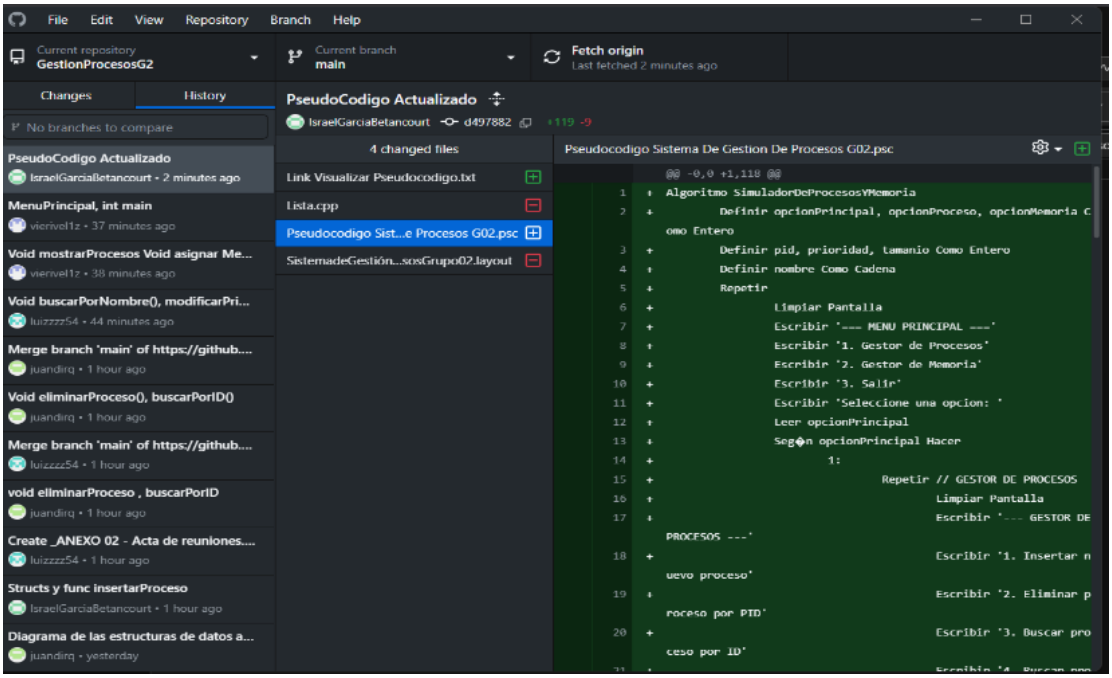
1. Repositorio con Control de Versiones (Capturas de Pantalla)

Repositorio 1 – Captura de participación en el GitHub



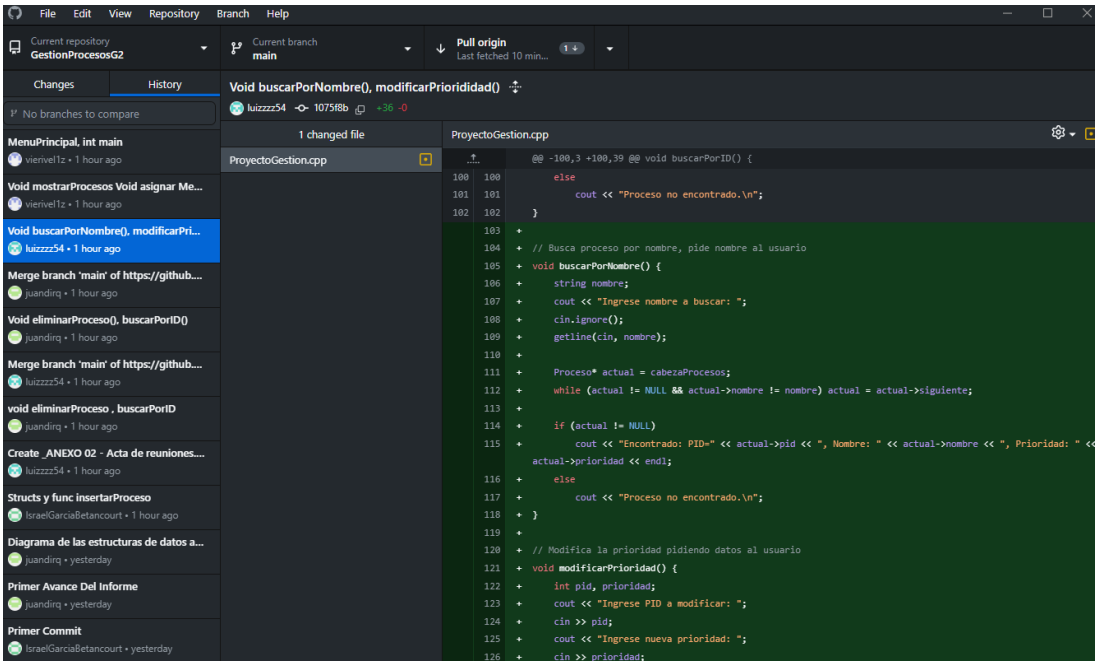
(Ramírez Quillatupa Juan Diego)

Repositorio 2 – Captura de participación en el GitHub



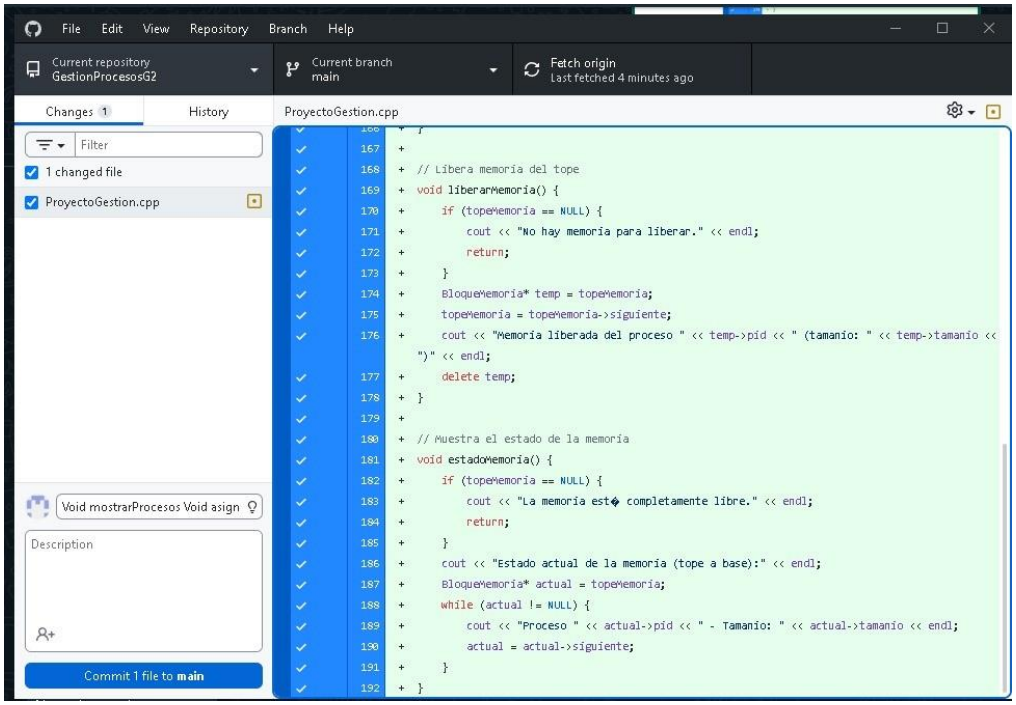
(García Betancourt Israel Jedidias)

Repositorio 3 – Captura de participación en el GitHub



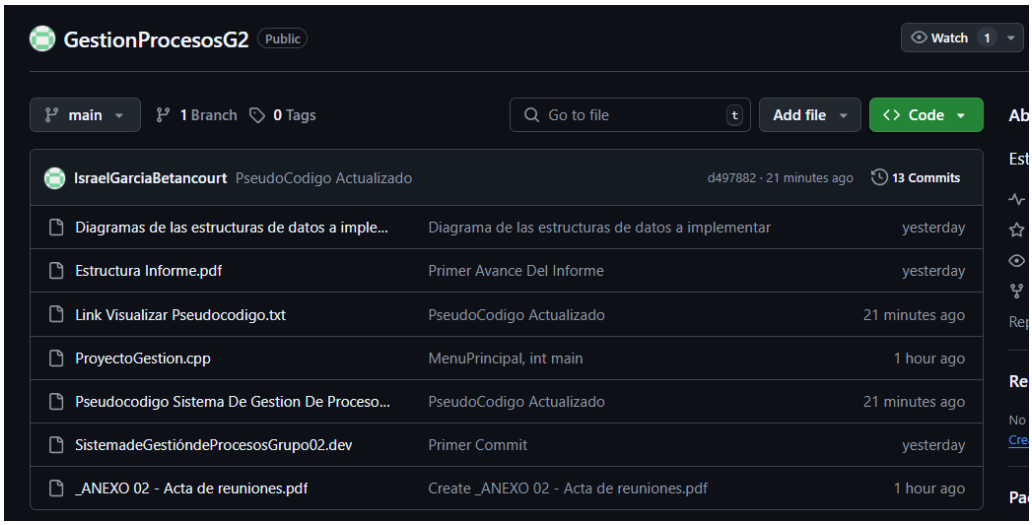
(Leon Armas Luis Aram)

Repositorio 4 – Captura de participación en el GitHub



(Veliz Durand Vieri Del Piero)

Repositorio 5 – Enlace al GitHub



https://github.com/IsraelGarciaBetancourt/GestionProcesosG2

Repositorio 6 – Canva para la exposición

https://www.canva.com/design/DAGos7CEUA0/SLzwxElwK6lQjbI2S3WHAQ/edit?utm_content=DAGos7CEUA0&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton

2. Plan de Trabajo y Roles Asignados

2.1. Plan de Trabajo y roles asignados

Integrante	Semana 1: Planificación y diseño	Semana 2: Implementación y desarrollo	Semana 3: Finalización y pruebas
Israel	• Participar en el análisis del problema. • Diseñar el Gestor de Procesos (lista enlazada). • Elaborar pseudocódigo de las operaciones: inserción eliminación, búsqueda, modificación de prioridad.	• Implementar el módulo del Gestor de Procesos. • Probar operaciones de lista enlazada.	• Finalizar detalles del módulo. • Realizar pruebas de funcionamiento. • Apoyar en la documentación y presentación.
Juan	• Diseñar el Planificador de CPU (cola de prioridad). • Redactar pseudocódigo de encolamiento, desencolamiento y visualización.	• Desarrollar e implementar la cola de prioridad. • Simular ejecución de procesos.	• Completar y probar el módulo. • Apoyar en integración y documentación.
Luis	• Diseñar el Gestor de Memoria (pila). • Desarrollar pseudocódigo para push, pop y ver estado de memoria.	• Programar el gestor de memoria con pila. • Verificar asignación y liberación correcta.	• Realizar pruebas de memoria. • Ajustes finales y apoyo en presentación.
Vieri	• Coordinar la elaboración del documento de diseño. • Diseñar diagramas generales del sistema. • Apoyar a los demás integrantes en la redacción del plan.	• Integrar los módulos en un solo programa. • Realizar pruebas iniciales de integración. • Supervisar calidad del código.	• Dirigir pruebas generales del sistema. • Elaborar la documentación final. • Coordinar el ensayo de la presentación.

2.2. Fecha de límite de la entrega de trabajo

ISRAEL

Semana	Fechas	Actividades	Fecha límite
Semana 1	15 – 21 de mayo	- Análisis del problema - Diseño del Gestor de Procesos (lista enlazada) - Pseudocódigo de inserción, eliminación, búsqueda, etc.	21 mayo
Semana 2	22 – 28 de mayo	- Implementación del módulo de lista enlazada - Pruebas de inserción y eliminación	28 mayo
Semana 3	29 mayo – 04 junio	- Ajustes finales del módulo - Pruebas generales y apoyo en documentación	04 junio

JUAN

Semana	Fechas	Actividades	Fecha límite
Semana 1	15 – 21 de mayo	- Diseño del Planificador de CPU (cola de prioridad) - Pseudocódigo de encolamiento, desencolamiento, etc.	21 mayo
Semana 2	22 – 28 de mayo	- Implementación del módulo de cola de prioridad - Simulación de ejecución de procesos	28 mayo
Semana 3	29 mayo – 04 junio	- Finalización del módulo - Pruebas y apoyo en documentación	04 junio

LUIS

Semana	Fechas	Actividades	Fecha límite
Semana 1	15 – 21 de mayo	- Diseño del Gestor de Memoria (pila) - Pseudocódigo de push, pop y estado de memoria	21 mayo
Semana 2	22 – 28 de mayo	- Programación del gestor de memoria - Verificación de asignación y liberación de memoria	28 mayo
Semana 3	29 mayo – 04 junio	- Pruebas del módulo - Ajustes finales y apoyo en presentación	04 junio

VIERI

Semana	Fechas	Actividades	Fecha límite
Semana 1	15 – 21 de mayo	- Coordinación del documento de diseño - Diseño de diagramas generales del sistema	21 mayo
Semana 2	22 – 28 de mayo	- Apoyo en redacción del plan - Integración inicial de módulos y pruebas	28 mayo
Semana 3	29 mayo – 04 junio	- Supervisión del código y pruebas generales - Documentación final y ensayo de presentación	04 junio

2.3. Acta de reuniones

Link de todas las actas de reuniones:

https://drive.google.com/drive/folders/132PNAIL82iWC6nggJknhpSNzJP8LsdGV?usp=sharing