Simulador de Procesos de Sistemas Operativos

Participantes**:**

Chanampa, Leandro Ariel
Gasparutti, Edgardo Maria
Luque Acosta, Edgar Yamil
Zapata, Rodrigo
Zini, Nicolás Adrian

Grupo: G9

Carrera: Ingeniería en Sistemas de la Información

Índice

Introducción

El presente informe responde al Trabajo Práctico Integrador de la cátedra de Sistemas Operativos, el objetivo del mismo es documentar el proceso de desarrollo de un programa que simulará el funcionamiento de un sistema operativo en lo que se refiere el tratamiento de procesos, más precisamente a la planificación de procesos a corto plazo y a la administración de memoria.

El programa deberá poder mostrar el ciclo de vida completo de un proceso desde que este llega hasta que finaliza y el desempeño de los distintos tipos de algoritmos de tratamiento de memoria y de procesos.

El simulador en cuestión consistirá de un proceso de recolección de datos (Entradas) en el que el usuario deberá suministrar los datos correspondientes a los procesos y las preferencias para el tratado de los mismos. También dispondrá de una lógica de tratamiento de los datos de entrada y por último de la visualización de los resultados (Salidas).

Para la realización de este simulador se utilizará los conceptos teóricos vistos en clase durante el cursado de la materia, principalmente los vistos en la unidad 2 (Administración de Memoria) y la unidad 3 (Procesos).

Para la gestión del proyecto se utilizará la metodología de trabajo “Scrum” (de metodologías ágiles). El proyecto será ejecutado basado en el orden que conforman los “sprint” indicados en la consigna, y será ejecutado por los integrantes de este grupo siguiendo los siguientes roles:

1. Zapata, Rodrigo - Scrum Master
2. Luque Acosta, Edgard Yamil - Programador
3. Zini, Nicolás - Tester
4. Chanampa, Leandro - Lógica
5. Gasparutti, Edgardo - Documentador

Entradas del simulador

Administración de memoria:

El usuario deberá cargar los datos asociados a la administración de memoria:

1) Tamaño de memoria: Se deberá cargar la longitud de la memoria.
2) Porcentaje utilizado por el sistema operativo: Indicar (de la longitud de memoria definida en el punto anterior) el porcentaje ocupara el sistema operativo.
3) Tipo de memoria: Se deberá indicar el tipo de memoria a utilizar entre el siguiente conjunto: “Particiones Fijas” o “Particiones Variables”.
4) Algoritmo de asignación: Se deberá ingresar el algoritmo de asignación de memoria a utilizar, en el caso de que el usuario en (2) haya seleccionado Memoria con partición fija deberá elegir entre el siguiente conjunto: (“Best Fit”, “First Fit”), y en caso de que haya seleccionado Memoria con partición variable deberá elegir entre el siguiente conjunto: (“First Fit”, “Worst Fit”).
5) Cantidad de particiones: En el caso de que en (2) el usuario haya seleccionado Memoria con partición fija, en el usuario deberá ingresar la cantidad de partes en que se fraccionó la memoria.
6) Tamaño de cada partición: En el caso de que en (2) el usuario haya seleccionado Memoria con partición fija, en el usuario deberá ingresar la longitud de cada partición de memoria seleccionada.

Administración de procesos:

El usuario deberá cargar los datos asociados a la administración de procesos:

1) Tipo de algoritmo de administración de procesos: Se deberá indicar el algoritmo a utilizar entre el conjunto predefinido (“FCFS, “Round-Robín”, “SJF”,” SRTF” y “Colas Multinivel”).

2) Quantum: En el caso de que el usuario en (1) haya seleccionado el algoritmo Round-Robín, se activará la opción para ingresar la cantidad de tiempo de quantum que se utilizara.

3) Colas: El caso de que el usuario en (1) haya seleccionado el algoritmo Colas Multinivel, se activará la opción para ingresar el algoritmo de planificación con el cual trabajara cada una.

4) Carga de procesos: El usuario podrá ingresar los procesos con los que desea trabajar. Como mínimo deberá ingresar uno, y deberá especificar los siguientes datos:

    a) Tiempo de Arribo: El usuario deberá ingresar el tiempo en que el proceso entra a la cola de nuevos.

    b) Tamaño del Proceso: El usuario deberá ingresar el tamaño en memoria que ocupará el proceso.

    c) Tiempo de Irrupción CPU: El usuario deberá ingresar el tiempo que el proceso le llevará finalizar en el procesador.

    d) Tiempo de Entrada(opcional): En el usuario deberá ingresar el tiempo de entrada que el proceso requerirá.

    e) Tiempo de Salida(opcional): En el usuario deberá ingresar el tiempo de salida que el proceso requerirá.

    f) En el caso de seleccionar ráfagas de e/s, luego de definirlas deberá volver a cargarse una ráfaga de CPU para finalizar.

    g) Colas: En el caso de que el usuario en (1) haya seleccionado el algoritmo por Colas Multinivel, se activará una opción para ingresar la cola que le corresponderá al proceso.

Restricciones de las entradas del simulador:

a) Tamaño de memoria: La memoria deberá tener un tamaño entre 10 kb y 4096 kb.

b) Sistema operativo en memoria: El sistema operativo no podrá ocupar menos de un 5% de la memoria ni más de un 20% de la misma.

c) Tamaño de cada partición: Ninguna partición puede tener longitud cero y la sumatoria de las particiones debe ser igual a la longitud de la memoria no ocupada por el sistema operativo, memoria que quede sin asignar no será tomada en cuenta para el uso del planificador.

d) Quantum: el tiempo de quantum seleccionado para el algoritmo round-robín no puede ser cero y no podrá ser mayor que 10.

e) Quantum: el tiempo de quantum seleccionado para el algoritmo colas multinivel no puede ser cero y no podrá ser mayor que 3 para la cola de alta prioridad, en cambio para la cola de media prioridad el quantum no podrá ser cero y no podrá ser mayor que 5.

f) Prioridad: el valor de prioridad ingresado para cada uno de los procesos deberá ser string seleccionado desde un combo box, y se podrá elegir alta, media o baja prioridad.

g) Colas: La cantidad de colas multinivel está fijada en únicamente, tres colas. A la hora de seleccionar el algoritmo de planificación de cada cola multinivel no se podrá elegir colas multinivel de nuevo.

h) Tamaño del proceso: la longitud del proceso debe ser mayor que cero y menor que la longitud de la memoria.

i) Tiempo de arribo/irrupción/entrada/salida: la cantidad ingresada deberá ser entera y positiva.

Salidas del simulador

Luego de que el simulador finalice el tratamiento de los procesos previamente cargados, mostrará por pantalla las siguientes salidas:

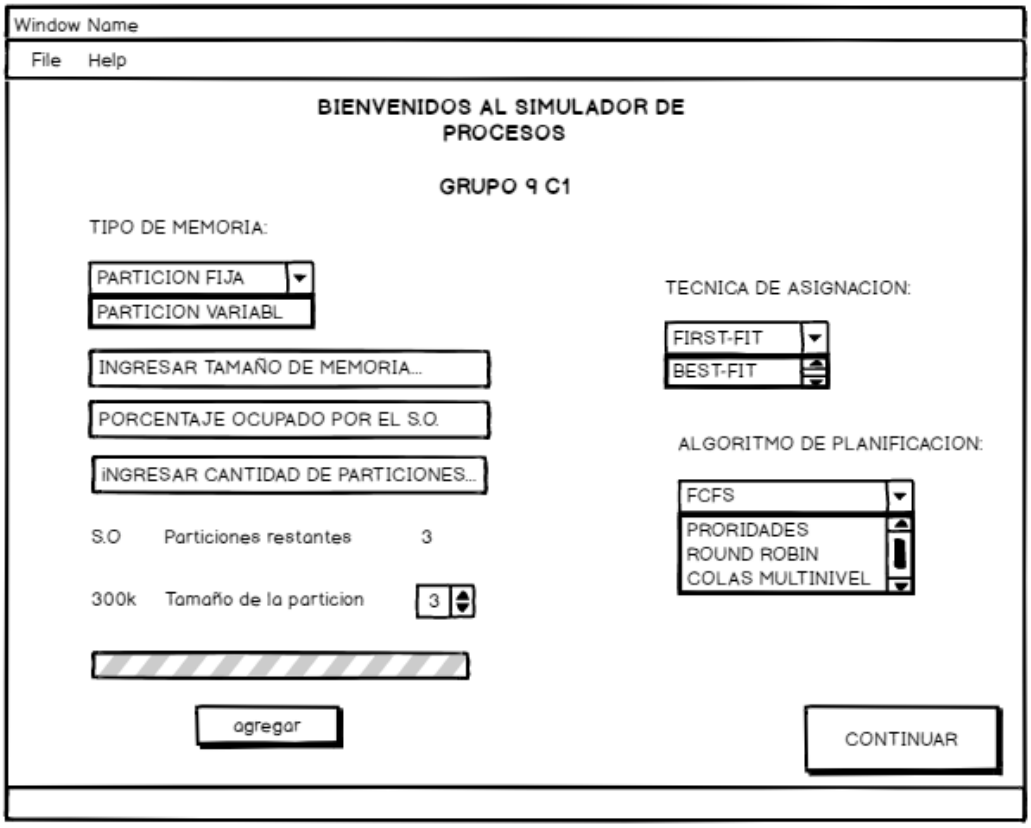
1) Mapa de memoria, donde se indicará qué parte de la memoria está libre y que parte de la misma estará ocupada y por qué proceso.
2) Cola de procesos nuevos.
3) Cola de procesos listos.
4) Diagrama de Gantt para el progreso de los procesos en la CPU y para el tratamiento de ráfagas de entrada y salida, este podrá descargarse para luego compararlo con otro diagrama .
5) Tiempo medio de espera y tiempo medio de retorno de los procesos.

El simulador tendrá la opción de mostrar en cada unidad de tiempo como fue evolucionando cada uno de las salidas mencionadas.

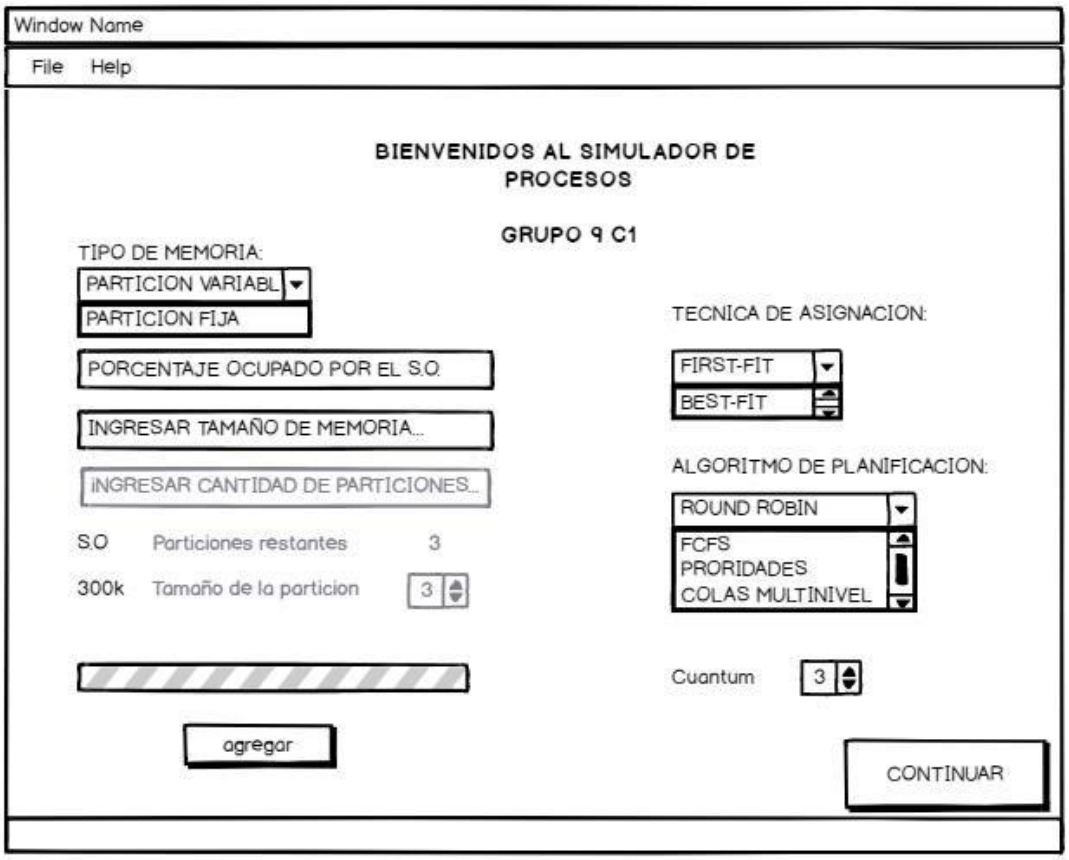
El simulador también tendrá la opción de comparar el resultado (salidas) de dos algoritmos y configuraciones distintas con la misma entrada de procesos, para poder mostrar que configuración es la más adecuada para una cierta lista de procesos.

Mockups del Simulador (navegación de pantallas)

En la ventana principal del simulador, la primera con la que nos encontramos al iniciar el programa, aparecerán todas las opciones de configuración de la memoria y de cpu, donde se deberán ingresar todos los datos necesarios para el funcionamiento del simulador. En ella podremos especificar el tipo de particiones de memoria, el tamaño y porcentaje de uso por parte del s.o. y además los algoritmos de asignación de memoria y de planificación de procesos. Todos estos datos están detallados anteriormente en la sección de “Entradas del Simulador”.

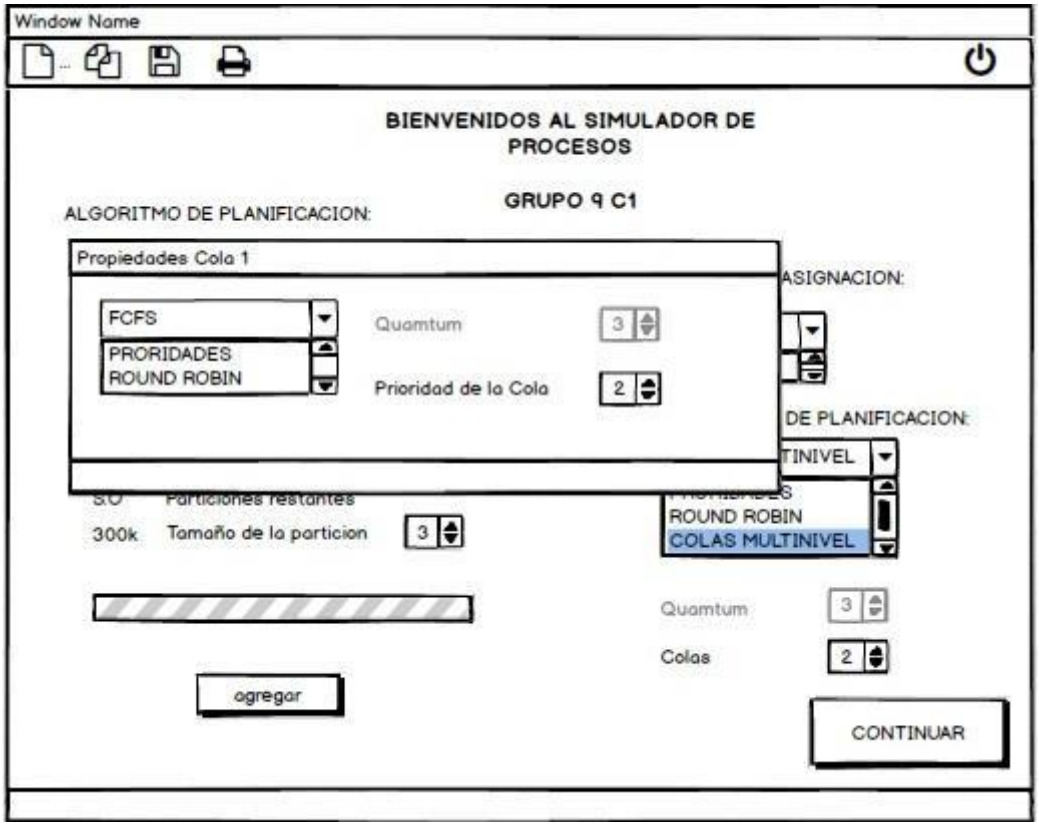


A medida que vayamos rellenando los campos solicitados, se irán ocultando algunas entradas para cumplir con ciertas restricciones detalladas en la sección de “Restricciones de Entrada”, por ejemplo que el usuario no agregue cantidad de particiones si este seleccionó previamente una memoria con particiones variables.



Si el usuario seleccionó particiones fijas, aparecerá la opción de agregar (una por una) cada partición con su respectivo tamaño, además el progreso en la carga de las particiones y la proporción de las mismas sobre el total se verán simbolizados en una barra como la se muestra en la imagen.

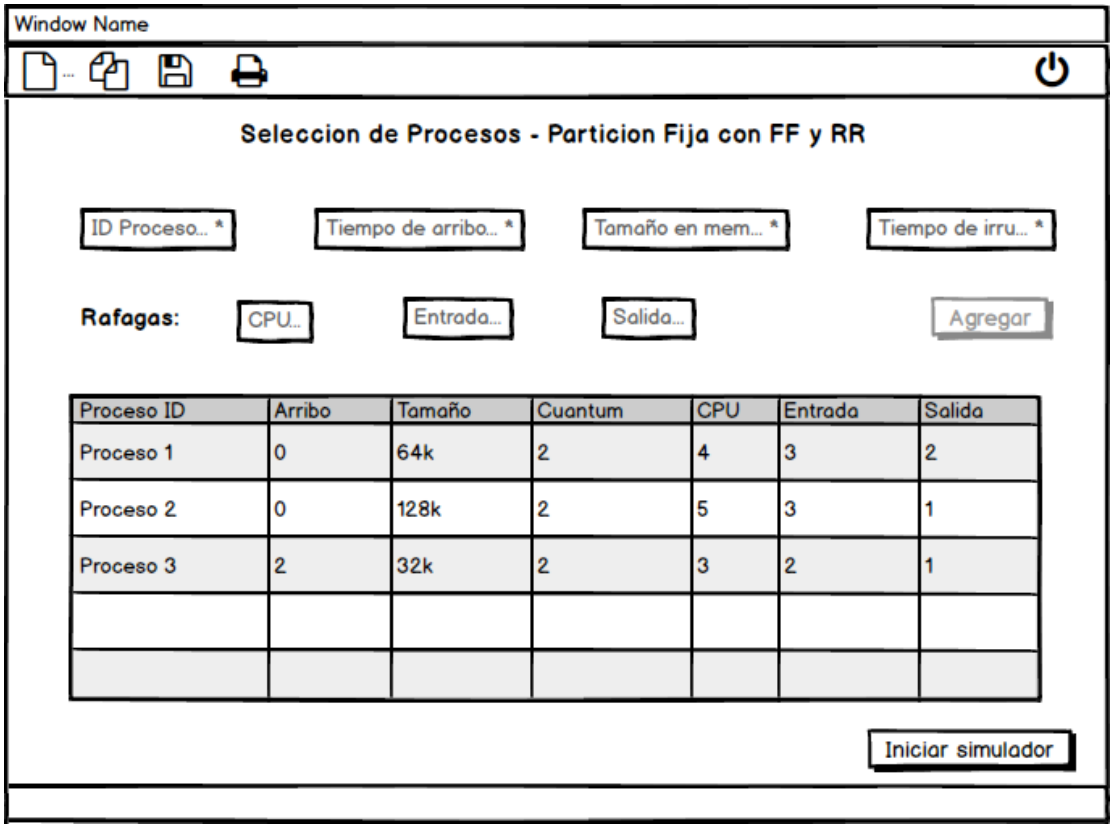
En el caso de que se seleccione tipo de algoritmo “colas multinivel” aparecerá una nueva ventana donde se deberá indicar el algoritmo de planificación correspondiente a cada cola seleccionada.



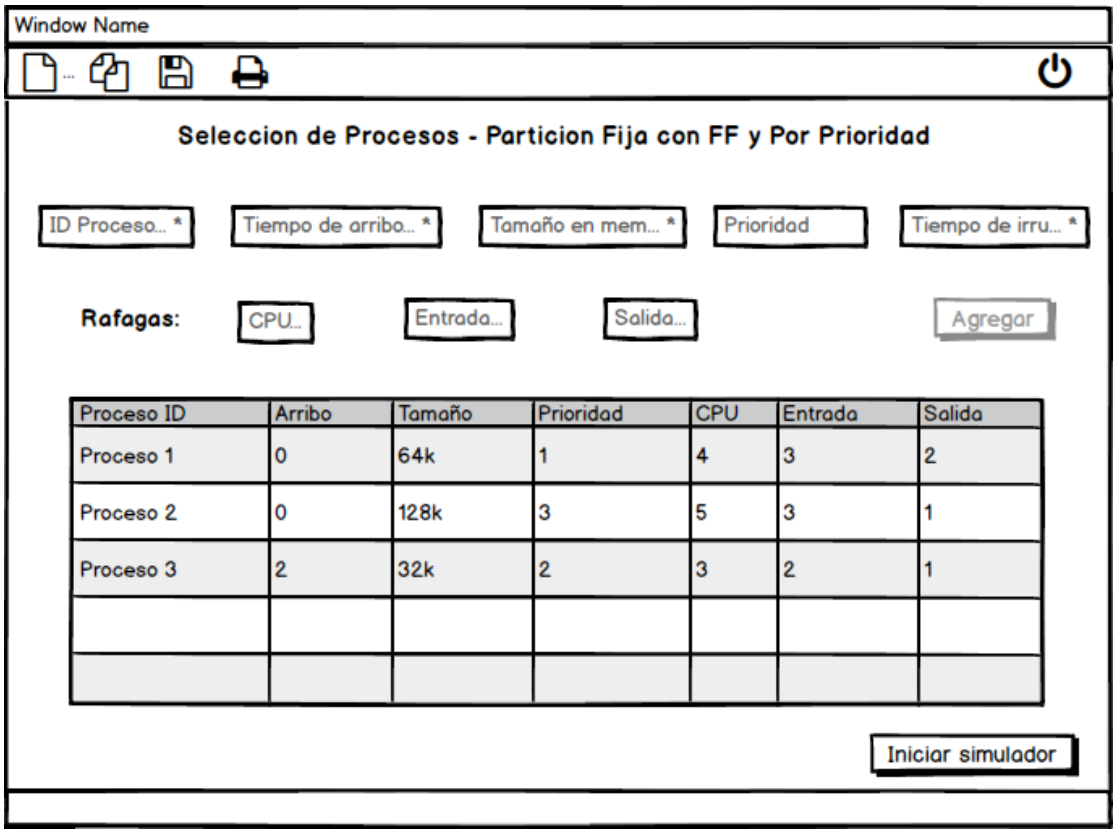
Cuando el usuario rellene todos los datos de entrada (todas las entradas de datos visibles son obligatorias), se podrá hacer clic en el botón de continuar para pasar a la siguiente ventana.

En la siguiente ventana el usuario deberá cargar los procesos con los que desea probar el simulador. En ellos deberá cargar las entradas correspondientes a los procesos, también anteriormente detallados en la sección de “Entradas del Simulador”.

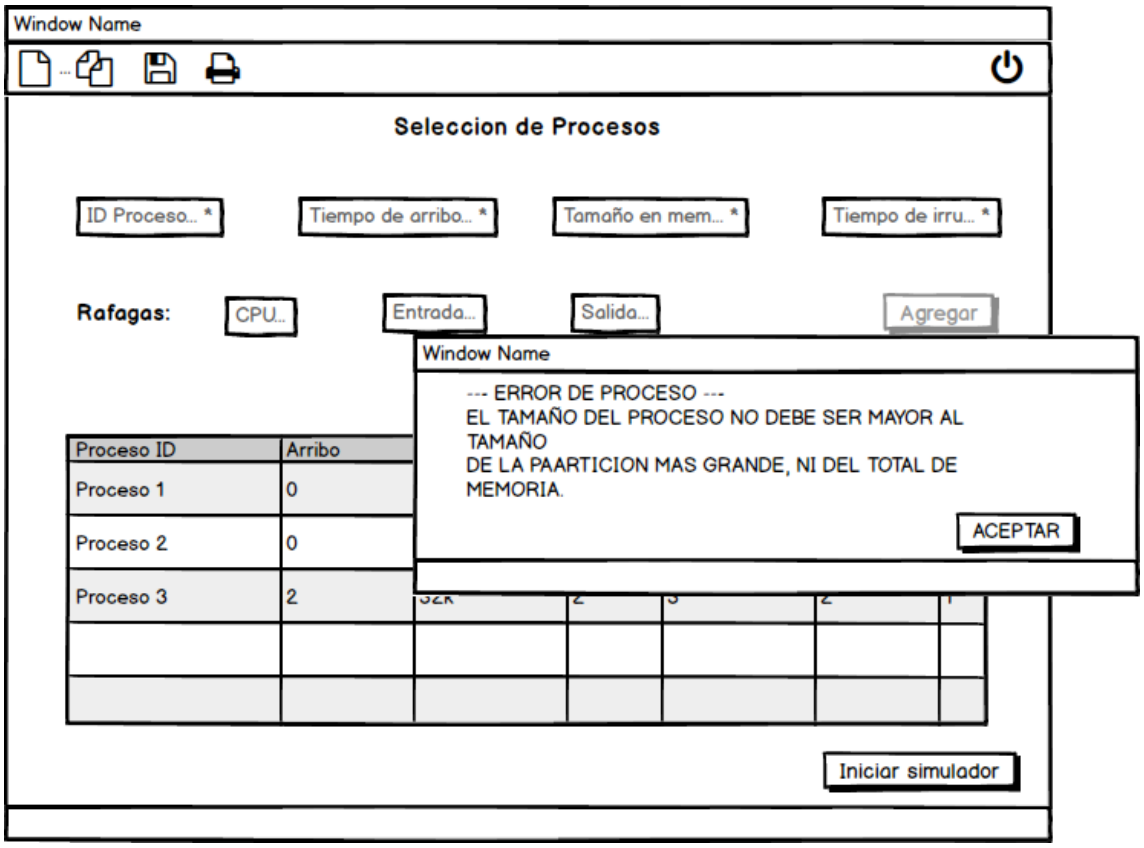
Los datos podrán ser ingresados en los cuadros de entrada y luego de ser agregados se podrán visualizar en un cuadro donde se listarán todos los procesos cargados por el usuario con la información referida a cada uno.



Dependiendo del tipo de algoritmo de planificación que se decidió utilizar van a variar los datos de entrada que se pida por cada proceso y el cuadro de listado de procesos, ya que datos como la prioridad y el quantum son inherentes a un tipo de algoritmo en particular.



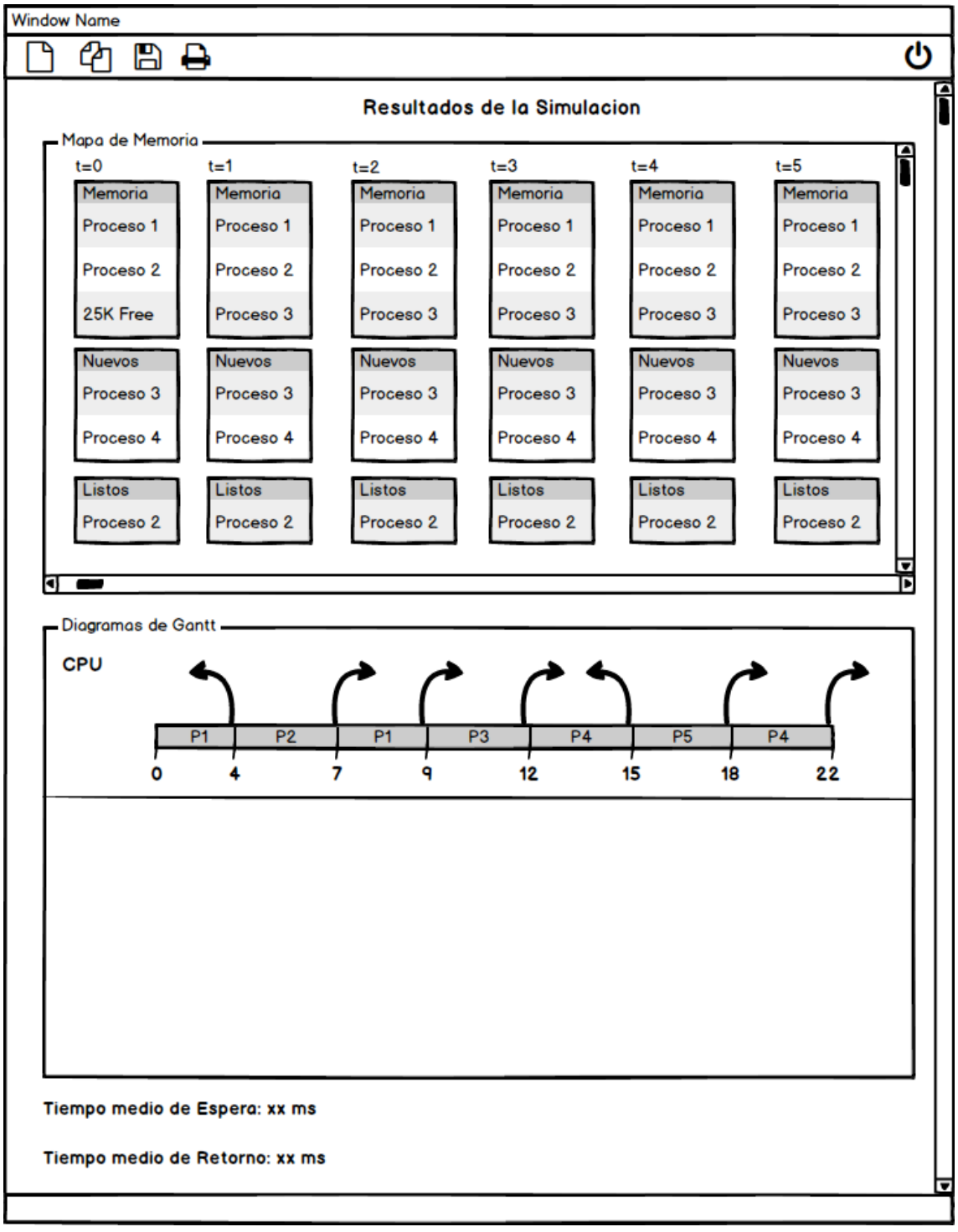
En el caso de querer agregar un proceso de un tamaño mayor al de la memoria, aparecerá otra ventana indicando el error cometido, restricción detallada en la sección “Restricciones de entrada”.



Una vez que se cargaron todos los procesos con los que se desea trabajar, se podrá comprobar los resultados de la simulación haciendo clic en el botón “iniciar simulador”.

Luego de iniciar el simulador aparecerá en una nueva ventana el detalle de cómo fueron tratados los procesos con la configuración seleccionada. En dicha ventana podremos ver por cada tiempo t=n, cuál era el estado de la memoria y de la cpu en lo que al tratamiento de procesos se refiere. Se podrá visualizar las listas de procesos nuevos y procesos listos, los mapas de memoria a través del tiempo y el diagrama de gantt de la cpu.

En el caso de que se haya necesitado una cantidad considerable de unidades de tiempo para el procesamiento por parte del simulador, las salidas respectivas al mapa de memoria y las colas de procesos se mostrarán en una diagrama que sobrepasa las dimensiones de la ventana por lo que solo se podrá de ver forma completa a través de una barra de desplazamiento.


Window Name
Resultados de la Simulacion
Mapa de Memoria
t=0
t=1
t=2
t=3
t=4
t=5
Memoria
Proceso 1
Proceso 2
25K Free
Proceso 3
Nuevos
Proceso 3
Proceso 4
Listos
Proceso 2
Diagramas de Gantt
CPU
P1
P2
P1
P3
P4
P5
P4
0
4
7
9
12
15
18
22
Tiempo medio de Espera: xx ms
Tiempo medio de Retorno: xx ms

Herramientas de desarrollo y Librerías externas

El lenguaje de programación elegido por el grupo para realizar el simulador es Python 3, para poder ejecutar el programa se deberá tener instalado el intérprete de Python, preferentemente la última actualización, la misma se podrá descargar a partir del siguiente enlace

https://www.python.org/downloads/

Para el desarrollo y diseño de las interfaces del simulador utilizamos la aplicación "QTDesigner" y también la librería de Python "PyQt5" con el que gestionamos la entrada y la salida de los datos a partir de las interfaces gráficas. No es necesario instalar la aplicación QTDesigner para ejecutar el programa, pero si se debe instalar la librería PyQt5 por ser una librería externa al lenguaje. Esta librería se puede instalar desde windows ejecutando el siguiente comando desde el símbolo del sistema

pip install PyQt5

Para linux se puede instalar la librería desde la terminal con el mismo comando que en windows, con la diferencia que antes de instalar una librería debemos instalar el complemento de instalación de paquetes de python (pip) previamente, ya que este no suele estar incluido en la instalación de python. Para hacerlo se ejecuta el siguiente comando desde la terminal

sudo apt install python-pip

Para gestionar y visualizar el gráfico de gantt, utilizamos la librería "matplotlib", al igual que PyQt5 se debe instalar antes de poder usarla, la cual nos permite mostrar de forma gráfica las salidas del simulador. La instalación de la liberaría se realiza de forma similar al caso anterior

pip install matplotlib

Algoritmos Preliminares de Asignación de Memoria

Desarrollamos a continuación los códigos correspondientes a los algoritmos de asignación de memoria que utilizaremos para el simulador. Como detallamos en sección de "Entradas del Simulador" y en la "Navegación de pantallas", los algoritmos que se utilizaran son los siguientes:

- First Fit (para particiones fijas y variables)
- Best Fit (para particiones fijas)
- Worst Fit (para particiones variables)

Por cuestiones de comodidad de los integrantes los mismos fueron escritos en Python.

Con el objetivo de centrarse en el funcionamiento de cada algoritmo y no complejizar el entendimiento de los mismos, en el planteo de los algoritmos se omitieron los aspectos relacionados al procesamiento de los procesos una vez que son almacenados, por lo tanto los algoritmos solo tratan el almacenamiento de los procesos hasta que se llene la memoria.

Best Fit (Partición Fija):

En este algoritmo por cada proceso de la lista, recorre todas las particiones y por cada una verifica si su tamaño es suficiente para almacenar el proceso y si está desocupado por cada partición que elige resguarda su longitud. Finalmente el proceso se almacena en la particion mas grande.

```
def BF():
    for i in range(0, len(procesos)):
        cond=False
        ant=1025
        tam=procesos[i][3]
        for x in range(0, len(mem_fija)):
            if mem_fija[x][0]<ant and mem_fija[x][0]>=tam and mem_fija[x][1]==0:
                ant=mem_fija[x][0]
                pos=x
                cond=True
        if cond==True:
            mem_fija[pos][1]=procesos[i][0]
```

Worst Fit (Partición Variable):

Este algoritmo, por cada proceso de la lista, recorre todas las particiones y verifica si la misma tiene la longitud suficiente y está desocupada, si es así resguarda la partición, y gracias a la variable ant se queda con la partición de mayor longitud. Al final divide la partición elegida y agrega una nueva a la lista de particiones cuya longitud será el fragmento de memoria que no ocupo el proceso actual.

```
def WF():
    for i in range(0, len(procesos)):
        cond=False
        ant=0
        tam=procesos[i][3]
        for x in range(0, len(mem_var)):
            if mem_var[x][0]>ant and mem_var[x][0]>=tam and mem_var[x][1]==0:
                ant=mem_var[x][0]
                pos=x
                cond=True
        if cond==True:
            mem_var[pos][1]=procesos[i][0]
            temp=mem_var[pos][0]-tam
            mem_var[pos][0]=tam
            mem_var.append([temp,0])
```

First Fit (Partición Fija o Variable):

El algoritmo de asignación de memoria First Fit recibe como parámetro el tipo de Partición a utilizar (Fija o Variable) y por medio de un alternativo selecciona el sub algoritmo a ejecutar. En el sub algoritmo para Particiones Fijas se le asigna el tamaño del proceso a una variable temporal, y a la variable cond (condición) “False”. Con un Ciclo for se compara uno a uno el tamaño de los espacios libres en memoria hasta poder asignar el proceso. Cuando esto sucede la variable cond cambia su valor a “True”.
El sub algoritmo para Particiones Variables es de similar funcionamiento, con la diferencia que cuando encuentra un espacio libre en memoria con la capacidad de albergar al proceso, se subdivide generando dos espacios en memoria, uno el cual contiene propiamente al proceso, y otro que contiene espacio libre, el tamaño del espacio libre está definido por la diferencia entre: Tamaño_del_espacio_libre - Tamaño_del_proceso.

```
def FF(tipo):
    if tipo=='Fija':
```

```
        for i in range(0,len(procesos)):
                tam=procesos[i][3]
                cond=False
                for x in range(0,len(mem_fija)):
                        if mem_fija[x][0]>=tam and mem_fija[x][1]==0 and cond==False:
                                pos=x
                                mem_fija[pos][1]=procesos[i][0]
                                cond=True
elif tipo=='Variable':
        for i in range(0,len(procesos)):
                cond=False
                tam=procesos[i][3]
                for x in range(0,len(mem_var)):
                        if mem_var[x][0]>=tam and mem_var[x][1]==0 and cond==False:
                                pos=x
                                cond=True
                if cond==True:
                        mem_var[pos][1]=procesos[i][0]
                        temp=mem_var[pos][0]-tam
                        mem_var[pos][0]=tam
                        mem_var.append([temp,0])
```

Algoritmos Preliminares de Planificación de Procesos

Desarrollamos a continuación los códigos correspondientes a los algoritmos de planificación de procesos que utilizaremos para el simulador. Como detallamos en sección de "Entradas del Simulador" y en la "Navegación de pantallas", los algoritmos que se utilizaran son los siguientes:

- First Come First Served
- Round Robin
- Por prioridad
- Colas multinivel

Por cuestiones de comodidad de los integrantes los mismos fueron escritos en Python o Pseudocódigo. Con el objetivo de centrarse en el funcionamiento de cada algoritmo y no complejizar el entendimiento de los mismos, en el planteo de los algoritmos se omitieron los aspectos relacionados al almacenamiento de los procesos, suponemos que todos los procesos están listos para competir por el uso del procesador desde que llega su tiempo de arribo.

FCFS:

La función FCFS recibe una lista con instancias de la clase proceso, cada proceso tiene un identificador (atributo id_proceso), tiempo de arribo (atributo arribo) y un tiempo de irrupción (atributo irrupcion).

La función ordena la lista de entrada según qué proceso llegó antes y devuelve una lista compuesta por los identificadores de cada proceso en el orden en el que estos fueron ejecutados según el momento en que fueron cargados. Para ello recorre la lista de procesos y por cada ciclo for (que simboliza cada tiempo de cpu) agrega a la lista de salida que proceso se ejecuta, o nulo si no hay ningún proceso en la cola de listos en ese momento. Como indica la técnica fcfs, una vez que el algoritmo encuentra el proceso que se debe ejecutar actualmente no libera la cpu hasta que este este no haya finalizado.

```
def fcfs(procesos):
        procesos=sorted(procesos, key=lambda objeto: objeto.arribo)
        gantt=list()
        tiempo=1
        for p in procesos:
                while True:
                        if tiempo>=p.arribo:
                                for j in range(p.irrupcion):
                                        gantt.append([tiempo , p.id_proceso])
                                        tiempo=tiempo+1
                                p.irrupcion=0
                                break
                        else:
```

```
                gantt.append([tiempo , None])
                tiempo=tiempo+1
    return gantt
```

Round Robin:

La función Round Robin recibe una lista de procesos al igual que el algoritmo FCFS, pero con la diferencia de que este algoritmo también debe recibir el quantum de tiempo a través de una variable entera. Además en cada proceso también será necesario representar la condición de finalizado o no a través de un atributo extra (condición).

Este algoritmo al igual que el anterior ordena la lista de procesos para procesar primero los que llegan antes. Luego recorre los procesos, una vez que llego su arribo lo agrega en la lista gantt tantas veces hasta que llegue al quantum o se termine el tiempo de irrupción, lo que ocurra primero, luego pasa al próximo proceso. Los procesos que terminan se marcan como finalizado, de forma tal que el ciclo principal terminará cuando todos los procesos hayan finalizado.

```
def round_robin(procesos,quantum):
    procesos=sorted(procesos, key=lambda objeto: objeto.arribo)
    cant_proc=len(procesos)
    gantt=list()
    time=procesos[0].arribo
    while (cant_proc!=0):
        for p in procesos:
            if (time>=p.arribo):
                if (p.condicion=="ready"):
                    if (p.irrupcion<=quantum):
                        for j in range(p.irrupcion):
                            gantt.append([time,p.id_proceso])
                            time+=1
                            p.irrupcion-=1
                        cant_proc-=1
                        p.condicion="finished"
                    else:
                        for k in range(quantum):
                            gantt.append([time,p.id_proceso])
                            time+=1
                        p.irrupcion=p.irrupcion-quantum
            else:
                gantt.append([time,None])
                time+=1
    return gantt
```

Por Prioridad:

La función genera de forma aleatoria una lista de procesos que almacena en la lista “colap”. Luego recorre esta lista, agregando a la lista “ejecucion”, hasta que este se termine de ejecutar o hasta que llegue un proceso con mayor valor de prioridad. Cuando otro proceso de mayor prioridad arriba, se quita el que estaba en la lista de ejecución y se agrega al proceso nuevo. Los procesos que ya se ejecutaron se almacenan en la lista “finalizado” y se eliminan de la cola de listos (colap). En el caso de que ningún proceso haya llegado al tiempo de cpu actual solo se incrementa el tiempo.

```
def PorPrioridad():
        cantProcesos = random.randint(1, 10)
        colaP = [] ; ejecucion = [] ; finalizado = []
        time = 0
        i=1
        while cantProcesos >= 0:
                proceso = []
                tiempoLlegada = random.randint(0,10)
                timeCpu = random.randint(1, 8)
                prioridad = random.randint(1, 5)
                proceso.append('P'+str(i))
                proceso.append(tiempoLlegada)
                proceso.append(timeCpu)
                proceso.append(prioridad)
                colaP.append(proceso)
                cantProcesos = cantProcesos - 1
                i+=1
        print('esto es colaP ', colaP)
        colaP.sort(key=lambda colaP:colaP[1])
        print('esto es colaP ordenado por tiempo de Llegada ', colaP)
        resPrioridad = 0 ; resTimeNext = 0 ; resIndice = 0
        flag = True
        while flag:
                for i in range(0, len(colaP)):
                        a = i + 1
                        if colaP:
                                resTimeNext = colaP[a][1]
                                print(resTimeNext)
                                if colaP[i][1] == time:
                                        resIndice = i
                                        print(resIndice)
                                        print(colaP[i])
                                        print('siguiente tiempo de llegada del ‘,end=’’)
                                        print(‘siguiente proceso ',resTimeNext)
                                        if resPrioridad == 0:
                                                resPrioridad = colaP[i][3]
                                                ejecucion.append(colaP[i])
                                                print('esto fue pasado a ejecución ', colaP[i])
                                                colaP.pop(i)
```

```
                    print('esto esta en ejecucion ', ejecucion)
                    ejecucion[0][2] = ejecucion[0][2] - 1
                    print('ejecutando ',ejecucion)
                elif colaP[i][3] < resPrioridad:
                    resIndice = i
                    colaP.append(ejecucion[0])
                    ejecucion.clear()
                    ejecucion.append(colaP[i])
                    ejecucion[0][2] = ejecucion[0][2] - 1
                else:
                    if ejecucion[0][2] == 0:
                        finalizado.append(ejecucion)
                        colaP.pop(i)
                        ejecucion.pop(int(ejecucion[0]))
                    else:
                        ejecucion[0][2]  = ejecucion[0][2] - 1
                time+=1
            else:
                while time != resTimeNext:
                    time +=1
                    print(time)
                    print(ejecucion)
                    if ejecucion:
                        if ejecucion[0][2] == 0:
                            finalizado.append(ejecucion)
                            colaP.pop(i)
                            ejecucion.pop(0)
                            resPrioridad = 0
                        else:
                            ejecucion[0][2]  -= 1
        else:
            flag = False
print('esto está en la cola ',colaP)
print('esto está finalizado ',finalizado)
print('esto esta en ejecucion ',ejecucion)
```

Colas Multinivel:

El algoritmo Colas Multinivel, recibe como parámetros los Arreglos (A1, A2 y A3) que contienen a los procesos y los algoritmos de planificación que se utilizaran en cada cola (AlgoritmoCola1, AlgoritmoCola2 y AlgoritmoCola3). Teniendo como mayor prioridad a la cola n° 1, luego la cola n° 2 y por último a la cola n° 3.
Contiene condicionales alternativos Si / Sino encadenados los cuales controlan si el Arreglo de la cola N a tratar contiene procesos, si la lista está vacía devuelve "error" y no se ejecuta, y en el caso de que contenga elementos se trata con el algoritmo especificado.
Las salidas de cada cola serán las establecidas previamente en el algoritmo correspondiente a la misma.
El algoritmo está planteado para tratar 3 colas:

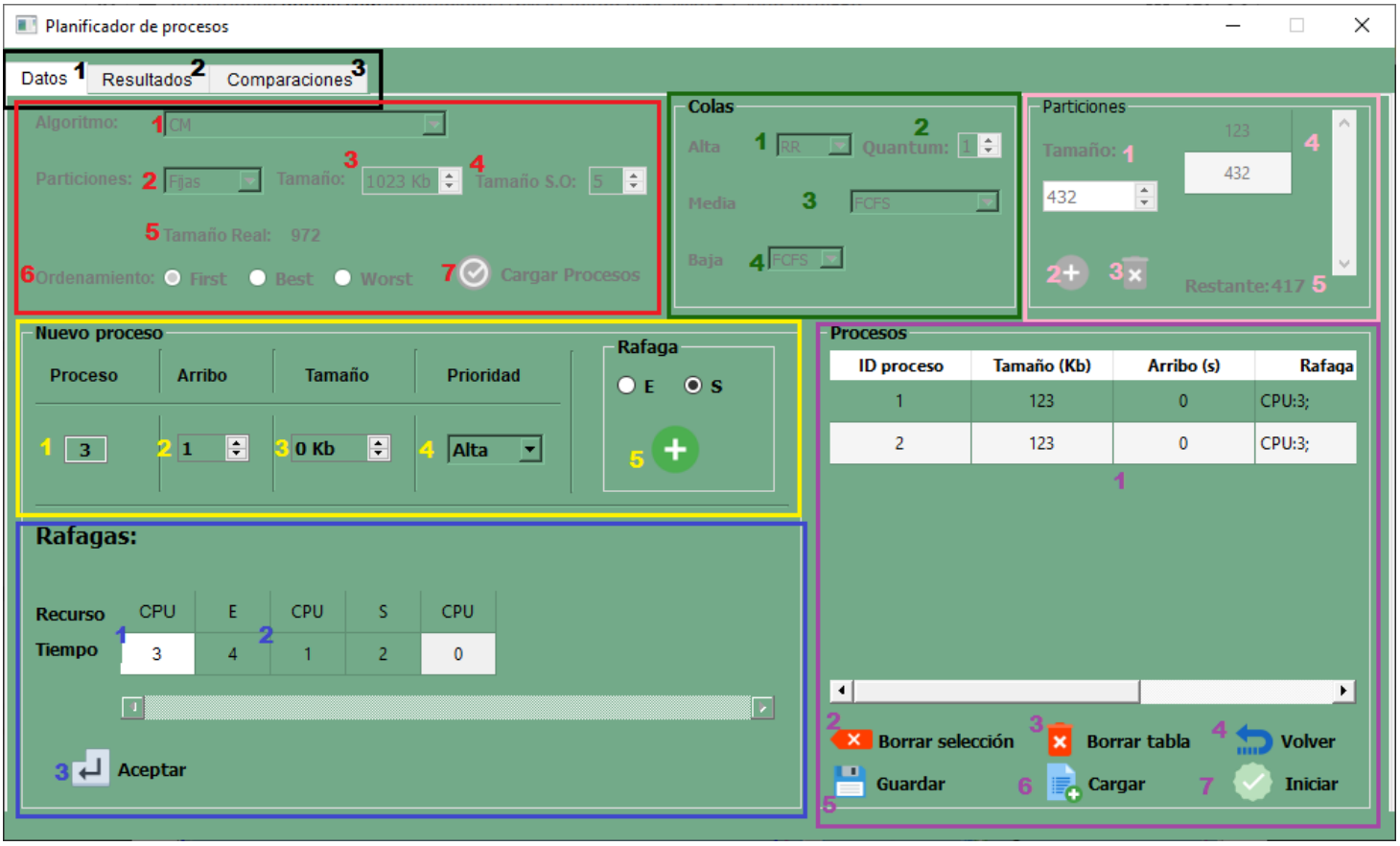
```
def colas_mlvl(A1, A2, A3, AlgoritmoCola1, AlgoritmoCola2, AlgoritmoCola3):
        if len(A1)==0:
                print('error: el arreglo A1 no contiene procesos')
        else:
                print('se empezará a tratar la cola 1, con el algoritmo:', AlgoritmoCola1)
                if AlgoritmoCola1 == 1:
                        FCFS(A1)
                elif AlgoritmoCola1 == 2:
                        round_robin(A1)
                elif AlgoritmoCola1 == 3:
                        porPrioridad(A1)
                else:
                        SJF(A1)
        print('el simulador terminó con los procesos de la cola 1')
        if len(A2)==0:
                print('error: El arreglo A2 no contiene procesos')
        else:
                print('se empezará a tratar la cola 2, con el algoritmo:', AlgoritmoCola2)
                if AlgoritmoCola2 == 1:
                        FCFS(A2)
                elif AlgoritmoCola2 == 2:
                        round_robin(A2)
                elif AlgoritmoCola2 == 3:
                        porPrioridad(A2)
                else:
                        SJF(A2)
        print('el simulador terminó con los procesos de la cola 2')

        if len(A3)==0:
                print('error: el arreglo A3 no contiene procesos')
        else:
                print('se empezará a tratar la cola 3, con el algoritmo: ',AlgoritmoCola3)
                if AlgoritmoCola3 == 1:
                        FCFS(A3)
                elif AlgoritmoCola3 == 2:
```

```
                round_robin(A3)
        elif AlgoritmoCola3 == 3:
                porPrioridad(A3)
        else:
                SJF(A3)
print('el simulador terminó con los procesos de la cola 3')
```

Funcionamiento de la Interfaz Gráfica

-Pantalla principal (entrada de datos):



Recuadro Negro:

1- Es la ventana Principal y donde se seleccionan los datos que serán Simulados.

2- Es la ventana donde se podrán ver los resultados de la simulación una vez que está finalice.

3-En esta ventana se puede visualizar la comparación de hasta 3 simulaciones ejecutados.

Recuadro Rojo:

1- Selecciona el Algoritmo a utilizar, el cual puede ser FCFS, SJF, SRTF, Round Robin o Colas Multinivel.
2- Permite seleccionar el tipo de partición de memoria, la cual puede ser FIJA o VARIABLE.
3- Establece el Tamaño(en KB) que poseerá la memoria total.
4- Permite seleccionar el porcentaje de memoria que se reservara para el SO.
5- Muestra el tamaño real de memoria disponible para la asignación de procesos.
6- Permite elegir el método ordenamiento deseado. En particiones variables puede ser "First Fit" y "Worst Fit", mientras que en particiones fijas se puede seleccionar "First Fit" y "Best Fit".
7- Botón para confirmar la selección de los procesos realizada, esto bloquea la carga de datos y habilita la carga de procesos.

Recuadro Verde:

*Este recuadro se habilita solo cuando el algoritmo seleccionado es **CM(Colas Multinivel)**. Dentro de este, se puede seleccionar los algoritmos FCFS, Round Robin, SJF y SRTF.

1- Selecciona el algoritmo que será utilizado para la cola de **mayor prioridad**.
2- En este caso particular, al haber sido seleccionado el algoritmo RR(Round Robin), nos permite elegir el valor del Quantum que deseamos para esta Cola.
3- Nos permite seleccionar el algoritmo de **prioridad media**.
4- En este cuadro, podemos seleccionar el algoritmo de **prioridad baja** o de **menor prioridad**.

Recuadro Rosa:

Este recuadro solo aparece habilitado cuando la Partición seleccionada es **FIJA**, ya que en "particiones variables" no se debe establecer la cantidad de particiones, ni el tamaño de cada Partición.

1- Permite definir el tamaño(en KB) de cada partición fija.
2- Este botón agrega una partición con el tamaño establecido previamente.
3- Este botón elimina una partición previamente seleccionada.
4- Lista de particiones cargadas.
5- Espacio restante para completar la memoria.

Recuadro Amarillo:

1- ID de Proceso que deseamos cargamos.
2- Tiempo de Arribo del proceso.
3- Establece el tamaño del proceso(En el caso de particiones fijas, este no puede ser mayor que la partición de más grande).
4- "Prioridad" se habilita solo en el algoritmo CM(Colas Multinivel) y nos permite elegir a qué Cola se enviará el proceso(Alta, Media o Baja prioridad).
5- Permite agregar Rafagas de "Entrada" o "Salida".

Recuadro Azul:

1- Define el tiempo en CPU(Tiempo de Irrupción) que utilizará el proceso.
2- Establece el tiempo de la Rafaga de Entrada y el tiempo de Irrupción posterior a la Rafaga.
3- Boton que permite confirmar la carga de un proceso, si no se cargan los valores de rafaga, no se cargara.

Recuadro Violeta:

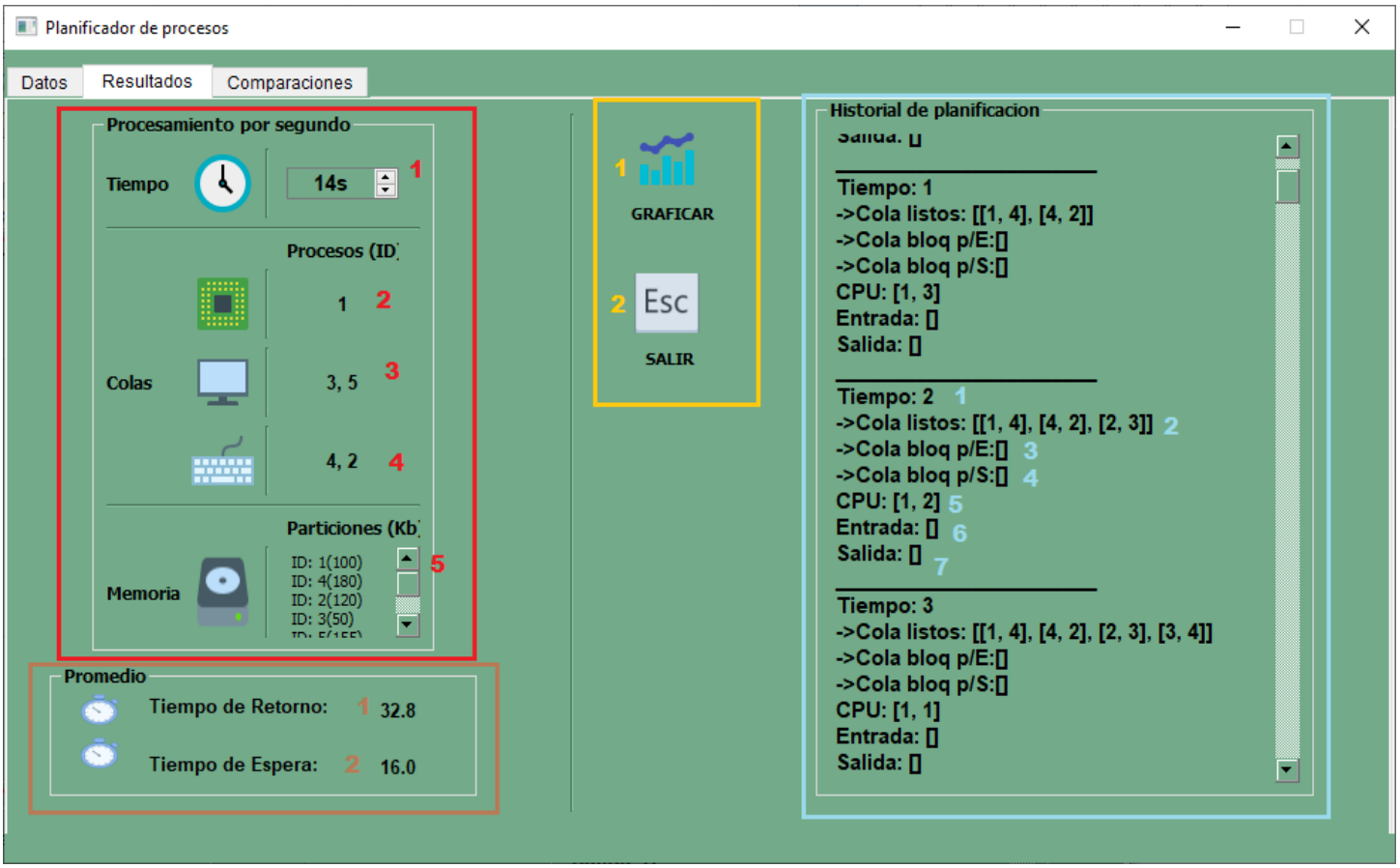
1- Lista de procesos cargados.
2- Permite borrar el procesos seleccionado.
3- Borra la tabla entera de procesos.
4- Permite regresar a la parte de selección de memoria.

5- Permite guardar los procesos cargados en un documento .txt para su posterior utilización.

6- Permite cargar procesos almacenados en un archivo externo .txt.

7- Inicia la simulación de la ejecución de los procesos procesos.

-Pantalla de Resultados (salida de datos):



Recuadro Rojo:

1- Permite seleccionar un tiempo determinado para visualizar las colas de listos y de bloqueados para E/S.
2- Muestra los procesos que se encuentran en la cola de listos en el tiempo seleccionado.
3- Muestra los procesos que se encuentran en la cola de bloqueados para Salida en el tiempo seleccionado.
4- Muestra los procesos que se encuentran en la cola de bloqueados para Entrada en el tiempo seleccionado.
5- Permite visualizar el estado de las particiones de memoria en el tiempo seleccionado.

Recuadro Marrón:

1- Muestra el tiempo de retorno promedio.
2- Muestra el tiempo de espera promedio.

Recuadro Celeste:

1- Referencia el tiempo de ejecución por unidad.
2- Muestra el estado de la cola de listos en un tiempo determinado.
3- Muestra el estado de la cola de bloqueados para Entrada en un tiempo determinado.
4- Muestra el estado de la cola de bloqueados para Salida en un tiempo determinado.
5- Muestra los procesos que se están ejecutando en CPU en el tiempo determinado.
6- Muestra los procesos que se están ejecutando en el disp. de Entrada en el tiempo determinado.
7- Muestra los procesos que se están ejecutando en el disp. de Salida en el tiempo determinado.

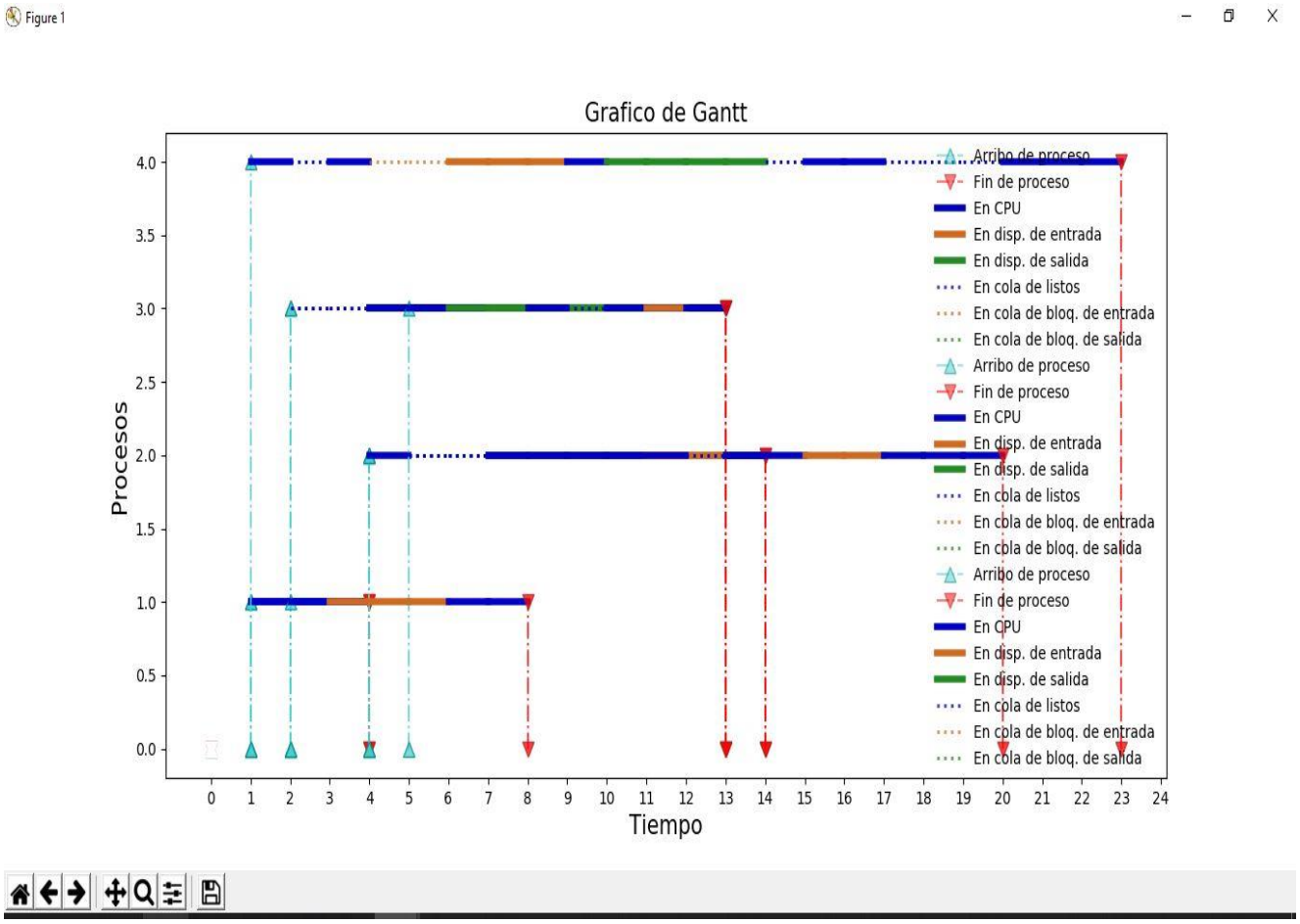
Recuadro Naranja:

1- Esta opción permite graficar el diagrama de Gantt, en base a los elementos obtenidos previamente por el Simulador.
2- Seleccionando “Esc” podremos cerrar el Simulador, es una alternativa a la “X” en la parte superior derecha de la pantalla.

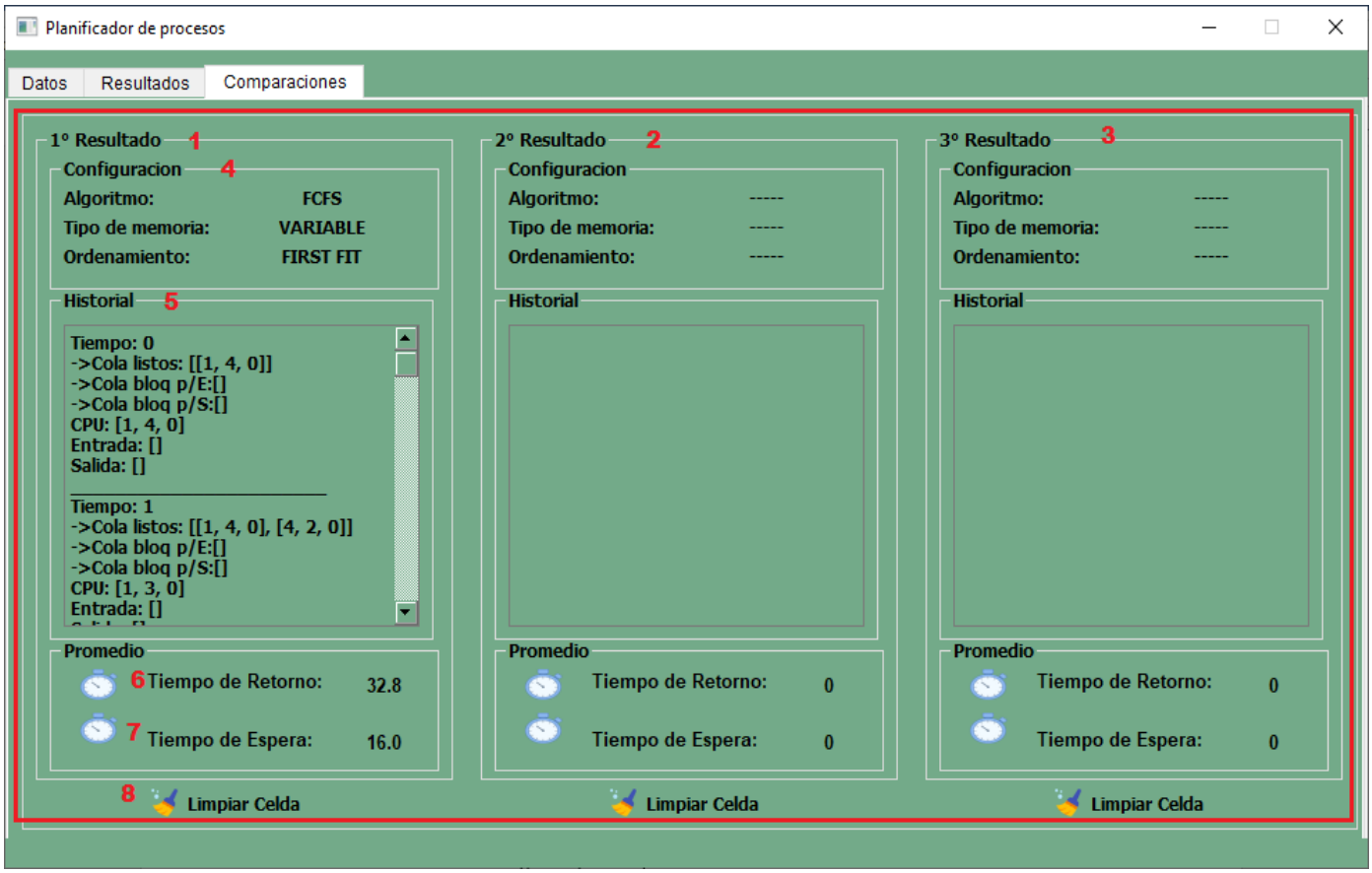
Si seleccionamos la opción 1, Lo que obtendremos será el gráfico expuesto a continuación.

Gráfico de Gantt

Este es el diagrama de Gantt creado por el simulador, a la derecha del mismo se encuentran nombrados cada uno de los elementos que se encuentran en este gráfico. Hemos elegido representar en el mismo gráfico los procesos que se ejecutaron en cada tiempo, pero también los que realizaron operaciones de entrada y salida y también que proceso formó parte de que cola.



-Pantalla de Comparaciones:



Recuadro Rojo:

1- Muestra los datos del primer proceso/simulación ejecutada.

2- Muestra los datos del segundo proceso/simulación.

3- Muestra los datos del tercer proceso/simulación (este se sobrescribirá si se siguen añadiendo ejecuciones).

4- Muestra la configuración seleccionada para ese proceso/simulación (mostrando el algoritmo seleccionado, el tipo de memoria, y el tipo de ordenamientos.)

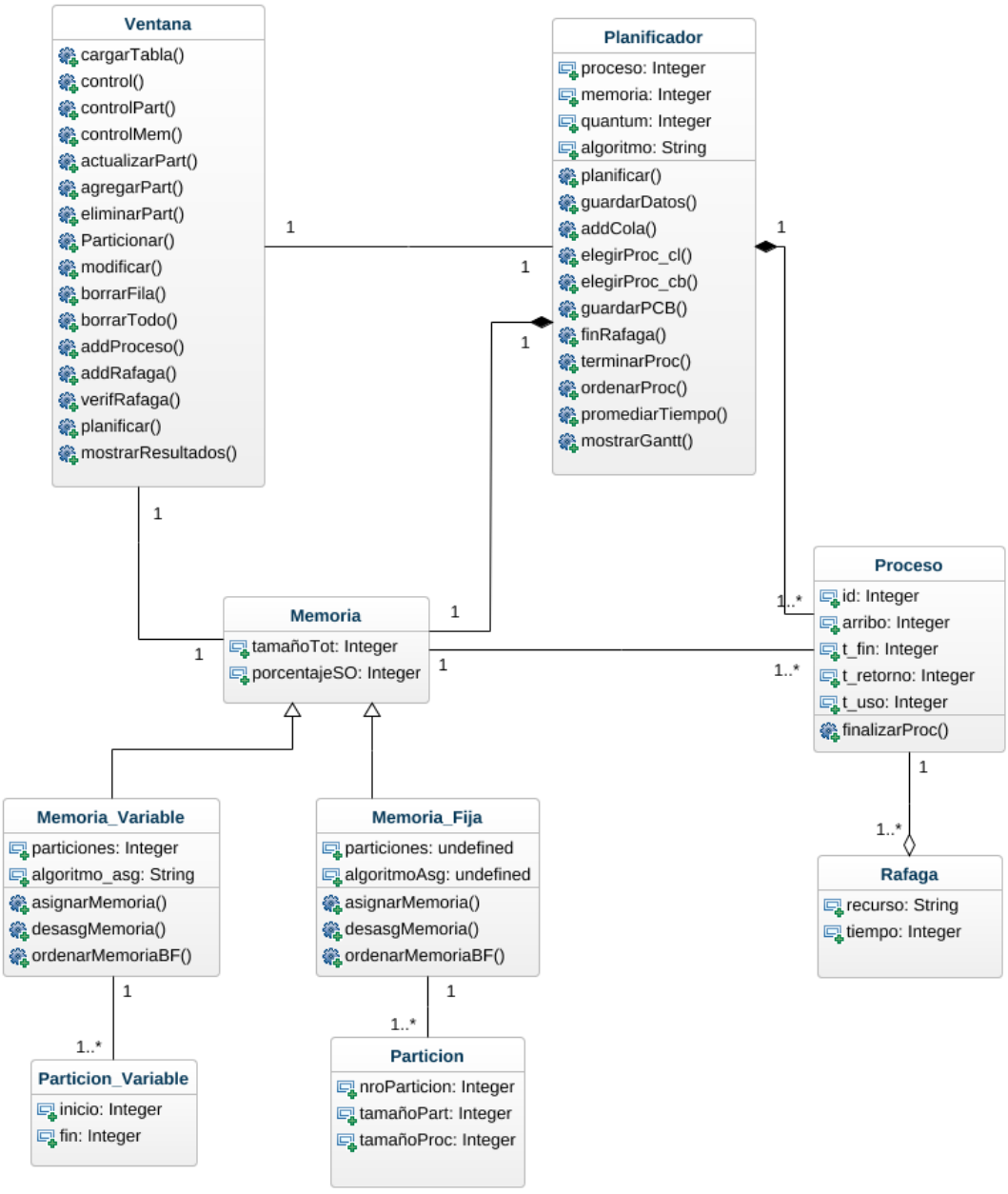
5- Muestra el gráfico correspondiente al "mapa de memoria" en cada unidad de tiempo (esto se muestra en cada columna de cada simulación).

6- Muestra el tiempo medio de retorno de la simulación correspondiente**.**

7- Muestra el tiempo medio de espera de la simulación correspondiente**.**

8- Botón que sirve para limpiar toda la información de los cuadros de configuración, historial y promedio para poder seguir comparando procesos/simulaciones.

Módulos Internos del programa



Funcionamiento interno del Programa

El tratamiento de interfaces de usuario, es decir el ingreso de datos de entrada y la visualización de los datos de salida, es operado por la clase Ventana, que es el módulo del programa que se encarga de interactuar directamente con el usuario.

El módulo principal del simulador es la clase Planificador, ésta recibe los datos ingresados por la clase ventana o a través de un documento de texto si se trata de datos ya ingresados anteriormente. Esta clase tiene la tarea principal de configurar el simulador a partir de los datos ingresados y de llevar a cabo la lógica de almacenamiento y procesamiento de los procesos, interactuando con casi todas las demás clases.

Cada proceso es visto por el simulador como una instancia de la clase Proceso, la cual almacenará todas sus propiedades en forma de atributo excepto las ráfagas de entrada, salida y CPU, donde cada ráfaga será representada como una instancia de la clase Ráfaga.

Los datos de la memoria son almacenados en la clase Memoria, que a su vez posee dos subclases llamadas Memoria_Fija y Memoria_Variable que se encargan de gestionar la asignación y desasignación de los procesos respetando la lógica de trabajo de su respectiva clasificación. El objeto memoria tendrá almacenado el tipo de algoritmo que posee y Las particiones que son representadas por la Clase Partición_Fija y Partición_Variable.

Índice de Clases y Métodos

Este índice ha sido confeccionado con la única finalidad de ser una manual de ayuda para que el usuario pueda hallar rápidamente en el código del Simulador, la Clase y/o método que desea.

-Método control_q
-Método actualizar_part
-Metodo agregar_part
-Método eliminar_part
-Método particionar
-Método modificar
-Método borrar
-Método borrar_todo
-Método planificar
-Método graficar_gantt
-Método imprimir_promedios
-Método impirimir_resultado2
-Método imprimir_resultado
-Método print_resultados
-Método quantum_Alta
-Método quantum_media
-Método clickEvent
-Método add_proceso
-Método verificar_rafagas
-Método limpiar_raf
-Método limpiar_part
-Método agregar_rafaga

Clase Planificador **0733 - 1066**

-Método __init__
-Método planificar
-Método guardar_datos
-Método add_cola
-Método elegir_proc_cl
-Método elegir_proc_cb
-Método guardar_pbc
-Método fin_rafaga
-Método terminar_proc
-Método ordenar_proc
-Método promediar_t
-Método print_gant
-Método print_gant2

Clase Proceso **1068 - 1094**

-Método __init__
-Método finalizar

Clase Ráfaga **1096 - 1099**

-Método __init__

Clase Particion_Variable **1101 - 1108**

-Método __init__

Clase Memoria_variable **1110 - 1229**

-Método __init__
-Método asignarMemoria
-Método desasignarMemoria
-Método ordenarMemoriaBF

Clase Particion **1231 -1240**

-Método __init__

Clase Memoria_fija **1242 - 1268**

-Método __init__
-Método asignarMemoria
-Método desasignarMemoria
-Método ordenarMemoriaBF

Detalle de los Métodos y Clases más Importantes

Esta sección pretende ser un manual de autoayuda para el lector que no esté muy acostumbrado al lenguaje Python, pero aun así busque profundizar en el funcionamiento del Simulador desarrollado.
Se dará una breve explicación de los Métodos y Clases más importantes en el funcionamiento del Simulador.
Queda totalmente a discreción del usuario/lector decidir si necesita leer esta sección.

clase ventana:

```
def planificar(self):
        self.tab_widget.setCurrentIndex(1)
        if self.control():
            nue_p=list()
            for i in range(self.tab_procesos.rowCount()):
                p=int(self.tab_procesos.item(i,0).text())
                a=int(self.tab_procesos.item(i,2).text())
                tam=int(self.tab_procesos.item(i,1).text())
                rafaga=self.tab_procesos.item(i,3).text()
                nue_p.append(proceso(p,a,tam,rafaga))
            if self.BF.isChecked():
                ordenamiento='BF'
            elif self.WF.isChecked():
```

Recorre las filas de la tabla de procesos y tomando los datos de cada columna crea un objeto proceso

Si esta seleccionado "rrq" y/o particiones "fijas" verifica que el campo de quantum y de particiones no este vecio o en cero

agrega cada proceso a la lista "nue_p"

```
                ordenamiento='WF'
            else:
            ordenamiento='FF'
            if self.particion.currentText()=='Variables':

mem=Memoria_variable(self.tam_memoria.value(),ordenamiento)
            elif self.particion.currentText()=='Fijas':
                mem=Memoria_fija(self.tam_memoria.value(),ordenamiento)
                for i in range(self.particiones.rowCount()):
                    h=int(self.particiones.item(i,0).text())
                    mem.particiones.append(Particion(i+1,h))
            if self.algoritmo.currentIndex()==0:
                algoritmo='fifo'
            elif self.algoritmo.currentIndex()==1:
                algoritmo='sjf'
            elif self.algoritmo.currentIndex()==2:
                algoritmo='srtf'
            else:
                algoritmo='rrq'
                q=self.quantum.value()
            plan=planificador(algoritmo,nue_p, mem,self.quantum.value())
            resultado=plan.planificar()
            plan.promediar_t()
            self.imprimir_resultado(resultado[0])
            self.imprimir_promedios(plan.ret_prom,plan.esp_prom)
            self.gantt=resultado[1]
            self.datos=plan.datos_proc
            self.box_tiempo.setEnabled(True)
            self.box_tiempo.setMaximum(len(self.datos)-1)
            self.imprimir_resultado2()
```

Si se selecciono memoria variable, instancia un objeto de la clase memoria_variable, con el tipo de ordenamiento y tamaño indicado

Si selecciono memoria fija ademas de instanciar la clase memoria_fija, crea un objeto particion y lo agrega a la lista "particiones" por cada fila de la tabla de particiones de la ventana

Verifica qué algoritmo fue seleccionado y con ese dato junto con la lita de procesos, quantum y memoria crea el objeto planificador

Llama al metodo planificar de la clase planificador el cual realiza el proceso de simulacion

Guarda el gantt recibido del resultado de la simulacion e imprime por pantalla los resultados

clase planificador:

```
def __init__(self,algoritmo,tab,memoria,quantum):
self.proc=tab
self.memoria=memoria
self.cola=list()
self.alg=algoritmo
self.q=''
if self.alg=='rrq':
        self.q=quantum
self.interrupcion=False
self.cpu=list()
self.ent=list()
self.sal=list()
```

El atributo proc guarda la lista de procesos que todavia no han iniciado su ejecucion

Self.memoria guarda un objeto de tipo memoria con las configuraciones elegidas por el usuario

Los atriburos algoritmo y q guardan un string ('rr', 'sjf', etc) y un entero (el quantum) respectivamente

En cada tiempo t, el proceso que se esta ejecutando en cada dispositivo se guardara el self.cpu, self.ent y self.sal

```
self.cola_bloq_e=list()
self.cola_bloq_s=list()
self.cola_list=list()
self.gant_cpu=list()
self.gant_e=list()
self.gant_s=list()
self.dat_proc=list()
self.ret_prom=0
        self.esp_prom=0
self.datos_proc=list()
self.ordenar_proc()
```

Habra un atributo de listo por cada cola de listos tanto para cpu, como entrada y salida

Habra un atributo de listo por cada cola de listos tanto para cpu, como entrada y salida

def planificar(self):

Se configura el gantt agregandole los titulos, el cuadro de referencia y los ejes

```
plt.title('Grafico de Gantt',size=15)
plt.xlabel('Tiempo',size=15)
plt.ylabel('Procesos',size=15)
plt.xticks(list(range(0,80)))
plt.plot([0,0],[0,0],marker="^",ls="-.",color="turquoise" ,label="Arribo de proceso")
plt.plot([0,0],[0,0],marker="v",ls="-.",color="red",mec="firebrick",label="Fin de proceso")
plt.plot([0,0],[0,0],marker="^",ls="-.",color="white",ms=8,alpha=1.0)
plt.plot([0,0],[0,0],marker="v",ls="-.",color="white",ms=8,alpha=1.0)
plt.plot([0,0],[0,0],color="mediumblue",lw=4,label="En CPU")
plt.plot([0,0],[0,0],color="chocolate",lw=4,label="En disp. de entrada")
plt.plot([0,0],[0,0],color="forestgreen",lw=4,label="En disp. de salida")
plt.plot([0,0],[0,0],ls=":",color="mediumblue",lw=2,alpha=0.75,label="En cola de listos")
plt.plot([0,0],[0,0],ls=":",color="chocolate",lw=2,label="En cola de bloq. de entrada")
plt.plot([0,0],[0,0],ls=":",color="forestgreen",lw=2,label="En cola de bloq. de salida")
t=0
quantum=self.q
exit=False
resultado_icono=list()
if self.memoria.algAsig=="BF":
        self.memoria.ordenarMemoriaBF()
resultado=''
while len(self.proc)>0 or not exit:
        resultado=resultado+'Tiempo: '+str(t)+'\n'
        self.add_cola(t)
        resultado=resultado+'->Cola listos: '+str(self.cola_list)+'\n'
        if (len(self.cpu)==0 or quantum==0 or self.interrupcion):
                if (quantum==0 or self.interrupcion):
                        if (quantum==0):
                                quantum=self.q
                                resultado=resultado+'#Q!'+'\n'
                                if (len(self.cpu)!=0):
                                        self.guardar_pbc()
                        if (self.interrupcion):
                                resultado=resultado+'#P!'+'\n'
```

Cada tiempo sera representado por t

Si el algoritmo de asignacion es “bf” es preferible que la memoria este ordenada por tamaño de menor a mayor

Recorre el ciclo mientras haya procesos en self.proc

Pasa de “proc” a “cola” los que estan listos para ejecutarse

En la variable resultado va acumulando la informacion que luego mostrara en el historial de planificacion

Si se alcanzo el quantum, la cpu esta vacia o llego otro proceso y se necesita verificar la prioridad, debe meter un nuevo proceso en la cpu

Si entra porque se acabo el quantum, lo escribe en resultado, restablece el quantum y resguarda el proceso que pierde la cpu

```
                        self.guardar_pbc()
                        self.interrupcion=False
                if (len(self.cola_list)>0):
                        self.elegir_proc_cl(self.alg)
                else:
                        self.gant_cpu.append([t,0])
        resultado=resultado+'->Cola bloq p/E:'+str(self.cola_bloq_e)+'\n'
        if (len(self.ent)==0):
                if (len(self.cola_bloq_e)>0):
                        self.elegir_proc_cb('e')
                else:
                        self.gant_e.append([t,0])
        resultado=resultado+'->Cola bloq p/S:'+str(self.cola_bloq_s)+'\n'
        if (len(self.sal)==0):
                if (len(self.cola_bloq_s)>0):
                        self.elegir_proc_cb('s')
                else:
                        self.gant_s.append([t,0])
        self.guardar_datos()
        resultado=resultado+'CPU: '+str(self.cpu)+'\n'
        if (len(self.cpu)>0):
                plt.plot([t,t+1],[self.cpu[0],self.cpu[0]],color="mediumblue",lw=4)
                if (self.alg=='rrq'):
                        quantum=quantum-1
                self.cpu[1]=(self.cpu[1])-1
                self.gant_cpu.append([t,self.cpu[0]])
                if (self.cpu[1]<1):
                        if (self.alg=='rrq'):
                                quantum=self.q
                        self.fin_rafaga(self.cpu,'cpu',t)
                        self.terminar_proc(t+1)
                        self.cpu.clear()
        resultado=resultado+'Entrada: '+str(self.ent)+'\n'
        if (len(self.ent)>0):
                plt.plot([t,t+1],[self.ent[0],self.ent[0]],color="chocolate",lw=4)
                self.ent[1]=(self.ent[1])-1
                self.gant_e.append([t,self.ent[0]])
                if (self.ent[1]<1):
                        self.fin_rafaga(self.ent,'e',t)
                        self.ent.clear()


        resultado=resultado+'Salida: '+str(self.sal)+'\n'
        if (len(self.sal)>0):
                plt.plot([t,t+1],[self.sal[0],self.sal[0]],color="forestgreen",lw=4)
                self.sal[1]=(self.sal[1])-1
                self.gant_s.append([t,self.sal[0]])
                if (self.sal[1]<1):
                        self.fin_rafaga(self.sal,'s',t)
                        self.sal.clear()
        exit=True
        for i in self.cola:
                if i.tfin==0:
```

Si entra por interrupcion tambien lo guarda en resultado y resguarda el proceso y vuelve a poner a cero a self.interrupcion

Hay que agregar un proceso a la cpu, si hay elementos en la cola de listos elige uno, sino agrega un cero a la lista de gantt

Al igual que para la cpu, verifica si la ent y sal estan vacias, si es asi verifica en la cola de bloqueados si hay un proceso para agregarlo

Grafica la linea del proceso que esta en cpu

Entra si hay un proceso en cpu, es decir hay uno ejecutandose en el tiempo actual

Decuenta el tiempo de irrupcion del proceso y si el algoritmo es rr, descuenta el quantum

Si tiempo de irrupcion del proceso en cpu llego a 0, termina su rafaga, lo quita de la cpu y verifica si finalizo el proceso llamando a terminar_proc

Repite la ultima accion hecha para “cpu”, en “ent” y ”sal”

Recorre la cola general, y busca un proceso que no haya terminado. Si no lo encuentra exit queda en true

```
                    exit=False
                    break
            resultado=resultado+'_____'+'\n'
            if len(self.cola_list)>0:
                for j in range(len(self.cola_list)):
                    plt.plot([t,t+1],[self.cola_list[j][0],self.cola_list[j][0]],":",color="blue")
            if len(self.cola_bloq_e)>0:
                    for j in range(len(self.cola_bloq_e)):

                        plt.plot([t,t+1],[self.cola_bloq_e[j][0],self.cola_bloq_e[j][0]],"
                        :")
            if len(self.cola_bloq_s)>0:
                    for j in range(len(self.cola_bloq_s)):

                        plt.plot([t,t+1],[self.cola_bloq_s[j][0],self.cola_bloq_s[j][0]],"
                        :")
            t=t+1
        resultado=resultado+'Tiempo: '+str(t) +'\n'
            resultado=resultado +'->Cola listos: '+str(self.cola_list) +'\n'
            resultado=resultado +'CPU: '+str(self.cpu)+'\n'
            resultado=resultado +'->Cola bloq p/E:'+str(self.cola_bloq_e)+ '\n'
            resultado=resultado +'Entrada: '+str(self.ent)+'\n'
            resultado=resultado +'->Cola bloq p/S:'+str(self.cola_bloq_s)+'\n'
            resultado=resultado +'Salida: '+str(self.sal)+'\n'
        resultado=resultado+'_____'+'\n'
        self.guardar_datos()
        plt.legend(loc="best",framealpha=0.0)
        return [resultado, plt]
```

Grafica las colas en el gantt

Agrega el ultimo ciclo del historial de planificacion al acumulador de resultados

El metodo devuelve cadena "resultado" y el grafico

```
def add_cola(self, tact):

        delete=list()
        for i in range(len(self.proc)):
            if(self.proc[i].arribo<=tact):
                if self.memoria.asignarMemoria(self.proc[i].id,self.proc[i].tam)==True:
                    nuevo=self.proc[i]
                    self.cola.append(nuevo)
                    plt.plot([tact,tact],[0,self.proc[i].id],"^-",color="turquoise")
                    delete.append(self.proc[i])
        for i in delete:
            if (i in self.proc):
                self.proc.remove(i)
        for i in self.cola:
            if i.tfin==0:
                while i.rafagas[i.ejec].tiempo==0:
                    i.ejec=i.ejec+1
                if (i.rafagas[i.ejec].recurso)=='cpu':
                    exist=False
```

Recibe el tiempo actual y agrega los procesos a las

Recorre todos los procesos (proc) y a aquellos a los que llego su tiempo de arribo y pudieron ser almacenados en la memoria, grafica su llegada y lo almacena en la cola general

Elimina de la lista de procesos (proc) los procesos que fueron almacenados en memoria

Recorre la cola general y por cada proceso no finalizado se posiciona en su rafaga actual

Copia los procesos de la cola general a la de listos si es que ya no formaba parte de la misma

```
                for j in self.cola_list:
                        if (j[0]==i.id):
                                exist=True
                                break
                if not exist:
                        self.cola_list.append([i.id,i.rafagas[i.ejec].tiempo])
                        if (len(self.cpu)>0 and self.alg=='srtf'):
                                self.interrupcion=True
```

Si entro un nuevo proceso a la cola de listos y se usa “srtf” se debe tratar la interrupcion ponendo este atributo en true

Lo mismo que hizo con la cola de listos hace con la cola de bloqueados de entrada y salida

```
        elif (i.rafagas[i.ejec].recurso)=='e':
                exist=False
                for j in self.cola_bloq_e:
                        if (j[0]==i.id):
                                exist=True
                                break
                if not exist:
                        self.cola_bloq_e.append([i.id,i.rafagas[i.ejec].tiempo])
        elif (i.rafagas[i.ejec].recurso)=='s':
                exist=False
                for j in self.cola_bloq_s:
                        if (j[0]==i.id):
                                exist=True
                                break
                if not exist:
                        self.cola_bloq_s.append([i.id,i.rafagas[i.ejec].tiempo])
```

```
def elegir_proc_cl(self,alg):
 if (alg=='sjf' or alg=='srtf'):
                tmin=self.cola_list[0][1]
                p=self.cola_list[0]
                for pr in self.cola_list:
                        if (pr[1]<tmin):
                                tmin=pr[1]
                                p=pr
                self.cpu=[p[0],p[1]]
                self.cola_list.remove(p)
                self.cola_list.insert(0,p)
 elif (alg=='fifo' or alg=='rrq'):
                p=self.cola_list[0]
                self.cpu=[p[0],p[1]]
```

Este metodo recibe un algoritmo de planificacion y elige un proceso de la cola de listos

Si el algortimo es “srtf” o “sjf” recorre la cola de listos y guarda el proceso de menor tamanño en pr

Una vez ordenado, selecciona el primer metodo y lo registra junto con su tiempo de irrupcion en el atributo self.cpu y lo agrega al principio de la cola

Si el algoritmo es “fifo” o “round robin” debe tomar el proceso que arribo primero por lo que toma el primer proceso de la cola y lo guarda en la lista cpu

```
def elegir_proc_cb(self, tipo):
 if (tipo=='e'):
                nue_proc=self.cola_bloq_e[0]
                self.ent=[nue_proc[0],nue_proc[1]]
 else:
                nue_proc=self.cola_bloq_s[0]
                self.sal=[nue_proc[0],nue_proc[1]]
```

Elegir_proc_cb elige un proceso de la cola de bolqueados (siempre el primero) de entrada o de salida dependiendo de el parametro “tipo”

```
def fin_rafaga(self,recurso,tipo,t):
        p=0
        while ((p<len(self.cola)) and (self.cola[p].id!=recurso[0])):
            p=p+1
        if (self.cola[p].id==recurso[0]):
            self.cola[p].ejec=self.cola[p].ejec+1
        if (tipo=='cpu'):
            del self.cola_list[0]
        elif (tipo=='e'):
            del self.cola_bloq_e[0]
        elif (tipo=='s'):
            del self.cola_bloq_s[0]
```

Recorre la lista self.cola hasta encontrar el proceso que llego por parametro (recurso) o hasta terminar la cola

Si lo encontro incrementa el contador de rafagas ejecutadas

Cuando finaliza una refaga este se debe quitar de la cola correspondiente al dispositivo (según el parametro "tipo") y como el proceso que se ejecuta siempre esta primero, elimina el elemento 0

```
def terminar_proc(self,t):
        x=0
        idp=self.cpu[0]
        while (x<len(self.cola) and idp!=self.cola[x].id):
            x=x+1
        if (idp==self.cola[x].id):
            if self.cola[x].ejec==len(self.cola[x].rafagas):
                self.memoria.desasignarMemoria(self.cpu[0])

                plt.plot([t,t],[0,self.cpu[0]],"v-.",color="red",mec="firebrick")
                self.cola[x].finalizar(t)
```

Recorre el atributo cola hasta en terminarla o encontrar al utlimo proceso que se ejecuto (el que se encuentra en la cola self.cpu)

Cuando sale del ciclo verifica si lo encontro y si a ese proceso ya no le quedan mas rafagas que ejecutar

Si los dos ultimos if son verdaderos significa que el proceso termino su cilco de vida, lo desaloja de la memoria y marca su fin en el gantt

Clase Proceso:

```
def __init__(self, idp, arribo, tam, rafagas):
        self.id=idp
        self.arribo=arribo
        self.tam=tam
        self.tuso=0
        rafagas=rafagas.split('; ')
        if '' in rafagas:
            rafagas.remove('')
        self.rafagas=list()
        for r in rafagas:
            r=r.split(': ')
            self.rafagas.append(rafaga(r[0].lower(),int(r[1])))
        for i in self.rafagas:
            self.tuso=self.tuso+i.tiempo
        self.tfin=0
        self.retorno=0
            self.espera=0
        self.ejec=0
```

Instancia la clase proceso por cada proceso que es cargado en memoria

Recibe como parámetros datos ingresados por pantalla

El atributo ráfagas se define como lista y guarda las rafagas (y su tipo) que se ingresaron junto con el tiempo de esta

Self.tuso es la sumatoria de todos los tiempos de irrupcion de rafaga

Este método se utiliza una vez por ejecución para calcular los tiempos de espera y retorno

```
def finalizar(self, t):
        self.tfin=t
        self.retorno=self.tfin-self.arribo
        self.espera=self.retorno-self.tuso
```

Recibe como parámetro t, que es tiempo total de ejecución

Clase Rafaga:

Se instancia por cada ráfaga de cada proceso, recibe los datos por parámetro

```
def __init__(self, r, t):
        self.recurso=r
        self.tiempo=t
```

Recurso refiere a que se utilizara (cpu o e/s)

clase Particion:

Habrá una instancia de la clase esta clase por cada particion que tenga la memoria. Guardara el tamaño, id y qué proceso lo tiene ocupado

```
def __init__(self,nro_particion,tamaño):
        self.nro_particion=nro_particion
        self.tamañoPart=tamaño
        self.tamañoP=0
        self.proceso_p=0
```

clase Memoria_fija:

Al crear la memoria fija se insanciara esta clase, en sus atributos se guardara una lista con sus particiones, su algoritmo y tamaño

```
def __init__(self,tamañoTot,algAsig):
        self.tamañoTot=tamañoTot
        self.particiones=[]
        self.algAsig=algAsig
```

```
def asignarMemoria(self,procesoId,tamProceso):
        for k in range(0,len(self.particiones)):
            if self.particiones[k].proceso_p ==0:
                if int(self.particiones[k].tamañoPart)>=tamProceso:
                    self.particiones[k].proceso_p=procesoId
                    self.particiones[k].tamañoP=tamProceso
                    return True
                    break
        return False
```

Recorre la lista de particiones

Por cada particion pregunta si esta vacio y su tamaño es menor al del proceso que recibe por parametro, si es asi lo marca como ocupdo por ese proceso

Si logra asignar memoria a ese proceso devuelve true, sIno false

```
def desasignarMemoria(self,procesoId):
    for i in range(0,len(self.particiones)):
        if self.particiones[i].proceso_p ==procesoId:
            self.particiones[i].proceso_p=0
            self.particiones[i].tamañoP=0
            break
```

Recorre la lista de particiones y por cada particion pregunta si el proceso que tiene almacenado es igual al que recibio por parametro, si es asi pone la particion en cero

clase Particion_variable:

```
def __init__(self,tamañoProceso):
    self.tamañoP=tamañoProceso
    self.inicio=0
    self.proceso_p=0
    self.fin=self.inicio+self.tamañoP
```

Por cada particion que se vaya generando en la memoria variable, se creara una instancia de este objeto , guarda lo mismo que la clase particion, incluyendo un limite inferior y superior (self.inicio y self.fin)

clase Memoria_variable:

```
def __init__(self,tamañoTot,algAsig):
    self.tamañoTot=tamañoTot
    self.particiones=[]
    self.algAsig=algAsig
```

La clase memoria variable tiene los mismos atriutos que memoria fija

```
def asignarMemoria(self,procesoId,tamP):
    if len(self.particiones)==0:
        x=Particion_variable(tamP)
        x.proceso_p=procesoId
        x.fin=tamP
        self.particiones.append(x)
        y=Particion_variable(self.tamañoTot-tamP)
```

Recibe un id proceso y su tamaño e intenta asiganrlo en alguna particion

Si la memoria no tiene particiones (esta vacia), crea dos, una que almacenara al proceso que recibe por parametro y la otra sera una particion vacia. Ambas las agrega a la lista de particiones

```
                y.proceso_p=0
                y.inicio=x.fin
                y.fin=self.tamañoTot
                self.particiones.append(y)
                return True
        else:
                if self.algAsig=="FF":
                        band=False
                        for i in range (0,len(self.particiones)):
                                if self.particiones[i].proceso_p==0:
                                        if self.particiones[i].tamañoP == tamP:
                                                self.particiones[i].proceso_p=procesoId
                                                return True
                                                break
                                        elif self.particiones[i].tamañoP > tamP:
                                                tamañoNuevo=self.particiones[i].tamañoP
                                                tamañoNuevo= tamañoNuevo- tamP
                                                z=Particion_variable(tamañoNuevo)
                                                self.particiones[i].tamañoP=tamP
                                                self.particiones[i].proceso_p=procesoId
                                                self.particiones[i].fin=self.particiones[i].inicio
                                                self.particiones[i].fin= self.particiones[i].fin+tamp
                                                z.proceso_p=0
                                                z.inicio=self.particiones[i].fin
                                                z.fin=z.inicio+z.tamañoP
                                                band=True
                                                pos=i+1
                                                break
                        if band==True:
                                self.particiones.insert(pos,z)
                                return True
                        else:
                                return False
                else:
                        mayor=0
                        max="null"
                        for i in range(0,len(self.particiones)):
                                if self.particiones[i].proceso_p==0:
                                        if self.particiones[i].tamañoP>=tam:
                                                if self.particiones[i].tamañoP>mayor:
                                                        mayor=self.particiones[i].tamañoP
                                                        max=i
                        if max!="null":
                                if self.particiones[max].tamañoP == tamP:
                                        self.particiones[max].proceso_p=procesoId
                                        return True
                                elif self.particiones[max].tamañoP > tamP:
                                        tamañoNuevo=self.particiones[max].tamañoP - tamP
                                        z=Particion_variable(tamañoNuevo)
                                        self.particiones[max].tamañoP=tamP
                                        self.particiones[max].proceso_p=procesoId
                                        self.particiones[max].fin=self.particiones[max].inicio
                                        self.particiones[max].fin + tamP
```

Si la cantidad de particiones no es cero, debe preguntar con que algorimto de asiganacion fue creada la memoria

Para First Fit recorremos las particiones hasta encontrar una particion cuyo tamaño sea mayor o igual al tamaño del proceso

Si encuentra un particion de igual tamaño que el proceso simplemete le asigna el id proceso a esa particion y retorna true

Pero si encuentra una paticion de tamaño mayor a la del proceso asigna el id del proceso y modifica tamaño y los limites, y crea otra particion con el espacio que sobraba de la particion original. Luego sale con un break sin antes resguardar el indice de la particion y la bandera que indica que el proceso fue amacenado

Si la bandera es verdadero almacena la particion vacia en la lista de particion en la posicion resguardada y retorna true, sino retorna false

Si entra en el else, significa que el algoritmo seleccionado es Worst Fit

Recorre todas las particiones, de todas las vacias guarda el tamaño de la mas grande en “mayor” y el indice de la misma en “max”

Si max es distinto de “null” significa que encontro al menos una particion

Al igual que para First Fit, debemos preguntar si la particion que encontro tiene tamaño igual o mayor al proceso, ya que si es igual lo asigna y sale, y si es mayor debe ajustar los limites de la particion y su tamaño y luego agregar la particion con el espacio que no se uso y retornar verdadero

```
                    z.proceso_p=0
                    z.inicio=self.particiones[max].fin
                    z.fin=z.inicio+z.tamañoP
                    band=True
                    pos=max+1
                    self.particiones.insert(pos,z)
                    return True
            else:
                return False

def desasignarMemoria(self,procesoId):
band1=False
band2=False
for i in range(0,len(self.particiones)):
        if self.particiones[i].proceso_p == procesoId:
            if i != 0 and i!= (len(self.particiones)-1):
                if self.particiones[i-1].proceso_p==0:
                    band1=True
                    pos1=i-1
                    self.particiones[i].inicio=self.particiones[i-1].inicio
                    ant= self.particiones[i-1].tamañoP
                    self.particiones[i].tamañoP=self.particiones[i].tamañoP+ant
                if self.particiones[i+1].proceso_p==0:
                    band2=True
                    pos2=i+1

                    ant= self.particiones[i+1].tamañoP
                    self.particiones[i].tamañoP=self.particiones[i].tamañoP+ant
                    self.particiones[i].fin=self.particiones[i+1].fin
                self.particiones[i].proceso_p=0
                break
            if i==0:
                if self.particiones[i+1].proceso_p==0:
                    band2=True
                    pos2=i+1
                    prox= self.particiones[i+1].tamañoP
                    self.particiones[i].tamañoP=self.particiones[i].tamañoP+prox
                    self.particiones[i].fin=self.particiones[i+1].fin
                self.particiones[i].proceso_p=0
                break
            if i==len(self.particiones)-1:
                if self.particiones[i-1].proceso_p==0:
                    band1=True
                    pos1=i-1
                    self.particiones[i].inicio=self.particiones[i-1].inicio

self.particiones[i].tamañoP=self.particiones[i].tamañoP+self.particiones[i-1].tamañoP
                    self.particiones[i].proceso_p=0
                    break
if band1==True:
        del(self.particiones[pos1])
```

Si no encontro ninguna particion devuelve falso

Recorre todas las particion es hasta encontra r la que almacena al proceso

Recibe el id del proceso e intentara desalojar este de la memoria

Las banderas serviran para comprobar si las particiones contiguas a la partcion donde se encuentra el proceso estan vacias o llenas

Una vez encontrada la particion, pregunta si esta es la primera, la ultima o esta en el medio. Necesita preguntar para saber si la particion tiene otras particiones contiguas

Luego desasigna el proceso de esa particion y agrega a la misma particion el limite inferior de la particion anterior y la superior de la particion siguiente si es que estos existen y el tamaño de los mismos si dichas particiones estan vacios. Guarda en “pos1” y “pos2” el indice de las particones contiguas si estas estan vacias

Si band1 es true significa que la particion anterior estaba vacia y si band2 es true lo mismo pero con la particion siguiente

```
            if band2==True:
                    del(self.particiones[pos2-1])
else:
            if band2==True:
                    del(self.particiones[pos2])
```

Elimina las particiones (con los indices pos1 y pos2) contiguas vacias ya que estas fueron absorvidas por la particion que desasigno al proceso