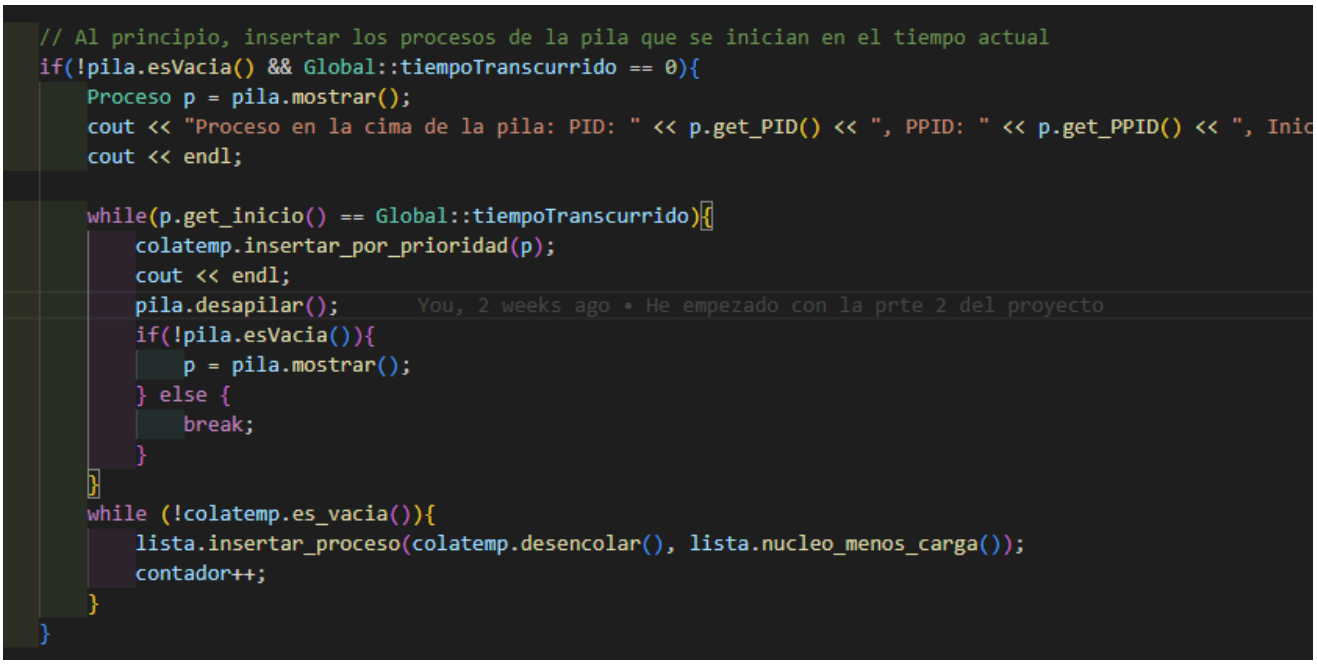
PLANIFICADOR DE PROCESOS DE SISTEMA OPERATIVO

Simulación del control de procesos en un sistema operativo



```
// Al principio, insertar los procesos de la pila que se inician en el tiempo actual
if(!pila.esVacia() && Global::tiempoTranscurrido == 0){
    Proceso p = pila.mostrar();
    cout << "Proceso en la cima de la pila: PID: " << p.get_PID() << ", PPID: " << p.get_PPID() << ", Inic
    cout << endl;

    while(p.get_inicio() == Global::tiempoTranscurrido){
        colatemp.insertar_por_prioridad(p);
        cout << endl;
        pila.desapilar();       You, 2 weeks ago • He empezado con la prte 2 del proyecto
        if(!pila.esVacia()){
            p = pila.mostrar();
        } else {
            break;
        }
    }
    while (!colatemp.es_vacia()){
        lista.insertar_proceso(colatemp.desencolar(), lista.nucleo_menos_carga());
        contador++;
    }
}
```

Alejandro Fernández

Matias Nicolas Vasquez

07/12/2024

G.I.I - UAH

ÍNDICE

INTRODUCCIÓN

En esta práctica, se complementará la simulación de un planificador de procesos de un sistema operativo, desarrollada en la Parte 2 de la práctica 1. El objetivo es analizar los procesos ejecutados para obtener datos clave como los procesos asignados a cada nivel de prioridad, tiempos promedio de ejecución y la carga de procesos por nivel de prioridad. Para ello, se implementarán estructuras de datos adicionales, incluyendo una lista de procesos y un árbol binario de búsqueda que organizará los procesos según su prioridad, permitiendo una gestión eficiente y estructurada.

CLASES Y FUNCIONES

Clase Proceso

La Clase Proceso encapsula toda la información relacionada para representar y administrar los procesos dentro del sistema planificador de procesos. Cada proceso se caracteriza por tener un identificador único (PID), el identificador de su proceso padre (PPID), su momento de inicio, duración, prioridad, y el núcleo al cual está asignado. La clase incluye métodos que permiten la creación del proceso y modificación de sus atributos.

Atributos de la Clase:

- ***static int contador_PID***: Contador estático utilizado para asignar un PID único a los procesos. Este se incrementa cada vez que se crea un nuevo proceso.
- ***int PID***: Identificador único del proceso.
- ***int PPID***: Identificador del proceso padre.
- ***int inicio***: Minuto en el que el proceso debe iniciar su ejecución, representado en minutos desde las 00:00 del sistema.
- ***int tiempoDeVida***: Duración del proceso indica cuánto tiempo en minutos se espera que esté en ejecución.
- ***int prioridad***: Nivel de prioridad del proceso de 0 a 9. Donde 0 representa la mayor prioridad, y 9 representa la menor.
- ***int nucleoAsignado***: Es el núcleo al cual se le asigna un proceso. Se inicia en -1, indicando que no ha sido asignado a ningún núcleo hasta que este se encuentre en ejecución.
- **int tiempoEjecucion**: Tiempo total que el proceso ha estado en ejecución. Se inicia con -1 hasta que el proceso entra en ejecución.

Funciones de la Clase:

- ***Proceso()***: Constructor que inicializa todos los atributos con valores predeterminados. Este constructor es útil para crear procesos sin valores definidos, con PID, PPID, inicio, tiempoDeVida, prioridad, y nucleoAsignado en -1.
- ***Proceso(int ppid, int inicio, int tiempoDeVida, int prioridad)***: Es un constructor que asigna valores específicos a los atributos principales del proceso (PPID, inicio, tiempoDeVida, y prioridad). Recordar que el PID se genera

automáticamente utilizando contador_PID, el nucleoAsignado y tiempoEjecucion se establece en -1, indicando que el proceso aún no ha sido asignado a un núcleo.

- ***Getters y Setter***: Estos métodos permiten acceder y modificar los atributos clave de cada proceso de forma controlada. Los getters ***(get_PPID, get_inicio, get_tiempo_de_vida, get_prioridad, get_nucleo_asignado y get_tiempo_ejecucion)*** devuelven valores que identifican el proceso ya previamente asignados. Y los setters ***(set_PPID, set_inicio, set_tiempo_de_vida, set_prioridad, set_nucleo_asignado y set_tiempo_ejecucion)*** permiten modificar estos atributos, ajustando la información de cada proceso según las necesidades del sistema.

Clase Global

La Clase Global contiene variables globales esenciales para el funcionamiento del sistema de planificación de procesos. Esto permite llevar el control del tiempo de los procesos y proporciona tiempo global para organizar los eventos en el sistema, de esta manera se puede administrar los tiempos y eventos a lo largo de la ejecución.

Atributos de la Clase:

- ***float contadorTiempoEstancia:*** Esta variable almacena la suma total del tiempo qué tardan todos los procesos desde su minuto de inicio hasta el momento en el que se termina su ejecución. se almacenan un float para que a la hora de dividir entre el número de procesos de un resultado con precisión decimal.
- ***int tiempoTranscurrido***: Variable que almacena el tiempo en minutos desde que comenzó la simulación. Este valor se incrementa conforme avanzan las operaciones en el sistema como un conteo acumulativo del tiempo de la ejecución.

Clase Pila

La clase Pila gestiona una estructura de datos de tipo LIFO (Last In, First Out) que permite organizar procesos ordenados y manipulados para facilitar el control de ejecución en un sistema de planificación de procesos. Su funcionalidad incluye apilar y desapilar elementos, ordenar y mostrar procesos, así como modificar la estructura de la pila según las necesidades del sistema.

Atributos de la Clase:

- ***pnodo cima***: Apunta al nodo superior de la pila (NodoPila), accediendo a los procesos en el orden en que fueron apilados.

Funciones de la Clase:

- ***Pila()***: Constructor por defecto. Inicializa cima a NULL, indicando una pila vacía al comienzo.
- ***bool esVacia()***: Verifica si la pila está vacía, devolviendo true si no hay procesos en ella.
- ***void apilar(Proceso p)***: Añade un proceso p a la cima de la pila, creando un nuevo NodoPila y ajustando el puntero cima al nuevo nodo.
- ***void desapilar()***: Elimina el proceso que se encuentra en la cima de la pila. Si la pila no está vacía, la cima se ajusta al siguiente nodo y el nodo anterior se elimina de la memoria.
- ***Proceso mostrar()***: Muestra el proceso que está en la cima de la pila. Si la pila está vacía, devuelve un proceso vacío.
- ***void ordenarPorTiempoInicio()***: Ordena la pila de menor a mayor en base al tiempo de inicio de cada proceso. Utiliza una pila auxiliar para organizar los procesos en el orden deseado.
- ***void mostrarTodos()***: Muestra todos los procesos de la pila, mostrando los atributos de cada proceso en orden desde la cima al fondo.

Clase Nodo Pila

La clase NodoPila es un nodo que se utiliza como componente básico de la estructura Pila. Cada NodoPila almacena un Proceso y un puntero al siguiente nodo, la estructura enlazada permite implementar la lógica de una pila.

Atributos de la Clase:

- ***Proceso proceso***: Contiene la instancia de Proceso almacenada en este nodo, que representa un proceso en el sistema.
- ***NodoPila siguiente****: Apunta al siguiente nodo en la pila. Permite enlazar nodos para mantener el orden y estructura de la pila.

Funciones de la Clase:

- ***NodoPila()***: Constructor del nodo por defecto que inicializa el nodo sin parámetros, nodo vacío.

- ***NodoPila(Proceso p, NodoPila sig = NULL)****: Constructor que inicializa el nodo con un proceso p y un puntero siguiente, que apunta al siguiente nodo en la pila (nodo siguiente = NULL si no se especifica).

Clase Cola

La Clase Cola permite gestionar los procesos en una estructura de cola, donde los procesos se ordenan en base a su prioridad y su orden de llegada. Esta estructura permite almacenar, organizar y manejar procesos para ser ejecutados en un sistema que sigue el orden FIFO (primero en entrar, primero en salir).

Atributos de la Clase

- ***NodoCola* primero***: Apunta al primer nodo de la cola, el cual es el siguiente en salir.
- ***NodoCola* ultimo***: Apunta al último nodo de la cola, al cual se añade cualquier nuevo proceso.
- ***int longitud***: Almacena la cantidad de nodos (procesos) actualmente en la cola.

Funciones de la Clase

- ***Cola()***: Constructor que inicializa la cola como vacía, con primero y último en NULL y longitud en 0.
- ***void encolar(Proceso proceso)***: Añade un proceso al final de la cola. Si la cola está vacía, el nuevo nodo se convierte tanto en el primero como en el último. Aumenta la longitud de la cola.
- ***void insertar_por_prioridad(Proceso proceso)***: Inserta un proceso en la cola y luego la ordena en función de la prioridad del proceso.
- ***Proceso desencolar()***: Retira y devuelve el proceso al frente de la cola, si no está vacía. Si la cola está vacía, devuelve un proceso vacío.
- ***bool es_vacia() const***: Verifica si la cola está vacía, devolviendo true si no contiene nodos y false en caso contrario.
- ***int get_longitud()***: Retorna el número de nodos (procesos) en la cola.
- ***void mostrarCola()***: Muestra en pantalla todos los procesos en la cola. Para cada proceso, muestra sus atributos en formato de tabla. Crea una copia temporal de la cola para restaurar los elementos a su estado original después de imprimir.
- ***void ordenar_por_prioridad()***: Ordena los procesos de la cola de menor a mayor prioridad, utilizando un algoritmo de burbuja para realizar el ordenamiento.

Clase Nodo Cola

La Clase NodoCola representa un nodo individual en la estructura de la clase Cola. Cada nodo contiene un proceso y un puntero al siguiente nodo en la cola. Además, cada nodo tiene un atributo de prioridad, lo que permite a la clase Cola organizar los procesos según este valor cuando se requiere

Atributos de la Clase

- ***NodoCola* siguiente***: Puntero que conecta el nodo actual con el siguiente en la cola, permitiendo el encadenamiento de nodos.
- ***Proceso proceso***: Almacena el proceso asociado a este nodo, incluyendo su PID, PPID, inicio, duración, y prioridad.
- ***int prioridad***: Representa el nivel de prioridad del proceso en el nodo, donde un valor más bajo indica mayor prioridad.

Funciones de la Clase

- ***NodoCola()***: Constructor por defecto que inicializa un nodo con un proceso vacío, siguiente apuntando a NULL y prioridad en 0. Este constructor es útil para crear nodos sin valores definidos inicialmente.
- ***NodoCola(Proceso proceso, int prioridad, NodoCola* sig = NULL)***: Constructor que permite establecer un proceso específico, su prioridad y un puntero al siguiente nodo en la cola.

Clase Núcleo

La clase Núcleo representa el núcleo de procesamiento del sistema planificador de procesos, encargado de manejar la ejecución de procesos y su cola de espera. Un núcleo mantiene un proceso en ejecución, una cola de procesos a la espera, y el tiempo de inicio y finalización del proceso actual. Su implementación permite administrar la asignación y ejecución de procesos dentro del sistema de manera organizada.

Atributos de la Clase

- ***static int Contador_ID***: Contador estático utilizado para asignar un identificador único a cada núcleo a medida que se crean.
- int id: Identificador único del núcleo.
- ***Proceso proceso_en_ejecucion***: El proceso actualmente en ejecución en el

núcleo.

- ***Cola cola_procesos***: La cola de procesos en espera que serán ejecutados en este núcleo cuando esté disponible.
- ***time_t tiempo_inicio***: Tiempo en el que el proceso en ejecución comenzó, basado en el tiempo del sistema.
- ***time_t tiempo_fin***: Tiempo calculado en el que se espera que el proceso en ejecución termine, determinado por el tiempo de inicio y el tiempo de vida del proceso.

Funciones de la Clase

- ***Nucleo()***: Constructor por defecto que inicializa el núcleo con un ID único generado por Contador_ID, y configura los tiempos de inicio y fin en -1 para indicar que el núcleo está libre.
- ***Nucleo(int id, Proceso proceso)***: Constructor que permite crear un núcleo con un identificador específico y un proceso inicial en ejecución. Los tiempos de inicio y fin se calculan automáticamente.
- ***void add_proceso(Proceso proceso)***: Añade un proceso al núcleo. Si el núcleo está libre, lo asigna a proceso_en_ejecucion; si no, el proceso es agregado a la cola de espera con base en su prioridad.
- ***void eliminar_proceso()***: Elimina el proceso en ejecución si ha terminado, desencola el siguiente proceso de la cola y lo asigna al núcleo. Si no hay procesos en la cola, se mantiene el núcleo libre.
- ***void detalles_proceso() / detalles_proceso(bool i)***: Muestra los detalles del proceso en ejecución en el núcleo. La función con parámetro bool i permite indicar si el proceso ha iniciado (true) o ha finalizado (false), proporcionando información del estado del proceso en el núcleo.
- ***void detalles_nucleo()***: Muestra los detalles completos del núcleo, incluyendo el proceso en ejecución y la cola de espera. Si no hay proceso en ejecución o la cola está vacía, se muestra un mensaje indicándolo.

Clase Lista

La Clase Lista representa una lista doblemente enlazada de núcleos. Cada núcleo en la lista se representa a través de un nodo (NodoLista), permitiendo la gestión de los núcleos en el sistema planificador de procesos. Esta estructura permite la inserción, eliminación, y búsqueda de núcleos, así como la administración de los procesos en cada núcleo y la

gestión de la carga de trabajo.

Atributos de la Clase

- ***NodoLista* primero***: Puntero al primer nodo de la lista, donde se almacena el núcleo inicial.
- ***NodoLista* ultimo***: Puntero al último nodo de la lista, donde se almacena el núcleo final.
- i***nt longitud***: Almacena el número total de núcleos en la lista, lo que también representa el número de núcleos operativos en el sistema.

Funciones de la Clase

- ***Lista()***: Constructor que inicializa la lista, asignando nullptr a primero y ultimo y estableciendo la longitud en 0.
- ***bool es_vacia()***: Verifica si la lista está vacía. Retorna true si no hay nodos en la lista.
- ***get_longitud()***: Retorna la longitud de la lista, que equivale al número de núcleos operativos en el sistema.
- ***void ordenar_menor_mayor()***: Ordena los núcleos en la lista de menor a mayor utilizando un algoritmo de burbuja modificado. Cambia el orden de los nodos y asegura que los punteros se mantengan en la estructura correcta.
- ***void insertar_nucleo()***: Crea un núcleo y lo añade al final de la lista. Si la lista está vacía, inicializa el primer y último nodo al nuevo núcleo.
- ***void eliminar(int posicion)***: Elimina el núcleo en la posición indicada. Ajusta los punteros de los nodos adyacentes para preservar la integridad de la lista.
- ***void insertar_proceso(Proceso proceso, int posicion = 0)***: Inserta un proceso en el núcleo de la posición especificada, usando el método add_proceso del núcleo.
- ***void eliminar_proceso(int posicion)***: Elimina el proceso en ejecución en el núcleo en la posición dada.
- **int nucleo_menos_carga(bool imprimir=false)**: Encuentra el núcleo con menos carga de procesos en su cola de espera. Si imprimir es true, muestra información sobre el núcleo con menor carga. Si todos los núcleos tienen más de dos procesos en espera, crea un nuevo núcleo e inserta el proceso en el nuclo nuevo,
- ***int nucleo_mas_carga(bool imprimir=false)***: Encuentra el núcleo con mayor carga de procesos en su cola de espera. Si imprimir es true, muestra información sobre el núcleo con mayor carga.
- ***void estado_nucleo(int posicion)***: Muestra la información del núcleo en la

posición indicada. Utiliza el método detalles_nucleo de Nucleo.

- ***void mostrar_estado_nucleos()***: Muestra el estado de todos los núcleos en la lista, iterando desde el primer nodo hasta el último.
- ***Nucleo coger(int n)***: Retorna el núcleo en la posición especificada. Si la posición es inválida, muestra un mensaje y retorna un núcleo vacío.
- ***NodoLista* obtener_nodo(int posicion):*** devuelve un puntero al nodo que se encuentra en la posición dada dentro de la lista.

Clase Nodo Lista

La clase NodoLista representa un nodo individual en una lista doblemente enlazada de núcleos. Cada NodoLista contiene un núcleo y punteros a los nodos anterior y siguiente en la lista, lo que permite la navegación en ambas direcciones.

Atributos de la Clase

- ***Nucleo nucleo***: Contiene el núcleo asociado a este nodo.
- ***NodoLista* siguiente***: Puntero al siguiente nodo en la lista.
- ***NodoLista* anterior***: Puntero al nodo anterior en la lista.

Funciones de la Clase

- ***NodoLista()***: Constructor por defecto que inicializa los punteros siguiente y anterior en nullptr, y crea un núcleo vacío.
- ***NodoLista(Nucleo nucleo, NodoLista siguiente = nullptr, NodoLista anterior = nullptr)****: Constructor que permite inicializar el nodo con un núcleo específico y opcionalmente enlazarlo a otros nodos en posiciones siguiente y anterior.

Clase Lista Procesos

La Clase ListaProcesos gestiona una estructura de datos tipo lista doblemente enlazada, diseñada para almacenar y manipular procesos terminados dentro del sistema de planificación. Su diseño permite un acceso ordenado a los procesos, así como la inserción, eliminación y ordenamiento eficiente según diversos criterios.

Atributos de la Clase

- ***NodoLista* primero***: Puntero al primer proceso de la lista.
- ***NodoLista* ultimo***: Puntero al último proceso de la lista.
- ***int longitud***: Almacena el número total de procesos en la lista.

Funciones de la Clase

- ***ListaProcesos()***: El constructor inicializa la lista vacía, con primero y último en NULL y longitud en 0.
- ***bool es_vacia()***: Verifica si la lista está vacía. Retorna true si no hay procesos en la lista.
- ***get_longitud()***: Retorna la longitud de la lista, que equivale al número de procesos terminados.
- ***void eliminar_proceso(int posicion)***: Elimina el nodo en la posición indicada y ajusta los punteros anterior y siguiente de los nodos vecinos.
- ***void insertar_proceso(Proceso proceso, int posicion = 0)***: Inserta un proceso en la lista con la posición especificada.
- ***void eliminar_proceso(int posicion)***: Elimina el nodo en la posición indicada y ajusta los punteros anterior y siguiente de los nodos vecinos.
- **Proceso primP() const, Procesos ultP() const**: Devuelve el primer y último proceso de la lista respectivamente.
- ***Proceso coger(int n)***: Retorna el proceso en la posición especificada

Clase Nodo Lista Procesos

La Clase NodoListaProcesos representa un nodo individual dentro de la ListaProcesos. Cada nodo almacena un proceso y punteros que lo conectan con sus nodos adyacentes.

Atributos de la Clase

- ***Proceso proceso***: Contiene el proceso almacenado a este nodo.

- ***NodoListaProcesos* siguiente***: Puntero al siguiente nodo en la lista.
- ***NodoListaProcesos* anterior***: Puntero al nodo anterior en la lista.

Funciones de la Clase

- ***NodoListaProcesos()***: Constructor por defecto que inicializa los punteros siguiente y anterior en nullptr.
- ***NodoListaProcesos(Proceso proceso, NodoListaProceso* sig = nullptr, NodoListaProcesos* ant = nullptr)***: Constructor que permite inicializar el nodo con un proceso específico y enlazarlo a otros nodos en posiciones siguiente y anterior.

Clase BST

La Clase BST representa un árbol binario de búsqueda que organiza procesos según su nivel de prioridad. Cada nodo del árbol almacena una lista de procesos que comparten la misma prioridad, permitiendo una gestión eficiente de los procesos mediante búsquedas, inserciones y recorridos.

Atributos de la Clase

- **NodoBST* raiz**: Puntero al nodo raíz del árbol. Este nodo actúa como punto de entrada al árbol binario de búsqueda.

Funciones de la Clase

- **BST(NodoBST* raiz)**: Constructor que inicializa el árbol con un nodo raíz específico.
- **BST(ListaProcesos lproc, NodoBST* hIz = nullptr, NodoBST* hDer = nullptr, int prioridad = 5)**: Constructor que crea un árbol con una lista de procesos en el nodo raíz y una prioridad dada, con hijos izquierdo y derecho opcionales.
- **void insertar_arbol(Proceso proc, NodoBST*& nodo)**: Inserta un proceso en el árbol. Si no existe un nodo con la misma prioridad, se crea uno nuevo. Si ya existe un nodo con esa prioridad, el proceso se agrega a la lista de ese nodo.
- **void verInorden(NodoBST* nodo)**: Muestra el contenido del árbol en orden (inorden), es decir, primero los nodos del subárbol izquierdo, luego el nodo actual, y finalmente los nodos del subárbol derecho.
- **void buscar(int prioridad, NodoBST* nodo)**: Busca un nodo con la prioridad especificada en el árbol. Si se encuentra, muestra los procesos asociados a ese

nodo.

- **void mostrarNiveles(NodoBST* nodo)**: Muestra los niveles del árbol que contienen procesos, sin importar la prioridad.
- **float tiempoPromedioProcesos(int prioridad, NodoBST* nodo)**: Calcula el tiempo promedio de ejecución de los procesos con la prioridad especificada.
- **void mostrar_tiempo_promedio_procesos_prioridad(NodoBST* nodo)**: Muestra el tiempo promedio de ejecución de los procesos en cada nivel de prioridad del árbol.
- **void nivelesMayorMenorProcesos(NodoBST* nodo, vector<NodoBST*>& nodosMayor, vector<NodoBST*>& nodosMenor, int& maxProcesos, int& minProcesos)**: Determina los nodos con la mayor y menor cantidad de procesos en el árbol, y los muestra.

Clase Nodo BST

La Clase NodoBST representa un nodo individual del árbol binario de búsqueda. Cada nodo almacena una lista de procesos que comparten la misma prioridad, así como punteros a sus hijos izquierdo y derecho.

Atributos de la Clase

- **ListaProcesos listaProc**: Lista de procesos asociados al nodo, todos con la misma prioridad.
- **NodoBST* hi**: Puntero al hijo izquierdo del nodo.
- **NodoBST* hd**: Puntero al hijo derecho del nodo.
- **int prioridad**: Nivel de prioridad de los procesos almacenados en el nodo.

Funciones de la Clase

- **NodoBST(ListaProcesos lproc, NodoBST* hIz = nullptr, NodoBST* hDer = nullptr, int prioridad)**: El constructor inicializa un nodo con una lista de procesos, hijos opcionales, y un nivel de prioridad específico.
- ***Getters y Setter***: Los getters permiten acceder a los atributos del nodo: **get_lista()** devuelve la lista de procesos, **get_izquierdo()** y **get_derecho()** devuelven los punteros a los hijos izquierdo y derecho, respectivamente, y **get_prioridad()** devuelve el nivel de prioridad del nodo. Los setters permiten modificar los atributos del nodo: **set_izquierdo(NodoBST* hIz)** y **set_derecho(NodoBST* hDer)** asignan los hijos izquierdo y derecho, mientras que **set_prioridad(int**

prioridad) establece el nivel de prioridad.

Clase Planificador

La Clase Planificador es responsable de gestionar la simulación del planificador de procesos. Contiene las funciones necesarias para manejar los procesos en el sistema, como su carga, ejecución, y monitoreo a través de una serie de estructuras de datos (Pila, Cola, Lista, BST). El planificador organiza y gestiona los procesos asignados a los núcleos, maneja el tiempo de ejecución y permite interactuar con el sistema a través de un menú de opciones.

Atributos de la Clase

- **Pila pila**: Pila donde se almacenan los procesos por ejecutar, organizados por el tiempo de inicio.
- **Proceso p**: Instancia de la clase Proceso, utilizada temporalmente para manipular procesos en diferentes funciones.
- **Lista lista**: Lista que contiene los núcleos del sistema y los procesos que están siendo ejecutados o esperando.
- **Cola colatemp**: Cola temporal utilizada para manejar los procesos que van a ser insertados en los núcleos disponibles.
- **BST arbolProcesos**: Árbol binario de búsqueda donde se almacenan los procesos, organizados por prioridad.
- **int contador**: Contador utilizado para llevar el registro de la cantidad de procesos que han sido gestionados.
- **int tiempoTranscurrido**: Variable que representa el tiempo transcurrido en el sistema, en minutos. Es utilizada para simular el paso del tiempo en el planificador.
- **float contadorTiempoEstancia**: Contador que acumula el tiempo de estancia de los procesos en el sistema operativo.

Funciones de la Clase

- **void mostrar_tiempo()**: Muestra el tiempo transcurrido en formato mm:ss. Utiliza la variable **tiempoTranscurrido** y la divide entre minutos y segundos.
- **void iniciar_nucleos()**: Inicializa los núcleos del sistema añadiendo tres núcleos a la lista de núcleos. Llama a la función **insertar_nucleo** de la clase Lista.
- **void cargar_procesos()**: Carga una serie de procesos predefinidos en la pila,

ordenándolos por el tiempo de inicio. Cada proceso es apilado en la pila usando la función **apilar** de la clase Pila.

- **void introducir_proceso()**: Permite al usuario introducir un nuevo proceso. El proceso se valida y se apila en la pila, ordenando la pila por el tiempo de inicio. El proceso debe tener un tiempo de inicio mayor o igual al tiempo actual.
- **void mostrar_procesos()**: Muestra todos los procesos presentes en la pila utilizando la función **mostrarTodos** de la clase Pila.
- **void borrar_procesos()**: Elimina todos los procesos de la pila, vaciando la estructura de la pila. Si la pila está vacía, muestra un mensaje indicando que no hay procesos.
- **void aumentar_tiempo_sistema()**: Aumenta el tiempo del sistema en los minutos especificados por el usuario. Durante el proceso, se insertan en los núcleos los procesos que comienzan en el tiempo actual, se eliminan los procesos terminados y los núcleos vacíos, y se asignan los nuevos procesos a los núcleos según su carga utilizando la función **nucleo_menos_carga**.
- **void mostrar_estado_nucleos():** Muestra el estado actual de los núcleos, incluyendo el número de núcleos operativos y los procesos que están asignados a cada núcleo. Llama a la función **mostrar_estado_nucleos** de la clase Lista.
- **void nucleo_menos_mas_carga()**: Consulta y muestra el núcleo con la menor y mayor carga utilizando las funciones **nucleo_menos_carga** y **nucleo_mas_carga** de la clase Lista.
- **void numero_nucleos_operativos()**: Muestra el número de núcleos operativos, que es igual a la longitud de la lista de núcleos. Llama a la función **get_longitud** de la clase Lista.
- **void introducir_proceso_BST()**: Permite al usuario introducir un nuevo proceso directamente en el árbol de procesos (BST), validando que la prioridad esté dentro del rango adecuado y que el tiempo de inicio sea válido.
- **void mostrar_procesos_BST()**: Muestra los procesos presentes en el árbol de procesos (BST) en orden, utilizando el recorrido en inorden con la función **verInorden** de la clase BST.
- **void mostrar_procesos_BST_prioridad()**: Permite al usuario ingresar una prioridad específica y muestra los procesos de esa prioridad en el árbol de procesos (BST), utilizando la función **buscar** de la clase BST.
- **void mostrar_niveles_BST():** Muestra los niveles de prioridad registrados en el árbol de procesos (BST) que tienen al menos un proceso asociado, utilizando la función **mostrarNiveles** de la clase BST.
- **void mostrar_niveles_BST_mayor_menor()**: Muestra los niveles con mayor y

menor número de procesos en el árbol de procesos (BST), utilizando la función **nivelesMayorMenorProcesos** de la clase BST.

- **void mostrar_tiempo_promedio_procesos_prioridad_insertada()**: Permite al usuario ingresar una prioridad específica y muestra el tiempo promedio de ejecución de los procesos con esa prioridad, utilizando la función **tiempoPromedioProcesos** de la clase BST.
- **void mostrar_tiempo_promedio_procesos_prioridad()**: Muestra el tiempo promedio de ejecución de los procesos en cada nivel de prioridad, recorriendo el árbol de procesos (BST) y utilizando la función **mostrar_tiempo_promedio_procesos_prioridad** de la clase BST.
- **void simular_ejecucion_procesos()**: Simula la ejecución de los procesos, avanzando el tiempo del sistema y ejecutando los procesos en los núcleos disponibles. Los procesos que terminan son eliminados de los núcleos, y los núcleos vacíos son también eliminados. Se muestra el estado de los núcleos y el tiempo medio de estancia de los procesos al finalizar.

DETALLES Y JUSTIFICACIONES

TADs creados

En esta implementación se han creado varios Tipos Abstractos de Datos (TADs) que facilitan la organización y gestión de los procesos dentro del sistema de planificación. Los TADs utilizados son los siguientes:

- **Pila**: Una estructura de datos que sigue el principio LIFO (Last In, First Out), utilizada para almacenar y manejar los procesos.
- **Cola**: Estructura de datos tipo FIFO (First In, First Out), utilizada para gestionar los procesos en espera de ser ejecutados por los núcleos.
- **Lista**: Esta estructura por defecto almacena los núcleos del sistema. Es doblemente enlazada y permite una flexible manipulación de los datos a la hora de insertar y eliminar elementos.
- **ListaProcesos**: Una lista doblemente enlazada que organiza los procesos de forma ordenada y permite gestionarlos de manera eficiente en relación con sus prioridades.
- **BST (Árbol Binario de Búsqueda)**: Estructura que permite almacenar procesos en nodos según sus prioridades, y facilita la búsqueda, inserción y eliminación de procesos.
- **Nodo**: Estructura básica que contiene un proceso o una lista, y sus referencias a otros nodos en la estructura de datos (por ejemplo, NodoPila, NodoCola, NodoListaProcesos y NodoBST).

Definición de TAD

A continuación se detallan los TADs y sus funciones:

- **Pila**:
    - Gestiona los procesos que aún no se han ejecutado, apilando y desapilando procesos en el orden de llegada.
    - Los métodos de la pila devuelven procesos en el orden LIFO, es decir, el último proceso agregado será el primero en ejecutarse.
- **Cola**:
    - Almacena los procesos en espera para ser ejecutados por los núcleos. Los

procesos son gestionados según su orden de llegada (FIFO).
    - Los métodos de la cola devuelven el primer proceso agregado (el más antiguo).
- **Lista:**
    - La lista gestiona los núcleos operativos del sistema, permitiendo insertar nuevos núcleos o eliminarlos dinámicamente según la carga del sistema.Cada núcleo en la lista se representa por un nodo que almacena un proceso en ejecución y una colla de procesos en espera.
    - La lista incluye métodos para insertar procesos en el núcleo disponible, eliminar núcleos sin carga y consultar el núcleo con menor o mayor carga.
- **ListaProcesos**:
    - Mantiene los procesos de forma ordenada por prioridad dentro de cada nodo de la estructura de árbol (BST). Permite la inserción, eliminación y modificación de procesos.
    - Devuelve la lista de procesos ordenada según la prioridad, facilitando el acceso y la manipulación de los procesos en la lista.
- **BST (Árbol Binario de Búsqueda)**:
    - Almacena nodos que contienen listas de procesos, organizados por su prioridad. Permite buscar, insertar y eliminar procesos según su prioridad.
    - Métodos como buscar e insertar devuelven valores de estado (si se encuentra el proceso o se inserta exitosamente).
- **Nodo**:
    - Representa una unidad de almacenamiento dentro de las estructuras de datos (Pila, Cola, ListaProcesos y BST). Contiene un proceso y referencias a otros nodos.
    - Los nodos no devuelven un valor directo, sino que facilitan el enlace entre elementos de la estructura.

Solución adoptada

Una de las principales soluciones adoptadas en este sistema fue el uso de un **Árbol Binario de Búsqueda (BST)** para gestionar los procesos en función de sus prioridades. Este enfoque permite tener acceso a los procesos terminados según su prioridad y gestionar eficientemente las operaciones de inserción, eliminación y búsqueda. Con el uso del BST, conseguimos:

- **Ordenar los procesos por prioridad**: Los procesos son insertados en el árbol según su nivel de prioridad, garantizando que se mantenga un acceso rápido y ordenado a los procesos con mayor prioridad.
- **Búsquedas rápidas**: El BST permite encontrar rápidamente un nodo con una prioridad específica, lo que mejora el rendimiento cuando se requiere acceder a los procesos de una prioridad determinada.
- **Manejo eficiente de la memoria**: Al insertar procesos en el árbol, se agrupan en nodos con la misma prioridad, lo que facilita cualquier operación de búsqueda de procesos y la optimización de la carga de trabajo.

Una dificultad importante fue garantizar que el árbol mantuviera un equilibrio adecuado al insertar y eliminar nodos de lista de procesos. Si bien el BST asegura un tiempo de ejecución logarítmico en las operaciones de búsqueda pues sabemos como base el número máximo de prioridad para saber como tener el máximo rendimiento de la búsqueda binaria, aun así la inserción de múltiples nuevos nodos con prioridades más grandes provocaría que la gestión de las listas de procesos dentro de los nodos requiere de un diseño cuidadoso para evitar problemas de desequilibrio y garantizar un buen rendimiento.

Diseño de la relación entre TAD implementados

Primero, consideremos el Tipo Abstracto de Datos (TAD) de pila, que se utiliza para almacenar los procesos antes de que sean ejecutados. Una vez que los procesos salen de esta estructura, se insertan en los núcleos de procesamiento, ya sea directamente o en la cola de procesos pendientes asociada a dichos núcleos.

Estos núcleos están organizados dentro de una lista dinámica cuya cantidad puede aumentar o disminuir en función de las necesidades del sistema. Este enfoque garantiza una asignación eficiente de recursos según la carga de trabajo.

Cuando los procesos concluyen su ejecución en un núcleo, se trasladan a una lista específica de procesos correspondiente a su prioridad. Esta lista está asociada al nodo pertinente dentro de un árbol binario de búsqueda, que organiza los procesos de acuerdo con sus prioridades, optimizando así la gestión y recuperación de la información.

Explicación de métodos descartados

A continuación se explican aquellos métodos usados en el programa considerados como más complejos o que son cruciales en la correcta ejecución del programa.

void aumentar_tiempo_sistema()

Este método de la clase Planificador es el encargado de simular el paso del tiempo en el sistema. No toma ningún parámetro de entrada, ya que el parámetro que necesita se pedirá por consola al usuario al ejecutar la función.

Antes de entrar en el bucle de tipo ¨for¨ para ejecutar el número de ciclos correspondiente a la cantidad de minutos que se quiere aumentar, se comprueba que el número de minutos no sea cero o negativo. Una vez hecho esto se establecen los parámetros para el bucle for de manera que se ejecute una vez por minuto. Justo después del inicio del bucle for encontramos una sentencia if que comprueba si la ejecución general del programa ha terminado. Esto lo hace chequeando si la pila está vacía, la cantidad de núcleos se han reducido al número mínimo y el primer núcleo de la lista este vacío, al igual que su cola de espera de procesos. Si todo eso se cumple significa que ya no queda ningún proceso por ejecutar y se dará por terminada la ejecución. Finalmente imprimirá el tiempo medio de estancia en el sistema operativo y saldrá del bucle.

Si la ejecución no ha terminado, sigue otra comprobación if qué solo se ejecutará si nos encontramos en el minuto cero. Esto es por qué según el diseño, antes de continuar con las siguientes funciones tiene que haber procesos en los núcleos. Por eso esta sentencia extraerá los procesos de la pila que se inicien en el minuto cero y los insertará en los núcleos correspondientes, creando más núcleos si fuese necesario.

Una vez tenemos los núcleos cargados, se comprueba si los procesos dentro de estos terminan en el minuto actual, si es así, se eliminarán los procesos correspondientes y se iniciaran aquellos que estén en la cola de espera del núcleo. Tras esto se recorre la lista de núcleos y se eliminan aquellos núcleos que estén vacíos y sin carga.

Una vez que se ha comprobado que todos los procesos a terminar se hayan eliminado y que todos los núcleos excedentes han sido terminados se procede a insertar los procesos de la pila que inicien en el minuto actual y solo si fuese necesario se crearían más núcleos. Y finalmente se muestra el estado de los núcleos en ese momento.

El método **void simular_ejecucion_procesos()** de la misma clase, tiene una estructura

similar, con las diferencias de que el bucle principal es un while cuya condición para continuar es la condición opuesta a la condición utilizada para finalizar dentro del método **aumentar_tiempo_sistema(),** este bucle contiene una sentencia qué pone a dormir la ejecución durante 0.25 segundos por ciclo (para que la simulación sea más visual), y la finalización, donde se muestra el tiempo medio de estancia, está tras el bucle.

int nucleo_menos_carga(bool imprimir)

El método **nucleo_menos_carga()** de la clase Lista tiene como propósito identificar el núcleo dentro de una lista que posea la menor carga de procesos. Si todos los núcleos existentes tienen más de dos procesos en espera, la función se encarga de crear un nuevo núcleo para mantener un equilibrio en la distribución de la carga. Además, si se pasa el argumento imprimir con el valor true, la función muestra el estado del núcleo con menor carga, lo que facilita una mejor visibilidad del sistema. En esencia, este método es clave para la gestión eficiente de recursos, ya que no solo distribuye la carga de manera equitativa, sino que también garantiza la escalabilidad del sistema al añadir nuevos núcleos cuando sea necesario.

El proceso comienza verificando si la lista de núcleos está vacía. En este caso, la función retorna -1, indicando que no existen núcleos disponibles para procesar. Luego, se inicializan diversas variables esenciales para la operación, como una bandera llamada nucleoEncontrado, que indica si se ha localizado un núcleo con un PID igual a -1. También se definen las variables posicion y posicionMenosCarga, que rastrean la posición actual y la del núcleo con la menor carga, respectivamente, junto con otras variables como pid y menorCarga para almacenar información sobre los núcleos durante el recorrido.

En un primer recorrido de la lista, la función busca un núcleo que contenga un proceso con PID -1. Si se encuentra, las variables relevantes se actualizan y se marca la bandera nucleoEncontrado como verdadera. Si no se encuentra ningún núcleo con esta característica, la posición de búsqueda se reinicia a 0 para llevar a cabo un segundo recorrido, cuyo objetivo es identificar el núcleo con la menor cantidad de procesos en espera. Durante este recorrido, se registra la posición del núcleo con menor carga para un posible retorno posterior.

Si todos los núcleos tienen más de dos procesos en espera, la función crea un nuevo núcleo en la lista, aumentando así la capacidad de procesamiento. Una vez añadido el nuevo núcleo, se invoca recursivamente la misma función para determinar cuál de todos

los núcleos, incluido el recién creado, tiene la menor carga. Este enfoque dinámico asegura que el sistema pueda adaptarse a las necesidades operativas.

Por último, si se especifica que se desea imprimir el estado del núcleo con menor carga mediante el argumento imprimir, la función muestra esta información para facilitar el monitoreo. Una vez completados todos los pasos, la función retorna la posición del núcleo con menor carga, permitiendo su uso en operaciones posteriores.

void eliminar_proceso()

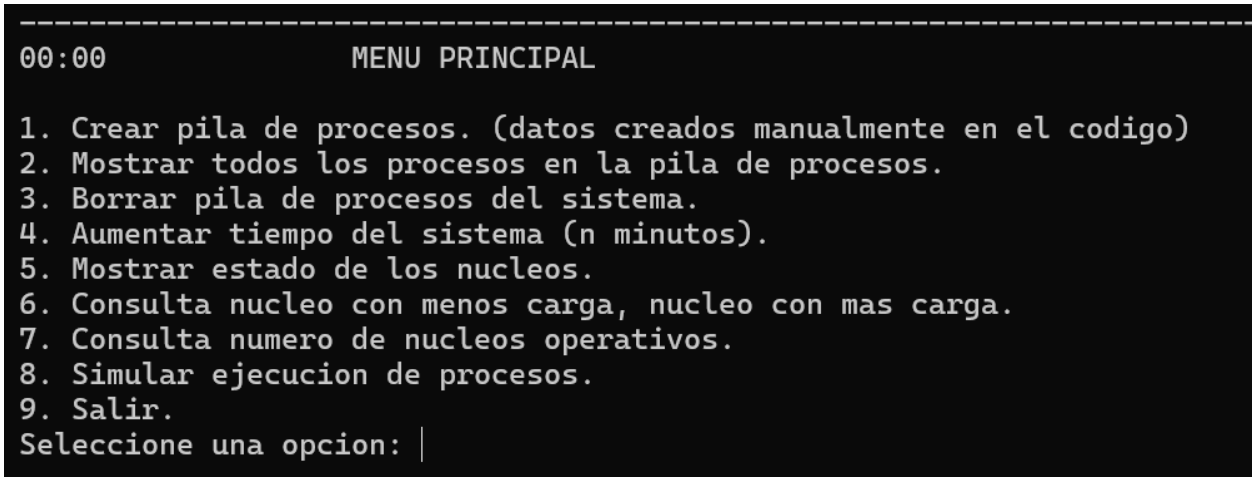
El método **eliminar_proceso()** de la clase Núcleo se encarga de gestionar la finalización de un proceso en ejecución dentro de un núcleo. Dependiendo de las condiciones del proceso actual y de la cola de procesos asociada al núcleo, el método toma decisiones como finalizar el proceso en ejecución, iniciar un nuevo proceso desde la cola o liberar el núcleo si no quedan procesos por gestionar. Esta funcionalidad asegura una administración eficiente de los recursos y una correcta transición entre procesos en el planificador.

En el primer caso, el método actúa cuando hay un proceso en ejecución, la cola de procesos no está vacía y el tiempo de ejecución del proceso actual ha finalizado. En esta situación, el método actualiza el contador del tiempo de estancia del planificador, registra el tiempo de ejecución del proceso terminado, muestra sus detalles, y lo inserta en el árbol de procesos del planificador. Posteriormente, inicia un nuevo proceso desde la cola de procesos, garantizando la continuidad de las operaciones en el núcleo.

En el segundo caso, si no hay un proceso en ejecución, pero la cola de procesos no está vacía, el método simplemente inicia un nuevo proceso desde dicha cola. Este escenario permite que el núcleo permanezca activo y productivo, asignando trabajo de inmediato sin necesidad de esperar a condiciones adicionales.

Por último, en el tercer caso, si hay un proceso en ejecución, la cola de procesos está vacía, y el tiempo de ejecución del proceso actual ha concluido, el método realiza varias acciones. Estas incluyen actualizar el contador del tiempo de estancia del planificador, registrar el tiempo de ejecución del proceso, mostrar los detalles del proceso terminado, e insertar el proceso en el árbol de procesos del planificador. Luego, el proceso en ejecución se reinicia a un estado vacío utilizando un proceso auxiliar.

COMPORTAMIENTO DEL PROGRAMA

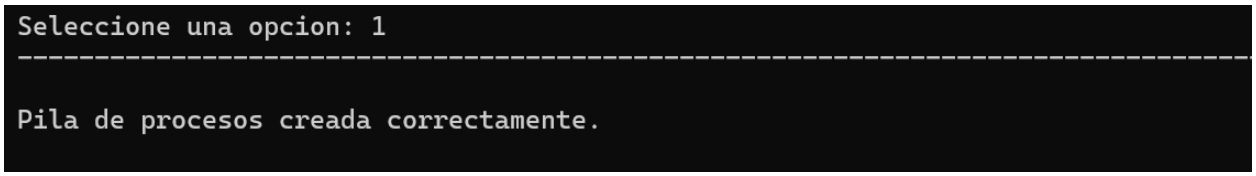


Descripción de las Opciones en el Menú:

1. **Crear pila de procesos**

Esta opción inicializa una pila de procesos y se crean Procesos con datos creados manualmente en el código y se almacenan en ella. Los procesos se agregan usando ***pila.apilar(Proceso p)***, y luego la pila se ordena por el tiempo de inicio de los procesos con ***pila.ordenarPorTiempoInicio()***. Dado que la pila se utiliza en toda instancia de la simulación, es importante ejecutar esta opción para tener una pila creada y ejecutar posteriormente las opciones 4 y/o 8.

Una vez el usuario haya seleccionado la opción 1, se ejecuta y muestra un mensaje de confirmación indicando que la pila de procesos se ha creado y ordenado correctamente.



2. **Crear e introducir proceso en la pila**

Esta opción permite crear e insertar un proceso directamente en la pila. El usuario ingresa los datos del proceso, como el PID del proceso padre, el tiempo de inicio, el tiempo de vida y la prioridad del proceso. Una vez que se introducen estos datos, el proceso es creado y se inserta en la pila.

```
Seleccione una opcion: 2
--------------------------------------------------------------------------

Introduzca el PID del proceso padre: 1
Introduzca el tiempo de inicio del proceso: 5
Introduzca el tiempo de vida del proceso: 7
Introduzca la prioridad del proceso: 3
Proceso introducido correctamente.
```

3. **Mostrar todos los procesos en la pila**

Esta opción permite ver todos los procesos cargados en la pila, mostrando sus atributos clave. Muestra datos como el PID, PPID, tiempo de inicio, duración, y prioridad de cada proceso, utilizando ***pila.mostrarTodos().*** Ofrecer al usuario una visión clara de los procesos que están disponibles y listos para entrar en ejecución, este listado previo puede ayudar a anticipar la ejecución y verificar el orden y detalles de los procesos antes de iniciar la simulación.

```
Seleccione una opcion: 2
--------------------------------------------------------------------------

PID: 1, PPID: 1, Minutos de inicio: 0, Tiempo de vida: 4 minutos, Prioridad: 0
PID: 2, PPID: 1, Minutos de inicio: 0, Tiempo de vida: 5 minutos, Prioridad: 1
PID: 3, PPID: 1, Minutos de inicio: 1, Tiempo de vida: 4 minutos, Prioridad: 3
PID: 4, PPID: 2, Minutos de inicio: 4, Tiempo de vida: 3 minutos, Prioridad: 7
PID: 5, PPID: 2, Minutos de inicio: 6, Tiempo de vida: 2 minutos, Prioridad: 2
PID: 6, PPID: 3, Minutos de inicio: 9, Tiempo de vida: 5 minutos, Prioridad: 1
PID: 7, PPID: 3, Minutos de inicio: 10, Tiempo de vida: 3 minutos, Prioridad: 4
PID: 8, PPID: 4, Minutos de inicio: 12, Tiempo de vida: 2 minutos, Prioridad: 6
PID: 9, PPID: 7, Minutos de inicio: 13, Tiempo de vida: 1 minutos, Prioridad: 8
PID: 10, PPID: 5, Minutos de inicio: 15, Tiempo de vida: 4 minutos, Prioridad: 2
PID: 11, PPID: 4, Minutos de inicio: 16, Tiempo de vida: 3 minutos, Prioridad: 3
PID: 14, PPID: 6, Minutos de inicio: 18, Tiempo de vida: 2 minutos, Prioridad: 4
PID: 12, PPID: 5, Minutos de inicio: 18, Tiempo de vida: 2 minutos, Prioridad: 5
PID: 13, PPID: 6, Minutos de inicio: 18, Tiempo de vida: 1 minutos, Prioridad: 1
PID: 16, PPID: 8, Minutos de inicio: 19, Tiempo de vida: 3 minutos, Prioridad: 6
PID: 15, PPID: 7, Minutos de inicio: 19, Tiempo de vida: 5 minutos, Prioridad: 0
PID: 17, PPID: 10, Minutos de inicio: 20, Tiempo de vida: 2 minutos, Prioridad: 7
PID: 19, PPID: 9, Minutos de inicio: 21, Tiempo de vida: 4 minutos, Prioridad: 2
PID: 18, PPID: 8, Minutos de inicio: 21, Tiempo de vida: 1 minutos, Prioridad: 8
```

En caso no se haya creado previamente los procesos y almacenados en la pila con la opción 1. Se mostrará en pantalla que la pila se encuentra vacía.

```
Seleccione una opcion: 2
--------------------------------------------------------------------------------

La pila est├í vac├¡a.
```

4. **Borrar pila de procesos**

Esta opción elimina todos los procesos de la pila, dejando el sistema sin procesos en ella. Esta operación reinicia la pila y se asegura de que el usuario pueda empezar desde cero si desea recargar una nueva pila con los procesos con la opción 1 pero con diferentes PID ya que estos son identificadores únicos para cada proceso. Para ello, se verifica si la pila está vacía usando ***pila.esVacia()***. Si contiene procesos, se elimina cada uno usando ***pila.desapilar()*** hasta que la pila esté vacía.

Se muestra un mensaje indicando que la pila ha sido vaciada correctamente, o si en su defecto si ya se encontraba vacía.

```
Seleccione una opcion: 3
--------------------------------------------------------------------------------

Pila de procesos eliminada correctamente.
```

En caso de que la pila esté vacía y se intente ejecutar la opción 3 de vaciar la pila. Este mostrará el siguiente mensaje de que la pila ya se encuentra vacía.

```
Seleccione una opcion: 3
--------------------------------------------------------------------------------

La pila de procesos ya est├í vac├¡a.
```

5. **Aumentar tiempo del sistema (n minutos)**

Aumenta el tiempo del sistema en una cantidad especificada por el usuario y simula la ejecución de procesos. Durante esta simulación, se realiza una serie de pasos para gestionar los procesos:

Inserción de procesos en núcleos y sus colas si su tiempo de inicio coincide con el tiempo inicial 0.

Se verifica si los procesos en ejecución han completado su tiempo de vida, en cuyo caso

son eliminados del núcleo y se eliminan los núcleos que están inactivos.

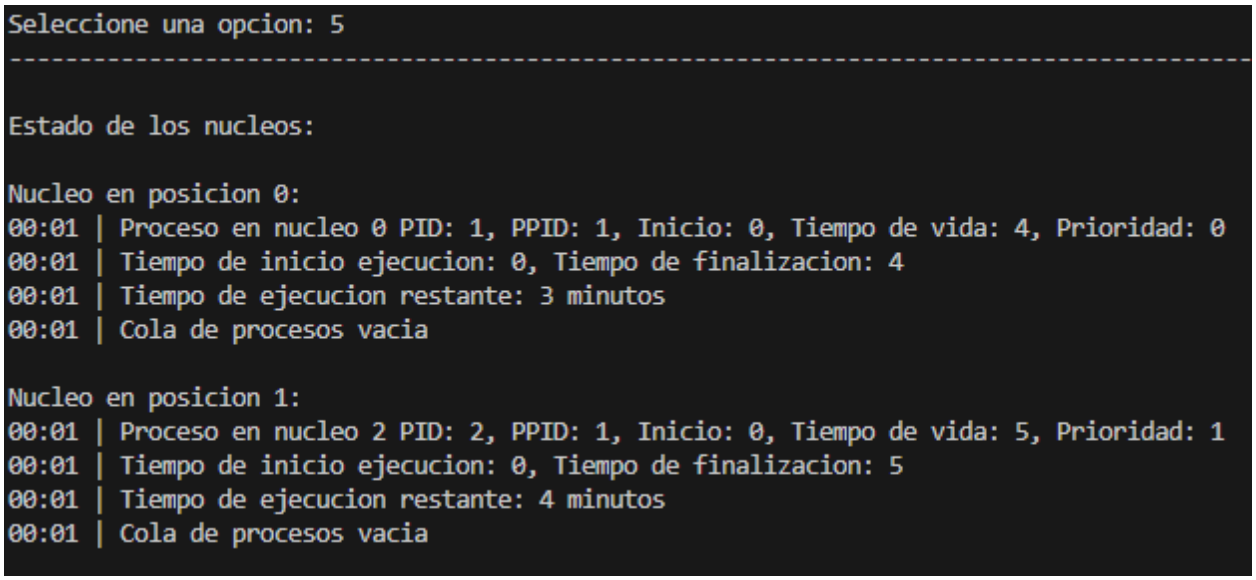
Se actualizan los tiempos de inicio y fin para calcular la estancia en el sistema.

Se insertan en los núcleos o sus colas de espera los procesos que inicien su ejecución en el tiempo actual.

```
Seleccione una opcion: 4
-----------------------------------------------------------------------------------------------------

Ingrese el numero de minutos a aumentar: 1


-----------------------------------------------------------------------------------------------------
Tiempo actual: 00:00

Proceso en la cima de la pila: PID: 1, PPID: 1, Inicio: 0, Tiempo de vida: 4, Prioridad: 0

00:00 | Proceso iniciado en nucleo 0: PID: 1, PPID: 1, Inicio: 0, Tiempo de vida: 4, Prioridad: 0

00:00 | Proceso iniciado en nucleo 2: PID: 2, PPID: 1, Inicio: 0, Tiempo de vida: 5, Prioridad: 1

Proceso en la cima de la pila: PID: 3, PPID: 1, Inicio: 1, Tiempo de vida: 4, Prioridad: 3


Nucleo en posicion 0:
00:00 | Proceso en nucleo 0 PID: 1, PPID: 1, Inicio: 0, Tiempo de vida: 4, Prioridad: 0
00:00 | Tiempo de inicio ejecucion: 0, Tiempo de finalizacion: 4
00:00 | Tiempo de ejecucion restante: 4 minutos
00:00 | Cola de procesos vacia

Nucleo en posicion 1:
00:00 | Proceso en nucleo 2 PID: 2, PPID: 1, Inicio: 0, Tiempo de vida: 5, Prioridad: 1
00:00 | Tiempo de inicio ejecucion: 0, Tiempo de finalizacion: 5
00:00 | Tiempo de ejecucion restante: 5 minutos
00:00 | Cola de procesos vacia
```

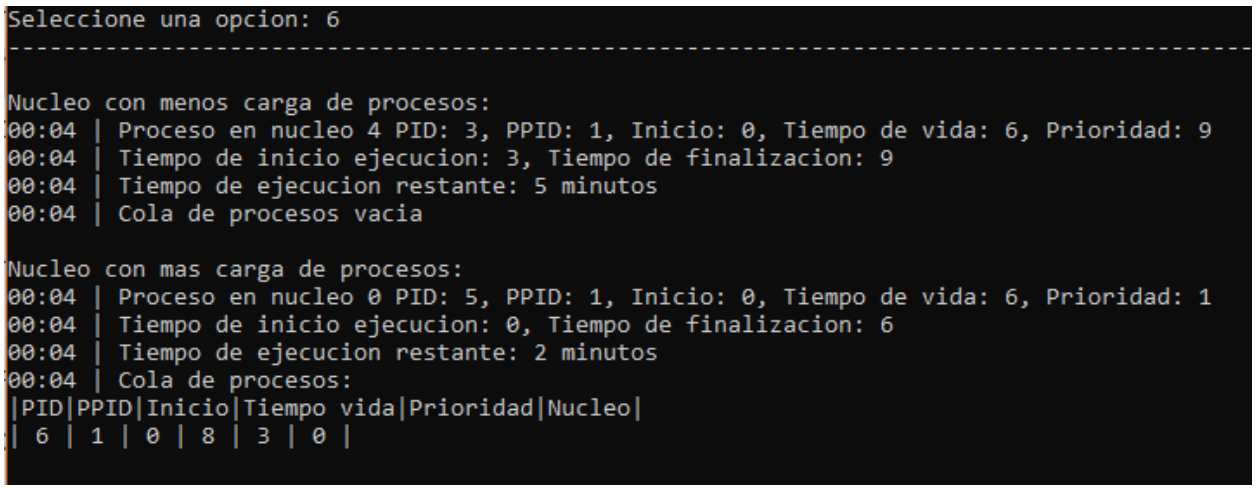
6. **Mostrar estado de los núcleos**

Muestra el estado actual de todos los núcleos en la lista. Se detalla cada núcleo, incluyendo el proceso en ejecución (si hay alguno), la cola de espera, el tiempo de inicio y fin del proceso en ejecución, y el tiempo de espera restante para completar su tarea.



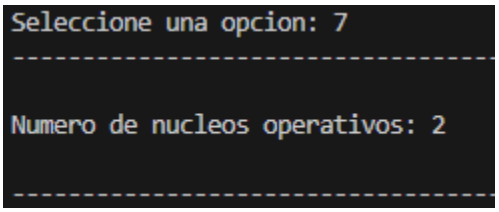
7. **Consulta núcleo con menos carga y núcleo con más carga**

Identifica y muestra los núcleos que tienen la menor y mayor carga de procesos en sus colas de espera. Esta consulta es útil para balancear la carga entre los núcleos, permitiendo la asignación dinámica de procesos a los núcleos menos saturados.



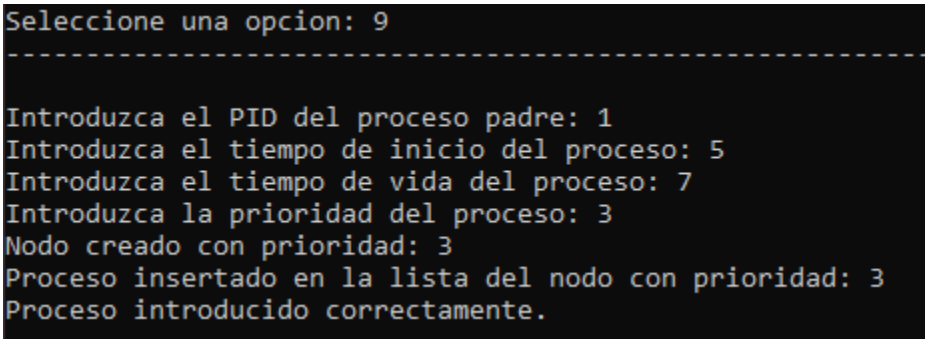
8. **Consulta número de núcleos operativos**

Muestra el número total de núcleos actualmente activos en la lista de núcleos. Esto permite al usuario conocer cuántos núcleos están siendo utilizados en el sistema para ejecutar los procesos.



9. **Insertar un proceso directamente al ABBProcesos**

Esta opción permite insertar un proceso directamente en el Árbol Binario de Búsqueda (BST) de procesos. El usuario ingresa los datos del proceso, como el PID del proceso padre, el tiempo de inicio, el tiempo de vida y la prioridad del proceso. Una vez que se introducen estos datos, el proceso es creado y se inserta en el ABB según su prioridad. Si ya existe un nodo con la misma prioridad, el proceso se agrega a la lista de procesos en ese nodo, si no, se crea un nuevo nodo para esa prioridad.



10. **Mostrar los datos almacenados en el ABBProcesos**

Esta opción muestra todos los procesos almacenados en el BST, recorriendo el árbol en inorden. Para cada nodo del árbol, se imprime la prioridad y la lista de procesos asociados a esa prioridad. Esto permite al usuario visualizar los procesos almacenados en el sistema y verificar su distribución en el árbol según su prioridad. Si no hay procesos en el árbol, se indica que el BST está vacío.

```
Seleccione una opcion: 10
--------------------------------------------------------------------------------

Nodo con prioridad: 0
Procesos en la lista:
PID: 1, PPID: 1, Inicio: 0, Tiempo de vida: 4, Prioridad: 0, Tiempo de ejecucion: 4

Nodo con prioridad: 1
Procesos en la lista:
PID: 20, PPID: 0, Inicio: 0, Tiempo de vida: 2, Prioridad: 1, Tiempo de ejecucion: 2
PID: 24, PPID: 1, Inicio: 0, Tiempo de vida: 2, Prioridad: 1, Tiempo de ejecucion: 2
PID: 2, PPID: 1, Inicio: 0, Tiempo de vida: 5, Prioridad: 1, Tiempo de ejecucion: 9
PID: 6, PPID: 3, Inicio: 9, Tiempo de vida: 5, Prioridad: 1, Tiempo de ejecucion: 7

Nodo con prioridad: 2
Procesos en la lista:
PID: 21, PPID: 1, Inicio: 0, Tiempo de vida: 2, Prioridad: 2, Tiempo de ejecucion: 4
PID: 5, PPID: 2, Inicio: 6, Tiempo de vida: 2, Prioridad: 2, Tiempo de ejecucion: 5
PID: 10, PPID: 5, Inicio: 15, Tiempo de vida: 4, Prioridad: 2, Tiempo de ejecucion: 5

Nodo con prioridad: 3
Procesos en la lista:
PID: 25, PPID: 1, Inicio: 0, Tiempo de vida: 2, Prioridad: 3, Tiempo de ejecucion: 4
PID: 3, PPID: 1, Inicio: 1, Tiempo de vida: 4, Prioridad: 3, Tiempo de ejecucion: 7
```

11. Mostrar los procesos con la prioridad n

Esta opción permite al usuario consultar los procesos que tienen una prioridad específica en el BST. El usuario ingresa el número de prioridad, y el programa busca en el árbol los nodos que contienen procesos con esa prioridad. Si se encuentran procesos con la prioridad solicitada, se muestran los detalles de cada uno.

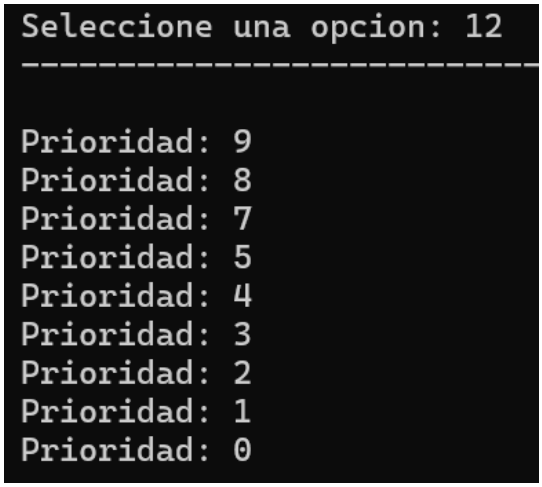
```
Seleccione una opcion: 11
--------------------------------------------------------------------------------

Introduzca la prioridad de los procesos a mostrar: 4
Nodo con prioridad 4 encontrado
Procesos en la lista:
PID: 7, PPID: 3, Inicio: 10, Tiempo de vida: 3, Prioridad: 4, Tiempo de ejecucion: 4
```

Si no hay procesos con la prioridad indicada, se muestra un mensaje informando que no existen procesos con esa prioridad.

```
Seleccione una opcion: 11
------------------------------------------------------------

Introduzca la prioridad de los procesos a mostrar: 11
No se ha encontrado un nodo con prioridad 11
```

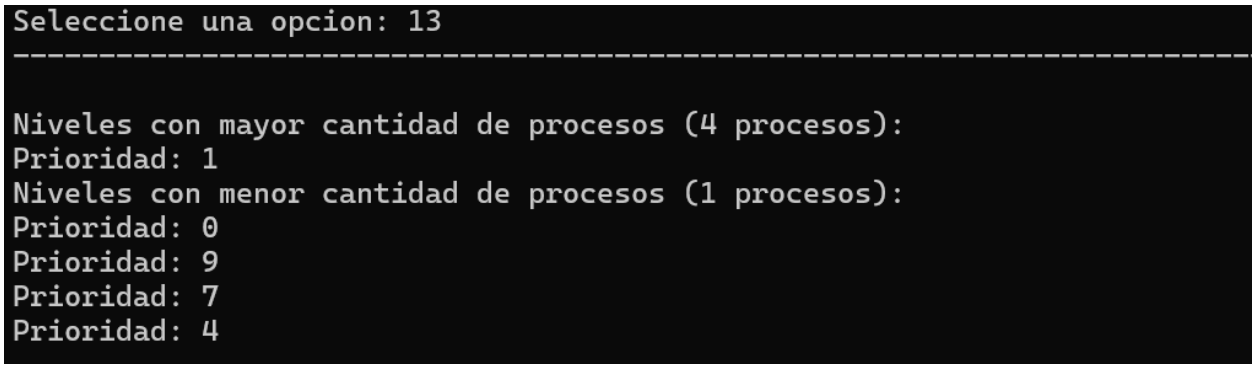
12. Niveles de prioridad registrados

Esta opción muestra los niveles de prioridad que tienen al menos un proceso registrado en el BST. El sistema recorre el árbol y muestra las prioridades de los nodos que contienen procesos. Esta opción permite visualizar las prioridades de los procesos que han sido cargados en el sistema y si se encuentran distribuidos en el árbol.



13. Nivel de prioridad con mayor y menor número de procesos

Esta opción permite al usuario consultar cuál es el nivel de prioridad con el mayor número de procesos y cuál tiene el menor número. El sistema recorre el árbol de procesos y evalúa cuántos procesos hay en cada nivel de prioridad. Luego, muestra las prioridades con la mayor y menor cantidad de procesos. Si existen varios niveles que tienen la misma cantidad, todos ellos serán mostrados como parte del resultado.



14. Tiempo promedio de ejecución de procesos con prioridad n

Esta opción calcula el tiempo promedio de ejecución de los procesos terminados con una

prioridad específica. El usuario ingresa la prioridad deseada y el sistema calcula el promedio de tiempo de ejecución de los procesos asociados a esa prioridad en el BST.

```
Seleccione una opcion: 14
--------------------------------------------------------------------------------

Introduzca la prioridad de los procesos a mostrar: 5
Tiempo promedio de ejecucion de procesos con prioridad 5: 6.5 minutos
```

15. Tiempo promedio de ejecución de procesos en cada nivel de prioridad

Esta opción calcula y muestra el tiempo promedio de ejecución de los procesos en cada nivel de prioridad en el BST. El sistema recorre el árbol, calcula el tiempo promedio para cada nodo que contiene procesos y muestra el resultado por cada prioridad. Esto permite al usuario obtener una visión general de los tiempos promedio de ejecución distribuidos entre las diferentes prioridades almacenadas en el árbol.

```
Seleccione una opcion: 15
--------------------------------------------------------------------------------

Tiempo promedio de ejecucion de procesos en cada nivel de prioridad:

Prioridad: 1 - Tiempo promedio: 5 minutos
Prioridad: 0 - Tiempo promedio: 4 minutos
Prioridad: 2 - Tiempo promedio: 4.66667 minutos
Prioridad: 3 - Tiempo promedio: 5.5 minutos
Prioridad: 9 - Tiempo promedio: 6 minutos
Prioridad: 7 - Tiempo promedio: 5 minutos
Prioridad: 5 - Tiempo promedio: 6.5 minutos
Prioridad: 4 - Tiempo promedio: 4 minutos
Prioridad: 8 - Tiempo promedio: 1.5 minutos
```

16. Simular ejecución de procesos

Simula el flujo completo de tiempo en el sistema operativo, ejecutando todos los procesos hasta su finalización. En cada minuto, se realizan las siguientes acciones:

Inserción de procesos en núcleos y sus colas si su tiempo de inicio coincide con el tiempo inicial 0.

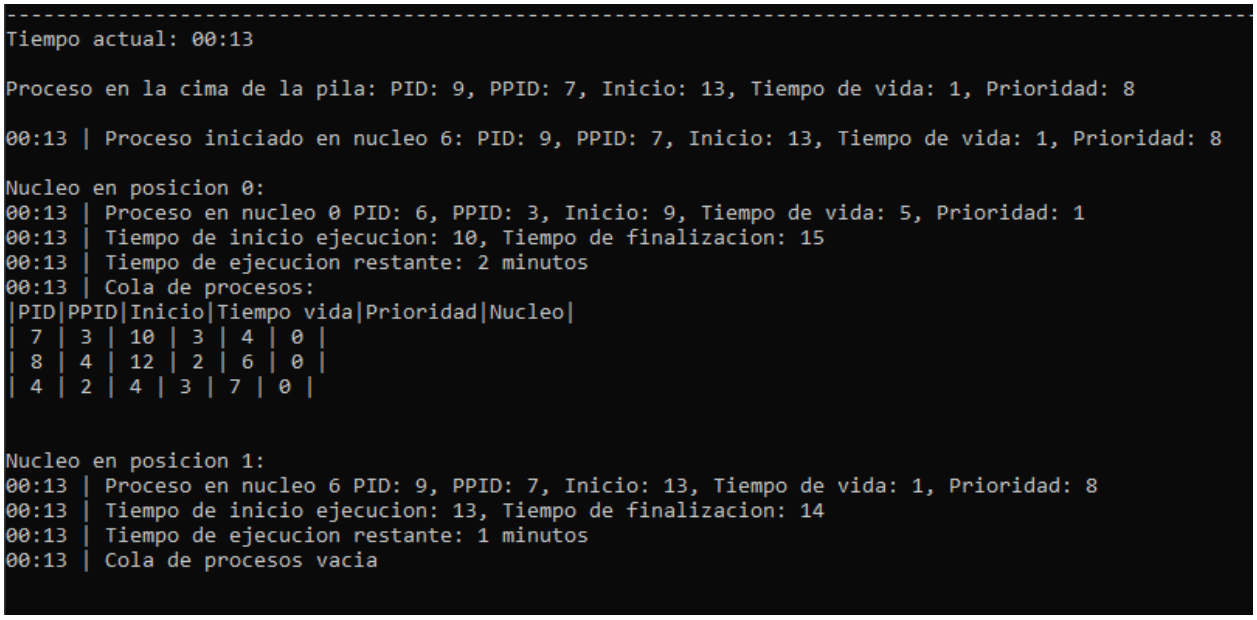
Eliminación de procesos completados y liberación de núcleos que están inactivos.

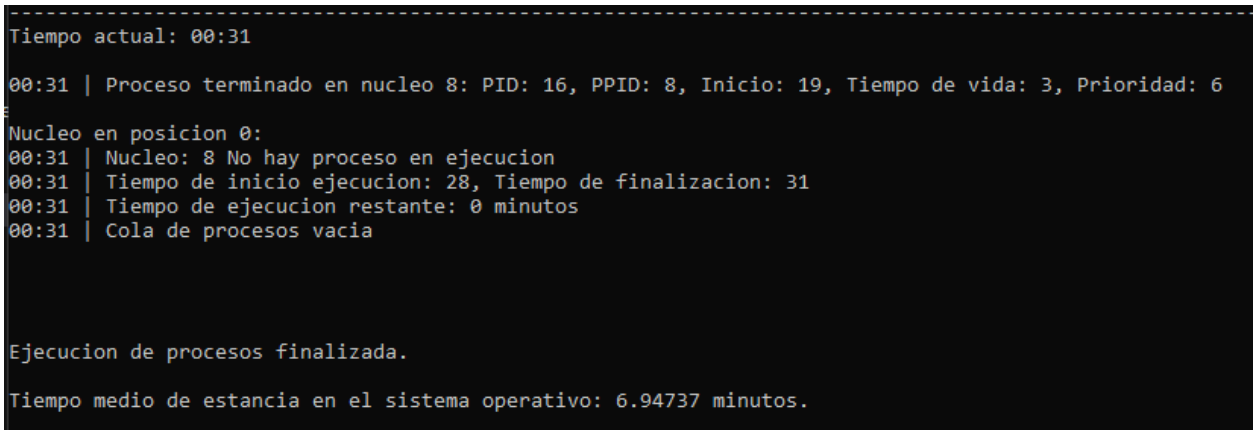
Se actualizan los tiempos de inicio y fin para calcular la estancia en el sistema.

Se insertan en los núcleos o sus colas de espera los procesos que inicien su ejecución en

el tiempo actual.

La simulación continúa hasta que todos los procesos en la pila, en las colas, y en ejecución han sido finalizados, y al finalizar muestra el tiempo medio de estancia en el sistema.





17. Salir

Finaliza el programa y cierra la ejecución de la simulación de procesos.

BIBLIOGRAFÍA

- Devdocs