Trabajo obligatorio

Segunda entrega

Ingeniería en informática

Sistemas operativos

Grupo 5

Integrantes del grupo:
Marcelo Arrarte
Federico Ferreira
Rafael Filardi

Docente teórico: Gerardo Guglielmetti
Auxiliar: Sebastián Torres

Repositorio:
https://github.com/RafaFil/Obligatorio_SistemasOperativos/releases/tag/v1.0.0

Fecha de informe 20/06/2022

Contenido

1. Introducción

El principal objetivo de este entregable fue programar una simulación lo más realista posible de un planificador a corto plazo. Para ello se nos dio una serie de pasos a seguir y requisitos que el mismo debía cumplir. Lo que se hizo fue generar una GUI (Graphical User Interface – Interfaz gráfica de usuario) que representara dicha situación, en la cual mediante la carga de procesos estos fuesen siendo asignados a un CPU para la ejecución. En otras palabras, la simulación es la puesta en práctica de los conceptos teóricos estudiados durante el curso.

2. Marco teórico

Previo a entrar en los conceptos concretos utilizados durante el trabajo, es necesario enumerar una serie de conceptos previos los cuales nos ayudarán a lograr un mejor entendimiento del trabajo en general.

Supongamos el siguiente escenario: llegan varias peticiones al sistema operativo, ¿Cómo se maneja esta situación? El sistema operativo modela estas peticiones como procesos. (Tanenbaum, 2009) En un modelo de procesos, todo software ejecutable en una computadora, (incluyendo al mismo sistema operativo) se organiza utilizando varios procesos secuenciales. Estos procesos no son más que instancias de distintos programas en ejecución. Cada uno de estos procesos tiene asociado un espacio de direcciones (el cual contiene el programa ejecutable, los datos del programa y su pila), donde el proceso puede escribir y leer información. A estos procesos también hay asociados conjuntos de recursos, usualmente conteniendo registros, listas de archivos abiertos, alarmas pendientes y listas de procesos relacionados.

Los procesos requieren de una CPU para que sus instrucciones de programa puedan ser ejecutadas, y salvo casos específicos, toda CPU es capaz de ejecutar un proceso determinado. Pero, incluso si el procesador posee varios núcleos físicos y/o procesadores con tecnología multihilo (mediante la cual se simulan dos núcleos lógicos a partir de uno solo físico), es imposible que estos puedan atender todos los procesos (ya sean de sistema operativo o de usuario) que existen en el sistema al mismo tiempo. Esto se soluciona con pseudoparalelismo, creando la ilusión de que todos los programas abiertos se ejecutan simultáneamente. Esto se consigue mediante un planificador que se encarga de manejar el tiempo en CPU de todos los procesos, dándoles y quitándoles la misma de la forma que el algoritmo del planificador determine. A partir de este sistema los procesos se pueden dividir en tres posibles estados: “en ejecución”, “bloqueado” y “listo”. De esta forma, dándole cortos periodos de tiempo en el CPU a todos los procesos se consigue la sensación de continuidad que dan las computadoras.

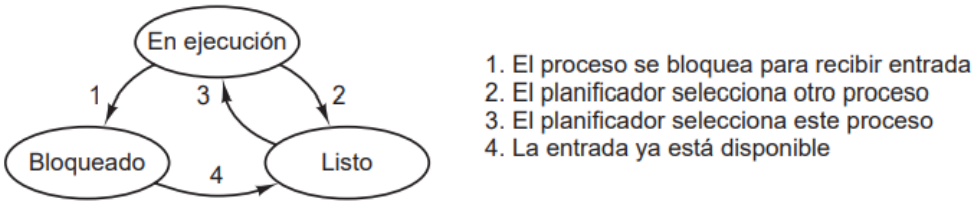


Figura 2-2. Un proceso puede encontrarse en estado “en ejecución”, “bloqueado” o “listo”. Las transiciones entre estos estados son como se muestran.

Ilustración 1 – Posibles estados de un proceso

(Cabalar, s.f.) Existen tres categorías principales de procesos: los procesos *batch*, los procesos interactivos y los procesos a tiempo real. Los procesos *batch* son procesos que se envían al sistema y esperan terminar en un tiempo prudencial,

utilizando algoritmos de planificación simples para no recargar el sistema. Suelen ejecutarse en ciclos libres del procesador y son mayormente de baja importancia ya que no hay ser humano interactuando, y los tiempos de retorno oscilan entre minutos y horas, dependiendo del uso que se le dé al sistema mientras tiene el lote de procesos pendiente por ejecutar.

Los procesos interactivos, son procesos que requieren la interacción de un ser humano. Se debe proporcionar una frecuente alternancia de procesos para simular paralelismo, que es lo habitualmente demandado por un usuario. Por lo general, los tiempos de respuesta se miden en segundos, ya que dependen del tiempo de respuesta del humano interactuando con ellos y priorizan una respuesta a una velocidad adecuada.

Respecto a los procesos a tiempo real, se trata de procesos de alta importancia y que requieren atención inmediata, tomando prioridad sobre procesos interactivos y *batch* la mayoría del tiempo. Suelen ser funcionas críticas del sistema operativo o procesos de usuarios especiales, pueden estar asociados a dispositivos especiales que requieren atención de interrupciones en forma inmediata.

Planificadores:

(Tanenbaum, 2009) Un planificador es una parte del sistema operativo que hace uso de un algoritmo de planificación para asignar la CPU a distintos procesos que están listos para ser ejecutados. Dependiendo del tipo de planificador, puede variar la forma en que se administra la CPU, encargándose de asignarle la CPU a otro proceso cuando el anterior finaliza en algunos casos, o en otros cediéndole la CPU a otro proceso luego de determinado tiempo.

Con el objetivo de almacenar información acerca de los procesos gestionados, muchos planificadores, si no todos, también requieren guardar cierta información de estos en lo que se conocen como bloques de control de proceso (BCP). Se trata de estructuras de datos que registran el estado de un proceso, recursos asignados, identificadores, permisos, etc. Esto les permite a los planificadores determinar el siguiente proceso a ejecutar, y las acciones en general cuando sucede algo con un proceso.

Clasificación según plazo de planificación

(Guglielmetti, 2022) Dentro de los planificadores, hay tres configuraciones usadas comúnmente, estas siendo a largo plazo, a mediano plazo y a corto plazo. Los planificadores a largo plazo deciden que procesos serán los siguientes en ejecutar y no se detienen hasta que dicho proceso libere la CPU voluntariamente o se bloquee. Es un tipo de planificador frecuente en sistemas de lotes, la planificación en este modelo puede ocurrir cada varios segundos, minutos o inclusive horas y, hoy en día, casi ningún sistema de uso interactivo lo utiliza.

La planificación a mediano plazo presenta como principal diferencia a la planificación a largo plazo que este tipo de planificador puede decidir si es conveniente bloquear algún proceso en la cola de procesos. Esto puede suceder por escasez o saturación de algún recurso, o en caso de recibir alguna solicitud que momentáneamente no puede ser satisfecha. A estos planificadores también se les conoce como “agendadores” o “*schedulers*”.

La tercera categoría de planificadores, los de corto plazo, evalúa constantemente cómo distribuir los recursos entre los procesos. Esta planificación

se lleva a cabo decenas de veces por segundo y es comúnmente usada en sistemas interactivos. También se conoce a este tipo de planificadores como "despachador" o "*dispatcher*".

Clasificación según asignación de CPU

(UDELAR, 2014) Existen varios algoritmos de planificadores que surgieron por modernización o por nuevas necesidades. Estos algoritmos se pueden clasificar en dos distintivos tipos, los algoritmos apropiativos y los algoritmos no apropiativos.

Los algoritmos de carácter no apropiativo son aquellos algoritmos que, una vez que el CPU ha sido dado a determinado proceso, este no se le quitará forzosamente, sino que esperará hasta que el proceso se bloquee (ya sea para esperar una operación de E/S o de otro proceso) o hasta que dicho proceso libere la CPU de forma voluntaria.

Como ejemplos de algoritmos no apropiativos tenemos al algoritmo First Come First Serve (FCFS), en el cual el orden de ejecución de los procesos es el mismo orden en el cual llegan a la cola de procesos listos. Como ventaja de este algoritmo, cabe mencionar su muy fácil implementación, simplemente utilizando una cola FIFO. Esto lo hace inadecuado para sistemas interactivos que no pueden tolerar tiempos de respuesta excesivos, pero es apropiado para sistemas por lotes, escenario en el cual no se presenta esta restricción.

Otro ejemplo de algoritmo no apropiativo (aunque se puede generar una implementación apropiativa) es el algoritmo Shortest Job First (SJF). Este algoritmo evalúa el orden de ejecución de los procesos tomando en cuenta su *CPU-burst*, cuando el CPU queda libre se le asignará al proceso con el *CPU-burst* más corto.

Este algoritmo tiene solamente una utilidad teórica, ya que necesita saber el tiempo de utilización de la CPU antes de que esta sea asignada.

Por otro lado, existen algoritmos apropiativos, con una filosofía distinta a la de los algoritmos ya mencionados. Este segundo grupo de algoritmos tiene la capacidad de quitarle la CPU a un proceso aunque este no haya finalizado o se haya bloqueado para asignársela a otro proceso, facilitando la ejecución de varios procesos de forma pseudoparalela. Dependiendo del algoritmo, puede variar el criterio en base al cual se otorga y quita CPU a los distintos procesos.

Un algoritmo de carácter apropiativo puede ser Shortest Remaining Time First (SRTF), es una versión apropiativa del algoritmo SJF mencionado antes, cada vez que entran nuevos procesos ocurre una interrupción, el tiempo de los procesos ya existentes y el nuevo se comparan, el proceso con menos tiempo de ejecución restante tomara el CPU.

Y como ejemplo final, el algoritmo Round-Robin (RR), es de carácter apropiativo y se puede traducir como asignación circular o por torneo. Cada proceso tiene un tiempo límite de uso de CPU denominado quantum; una vez superado este tiempo el CPU será quitado del proceso y asignado a uno distinto. Es importante elegir cuidadosamente el quantum a utilizar, ya que en caso de ser muy pequeño, se requerirá frecuentemente el cambio de proceso asignado, y si es muy grande, el algoritmo tenderá a comportarse como un FCFS. Los procesos preparados para ser ejecutados se organizan en una cola FIFO, de forma que luego de ejecutar un proceso y agotar su quantum, este pasa al final de la cola, manteniendo el orden de ejecución en cada ronda.

3. Proceso de diseño de la simulación:

1.- PLANTEO DEL PROBLEMA:

Según (Guglielmetti, 2022) la letra del problema es la siguiente:

"Se nos ha contratado para realizar un planificador de corto plazo para un sistema operativo servidor que se instalará en un servidor de mediano porte. Pero antes de comenzar el desarrollo, se nos pide que generemos una simulación del mismo como para poderla evaluar su comportamiento. Esto es con el objetivo de validar si el diseño es correcto para este servidor."

(Guglielmetti, 2022, p4)

Esta consigna iba acompañada de los siguientes requerimientos:

"Para que la evaluación del planificador sea lo más realista posible, se nos pide que el mismo contemple (entre otras cosas):

- ✓ *Poder ingresar la cantidad de procesadores o cores.*
- ✓ *Poder modificar la cantidad de tiempo que los procesos se encuentran en CPU.*
- ✓ *Poder modificar la prioridad de los mismos en tiempo de ejecución (prioridad de 1 a 99).*
- ✓ *Poder bloquear un proceso en cualquier momento.*
- ✓ *Poder cargar (de alguna forma) múltiples procesos de un solo ingreso*
- ✓ *Poder insertar procesos ya sea del S.O. como de usuario indicando:*
  - ❖ *Tiempo total de ejecución*
  - ❖ *Cada qué tiempo realiza una E/S (periódica sin modificación)*
  - ❖ *Tiempo en que espera por la E/S (puede ser diferente para cada proceso).*

En todo momento se deberá visualizar lo siguiente:

- ✓ *Proceso ejecutando en CPU (o en CPUs)*
- ✓ *Lista de los procesos listos indicando el orden en que ingresaran a CPU.*
- ✓ *Lista de los procesos bloqueados (indicando si se encuentra bloqueado por el usuario o por una entrada salida) ordenada en cada momento por quien sería el próximo a ser desbloqueado."*

(Giuglielmetti, 2022, p4)

2.- INVESTIGACION

Lo primero que hicimos al recibir el problema fue investigar y pensar un poco como representar, así como entender que era lo que se debía hacer. Lo primero que se realizó fue una búsqueda acerca de lo que es un planificador, así como de los diferentes tipos de algoritmos. Todo el trabajo de investigación se vio reflejado en el marco teórico ya expuesto.

3.- DIFERENTES POSIBLES SOLUCIONES

Tras la investigación y conociendo mejor el alcance del problema, se comenzó a hacer bosquejos de diferentes posibles soluciones que resolvieran el problema planteado, para eso lo que se realizó fueron diferentes bosquejos a lápiz, como por ejemplo el siguiente.

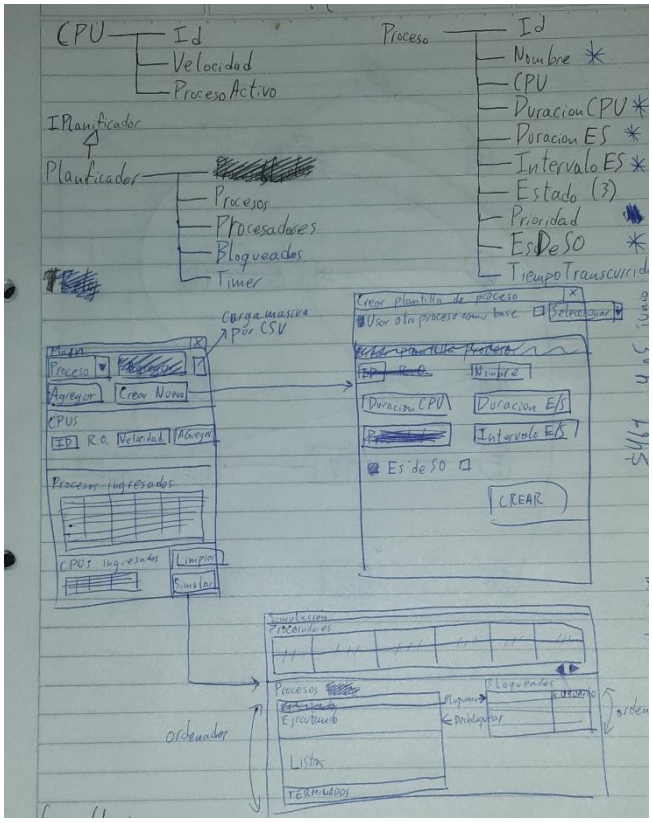


Ilustración 2 - Bosquejo a papel de la solución

En la imagen de arriba se puede observar el último diseño en papel antes de comenzar con los desafíos que implicaba la codificación del programa. En él se puede ver las tres ventanas que se consideraron indispensables en el programa de acuerdo con los requerimientos dados, maquetando una interfaz gráfica y una serie de objetos que van a interactuar dentro del mismo programa (Proceso, CPU, etc.).

También se decidió el tipo de algoritmo a utilizar, y en este caso fue una variante del algoritmo Round Robin que incluyera un sistema de envejecimiento dinámico, por el cual los procesos de menor prioridad fuesen incrementándola, pasando eventualmente a ser ejecutados. Si bien no es el más difícil de implementar, tampoco es de los más sencillos, por lo que se consideró desafiante y acorde a lo requerido en el trabajo.

4.- PLANEACIÓN DE LA SOLUCIÓN

El primer desafío fue ver la forma de conectar todos los requerimientos y conseguir expresarlos claramente para así poder empezar a elaborar la solución. En todo momento, se intentó seguir la idea principal concebida inicialmente, manteniendo abierta la puerta a modificaciones que posteriormente fueran necesarias. La creación de la aplicación se realizó en varios pasos, que se desarrollan a continuación.

1. GUI

Por familiaridad del equipo con el lenguaje C# del .NET Framework de Microsoft, y tras verificar que en dicho framework se encontraban disponibles estructuras de datos útiles para la naturaleza del problema a resolver, se escogieron C# 10 como lenguaje de desarrollo y Windows Forms como framework de interfaz gráfica. Con Windows Forms, la cantidad de código escrito para diseñar las interfaces es mínima, ya que en gran parte del mismo es generado automáticamente por el IDE en archivos separados, permitiendo un mayor enfoque en la lógica del problema. Gracias a ello, fue posible diseñar los siguientes bosquejos detallados, con la certeza de que diseñarlo sería sencillo gracias a las herramientas escogidas.

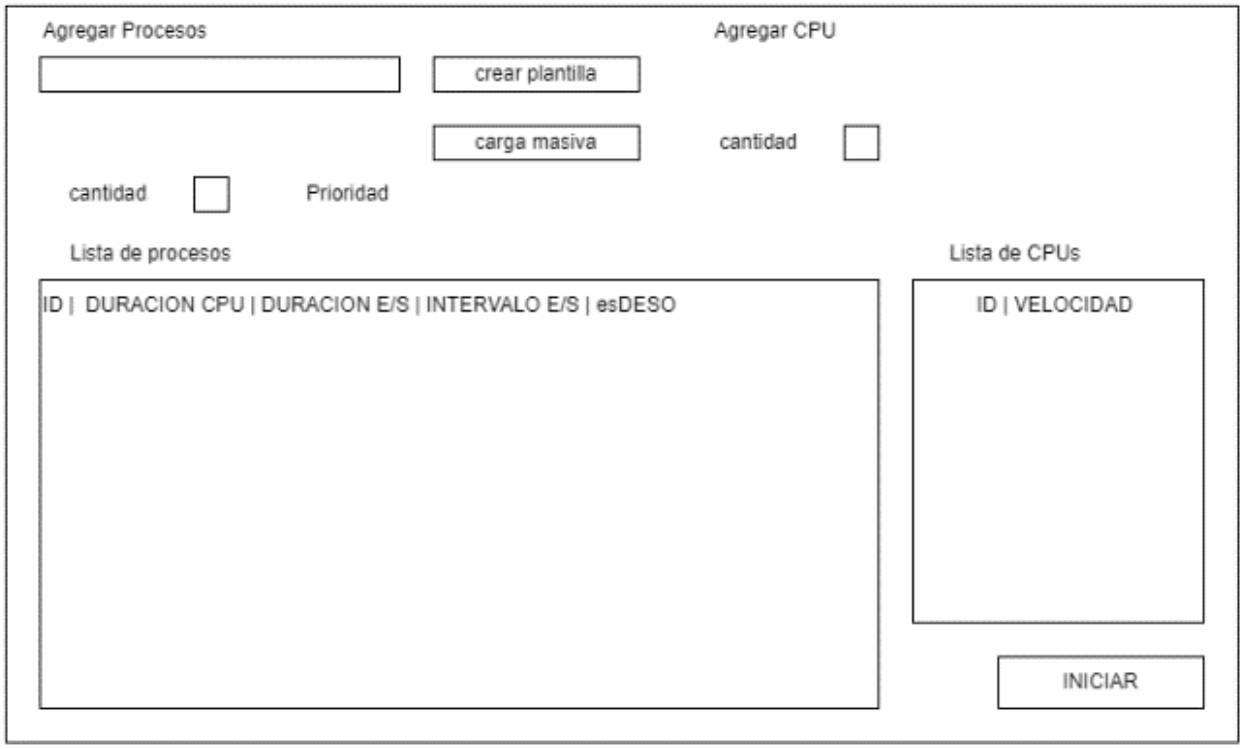


Ilustración 3 - Diseño principal

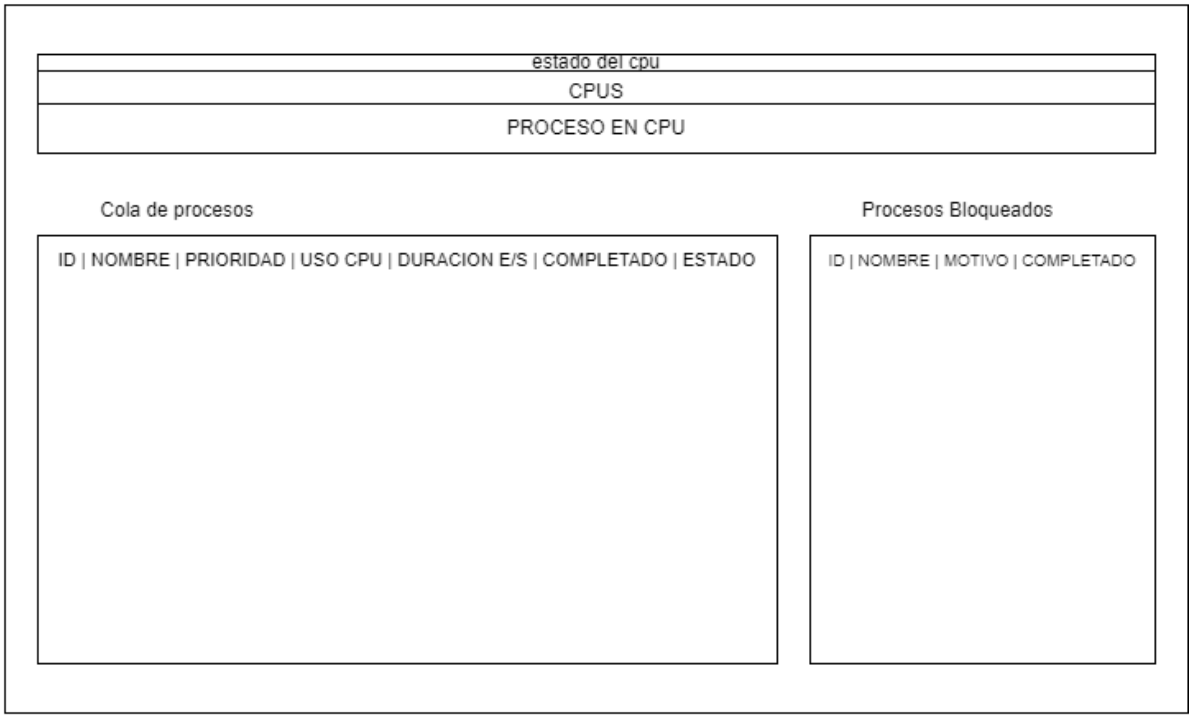


Ilustración 4 - Diseño de la simulación

2. Representación del planificador:

La parte más importante del trabajo es definir la forma de representar el planificador a corto plazo, tanto a nivel de algoritmia como de datos que resulta necesario almacenar. Lo primero y más evidente es que este algoritmo va a aplicar el TDA cola de prioridad, esto se debe a que un planificador Round Robin con envejecimiento dinámico como el planteado para la solución, se beneficia de una estructura de datos que le permita determinar fácilmente el siguiente proceso a ejecutar, utilizando una función comparadora determinada. Buscando una función que tome en cuenta prioridad, envejecimiento y dueño del proceso (SO o usuario), se llegó a la siguiente expresión lambda, que se utiliza como comparador en la cola de procesos listos para ser ejecutados, comparando a y b, los cuales representan sus respectivos bloques de control, que almacenan información acerca de cada proceso:

```
(a, b) =>
{
    if (a.Proceso.esDeSo == b.Proceso.esDeSo)
    {
        return (a.Prioridad - a.Envejecimiento).CompareTo(b.Prioridad - b.Envejecimiento);
    }
    else
    {
        if (a.Proceso.esDeSo)
        {
            return -1;
        }
        else
        {
            return 1;
        }
    }
}
```

Ilustración 5 - Comparador de prioridad de procesos listos

A su vez como se implementará una variante de Round Robin, se necesita una manera de calcular el quantum. Por ello, y aprovechando que debido a la naturaleza

de la simulación, la duración de los procesos, así como sus operaciones E/S son conocidas de antemano, se llegó a la siguiente función:

```
/// <summary>
/// Asigna un quantum adecuado al BCP correspondiente al proceso recibido.
/// </summary>
/// <param name="p">el proceso cuyo BCP será modificado</param>
2 referencias
private void AsignarQuantum(Proceso p)
{
    TimeSpan quantum;
    // Si no tiene operaciones E/S, o basándome en el intervalo de E/S veo que no realizará
    // ninguna más antes de terminar
    if (p.intervaloES == TimeSpan.Zero
        || p.tiempoCPUTranscurrido - p.tiempoCPUTranscurrido.Mod(p.intervaloES) + p.intervaloES > p.duracionCPU)
    {
        quantum = new [] {
            tiempoMaximoEnCPU,
            (p.duracionCPU - p.tiempoCPUTranscurrido) / p.cpu!.Velocidad }.Min();
    }
    // Si realizará alguna operación E/S antes de terminar
    else
    {
        quantum = new[] { tiempoMaximoEnCPU,
            (p.intervaloES - p.tiempoCPUTranscurrido.Mod(p.intervaloES)) / p.cpu!.Velocidad }.Min();
    }

    bloquesDeControl[p].Quantum = quantum;
}
```

Ilustración 6 - Función AsignarQuantum(Proceso)

La misma evalúa las posibles operaciones E/S del proceso, y si determina que ocurrirá alguna antes de que termine el proceso o de que transcurra el tiempo máximo que el proceso podrá usar la CPU, asigna el tiempo restante hasta la operación como quantum.

Las operaciones relacionadas con la determinación de prioridad de procesos y asignación de CPU se realizan utilizando una clase que representa los BCP de un planificador real. En cada instancia de esta clase, se almacena el proceso asociado, así como el valor de envejecimiento actual, utilizado para ponderar la prioridad de procesos listos, y variables que indican en todo momento el progreso de ejecución y de operaciones E/S. También, dado el algoritmo elegido, se consideró el lugar indicado para almacenar el quantum restante de cada proceso. De esta forma, se logró que toda la información del proceso en sí, junto con la información adicional

para uso del planificador, se almacenara en una misma estructura de datos, facilitando la manipulación de los objetos durante el desarrollo del programa. A continuación, se muestra el diseño final de la clase BCP implementada:

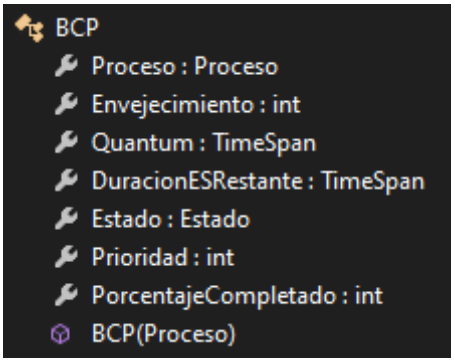


Ilustración 7 - Clase BCP

Otro de los aspectos que presentaron alguna dificultad fue una forma eficiente de comunicar el planificador y la interfaz de usuario. Un aspecto importante a considerar fue la frecuencia con que el planificador actualiza los estados de los procesos, lo cual puede suceder cada pocos segundos. Además, es esperable que la interfaz debiera mostrar el estado de los procesos en tiempo real, lo cual fortalece aún más esta restricción. Por otro lado, no resulta del todo cómodo que sea la interfaz la cual refresque la información mostrada al usuario, debido a que es el planificador el que conoce los tiempos de cada proceso, y cuando ocurre algún evento que deba ser transmitido a la interfaz.

Como solución a esta situación, se decidió implementar el patrón Observer, considerando la interfaz gráfica (concretamente, la clase FrmSimulacion) como el observador, y la clase Planificador como el observable. De esta forma, desligamos a la interfaz de la necesidad de extraer del planificador la información a mostrar al usuario, y el planificador no precisa mostrar la información directamente en la interfaz,

sino que se limita a notificar al observador cuando sucede un cambio en la información de los procesos que implique una actualización en los datos mostrados. Además, hace que la aplicación sea más fácilmente escalable, ya que permite que el planificador únicamente recolecte y envíe su propia información, mientras que cada observador se encarga de interpretarla y desplegarla de la forma más adecuada.

Con la posibilidad de lograr una aplicación lo más desacoplada, mantenible y modificable, se buscó desarrollar el planificador de forma que favoreciera la eventual adición de otros algoritmos de planificación a la aplicación. Por ejemplo, la GUI interactúa con el planificador únicamente a través de la interfaz IPlanificador, que define los métodos elementales necesarios para iniciarlo, pausarlo, y para bloquear o desbloquear procesos manualmente. Además, todo el código relacionado con la implementación del patrón Observer, así como la declaración de métodos internos comunes a cualquier tipo de planificador, se encuentra en la clase abstracta PlanificadorBase. De esta forma, cada algoritmo de planificación, representado por una clase concreta que hereda de la clase abstracta, debe preocuparse únicamente por implementar la lógica que es específica para ese algoritmo, reutilizando la mayor cantidad de código posible. Esto garantiza que la implementación de nuevos algoritmos de planificación es lo más sencilla posible, reduciendo también la posibilidad de errores.

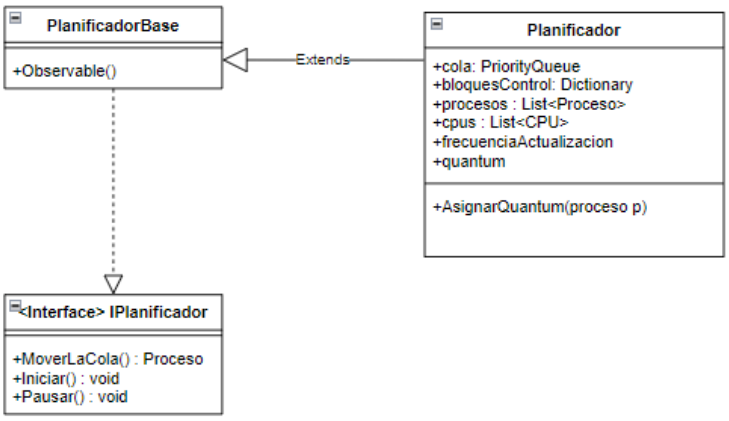


Ilustración 8 - Diagrama UML del planificador, se omite diagramas relacionados al patrón Observer

3. Representación de un Proceso:

Claramente, al trabajar con el paradigma de POO, un Proceso debe ser una abstracción de un proceso real (es decir que tenga ciertas propiedades, así como funcionalidades dentro de nuestro programa). La idea era representar lo más parecido a un Proceso real, es decir que posea todas las características que un proceso tiene. Por lo tanto se decidió que nuestra representación tendría una Id, una asignación (si es de SO o no es de SO), prioridad, un estado, duración de CPU, etc. (Se pueden ver en el diagrama UML). Otras características de los procesos (como la pila o el código del programa) no son necesarios puesto que no afectan la simulación como tal.

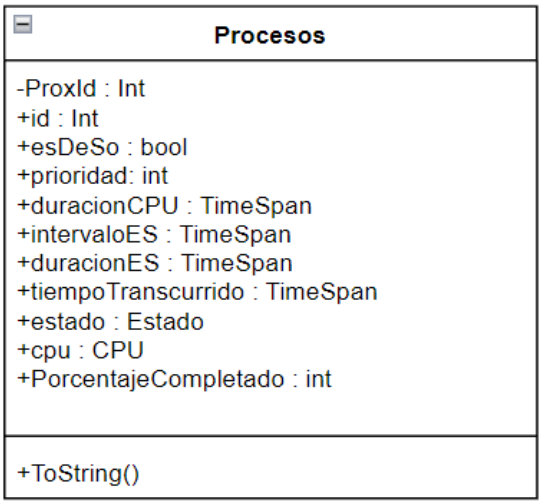


Ilustración 9 - UML procesos

También fue necesario crear una clase CPU, así como un enumerado de los posibles estados de un proceso (esto principalmente se debe a que un proceso solo puede tomar tres estados posibles además de bloqueado por usuario y finalizado). La clase CPU es más que nada debido a que una representación booleana de si tiene un CPU activo o no podría generar conflictos con una parte de la letra del obligatorio, la cual hace referencia a “Poder ingresar la cantidad de procesadores o cores”. Si se tuviese más de un procesador y la implementación fuese como se dijo anteriormente sería imposible monitorear en que CPU se está ejecutando el proceso. Además de almacenar el ID de cada CPU, se decidió incluir la posibilidad de configurar antes de iniciar la simulación la velocidad relativa de cada CPU, para ofrecer mejores capacidades de simulación.

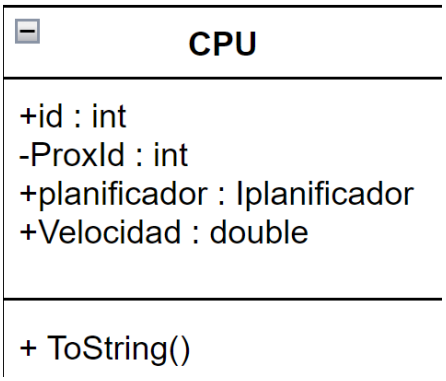


Ilustración 10 - UML de la clase CPU

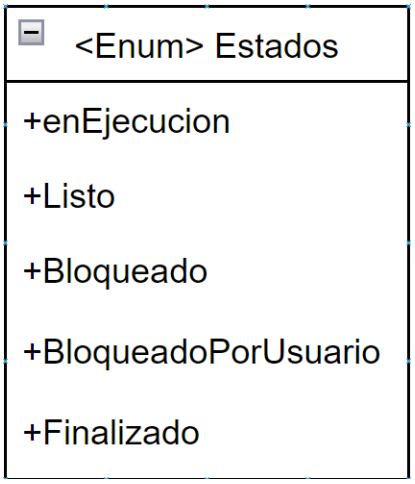


Ilustración 11 - UML del enum Estados

4. Funcionamiento del planificador:

Ahora que definimos todos los agentes involucrados en el planificador mismo, queda abordar el funcionamiento del planificador. Tomamos como punto de partida que el planificador deberá conocer la lista de procesos que debe ejecutar, y los CPU que tiene disponibles para asignarles. Se define que, con el fin de simplificar la ubicación de los procesos, se utilizarán distintas colas de procesos, lo cual vuelve también más intuitiva la programación. En total, el planificador conoce:

- La cola de procesos listos, ordenados por prioridad.
- La cola de procesos bloqueados por operaciones E/S, ordenados por el tiempo restante para terminar y volver a la cola de procesos listos.
- La lista de procesos bloqueados por usuario.
- La cola de procesos finalizados, ordenados por su ID.
- Los bloques de control asociados a cada proceso, los cuales almacenan datos necesarios para que el planificador pueda gestionarlos.

Otro aspecto importante del planificador es la presencia de un objeto de tipo Timer. Este se configura en cada actualización, calculando el tiempo hasta el próximo suceso de alguno de los procesos (finalización de proceso, operación E/S o quantum asignado a un proceso). De esta forma, cuando el Timer dispara su evento Elapsed, ha transcurrido el tiempo, y el planificador ejecuta una función que actualiza el estado de los eventos que sea necesario, notificando luego a la interfaz del nuevo estado de los procesos, y repitiendo el proceso hasta que detecte que no queda ningún proceso bloqueado, listo o en ejecución, momento en el cual se da por terminada la simulación.

5.- IMPLEMENTACION DE LA SOLUCION

PROCESOS: Lo primero que teníamos que implementar era la parte más importante de todas: los procesos, es decir teníamos que programar en C# una clase que modelara a estos mismos. Para esto lo que hicimos fue: tras la investigación de que es un proceso, así como lo que se nos pide decidimos programar la clase Proceso de la siguiente manera (basándonos en la implementación previamente realizada):

```
public class Proceso {

    private static int ProxId {get; set; }
    public int id { get; }
    public string nombre;
    public bool esDeSo;
    public int prioridad;

    public TimeSpan duracionCPU;
    public TimeSpan intervaloES;
    public TimeSpan duracionEs;
    public TimeSpan tiempoCPUTranscurrido;
    public TimeSpan tiempoESTranscurrido;
    public Estado estado;
    public CPU? cpu;

    public Proceso (PlantillaProceso plantilla){
        this.nombre = plantilla.nombre;
```

```
            this.duracionCPU = plantilla.duracionCPU;
            this.esDeSo = plantilla.esDeSo;
            this.intervaloES = plantilla.intervaloEs;
            this.duracionEs = plantilla.duracionES;
            this.estado = Estado.listo;
            this.tiempoCPUTranscurrido = TimeSpan.Zero;
            this.tiempoESTranscurrido = TimeSpan.Zero;
            this.id = ProxId;
            ProxId++;
        }
…
```

En esta implementación, se destacan dos cosas. La primera es la utilización de la estructura TimeSpan de C#, la cual permite una manipulación muy intuitiva de intervalos de tiempo, representándolos con fidelidad y facilitando las operaciones. Esto se complementa con una implementación de la operación módulo a través de un método de extensión Mod(this TimeSpan, TimeSpan), lo cual completa el conjunto de operaciones que es necesario realizar con los intervalos para esta aplicación.

La segunda es que en el constructor de proceso le pasamos un tipo PlantillaProceso, la cual es otra clase auxiliar que fue creada para una mejor legibilidad. Principalmente, permite añadir más ágilmente varios procesos a la simulación, en vez de tener que crearlos uno por uno. La clase en cuestión es la siguiente:

```
    public class PlantillaProceso
    {
        public string nombre {get;}
        public TimeSpan duracionCPU{get;}
        public TimeSpan duracionES {get;}
        public TimeSpan intervaloEs {get;}
        public bool esDeSo{get;}

        public PlantillaProceso(string nombre, int duracionCPU, int
duracionES, int intervaloEs, bool esDeSo)
```

```
        {
            this.nombre = nombre;
            this.duracionCPU = new TimeSpan(0, 0, duracionCPU);
            this.duracionES = new TimeSpan(0, 0, duracionES);
            this.intervaloEs = new TimeSpan(0, 0, intervaloEs);
            this.esDeSo = esDeSo;
        }
…
```

Como se puede observar plantilla proceso lo único que tiene son los elementos necesarios para crear un proceso inicialmente, por lo tanto, cuando Proceso recibe una instancia como parámetro lo único que se realiza es una asignación de las propiedades que la plantilla tiene.

En cuanto a la carga masiva de procesos, se creó una clase específica para la misma. La clase en cuestión es la siguiente:

```
    public static class CargaMasivaDatos{


        public static List<Proceso> CargarProcesos(string ruta){
            List<Proceso> lista = new List<Proceso>();
            string[] lineas = ManejadorArchivos.Leer(ruta);
            int duracionCPU;
            int duracionES;
            int intervaloEs;
            bool esDeSo;
            foreach (string linea in lineas){
                string[] datos = linea.Split(',');
                if(datos.Length == 5 && int.TryParse(datos[1],out duracionCPU)
                    && int.TryParse(datos[2], out duracionES) &&
                    int.TryParse(datos[3], out intervaloEs) &&
bool.TryParse(datos[4], out esDeSo))
                {
                    string nombre = datos[0];
                    PlantillaProceso plantilla = new PlantillaProceso(nombre,
duracionCPU, duracionES, intervaloEs, esDeSo);
                    Proceso proceso = new Proceso(plantilla);
                    lista.Add(proceso);
                }
```

```
            }
            return lista;
        }

    }
}
```

Lo que hace es dada una línea de un archivo de valores separados por coma y a cada valor asignarlo donde corresponde. Cabe destacar la siguiente línea:

```
                if(datos.Length == 5 && int.TryParse(datos[1],outduracionCPU)
                   && int.TryParse(datos[2], out duracionES) &&
                   int.TryParse(datos[3], out intervaloEs) &&
bool.TryParse(datos[4], out esDeSo))
```

Esta línea verifica si los datos ingresados por el archivo fueron cargados correctamente (Función TryParse) si se encuentra un error en la línea del archivo que representa un proceso cualquiera, se omite dicha línea.

CPU: La implementación no es muy diferente a la que se realizó inicialmente en el diagrama. Lo único a tomar en cuenta es la propiedad ProcesoActivo:

```
        public Proceso? ProcesoActivo
        {
            get
            {
                return procesoActivo;
            }
            set
            {
                if (value != null)
                {
                    value.Cpu = this;
                }
                procesoActivo = value;
            }
```

El método set de la propiedad recibe el proceso que se quiere asignar al procesador, y en caso de no ser nulo, se le asigna esta instancia su atributo Cpu, para luego asignar el proceso a la propiedad de la instancia de CPU. De esta forma, ambas instancias quedan correctamente enlazadas, para facilitar la navegabilidad entre los objetos. Otra aclaración que vale hacer es la presencia del carácter “?” junto al tipo de la propiedad. Este operador únicamente especifica que está previsto que esta propiedad almacene valores nulos, de forma que el código que accede a ella puede tenerlo en consideración y evitar de esta forma errores en tiempo de ejecución, al realizar los chequeos pertinentes.

El planificador: Aquí como bien se dijo antes, el planificador no está representado por una clase sola, sino es una serie de interfaces y herencias para que el planificador quede lo más reutilizable, así como lo menos cargado posible. Cabe destacar que tanto las interfaces IPlanificador como IObservable son exactamente igual a como estaban pensadas en un inicio. Por lo que su implementación en código será omitida lo único a destacar es que el tipo que recibe IObservable en el IPlanificador es PlanificadorBase.Estado. Esto indica que el observable enviará información de ese tipo a sus observadores, limitando al mismo tiempo sus observadores a aquellos que se declare que implementan IObservador<PlanificadorBase.Estado>, lo cual permite aprovechar al máximo el tipado estático de C#, sin sacrificar flexibilidad gracias a los tipos genéricos.

```
    public interface IPlanificador : IObservable<PlanificadorBase.Estado>{
        /// <summary>
        /// Inicia la ejecuci�n de los procesos asignados a este
planificador.
```

```
        /// </summary>
        void Iniciar();

        /// <summary>
        /// Pausa la ejecuci?n de los procesos asignados a este planificador.
        /// </summary>
        void Pausar();

        bool Pausado { get; set; }
        event EventHandler PausadoChanged;

        void BloquearProceso(Proceso p);
        void DesbloquearProceso(Proceso p);
    }
```

Como se mencionó anteriormente, tenemos una clase PlanificadorBase la cual es una clase abstracta que sirve de molde para los demás planificadores. En el mismo lo que podemos observar es:

```
…
      public PlanificadorBase()
        {
            observadores = new List<IObservador<Estado>>();
        }
…
        public abstract void Iniciar();
        public abstract void Pausar();
        public abstract void BloquearProceso(Proceso p);
        public abstract void DesbloquearProceso(Proceso p);
        protected abstract void Notificar();
        protected abstract void Notificar(IObservador<Estado> observador);
        protected abstract Estado GenerarEstado();
…
        public class Estado
        {
            public IOrderedEnumerable<Proceso> listos;
            public IOrderedEnumerable<Proceso> bloqueados;
            public IOrderedEnumerable<Proceso> finalizados;
            public List<Proceso> bloqueadosPorUsuario;
            public List<CPU> cpus;

            public Estado(IOrderedEnumerable<Proceso> listos,
IOrderedEnumerable<Proceso> bloqueados,
```

```
                IOrderedEnumerable<Proceso> finalizados, List<Proceso>
bloqueadosPorUsuario,
                List<CPU> cpus)
            {
                this.listos = listos;
                this.bloqueados = bloqueados;
                this.finalizados = finalizados;
                this.bloqueadosPorUsuario = bloqueadosPorUsuario;
                this.cpus = cpus;
            }
        }
…
```

Solo se agregaron las partes de la clase que nos parece importante destacar.

Primero es que todo planificador se va a crear con una lista de observadores. Esto sin discriminar que tipo de algoritmo se utilice. Luego definimos las operaciones básicas que ha de tener el mismo (Estas sin métodos, pues es una clase abstracta y no nos interesa definir la forma en que van a ser utilizados, puesto que puede variar dependiendo el algoritmo y/o de la implementación). Estas operaciones son: iniciar, pausar, bloquear proceso, desbloquear proceso y generar estado. Además, contiene la definición de una clase anidada Estado, que representa el tipo de datos que el planificador notificará a sus observadores. Esto remarca la importancia de que las interfaces IObservador e IObservable se declaren utilizando el tipo genérico correcto.

Por último, tenemos la clase que va a implementar nuestro Planificador con el algoritmo que queremos la cual se llama PlanificadorRoundRobin, la cual extiende de PlanificadorBase. De acuerdo con su lógica específica, se implementaron los siguientes métodos:

- Constructor único, que recibe una lista de procesos y CPUs, junto con el tiempo máximo que un proceso podrá usar una CPU, y el intervalo que se utilizará para realizar la actualización y notificación de datos. El constructor

almacena los parámetros recibidos, e inicializa las estructuras de datos requeridas y el comparador de prioridad necesario para cada cola. Además, realiza la asignación inicial de procesos a CPUs y determina su quantum.

- Iniciar(): configura el Timer interno y lo inicia.
- Pausar(): detiene el Timer interno.
- MoverLaCola(): verifica si existe algún proceso listo que pueda ser asignado a un CPU. Si es así, setea su envejecimiento a 0 para restablecer su prioridad a la inicial, mientras incrementa el envejecimiento de los otros procesos listos, y devuelve el proceso extraído de la cola.
- AsignarQuantum(): almacena en el BCP de un proceso su quantum, considerando posibles operaciones E/S, tiempo estimado para finalización, y tiempo máximo que cualquier proceso puede usar una CPU.
- DuracionSiguienteTimer(): propiedad solo lectura que devuelve el tiempo estimado hasta cualquier suceso de un proceso, o el tiempo hasta la siguiente actualización de estado, si es menor.
- ActualizarEstado(): se ejecuta cada vez que el Timer dispara el evento Elapsed. Actualiza el tiempo transcurrido de operaciones E/S pendientes y desbloquea los que hayan terminado. Además, reasigna los CPU que estén inactivos, o cuyos procesos hayan llegado a una operación E/S, hayan finalizado o hayan agotado su quantum. Notifica a los observadores del

planificador del nuevo estado, y configura nuevamente el Timer a menos que la simulación haya terminado.

- ReasignarCPU(): actualiza el estado del proceso actual del CPU, lo coloca en la nueva cola o lista según corresponda, y asigna un nuevo proceso en caso de ser posible.
- BloquearProceso(): bloquea manualmente un proceso listo o en ejecución, hasta que se desbloquee externamente, y notifica el nuevo estado.
- DesbloquearProceso(): desbloquea manualmente un proceso bloqueado por usuario, y lo coloca en la cola de listos.
- Notificar(): notifica a el/los observadores del estado actual de los procesos del planificador.
- GenerarEstado(): genera un objeto del tipo PlanificadorBase.Estado que almacena el estado de todos los procesos que gestiona el planificador.

Se destacan las decisiones de diseño tomadas, que se cree favorecen la mantenibilidad y extensión del código. Al tener todas estas partes por separado, es decir el IPlanificador, tener un planificador base lo que nos permite y nos va a habilitar es tener una buena reutilización del código. En caso de que a futuro se deseara implementar un planificador con un nuevo tipo de algoritmo lo único que se debe hacer es crear una clase que extienda de PlanificadorBase y programar solamente el algoritmo.

4. Análisis de la implementación:

Tras programar la interfaz de usuario, así como las clases el resultado obtenido fue el siguiente:

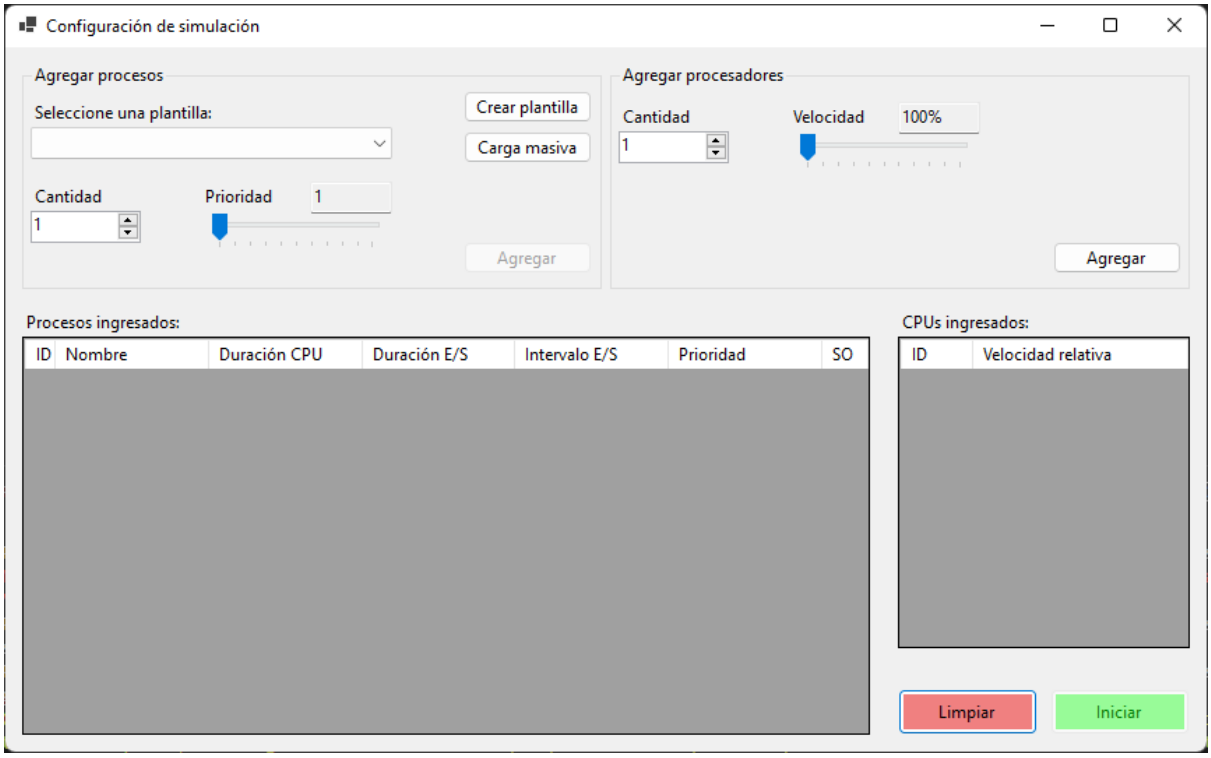


Ilustración 12 - Interfaz Usuario inicial del programa

En la siguiente ventana podemos diferenciar varias secciones. Iremos comentando una a una. La primera es la zona de Agregar procesos. Como bien dice el nombre, Agregar procesos es donde se van a cargar los procesos. Tal como pedía la letra hay dos formas de realizarlo:

- **Procesos individuales:**

En crear plantilla lo que nos permite es crear una plantilla de un proceso, es decir nos da un molde, donde tenemos que rellenar con las características que tiene un proceso (Descritas en secciones anteriores). En la misma se pide seleccionar un nombre, así como una duración del CPU, una duración de E/S si es que tiene y si tiene también tiene un intervalo entre E/S (la cual por defecto es 1).

Una vez creada la plantilla se puede agregar la cantidad de procesos que uno desee, así como también asignarle una prioridad. Si se desea crear un proceso nuevo lo único que se debe hacer es repetir el paso anterior con el proceso que se desee usar.

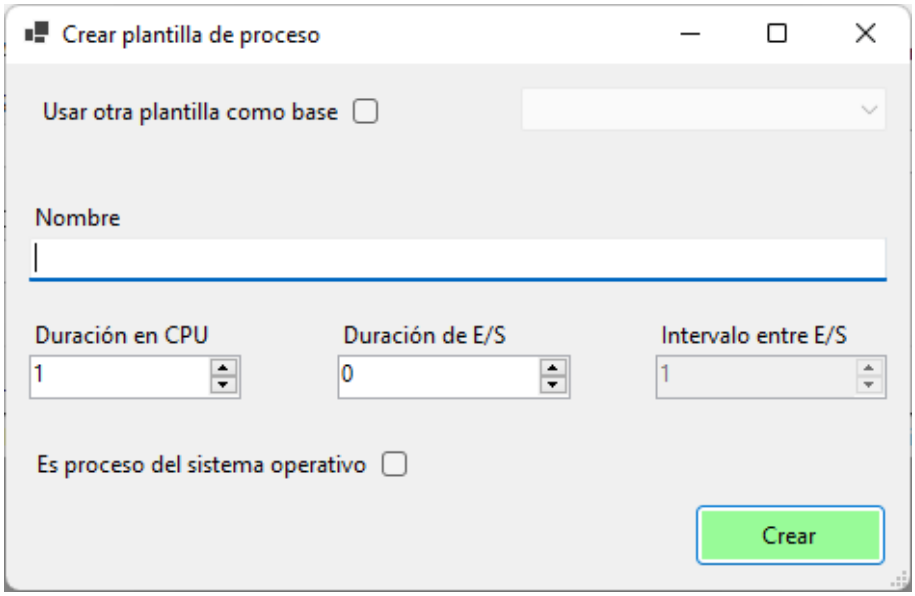


Ilustración 13 - Cuadro de dialogo para crear una plantilla de proceso

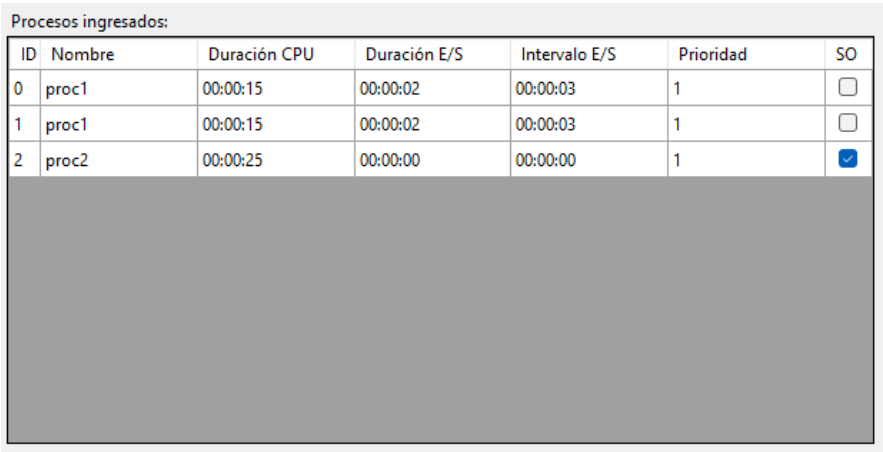


Ilustración 14 - 3 procesos agregados

- Carga masiva: El botón de carga masiva abre una pestaña de Windows.

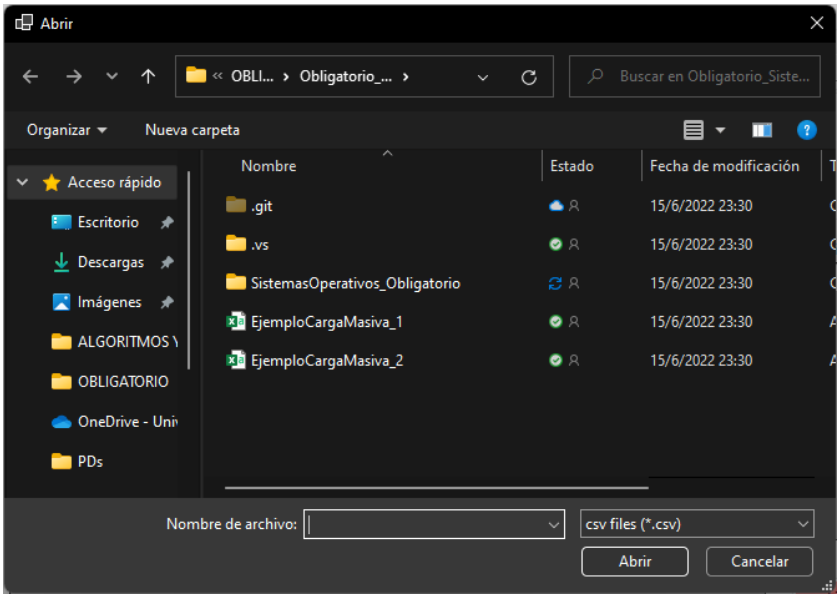


Ilustración 15 - Ventana de búsqueda

La única forma de realizar la carga masiva de procesos es ingresarlos como un archivo de valores separado por comas (.csv). Con el siguiente formato:

```
Nombre Proceso, duración CPU, duración E/S, intervalo E/S, es de SO
```

Si no posee esta forma, el código no va a permitir ingresar el proceso. Es decir, se debe respetar los tipos asignados a cada valor, en caso de que no se pueda realizar esto, el programa mismo no lo identificara como un valor valido.

Un ejemplo a esto es:

```
proc0,29,7,5,false

proc1,54,6,9,false

proc2,44,7,1,true
```

Estos valores fueron extraídos del EjemploCargaMasiva2.csv provisto en el repositorio del proyecto. Como se puede observar la idea de los valores es que sea

palabra, numero, numero, numero, true o false. Esto se debe a que al cargarlos en el programa van a ser transformados a distintos tipos de datos dependiendo de lo que represente, por lo tanto, si uno de estos datos está mal ingresado el programa no lo tomará como correcto y no lo tendrá en cuenta a la hora de la representación.

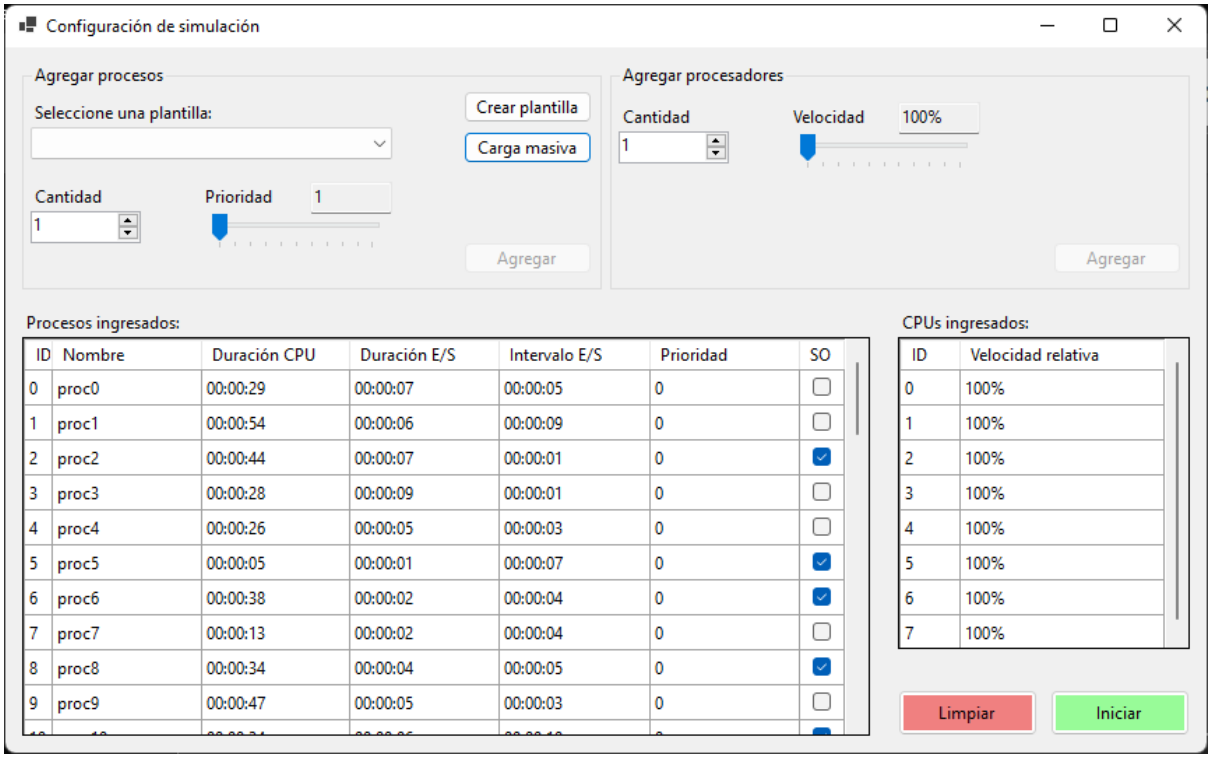


Ilustración 16 - Carga masiva de Procesos

La sección de la derecha es la sección que denominamos agregar procesadores, esto lo que hará es más que nada agregar nuevos procesadores a nuestro planificador, podemos modificarle la velocidad al mismo y agregar hasta 8 procesadores. Una vez ingresado queda así:

CPUs ingresados:

ID	Velocidad relativa
0	100%
1	100%
2	100%
3	100%
4	100%
5	100%
6	100%
7	100%

Ilustración 17 - CPUs Ingresados

Una vez que tanto hallan procesos agregados así como CPUs, podemos darle a iniciar (Sin ninguno de estos dos no arranca ninguno). Y ahí empieza la simulación. Se mostrará la siguiente ventana:

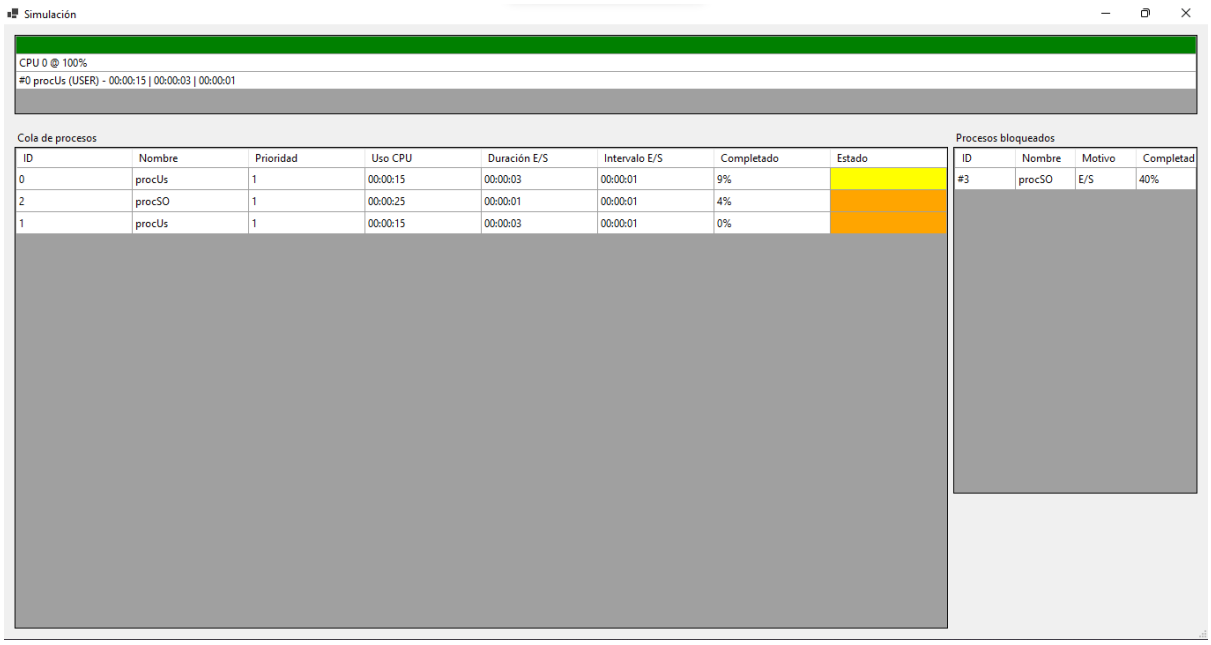


Ilustración 18 - Simulación

En el rectángulo superior se observa los CPUs, en este caso tenemos solo uno, pero podría haber hasta ocho. En el mismo muestra que CPU esta activo y a qué velocidad

está funcionando. Así como también que proceso se está ejecutando en ese momento.

La cola de procesos nos muestra todos los procesos que están en la cola, esperando a que se le asigne el CPU. Para esto usamos un código de colores que funciona de la siguiente manera, cada uno mostrando un estado del proceso.

Amarillo: En ejecución.

Naranja: Listo

Verde: Terminado

Por último, tenemos la cola de bloqueados, esta representa los procesos que, justamente, se encuentran bloqueados, estos pueden ser por diversos motivos (tal como lo especificaba la letra). El motivo aparece especificado en el programa.

Los motivos pueden darse por dos maneras (según estipulaba la letra):

- Por E/S: es totalmente automático y ajeno al usuario. Se bloqueará automáticamente.
- Por Usuario: es manual. Para realizarlo el usuario debe detener el planificador en el botón “Detener”, luego de esto debe seleccionar el proceso o procesos a bloquear y apretar el botón correspondiente. Luego de esto se podrá visualizar como bloqueado en la lista de bloqueados. Para desbloquear el usuario debe volver a pausar la simulación y desbloquear los procesos que él mismo quiera.

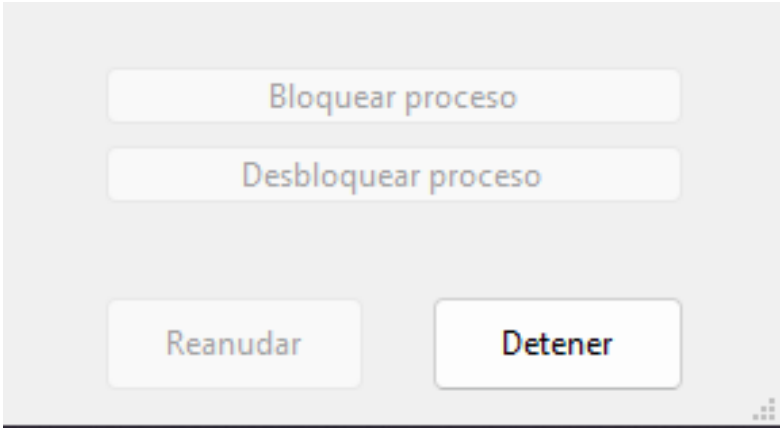


Ilustración 19 - Detener planificador

Cabe destacar que los procesos de SO tienen mayor prioridad que los de usuario. Al terminar la simulación, el programa se visualiza de la siguiente manera:

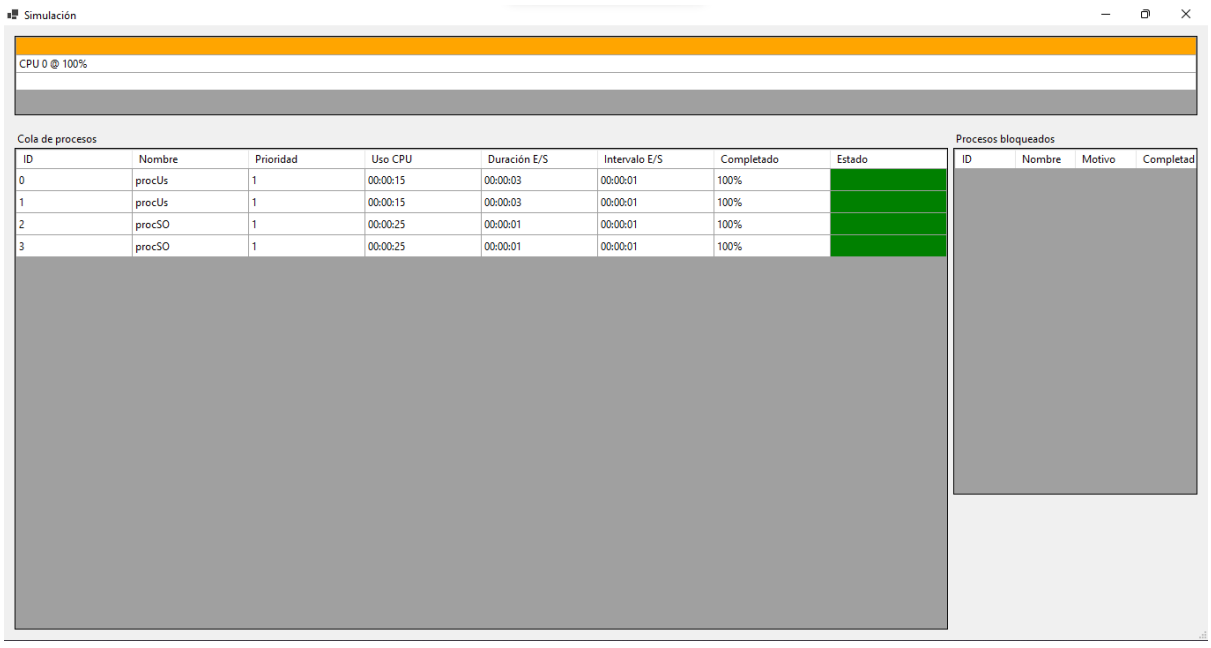
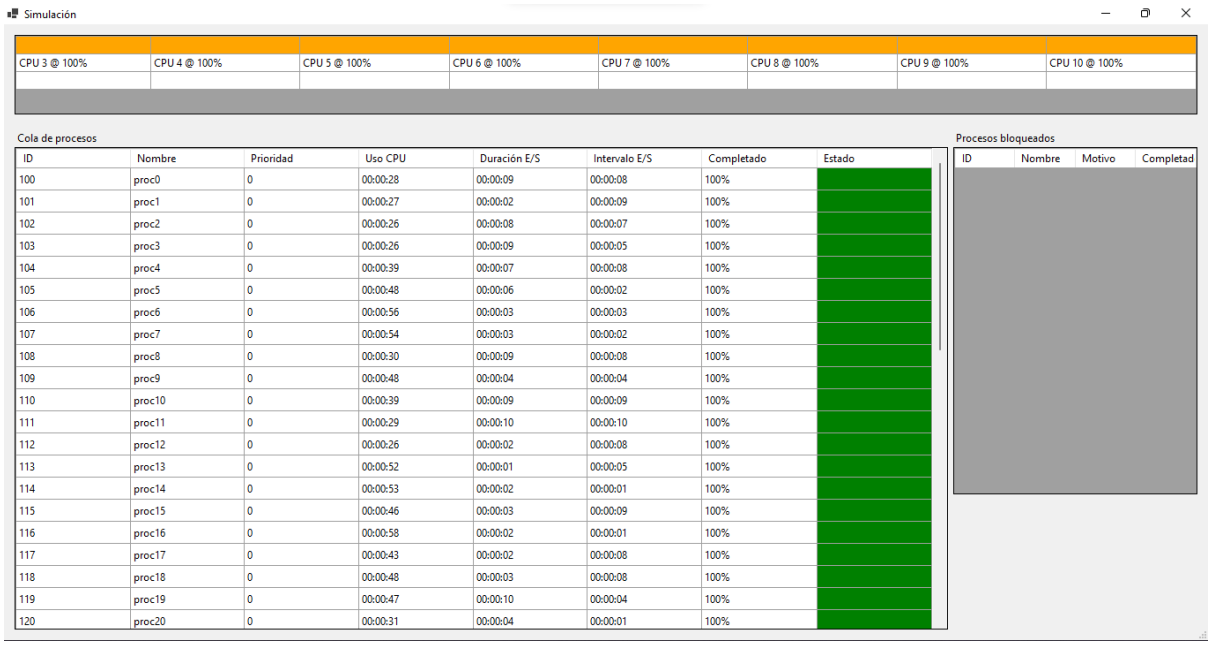


Ilustración 20 - Simulación terminada

En la ilustración de arriba se puede observar que todos los procesos tienen el estado de color verde, lo que significa que el mismo ha terminado y, por lo tanto,

también la simulación. A destacar que en la pestaña de CPU el color naranja significa sin asignar, y hace referencia a que el CPU no está asignado a ningún proceso.

5. Conclusiones:



ID	Nombre	Prioridad	Uso CPU	Duración E/S	Intervalo E/S	Completado	Estado
100	proc0	0	00:00:28	00:00:09	00:00:08	100%	
101	proc1	0	00:00:27	00:00:02	00:00:09	100%	
102	proc2	0	00:00:26	00:00:08	00:00:07	100%	
103	proc3	0	00:00:26	00:00:09	00:00:05	100%	
104	proc4	0	00:00:39	00:00:07	00:00:08	100%	
105	proc5	0	00:00:48	00:00:06	00:00:02	100%	
106	proc6	0	00:00:56	00:00:03	00:00:03	100%	
107	proc7	0	00:00:54	00:00:03	00:00:02	100%	
108	proc8	0	00:00:30	00:00:09	00:00:08	100%	
109	proc9	0	00:00:48	00:00:04	00:00:04	100%	
110	proc10	0	00:00:39	00:00:09	00:00:09	100%	
111	proc11	0	00:00:29	00:00:10	00:00:10	100%	
112	proc12	0	00:00:26	00:00:02	00:00:08	100%	
113	proc13	0	00:00:52	00:00:01	00:00:05	100%	
114	proc14	0	00:00:53	00:00:02	00:00:01	100%	
115	proc15	0	00:00:46	00:00:03	00:00:09	100%	
116	proc16	0	00:00:58	00:00:02	00:00:01	100%	
117	proc17	0	00:00:43	00:00:02	00:00:08	100%	
118	proc18	0	00:00:48	00:00:03	00:00:08	100%	
119	proc19	0	00:00:47	00:00:10	00:00:04	100%	
120	proc20	0	00:00:31	00:00:04	00:00:01	100%	

ID	Nombre	Motivo	Completad

Ilustración 21 - Simulación terminada

Como conclusiones generales podemos decir que el obligatorio fue existoso, esto se debe a que cumplimos con todos los requisitos planteados por la cátedra según lo detallado en la sección “Analisis de la implementacion”. Se puede apreciar cómo en una simulación con una cantidad considerable de procesos y CPUs todos los procesos se desplazan entre los estados posibles de acuerdo a lo estudiado en el marco teórico. También se puede observar a través de esto la ejecución exitosa del algoritmo de planificación a corto plazo escogido, tanto en la determinación de prioridad de los procesos como en la asignación de recursos simulados, lo cual cumple con las características de la planificación de un SO real que se buscaban representar con la realización de esta aplicación.

Como conclusiones especificas podemos decir que la implementacion de un proceso tambien fue exitosa. Puesto que se logró representar los procesos con todos los atributos relevantes para la aplicación, a la vez que se consiguió manipularlos y visualizarlos de manera adecuada. Además, se logró validar que las transiciones entre estados de un proceso siempre se dieran siguiendo el funcionamiento estudiado de un planificador. También se considera digno de mención que se haya logrado una GUI cómoda para el usuario, tanto para la configuración inicial de la simulación, como para la interacción con los procesos durante la simulación, permitiendo el bloqueo y desbloqueo manual de procesos.

Bibliografía

Cabalar, P. (s.f.). *Depto. de Computación Universidade da Coruña - TEMA III. PROCESOS*. Obtenido de https://www.dc.fi.udc.es/~so-grado/SO-Procesos-planif.pdf

D., S. G. (2005). Obtenido de https://www.linuxtotal.com.mx/index.php?cont=info_admon_008#:~:text=Los%20usuarios%20en%20Unix%2FLinux,m%C3%A1s%20grupos%20adem%C3%A1s%20del%20principal.&text=Tambi%C3%A9n%20llamado%20superusuario%20o%20administrador

Guglielmetti, G. (2022). *4 - Planificacion.* Universidad Católica Uruguay.

Guglielmetti, G. (2022). *Sistemas Operativos 2022, Obligatorio.*

Microsoft Corporation. (2022). *.NET documentation*. Obtenido de https://docs.microsoft.com/en-us/dotnet/

Microsoft Corporation. (2022). *PriorityQueue<TElement,TPriority> Class*. Obtenido de https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.priorityqueue-2?view=net-6.0

Tanenbaum, A. S. (2009). *Sistemas operativos modernos.* Mexico: Pearson Prentice Hall.

UDELAR, F. (2014). *Curso 2014 - Planificacion*. Obtenido de https://www.fing.edu.uy/inco/cursos/sistoper/recursosTeoricos/6-SO-Teo-Planificacion.pdf

Hoja testigo