Universidad Nacional Autónoma de México

Facultad de Ingeniería

Sistemas Operativos

Proyecto 2: (Micro) sistema de archivos multihilos

Ávila Martínez Alonso
320237988
Medina Villa Samuel
320249538

Fecha de entrega: 20 de noviembre de 2025

Profesor: Gunnar Eyal Wolf Iszaevich

1. Introducción

1.1. Objetivos

El presente proyecto tiene como objetivo implementar un micro sistema de archivos con las siguientes caracterısticas:

1. Listar los contenidos del directorio del sistema de archivos FiUnamFS
2. Copiar uno o varios archivos desde FiUnamFS hacia el sistema operativo
3. Copiar uno o varios archivos desde el sistema operativo hacia FiUnamFS
4. Eliminar uno o varios archivos del FiUnamFS
5. Contar con múltiples hilos de ejecución operando concurrentemente, comunicándose mediante mecanismos de sincronización
6. Implementar un monitor de integridad que verifique el estado del sistema de archivos en segundo plano

1.2. Entorno de desarrollo

Este programa se trabajó en el IDE Visual Studio Code, utilizando el lenguaje de programación Python en su versión 3.14.0. Se emplearon las siguientes bibliotecas:

1. os - Operaciones con el sistema operativo
2. struct - Empaquetado y desempaquetado de datos binarios
3. threading - Manejo de hilos y sincronizacı´on (Thread, RLock, Condition, Event)
4. datetime - Manejo de fechas y timestamps
5. tkinter - Interfaz gráfica de usuario (ttk, filedialog, messagebox, simpledialog)
6. time - Funciones de tiempo
7. math - Operaciones matemáticas

Todas estas bibliotecas forman parte de la biblioteca estándar de Python, excepto Tkinter que en algunos sistemas operativos puede requerir instalación manual.

2. Arquitectura del sistema

El código se organiza en múltiples capas para separar responsabilidades:

2.1. Clase DiscoVirtual

Esta clase proporciona una abstracción sobre el archivo de imagen del sistema de archivos, implementando operaciones de lectura y escritura protegidas con un RLock

```
class DiscoVirtual:
    SECTOR_SIZE = 512
    CLUSTER_SIZE = 1024
    SUPERBLOCK_CLUSTER = 0
    DIR_START_CLUSTER = 1
    DIR_CLUSTERS = 4  # Clusters 1-4 para directorio
    TOTAL_SIZE = 1440 * 1024  # 1440 KB

    def __init__(self, path):
        self.path = path
        self.lock = RLock()
```

Las constantes definen la estructura física del disco, tamaño de cluster de 1024 bytes, superbloque en cluster 0, directorio en clusters 1-4, y tamaño total de 1440 KB equivalente a un disquete tradicional.

Lectura de cluster: Esta función lee un cluster completo del disco.
El uso del RLock garantiza que no haya condiciones de carrera cuando múltiples hilos intentan leer simultáneamente.

```
def leer_cluster(self, cluster_num):
    with self.lock:
        with open(self.path, 'rb') as f:
            f.seek(cluster_num * self.CLUSTER_SIZE)
            return f.read(self.CLUSTER_SIZE)
```

Escritura de cluster: Similar a la lectura, pero permite modificar el contenido del disco.
Esta operación también está protegida por el lock.

```
def escribir_cluster(self, cluster_num, data):
    with self.lock:
        with open(self.path, 'r+b') as f:
            f.seek(cluster_num * self.CLUSTER_SIZE)
            f.write(data)
```

Operaciones con múltiples clusters: Para mejorar el rendimiento, se implementaron funciones que leen o escriben múltiples clusters contiguos en una sola operación. Esta función es especialmente útil al extraer archivos grandes, reduciendo el número de operaciones de E/S.

```python
def leer_multiples_clusters(self, cluster_inicial, num_clusters):
    """Lee múltiples clusters contiguos"""
    with self.lock:
        data = b''
        for i in range(num_clusters):
            data += self.leer_cluster(cluster_inicial + i)
        return data

def escribir_multiples_clusters(self, cluster_inicial, data):
    """Escribe datos en múltiples clusters contiguos"""
    with self.lock:
        num_clusters = math.ceil(len(data) / self.CLUSTER_SIZE)
        for i in range(num_clusters):
            inicio = i * self.CLUSTER_SIZE
            fin = min(inicio + self.CLUSTER_SIZE, len(data))
            cluster_data = data[inicio:fin]

            if len(cluster_data) < self.CLUSTER_SIZE:
                cluster_data += b'\x00' * (self.CLUSTER_SIZE - len(cluster_data))
            self.escribir_cluster(cluster_inicial + i, cluster_data)
```

2.2. Clase Superbloque

El superbloque contiene información crítica sobre el sistema de archivos. Su validación es esencial para garantizar la integridad del sistema.

```python
class Superbloque:
    def __init__(self, disco):
        self.disco = disco
        self._cargar_y_validar()
```

Carga y validación: Esta función lee el superbloque y verifica que corresponda a un sistema FiUnamFS válido. Utiliza struct.unpack para interpretar los bytes según la especificación del formato FiUnamFS. Si alguna validación falla, se lanza una excepción que impide el uso del sistema de archivos.

```python
def _cargar_y_validar(self):
    sb_data = self.disco.leer_cluster(0)

    try:
        nombre, version, etiqueta, tam_cluster, dir_clusters, total_clusters = struct.unpack(
            '<9s1x5s5x16s4xI1xI1xI', sb_data[:54]
        )

        self.nombre = nombre.decode('ascii').strip('\x00')
        self.version = version.decode('ascii').strip('\x00')
        self.label = etiqueta.decode('ascii').strip('\x00')
        self.cluster_size = tam_cluster
        self.dir_clusters = dir_clusters
        self.total_clusters = total_clusters

    except:
        raise ValueError("Error leyendo superbloque")

    if self.nombre != 'FiUnamFS':
        raise ValueError(f"Sistema de archivos no válido: {self.nombre}")


    if self.version != '26-1':
        raise ValueError(f"Versión incorrecta: esperada 26-1, encontrada {self.version}")
```

Las validaciones verifican que el nombre sea “FiUnamFS” y que la versión corresponda a “26-1”. Hicimos la prueba con “26-2” como lo menciona, pero no funcionó, así que decidimos utilizar una anterior.

2.3. Clase EntradaDirectorio

Representa una entrada individual en el directorio, que puede ser un archivo válido o una entrada vacía. Cada entrada ocupa exactamente 64 bytes y contiene: tipo, nombre, cluster inicial, tamaño, fecha de creación, fecha de modificación y espacio reservado.

Parse: Esta función interpreta los bytes de una entrada del directorio, extrayendo cada campo según las especificaciones del formato.

```python
class EntradaDirectorio:
    SIZE = 64
    EMPTY_MARKER = b'-'
    DELETED_MARKER = b'#'
    VALID_TYPE = b'.'
    EMPTY_NAME = '...............'  # 15 puntos
```

```python
def _parse(self, data):
    self.tipo = chr(data[0])
    self.nombre = data[1:16].decode('ascii', errors='ignore').strip('\x00').strip()
    self.cluster_inicial = struct.unpack('<I', data[16:20])[0]
    self.tamano = struct.unpack('<I', data[20:24])[0]
    self.fecha_creacion = data[24:38].decode('ascii', errors='ignore').strip('\x00')
    self.fecha_modificacion = data[38:52].decode('ascii', errors='ignore').strip('\x00')
```

Verificación de entrada vacía: Una entrada se considera vacía si su tipo es ‘-’ o ‘#’, si el nombre son 15 puntos, o si el nombre está vacío.

```
def is_empty(self):
    return (self.tipo == '-' or
            self.tipo == '#' or
            self.nombre == self.EMPTY_NAME or
            self.nombre == '' or
            '.............' in self.nombre)
```

Serialización: Para escribir una entrada al disco, se convierte a bytes usando struct.pack, asegurando que cada campo tenga el tamaño correcto.

```
def to_bytes(self):
    nombre_enc = self.nombre.encode('ascii')[:15].ljust(15, b'\x00')
    fecha_cr = self.fecha_creacion.encode('ascii')[:14].ljust(14, b'\x00')
    fecha_mod = self.fecha_modificacion.encode('ascii')[:14].ljust(14, b'\x00')

    return struct.pack('<c15sII14s14s12s',
                       self.tipo.encode('ascii') if self.tipo else b'.',
                       nombre_enc,
                       self.cluster_inicial,
                       self.tamano,
                       fecha_cr,
                       fecha_mod,
                       b'\x00' * 12)
```

2.4. Clase Directorio

Gestiona todas las entradas del directorio, que ocupan los clusters 1-4, permitiendo hasta 64 archivos (16 entradas por cluster × 4 clusters).

Carga de entradas: Al inicializar, se leen todas las 64 entradas posibles recorriendo los 4 clusters del directorio.

Listar archivos válidos: Filtra y retorna solo las entradas que representan archivos válidos, excluyendo las entradas vacías o eliminadas.

Búsqueda y actualización: El directorio también proporciona métodos para buscar archivos por nombre, encontrar entradas libres y actualizar entradas existentes en el disco.

```python
class Directorio:
    def __init__(self, disco):
        self.disco = disco
        self.lock = RLock()
        self.entradas = self._cargar_entradas()

    def _cargar_entradas(self):
        entradas = []
        # Clusters 1-4 para directorio
        for cluster in range(1, 5):
            for entrada_idx in range(16):
                offset = (cluster * DiscoVirtual.CLUSTER_SIZE) + (entrada_idx * EntradaDirectorio.SIZE)
                data = self.disco.leer_bytes(offset, EntradaDirectorio.SIZE)
                entrada = EntradaDirectorio(data)
                idx_global = (cluster - 1) * 16 + entrada_idx
                entradas.append((idx_global, entrada))
        return entradas
```

```python
def listar_archivos(self):
    with self.lock:
        return [(idx, e) for idx, e in self.entradas if not e.is_empty()]

def buscar_por_nombre(self, nombre):
    with self.lock:
        for idx, entrada in self.entradas:
            if entrada.nombre == nombre and not entrada.is_empty():
                return idx, entrada
        return None, None
```

```python
def actualizar_entrada(self, indice, entrada):
    with self.lock:
        cluster = 1 + (indice // 16)
        entrada_en_cluster = indice % 16
        offset = (cluster * DiscoVirtual.CLUSTER_SIZE) + (entrada_en_cluster * EntradaDirectorio.SIZE)

        self.disco.escribir_bytes(offset, entrada.to_bytes())
        self.entradas[indice] = (indice, entrada)
```

2.5. Clase GestorEspacio

Administra el espacio disponible y asigna clusters para nuevos archivos mediante un algoritmo de búsqueda de espacio contiguo.

Búsqueda de espacio contiguo: Esta función implementa un algoritmo de búsqueda lineal para encontrar clusters libres contiguos. El algoritmo recorre secuencialmente los clusters desde el final del directorio, buscando bloques completamente llenos de ceros.

Mantiene un contador de clusters consecutivos libres y retorna el cluster inicial cuando encuentra suficiente espacio contiguo.

```python
def buscar_espacio_contiguo(self, num_clusters):
    """Busca espacio contiguo para num_clusters"""
    with self.lock:
        inicio = DiscoVirtual.DIR_START_CLUSTER + DiscoVirtual.DIR_CLUSTERS
        total = self.total_clusters

        clusters_libres_consecutivos = 0
        cluster_inicial = None

        for c in range(inicio, total):
            data = self.disco.leer_cluster(c)
            if all(b == 0 for b in data):
                if cluster_inicial is None:
                    cluster_inicial = c
                clusters_libres_consecutivos += 1

                if clusters_libres_consecutivos == num_clusters:
                    return cluster_inicial
            else:
                clusters_libres_consecutivos = 0
                cluster_inicial = None

        return None
```

Liberación y consulta de espacio: Al eliminar un archivo, sus clusters se marcan como libres escribiendo ceros. El gestor también proporciona información detallada sobre el espacio disponible en el disco.

```python
def liberar_clusters(self, cluster_inicial, num_clusters):
    """Libera clusters escribiendo ceros"""
    with self.lock:
        for i in range(num_clusters):
            self.disco.escribir_cluster(cluster_inicial + i, b'\x00' * DiscoVirtual.CLUSTER_SIZE)

def obtener_espacio_disponible(self):
    """Retorna información de espacio disponible"""
    with self.lock:
        inicio = DiscoVirtual.DIR_START_CLUSTER + DiscoVirtual.DIR_CLUSTERS
        total = self.total_clusters
        clusters_libres = 0
        bytes_libres = 0

        for c in range(inicio, total):
            data = self.disco.leer_cluster(c)
            if all(b == 0 for b in data):
                clusters_libres += 1
                bytes_libres += DiscoVirtual.CLUSTER_SIZE

        return {
            'clusters_libres': clusters_libres,
            'bytes_libres': bytes_libres,
            'clusters_totales': total - inicio,
            'bytes_totales': (total - inicio) * DiscoVirtual.CLUSTER_SIZE
        }
```

2.6. Clase FileSystemOps

Coordina todas las operaciones del sistema de archivos y maneja la sincronización entre hilos. Esta clase integra todas las capas anteriores y proporciona una interfaz para las operaciones.
La variable de condición cambio notificación permite que los hilos se notifiquen mutuamente cuando el directorio cambia.

Listar archivos: Recarga el directorio desde el disco y retorna los archivos válidos, garantizando que la información esté actualizada.

```
def listar(self):
    with self.lock_ops:
        self.directorio.recargar()
        return self.directorio.listar_archivos()
```

Extraer archivo: Copia un archivo del sistema FiUnamFS al sistema operativo. La función busca el archivo en el directorio, calcula cuántos clusters ocupa, lee los datos y los escribe en el destino, truncando el padding de ceros al final.

```
def extraer_archivo(self, nombre_fs, ruta_destino):
    """Extrae archivo del FS a la computadora"""
    with self.lock_ops:
        idx, entrada = self.directorio.buscar_por_nombre(nombre_fs)

        if entrada is None:
            raise FileNotFoundError(f"Archivo '{nombre_fs}' no encontrado")

        clusters_necesarios = math.ceil(entrada.tamano / DiscoVirtual.CLUSTER_SIZE)

        data = self.disco.leer_multiples_clusters(
            entrada.cluster_inicial,
            clusters_necesarios
        )

        data = data[:entrada.tamano]

        if os.path.isdir(ruta_destino):
            ruta_destino = os.path.join(ruta_destino, nombre_fs)

        with open(ruta_destino, 'wb') as f:
            f.write(data)

        return ruta_destino
```

Agregar archivo: Copia un archivo del sistema operativo al FiUnamFS. Esta es la operación más compleja, que realiza múltiples validaciones: verifica la longitud del nombre, confirma que no exista un archivo con el mismo nombre, calcula el espacio requerido, verifica espacio disponible, busca una entrada libre en el directorio, busca espacio contiguo en el disco, escribe los datos, actualiza el directorio y notifica el cambio a otros hilos.

```
def agregar_archivo(self, ruta_origen, nombre_fs=None):
    """Agrega archivo al FS"""
    with self.lock_ops:
        if nombre_fs is None:
            nombre_fs = os.path.basename(ruta_origen)

        if len(nombre_fs) > 15:
            raise ValueError("Nombre demasiado largo (máx 15 caracteres)")

        idx_existente, _ = self.directorio.buscar_por_nombre(nombre_fs)
        if idx_existente is not None:
            raise ValueError(f"Ya existe un archivo con el nombre '{nombre_fs}'")

        with open(ruta_origen, 'rb') as f:
            contenido = f.read()
```

…..

Eliminar archivo: Elimina un archivo liberando sus clusters y marcando su entrada como disponible. Al finalizar, notifica a los hilos observadores para que actualicen su vista del directorio.

```
def eliminar_archivo(self, nombre_fs):
    """Elimina archivo del FS"""
    with self.lock_ops:
        idx, entrada = self.directorio.buscar_por_nombre(nombre_fs)

        if entrada is None:
            raise FileNotFoundError(f"Archivo '{nombre_fs}' no encontrado")

        clusters_a_liberar = math.ceil(entrada.tamano / DiscoVirtual.CLUSTER_SIZE)

        self.gestor_espacio.liberar_clusters(entrada.cluster_inicial, clusters_a_liberar)

        self.directorio.marcar_como_libre(idx)

        with self.cambio_notificacion:
            self.cambio_notificacion.notify_all()
```

2.7. Clase OperacionConcurrente

Encapsula una operación en un hilo independiente con callbacks para éxito y error.

Esta clase permite ejecutar operaciones del sistema de archivos de forma asíncrona sin bloquear la interfaz gráfica.

```python
class OperacionConcurrente(Thread):
    def __init__(self, operacion, args, callback_exito=None, callback_error=None):
        super().__init__()
        self.operacion = operacion
        self.args = args
        self.callback_exito = callback_exito
        self.callback_error = callback_error
        self.daemon = True

    def run(self):
        try:
            resultado = self.operacion(*self.args)
            if self.callback_exito:
                self.callback_exito(resultado)
        except Exception as e:
            if self.callback_error:
                self.callback_error(str(e))
```

2.8. Clase MonitorIntegridad

Un hilo en segundo plano que verifica periódicamente el estado del sistema de archivos. El monitor ejecuta dos tareas importantes:

- Valida el superbloque para detectar corrupciones
- Actualiza la información de espacio disponible.

El uso de Event permite una terminación limpia del hilo.

```python
class MonitorIntegridad(Thread):
    def __init__(self, fs_ops, gui_callback=None, intervalo=30):
        super().__init__(daemon=True)
        self.fs_ops = fs_ops
        self.gui_callback = gui_callback
        self.intervalo = intervalo
        self.activo = True
        self.evento_parada = Event()

    def run(self):
        while self.activo:
            if self.evento_parada.wait(self.intervalo):
                break

            try:
                _ = Superbloque(self.fs_ops.disco)
                info_espacio = self.fs_ops.obtener_info_espacio()

                if self.gui_callback:
                    self.gui_callback(info_espacio)

            except Exception as e:
                if self.gui_callback:
                    self.gui_callback({'error': str(e)})

    def detener(self):
        self.activo = False
        self.evento_parada.set()
```

3. Uso de hilos y sincronización

3.1. Exclusión mutua con RLock

Cada capa utiliza un RLock para proteger el acceso a recursos compartidos. Se utiliza RLock en lugar de Lock simple porque permite que un mismo hilo adquiera el lock múltiples veces, útil para funciones que se llaman entre sí. Los locks principales son: DiscoVirtual.lock que protege operaciones de lectura/escritura en el archivo, Directorio.lock que protege la caché de entradas del directorio, GestorEspacio.lock que protege operaciones de asignación de clusters, y FileSystemOps.lock ops que protege operaciones completas del sistema de archivos.

3.2. Variables de condición

La variable cambio notificación implementa el patrón productor-consumidor. Cuando se agrega o elimina un archivo, se notifica a todos los hilos esperando, causando que la GUI se actualice automáticamente.

```
with self.cambio_notificacion:
    self.cambio_notificacion.notify_all()
```

Hilo productor

```
def _monitor_cambios(self):
    """Hilo para actualizar la GUI cuando hay cambios"""
    while True:
        with self.fs.cambio_notificacion:
            self.fs.cambio_notificacion.wait()
            self.root.after(100, self.actualizar_lista)
```

Hilo consumidor

3.3. Event para terminación

El monitor de integridad utiliza un Event para permitir su terminación ordenada. El hilo espera con timeout, lo que permite interrupciones limpias al cerrar la aplicación.

```
if self.evento_parada.wait(self.intervalo):
    break
```

Espera con timeout, permitiendo interrupciones

```
def detener(self):
    self.activo = False
    self.evento_parada.set()
```

4. Interfaz gráfica

4.1. Clase AplicacionFS

La GUI está construida con Tkinter y se comunica con el sistema de archivos a través de hilos.

Inicialización: Se inician dos hilos auxiliares al crear la aplicación: el monitor que verifica integridad y el hilo de actualización que escucha notificaciones de cambios en el directorio.

```
def __init__(self, root, fs_ops):
    self.root = root
    self.fs = fs_ops
    self.monitor = MonitorIntegridad(fs_ops, self.actualizar_info_espacio)

    # Hilo para actualización automática
    self.hilo_actualizacion = Thread(target=self._monitor_cambios, daemon=True)

    self.root.title("FiUnamFS Manager")
    self.root.geometry("900x600")

    self._crear_interfaz()
    self.monitor.start()
    self.hilo_actualizacion.start()

    # Actualizar lista al inicio
    self.root.after(100, self.actualizar_lista)
```

Monitor de cambios: Este hilo espera notificaciones y actualiza la GUI de forma segura. El uso de root.after() es crucial porque las operaciones de GUI deben ejecutarse en el hilo principal de Tkinter.

```
def _monitor_cambios(self):
    """Hilo para actualizar la GUI cuando hay cambios"""
    while True:
        with self.fs.cambio_notificacion:
            self.fs.cambio_notificacion.wait()
            self.root.after(100, self.actualizar_lista)
```

Actualización de lista: Refresca la visualización mostrando solo archivos válidos, eliminando las entradas vacías del directorio.

```python
def actualizar_lista(self):
    """Actualiza la lista mostrando SOLO archivos válidos"""
    # Limpiar lista
    for item in self.tree.get_children():
        self.tree.delete(item)

    # Obtener y mostrar solo archivos válidos
    archivos = self.fs.listar()

    for idx, entrada in archivos:
        # Solo mostramos archivos válidos, NO las entradas vacías
        tamano_fmt = f"{entrada.tamano}"

        # Mostrar fechas en formato AAAAMMDDHHMMSS como están en el disco
        self.tree.insert('', tk.END, values=(
            entrada.nombre,
            tamano_fmt,
            entrada.cluster_inicial,
            entrada.fecha_creacion,
            entrada.fecha_modificacion
        ))

    # Actualizar información del sistema
    self.actualizar_info_sistema()
```

Operaciones múltiples: Las funciones para extraer, agregar y eliminar soportan selección múltiple, ejecutando cada operación en su propio hilo.

```python
def extraer_multiple(self):
    """Extrae múltiples archivos seleccionados"""
    seleccion = self.tree.selection()
```

….

```python
def agregar_multiple(self):
    """Agrega múltiples archivos"""
    archivos = filedialog.askopenfilenames(title="Seleccionar archivo(s)")
```

….

```python
def eliminar_multiple(self):
    """Elimina múltiples archivos seleccionados"""
    seleccion = self.tree.selection()
```

….

5. Ejecución

5.1. Requisitos

Para ejecutar el programa, como se mencionó antes se utilizó Python 3.14.0 con la biblioteca Tkinter instalada, que viene incluida en la mayoría de instalaciones estándar de Python. Además, es necesario contar con un archivo de imagen FiUnamFS válido con extensión .img

5.2. Iniciar programa

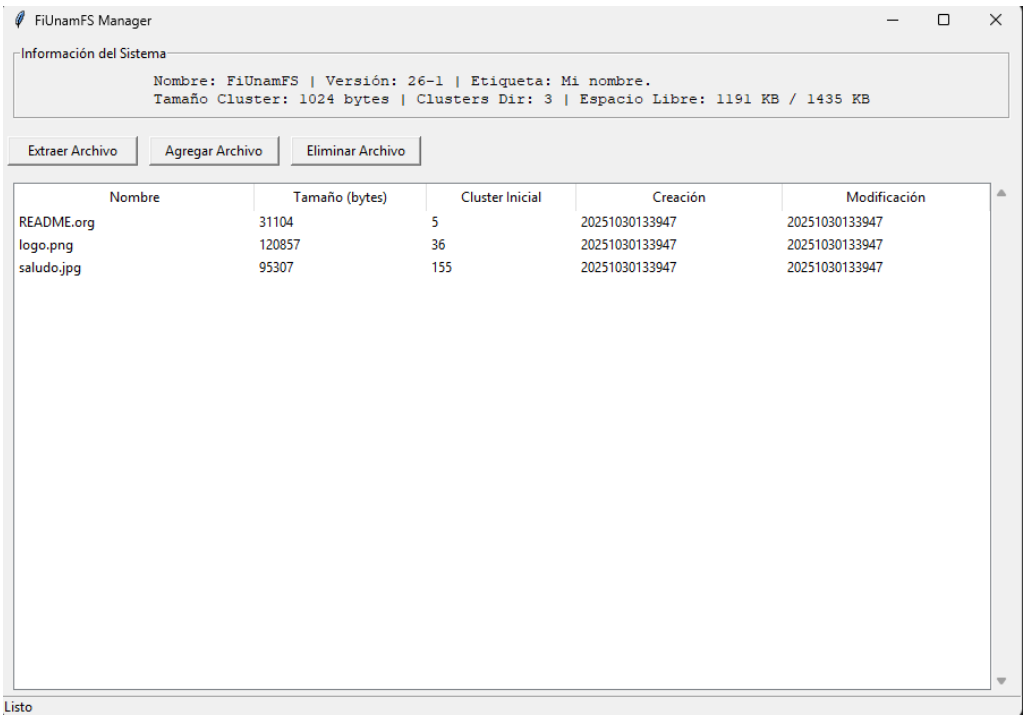
Para ejecutar el programa, se utiliza el comando: python proyecto2.py
Al iniciar, se mostrará un diálogo para seleccionar el archivo del sistema de archivos.

5.3. Validación de sistema de archivos

Si el archivo seleccionado no es un sistema FiUnamFS válido o la versión no corresponde, se mostrará un mensaje de error. Los errores pueden ser por nombre del sistema incorrecto (esperado: FiUnamFS), versión incorrecta (esperada: 26-1).

5.4. Pantalla principal

Una vez validado el sistema de archivos, se muestra la interfaz principal con varios componentes: el panel superior con información del superbloque incluyendo nombre, versión, etiqueta, tamaño de cluster y espacio disponible; los botones de operación para extraer, agregar y eliminar archivos; la lista de archivos en forma de tabla con nombre, tamaño, cluster inicial, fechas de creación y modificación; y la barra de estado que muestra el espacio libre actualizado automáticamente.



5.5. Listar archivos

Los archivos se cargan automáticamente al iniciar. La lista muestra el nombre del archivo con máximo 15 caracteres, el tamaño en bytes, el cluster inicial donde comienza el archivo, y las fechas de creación y modificación en formato AAAAMMDDHHMMSS.
La lista se actualiza automáticamente cuando se agregan o eliminan archivos gracias al sistema de notificaciones entre hilos.

5.6. Extraer archivos

Para copiar archivos del FiUnamFS al sistema operativo, el usuario debe seleccionar uno o más archivos de la lista usando Ctrl+clic para selección múltiple, hacer clic en "Extraer Archivo", seleccionar la carpeta de destino, y los archivos se copiaran con su nombre original.
Si la operación es exitosa, se muestra un mensaje indicando cuántos archivos se copiaron. Si hay errores, se listan los archivos que fallaron con el motivo del error.

5.7. Agregar archivo

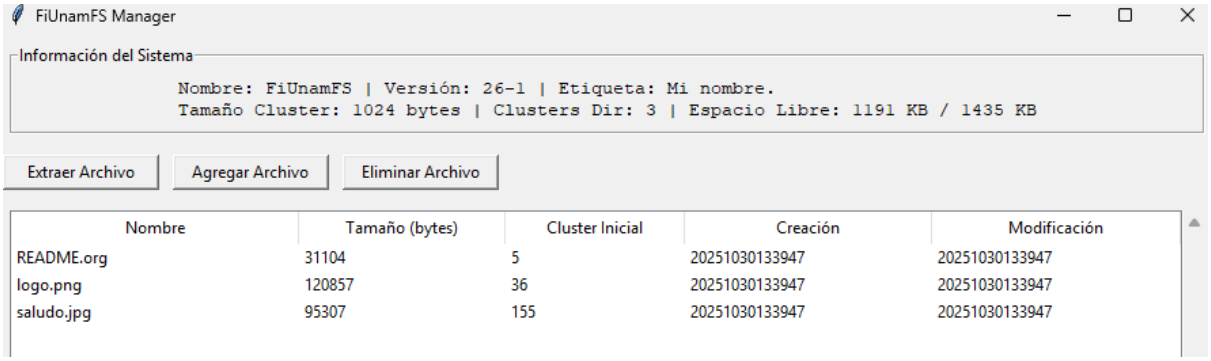
Para copiar archivos del sistema operativo al FiUnamFS, el usuario hace clic en "Agregar Archivo", selecciona uno o m´as archivos (la selección múltiple est´a disponible), y si el nombre excede 15 caracteres, el sistema solicita uno nuevo. Los archivos se copian entonces al sistema FiUnamFS.

5.7.1. Validaciones

El sistema realiza validaciones exhaustivas: la longitud del nombre debe ser máximo 15 caracteres, no pueden existir dos archivos con el mismo nombre, debe haber suficientes clusters libres, debe existir espacio contiguo para el archivo.

5.8. Eliminar archivos

Para eliminar archivos del FiUnamFS, el usuario selecciona uno o más archivos, hace clic en "Eliminar Archivo", confirma la eliminación, y los archivos se eliminan mientras sus clusters se liberan.



5.9. Monitor de espacio

La barra de estado inferior muestra información actualizada automáticamente cada 30 segundos:

- Espacio libre en KB
- Espacio total en KB
- Porcentaje de espacio libre.

Esta actualización ocurre en un hilo independiente sin interrumpir las operaciones del usuario

5.10. Operaciones concurrentes

Cuando se seleccionan múltiples archivos para extraer, agregar o eliminar, cada archivo se procesa en su propio hilo, mantiene la interfaz gráfica responsiva durante operaciones largas, y permite el procesamiento paralelo de archivos independientes.

5.11. Cerrar la aplicación

Al cerrar la ventana, el programa detiene el monitor de integridad de forma ordenada, finaliza todos los hilos pendientes, cierra el archivo del sistema de archivos y libera todos los recursos. No es necesario guardar cambios, ya que todas las operaciones se escriben inmediatamente al disco.

6. Conclusiones

El desarrollo de este proyecto nos permitió aterrizar los diferentes conceptos que vimos durante las diferentes clases.

Por otro lado, la implementación de la concurrencia fue la parte más compleja. Al principio, enfrentamos problemas donde la interfaz gráfica se congelaba al copiar archivos grandes o no mostraban el correcto tamaño de los archivos, etc. Aprendimos que la sincronización no es solo una medida de seguridad para evitar condiciones de carrera, sino una necesidad para garantizar una experiencia de usuario fluida.

Este proyecto nos deja con una sólida comprensión del ciclo de vida de los archivos y de la complejidad oculta detrás de operaciones tan cotidianas que hacemos cómo lo es copiar, pegar y eliminar un archivo.