



Estructura de Datos

NRC: **35458**

Título: Momento o torque y fuerzas paralelas

Docente:

Integrantes:

- ❖ Cutipa Jara Juan Alex
- ❖ Espinoza Mora Johanna Alexandra
- ❖ Guzman Condori Flavio Cesar Guzman
- ❖ Pérez Olivera Jaasiel Joana
- ❖ Pontecil Alvarez Bruce Anderson

Carrera profesional:

- ❖ Ingeniería de Sistemas e Informática
- ❖ Ingeniería de Sistemas e Informática
- ❖ Ingeniería de Sistemas e Informática
- ❖ Ingeniería de Sistemas e Informática
- ❖ Ingeniería de Sistemas e Informática

Número de contacto:

- ❖ 961463746
- ❖ 942402711
- ❖ 973134173
- ❖ 993656458
- ❖ 977612797

Descripción del problema:

Como parte del reto de desarrollar un gestor de procesos, aplicaremos los conocimientos adquiridos durante este periodo para construir un programa que permita administrar procesos de manera eficiente.

Para lograrlo, es fundamental elegir y utilizar correctamente las estructuras de datos adecuadas, como:

Listas enlazadas, para almacenar y recorrer todos los procesos registrados.

Colas, para organizar los procesos que serán ejecutados por la CPU según su prioridad.

Pilas, para simular la asignación y liberación de memoria, siguiendo un orden de último en entrar, primero en salir.

El objetivo es que el sistema funcione de forma ordenada, flexible y rápida, permitiendo al usuario agregar, modificar, buscar y eliminar procesos, además de simular cómo se comportan estos dentro de la CPU y la memoria.

Requerimientos del sistema:

- Funcionales:

1. RF01: El sistema debe permitir crear nuevos procesos con atributos configurables.
2. RF02: El sistema debe asignar automáticamente un ID único a cada proceso.
3. RF03: El sistema debe permitir seleccionar el algoritmo de planificación de procesos (ej. FCFS, Round Robin, Prioridad).
- RF04: El sistema debe simular el cambio de estados de los procesos (nuevo, listo, ejecutando, bloqueado, finalizado).
4. RF05: El sistema debe permitir pausar, reanudar y terminar procesos manualmente.
5. RF06: El sistema debe mostrar en tiempo real una tabla o lista con el estado actual de todos los procesos.

6. RF07: El sistema debe simular el paso del tiempo para representar la ejecución de procesos.

- **No funcionales:**

1. RNF01: El sistema debe estar desarrollado completamente en lenguaje C++.
2. RNF02: El sistema debe contar con una interfaz de usuario en consola clara e intuitiva.
3. RNF03: El sistema debe poder ejecutarse en sistemas operativos Windows, Linux o MacOS sin necesidad de librerías externas.
4. RNF04: El código debe estar modularizado y documentado adecuadamente para facilitar su comprensión y mantenimiento.
5. RNF05: El sistema debe ser capaz de gestionar al menos 20 procesos simultáneamente sin errores ni pérdidas de información.

Estructuras de datos propuestas:

Listas enlazadas, para almacenar y recorrer todos los procesos registrados.

Colas, para organizar los procesos que serán ejecutados por la CPU según su prioridad.

Pilas, para simular la asignación y liberación de memoria, siguiendo un orden de último en entrar, primero en salir.

Justificación de la elección

- **Listas enlazadas**

Las listas enlazadas se usan para los procesos principalmente porque permiten una gestión dinámica y flexible de la memoria, así como una gestión eficiente de la lista de procesos en ejecución.

Beneficios clave:

Gestión dinámica de la memoria:

Las listas enlazadas pueden crecer o reducir su tamaño durante la ejecución del programa, lo que es ideal cuando no se conoce de antemano el número de procesos que se ejecutarán.

Inserción y eliminación eficientes:

Las listas enlazadas permiten añadir o quitar procesos de la lista sin necesidad de mover grandes cantidades de datos en la memoria, lo que es muy útil en sistemas operativos donde los procesos cambian constantemente de estado.

Lista de procesos en ejecución:

En sistemas operativos, los programadores de procesos utilizan listas enlazadas para mantener la lista de todos los procesos que están en ejecución, permitiendo una gestión eficiente de la planificación y la ejecución de los procesos.

Gestión de la memoria en sistemas operativos:

Las listas enlazadas ayudan a gestionar la asignación de memoria a los procesos, lo que es crucial para la eficiencia del sistema operativo y la prevención de problemas como fragmentación de memoria.

- **Colas**

Las colas se usan para la gestión de procesos en la CPU en sistemas operativos y para la ejecución de tareas en orden en diversas aplicaciones. En sistemas operativos, las colas, como la cola de listos, organizan y priorizan los procesos que deben ejecutarse. En aplicaciones como chatbots o servidores de mensajería, las colas permiten manejar mensajes entrantes de manera ordenada.

Gestión de procesos en la CPU:

En un sistema operativo, los procesos no se ejecutan de inmediato cuando están listos. Se colocan en una cola (cola de listos) para que la CPU pueda elegir qué proceso ejecutar a continuación.

Orden de ejecución:

Las colas siguen el principio FIFO (First In, First Out), es decir, el primer elemento en entrar es el primero en salir. Esto garantiza que los procesos se ejecuten en el orden en que fueron colocados en la cola.

Priorización:

La cola de listas puede utilizar algoritmos de programación (como el programador de CPU) para priorizar ciertos procesos, dando más tiempo de CPU a aquellos que lo necesitan más o que están más cerca de terminar.

- **Pilas**

Las pilas en la memoria, también conocidas como estructuras de datos de tipo "último en entrar, primero en salir" (LIFO), se utilizan principalmente para gestionar la memoria durante la ejecución de programas, especialmente al realizar llamadas a funciones y al manejar la recursividad. Permiten almacenar información temporalmente, como variables locales, parámetros de funciones y direcciones de retorno, para luego restaurar el estado del programa cuando la función termina.

Descripción de estructuras de datos y operaciones:

Listas enlazadas

Una lista enlazada es una estructura de datos dinámica compuesta por nodos, donde cada nodo contiene un dato (en este caso, un proceso) y un puntero o enlace al siguiente nodo en la secuencia. Esto permite almacenar procesos de forma ordenada sin necesidad de que estén contiguos en memoria.

Operaciones básicas:

- Creación e inicialización: Se crea una lista vacía con un puntero inicial que apunta a null.
- Inserción: Se puede insertar un nuevo proceso en cualquier posición de la lista (inicio, medio o final) modificando los punteros, lo que se realiza en tiempo constante si se tiene el nodo previo localizado.
- Búsqueda: Recorrer la lista secuencialmente para encontrar un proceso por su ID u otro atributo.
- Eliminación: Se elimina un nodo ajustando los punteros del nodo anterior para saltar el nodo eliminado.
- Recorrido: Visitar cada nodo para mostrar o modificar información.

Justificación:

Las listas enlazadas permiten gestionar dinámicamente la memoria y la cantidad de procesos sin necesidad de definir un tamaño fijo, facilitando la inserción y eliminación eficiente de procesos que cambian de estado o prioridad. Esto es fundamental para un gestor de procesos donde la cantidad y estado de procesos varía constantemente.

Colas

Una cola es una estructura de datos lineal que sigue el principio FIFO (First In, First Out), donde los elementos se insertan en un extremo (final) y se extraen del otro (frente).

Operaciones básicas:

- Encolar (enqueue): Agregar un proceso al final de la cola.
- Desencolar (dequeue): Extraer el proceso que está al frente, es decir, el que lleva más tiempo esperando.
- Consulta del frente: Ver cuál es el proceso que será ejecutado próximamente.

Justificación:

Las colas son ideales para organizar los procesos que serán ejecutados por la CPU según su orden de llegada o prioridad. En la planificación de procesos, la cola de listos mantiene el orden de ejecución y permite implementar algoritmos como FCFS o Round Robin. Además, las colas pueden adaptarse para soportar prioridades, garantizando que los procesos más importantes se atiendan primero.

Pilas

Una pila es una estructura de datos que sigue el principio LIFO (Last In, First Out), donde el último elemento agregado es el primero en ser removido.

Operaciones básicas:

- Apilar (push): Insertar un nuevo elemento en la cima de la pila.
- Desapilar (pop): Remover el elemento que está en la cima.
- Consulta de cima: Obtener el elemento superior sin removerlo.

Justificación:

Las pilas se utilizan para simular la asignación y liberación de memoria en la ejecución de procesos, especialmente para manejar llamadas a funciones, variables locales y recursividad. La memoria se asigna y libera siguiendo un orden LIFO, lo que facilita la gestión eficiente y ordenada del espacio de memoria para cada proceso. Además, las pilas permiten simular el comportamiento real de la

memoria en sistemas operativos, donde la última información almacenada es la primera en ser liberada.

En conjunto, estas estructuras permiten al sistema cumplir con los requerimientos funcionales y no funcionales, proporcionando una gestión eficiente, flexible y ordenada de los procesos, su planificación y la simulación del uso de memoria.

Algoritmos principales:

- ***Pseudocódigo para agregar proceso.***

INICIO

Crear función insertarProceso con parámetros:

int identificador

string nombre

int prioridad

Crear un nuevo objeto nuevoProceso de tipo Proceso (puntero).

Asignar atributos a nuevoProceso:

nuevoProceso->identificador = identificador

nuevoProceso->nombre = nombre

nuevoProceso->prioridad = prioridad

nuevoProceso->siguiente = NULL // indica que será el último nodo

Si cabeza es NULL (lista vacía) ENTONCES:

cabeza = nuevoProceso // nuevoProceso es el primer nodo

SINO:

Crear puntero temporal y asignarlo a cabeza

MIENTRAS temporal->siguiente sea distinto de NULL HACER:

temporal = temporal->siguiente // avanzar al siguiente nodo

FIN MIENTRAS

Asignar temporal->siguiente = nuevoProceso // agregar al final de la lista

Imprimir: "Su proceso fue añadido correctamente (^)"

FIN

- ***Pseudocódigo para cambiar el estado del proceso.***

INICIO

Función modificarPrioridad(identificador, nuevaPrioridad)

temporal ← cabeza //temporal apunta al nodo cabeza

MIENTRAS temporal ≠ NULO

SI temporal.identificador = identificador ENTONCES

temporal.prioridad ← nuevaPrioridad // nuevaPrioridad reemplaza al atributo
prioridad

IMPRIMIR "Su prioridad fue modificada correctamente (^)"

RETORNAR

FIN SI

temporal ← temporal.siguiente //temporal avanza al siguiente nodo de la lista

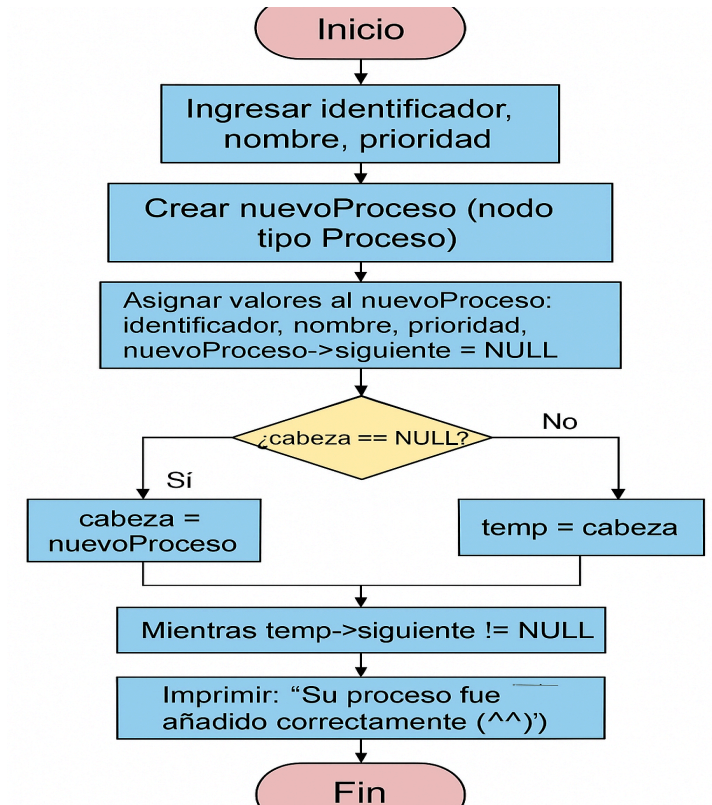
FIN MIENTRAS

IMPRIMIR "Su proceso no fue encontrado (T-T)"

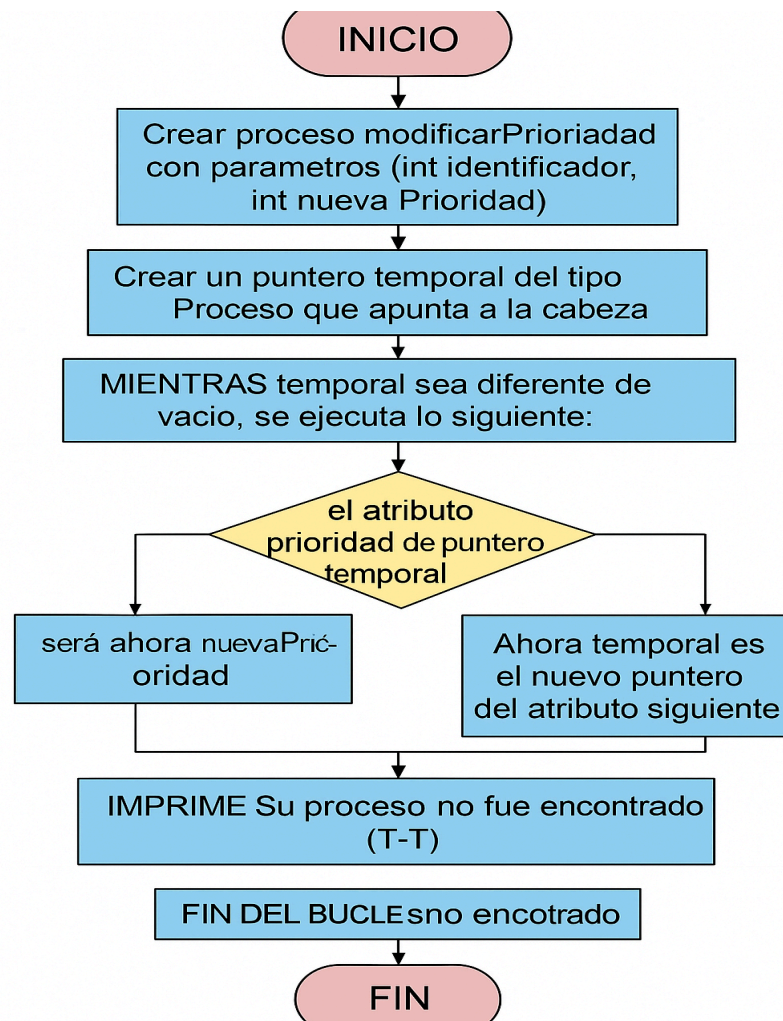
FIN

Diagramas de Flujo

- Primer pseudocódigo



- Segundo pseudocódigo



Justificación del diseño:

(Ventajas, eficiencia, etc.)

Ventajas del diseño

- Inserciones y eliminaciones eficientes:

Las listas enlazadas permiten insertar y eliminar procesos (nodos) de manera eficiente, especialmente cuando la posición es conocida. Modificar un puntero es mucho más rápido que reestructurar toda una estructura de datos como ocurre en los arrays. Esto es especialmente útil en sistemas donde los procesos pueden entrar y salir dinámicamente.

- Tamaño dinámico:

No es necesario definir el tamaño máximo de la lista de procesos al inicio. La lista puede crecer o decrecer según sea necesario, adaptándose al número

real de procesos activos en cada momento. Esto optimiza el uso de memoria y evita la sobreasignación o la necesidad de redimensionar estructuras estáticas.

- Flexibilidad en la gestión:

Puedes añadir procesos en cualquier posición de la lista (por ejemplo, según prioridad), y modificar atributos como la prioridad de manera sencilla recorriendo la lista hasta encontrar el proceso deseado.

Eficiencia en operaciones específicas:

- Inserción: Si el punto de inserción es conocido, se realiza en tiempo constante.
- Eliminación o modificación: Recorrer la lista para encontrar el nodo deseado, pero una vez localizado, la operación es inmediata.
- Bajo acoplamiento de memoria: Los nodos no necesitan estar almacenados de forma contigua, lo que reduce problemas de fragmentación de memoria y permite un uso más eficiente de la misma.

Eficiencia en el contexto de planificación de procesos

- Gestión de prioridades:

El uso de listas enlazadas permite implementar fácilmente algoritmos de planificación por prioridad, donde cada proceso tiene un atributo de prioridad y la lista puede recorrer para encontrar o modificar procesos según este criterio.

- Facilidad de actualización:

El segundo diagrama muestra cómo modificar la prioridad de un proceso recorriendo la lista hasta encontrar el identificador correspondiente. Esta

operación es directa y no requiere reestructuración masiva de la estructura de datos, solo cambiar el valor de un campo.

Consideraciones adicionales

- Consumo de memoria:

Cada nodo requiere espacio adicional para almacenar el puntero al siguiente nodo, lo que puede aumentar el uso de memoria en comparación con arrays, pero este costo es compensado por la flexibilidad y eficiencia en inserciones y eliminaciones.

- Acceso secuencial:

El acceso a elementos específicos es secuencial, lo que puede ser menos eficiente que el acceso aleatorio de un array. Sin embargo, en sistemas donde las operaciones más frecuentes son inserciones, eliminaciones y modificaciones, esta desventaja es mínima frente a las ventajas obtenidas.

Código limpio, bien comentado y estructurado.

```
1  #include <iostream>
2  #include <cstring>
3
4  using namespace std;
5
6  // ESTRUCTURA DEL PROCESO: representa un nodo en una lista enlazada
7  struct Proceso {
8      int identificador;           // ID del proceso
9      string nombre;              // Nombre del proceso
10     int prioridad;               // Prioridad del proceso
11     Proceso* siguiente;          // Puntero al siguiente nodo
12 };
13
14 // LISTA ENLAZADA PARA GESTIONAR PROCESOS
15 class GestorDeProcesos {
16 private:
17     Proceso* cabeza;             // Puntero al primer proceso en la lista
18
19 public:
20     GestorDeProcesos() {
21         cabeza = NULL;           // Inicialmente, la lista está vacía
22     }
23
24     // Inserta un nuevo proceso al final de la lista
25     void insertarProceso(int identificador, string nombre, int prioridad) {
26         Proceso* nuevoProceso = new Proceso;
27         nuevoProceso->identificador = identificador;
28         nuevoProceso->nombre = nombre;
29         nuevoProceso->prioridad = prioridad;
30         nuevoProceso->siguiente = NULL;
31
32         if (cabeza == NULL) {
33             cabeza = nuevoProceso;
34         } else {
35             Proceso* temporal = cabeza;
36             while (temporal->siguiente != NULL)
37                 temporal = temporal->siguiente;
38             temporal->siguiente = nuevoProceso;
39         }
40         cout << "Su proceso fue añadido correctamente (^^\n";
41     }
42
43     // Elimina un proceso por su ID
44     void eliminarProceso(int identificador) {
45         Proceso* temporal = cabeza;
46         Proceso* anterior = NULL;
47         while (temporal != NULL && temporal->identificador != identificador) {
48             anterior = temporal;
49             temporal = temporal->siguiente;
50         }
51         if (temporal == NULL) {
52             cout << "Su proceso no fue encontrado (T-T)\n";
53             return;
54         }
55         if (anterior == NULL)
56             cabeza = temporal->siguiente;
57         else
58             anterior->siguiente = temporal->siguiente;
59         delete temporal;
60         cout << "Su proceso fue eliminado correctamente (^^\n";
```

```

61 }
62
63 // Busca un proceso por su ID y lo muestra
64 void buscarProceso(int identificador) {
65     Proceso* temporal = cabeza;
66     while (temporal != NULL) {
67         if (temporal->identificador == identificador) {
68             cout << "ID: " << temporal->identificador << " | Nombre: " << temporal->nombre << " | Prioridad: " << temporal->prioridad << " (0.0)\n";
69             return;
70         }
71         temporal = temporal->siguiente;
72     }
73     cout << "Su proceso no fue encontrado (T-T)\n";
74 }
75
76 // Modifica la prioridad de un proceso existente
77 void modificarPrioridad(int identificador, int nuevaPrioridad) {
78     Proceso* temporal = cabeza;
79     while (temporal != NULL) {
80         if (temporal->identificador == identificador) {
81             temporal->prioridad = nuevaPrioridad;
82             cout << "Sy prioridad fue modificada correctamente (^)\n";
83             return;
84         }
85         temporal = temporal->siguiente;
86     }
87     cout << "Su proceso no fue encontrado (T-T)\n";
88 }
89
90 // Muestra todos los procesos registrados
91 void mostrarProcesos() {
92     Proceso* temporal = cabeza;
93     if (temporal == NULL) {
94         cout << "No hay procesos registrados (T-T)\n";
95         return;
96     }
97     while (temporal != NULL) {
98         cout << "ID: " << temporal->identificador << ", Nombre: " << temporal->nombre << ", Prioridad: " << temporal->prioridad << " (0.0)\n";
99         temporal = temporal->siguiente;
100     }
101 }
102
103 // Devuelve el puntero al primer proceso de la lista
104 Proceso* obtenerCabeza() {
105     return cabeza;
106 }
107 };
108
109 // NODO PARA LA COLA DE PRIORIDAD
110 struct NodoCola {
111     Proceso* proceso;
112     NodoCola* siguiente;
113 };
114
115 // COLA DE PRIORIDAD PARA EL PLANIFICADOR DE CPU
116 class PlanificadorCPU {
117 private:
118     NodoCola* cabeza; // Puntero al primer nodo de la cola
119
120 public:
121     PlanificadorCPU() {
122         cabeza = NULL; // Inicialmente, la cola está vacía
123     }
124
125     // Encolar un proceso según su prioridad
126     void encolarProceso(Proceso* proceso) {
127         NodoCola* nuevoNodo = new NodoCola;
128         nuevoNodo->proceso = proceso;
129         nuevoNodo->siguiente = NULL;
130
131         if (cabeza == NULL || proceso->prioridad > cabeza->proceso->prioridad) {
132             nuevoNodo->siguiente = cabeza;
133             cabeza = nuevoNodo;
134         } else {
135             NodoCola* temporal = cabeza;
136             while (temporal->siguiente != NULL && proceso->prioridad <= temporal->siguiente->proceso->prioridad) {
137                 temporal = temporal->siguiente;
138             }
139             nuevoNodo->siguiente = temporal->siguiente;
140             temporal->siguiente = nuevoNodo;
141         }
142         cout << "Su proceso fue encolado correctamente (^)\n";
143     }
144
145     // Desencolar y 'ejecutar' un proceso
146     void desencolarProceso() {
147         if (cabeza == NULL) {
148             cout << "No hay procesos en la cola (T-T)\n";
149             return;
150         }
151         NodoCola* nodo = cabeza;
152         cabeza = cabeza->siguiente;
153         cout << "Ejecutando proceso: ID " << nodo->proceso->identificador << ", Nombre " << nodo->proceso->nombre << ", Prioridad " << nodo->proceso->prioridad << " (0.0)\n";
154         delete nodo;
155     }

```

```

157 // Mostrar la cola actual
158 void mostrarCola() {
159     NodoCola* temporal = cabeza;
160     if (temporal == NULL) {
161         cout << "La cola está vacía (T-T)\n";
162         return;
163     }
164     while (temporal != NULL) {
165         cout << "ID: " << temporal->proceso->identificador << ", Nombre: " << temporal->proceso->nombre << ", Prioridad: " << temporal->proceso->prioridad << " (0.0)\n";
166         temporal = temporal->siguiente;
167     }
168 }
169 };
170
171 // PILA PARA EL GESTOR DE MEMORIA
172 class GestorDeMemoria {
173 private:
174     Proceso* tope; // Puntero al tope de la pila
175 public:
176     GestorDeMemoria() {
177         tope = NULL; // Inicialmente, la pila está vacía
178     }
179
180 // Asignar memoria a un proceso (push)
181 void asignarMemoria(Proceso* proceso) {
182     proceso->siguiente = tope;
183     tope = proceso;
184     cout << "La memoria fue asignada correctamente (^)\n";
185 }
186
187 // Liberar memoria (pop)
188 void liberarMemoria() {
189     if (tope == NULL) {
190         cout << "No hay memoria asignada (T-T)\n";
191         return;
192     }
193     Proceso* temporal = tope;
194     tope = tope->siguiente;
195     cout << "La memoria fue liberada correctamente (^)\n";
196     delete temporal;
197 }
198
199 // Verificar el estado actual de la memoria
200 void verificarEstadoMemoria() {
201     Proceso* temporal = tope;
202     if (temporal == NULL) {
203         cout << "La memoria está vacía (T-T)\n";
204         return;
205     }
206     cout << "El estado actual de la memoria es: (0.0)\n";
207     while (temporal != NULL) {
208         cout << "ID: " << temporal->identificador << ", Nombre: " << temporal->nombre << ", Prioridad: " << temporal->prioridad << "\n";
209         temporal = temporal->siguiente;
210     }
211 }
212 }
213 };
214
215 // INTERFAZ DE USUARIO
216 void mostrarMenu() {
217     cout << "==== Sistema de gestion de procesos ==== \n";
218     cout << "1. Insertar proceso (^)\n";
219     cout << "2. Eliminar proceso (T-T)\n";
220     cout << "3. Buscar proceso (0.0)\n";
221     cout << "4. Modificar prioridad (^)\n";
222     cout << "5. Mostrar todos los procesos (0.0)\n";
223     cout << "6. Encolar proceso en CPU (^)\n";
224     cout << "7. Desencolar y ejecutar proceso (0.0)\n";
225     cout << "8. Mostrar cola de CPU (0.0)\n";
226     cout << "9. Asignar memoria (^)\n";
227     cout << "10. Liberar memoria (T-T)\n";
228     cout << "11. Verificar estado de memoria (0.0)\n";
229     cout << "0. Salir (T-T)\n";
230     cout << "==== \n";
231 }
232
233 int main() {
234     GestorDeProcesos gestorProcesos;
235     PlanificadorCPU planificadorCPU;
236     GestorDeMemoria gestorMemoria;
237
238     int opcion;
239     do {
240         mostrarMenu();
241         cout << "Ingrese una opción: ";
242         cin >> opcion;
243         cin.ignore(); // Limpiar el buffer de entrada
244
245         switch (opcion) {
246             case 1: {
247                 int id, prioridad;
248                 string nombre;
249                 cout << "Ingrese ID del proceso: ";
250                 cin >> id;
251                 cout << "Ingrese nombre del proceso: ";
252                 cin.ignore();
253                 getline(cin, nombre);
254                 cout << "Ingrese prioridad del proceso: ";
255                 cin >> prioridad;
256                 gestorProcesos.insertarProceso(id, nombre, prioridad);
257                 break;
258             }
259             case 2: {
260                 int id;
261                 cout << "Ingrese ID del proceso a eliminar: ";
262                 cin >> id;
263                 gestorProcesos.eliminarProceso(id);
264                 break;
265             }
266             case 3: {
267                 int id;
268                 cout << "Ingrese ID del proceso a buscar: ";
269                 cin >> id;
270                 gestorProcesos.buscarProceso(id);
271                 break;
272             }
273             case 4: {
274                 int id, nuevaPrioridad;
275                 cout << "Ingrese ID del proceso: ";
276                 cin >> id;
277                 cout << "Ingrese nueva prioridad: ";

```



```

278         cin >> nuevaPrioridad;
279         gestorProcesos.modificarPrioridad(id, nuevaPrioridad);
280         break;
281     }
282     case 5:
283         gestorProcesos.mostrarProcesos();
284         break;
285     case 6: {
286         int id;
287         cout << "Ingrese ID del proceso a encolar: ";
288         cin >> id;
289         Proceso* proceso = gestorProcesos.obtenerCabeza();
290         while (proceso != NULL && proceso->identificador != id) {
291             proceso = proceso->siguiente;
292         }
293         if (proceso != NULL) {
294             planificadorCPU.encolarProceso(proceso);
295         } else {
296             cout << "El proceso no fue encontrado (T-T)\n";
297         }
298         break;
299     }
300     case 7:
301         planificadorCPU.desencolarProceso();
302         break;
303     case 8:
304         planificadorCPU.mostrarCola();
305         break;
306     case 9: {
307         int id;
308         cout << "Ingrese ID del proceso a asignar memoria: ";
309         cin >> id;

```

```

310         Proceso* proceso = gestorProcesos.obtenerCabeza();
311         while (proceso != NULL && proceso->identificador != id) {
312             proceso = proceso->siguiente;
313         }
314         if (proceso != NULL) {
315             gestorMemoria.asignarMemoria(proceso);
316         } else {
317             cout << "EL proceso no fue encontrado (T-T)\n";
318         }
319         break;
320     }
321     case 10:
322         gestorMemoria.liberarMemoria();
323         break;
324     case 11:
325         gestorMemoria.verificarEstadoMemoria();
326         break;
327     case 0:
328         cout << "Hasta luego... ^^ \n";
329         break;
330     default:
331         cout << "La opción es inválida (T-T)\n";
332     }
333 } while (opcion != 0);
334
335 return 0;
336 }

```

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

```
===== Sistema de gestion de procesos =====
1. Insertar proceso (^)
2. Eliminar proceso (T-T)
3. Buscar proceso (0.0)
4. Modificar prioridad (^)
5. Mostrar todos los procesos (0.0)
6. Encolar proceso en CPU (^)
7. Desencolar y ejecutar proceso (0.0)
8. Mostrar cola de CPU (0.0)
9. Asignar memoria (^)
10. Liberar memoria (T-T)
11. Verificar estado de memoria (0.0)
0. Salir (T-T)
=====
Ingrese una opción:
```

Insertar proceso:

```
Ingrese ID del proceso: 123456
Ingrese nombre del proceso: proceso1
Ingrese prioridad del proceso: 1
Su proceso fue añadido correctamente (^)
```

Eliminar proceso:

```
Ingrese una opción: 2
Ingrese ID del proceso a eliminar: 123456
Su proceso fue eliminado correctamente (^)
```

```
Ingrese una opción: 2
Ingrese ID del proceso a eliminar: 123321
Su proceso no fue encontrado (T-T)
```

Buscar proceso:

```
Ingrese una opción: 3
Ingrese ID del proceso a buscar: 12345
ID: 12345 | Nombre: Proceso 2 | Prioridad: 1 (0.0)
```

```
Ingrese una opción: 3
Ingrese ID del proceso a buscar: 13324
Su proceso no fue encontrado (T-T)
```

Modificar prioridad:

```
Ingrese una opcion: 4
Ingrese ID del proceso: 12345
Ingrese nueva prioridad: 2
Su prioridad fue modificada correctamente (^)
```

Mostrar procesos:

```
Ingrese una opcion: 5
ID: 12345, Nombre: Proceso 2, Prioridad: 2 (0.0)
```

Encolar proceso en CPU:

```
Ingrese una opcion: 6
Ingrese ID del proceso a encolar: 12345
Su proceso fue encontrado correctamente (^)
```

Desencolar y ejecutar proceso:

```
Ingrese una opcion: 7
Ejecutando proceso: ID 12345, Nombre Proceso 2, Prioridad 2 (0.0)
```

Mostrar cola de CPU:

```
Ingrese una opcion: 8
ID: 54321, Nombre: Proceso 3, Prioridad: 4 (0.0)
ID: 13425, Nombre: Proceso 5, Prioridad: 3 (0.0)
```

Asignar memoria:

```
Ingrese una opcion: 9
Ingrese ID del proceso a asignar memoria: 12345
La memoria fue asignada correctamente (^)
```

Liberar memoria:

```
Ingrese una opcion: 10
La memoria fue liberada correctamente (^)
```

Verificar estado de memoria:

```
Ingrese una opcion: 11
La memoria esta vacia (T-T)
```

```
Ingrese una opcion: 11
El estado actual de la memoria es: (0.0)
ID: 12345, Nombre: Proceso 2, Prioridad: 1
```

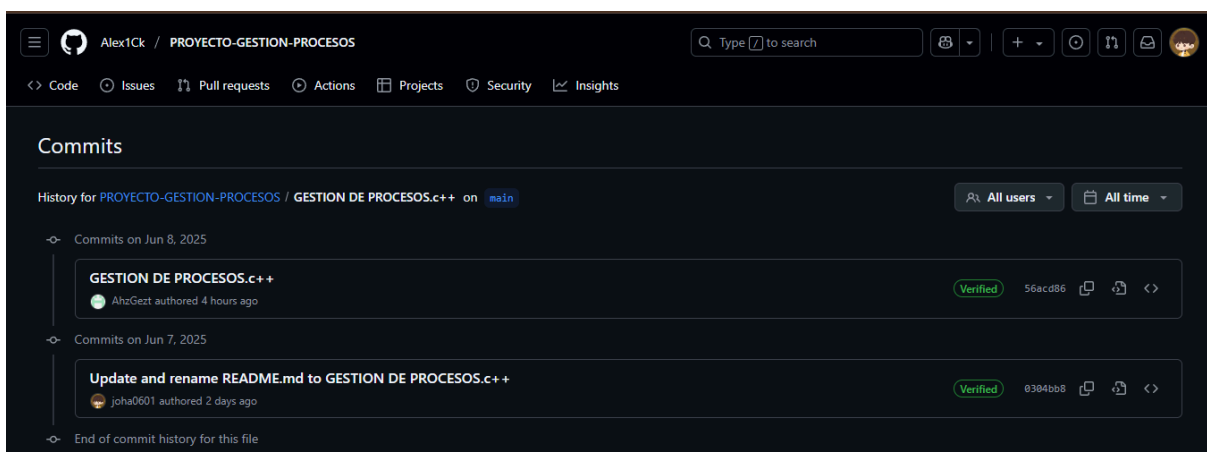
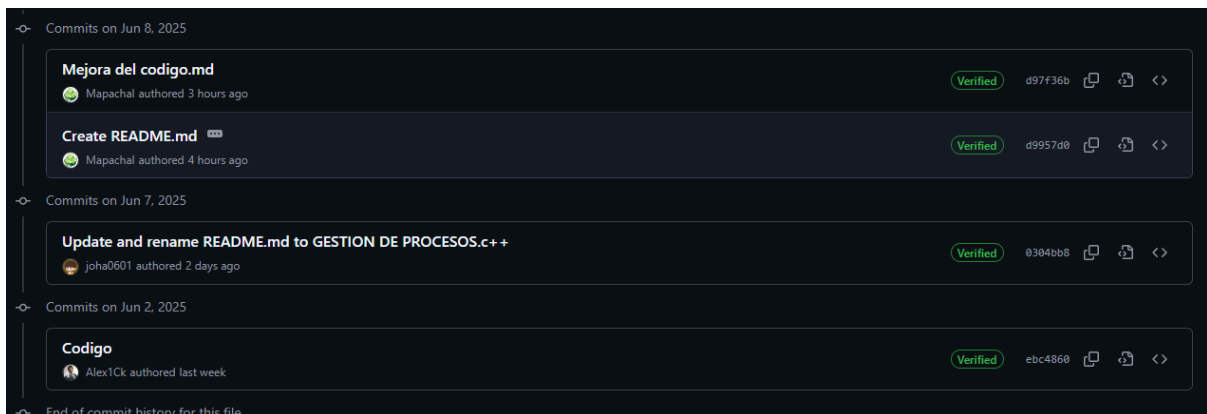
Salir del menú:

```
Ingrese una opcion: 0
Hasta luego... ^^
```

Manual de usuario

<https://docs.google.com/document/d/1SNtS-R6pwOoZpdfE-KA7qZjiDXfdkGKLWS7WNQllwyE/edit?usp=sharing>

Repositorio con Control de Versiones (Capturas de Pantalla)



Commits


History for [PROYECTO-GESTION-PROCESOS](#) / `GESTION DE PROCESOS MEJORADO.c++` on `main`

All users

All time

Commits on Jun 8, 2025


Update `GESTION DE PROCESOS MEJORADO.c++`

 joha0601 authored 5 minutes ago

Verified70657da

<>


Update `GESTION DE PROCESOS MEJORADO.c++`

 BrunoPontecil authored 5 minutes ago

Verifiedc13434d

<>


Update `GESTION DE PROCESOS MEJORADO.c++`

 BrunoPontecil authored 6 minutes ago

Verified16a0451

<>


Update `GESTION DE PROCESOS MEJORADO.c++`

 BrunoPontecil authored 7 minutes ago

Verified2f9a7ec

<>


Update `GESTION DE PROCESOS MEJORADO.c++`

 BrunoPontecil authored 29 minutes ago

Verifiede80e33e

<>

Rename `README.md` to `GESTION DE PROCESOS MEJORADO.c++`

 joha0601 authored 44 minutes ago

Verified42cc91d

<>

Renamed from `README.md`(Browse History)

Plan de Trabajo y Roles Asignados

Todos: código

Alex y Flavio: Informe

Johana y Bruce : Manual de usuario

Jaasiel: Presentación

Link del GitHub

<https://github.com/Alex1Ck/PROYECTO-GESTION-PROCESOS>