Informe del Trabajo Práctico Final

“Programación de Procesos”

Alumno: Matías Wasyluk

Materia: Complejidad Temporal, Estructuras de datos y Algoritmos, Comisión 3, 2do cuatrimestre 2024.

Profesor: Mauro Salina

Índice

1. Diagrama de Clases UML

El diagrama de clases está dispuesto en el anexo 1: Diagrama UML

2. Descripción general

En este informe se detalla la implementación de los métodos solicitados en la consigna del trabajo práctico final de la materia Complejidad Temporal, Estructuras de datos y Algoritmos, Comisión 3, 2do cuatrimestre 2024.

El trabajo consiste en desarrollar un simulador de planificación de CPU para un sistema operativo, tomando como base datos extraídos de archivos CSV que contienen información de procesos (nombre, tiempo de uso de la CPU y prioridad). El sistema permite realizar simulaciones de planificación: *Shortest Job First* (SJF), que prioriza los procesos con menor tiempo de uso de CPU, y *Preemptive Priority CPU Scheduling Algorithm* (PPCSA), donde los procesos con mayor prioridad toman la CPU primero. El usuario selecciona el archivo de datos, define los parámetros de la simulación (cantidad y velocidad de CPUs) y ejecuta la planificación elegida.

La implementación abarca la clase Estrategia, con métodos que ordenan los procesos utilizando MinHeap y MaxHeap según el tipo de planificación seleccionado. Además, se desarrollaron consultas para analizar las propiedades de las estructuras utilizadas, como hojas, alturas y niveles de las Heaps. El informe detalla los aspectos técnicos del desarrollo, como las clases y algoritmos implementados, los problemas encontrados, las soluciones aplicadas, y propone posibles mejoras para futuras versiones del simulador.

3. Análisis del programa

a. PP y SJF

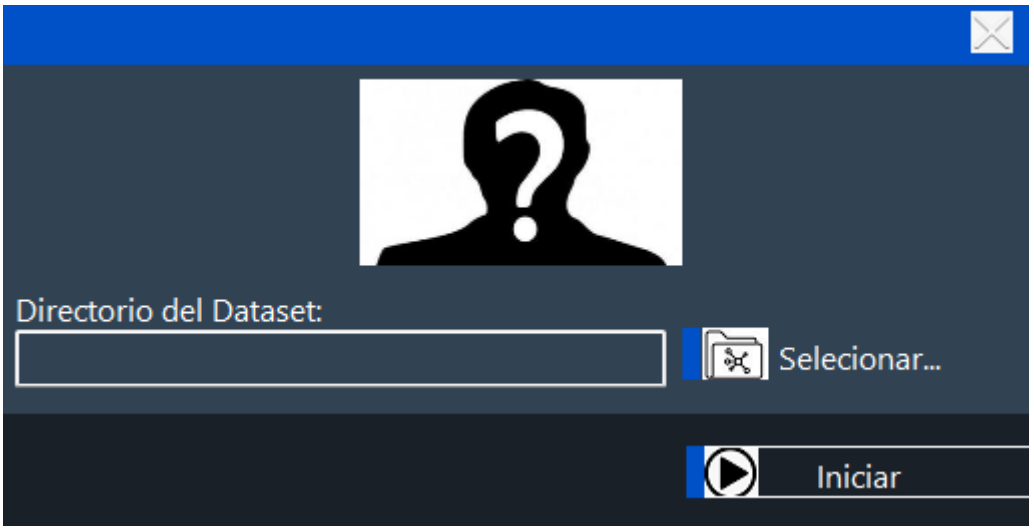

Imagen 1: Selección de dataset

Como primer menú, al iniciar el programa el usuario debe seleccionar el archivo .csv que contiene la lista de procesos con su nombre, tiempo y prioridad, para este caso de estudio utilizaremos el dataset provisto por la cátedra.

Una vez seleccionado el dataset, el usuario puede seleccionar que a través de un switch A/B qué tipo de planificación de procesos usar.

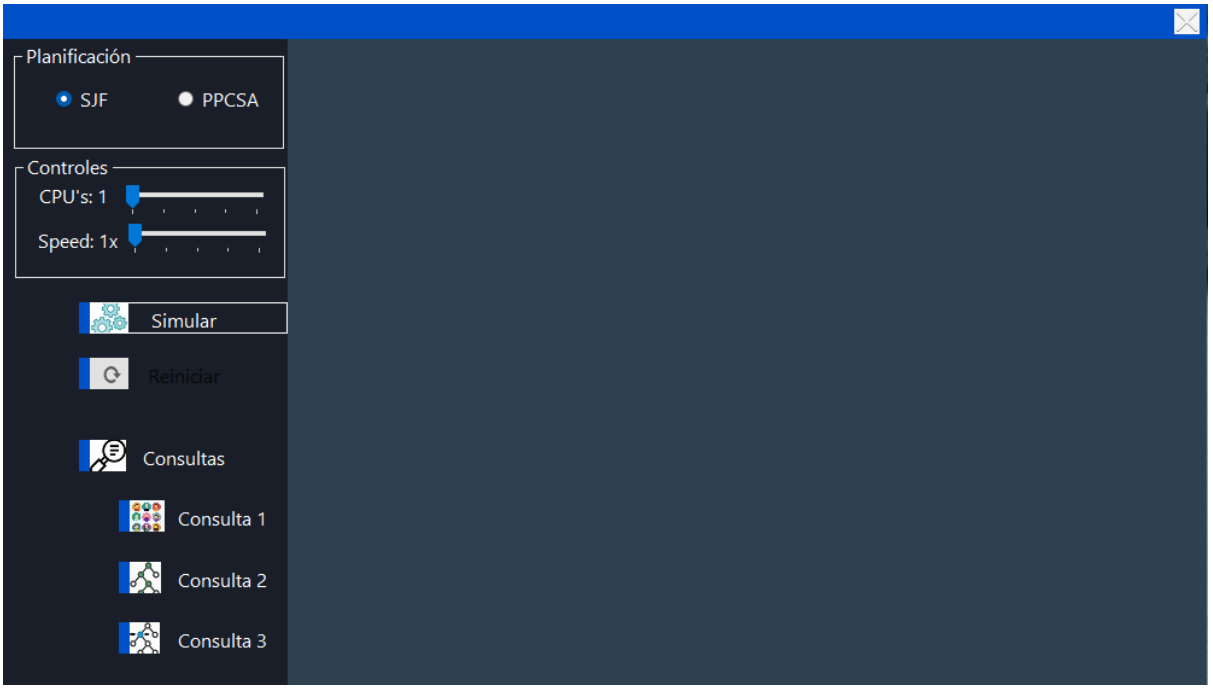

Imagen 2: Pantalla principal del programa

La planificación SJF llama al método ShortestJobFirst(datos, collected) en la clase Estrategia. Este método recibe por parámetro ***datos*** de tipo *List<Proceso>* y ***collected*** de tipo *List<Proceso>*. El alumno implementa mediante el patrón *FactoryMethod*, la creación de la *IHeapStrategy* concreta *SJFHeapStrategy,* la cual implementa una *MinHeap* para almacenar los procesos.

El alumno luego implementó un bucle ***foreach*** que recorre la variable ***datos*** pasada por parámetros insertando los datos uno a uno en la Heap correspondiente con el método Insertar().

```
public void Insertar(IComparable comparable)

{

	comparable.SetEstrategia(this.estrategia);

	elementos.Add(comparable);

	FiltrarArriba(this.tamaño- 1);

}
```

```
private void FiltrarArriba(int index){
	while (index > 0){
		int padre = (index - 1) / 2;
		if (elementos[index].EsMayor(elementos[padre]))
			break;

		Intercambiar(index, padre);
		index = padre;
	}
}
private void Intercambiar(int i, int j){
	IComparable temp = elementos[i];
	elementos[i] = elementos[j];
	elementos[j] = temp;
}
```

Se modificó la clase Proceso para que implemente la interfaz IComparable. Esto ayuda a mantener la abstracción en las clases Heap.

```
public interface IComparable{
	void SetEstrategia(IEstrategiaComparacion estrategia);
	bool EsMayor(IComparable comparable);
	bool EsMenor(IComparable comparable);
	bool EsIgual(IComparable comparable);
}
```

```
public class Proceso : IComparable
	{
		...
		public IEstrategiaComparacion estrategia;
		public void SetEstrategia(IEstrategiaComparacion estrategia){
			this.estrategia = estrategia;
		}
		public Proceso(string nombre, int tiempo, int prioridad){
			...
			this.estrategia = null;
		}
		public bool EsMayor(IComparable unProceso){
			return this.estrategia.EsMayor(this , unProceso);
		}
		public bool EsMenor(IComparable unProceso){
			return this.estrategia.EsMenor(this , unProceso);
		}
		public bool EsIgual(IComparable unProceso){
			return this.estrategia.EsIgual(this , unProceso);
		}
		...
	}
}
```

Luego mediante un bucle ***while***, mientras la Heap no esté vacía, llama al método GetRaiz() de la misma, la cual retorna el proceso de la raíz y lo inserta en la variable ***collected*** pasada por parámetro con el método Add().

```
public IComparable GetRaiz(){
	if (esVacia)
		throw new InvalidOperationException("La heap está vacía.");
	IComparable min = elementos[0];
	elementos[0] = elementos[tamaño - 1];
	elementos.RemoveAt(tamaño - 1);
	FiltrarAbajo(0);
	return min;
}
private void FiltrarAbajo(int index){
	int tamañoHeap = this.tamaño;
	while (index < tamañoHeap){
		int hijoIzq = 2 * index + 1;
		int hijoDer = 2 * index + 2;
		int menor = index;
		if (hijoIzq < tamañoHeap &&
elementos[hijoIzq].EsMenor(elementos[menor]))
			menor = hijoIzq;
		if (hijoDer < tamañoHeap &&
elementos[hijoDer].EsMenor(elementos[menor]))
			menor = hijoDer;
		if (menor == index)
			break;
		Intercambiar(index, menor);
		index = menor;
	}
}
```

La planificación PP llama al método PreemptivePriority(datos, collected) en la clase Estrategia. Este método recibe por parámetro ***datos*** de tipo *List<Proceso>* y ***collected*** de tipo *List<Proceso>*. El alumno implementa mediante el patrón *FactoryMethod*, la creación de la *IHeapStrategy* concreta *PPHeapStrategy*.

La heap se construye y los procesos se agregan a la variable ***collected*** de la misma manera que en el método ShortestJobFirst().

En la imagen 2, la sección de “Controles”, permite al usuario subir la cantidad de CPUs y la velocidad de simulación. Por debajo, se encuentra el botón de “Simular” y “Reiniciar” simulación y son autodescriptivos.

La última sección es la de consultas, en donde el usuario puede consultar datos acerca de la Heap generada con los métodos anteriormente descritos.

3. b. Consulta 1:

La consulta 1 llama al método Consulta1(datos: List<Proceso>) la cual retorna un string con la cantidad de hojas que tiene cada Heap creada por la estrategia PP y SJF.

El alumno implementó dentro de la clase MinHeap y MaxHeap, un método CantidadHojas() el cual retorna un int con la cantidad de hojas de la Heap.

Sabiendo que en una Heap, desde la mitad en adelante los nodos serán hoja, se aprovecha esta característica para calcular la cantidad de hojas en T(n) = O(1).

```
public int CantidadHojas(){

	if (esVacia)

		return 0;

// La cantidad de hojas es el total de elementos menos el número de nodos que tienen al menos un hijo

	int nodosConHijos = this.tamaño / 2;

	return this.tamaño - nodosConHijos;

}
```

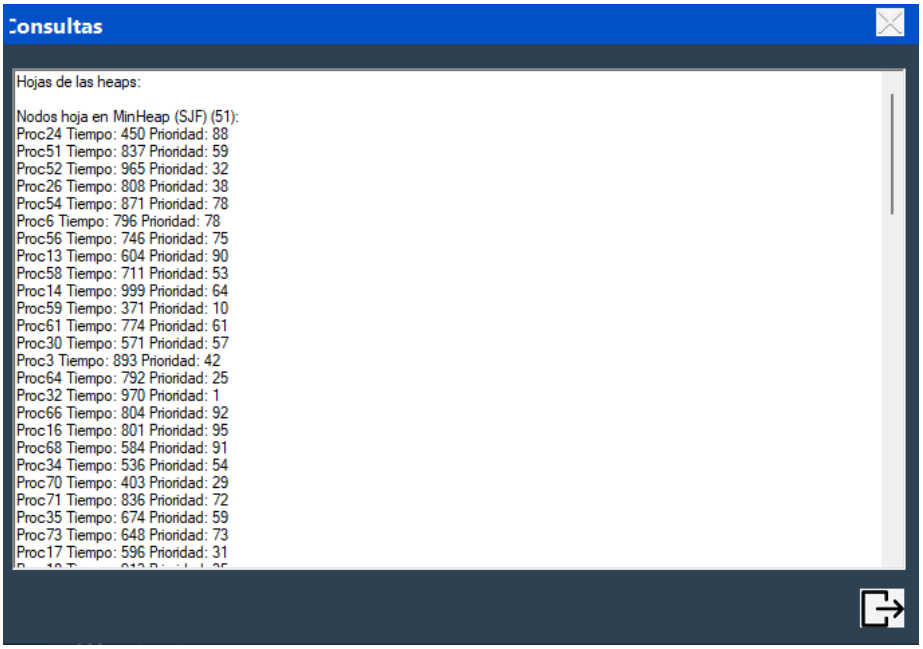

Imagen 3: Menú Consulta 1

Para imprimir todos los nodos hoja se llama al método ImprimirHojas() de la clase MinHeap y MaxHeap el cual recorre los nodos Lista interna de la clase y los concatena y retorna en un ***string*** sin exponer su implementación interna. Si tanto el hijo izquierdo como el hijo derecho se encuentran en un índice mayor al tamaño de la lista se concatena el nodo padre al string ***hojas***.

```
public String ImprimirHojas(){

	String hojas = "";

	for(int i = 0; i < this.tamaño-1; i++){

		int hijoIzq = 2 * i + 1;

		int hijoDer = 2 * i + 2;

		if(hijoIzq >= this.tamaño && hijoDer >= this.tamaño){

			hojas += "\n" + this.elementos[i] + " ";

		}

	}

	return hojas;

}
```

3. c. Consulta 2

La consulta 2 consiste en visualizar la altura de las Heaps creadas por SJF y PP. El alumno implementó el método AlturaHeap() en la clase MinHeap y MaxHeap el cual retorna la altura de la heap correspondiente.

El cálculo de la altura de la heap se determina a partir de la siguiente fórmula:

$$log_2(n)$$ n = cantidad de elementos de la lista de nodos

La implementación del método AlturaHeap() es de complejidad t(n) = O(1).

```
public int AlturaHeap(){

      if (this.elementos.Count == 0)

            return 0; // Si el heap está vacío, la altura es 0.

      return (int)Math.Ceiling(Math.Log(this.elementos.Count,2));

}
```

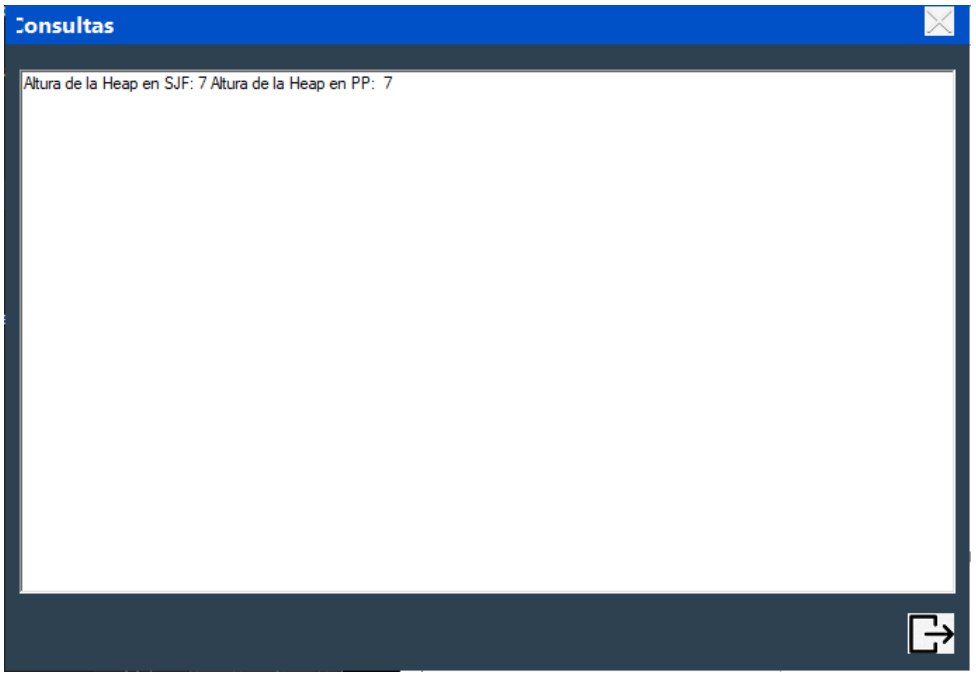

Imagen 4: Menú Consulta 2

3. d. Consulta 3

La consulta 3 llama al método Consulta3() de la clase estrategia y consiste en retornar un string que contiene cada uno de los nodos de cada Heap y el nivel de la Heap en la que se encuentran.

Para cada una de las Heaps se ejecuta un bucle ***for*** i veces hasta recorrer el tamaño de la Heap correspondiente. Por cada iteración se calcula el nivel del nodo con la siguiente fórmula:

$$log_2(n + 1)$$ n = número de iteración

Se concatenan el nivel y los datos del nodo obteniéndose con el método GetRaiz().

```
for (int i = 0; i < tamañoMinHeap; i++){

      int nivel = (int)Math.Floor(Math.Log(i + 1,2)); // Calcular nivel

      result += "\nNivel " + nivel.ToString() + ": " +
SJF.GetRaiz().ToString();

}

for (int i = 0; i < tamañoMaxHeap; i++){

      int nivel = (int)Math.Floor(Math.Log(i + 1,2)); // Calcular nivel

      result += "\nNivel " + nivel.ToString() + ": " +
PP.GetRaiz().ToString();

}
```

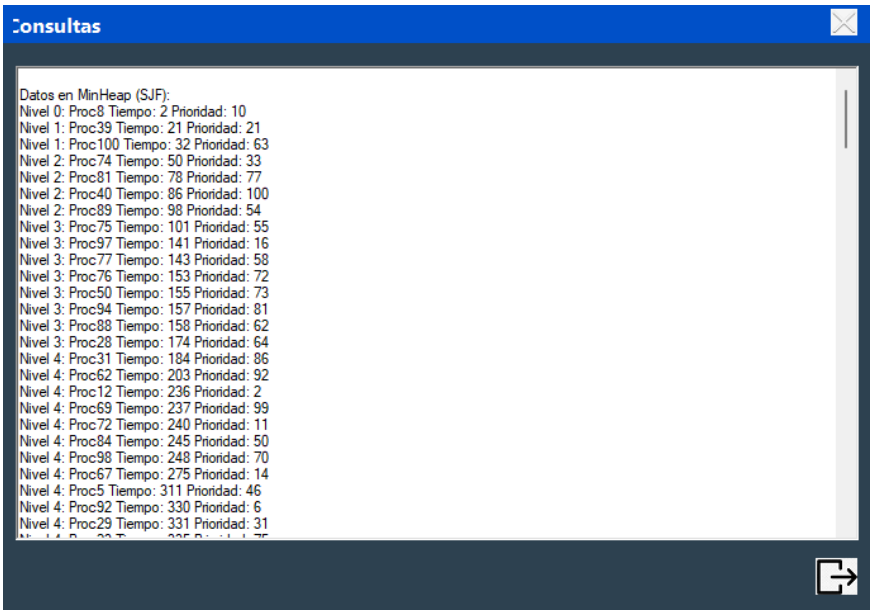

Imagen 5: Menú Consulta 3

4. Estructuras de Datos y Algoritmos

Para la construcción del programa se utilizaron los patrones:

4. a. Factory Method:

Se utilizó en la clase HeapFactory para encapsular la creación de las instancias de las diferentes IHeapStrategy. Esto permite decidir en tiempo de ejecución que tipo de heap se crea según el parámetro que se proporcione.

El beneficio del factory method es la reusabilidad de la fábrica con nuevas IHeapStrategy y la reducción del acoplamiento entre la clase estrategia y las clases concretas de las heaps.

4. b. Strategy Method

Implementado mediante la interfaz IHeapStrategy y sus implementaciones (PPHeapStrategy y SJFHeapStrategy). Este patrón permite cambiar dinámicamente el comportamiento de la planificación dependiendo de la estrategia seleccionada.

Su beneficio principal es la posibilidad de extender el sistema para incorporar nuevas estrategias sin modificar el código existente.

4. c. Heap (MaxHeap y MinHeap):

La estructura principal del proyecto. Se implementaron las versiones de MaxHeap (para PPCSA) y MinHeap (para SJF), basadas en una lista dinámica (List<IComparable>).

Estas heaps soportan las operaciones esenciales como inserción, eliminación del nodo raíz y cálculo de propiedades como hojas y altura.

Garantizan complejidad O(log n) en las operaciones de inserción y eliminación, lo que las hace ideales para el scheduling.

5. Ideas y Mejoras

Como mejora el alumno propone implementar el algoritmo Round Robin, el cual usa un quantum de tiempo fijo para cada proceso y va alternando hasta terminar la ejecución. La rotación de procesos se simula usando una cola circular.

Además de implementar el código se puede actualizar la interfaz del usuario para poder seleccionar el quantum con un slider o un número determinado.

La complejidad temporal de este algoritmo dependerá del número de procesos y el número de rotaciones necesarias para que el algoritmo termine:

$$T(n) = O(n \cdot r)$$

n = cantidad procesos, r = cantidad rotaciones

```
public void RoundRobin(List<Proceso> datos, List<Proceso> collected, int quantum){

    Queue<Proceso> cola = new Queue<Proceso>(datos);

    Dictionary<Proceso, int> tiemposRestantes = new Dictionary<Proceso, int>();

// Inicializamos los tiempos restantes de los procesos

    foreach (var proceso in datos){

        tiemposRestantes[proceso] = proceso.Tiempo; // Usamos la propiedad "Tiempo"

    }

    while (cola.Count > 0){

        Proceso proceso = cola.Dequeue();

        // Procesamos el quantum

        int tiempoRestante = tiemposRestantes[proceso];

        if (tiempoRestante > quantum){

            // Resta el quantum y vuelve a encolar

            tiemposRestantes[proceso] -= quantum;

            collected.Add(proceso); // Se agrega al historial de ejecución

            cola.Enqueue(proceso);

        }

        else{

            // El proceso termina

            tiemposRestantes[proceso] = 0;

            collected.Add(proceso); // Se registra su finalización

        }
```

```
    }
}
```

6. Conclusión y Reflexión

En conclusión, el desarrollo del simulador de planificación de CPU demuestra cómo la combinación de patrones de diseño vistos en la materia de Metodología de Programación y las estructuras de datos vistas durante la cursada de Complejidad Temporal, Estructuras de Datos y Algoritmos pueden ofrecer un sistema eficiente, modular y extensible.

Una dificultad que halló el alumno en el desarrollo fue que al refactorizar la clase Proceso para que implemente la interfaz IComparable, se complejiza la inserción en la Heap, teniendo que determinar la estrategia de comparación al momento de insertar el Proceso. Debido a esto el alumno optó por insertar uno a uno los Procesos y no usar el algoritmo BuildHeap ya que requería un recorrido ***for*** que dejaría la complejidad de inserción en t(n) = n*log(n).

La implementación de estrategias como Shortest Job First y Preemptive Priority Scheduling mediante el uso de heaps proporciona una base sólida para gestionar los procesos, permitiendo recorrer su estructura interna en t(n)=log(n) sin exponer su implementación interna.

Además, el diseño orientado a interfaces y la incorporación del Factory Method y Strategy Method para manejar la creación de estrategias aseguran la flexibilidad del sistema, permitiendo futuras expansiones. Esto se ve reflejado en la posibilidad de implementar nuevos algoritmos de scheduling como Round Robin, propuesto por el alumno.

El desarrollo del programa cumple con los requerimientos actuales y permite sentar bases para un mantenimiento más sencillo y la adaptación a nuevos escenarios en el futuro.

Anexos y apéndices

Anexo 1: Diagrama de clases UML

Anexo 2: Repositorio del proyecto