GESTION DE PROCESOS

EQUIPO DE DESARROLLO

TORRES HUAMANI ALDAIR ALEXIS

LAOS AYALA JOSEPH KARL

SILVA CRISPIN, YANPOL

JEFERSON ELI RIOS AYALA

```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
```

Inclusión de loibrerías esenciales (iostream para I/o, string, cstdin para system() y cctype para validación de caracteres)

```cpp
// 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
}
```

Función auxiliar para limpieza de datos. Elimina espacios en blanco iniciales (L10-L11) y finales (L13 -L14) de cualquier entrada e texto, previniendo errores de formato.

```cpp
// 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
}
```

Convierte la cadena de texto limia en un valor entero, utilizando un bucle para realizar la conversión carácter por uno de forma manual.

```
27  // Lee un número entero positivo desde el teclado
28  int leerEnteroPositivo(const string &mensaje) {
29      string entrada;  // Variable para almacenar la entrada del usuario
30      while (true) {
31          cout << mensaje;          // Muestra el mensaje
32          getline(cin, entrada);    // Lee la entrada completa
33          entrada = trim(entrada);  // Elimina espacios
34
35          if (entrada.empty()) {    // Si está vacío
36              cout << "Error: Ingrese un número entero positivo.\n";
37              continue;
38          }
39
40          bool valido = true;       // Bandera para validar caracteres
41          for (size_t i = 0; i < entrada.size(); ++i) {
42              if (!isdigit((unsigned char)entrada[i])) {  // Si hay letras o símbolos
43                  valido = false;
44                  break;
45              }
46          }
47
48          if (!valido) {
49              cout << "Error: Solo se permiten números.\n";
50              continue;
51          }
52
53          int valor = stringADecimal(entrada);  // Convierte la cadena a número
54          if (valor <= 0) {
55              cout << "Error: El número debe ser mayor que 0.\n";
56              continue;
57          }
58
59          return valor;  // Devuelve el número válido
60      }
61  }
62
63  //---------------- ESTRUCTURAS PRINCIPALES ----------------//
64
65  // Cada proceso contiene su identificador (PID), nombre, prioridad y un puntero al siguiente
66  struct Proceso {
```

int leerEnteroPositivo(const string &mensaje)

Función de validación crítica. Entra en un bucle while(true) hasta que la entrada es perfecta:

L32-L35	Valida que la entrada no esté vacía.
L37-L47	Valida carácter por carácter que solo haya dígitos (!isdigit), cumpliendo el requisito de entrada numérica.
L50-L54	Convierte la cadena validada y verifica que el valor sea mayor que 0, rechazando PIDs o prioridades nulas o negativas.

```
66  struct Proceso {
67      int pid;                // Identificador único del proceso
68      string nombre;          // Nombre descriptivo del proceso
69      int prioridad;          // Nivel de prioridad (menor número = mayor prioridad)
70      Proceso* siguiente;     // Enlace al siguiente proceso en la lista
71  };
72  Proceso* cabezaProcesos = NULL;   // Puntero al inicio de la lista de procesos
73
74  // Nodo para representar la cola de planificación de CPU
75  struct NodoCola {
76      Proceso* proceso;       // Apunta al proceso asociado
77      NodoCola* siguiente;    // Enlace al siguiente nodo en la cola
78  };
79  NodoCola* cabezaCola = NULL;      // Inicio de la cola (vacía al comienzo)
80
81  // Nodo para representar los bloques de memoria como una pila
82  struct BloqueMemoria {
83      Proceso* proceso;        // Proceso que ocupa el bloque de memoria
84      int tamanio;             // Tamaño asignado al bloque
85      BloqueMemoria* siguiente;// Enlace al siguiente bloque
86  };
87  BloqueMemoria* topeMemoria = NULL; // Puntero al bloque superior de la pila
88
89  //---------------- FUNCIONES DE BÚSQUEDA ----------------//
90
```

L70-L76
struct Proceso

Nodo de la Lista Enlazada Simple (Gestor de Procesos). Contiene la información del proceso (pid, nombre, prioridad) y el puntero Proceso* siguiente para enlazar.

L77
Proceso* cabezaProcesos

Puntero inicial de la Lista Maestra de Procesos.

L80-L83
struct NodoCola

Nodo de la Cola con Prioridad (Planificador). Almacena un puntero al Proceso* existente (para no duplicar datos) y el puntero NodoCola* siguiente.

L84
NodoCola* cabezaCola

Puntero inicial de la Cola de Planificación.

L87-L90

struct BloqueMemoria

BloqueMemoria* topeMemoria

struct BloqueMemoriaNodo de la Pila LIFO (Gestor de Memoria). Almacena el proceso, el tamanio asignado y el puntero BloqueMemoria* siguiente. BloqueMemoria* topeMemoriaPuntero que marca la cima de la Pila (Stack).

L95-L101 Proceso* buscarProcesoPorPID(int pid) Búsqueda

```
92  Proceso* buscarProcesoPorPID(int pid) {
93      Proceso* actual = cabezaProcesos;  // Apunta al primer proceso
94      while (actual != NULL) {
95          if (actual->pid == pid) return actual;  // Si encuentra coincidencia
96          actual = actual->siguiente;             // Avanza al siguiente
97      }
98      return NULL;  // Retorna NULL si no lo encuentra
99  }
100
101 // Verifica si un proceso ya está en la cola del planificador
102 bool estaEnCola(int pid) {
103     NodoCola* actual = cabezaCola;  // Apunta al primer nodo
104     while (actual != NULL) {
105         if (actual->proceso->pid == pid) return true;  // Encontrado
106         actual = actual->siguiente;                    // Avanza
107     }
108     return false;  // No encontrado
109 }
110
111 //---------------- GESTOR DE PROCESOS ----------------//
112
113 // Inserta un nuevo proceso al final de la lista enlazada
114 void insertarProceso() {
115     cout << "\n--- Insertar nuevo proceso ---\n";
116
117     int pid = leerEnteroPositivo("Ingrese el PID: ");  // PID único
118
119     if (buscarProcesoPorPID(pid) != NULL) {            // Verifica duplicados
120         cout << "Error: Ya existe un proceso con ese PID.\n";
121         return;
122     }
123
124     string nombre;
125     cout << "Ingrese el nombre del proceso: ";
126     getline(cin, nombre);
127     nombre = trim(nombre);
```

Recorre la lista desde cabezaProcesos hasta encontrar una coincidencia o llegar al final (NULL).

L113-L149 void insertarProceso() **Inserción al Final**

```
113  // Inserta un nuevo proceso al final de la lista enlazada
114  void insertarProceso() {
115      cout << "\n--- Insertar nuevo proceso ---\n";
116
117      int pid = leerEnteroPositivo("Ingrese el PID: ");  // PID único
118
119      if (buscarProcesoPorPID(pid) != NULL) {            // Verifica duplicados
120          cout << "Error: Ya existe un proceso con ese PID.\n";
121          return;
122      }
123
124      string nombre;
125      cout << "Ingrese el nombre del proceso: ";
126      getline(cin, nombre);
127      nombre = trim(nombre);
128
129      while (nombre.empty()) {  // Evita nombres vacíos
130          cout << "Error: El nombre no puede estar vacío.\n";
131          cout << "Ingrese el nombre del proceso: ";
132          getline(cin, nombre);
133          nombre = trim(nombre);
134      }
135
136      int prioridad = leerEnteroPositivo("Ingrese la prioridad: ");  // Prioridad
137
138      // Crea un nuevo nodo de proceso
139      Proceso* nuevo = new Proceso;
140      nuevo->pid = pid;
141      nuevo->nombre = nombre;
142      nuevo->prioridad = prioridad;
143      nuevo->siguiente = NULL;
144
145      // Inserta el proceso al final de la lista
146      if (cabezaProcesos == NULL)
147          cabezaProcesos = nuevo;
148      else {
149          Proceso* actual = cabezaProcesos;
150          while (actual->siguiente != NULL) actual = actual->siguiente;
151          actual->siguiente = nuevo;
152      }
```

Valida la unicidad del PID (L118-L121) y crea el nuevo nodo. Se inserta al final recorriendo la lista si no está vacía (L144-L146).

L113-L149 void insertarProceso() **Inserción al Final**

```
113  // Inserta un nuevo proceso al final de la lista enlazada
114  void insertarProceso() {
115      cout << "\n--- Insertar nuevo proceso ---\n";
116
117      int pid = leerEnteroPositivo("Ingrese el PID: ");  // PID único
118
119      if (buscarProcesoPorPID(pid) != NULL) {            // Verifica duplicados
120          cout << "Error: Ya existe un proceso con ese PID.\n";
121          return;
122      }
123
124      string nombre;
125      cout << "Ingrese el nombre del proceso: ";
126      getline(cin, nombre);
127      nombre = trim(nombre);
128
129      while (nombre.empty()) {  // Evita nombres vacíos
130          cout << "Error: El nombre no puede estar vacío.\n";
131          cout << "Ingrese el nombre del proceso: ";
132          getline(cin, nombre);
133          nombre = trim(nombre);
134      }
135
136      int prioridad = leerEnteroPositivo("Ingrese la prioridad: ");  // Prioridad
137
138      // Crea un nuevo nodo de proceso
139      Proceso* nuevo = new Proceso;
140      nuevo->pid = pid;
141      nuevo->nombre = nombre;
142      nuevo->prioridad = prioridad;
143      nuevo->siguiente = NULL;
144
145      // Inserta el proceso al final de la lista
146      if (cabezaProcesos == NULL)
147          cabezaProcesos = nuevo;
148      else {
149          Proceso* actual = cabezaProcesos;
150          while (actual->siguiente != NULL) actual = actual->siguiente;
151          actual->siguiente = nuevo;
152      }
```

Valida la unicidad del PID (L118-L121) y crea el nuevo nodo. Se inserta al final recorriendo la lista si no está vacía (L144-L146).

L153-L182 void eliminarProceso() Eliminación

```
153
154         cout << "Proceso agregado correctamente.\n";
155     }
156
157     // Elimina un proceso de la lista según su PID
158     void eliminarProceso() {
159         cout << "\n--- Eliminar proceso ---\n";
160         int pid = leerEnteroPositivo("Ingrese el PID a eliminar: ");
161
162         if (cabezaProcesos == NULL) {  // Si no hay procesos
163             cout << "No hay procesos registrados.\n";
164             return;
165         }
166
167         if (cabezaProcesos->pid == pid) {  // Si el proceso está al inicio
168             Proceso* temp = cabezaProcesos;
169             cabezaProcesos = cabezaProcesos->siguiente;
170             delete temp;  // Libera memoria
171             cout << "Proceso eliminado correctamente.\n";
172             return;
173         }
174
175         Proceso* actual = cabezaProcesos;
176         while (actual->siguiente != NULL && actual->siguiente->pid != pid)
177             actual = actual->siguiente;  // Recorre la lista
178
179         if (actual->siguiente != NULL) {
180             Proceso* temp = actual->siguiente;
181             actual->siguiente = temp->siguiente;
182             delete temp;
183             cout << "Proceso eliminado correctamente.\n";
184         } else {
185             cout << "Error: No se encontró el proceso.\n";
186         }
```

Maneja el caso de eliminación de la cabeza (L160-L165). Para nodos intermedios, busca al nodo anterior al que se quiere eliminar para reajustar los punteros.

Inserción Ordenada (Priority Queue)

L206-L249

void encolarProcesoEnPlanificador()

Lógica central del planificador. Verifica existencia y duplicidad (L211-L218).

```
206
207  // Encola un proceso según su prioridad
208  void encolarProcesoEnPlanificador() {
209      cout << "\n--- Encolar proceso ---\n";
210      int pid = leerEnteroPositivo("Ingrese el PID: ");
211      Proceso* p = buscarProcesoPorPID(pid);
212
213      if (p == NULL) {
214          cout << "Error: No existe un proceso con ese PID.\n";
215          return;
216      }
217      if (estaEnCola(pid)) {
218          cout << "Error: El proceso ya está en la cola.\n";
219          return;
220      }
221
222      NodoCola* nuevo = new NodoCola;
223      nuevo->proceso = p;
224      nuevo->siguiente = NULL;
225
226      // Inserta ordenado por prioridad
227      if (cabezaCola == NULL || cabezaCola->proceso->prioridad > p->prioridad) {
228          nuevo->siguiente = cabezaCola;
229          cabezaCola = nuevo;
230      } else {
231          NodoCola* actual = cabezaCola;
232          while (actual->siguiente != NULL &&
233                 actual->siguiente->proceso->prioridad <= p->prioridad)
234              actual = actual->siguiente;
235          nuevo->siguiente = actual->siguiente;
236          actual->siguiente = nuevo;
237      }
238      cout << "Proceso encolado correctamente.\n";
239  }
240
241  // Desencola el proceso de mayor prioridad (al frente)
242  void desencolarYEjecutarProceso() {
243      cout << "\n--- Desencolar y ejecutar ---\n";
244      if (cabezaCola == NULL) {
245          cout << "La cola está vacía.\n";
246          return;
247      }
248
```

L228-L231

El nuevo nodo se convierte en cabezaCola si su prioridad es menor que la cabeza actual (es decir, mayor prioridad).

L233-L237

El bucle while busca el punto donde el siguiente nodo tenga prioridad menor o igual (<=), deteniéndose para insertar justo antes de que la prioridad sea más alta.

```
253         << " | Prioridad: " << temp->proceso->prioridad << endl;
254     delete temp;
255 }
256
257 // Muestra el contenido de la cola
258 void mostrarColaPlanificador() {
259     cout << "\n--- Cola de planificación ---\n";
260     if (cabezaCola == NULL) {
261         cout << "La cola está vacía.\n";
262         return;
263     }
264     NodoCola* actual = cabezaCola;
265     while (actual != NULL) {
266         cout << "PID: " << actual->proceso->pid
267              << " | Nombre: " << actual->proceso->nombre
268              << " | Prioridad: " << actual->proceso->prioridad << endl;
269         actual = actual->siguiente;
270     }
271 }
```

L253-L267

Desencolamiento

Elimina y libera el nodo apuntado por cabezaCola, pues es, por diseño, el de más alta prioridad.

```
276 void asignarMemoria() {
277     cout << "\n--- Asignar memoria ---\n";
278     int pid = leerEnteroPositivo("Ingrese el PID: ");
279     Proceso* p = buscarProcesoPorPID(pid);
280     if (p == NULL) {
281         cout << "Error: No existe un proceso con ese PID.\n";
282         return;
283     }
284
285     int tamanio = leerEnteroPositivo("Ingrese el tamaño de memoria: ");
286     BloqueMemoria* nuevo = new BloqueMemoria;
287     nuevo->proceso = p;
288     nuevo->tamanio = tamanio;
289     nuevo->siguiente = topeMemoria;
290     topeMemoria = nuevo;
291
292     cout << "Memoria asignada correctamente.\n";
```

276 - 291

PUSH (Apilar)

void asignarMemoria( )

Crea un nuevo nodo. La línea nuevo->siguiente = topeMemoria; es clave: el nuevo nodo apunta al antiguo tope. Luego, topeMemoria = nuevo; lo convierte en el nuevo tope, garantizando la inserción LIFO.

```
294
295 // Libera el último bloque asignado (operación pop)
296 void liberarMemoria() {
297     cout << "\n--- Liberar memoria ---\n";
298     if (topeMemoria == NULL) {
299         cout << "No hay memoria para liberar.\n";
300         return;
301     }
```

```
337
338         opcionPrincipal = leerEnteroPositivo("Seleccione una opción: ");
339
340         switch (opcionPrincipal) {
341         case 1:
342             do {
343                 cout << "\n--- GESTOR DE PROCESOS ---\n";
344                 cout << "[1] Insertar proceso\n";
345                 cout << "[2] Eliminar proceso\n";
346                 cout << "[3] Mostrar procesos\n";
347                 cout << "[4] Volver\n";
348                 opcionSecundaria = leerEnteroPositivo("Opción: ");
349                 if (opcionSecundaria == 1) insertarProceso();
350                 else if (opcionSecundaria == 2) eliminarProceso();
351                 else if (opcionSecundaria == 3) mostrarProcesos();
352                 system("pause");
353                 system("cls");
354             } while (opcionSecundaria != 4);
355             break;
356
357         case 2:
358             do {
359                 cout << "\n--- PLANIFICADOR DE CPU ---\n";
360                 cout << "[1] Encolar proceso\n";
361                 cout << "[2] Ejecutar proceso\n";
362                 cout << "[3] Mostrar cola\n";
363                 cout << "[4] Volver\n";
364                 opcionSecundaria = leerEnteroPositivo("Opción: ");
365                 if (opcionSecundaria == 1) encolarProcesoEnPlanificador();
366                 else if (opcionSecundaria == 2) desencolarYEjecutarProceso();
367                 else if (opcionSecundaria == 3) mostrarColaPlanificador();
368                 system("pause");
369                 system("cls");
370             } while (opcionSecundaria != 4);
371             break;
372
373         case 3:
374             do {
375                 cout << "\n--- GESTOR DE MEMORIA ---\n";
376                 cout << "[1] Asignar memoria\n";
377                 cout << "[2] Liberar memoria\n";
378                 cout << "[3] Mostrar estado\n";
379                 cout << "[4] Volver\n";
380                 opcionSecundaria = leerEnteroPositivo("Opción: ");
381                 if (opcionSecundaria == 1) asignarMemoria();
382                 else if (opcionSecundaria == 2) liberarMemoria();
383                 else if (opcionSecundaria == 3) estadoMemoria();
384                 system("pause");
385                 system("cls");
386             } while (opcionSecundaria != 4);
387             break;
388         }
389     } while (opcionPrincipal != 4);
390     cout << "Saliendo del programa...\n";
391 }
392
```

void menu()	337 - 405	Control de Flujo Principal
L353-L376		Menú Gestor de Procesos. Bucle do-while anidado que llama a insertarProceso(), eliminarProceso() o mostrarProcesos().
L379-L394		Menú Planificador de CPU. Llama a encolarProcesoEnPlanificador() y desencolarYEjecutarProceso().

```
337
338         opcionPrincipal = leerEnteroPositivo("Seleccione una opción: ");
339
340         switch (opcionPrincipal) {
341         case 1:
342             do {
343                 cout << "\n--- GESTOR DE PROCESOS ---\n";
344                 cout << "[1] Insertar proceso\n";
345                 cout << "[2] Eliminar proceso\n";
346                 cout << "[3] Mostrar procesos\n";
347                 cout << "[4] Volver\n";
348                 opcionSecundaria = leerEnteroPositivo("Opción: ");
349                 if (opcionSecundaria == 1) insertarProceso();
350                 else if (opcionSecundaria == 2) eliminarProceso();
351                 else if (opcionSecundaria == 3) mostrarProcesos();
352                 system("pause");
353                 system("cls");
354             } while (opcionSecundaria != 4);
355             break;
356
357         case 2:
358             do {
359                 cout << "\n--- PLANIFICADOR DE CPU ---\n";
360                 cout << "[1] Encolar proceso\n";
361                 cout << "[2] Ejecutar proceso\n";
362                 cout << "[3] Mostrar cola\n";
363                 cout << "[4] Volver\n";
364                 opcionSecundaria = leerEnteroPositivo("Opción: ");
365                 if (opcionSecundaria == 1) encolarProcesoEnPlanificador();
366                 else if (opcionSecundaria == 2) desencolarYEjecutarProceso();
367                 else if (opcionSecundaria == 3) mostrarColaPlanificador();
368                 system("pause");
369                 system("cls");
370             } while (opcionSecundaria != 4);
371             break;
372
373         case 3:
374             do {
375                 cout << "\n--- GESTOR DE MEMORIA ---\n";
376                 cout << "[1] Asignar memoria\n";
377                 cout << "[2] Liberar memoria\n";
378                 cout << "[3] Mostrar estado\n";
379                 cout << "[4] Volver\n";
380                 opcionSecundaria = leerEnteroPositivo("Opción: ");
381                 if (opcionSecundaria == 1) asignarMemoria();
382                 else if (opcionSecundaria == 2) liberarMemoria();
383                 else if (opcionSecundaria == 3) estadoMemoria();
384                 system("pause");
385                 system("cls");
386             } while (opcionSecundaria != 4);
387             break;
388         }
389     } while (opcionPrincipal != 4);
390     cout << "Saliendo del programa...\n";
391 }
392
```

Menú Gestor de Procesos. Bucle do-while anidado que llama a insertarProceso(), eliminarProceso() o mostrarProcesos().

Menú Planificador de CPU. Llama a encolarProcesoEnPlanificador() y desencolarYEjecutarProceso().

Menú Planificador de CPU. Llama a encolarProcesoEnPlanificador() y desencolarYEjecutarProceso().

Menú Gestor de Memoria. Llama a asignarMemoria() y liberarMemoria().

int main() Punto de Entrada

MUCHAS GRACIAS