INFORME DE LABORATORIO III

SISTEMAS OPERATIVOS.

Docente: Oswaldo Romero.

- Carlos Alberto Hernández Córdoba – 20172020008.
- Diego Alejandro Vélez García – 20172020075.
- Maicol Andrés Garzón Fonseca – 20172020011.

1. OBJETIVOS:

Hacer la implementación (simulación gráfica) en el lenguaje de programación preferido de los siguientes modelos de memoria:

- Particiones estáticas fijas
- Particiones estáticas variables
- Particiones dinámicas (con y sin compactación)
- Segmentación
- Paginación

1. Tomar como base un sistema con 16 MiB de memoria principal (2^24 = 0x000000 - 0xFFFFFF)
2. Tener en cuenta los algoritmos de asignación (primer ajuste, mejor ajuste, peor ajuste).

2. MARCO TEÓRICO:

Particiones fijas y dinámicas

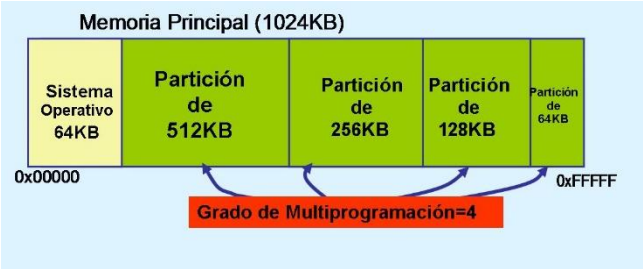
¿Que son Particiones?

Es el nombre genérico que recibe cada división presente en una sola unidad física de almacenamiento de datos. Toda partición tiene su propio sistema de archivos (formato); generalmente, casi cualquier sistema operativo interpreta, utiliza y manipula cada partición como un disco físico independiente, a pesar de que dichas particiones estén en un solo disco físico.

PARTICIONES FIJAS

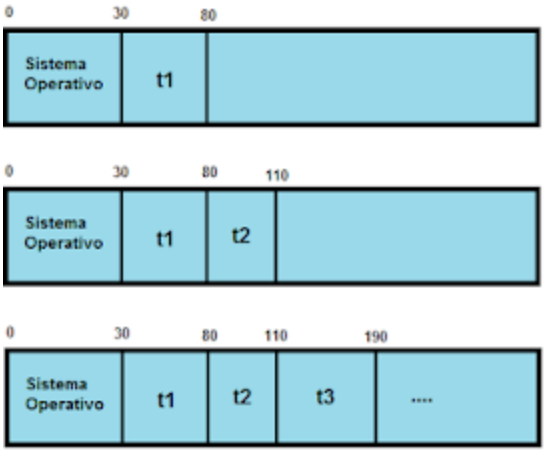
El primer intento para posibilitar la multiprogramación fue la creación de particiones fijas o estáticas, en la memoria principal, una partición para cada tarea. El tamaño de la partición se especificaba al encender el sistema, cada partición podía reconfigurarse al volver a encender el sistema o reiniciar el sistema.

Este esquema introdujo un factor esencial, la protección del espacio de memoria para la tarea. Una vez asignada una partición a una tarea, no se permitía que ninguna otra tarea entrara en sus fronteras. Este esquema de partición es más flexible que la de usuario único, porque permite que varios programas estén en memoria al mismo tiempo.



PARTICIONES DINÁMICAS

Las particiones dinámicas son variables en número y longitud, esto quiere decir que cuando se carga un proceso a memorial principal se le asigna el espacio que necesita en memoria y no más. Esta partición comienza siendo muy buena, pero en el trascurso de uso deja un gran número de huecos pequeños en la memoria lo cual se le denomina fragmentación externa. Se debe usar la compactación para evitar esta fragmentación, el sistema operativo desplaza los procesos para que estén contiguos de forma que todos los espacios de memoria libre se agrupen en un bloque, es decir:



t	tareas
t1	50
t2	30
t3	80
t4	10
t5	20

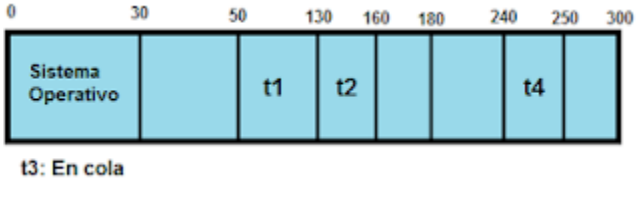
Al cabo de un tiempo las particiones dinámicas se empiezan a comportar como particiones fijas, así que en las particiones dinámicas también hay fragmentación, pero esta fragmentación es externa. Para esto existen tres algoritmos: mejor ajuste, primer ajuste o próximo ajuste

Mejor ajuste: Consiste en ubicar el proceso en el espacio de memoria que más se ajuste a su tamaño

Ejemplo: se quiere ingresar la tarea 3, pero si nos fijamos no hay una partición con un espacio lo suficientemente grande para esta tarea, por lo cual, al igual que las particiones fijas esta tarea quedaría en cola.

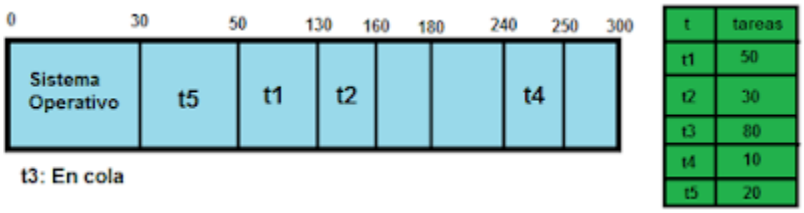
t	tareas
t1	50
t2	30
t3	80
t4	10
t5	20

Ahora vamos a ingresar la tarea 4 que pesa 10, entonces si miramos en la imagen la partición que más se ajusta al tamaño de la tarea es la de (240 a 250), ya que, si se asigna la tarea allí, no se va a gastar espacio de memoria entonces quedaría:



t	tareas
t1	50
t2	30
t3	80
t4	10
t5	20

Lo siguiente es ingresar la tarea 5 que pesa 20, pero hay dos particiones que están disponibles y son adecuados para esta tarea que son la partición de (30 a 50) y de (160 a 180), como el mejor ajuste dice que es la partición más se ajuste a su tamaño, por consiguiente, se podría asignar a cualquiera de los dos espacios, en esta ocasión lo pondremos en la partición de (30 a 50). Quedaría:



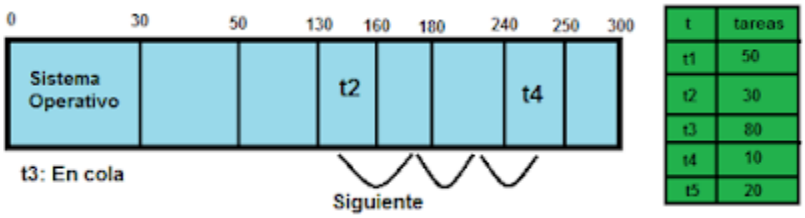
t	tareas
t1	50
t2	30
t3	80
t4	10
t5	20

Primer ajuste: Consiste en ubicar el proceso en el primer hueco disponible, recorriendo desde el inicio de la memoria, cuyo tamaño sea suficiente para el proceso.

Utilizaremos el ejemplo anterior para una mayor comprensión, vamos a ingresar la tarea 5 de nuevo, pero esta vez como es primer ajuste obligatoriamente debe de ir en la partición de (30 a 50) ya que el primer ajuste indica que se debe de asignar el proceso en la primera partición que sea más ajustada a su tamaño. Quedaría como la imagen anterior.

Siguiente ajuste: Consiste en ubicar el siguiente hueco disponible, que sea suficientemente grande, a partir de la última asignación de memoria.

Ejemplo: En este ejemplo asignamos la tarea 4, pero el punto de partida es la tarea 2 ya que fue la última en asignarse, por consiguiente, la siguiente casilla se asignará, pero en el siguiente ajuste hay que tener en cuenta, que también es la partición que más se ajuste al tamaño de la tarea, así este algoritmo pasara a la siguiente partición hasta que encuentre una partición del mismo tamaño que la tarea o un poco más grande para poder optimizar memoria. Quedaría:



t	tareas
t1	50
t2	30
t3	80
t4	10
t5	20

Des asignación en las particiones dinámicas:

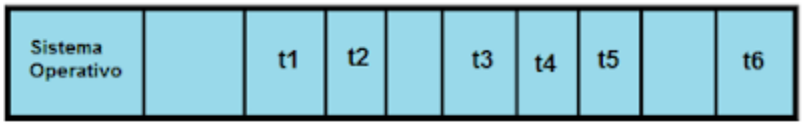
consiste en sacar las tareas de la memoria, las tareas salen por tiempo de ejecución, pero para que salgan de dan 3 casos:

Particiones ocupadas

Particiones entre una libre y una ocupada

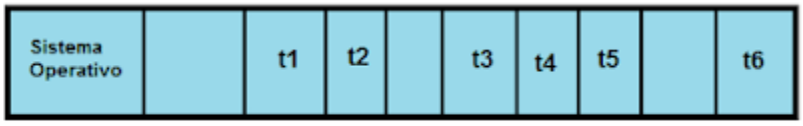
Particiones entre dos libres

Particiones ocupadas:

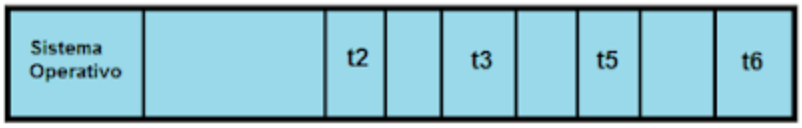


Vamos a eliminar la tarea 4, como se aprecia en la imagen esta entre dos particiones ocupadas, entonces el algoritmo lo que hace es preguntar si al lado izquierdo está ocupado y al lado derecho está ocupado, si es así entonces la tarea 4 se puede liberar y quedaría:

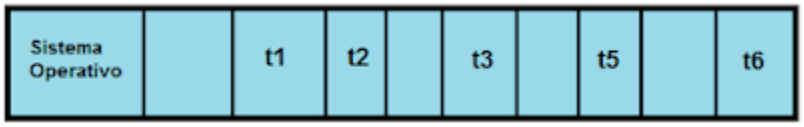
Particiones entre una libre y una ocupada:



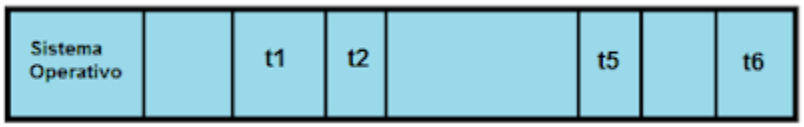
Vamos a eliminar la tarea 1, antes de eliminar el algoritmo pregunta si a la izquierda de la tarea 1 está libre y a la derecha está ocupado, o viceversa, si estas condiciones se cumplen entonces se puede liberar la tarea 1, pero además la partición que está libre se fusiona con la partición en la que se liberó la tarea quedando así una partición más grande, es decir:



Particiones entre dos libres:



En este ejemplo vamos a sacar la tarea 3 o termino de ejecutar y va a salir, entonces vemos que la tarea 3 esta entre dos particiones libres. El algoritmo pregunta si a la izquierda y a la derecha de la tarea 3 está libre, si es así, libera la tarea 3 y une las tres particiones para que quede una más grande, es decir:



RELOCALIZACIÓN

La relocalización solamente existe en las particiones dinámicas.

Con este esquema de asignación de memoria, el administrador de memoria relocaliza los programas para reunir los bloques vacíos y compactarlos, para hacer un bloque de memoria lo bastante grande para aceptar algunas o todas las tareas en espera de entrar

La relocalización sucede por 3 instancias:

Tiempo

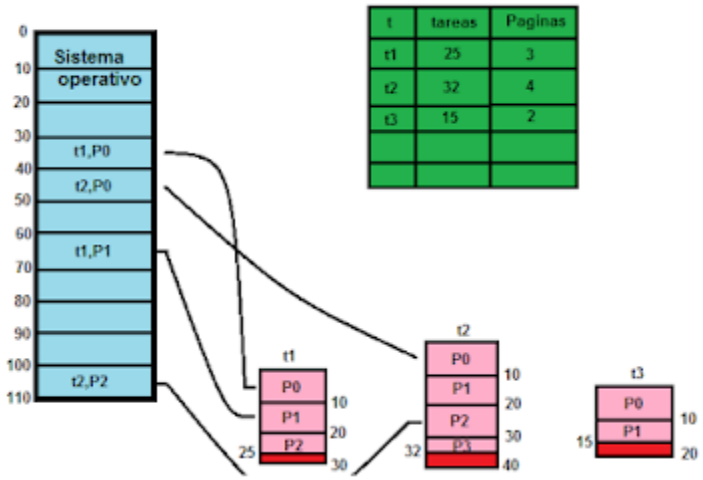
Cantidad de tareas en cola

Porcentaje de como este ocupada la memoria

PAGINACIÓN

La paginación consiste en dividir la memoria en un conjunto de marcos de igual tamaño, cada proceso se divide en una serie de páginas del tamaño de los marcos, incluso el espacio para el sistema operativo también esta paginado, un proceso se carga en los marcos que requiera (todas las paginas), no necesariamente contiguas es decir seguidas.

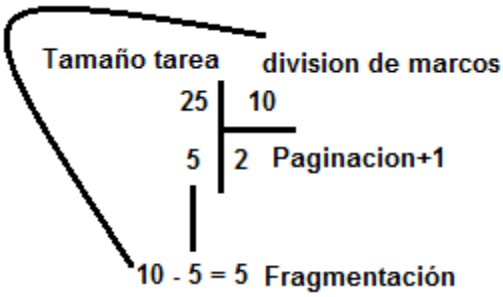
Ejemplo:



En la imagen podemos ver como se paginan los procesos y la memoria, al paginar los procesos se pueden llevar a la memoria aleatoriamente es así entonces como funciona la paginación.

nosotros podemos saber cuántas paginas salieron al dividir un proceso, y también la fragmentación a partir de una sencilla división y una resta

ejemplo:



Se divide el tamaño de la tarea entre la división de los marcos, el (cociente + 1) va a ser la paginación, después para saber la fragmentación se toma la división de marcos y se le resta el residuo (10 - 5 = 5), esta será la fragmentación.

De esta forma se puede cargar una página de información en cualquier marco de página. Las páginas sirven como unidad de almacenamiento de información y transferencia en la memoria principal y memoria secundaria

Las páginas de un programa necesitan estar contiguamente en memoria.

Los mecanismos de paginación permiten la correspondencia correcta entre las direcciones virtuales (dadas por los programas) y las direcciones reales de la memoria que se reverencien.

Cada página consiste en z palabras contiguas; un espacio de direcciones N de un programa consiste de n paginas (0,1,2,3…n-1) (n*z direcciones virtuales) y el espacio de memoria consiste de m marcos de páginas (0,z,2z,…,(m-1)z)(m*z posiciones).

PARA TENER EL CONTROL DE LAS PÁGINAS: Debe mantenerse una tabla en memoria que se denomina tabla de Mapas de Pagina (PMT) para cada uno de los procesos.

Tabla de páginas

Cada página tiene un número que se utiliza como índice en la tabla de páginas, lo que da por resultado el número del marco correspondiente a esa página virtual. Si el bit presente / ausente es 0, se provoca un señalamiento (trap) hacia el sistema operativo. Si el bit es 1, el número de marco que aparece en la tabla de páginas se copia en los bits de mayor orden del registro de salida. La finalidad de las tablas es asociar las páginas virtuales con los marcos.

Características de la paginación

• El espacio de direcciones lógico de un proceso puede ser no contiguo.

• Se divide la memoria física en bloques de tamaño fijo llamados marcos (frames).

• Se divide la memoria en bloques de tamaño llamados páginas.

• Se mantiene información en los marcos libres.

• Para correr un programa de n páginas de tamaño, se necesitan encontrara n marcos y cargar el programa.

• Se establece una tabla de páginas para trasladar las direcciones lógicas a físicas.

• Se produce fragmentación interna.

La dirección generada por la CPU se divide en:

Numero de página (p): utilizado como índice en la tabla de páginas que contiene la dirección base de cada página en la memoria física.

De la página (d): combinado con la dirección base define la dirección física que será enviada a la unidad de memoria.

Existen dos funciones

1- Llevar a cabo la transformación de una dirección virtual a física, o sea, la determinación de la página a la que corresponde una determinada dirección de un programa.

2- Transferir, cuando haga falta, páginas de la memoria secundaria a la memoria principal, y de la memoria principal a la memoria secundaria cuando ya no sean necesarias.

SEGMENTACIÓN

Divide la memoria en segmentos, cada uno de los cuales tiene una longitud variable, que está definida intrínsecamente por el tamaño de ese segmento del programa.

Los elementos dentro de un segmento se identifican por su desplazamiento, esto con respecto al inicio del segmento.

Definición y aspectos generales

La segmentación de memoria es un esquema de manejo de memoria mediante el cual la estructura del programa refleja su división lógica. Llevándose a cabo una agrupación lógica de la información en bloques de tamaño variable denominados segmentos, cada uno de ellos tienen información lógica del programa:

Subrutina, arreglo, etc.

Después cada espacio de direcciones de programa consiste de una colección de segmentos, que generalmente reflejan la división lógica del programa, este sistema de gestión de memoria es utilizado en sistemas operativos avanzados, pero ya existían muestras de su actividad desde los S.O. UNIX y D.O.S

Objetivos alcanzados con la segmentación de memoria

Modularidad de programas: Cada rutina del programa puede ser un bloque sujeto a cambios y recopilaciones, sin afectar por ello al resto del programa.

Estructuras de datos de largo variable: Donde cada estructura tiene su propio tamaño y este puede variar.

(stack)

Protección: Se puede proteger los módulos del segmento contra accesos no autorizados.

Compartición: Dos o más procesos pueden ser un mismo segmento, bajo reglas de protección; aunque no sean propietarios de los mismos.

Enlace dinámico entre segmentos: Puede evitarse realizar todo el proceso de enlace antes de comenzar a ejecutar un programa. Los enlaces se establecerán solo cuando sea necesario.

La segmentación paginada ayuda al proceso de gestión de memoria, puede hacerse una combinación de segmentación y paginación para obtener las ventajas de ambas.

Cada segmento puede ser descrito por su propia tabla de páginas.

Los segmentos son usualmente múltiplos de páginas en tamaño, y no es necesario que todas las páginas se encuentren en memoria principal a la vez; además las páginas de un mismo segmento, aunque se encuentren contiguas en memoria virtual, no necesitan estarlo en memoria real.

La segmentación paginada tiene su propio esquema, las páginas de almacenamiento virtual, que son contiguas en este almacenamiento, no necesitan ser contiguas en el almacenamiento real.

El direccionamiento es tridimensional con una dirección de almacenamiento virtual "v=(s,p,d)". S= núm. segmento, P= num.pag. D= desplazamiento

Compartición de segmentos

En un sistema de segmentación, una vez que un segmento ha sido declarado como compartido, entonces las estructuras que lo integran pueden cambiar de tamaño. Dos procesos pueden compartir un segmento con solo tener entradas en sus tablas generales que apunten al mismo segmento del almacenamiento primario.

SEGMENTOS DE MEMORIA

El segmento de código (tiene como base el contenido del registro CS). En este segmento se encuentran las instrucciones que forman el programa. Para acceder a los datos contenidos en él, se usa el registro IP como desplazamiento.

El segmento de datos (que tiene como base el registro DS). Contiene los datos que utiliza el programa (variables, etc.) para acceder a los datos contenidos en él, se suele utilizar los registros SI y DI como desplazamiento.

El segmento de pila (con SS como base). En él se desarrolla la pila del programa, utilizada para almacén temporal de datos, llamadas a funciones, etc. Debe estar presente en todos los programas EXE de forma obligada. Se utiliza el registro SP para acceder a los datos de este segmento.

El segmento extra (con ES como base). Su uso es opcional, y en él se encuentra un segmento definido por el usuario y que, regularmente, contiene datos adicionales. Al igual que ocurre con el segmento de datos, para acceder a los datos contenidos en él, se suelen utilizar los registros SI y DI.

Algoritmos:

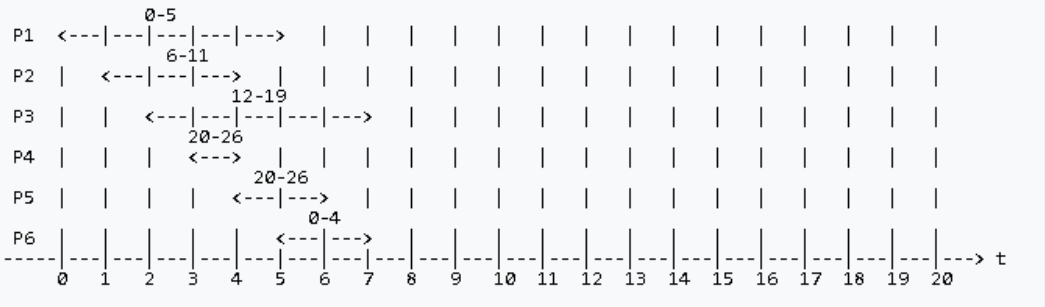
Primer ajuste

Consiste en asignar el primer hueco disponible que tenga un espacio suficiente para almacenar el programa. La principal desventaja es el reiterado uso de las primeras posiciones de memoria. Este último inconveniente repercute negativamente en la circuitería, debido a que se produce un mayor desgaste en dichas posiciones.

Ejemplo: Suponiendo una memoria principal de 32 KB.

	H0	t	M
P1	0	5	6
P2	1	3	6
P3	2	5	8
P4	3	1	7
P5	4	2	7
P6	5	2	5

Solución



Mejor ajuste

Consiste en asignarle al proceso el hueco con menor desperdicio interno, i.e, el hueco el cual al serle asignado el proceso deja menos espacio sin utilizar. Su mayor inconveniente es su orden de complejidad (orden lineal, O(n)) debido a que hay que recorrer todo el mapa de bits o toda la lista de control (una posible solución sería usar una lista de control encadenada que mantenga los huecos ordenados por tamaño creciente). Otro problema es la fragmentación externa, debido a que se asigna el menor hueco posible, el espacio sobrante será del menor tamaño posible lo que da lugar a huecos de tamaño normalmente insuficiente para contener programas.

Peor ajuste

Al contrario que el criterio anterior, se busca el hueco con mayor desperdicio interno, i.e, el hueco el cual al serle asignado el proceso deja más espacio sin utilizar, y se corta de él el trozo necesario (así la porción sobrante será del mayor tamaño posible y será más aprovechable). Tiene el mismo

inconveniente en cuanto a orden de complejidad que el mejor ajuste (debido a la longitud de las búsquedas) y la fragmentación no resulta demasiado eficiente.

5. Desarrollo del laboratorio

Programas y lenguajes usados:

- Python
- GitHub
- Visual Studio Code

Desarrollo del programa:

Crearemos 6 archivos donde haremos la implementación del código para la creación del proyecto donde veremos cada uno su estructura y que es lo que elaboran dentro del proyecto

Aplicación.py

Creamos nuestra aplicación en donde pondremos las funciones que crearan la ventana con la librería de tkinter que es para interfaz de usuario después creamos unos paneles

```
class Aplicacion:
    def __init__(self):
        self.init_ven_sel_gestores()


    def panel_nproceso(self):
        self.lf1=ttk.LabelFrame(self.ventana,text="Nuevo Proceso")
        self.lf1.grid(column=0, row=0, sticky="w")
        self.label1=ttk.Label(self.lf1, text="Tamaño:")
        self.label1.grid(column=0,row=0, padx=5, pady=5)
        self.dato1=tk.StringVar()
        self.entry1=ttk.Entry(self.lf1, textvariable=self.dato1)
        self.entry1.grid(column=1, row=0, padx=5, pady=5)
        self.boton1=ttk.Button(self.lf1, text="Agregar Proceso", command=self.nuevo_proceso)
        self.boton1.grid(column=0, row=3, columnspan=2, padx=5, pady=5, sticky="we")

    def panel_qproceso(self):
        self.lf2=ttk.LabelFrame(self.ventana,text="Quitar Proceso")
        self.lf2.grid(column=1, row=0, sticky="w")
        self.ipr = tk.IntVar()

    def act_panel_qproceso(self):
        for w in self.lf2.grid_slaves():
            w.destroy()

        for i in range(len(self.gestor.procesos)):
            if self.gestor.procesos[i][1] == True:
                cl = self.RAM.colores[(i+1)%len(self.RAM.colores)]
                tk.Radiobutton(self.lf2, text="Proceso "+str(i), bg=cl, variable=self.ipr, value=i).grid(column=i%10, row=1+floor(i/10), padx=5, pady=5, sticky="we")

        ttk.Button(self.lf2, text="Terminar Proceso", command=self.quitar_proceso).grid(column=0, row=3+floor(i/10), columnspan=2, padx=5, pady=5, sticky="we")

    def panel_inf(self):
        self.lf3=ttk.Labelframe(self.ventana, text="Informacion")
        self.lf3.grid(column=0, row=1, columnspan=2, sticky="w")
        self.lbl=ttk.Label(self.lf3, text="Informacion")
        self.lbl.grid(column=0, row=0, padx=5, pady=5, sticky="we")
```

Gestor.py

Creamos la clase gestor que contiene un arreglo de procesos y a partir de acá las clases de paginación y segmentación heredaran la información que requieran

```
class Gestor():
    def __init__(self, ram):
        self.procesos = []
        self.RAM = ram
        self.tam_ram = 0x1000000

    def nuevo_proceso(self, tam):
        pass

    def terminar_proceso(self, i):
        pass
```

Memoria.py

Aquí definiremos el canvas de la memoria donde pintamos el fondo de negro y pondremos un color a cada método que posteriormente se verá pintado el proceso proceso y las divisiones

```
class canvasRAM(Canvas):
    def __init__(self, master=None, h=20):
        super().__init__(master, bg='black', width=1280, height=h*16)

        self.tcr = 1024
        self.crX = ((int(self['width'])-self.tcr)/2)-1
        self.crY = 5
        self.hcr = h-self.crY

        for i in range(16):
            hc = i*(self.crY+self.hcr)+self.crY
            p0 = i*0x100000
            p1 = p0+0xFFFFF
            self.create_text(self.crX-35, hc+self.hcr/2, text=hex(p0), fill="white", font="consolas 10")
            self.create_text(35+self.crX+self.tcr, hc+self.hcr/2, text=hex(p1), fill="white", font="consolas 10")

            x1 = self.crX
            x2 = self.crX+self.tcr+1
            y1 = hc
            y2 = hc+self.hcr
            self.create_polygon(x1, y1, x2, y1, x2, y2, x1, y2, outline='white')

        #Lista de divisiones
        self.divisiones = []
        #Lista de colores
        self.colores = ['violet red', 'green', 'orange', 'cyan', 'pink', 'blue violet', 'orchid', 'lawn green']

    def pintar_division(self, pos, tam):
        tam = floor(tam/1024)
        pos = floor(pos/1024)

        self.divisiones.append([pos, tam])

        r = floor(pos/1024)
        x = self.crX+1+pos-(r*1024)
        y1 = r*(self.crY+self.hcr)+self.crY+(self.hcr/2)
        y2 = r*(self.crY+self.hcr)+self.crY+self.hcr+1
        self.create_line(x,y1,x,y2,fill='red')

        pos = pos + tam -1

        r = floor(pos/1024)
        x = self.crX+1+pos-(r*1024)
        y1 = r*(self.crY+self.hcr)+self.crY+(self.hcr/2)
        y2 = r*(self.crY+self.hcr)+self.crY+self.hcr+1
        self.create_line(x,y1,x,y2,fill='red')

    def pintar_proceso(self, i, tam, cl):
```

Segmentacion.py

Dentro del código añadiremos el gestor de segmentación incluyendo todas las funciones necesarias para su correcto funcionamiento

```python
from math import floor
from random import random, randint
from gestor import Gestor

class Segmentacion(Gestor):
    def __init__(self, ram):
        super().__init__(ram)

        self.segmentos = []
        self.espacios = [[0, self.tam_ram]]


    def nuevo_proceso(self, tam):
        proceso = []

        n_segmentos = randint(2,5)
        tam_ac = 0
        for i in range(n_segmentos):
            if i==n_segmentos-1:
                tam_seg = tam-tam_ac
            else:
                t = floor((tam-tam_ac)*randint(3,7)*(0.1))
                tam_seg = t-(t%1024)
                tam_ac += tam_seg

            #Mejor ajuste
            op = []
            for i in range(len(self.espacios)):
                op.append(self.espacios[i][1]-tam_seg)

            for i in range(len(op)):
                if op[i]<0:
                    op[i]=self.tam_ram+1
            i = op.index(min(op))
            if op[i]>self.tam_ram:
                return False
            pos = self.espacios[i][0]
            self.espacios[i][0] += tam_seg
            self.espacios[i][1] -= tam_seg

            self.segmentos.append([pos, tam_seg])
            proceso.append([len(self.segmentos)-1, tam_seg])

        self.procesos.append([proceso, True])
```

Paginacion.py

Donde el gestor de paginación será incluido dentro del programa

```python
from math import floor
from gestor import Gestor

class Paginacion(Gestor):
    def __init__(self, ram):
        super().__init__(ram)
        self.tam_pag = 0x10000
        self.tam_ram = 0x1000000
        self.marcos=[]
        self.n_marcos = int(self.tam_ram/self.tam_pag)
        for i in range(self.n_marcos):
            self.marcos.append(0)
        self.hacer_marcos()

    def nuevo_proceso(self, tam):
        proceso = []
        n = floor(tam/self.tam_pag)
        if(tam%self.tam_pag != 0):
            n+=1
        for i in range(n):
            j = 0
            m = self.marcos[j]
            while m!=0:
                j+=1
                m = self.marcos[j]
            if(tam <= self.tam_pag):
                self.marcos[j] = tam
            else:
                tam = tam-self.tam_pag
                self.marcos[j] = self.tam_pag

            proceso.append(j)

        self.procesos.append([proceso, True])

        cl = self.RAM.colores[len(self.procesos)%len(self.RAM.colores)]
        for pag in proceso:
            self.RAM.pintar_proceso(pag, self.marcos[pag], cl)

    def terminar_proceso(self, i):
        proceso = self.procesos[i]
        for pag in proceso[0]:
            self.RAM.pintar_proceso(pag, self.marcos[pag], 'black')
            self.marcos[pag] = 0
        proceso[1] = False

    def hacer_marcos(self):
        for i in range(len(self.marcos)):
            self.RAM.pintar_division(i*self.tam_pag, self.tam_pag)
```

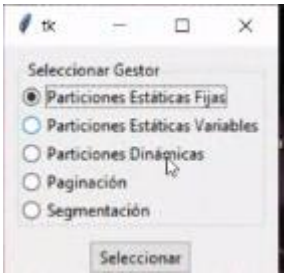
Particiones.py

En el archivo de particiones.py definiremos las funciones necesarias para poder particionar la memoria creada donde podemos poner el tamaño de los procesos y si lo excede o no al tamaño de la partición

```
class EstaticaFija(Gestor):
    def __init__(self, ram):
        super().__init__(ram)
        self.tam_par = 0x40000
        self.particiones=[]
        for i in range(int(self.tam_ram/self.tam_par)):
            self.particiones.append(0)
        self.hacer_particiones()

    def nuevo_proceso(self, tam):

        if (tam>self.tam_par):
            print("El tamano del proceso exede la particion")
        else:
            for i in range(len(self.particiones)):
                if self.particiones[i] == 0:
                    self.particiones[i] = tam
                    break
            proceso = i
            self.procesos.append([proceso, True])

            cl = self.RAM.colores[len(self.procesos)%len(self.RAM.colores)]
            self.RAM.pintar_proceso(proceso, self.particiones[proceso], cl)
```

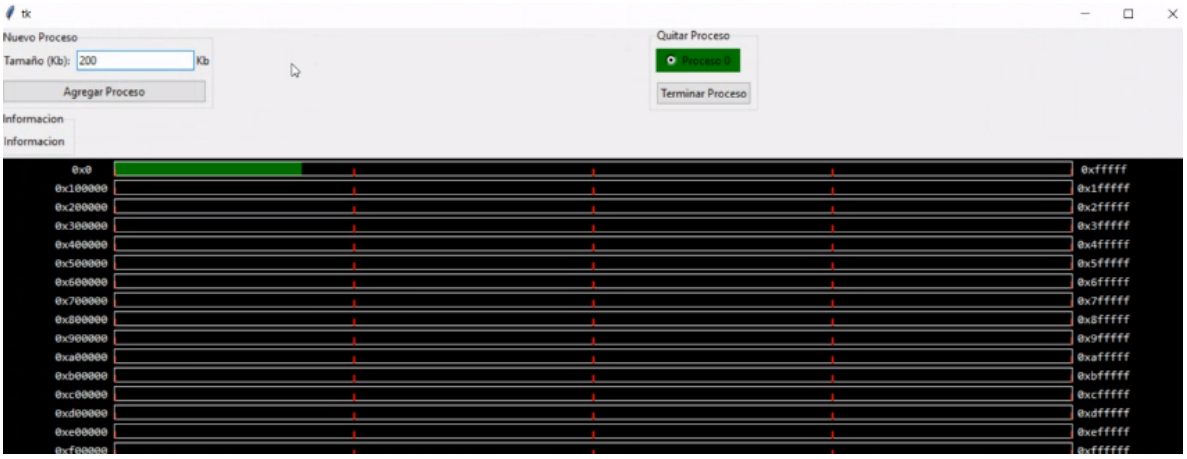
Ejecución del programa

Seleccionamos el gestor

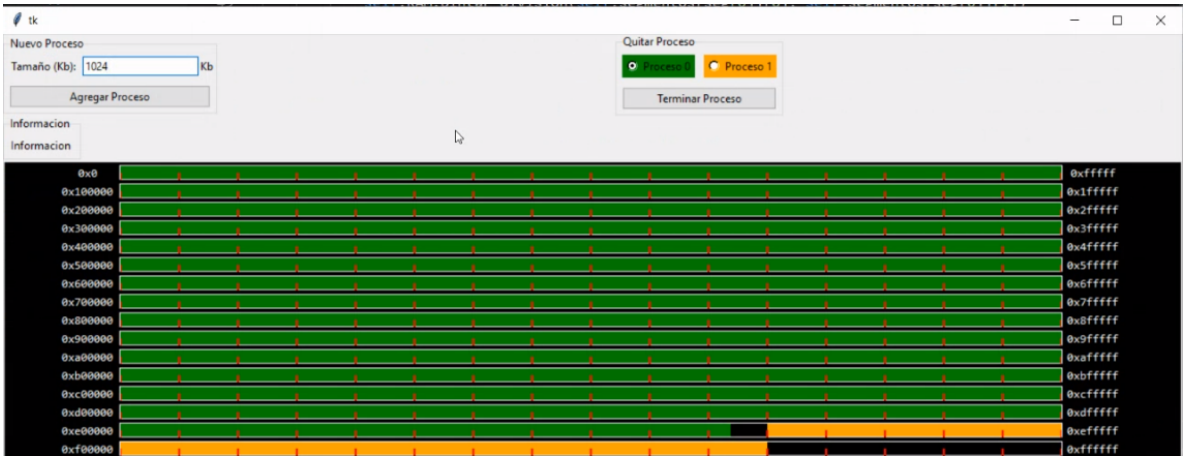


Incluimos el tamaño del proceso que queremos agregar en la parte superior izquierda

y los procesos quedan en kilobyts



Podemos ver como se va pintando la cantidad de memoria que esta usando el proceso



6. CONCLUSIONES

En conclusión, el programa puede definir el tamaño de cada uno de los segmentos en las clases que se han visto en la parte de la paginación y la segmentación, además de que podremos ver la memoria que ocupa los procesos que se seleccionan y si no queda más memoria que usar al final, saltara una ventana emergente cuando se intentase introducir un proceso que exceda la memoria además de que el programa ha incluido algunos de los algoritmos nombrados en los objetivos como el de mejor ajuste que se vio en la segmentación y el de primer ajuste en la de paginación.

7. REFERENCIAS:

[1] NtQueryInformationProcess function (winternl.h) - Win32 apps | Microsoft Docs [online] [Date 5/05/21]

[2] Bloque de control de procesos - Wiki de Sistemas Operativos (us.es) [online] [Date 5/05/21]

[3] Process Descriptor and the Task Structure (gigatux.nl) [online] [Date 5/05/21]

[4] Gestión de Procesos en los Sistemas Operativos (uoc.edu) [online] [Date 5/05/21]

[5] Particiones fijas y dinamicas [online] https://sistemasopers.blogspot.com/2015/09/particiones-fijas-y-dinamicas.html [Date 7/09/21]