TP_Final Complejidad Temporal

2024

Planificador de Procesos

C#(POO + Forms)

Integrantes:

Nombre	*Apellido*	*Contacto*
Elvis	Huarachi	ehmwill12@gmail,com
Leonel	Paco	LPACO249@GMAIL.COM

Introduccion

En este informe se desarrollara la implementacion de la “Estrategia de comparacion” para un simulador de planificación de CPU, esto se nos encomienda en el Trabajo final de la materia. El trabajo tiene el objetivo de reforzar nuestros conocimientos con la primera parte de la materia: Heaps (Montículos).

Al principio del “Anexo I” podemos encontrar el enunciado del Simulador, así como también cada imagen a la que haremos referencia en este informe.

Comenzaremos describiendo cada pantalla del sistema/Simulador codificado, para luego profundizar en el funcionamiento (métodos) y relaciones de la clase “Estrategia”, así como también la dificultad o problemas que se tuvo para implementar sus métodos y como se podría mejorarlos.

Por ultimo una breve conclusión del trabajo realizado.

Ejecucion del Simulador

Al ejecutar el programa, el usuario se encontrará al principio con la tarea de cargar el “Archivo.csv” , Archivo a simular (fig. 1). En segundo lugar, finalizada la carga, un menú principal se desplegara con las siguientes opciones: “Planificación”, “Controles”, “Simular”, “Reiniciar”, “Consultas” (fig. 1.1).

En **Consultas**, encontraremos información de los procesos que próximamente se Simularan al iniciar el programa. Este apartado nos permitirá ver como están los procesos posicionados en la estructura de datos, la relevancia que tendrán y una idea del flujo con el cual se procesaran (fig. 2; fig. 2.1; fig. 2.2; fig 2.3). En segundo lugar, la **Planificación** nos permitirá elegir el algoritmo con el cual comparamos los procesos SJF (ShortesJobFirst) y PP CSA(PreemptivePriority)(fig. 1.1). En tercer lugar, los **Controles** nos permiten aumentar la cantidad de CPU´s que procesaran la información y a su vez la velocidad de los mismos (fig. 3). En cuarto lugar, el **Simular** nos permitirá darle el inicio al programa (fig. 3).

Por último, **Reiniciar** nos permite finalizar una Simulación y volver a seleccionar un algoritmo de comparación .

Comenzada la ejecución del programa, se cargara la parte grafica de la clase "**Inicio.cs**" en donde el usuario va poder seleccionar la ruta del "archivo.csv", para posterior mente ser procesada al tipo de formato esperado por el simulador. Cargado el archivo.csv al simulador, se deberá elegir una Planificación, es decir, un algoritmo de Comparación(fig. 1; fig.1.1). Se cuenta con 2 algoritmos, por un lado, ShortesJobFirst(), este algoritmo ejecuta el proceso con el menor tiempo de ejecución. Por otro lado, tenemos al algoritmo PreemptivePriority(), este le sede la CPU al proceso con mayor prioridad.

Iniciada la simulación, se verá como los procesos van siendo ejecutados en función del algoritmo, de la cantidad CPU´s y el speed. Tanto la cantidad de CPU´s como la velocidad se podrán modificar en un rango de 5 unidades, esto se podrá ver gráficamente en la cantidad de CPU´s trabajando simultáneamente a una velocidad determinada en la pantalla del usuario. Además de ver que cada CPU nos muestra el nombre del proceso, su prioridad y su tiempo de ejecución. (fig. 3).

En paralelo a la simulación, se podrá ver las consultas y a su vez reiniciar la simulación en caso de que se requiera cambiar de estrategia(fig. 2; fig. 2.1;fig. 2.2; fig. 2.3). Una vez terminada la simulación el programa retornara el tiempo de ejecución que le llevara al sistema operativo procesar una determinada cantidad de datos siguiendo un determinado algoritmo de prioridad de tareas(fig. 4).

Explicado el funcionamiento del simulador, en la fig.5 podemos ver un diagrama UML que pone en foco las relaciones internas de como se implementa el simulador y como influye la clase **Estrategia** con la lógica de comparación.

La clase **Estrategia** tiene una relación de dependencia con la clase **Proceso**, por otro lado, con las Clases **MaxHeap<T>** y **MinHeap<T>** tiene una relación de composición y dependencia. En el primer caso, Estrategia tiene que sus métodos utilizan como parámetro "Listas de Procesos", de ahí la dependencia. A su vez, las clases **MinHeap** y **MaxHeap** están declaradas como atributos de la clase estrategia, y sus métodos de consulta también instancian objetos de este tipo, de ahí la Composicion y dependencia con los heaps. Por último, **Backend** depende de estrategia, ya que utiliza instancias de estrategia dentro de sus métodos.

La clase **Estrategia** implementa cinco métodos y declara 2 atributos. Los atributos declarados sirven de estructura de datos para los algoritmos de planificación. Por el lado de los métodos, en el apartado de Consultas (Consulta1(), Consulta2() Consulta3()), son consultas que operan sobre estructuras de datos también del tipo heap, instanciados dentro del bloque perteneciente al metodo. Los otros 2 métodos, ShortestJobFirst() y PreemptivePriority(), están diseñados para ordenar y manipular los procesos con criterios específicos, valiéndose de la lógica de los heap para este fin, ya que estas estructuras contienen la lógica pura de comparación. La clase Estrategia encapsula y se le delega la responsabilidad de la lógica de comparación, siendo esta usada por Backend, que a su vez será utlizada por el sistema/simulador.

A continuación, se detallan cada uno de estos métodos y su funcionamiento:

- Método Consulta1():

El método Consulta1 retorna las hojas de ambas heaps, minHeapSJF y maxHeapPriority. Para ello recibe como parámetro una List<Procesos>, que vacia su contenido dentro de Heap auxiliares. Entonces, el método simplemente calcula el índice donde comienza el último nivel de la heap (el nivel de las hojas) y luego recorre los elementos de forma secuencial desde ese índice hasta el final de la lista interna. aprovechando la estructura de la lista(List<T>) sin necesidad de realizar un recorrido específico como en un árbol tradicional (inorden, postorden, etc.). Cumpliendo con la petición.

- Método Consulta2():

Este método es similar al Cta.1 al principio. Pero, este calcula la altura de cada heap. La fórmula que usa se basa en la naturaleza de una heap completo, donde la altura h de un árbol binario completo con n nodos se puede calcular como $h = (\log_2(n + 1))$.

- Método Consulta3():

En este método, se retorna los datos que almacenan los heaps junto con su nivel en el árbol.

Se crean copias de los heaps para no vaciar las heaps originales al insertar los procesos. Luego, para cada proceso en los heaps se calcula su nivel. El nivel de un nodo en un heap se calcula usando la fórmula “Nivel= log2(i+1)”, donde i es el índice del nodo en la heaps. Esto retorna una lista de procesos ordenados por niveles en ambos heaps (SJF y Prioridad).

- **ShortesJobFirst(List datos, List collected)**

  Se insertan todos los procesos en la MinHeap. Esta heap ordena los procesos en función del tiempo de CPU (proceso.tiempo). La propiedad de la heap asegura que el proceso con el menor tiempo de CPU estará en la raíz. Se extraen los procesos con ExtractMin(), que siempre devolverá el proceso con el menor valor de tiempo de CPU en la raíz de la MinHeap. Los procesos extraídos se agregan a la lista collected.

- **PreemptivePriority(List datos, List collected**)

  Se insertan todos los procesos en la **MaxHeap**. Esta heap ordena los procesos en función de la prioridad (proceso.prioridad). En una MaxHeap, la raíz siempre contiene el proceso con la mayor prioridad (el valor más alto). Con ExtractMax(), se devolverá el proceso con la mayor prioridad en la raíz de la MaxHeap. Al igual que en SJF, los procesos extraídos se agregan a la lista collected.

Problemas y Soluciones

Problema: Al no tener una especificación de como implantar los Heaps, se nos presento el dilema de si implementarlo con una árbol binario(recursivo) o con una lista de hijos(array dinámico).

Solución: Optamos por implementarlo utilizando una lista de hijos, ya que esto facilitó tanto la implementación de los métodos como la comprensión de la lógica del programa, haciéndolo más intuitivo y eficiente.

Problema:
Al principio, tuvimos dificultades para asegurar que los Heaps mantuvieran sus propiedades después de realizar operaciones de inserción y extracción. Esto ocurrió porque la lógica para implementarlo resultaba algo compleja y **requería** un **manejo cuidadoso** de las posiciones y las comparaciones entre nodos.

Solución:
Para dar solución al problema se utilizaron las siguientes formulas,
para la implementación de un Heaps con **lista de hijos**:

$$\text{índice\_inicial\_hojas} = \left\lfloor \frac{n}{2} \right\rfloor$$

$$\text{hijo\_izquierdo} = 2i + 1$$

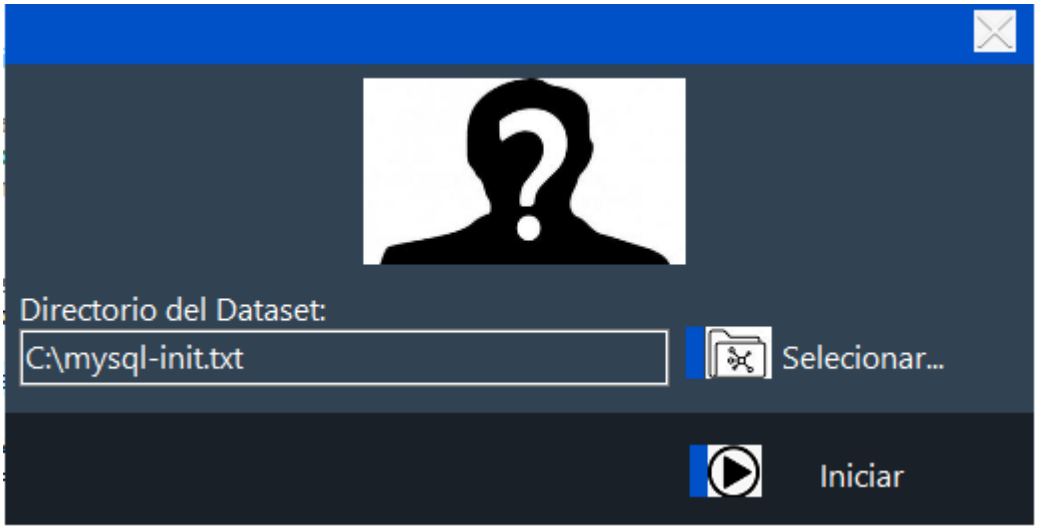
$$\text{hijo\_derecho} = 2i + 2$$

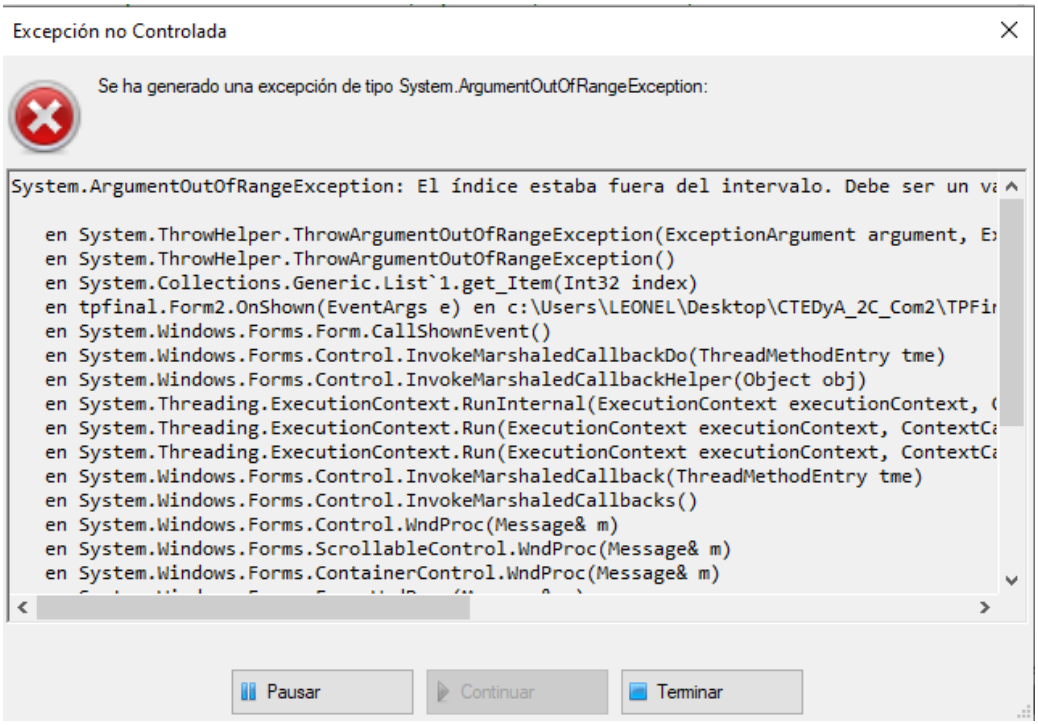
$$\text{padre} = \left\lfloor \frac{i-1}{2} \right\rfloor$$

$$\text{altura} = \lfloor \log_2(n) \rfloor$$

Ideas & Sugerencias

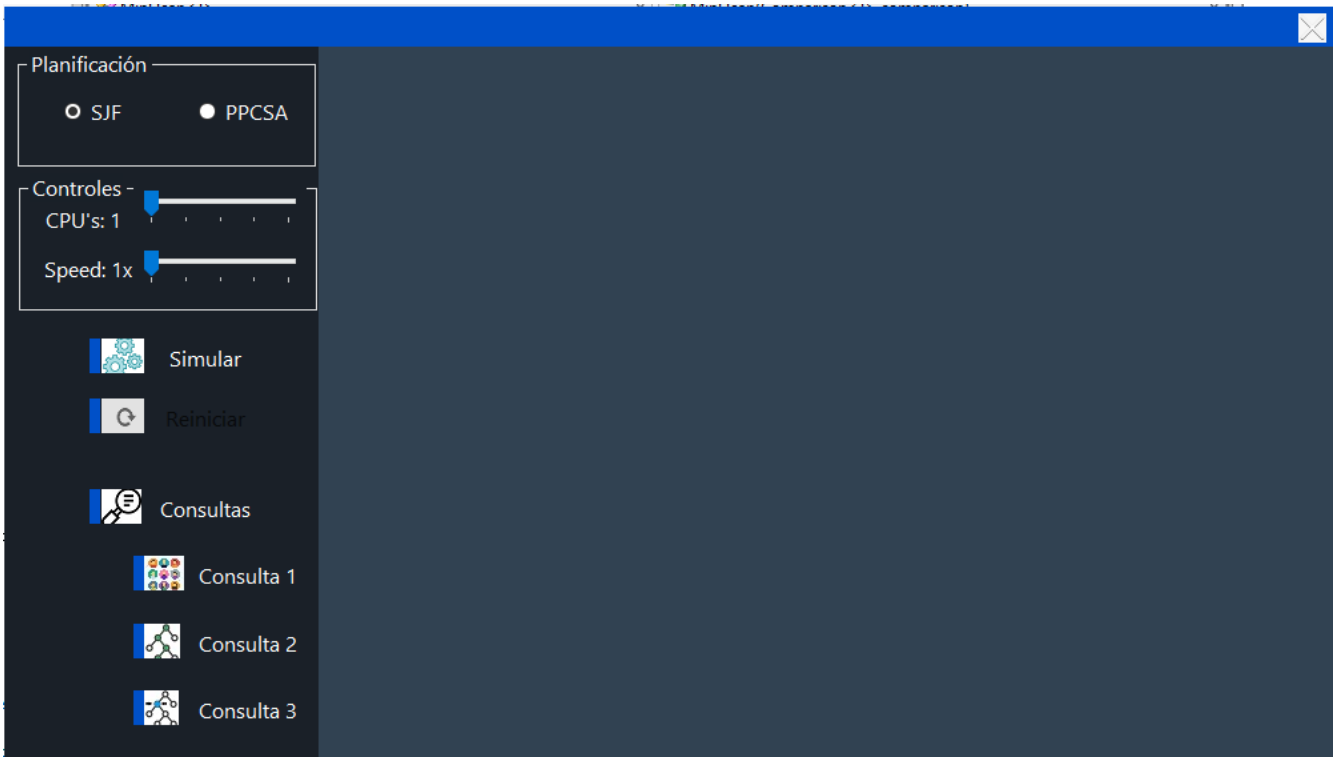
El programa tiene un **Error** al inicio de la simulación ya que si el usuario no selecciona un **archivo.csv** salta la advertencia de “**Excepción no Controlada**” por ende, se corta la ejecución de toda la Aplicación.





Este problema puede ser corregido usando un manejador de **Excepciones** (*Try-Catch*) dentro del código, como sugerencia estaría genial poder agregar un bucle el cual diga que: “Seleccione el archivo nuevamente debido a que el formato es incompatible”.

Otra sugerencia es que se puede Mejorar mucho mas el diseño de la Aplicación de C#, es decir que sea mas **intuitiva para el usuario** mejorando la paleta de colores usadas, las imágenes dentro del proyecto, el formato de letra, tamaño y propiedades usadas para darle estilo a cada "*button*", "*label*", "*form*", "*check button etc****…."***.



No estaría de más poder agregar la **funcionalidad** de desplegar una **ventana aparte** cuando seleccionamos una ***Consulta en concreto*** de esa manera se podría ver simultáneamente como los procesos se van ejecutando en una "***ventana"***, mientras que en la ***"otra_ventana"*** poder ver las hojas de los Heaps o cualquier otra consulta.

Para concluir como sugerencia se podría agregar las funcionalidades de

- "**Minimizar**" permite que la ventana este en la ***barra de tareas***.
- "**Agrandar**" permite ver la ventana en **pantalla completa**.
- "**Responsive**" permite que el contenido dentro de una ventana se **ajuste automáticamente** según el **tamaño de la ventana**.

Conclusión

En conclusión, se superaron varios desafíos, como la manipulación de las Heaps sin modificarlas, la solución fue crear copias de las estructuras. Además, se identificaron áreas de mejora, como el manejo de excepciones y la mejora del diseño de la interfaz, lo que podría optimizar la experiencia del usuario.

En resumen, este proyecto refuerza nuestros conceptos a la hora de elegir una determinada estructura de datos como base de algún algoritmo a implementar, ya que en función de la estructura elegida tenemos determinadas ventajas, que en términos de Big-O son cruciales para determinar la eficiencia temporal de algoritmo.