


UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Facultad de Ingeniería

Sistemas Operativos

(Micro) sistema de archivos multihilos

Alumno:

Hernandez Gallardo Daniel Alonso

Profesor:
Gunnar Wolf
Grupo 6, Semestre 2025-1

November, 2024

Índice general

1 Introducción

1.1 Objetivos

Implementar un micro sistema de archivos con las siguientes características:

1. Listar los contenidos del directorio
2. Copiar uno de los archivos de dentro del FiUnamFS hacia tu sistema
3. Copiar un archivo de tu computadora hacia tu FiUnamFS
4. Eliminar un archivo del FiUnamFS
5. Contar con por lo menos dos hilos de ejecución, operando concurrentemente, y que se comuniquen su estado mediante mecanismos de sincronización.

1.2 Entorno de desarrollo

Este programa se trabajo en el IDE Visual Studio Code, en el lenguaje de programación Python en su versión 3.13.0, ademas se usaron las siguientes bibliotecas:

1. import os
2. import struct
3. import threading
4. import math
5. import tkinter as tk
6. from datetime import datetime
7. from tkinter import ttk, messagebox, filedialog
8. from tkinter import messagebox, filedialog

Todas son parte de la biblioteca estándar de python, pero en caso de Tkinter (versión 8.6) es posible que en algunos sistemas operativos sea necesario instalarla.

2 Desarrollo

El código se divide principalmente en dos secciones:

1. La clase FIUnamFS que maneja el sistema de archivos
2. La Clase FiUnamFSApp que maneja la interfaz gráfica y la implementación de hilos

Para cada función de la clase tiene su homologa de la otra clase, es decir, la función que se encarga de eliminar en el disco para la clase FiUnamFS tiene como nombre:

```
def __EliminarDelDisk__(self, NombreArchivoAEliminar):
```

Su homologa en la clase FiUnamFSApp sera:

```
def delete_file(self):
```

Esta convención se usa porque realmente la clase FiUnamFSApp solo se encarga de mandar a llamar las funciones del sistema de archivos, pero se hace de manera separada para diferencias claramente el manejo de hilos y la interfaz gráfica de la lógica del manejo de archivos. Ademas en este documento he eliminado los comentarios del código, para mostrarlo de una forma más cómoda.

2.0.1 Inicialización

Esta es la parte del código que realiza la instanciación de las clases (Main) y ademas al principio del código se declara la variable de condición para la "función enlistadora":

```
VCListFiles = threading.Condition()
fs = FiUnamFS("../fiunamfs.img")
root = tk.Tk()
app = FiUnamFSApp(root, fs)
root.mainloop()
```

2.1 FIUnamFS

2.1.1 Constructor de la clase

El constructor de esta clase se ve de la siguiente manera:

```
def __init__(self, disk):
    self.disk = disk
    self.lock = threading.Lock()
    self.archivos = None
```

Es muy simple, solamente recibe el sistema de archivos que va a trabajar (disk), un mutex para manejar cómodamente el acceso al recurso compartido (que en este caso es el sistema de archivos) y declara una variable vacía que servirá para mantener los archivos de una manera local (Esto solo es útil porque el sistema pesa poco) y evitar accesos innecesarios, por ejemplo, si quiero copiar de FiUnamFs a mi computadora es más tardado buscar el archivo que quiero en el sistema de archivos a unicamente sacarlo del diccionario de la copia local.

2.1.2 Lectura del superbloque

La función se ve de la siguiente manera:

```
def __LeerSuperBloque__(self):
    with open(self.disk, 'rb') as f:
        f.seek(0)
        datos = f.read(54)
        nombre, version, Eti_volumen, Tam_cluster, dir_clusters,
        ↪  total_clusters = struct.unpack('<9s1x5s5x16s4xI1xI1xI',
        ↪  datos[:54])
        nombre = nombre.decode('ascii').strip('\x00')
        version = version.decode('ascii').strip('\x00')
        self.__validacion__(version,nombre)
        Eti_volumen = Eti_volumen.decode('ascii').strip('\x00')
        return {
            "Nombre": nombre,
            "Versión": version,
            "Etiqueta de Volumen": Eti_volumen,
            "Tamaño de Cluster": Tam_cluster,
            "Número de Clusters de Directorio": dir_clusters,
            "Total de Clusters": total_clusters
        }
```

Obtiene los datos según las especificaciones, he de mencionar que había errores pues la especificación indicaba una manera de extraer los datos del superbloque pero al momento de hacerla tal cual no obtenía los datos correctamente, así que se tuvieron que hacer unos ajustes.

2.1.3 Validación del sistema de archivos

La función que realiza este procedimiento se ve de la siguiente manera:

```
def  __validacion__(self,version,nombre):
    if(nombre!="FiUnamFS"):
        messagebox.showerror("Error de nombre", f"Nombre incorrecto:
        ↪  {nombre}. Se esperaba: {"FiUnamFS"}.")
        self.root.destroy()
    if version != "25-1":
```

```
            messagebox.showerror("Error de Versión", f"Versión incorrecta:
            ↪  {version}. Se esperaba: {"25-1"}.")
            self.root.destroy()
```

Para esto se reciben los datos obtenidos del superbloque y se validan según la especificación, si alguna de las dos cosas no cuadra (nombre o versión) se cierra todo el programa.

2.1.4 Enlistar Directorio

La función se ve de la siguiente manera:

```
def __EnlistarDirectorio__(self):
    Archivos = []
    with open(self.disk,'rb') as f:
        for i in range(1,5):
            f.seek(i*1024)
            for _ in range (16):
                entry = f.read(64)
                Tipo_Archivo, nombre, tamaño, clusterInicial, creado,
                ↪  modificado = struct.unpack('<c15sII14s14s12x',
                ↪  entry)
                if Tipo_Archivo.decode("ascii") == '#':
                    continue
                nombre = nombre.decode("ascii").strip("\x00").strip()
                if "---------------" in nombre :
                    continue
                Archivos.append({
                    "Nombre": nombre,
                    "Tamaño": tamaño,
                    "Creado": creado.decode("ascii",
                    ↪  errors='ignore').strip("\x00"),
                    "Modificado": modificado.decode("ascii",
                    ↪  errors='ignore').strip("\x00"),
                    "Cluster Inicial": clusterInicial
                })
    self.archivos = Archivos
    return Archivos
```

Esta función se encarga de obtener los archivos del directorio, desempaquetarlos, decodificarlos y pasarlos a la variable de archivos locales.

2.1.5 Copiar del disco

La funcion es la siguiente:

```
def __CopiarDelDisk__(self,NombreArchivoACopiar, DireccionAGuardar):
        with self.lock:
            ArchivoACopiar = next((f for f in self.archivos if f["Nombre"]
            ↪  == NombreArchivoACopiar), None)
            if ArchivoACopiar:
                Cluster_inicial = ArchivoACopiar["Cluster Inicial"]
                Tamaño = ArchivoACopiar["Tamaño"]
                with open(self.disk, 'rb') as Archivo:
                    Archivo.seek(Cluster_inicial*1024)
                    DatosArchivo = Archivo.read(Tamaño)
                with open(DireccionAGuardar, 'wb') as ArchivoGuardado:
                    ArchivoGuardado.write(DatosArchivo)
                messagebox.showinfo("Éxito", f"Archivo
                ↪  '{NombreArchivoACopiar}' copiado exitosamente a
                ↪  '{DireccionAGuardar}'.")
```

```
        else:
            messagebox.showerror("Error", "Archivo no encontrado en el
            ↪  sistema de archivos.")
```

Esta función esta protegida con un mutex, para que cuando se desee copiar las variables no se sobrescriban según los hilos que se accedan, por ejemplo, la variable archivos si es accedida por varios hilos puede ser modificada erróneamente, si una operación de eliminación sucede al mismo tiempo que esta podría caer en una condición de carrera que es evitada por este mutex; así evitamos que por ejemplo copiar un archivo que esta siendo eliminado por otro hilo.

2.1.6 Copiar al disco

Esta es sin duda la función más larga, pues el sistema operativo aquí ya no nos ayuda y tenemos que pasar todo manualmente al disco; se ve de la siguiente manera:

```
def __CopiarAlDisk__(self,DireccionArchivoACopiar):
with self.lock:
    if self.archivos == None:
        self.archivos = self.__EnlistarDirectorio__()
Nombre =
↪  os.path.basename(DireccionArchivoACopiar).encode("ascii").ljust(15,
↪  b'\x00')
for archivo in self.archivos:
    if archivo["Nombre"].encode("ascii").strip(b'\x00') ==
    ↪  Nombre.strip(b'\x00'):
        messagebox.showerror("Error", f"Ya existe un archivo con
        ↪  el mismo nombre en el sistema de archivos:
        ↪  {os.path.basename(DireccionArchivoACopiar)}")
        return
with open(DireccionArchivoACopiar, 'rb') as ArchivoFuente:
    Nombre =
    ↪  os.path.basename(DireccionArchivoACopiar).encode("ascii").ljust(15,b'\x00
    Tamaño = os.path.getsize(DireccionArchivoACopiar)
    Creado =
    ↪  datetime.now().strftime('%Y%m%d%H%M%S').encode('ascii')
    Modificado = Creado
    with self.lock:
        ClusterInicial = self.__HayEspacio__(Tamaño)
        if not ClusterInicial:
            messagebox.showerror("Error", f"Espacio insuficiente
            ↪  en el disco:
            ↪  '{os.path.basename(DireccionArchivoACopiar)}'")
            return False
        with open(self.disk, 'r+b') as disk_file:
            PosicionDirectorio =
            ↪  self.__PosicionDeDirectorioLibre__()
            if not PosicionDirectorio:
                messagebox.showerror("Error", f"No hay espacios en
                ↪  el directorio disponibles:
                ↪  '{os.path.basename(DireccionArchivoACopiar)}'")
```

```
                    return False
                disk_file.seek(PosicionDirectorio)
                disk_file.write(b'.')
                disk_file.write(struct.pack('<15s', Nombre))
                disk_file.write(struct.pack('<I', Tamaño))
                disk_file.write(struct.pack('<I', ClusterInicial))
                disk_file.write(struct.pack('<14s', Creado))
                disk_file.write(struct.pack('<14s', Modificado))
                disk_file.write(b'\x00' * 12)
            with open(self.disk, 'r+b') as disk_file:
                cluster = ClusterInicial*1024
                i=0
                while True:
                    data = ArchivoFuente.read(1024)
                    if not data:
                        break
                    disk_file.seek(cluster + i * 1024)
                    disk_file.write(data)
                    i+=1
                    if not cluster:
                        messagebox.showerror("Error", f"Espacio
                        ↪  insuficiente en los
                        ↪  clusters'{os.path.basename(DireccionArchivoACopiar)}'")
                        return False

        with VCListFiles:
            VCListFiles.notify_all()
        messagebox.showinfo("Éxito", f"Archivo  copiado exitosamente a
        ↪  el disco.'{os.path.basename(DireccionArchivoACopiar)}'")
        return True
```

Esta función copia al disco un archivo desde la computadora, para esto primero tiene que visualizar los archivos que hay en el disco por lo que, si no se han visualizado antes (es decir, la variable de archivos locales sigue en None) habrá que obtenerlos, esto se protege con un mutex para que los hilos queden a la espera de que el hilo que llego primero obtenga los archivos y ahora si puedan continuar con la ejecución de la función. También hace uso de la variable de condición que llama a la función 'enlistadora de archivos' que se encarga de actualizar la GUI y los archivos locales. La función verifica que no exista ya un archivo con el mismo nombre, extrae los metadatos del archivo y los codifica y empaqueta según la especificación para poder escribirlos en el archivo, la escritura se realiza con ayuda de dos funciones:

HayEspacio

Esta función luce de la siguiente manera:

```
def __HayEspacio__(self, Tamaño):
      cluster_size = 1024
      clusters_necesarios = math.ceil((Tamaño) / cluster_size)
      espacio_ocupado = sum(archivo["Tamaño"] for archivo in
      ↪  self.archivos)
      if espacio_ocupado + Tamaño 1024*5> 1440 * 1024:
```

```
            return False


        with open(self.disk, 'rb') as disk_file:
            espacios_contiguos = 0
            for cluster in range(1440):
                data = disk_file.read(cluster_size)
                if all(b == 0 for b in data):

                    espacios_contiguos += 1
                    if espacios_contiguos == clusters_necesarios:
                        return cluster-clusters_necesarios+1
                else:
                    espacios_contiguos = 0
        return  None
```

Lo primero que hace es medir cuanto espacio necesita y preguntarse ¿Tengo espacio total suficiente?, si es así prosigue a hacer la siguiente comprobación que es ¿Hay espacio contiguo suficiente?, si es así devuelve el cluster donde empieza el espacio suficiente encontrado.

Posición del directorio libre

Esta función encuentra un espacio libre en el directorio y luce de la siguiente manera:

```
def __PosicionDeDirectorioLibre__(self):
        with open(self.disk,'rb') as f:
            for i in range(1,5):
                f.seek(i*1024)
                for entry_index in range (16):
                    entry = f.read(64)
                    tipo, nombre = struct.unpack('<c15s48x', entry)
                    nombre = nombre.decode("ascii").strip("\x00").strip()
                    if tipo == b'#' or nombre == "---------------":
                        return i*1024+entry_index*64
        return None
```

Básicamente busca entre los 4 clusters que ocupa el directorio una entrada disponible, es decir, 64 bytes disponibles, el for abarca de 0 a 15 porque al cada cluster medir 1024 bytes y cada entrada 64 bytes, la división resulta en que hay 16 entradas por cada cluster, la verificación de disponibilidad se hace mediante el símbolo de validación '#' o el nombre '———————', finalmente devuelve el indice de la entrada disponible.

2.2 Eliminar del Disco

La ultima función resulta en:

```
archivo_a_eliminar= next((f for f in self.archivos if f["Nombre"]
↪  == NombreArchivoAEliminar), None)
cluster_inicial = archivo_a_eliminar["Cluster Inicial"]
tamaño = archivo_a_eliminar["Tamaño"]
clusters_necesarios = math.ceil(tamaño / 1024)
with self.lock:
    with open(self.disk, 'r+b') as disk_file:
        for i in range(1, 5):
            disk_file.seek(i * 1024)
            for entry_index in range(16):
                entry = disk_file.read(64)
                tipo ,nombre = struct.unpack('<1b15s48x', entry)
                nombre =
                ↪  nombre.decode("ascii").strip("\x00").strip()
                if nombre == NombreArchivoAEliminar:
                    disk_file.seek(i * 1024 + entry_index * 64)
                    disk_file.write(b'#')
                    break
        for cluster in range(cluster_inicial, cluster_inicial +
        ↪  clusters_necesarios):
            disk_file.seek(cluster * 1024)
            disk_file.write(b'\x00' * 1024)

    self.archivos.remove(archivo_a_eliminar)
with VCListFiles:
    VCListFiles.notify()
messagebox.showinfo("Éxito", f"Archivo '{NombreArchivoAEliminar}'
↪  eliminado exitosamente.")
return True
```

Aquí podemos apreciar varias cosas, en primera las variables que identifican nombre y tamaño no están protegidas con un mutex, esto es para hacer útil el paralelismo pues si todo tuviera un mutex al inicio es prácticamente como si fuera secuencial y en segunda es porque estamos hablando de la función de eliminación, por lo que si estoy eliminando un archivo, por como estos se seleccionan, no es posible que un hilo elimine un archivo que estoy a punto de eliminar, después de eso, el acceso al disco y al puntero si esta protegido por un mutex para evitar toda posible condición de carrera, especialmente para que el puntero no se cambie al eliminar archivos o al agregar el identificador de que la entrada del directorio esta disponible. El proceso de eliminación es simplemente localizar el archivo, reemplazar datos por ceros, liberar el espacio del directorio y eliminar del directorio local, finalmente también llama a la función .[en]listadora de archivos"

2.3 FiUnamFSAPP

Esta como mencione es la clase que se encarga de los hilos y la GUI.

2.3.1 Constructor

De aquí solo hay dos cosas que destacar, la clase usa el patrón de diseño bridge pues usa la composición de clases de tal modo que la GUI y los hilos se enlazan con el sistema de archivos que desees componerle, es decir, dentro de FiUnamFSApp hay un FiUnamFS (Un FileSystem vaya...); segunda cosa a denotar del constructor es que aquí se inicia la función enlistadora de archivos:

```
self.list_thread = threading.Thread(target=self.list_files)
self.list_thread.start()
```

2.3.2 Función enlistadora de archivos

Esta función se encarga de actualizar la GUI y es un hilo independiente que funciona con una variable de condición, es decir, es despertada cuando un hilo le notifica que es necesitada, la función que la despierta es:

```
def notify_list_files(self):
    with VCListFiles:
        VCListFiles.notify()
```

Y la función que el hilo esta ejecutando constantemente es:

```
def list_files(self):
    while True:
        with VCListFiles:
            VCListFiles.wait()
            self.tree.delete(*self.tree.get_children())
            files = self.fs.__EnlistarDirectorio__()
            for file in files:
                self.tree.insert("", "end", values=(file["Nombre"],
                ↪  file["Tamaño"],
                ↪  datetime.strptime(file["Creado"],"%Y%m%d%H%M%S").strftime("%Y-%m-
                ↪  %H:%M:%S"), file["Cluster Inicial"],
                ↪  datetime.strptime(file["Modificado"],"%Y%m%d%H%M%S").strftime("%Y
                ↪  %H:%M:%S")))
            self.tree.grid()
            self.tree_scroll.grid(row=1, column=2, sticky='ns')
```

Esta función cuando el hilo es despertado lo que hace es, borrar la GUI actual, actualizar los archivos locales revisando el sistema de archivos y finalmente les da formato y los coloca en la interfaz gráfica.

2.3.3 Mostrar Superbloque

Esta función manda a leer al superbloque que realiza la validación y todo lo necesario, si todo es correcto lo muestra en la GUI.

```
def show_superblock_info(self):
    superblock_data = self.fs.__LeerSuperBloque__()
    superblock_text = (
        f"Nombre: {superblock_data['Nombre']}\n"
        f"Versión: {superblock_data['Versión']}\n"
```

```
            f"Etiqueta de Volumen: {superblock_data['Etiqueta de
            ↪  Volumen']}\n"
            f"Tamaño de Cluster: {superblock_data['Tamaño de Cluster']}\n"
            f"Número de Clusters de Directorio: {superblock_data['Número
            ↪  de Clusters de Directorio']}\n"
            f"Total de Clusters: {superblock_data['Total de Clusters']}"
        )
        self.superblock_info.config(text=superblock_text)
```

2.3.4 Copiar a PC

Esta clase unicamente obtiene los archivos seleccionados a copiar de la interfaz gráfica, tiene una función interna que unicamente sirve para enlazarla a los hilos, cada archivo que se le pasa a esta función funge como un hilo que es trabajado por el sistema de archivos. Se trabaja una lista de hilos para poder iniciarlos 'Al mismo tiempo'.

```
    def copy_to_pc(self):
        selected_item = self.tree.selection()
        if not selected_item:
            messagebox.showwarning("Atención", "Selecciona un archivo para
            ↪  copiar.")
            return

        ↪  #-------------------------------------------------------------------------
        def CopiarAPC(NombreArchivoACopiar,DireccionAGuardar):

            ↪  self.fs.__CopiarDelDisk__(NombreArchivoACopiar,DireccionAGuardar)

        ↪  #-------------------------------------------------------------------------
        for item in selected_item:
            NombreArchivoACopiar = self.tree.item(item)["values"][0]
            DireccionAGuardar =
            ↪  filedialog.asksaveasfilename(initialfile=NombreArchivoACopiar)
            if not DireccionAGuardar:
                continue
            hilo = threading.Thread(target=CopiarAPC,
            ↪  args=(NombreArchivoACopiar,DireccionAGuardar))
            hilo.start()
```

2.3.5 Copiar al sistema de archivos

Esta función solo obtiene los archivos de la selección de la computadora, obtiene su dirección, verifica la longitud del nombre del archivo y finalmente cada archivo realiza su procedimiento como un hilo independiente.

```
def copy_to_fs(self):
        hilos = []
        file_paths= filedialog.askopenfilenames()

        ↪  #-------------------------------------------------------------------------
```

```
        def CopiarAFS(file_path):
            fs.__CopiarAlDisk__(file_path)

        ↪  #-------------------------------------------------------------------------
        if not file_paths:
            return
        for file_path in file_paths:
            NombreArchivo = os.path.basename(file_path)
            if len(NombreArchivo) > 15:
                messagebox.showerror("Error", f"El nombre del archivo es
                ↪  demasiado largo '{NombreArchivo}' (máx. 15
                ↪  caracteres).")
                continue
            hilo = threading.Thread(target=CopiarAFS, args=(file_path,))
            hilos.append(hilo)
        for hilo in hilos:
            hilo.start()
        hilos.clear()
```

2.3.6 Eliminar Archivos

Esta función obtiene la selección de la GUI, le pide al usuario una confirmación de eliminación, si todo procede entonces ejecuta el proceso de eliminación para cada archivo como un hilo independiente; aquí es vital manejar los hilos con la lista pues si se ejecutan mientras navegamos por los items seleccionados de la GUI, el proceso de eliminación puede alterar esta lista, dando errores.

```
    def delete_file(self):
        hilos = []
        selected_item = self.tree.selection()
        if not selected_item:
            messagebox.showwarning("Atención", "Selecciona un archivo para
            ↪  eliminar.")
            return

        ↪  #-------------------------------------------------------------------
        def EliminarArchivo(NombreArchivoAEliminar):
            self.fs.__EliminarDelDisk__(NombreArchivoAEliminar)

        ↪  #-------------------------------------------------------------------

        for item in selected_item:
            NombreArchivoAEliminar = self.tree.item(item)["values"][0]
            confirmacion = messagebox.askyesno("Confirmar", f"¿Estás
            ↪  seguro de que deseas eliminar el archivo
            ↪  '{NombreArchivoAEliminar}'?")
            if confirmacion:
                hilo = threading.Thread(target=EliminarArchivo,
                ↪  args=(NombreArchivoAEliminar,))
                hilos.append(hilo)
```

```
        for hilo in hilos:
            hilo.start()
        hilos.clear()
```

2.4 Sobre el uso de hilos

El programa usa el patrón principalmente de exclusión mutua (Mutex) pues así maneja el uso del recurso compartido que en este caso es el sistema de archivos, así las funciones entre si del sistema de archivos no se entorpecen, bloqueen, o sobrescriben datos. Ademas para la GUI se utiliza una variable de condición, donde un hilo es notificado si sé requiere, que en este caso se listen los archivos.

3 Ejecución

El programa no requiere un comando especial para su ejecución, solo que el sistema de archivos este localizado una carpeta antes de donde esta el programa.
Al ejecutarse se muestra lo siguiente:

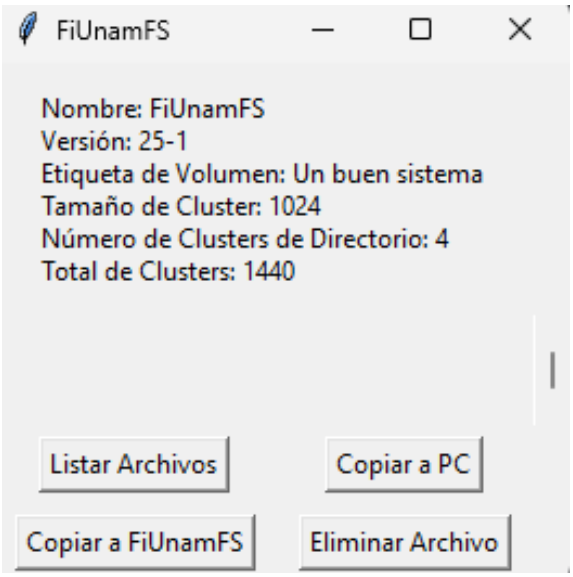

Figura 3.1: Programa al iniciarse

Muestra los datos del superbloque, si alguno de los datos es incorrecto, es decir, la versión o el nombre no son correctos entonces el programa muestra un mensaje de alerta y se cierra.
Por suerte en el proceso de creacion de este proyecto logre corromper un sistema de archivos, asi que el mensaje que muestra es el siguiente:

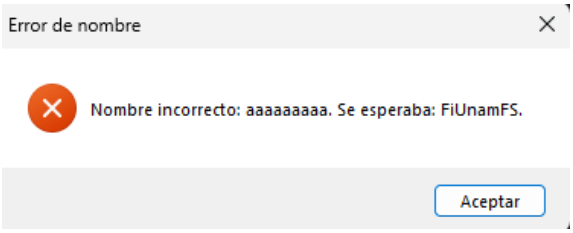

Figura 3.2: Nombre incorrecto

Si se da clic en listar archivos la GUI pasa a verse así:

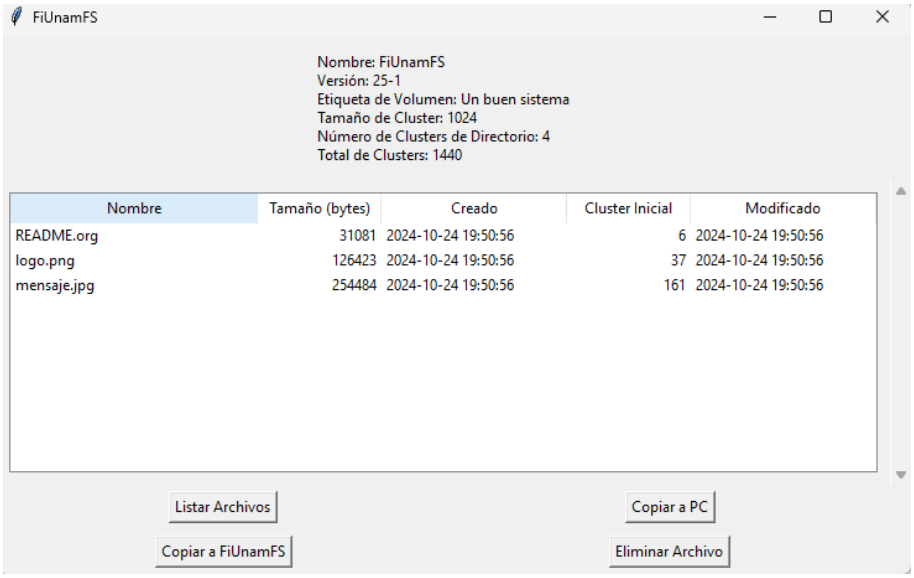

Figura 3.3: Listar Archivos

En este caso muestra los archivos disponibles y sus metadatos.
Para seleccionar archivos simplemente hacemos clic en ellos, si usamos ctrl + clic podremos seleccionar más de uno, viéndose de la siguiente manera:

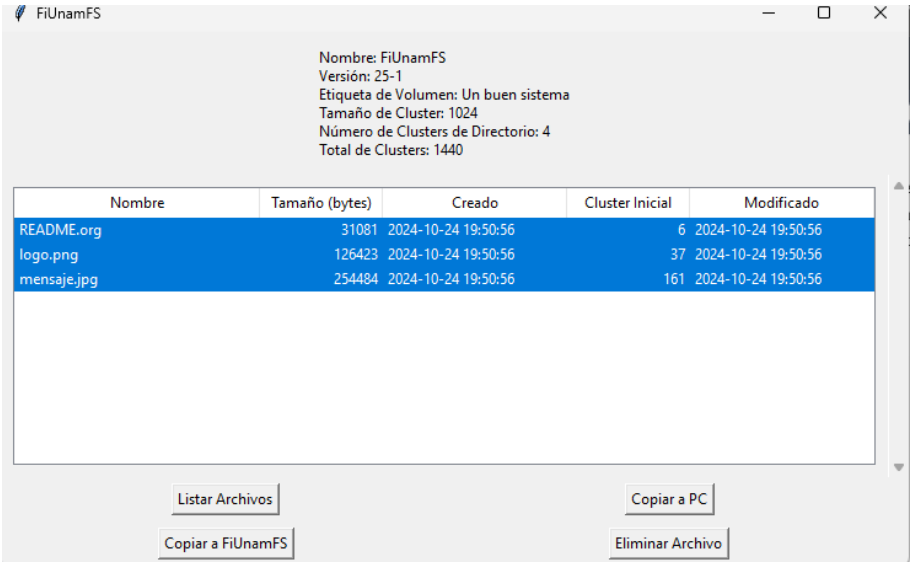

Figura 3.4: Selección de archivos

Ahora si le damos a Copiar a PC salta la siguiente ventana:

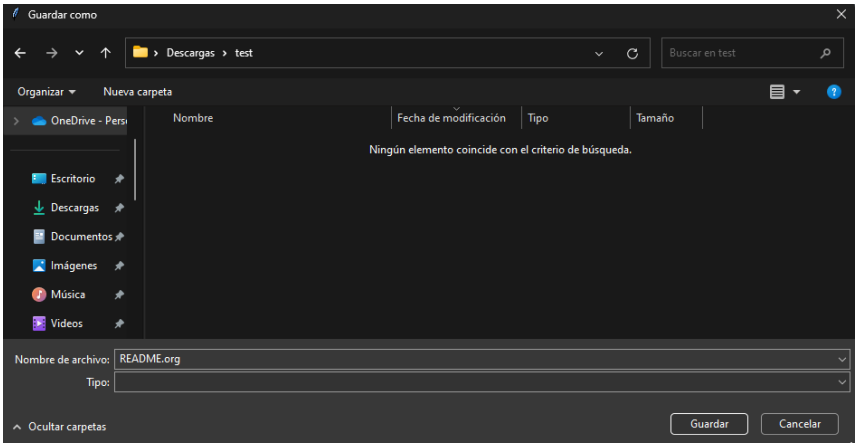

Figura 3.5: Ventana del primer archivo detectado

Esta ventana saltara para cada uno de los archivos seleccionados, para asi seleccionar la ruta en la que se quiere guardar a cada uno. En este caso los guardaremos todos en la carpeta test.

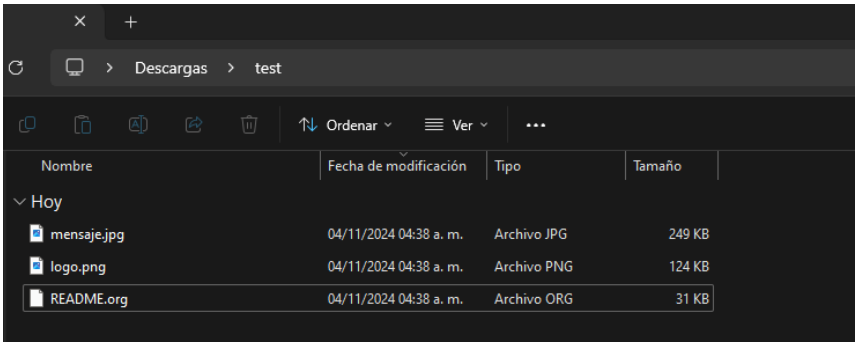

Figura 3.6: Archivos guardados en test

Abrimos los archivos, esperando que no haya alguna corrupción en el copiado:

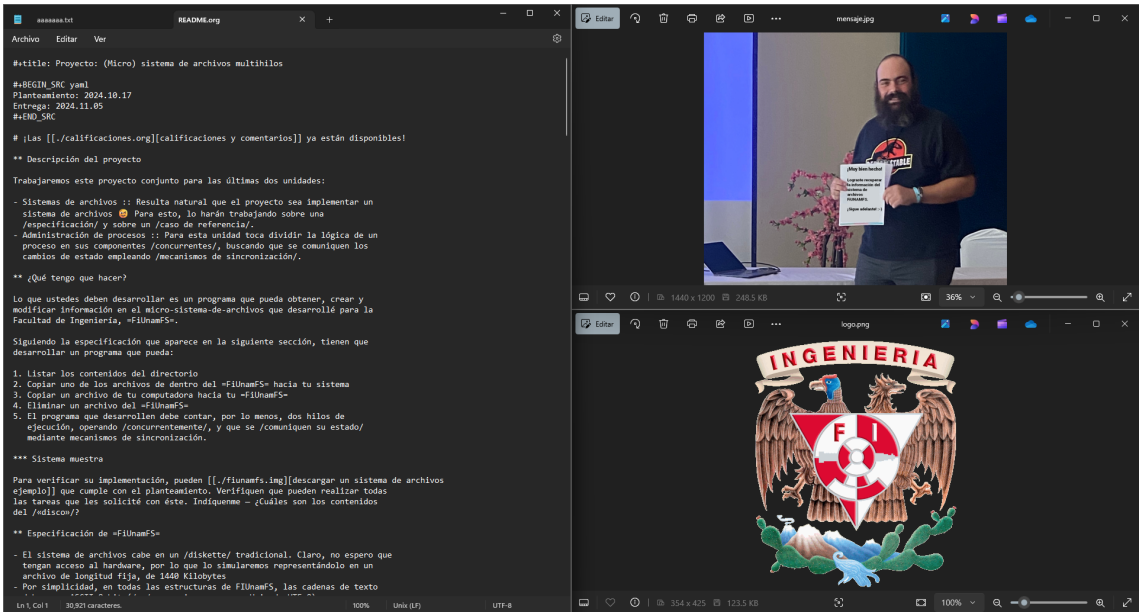

Figura 3.7: Archivos copiados de FiUnamFS

Todo se aprecia correctamente.

Ahora si seleccionamos los mismos archivos y le damos a eliminar:

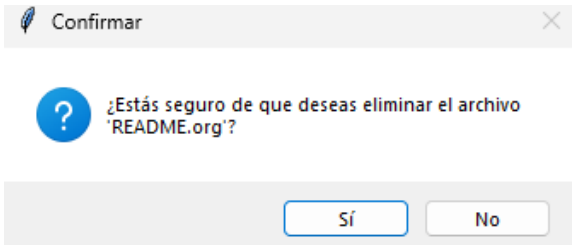

Figura 3.8: Confirmación de eliminado

Solicitara una confirmación por cada archivo

Finalmente indicara la eliminación exitosa para cada archivo:

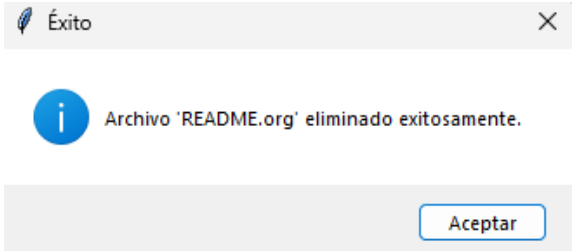

Figura 3.9: Eliminación Exitosa

Al ser multihilo los archivos se eliminaran todos casi al mismo tiempo, a pesar de que los mensajes de eliminación exitosa aparezcan después:

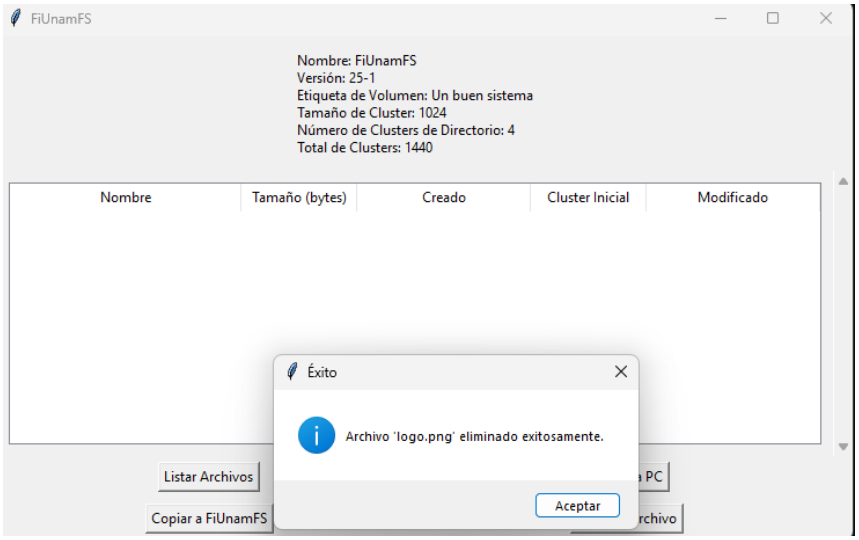

Figura 3.10: Eliminacion Exitosa

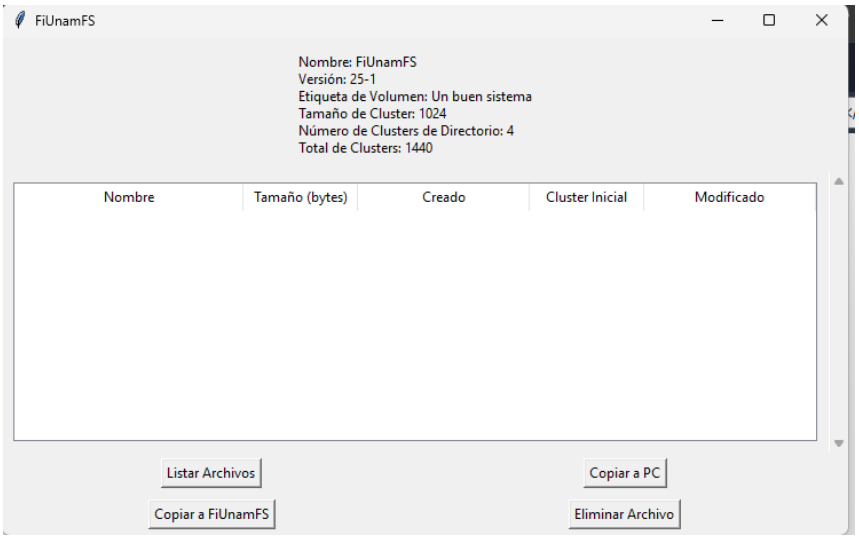

Figura 3.11: Archivos Eliminados

Ahora si copiamos archivos a FIUnamFS

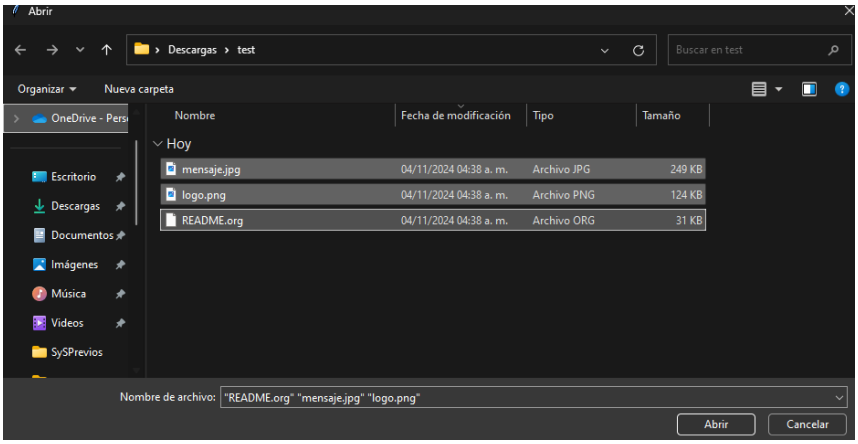

Figura 3.12: Archivos a copiar A FiUnamFS

Al hacer la selección múltiple y darle en aceptar resulta en:

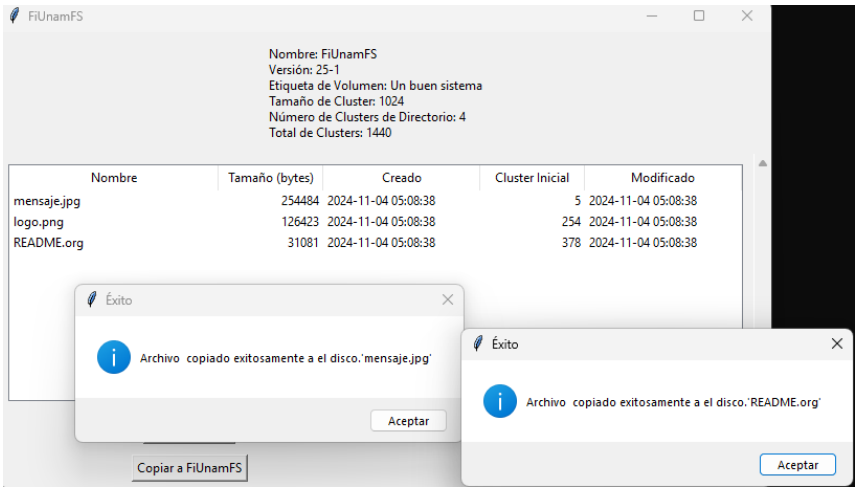

Figura 3.13: Aceptar

Al ser multihilo la GUI se actualizara y mostrara los archivos que se copiaron a pesar de que los mensajes de alerta aparezcan después.
Aquí cabe aclarar una cosa, los clusters iniciales aparecen distintos por como se guardan los archivos y en segunda, los archivos originales por alguna razón se saltaban el cluster 5, siendo que el superbloque era el cluster 0 y el directorio del 1 al 4. Pero si analizamos mi programa realiza los cálculos de cluster correctamente.

Si un archivo excede el limite de caracteres se ve el siguiente error:

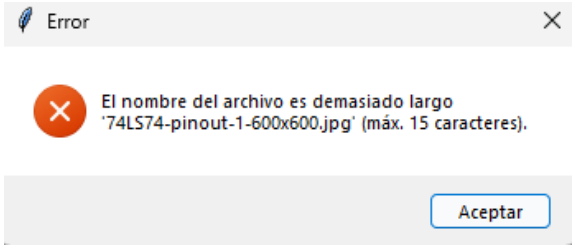

Figura 3.14: Error Limite de caracteres

Ahora si llenamos el sistema de archivos con muchos archivos este pondrá los que pueda y "Salgan"primero y los que ya no quepan saltara un error, indicando que archivo no cupo.

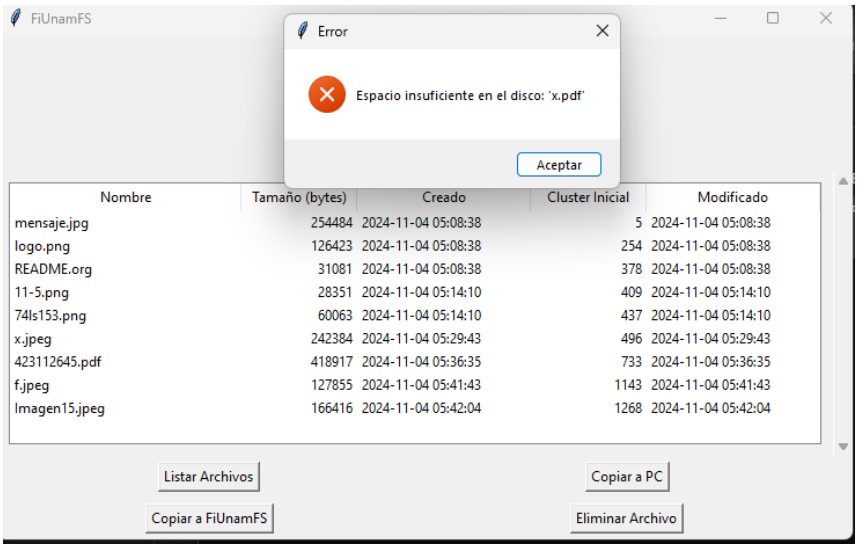

Figura 3.15: Sistema de Archivos lleno

Por ejemplo aquí si realizamos los cálculos, el espacio total es de:

$$Total = 1440 * 1024 = 1474560 \tag{3.1}$$

y considerando el cluster de superbloque y los cuatro de directorio:

$$Reservado = 5 * 1024 = 5120 \tag{3.2}$$

Realizando la resta tenemos que el espacio disponible en bytes es:

$$EspacioDisp = 13,466 \tag{3.3}$$

y tomando en cuenta que x.pdf media 379 KB obviamente no habia espacio para el. El programa también tiene una medida si el directorio esta lleno pero al ser tantas entradas, no tengo tantos archivos con nombres diferentes y de poco espacio para probarlo.
PD: Si se le da a copiar a FiUnamFs sin haber listado los archivos previamente, aun así la función se ejecutara correctamente y posteriormente se listaran los archivos. Las demás funciones requieren si o si que se listen los archivos pues asi es la unica manera de poder elegir los archivos que ocupan estas funciones.