Universidad Nacional Autónoma de México

Facultad de Ingeniería

Proyecto 3

(Micro) sistema de archivos

Materia: ***Sistemas Operativos***

Grupo: ***06***

Profesor: ***GUNNAR EYAL WOLF ISZAEVICH***

Integrantes:

Hernandez Hernandez Samuel

Vázquez Reyes Sebastián

Semestre: ***2024-1***

Proyecto 3: Micro sistema de archivos

El programa se encarga de dar solución al siguiente problema planteado: se tiene un archivo que simula el funcionamiento de un disco: “FiUNAMFS”, dentro de él, se alojan algunos archivos. El programa debe de:

- Listar los contenidos del directorio dentro del disco
- Copiar uno de los archivos de dentro del FiUnamFS hacia nuestro sistema
- Copiar un archivo de tu computadora hacia tu FiUnamFS
- Eliminar un archivo del FiUnamFS
- Desfragmentar al FiUnamFS.

Además, se implementó una interfaz gráfica sencilla utilizando Tkinter (la librería de python) para manejar de una manera más cómoda el programa, junto a una pequeña bitácora que lleva la cuenta de todas las operaciones con metadatos.

Antes de comenzar, es importante destacar las funciones que se utilizaran a lo largo del código para traducir contenido del disco a nuestra computadora: leerstring() y leerint().

```
#Funcion que lee enteros en el archivo fuente
def leerint(pocision, numero):
    with open("fiunamfs.img", "rb") as diskito:
        diskito.seek(pocision)
        dato = diskito.read(numero)
        num= struct.unpack("<I", dato)[0]
        return num

#Funcion que lee cadenas en el archivo fuente
def leerstring(pocision, numero):
    with open("fiunamfs.img", "rb") as diskito:
        diskito.seek(pocision)
        cadena = diskito.read(numero)
        string = cadena.decode("ascii")
        return string
```

Leerint() toma una posición de inicio y una cantidad de bytes que leera (numero), al igual que leerstring(). Ambas se encargan de posicionar el apuntador del archivo en la posición requerida, y luego leen la cantidad de datos que se les solicitó. La diferencia radica en que leerint() decodifica los datos utilizando el formato LITTLE ENDIAN, mientras que leerstring lo hace con el formato ASCII. Ambas retornan la cadena traducida como valor.

Ahora sí, analizaremos el programa siguiendo el orden de las operaciones mostradas anteriormente:

1. Código de las operaciones:

1.1 Listar los contenidos del directorio:

En primera instancia, se nos informa que la información básica acerca del disco se encuentra en el primer clúster del directorio. Por lo tanto, tenemos una primera función que se encarga de mostrar estos elementos:

```
#Funcion que recopila la informacion del server ubicada en el primer cluster
def validarserver():
    neim = leerstring(0,9)
    version = leerstring(10,5)
    etiqueta = leerstring(20,20)
    tamaño = leerint(40,4)
    cantidadclusters = leerint(45,4)
    ccl = leerint(50,4)
    #se imprimen en la interfaz las caracteristicas del sistema
    nom=Label(frame2, text = "El sistema se llama: "+neim, bg="gray71",fg="gray1",font=fonttext)
    nom.pack()
    nom=Label(frame2, text = "La version es: "+version, bg="gray71",fg="gray1",font=fonttext)
    nom.pack()
    nom=Label(frame2, text = "Etiqueta del volumen: "+etiqueta, bg="gray71",fg="gray1",font=fonttext)
    nom.pack()
    nom=Label(frame2, text = "Tamaño del Cluster: "+str(tamaño)+" bytes", bg="gray71",fg="gray1",font=fonttext)
    nom.pack()
    nom=Label(frame2, text = "Numero de clusters que mide el directorio: "+str(cantidadclusters), bg="gray71",fg="gray1",font=fonttext)
    nom.pack()
    nom=Label(frame2, text = "Numero de clusters que mide toda la unidad: "+str(ccl), bg="gray71",fg="gray1",font=fonttext)
    nom.pack()
```

Esta función se encarga de leer las cadenas y los enteros ubicados en el primer clúster para obtener la información. La información no la imprime en consola, si no que utiliza etiquetas para mostrarlas en la interfaz gráfica (se abordara más adelante en la sección de ejecución)

Aunado a la función anterior, también tenemos una función que se encarga de listar los archivos contenidos en el directorio:

```
#Funcion que lista el contenido del directorio, ignorando las entradas vacias
def listado():
    tabla.delete(*tabla.get_children())
    for i in range (0, (cluster*4), 64):
        r=leerstring(cluster+i,16)
        if (r[0]!="/"):
            a=leerint(cluster+i+16,4)
            b=leerint(cluster+i+20,4)
            c=leerstring(cluster+i+24,14)
            d=leerstring(cluster+i+38,14)
            tabla.insert("", END, text=r, values=(a,b,c,d))#se almacenan en la tab
```

Esta función realiza un barrido entero alrededor de los clústeres 1 a 4 dentro del archivo. Es en estos clústeres donde están los directorios que hacen referencia a los archivos almacenados en el disco. La función recorre los contenidos del directorio, y cuando encuentra una entrada que no es vacía (una que no tiene como signo inicial el carácter “/”) almacena los datos de este archivo en una tabla de tipo TreeView, objeto de la librería Tkinter para mostrar tabulaciones en modo gráfico.

La función sabe que dentro del directorio, cada entrada consta de 64 bits en donde se guardan todos los datos de los archivos almacenados en el disco o simplemente cadenas vacías indicando que no hay nada. En los requisitos del proyecto se nos explicó que los 64 bits se dividen de la siguiente forma:

- 0: Tipo de archivo.
- 1–15: Nombre del archivo
- 16–19: Tamaño del archivo, en bytes
- 20–23: Clúster inicial
- 24–38: Hora y fecha de creación del archivo
- 38-52: Hora y fecha de última modificación del archivo
- 52–64 :Espacio no utilizado

Es gracias a estos datos que somos capaces de definir los índices de búsqueda para las funciones leerint() y leerstring().

Esta función se ejecuta varias veces a lo largo del código para mantener actualizada la tabla después de operaciones de borrado o copiado.

1.2 Copiar uno de los archivos de dentro del FiUnamFS hacia nuestro sistema

Para esta parte del código se implementa la función moverdesde(). Cuando listamos los datos del clúster 0, obtuvimos el tamaño de cada clúster en total: 2048 bytes. Por lo tanto, tomamos este valor como referencia para analizar el archivo a copiar.

```
#La siguiente funcion mueve un objeto desde el disco hasta nuestro computador. La copia se hace directamente en el directorio del programa
def moverdesde(var):
    global cluster, directorioact
    tamaño, cl_ini, cadenafull = buscar(var)
    datos = leercont(cluster*cl_ini, tamaño)

    for r,_, archivos in os.walk(directorioact):
        if cadenafull in archivos:
            nom=Label(frame2, text = "El archivo: "+cadenafull +" ya se encuentra en el direcotorio del programa", bg="gray71",fg="gray1",font=fontte
            nom.pack()
            registrar_bitacora("Intento de mover el archivo " + cadenafull + ", pero ya existe en el directorio")
            break
        else:
            archcopy = open(cadenafull,"wb")
            archcopy.write(datos)
            archcopy.close()
            nom=Label(frame2, text = "El archivo: "+cadenafull +" se ha copiado exitosamente en el direcotorio del programa", bg="gray71",fg="gray1",
            nom.pack()
            registrar_bitacora("Archivo " + cadenafull + " copiado exitosamente al directorio del programa")
            break
```

En primera instancia, esta función hace una llamada a otra función llamada buscar(var), que busca el nombre del archivo que el usuario eligió para copiar en la tabla donde se almacenan los archivos y sus datos. Si lo encuentra, entonces devuelve los datos del archivo a copiar.

```
def buscar(var):#Esta funcion se encarga de buscar
    nombre = var.get()
    for item in tabla.get_children():
        l = tabla.item(item, "text")
        if l == nombre:
            cadena2 = tabla.set(item, "Tamaño")
            cadena3 = tabla.set(item, "Cluster")
            break
    tamaño = int(cadena2)
    cluster = int(cadena3)
    cadenafull = limpiar_cadena(nombre)
    return tamaño, cluster, cadenafull
```

Con los datos del archivo, la función ahora copia el contenido del archivo que piensa copiar en una variable a través de una función que se encarga de hacer esta copia. Luego, ejecuta un ciclo for para buscar en el directorio, donde se ubica el programa fuente, si el archivo que se planea copiar ya ha sido copiado, si es así, entonces se avisa al usuario que el archivo ya ha sido copiado desde FiUNAMFS a su sistema. Sin embargo, si el archivo no se encuentra en el directorio, significa que no ha sido copiado y se procede a copiar dicho archivo.

Para realizar el copiado, se abre el archivo a copiar en modo binario y se procede a copiar cada uno de sus bytes en el archivo copia.

Se realice o no la copia de un archivo, se registran en la bitácora cualquiera de ambos movimientos.

1.3. Copiar un archivo de tu computadora hacia tu FiUnamFS

En primer lugar, es importante recalcar que esta funcionalidad está incompleta y no realiza adecuadamente su trabajo. El trabajo se llevaría a cabo con la función moverhacia(), que le permite al usuario abrir un archivo de cualquier tipo, y posteriormente compara si el nombre del archivo tiene la longitud permitida por el directorio del disco. En caso de que la longitud no fuera lo suficientemente corta, la función simplemente le avisaba al usuario que el nombre era muy largo y no se ejecutaba nada. Si el nombre resulta ser correcto, entonces se busca dentro del disco si el archivo ya existe, para no volver a copiarlo. En caso de que el archivo no existiera, se procedía al copiado del archivo, pero esa parte no fue definida.

```
def moverhacia(): #Funcion para mover un archivo a fiUNAMFS
    global cluster, directorioact
    #Ventana de seleccion de archivos para la funcion de mover archivos hacia FiUNAMFS
    archmover = filedialog.askopenfilename(initialdir=directorioact, title="Selecciona un archivo", filetypes=(("png files", "*.png"), ("all files",
    tamaño = os.path.getsize(archmover)
    nombre2, extension_archivo = os.path.splitext(archmover)
    print (str(tamaño))
    print (nombre2)
    print (extension_archivo)
    if (len(nombre2)+len(extension_archivo))>15:
        nom = Label(frame2, text="El nombre del archivo es demasiado grande", bg="gray71", fg="gray1", font=fonttext, wraplength=700)
        nom.pack()
        return
    for item in tabla.get_children():
        l = tabla.item(item, "text")
        if l == nombre2:
            nom = Label(frame2, text="Este archivo ya esta en el sistema", bg="gray71", fg="gray1", font=fonttext, wraplength=700)
            nom.pack()
            break
    nombre = "-"+ nombre2 + extension_archivo + " " * (14 - len(nombre))
    fecha_modificacion = os.path.getmtime(archmover)
    fecha_creacion = os.path.getctime(archmover)
```

1.4. Eliminación

La eliminación de un archivo dentro del disco se lleva a cabo con la siguiente función:

```
def eliminar(var): #Esta funcion se encarga de eliminar el archivo del directorio que escoja el usuario
    global cluster, directorioact
    tamaño, cl_ini, cadenafull = buscar(var)
    encontrar = False
    for i in range(0, (cluster*4), 64):
        r = limpiar_cadena(leerstring(cluster+i, 16))
        if r == cadenafull:
            with open("fiunamfs.img", "rb+") as diskito:
                diskito.seek(cluster+i)
                diskito.write(b'/..............\x00')
                diskito.seek(cluster+i+16)
                diskito.write(b'\x00'*8)
                diskito.seek(cluster+i+24)
                diskito.write(b'00000000000000'*2)
                encontrar=True
    if encontrar==True:
        with open("fiunamfs.img", "rb+") as diskito:
            diskito.seek(cluster*cl_ini)
            diskito.write(b'\x00'*tamaño)
        nom = Label(frame2, text="El archivo: " + cadenafull + " se ha eliminado exitosamente", bg="gray71", fg="gray1", font=fonttext, wraplength=70
        nom.pack()
        registrar_bitacora(f"Eliminación del archivo: {cadenafull}")  # Registro en la bitácora
    else:
        nom = Label(frame2, text="El archivo: " + cadenafull + " ya ha sido eliminado", bg="gray71", fg="gray1", font=fonttext, wraplength=700)
        nom.pack()
```

La función primero busca al archivo a eliminar en cuestión dentro de la tabla generada, llamando a la funcion buscar(var) definida anteriormente. Si encuentra el archivo, reemplaza las entradas del directorio por las cadenas: b'/..............\x00' y b'\x00'. Ahora, cuando realizamos el listado nos dimos cuenta que las referencias a los archivos en los primeros 4 clusters no aparecen una sola vez, sino varias veces, entonces hay que eliminar todas las entradas del directorio que hagan referencia al mismo archivo a eliminar. Para esto, se recorre el directorio y cada vez que se encuentra una coincidencia con el nombre del archivo a eliminar, se reemplazan las cadenas que almacenan sus datos por cadenas vacías.

Posteriormente, si el archivo ha sido encontrado y se borró del directorio, se borra ahora del disco, es decir, borramos sus datos a partir de donde está ubicado (el cluster inicial). Para borrar sus datos, simplemente reemplazamos estos por la cadena vacía:

b'\x00', que es la cadena que representa a los datos vacíos dentro del disco. Cuando borramos algún archivo, se guarda este dato en la bitácora.

1.5. Desfragmentación

Para empezar el proceso de desfragmentación primero debemos identificar archivos fragmentados en el sistema de archivos. Esto lo hacemos con la funcion identificar_fragmentados().

La función primero abre el archivo de imagen del sistema de archivos (fiunamfs.img) en modo lectura binaria.

```
# Función para identificar archivos fragmentados en el sistema de archivos,
def identificar_fragmentados():
    fragmentados = []
    with open("fiunamfs.img", "rb") as diskito:
```

Después lee el contenido del archivo buscando archivos fragmentados.

```
diskito.seek(1 * cluster)
entrada_tamano = 64
while True:
    entrada = diskito.read(entrada_tamano)
    if not entrada or len(entrada) < 24:  # Asegurarse de que la entrada sea lo suficientemente larga
        break
    tipo_archivo = entrada[0]  # Tipo de archivo
    cluster_inicial = struct.unpack("<I", entrada[20:24])[0]  # Cluster inicial
    tamano_archivo = struct.unpack("<I", entrada[16:20])[0]  # Tamaño del archivo
```

Verifica si el tamaño de un archivo es mayor que el tamaño de un solo clúster, lo cual indica que está fragmentado.

```
# Verificar si el archivo ocupa más de un cluster (fragmentado)
if tipo_archivo != 47 and tamano_archivo > cluster * 4:
    fragmentados.append((cluster_inicial, tamano_archivo))
```

Finalmente devuelve una lista de tuplas que contienen información sobre archivos fragmentados (inicio del clúster, tamaño del archivo).

```
return fragmentados
```

Ahora hay que encontrar espacio libre en el disco para ubicar un archivo elegido para desfragmentar. Esto lo hacemos con la función encontrar_espacio_libre(disco, tamano_archivo)

Primero se determina el tamaño total del disco leyendo el archivo completo y contando los bytes vacíos (\x00).

```python
# Función para encontrar espacio libre en el disco
def encontrar_espacio_libre(disco, tamano_archivo):
    # Verificar el tamaño del archivo antes de buscar espacio
    disco.seek(0, 2)  # Ir al final del archivo
    tamano_disco = disco.tell()  # Obtener el tamaño total del disco
    disco.seek(0)  # Volver al inicio del archivo
    espacio_total = disco.read().count(b'\x00')  # Contar el número total de espacios libres
```

Verifica si el tamaño del archivo que se desea guardar es mayor que el espacio total disponible en el disco.

```python
if tamano_archivo > espacio_total:
    print(f"Tamaño total del disco: {espacio_total}")
    print(f"El archivo es demasiado grande para el espacio libre disponible. Tamaño del archivo: {tamano_archivo},
          Espacio libre: {espacio_total}")
    raise ValueError("No se encontró espacio suficiente para el archivo.")
```

Después busca un espacio vacío de tamaño suficiente para almacenar el archivo.

```python
# Buscar espacio en el disco
disco.seek(1 * cluster)  # Inicio del espacio de datos
espacio_libre = b'\x00' * tamano_archivo  # Espacio libre requerido
datos = disco.read()  # Leer todos los datos
indice = datos.find(espacio_libre)  # Buscar la primera aparición del espacio libre
```

Si el archivo es demasiado grande para un espacio libre individual, maneja este caso especial

```python
# Manejar archivos grandes de manera diferente
if tamano_archivo > espacio_total:
    print(f"El archivo es demasiado grande para cualquier espacio libre individual.
          Considera dividir el archivo en partes más pequeñas. Tamaño del archivo: {tamano_archivo}")
    raise ValueError("No se encontró espacio suficiente para el archivo.")
```

Para esta función hay una cosa que destacar y es que presenta un pequeño error de lógica en la programación al momento de calcular el tamaño del disco, ya que cuenta todos los bytes vacíos presentes en el archivo de imagen del disco. Sin embargo, el espacio disponible para almacenar un archivo grande no es simplemente la suma de todos los bytes vacíos.
Se pueden encontrar fragmentos pequeños de bytes vacíos dispersos a lo largo del disco que, aunque cuenten en el recuento total, no podrían acomodar un archivo grande debido a su tamaño fragmentado. Por lo tanto, contar simplemente los bytes vacíos podría subestimar el espacio disponible real para un archivo grande, arrojando el siguiente mensaje, a pesar de que la siguiente función opera correctamente.

```
/.............
/.............
Tamaño total del disco: 25223492
El archivo es demasiado grande para el espacio libre disponible. Tamaño del archivo: 976363631, Espacio libre: 25223492
Exception in Tkinter callback
```

Finalmente, aunque no menos importante, se encuentra la función "desfragmentar", cuya responsabilidad principal es realizar la desfragmentación del sistema de archivos. Esta función identifica los archivos fragmentados utilizando la función identificar_fragmentados().

```
fragmentados = identificar_fragmentados()
cambios = {}
```

Después Itera sobre los archivos fragmentados:

El propósito es encontrar el espacio libre para cada fragmento utilizando encontrar_espacio_libre().

```
with open("fiunamfs.img", "r+b") as disco:
    for cluster_inicial, tamano_archivo in fragmentados:
        espacio_libre = encontrar_espacio_libre(disco, tamano_archivo)
        nuevo_cluster_inicial = espacio_libre // cluster
```

Realiza la transferencia de archivos al nuevo espacio y registra estos cambios.

```
registrar_bitacora(f"Moviendo archivo fragmentado desde el cluster {cluster_inicial} al cluster {nuevo_cluster_inicial}")
cambios[cluster_inicial] = nuevo_cluster_inicial
contenido_archivo = leercont(cluster * cluster_inicial, tamano_archivo)
disco.seek(espacio_libre)
disco.write(contenido_archivo)
```

Aplica los cambios realizados en una sola pasada, actualizando los clústeres de los archivos fragmentados en el disco.

```
# Aplicar todos los cambios en el disco en una sola pasada
for cluster_inicial, nuevo_cluster in cambios.items():
    disco.seek(1 * cluster + cluster_inicial * cluster)
    disco.write(struct.pack("<I", nuevo_cluster))
    disco.seek(cluster * cluster_inicial)
    disco.write(b'\x00' * tamano_archivo)
nom = Label(frame2, text="Desfragmentacion completa", bg="gray71", fg="gray1", font=fonttext, wraplength=700)
nom.pack()
registrar_bitacora("Sistema de archivos desfragmentado exitosamente")
```

Notifica la finalización de la desfragmentación y registra este evento en la bitácora.

Por último, tenemos el código de la bitácora, que es una sencilla función que recibe como parámetro una cadena que describe el evento a registrar, y luego calcula su fecha de operación para registrarla igualmente. A su vez, también tenemos una función que se encarga de mostrarle al usuario la bitácora en una ventana nueva.

```
#Esta funcion se encarga de registrar las acciones realizadas en la bitacora con su hora de entrada
def registrar_bitacora(accion):
    with open("bitacora.txt", "a") as bitacora:
        from datetime import datetime
        fecha_hora_actual = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
        bitacora.write(f"{accion}, {fecha_hora_actual}\n")

#Esta funcion le permite al usuario ver la bitacora
def ver_bitacora():
    global directorioact
    bitacoraencont=False
    for r,_, archivos in os.walk(directorioact):
        if "bitacora.txt" in archivos:
            with open("bitacora.txt", "r") as bitacora:
                bitacora_content = bitacora.readlines()
                # Crear ventana para mostrar la bitácora
                ventana_bitacora = Toplevel(root)
                ventana_bitacora.title("Bitácora de operaciones")
                ventana_bitacora.geometry("500x500")
                bitacora_texto = Text(ventana_bitacora)
                bitacora_texto.pack(fill="both", expand=True)
                for linea in bitacora_content:
                    bitacora_texto.insert(END, linea)
            bitacoraencont = True
    if bitacoraencont==False:
        nom = Label(frame2, text="La bitacora no existe porque no se ha realizado alguna operacion", bg="gray71", fg="gray1", font=fonttext, wrapleng
        nom.pack()

"""
```

2. Código de la interfaz:

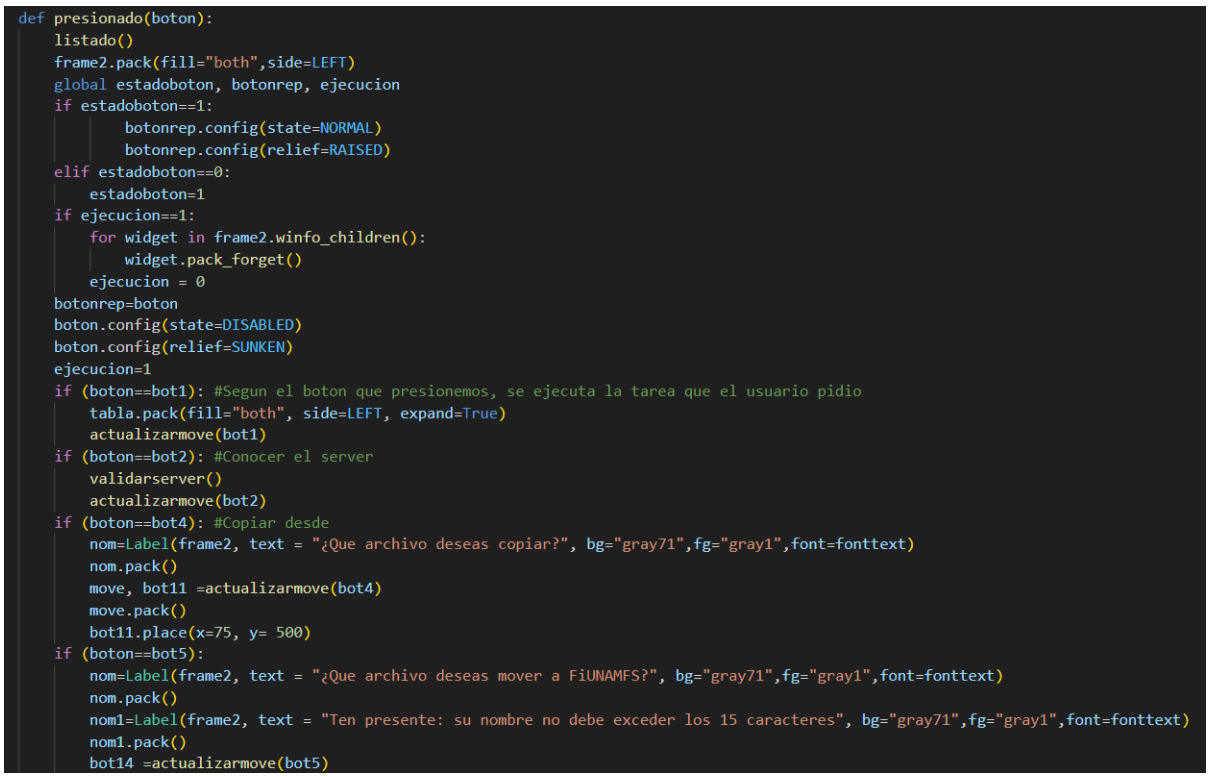
Es gracias a la interfaz grafica que nos podemos apoyar en algunos complementos para que el código funcione. Uno de ellos es la tabla donde se almacenan los archivos existentes en el directorio: tabla.treeview. Con ella se realizan varias operaciones a lo largo del programa, como si fuera una base de datos de la que se extraen nombres, tamaños, etc.

La interfaz gráfica no es un gran problema de entender, salvo algunas funciones implementadas para evitar problemas cuando el usuario la utilice. Estas funciones son:

2.1 presionado(boton)

Esta función se encarga de evitar que el usuario sea capaz de presionar mas de 1 vez el mismo botón que selecciona una operación de las descritas anteriormente. Lo que hace es desactivar el botón que fue presionado y darle una apariencia de sumido, de esta forma el usuario identifica que botón ha sido presionado y que no puede volver a presionar hasta que se libere. El botón se libera cuando se selecciona otra operación, bloqueando ahora el nuevo botón presionado.

Además, la función también se encarga de mostrar o desaparecer los botones, frames, mensajes y demás contenidos que la interfaz utiliza para mostrar los resultados del código. Por ejemplo, para la función de listado, solo ocupa mostrar la lista



```
def presionado(boton):
    listado()
    frame2.pack(fill="both",side=LEFT)
    global estadoboton, botonrep, ejecucion
    if estadoboton==1:
            botonrep.config(state=NORMAL)
            botonrep.config(relief=RAISED)
    elif estadoboton==0:
        estadoboton=1
    if ejecucion==1:
        for widget in frame2.winfo_children():
            widget.pack_forget()
        ejecucion = 0
    botonrep=boton
    boton.config(state=DISABLED)
    boton.config(relief=SUNKEN)
    ejecucion=1
    if (boton==bot1): #Segun el boton que presionemos, se ejecuta la tarea que el usuario pidio
        tabla.pack(fill="both", side=LEFT, expand=True)
        actualizarmove(bot1)
    if (boton==bot2): #Conocer el server
        validarserver()
        actualizarmove(bot2)
    if (boton==bot4): #Copiar desde
        nom=Label(frame2, text = "¿Que archivo deseas copiar?", bg="gray71",fg="gray1",font=fonttext)
        nom.pack()
        move, bot11 =actualizarmove(bot4)
        move.pack()
        bot11.place(x=75, y= 500)
    if (boton==bot5):
        nom=Label(frame2, text = "¿Que archivo deseas mover a FiUNAMFS?", bg="gray71",fg="gray1",font=fonttext)
        nom.pack()
        nom1=Label(frame2, text = "Ten presente: su nombre no debe exceder los 15 caracteres", bg="gray71",fg="gray1",font=fonttext)
        nom1.pack()
        bot14 =actualizarmove(bot5)
```

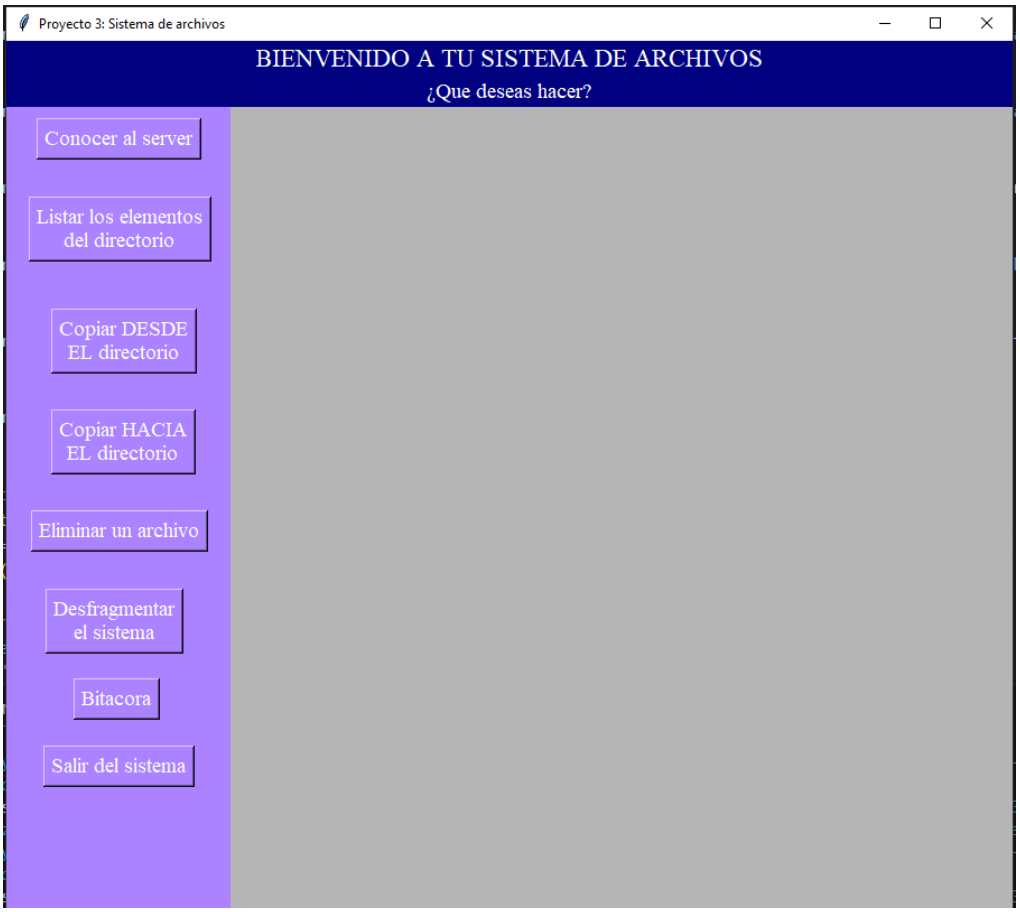
2.2 actualizarmove(bot):

Esta función se encarga de mostrar o desaparecer los botones que ayudan al usuario a seleccionar que archivo desea eliminar, mover, etc. Además, también se encarga de administrar que función ejecutar cuando se presionan los botones que ayudan al usuario a seleccionar archivos.

```
#Menu de seleccion para las funciones que le ofrecen al usuario escoger archivos dentro del directorio
def actualizarmove(bot):
    var=StringVar()
    neims = tabla.get_children()
    archivos = [tabla.item(item, "text") for item in neims]
    var.set(archivos[0])
    listado()
    move=OptionMenu(frame2, var, *archivos)
    bot11=Button (frame2, text="Escoger", bg="gray71",fg="gray1",font=fonttext, command=lambda: moverdesde(var))
    bot12=Button (frame2, text="Escoger", bg="gray71",fg="gray1",font=fonttext, command=lambda: eliminar(var))
    bot13=Button (frame2, text="Aceptar", bg="gray71",fg="gray1",font=fonttext, command=lambda: desfragmentar())
    bot14=Button (frame2, text="Escoger", bg="gray71",fg="gray1",font=fonttext, command=lambda: moverhacia())
    if (bot==bot6):
        return move, bot12
    elif (bot==bot4):
        return move, bot11
    elif (bot==bot7):
        return move, bot13
    elif (bot==bot5):
        return bot14
    else:
        bot11.place_forget()
        bot12.place_forget()
        bot13.place_forget()
        bot14.place_forget()
```

3. Ejecución:

Cuando se ejecuta el código, se muestra la interfaz gráfica (menu) de la siguiente manera:

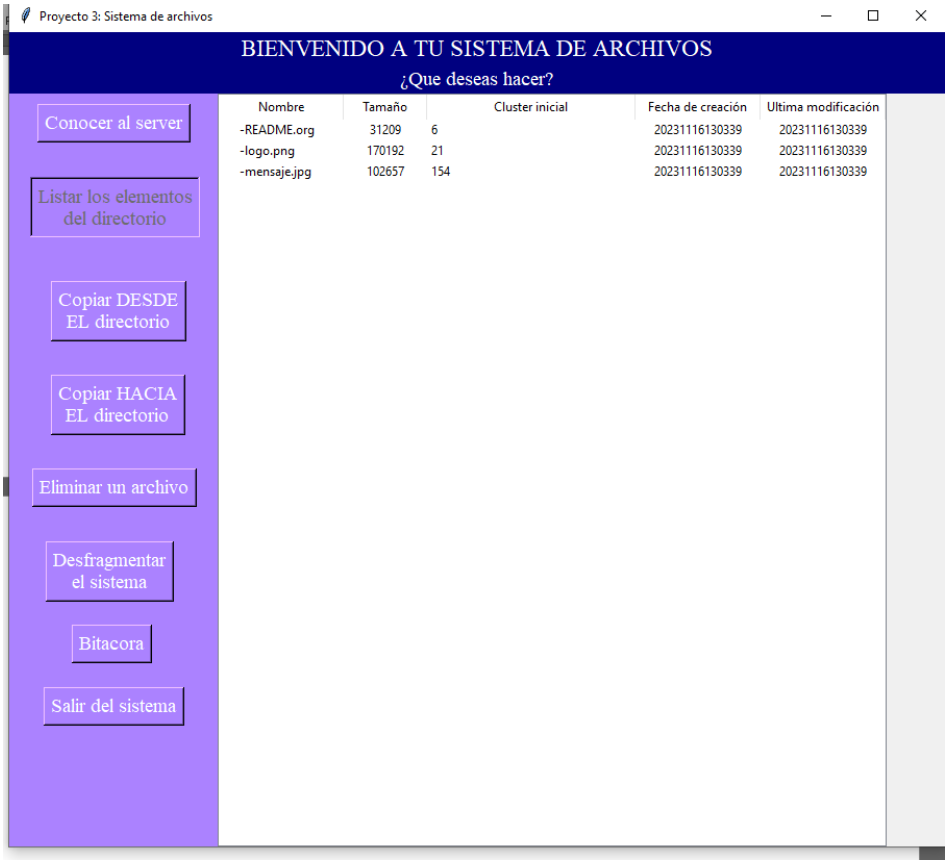


Cada botón muestra su funcionalidad. Empezamos de arriba a abajo:

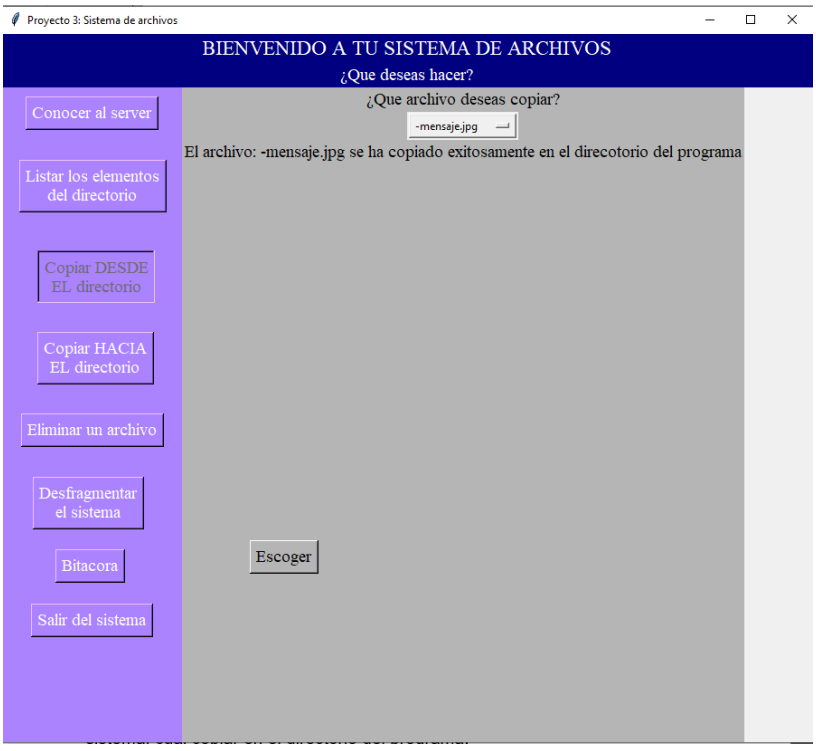
- Conocer el server: Cuando presionamos este botón, vemos la siguiente respuesta:

Son los datos del sistema

- Listar los elementos del directorio: Se muestran los archivos disponibles en el disco:



- Copiar DESDE el sistema: Nos permite escoger de entre los archivos del sistema, cual copiar en el directorio del programa.



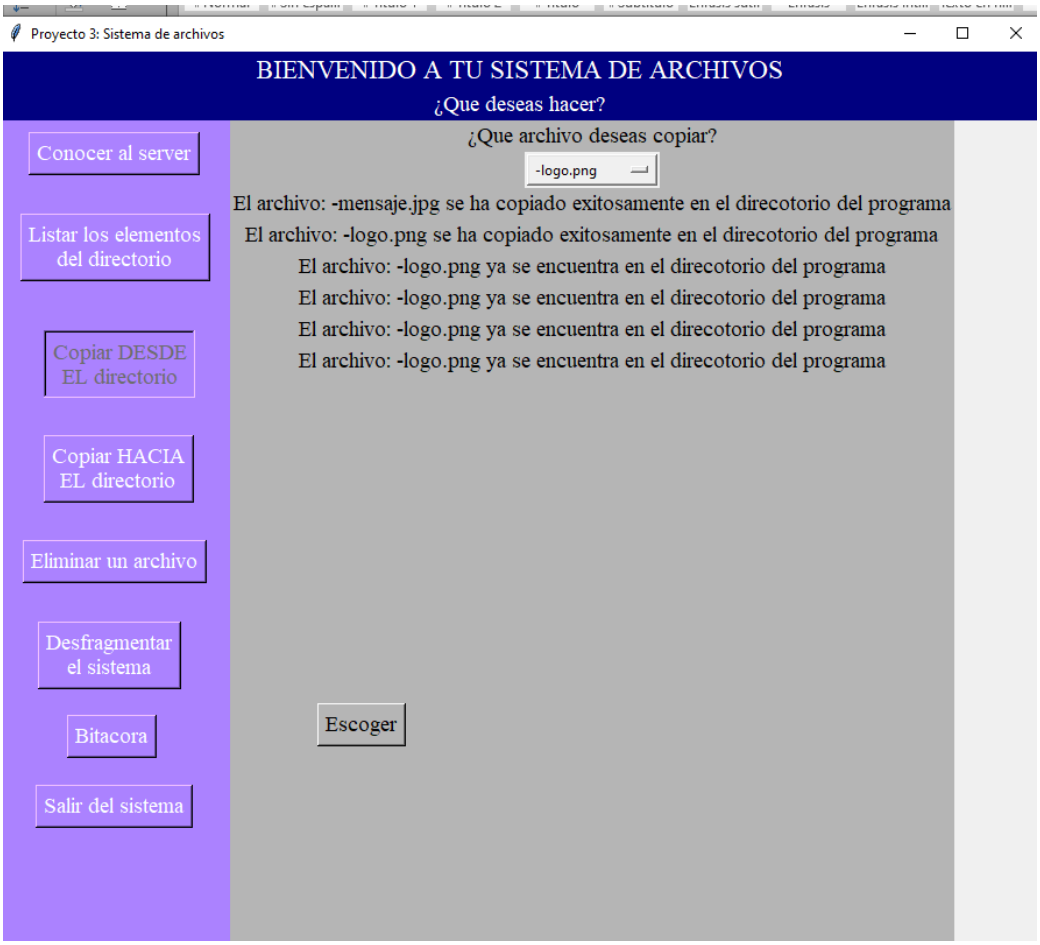
Cuando elegimos uno, se muestra el mensaje de que ha sido copiado. Si repetimos la operación para el mismo archivo, nos dirá que ya ha sido copiado al directorio. Además, podemos consultar el directorio para revisar el archivo copiado. Para el caso de mensaje.jpg, el archivo es este:

```
¡Muy bien hecho!
```

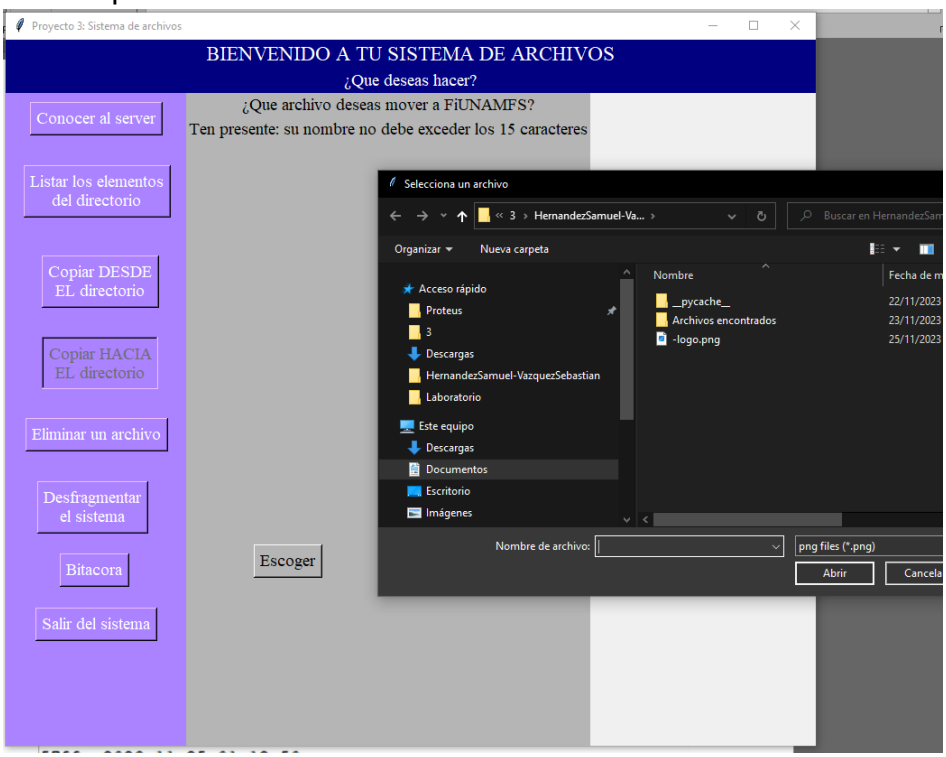
```
¡Lograste resolverlo! :-)
```

Y para el caso de logo.png, el archivo es este:

Ejemplo de varias ejecuciones del mismo archivo:

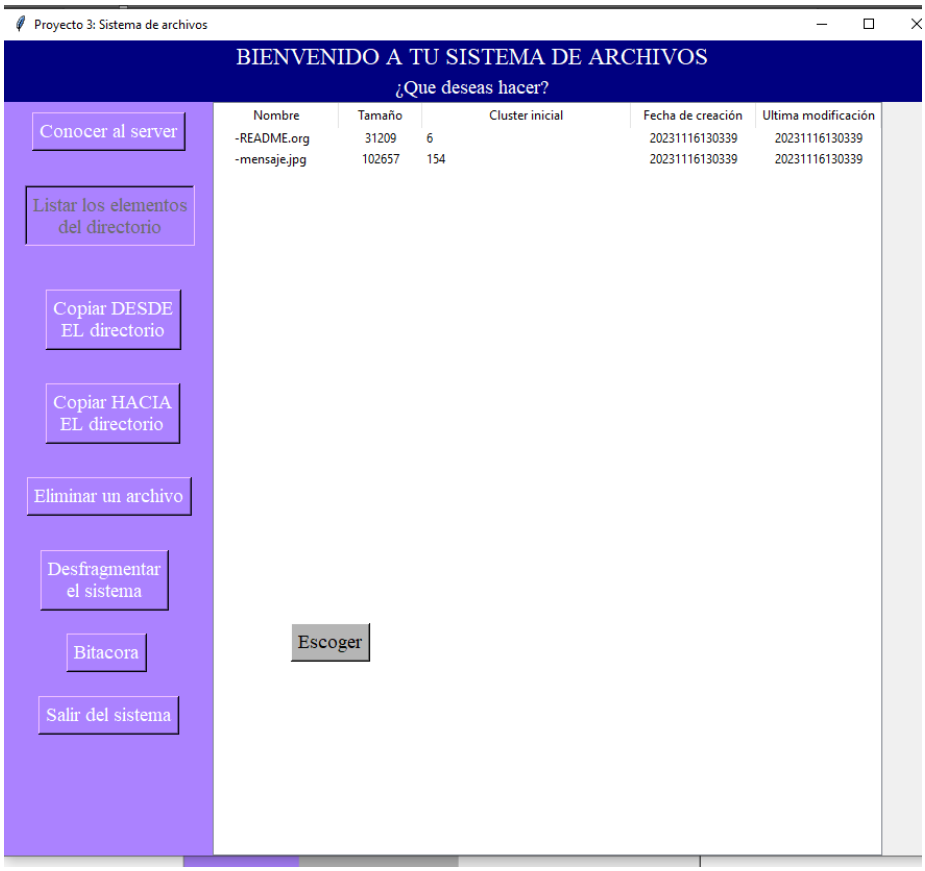


- Copiar HACIA el directorio: Nos permite escoger un archivo, pero no muestra nada más en pantalla:



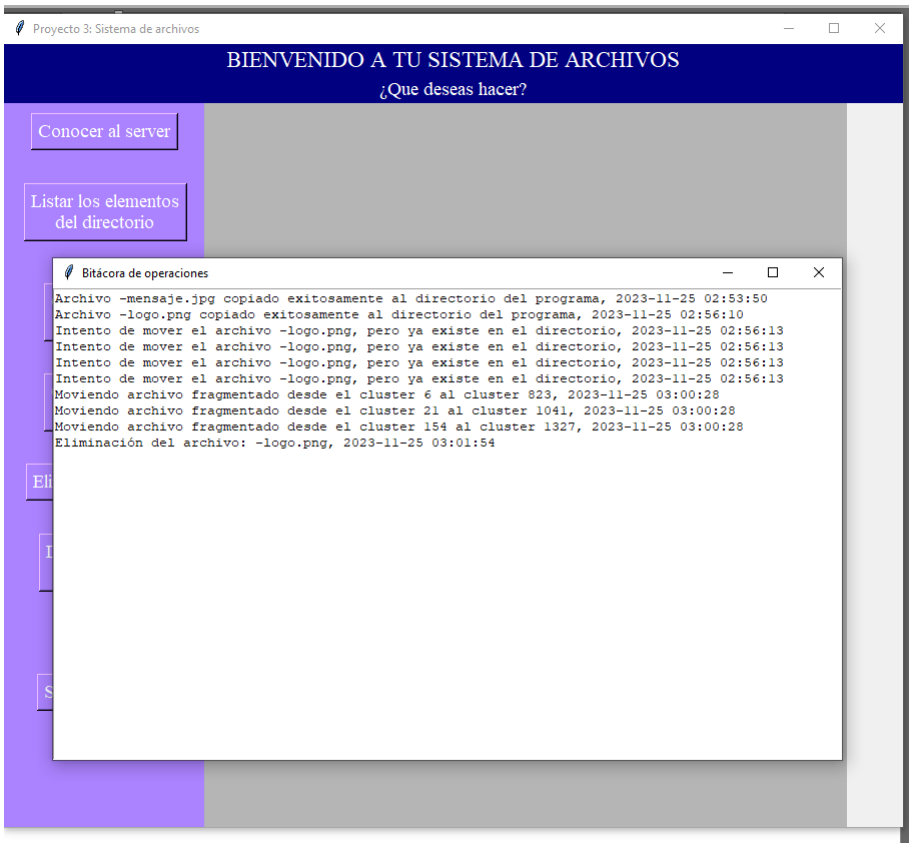
- Eliminar archivo: Le ofrece al usuario escoger que archivo eliminar.
  Cuando escoges por primera vez un archivo se elimina, volver a intentarlo solo lleva a mostrar un mensaje de error, marcando que ya se ha eliminado el archivo.

Si consultamos los datos de los archivos, ya no aparece el archivo eliminado:



- Desfragmentar el sistema: Le ofrece al usuario escoger que archivo desfragmentar. Al escoger uno, el usuario no visualiza ningún cambio:

- Bitácora: Muestra la bitácora de operaciones realizadas hasta ahora. Si no hay operaciones realizadas, le avisa al usuario que no se han realizado operaciones aun y no hay bitácora.



- Salir del sistema: Seleccionar esta opción terminara con la ejecución del programa