



Universidad Autónoma de México
Facultad de Ingeniería



Proyecto 2

“(Micro) sistema de archivo multihilos”

Sistemas Operativos

Hernández Saldívar Héctor Saúl
319276017
Miyasaki Sato Yuichi Vicente
318586465

Ing. Gunnar Eyal Wolf Iszaevich

Grupo: 6

Semestre 2025-1

Fecha de entrega:
5 de noviembre del 2024



Índice:

1. Descripción del proyecto.....	2
1.1 Especificaciones de FiUnamFS.....	2
2. Lenguaje y entorno de desarrollo.....	4
3. Estrategia empleada.....	5
3.1 Parámetros del proyecto.....	5
3.2 Sincronización de Operaciones Concurrentes.	5
3.3 SistemaArchivosFiUnamFS.....	6
3.4 Operaciones_directory.....	9
3.5 RutaArchivo.....	13
3.6 ManejoDeHilos.....	14
3.8 Función global para verificar el sistema mediante hilos.....	15
3.7 Características adicionales.....	15
4. Requisitos.....	16
5. Descripción de la sincronización empleada.	16
5.1 Sincronización de Operaciones.....	16
5.2 Clases Relacionadas con Hilos.....	16
5.3 Verificación del Sistema de Archivos.....	17
6. Ejemplos de uso.....	17
6.1 Cursos de sistemas operativos.....	17
6.2 Desarrollo y pruebas de programas en un entorno controlado.....	17
6.3 Carga y recuperación en memoria limitada..	18
7. Referencias Electrónicas.....	18



1. Descripción del proyecto.

Se debe desarrollar un programa que permita la obtención, creación y modificación de la información en un micro-sistema de archivos desarrollado para la Facultad de Ingeniería, **FiUnamFS**.

Siguiendo las especificaciones que se proveerán más adelante, el programa debe de:

- Listar los contenidos del directorio.
- Copiar uno de los archivos de dentro del **FiUnamFS** hacia tu sistema.
- Copiar un archivo de tu computadora hacia tu **FiUnamFS**.
- Eliminar un archivo del **FiUnamFS**.
- El programa a desarrollar debe contar, por lo menos, dos hilos de ejecución, operando concurrentemente, y que se comuniquen su estado por medio de mecanismos de sincronización.

1.1 Especificaciones de **FiUnamFS**.

- El sistema de archivos cabe en un diskette tradicional, lo simularemos representándolo en un archivo de longitud fija, de 1440 Kilobytes[1].
- En todas las estructuras de **FiUnamFS**, las cadenas de texto deben ser ASCII 8-bit[1].
- En las estructuras del disco, todos los enteros serán representados como valores de 32 bits, en formato little endian[1].
- Para hacer las conversiones desde o hacia este formato, se usará el formato <I con las funciones pack() y unpack(), disponible en Python[1].



- La superficie del disco se divide en sectores de 256 bytes. Cada cluster mide cuatro sectores[1].

- El pseudodispositivo no maneja tabla de particiones, sino que hospeda directamente un volumen en la totalidad de su espacio[1].

FiUnamFS maneja únicamente un directorio plano, no se consideran subdirectorios[1].

- El primer cluster #0 del pseudodispositivo es el superbloque. Este contiene información en los siguientes bytes:

➤ 0-8: Para identificación, el nombre del sistema de archivos. Se debe validar el nombre del sistema de archivos, este debe ser la cadena **FiUnamFS**[1].

➤ 10-14: Versión de la implementación. Estamos implementando la versión 25-1. Se debe validar que el sistema de archivos a utilizar sea exactamente esta versión, para evitar la corrupción de datos[1].

➤ 20-35: Etiqueta del volumen[1].

➤ 40-44: Tamaño del cluster en bytes[1].

➤ 45-49: Número de clusters que mide el directorio[1].

➤ 50-54: Número de clusters que mide la unidad completa[1].

- El resto del superbloque puede quedar vacío[1].

- El sistema de archivos es de asignación continua. Toda la información de los archivos está en el directorio[1].

- El directorio está ubicado en los clusters 1 a 4. Cada entrada del directorio mide 64 bytes, consistentes en:

■ 0: Tipo de archivo. Dado que el sistema de archivos actual no tiene soporte para directorios, dispositivos, pipes, ni otros archivos especiales, siempre será el carácter . (0x2e, 46). Cuando la entrada está vacía, se indica con el carácter #(0x23, 35)[1].

■ 1-15:Nombre del archivo[1].

■ 16-20:Tamaño del archivo, en bytes[1].



- 20-23:Cluster inicial[1].
 - 24-37:Hora y fecha de creación del archivo, especificando AAAAMMDDHHMMSS[1].
 - 38-51:Hora y fecha de última modificación del archivo, especificando AAAAMMDDHHMMSS[1].
 - 52-64:Espacio no utilizado (reservado para expansión futura?) [1].
- Las entradas no utilizadas del directorio se identifican porque en el campo de nombre llevan la cadena -----[1].
- Los nombres de archivos pueden componerse de cualquier carácter dentro del subconjunto ASCII de 7 bits[1].
- Es un sistema de archivos plano – No maneja subdirectorios[1].
- Después del directorio, todo el espacio restante es espacio de datos[1].

2. Lenguaje y entorno de desarrollo.

El lenguaje escogido para el desarrollo de este proyecto fue Python en una versión igual o superior a la 3.10, para verificar la versión de python que tengamos instalada en nuestro equipo basta con colocar en nuestra consola el comando python --version. En caso de no tener instalado Python, se debe instalar desde su página oficial.

Para el proyecto, nosotros hicimos uso del editor de texto Visual Studio Code versión 1.95 en Windows 11, junto con la extensión de Python 3.11.9 que proporciona Microsoft. Las bibliotecas que usamos para el proyecto son: os, struct, threading y datetime; todos incluidos en la extensión de Python.



3. Estrategia empleada.

Para la estrategia empleada nosotros decidimos para el proyecto dividir las asignaciones, funcionalidades y procesos en varios componentes o clases clave, los cuales son:

3.1 Parámetros del proyecto.

FiUnamFS simula un sistema de archivos similar a un disquete de 1440 KB. Su estructura se organiza en sectores de 256 bytes y clusters de 1024 bytes, o sea, cuatro sectores por cluster. El primer cluster, el superbloque, almacena información esencial como el nombre del sistema, la versión, y detalles del volumen. Los clusters de 1 a 4 están dedicados al directorio, donde cada archivo tiene una entrada con su nombre, tamaño, tipo, ubicación y fechas de creación y modificación.

```
# Parámetros del proyecto
TAMANO_DISQUETE = 1440 * 1024 # 1440 KB
TAMANO_SECTOR = 256
TAMANO_CLUSTER = 1024 # 4 sectores de 256 bytes
TAMANO_ENTRADA = 64
SUPERBLOQUE_CLUSTER = 0
CLUSTERS_DIRECTORIO = 4
NOMBRE_SISTEMA_ARCHIVOS = 'FiUnamFS'
VERSION_SISTEMA = '25-1'
ARCHIVO_IMAGEN = 'fiunamfs.img'
```

3.2 Sincronización de Operaciones Concurrentes.

Para evitar conflictos en las operaciones concurrentes, implementamos sincronización mediante 2 locks y un semáforo, para sincronizar operaciones usamos: directorio_mutex y lock_verificacion aseguran que solo un hilo acceda al directorio o al proceso de verificación a la vez, mientras que sem_superbloque limita el acceso al superbloque a un hilo.

```
# Sincronización de operaciones concurrentes
directorio_mutex = threading.Lock()
sem_superbloque = threading.Semaphore(1)
lock_verificacion = threading.Lock()
```



3.3 SistemaArchivosFiUnamFS.

Esta clase representa el sistema de archivos. Inicializa con la imagen del sistema de archivos y verifica su existencia, y valida el nombre y la versión del sistema de archivos, esto permite la lectura del directorio y el procesamiento de entradas de archivo. Algunas funciones que tiene la clase son las siguientes:

- **verificar_archivos:** Esta función tiene como objetivo comprobar si el archivo de imagen del sistema de archivos "fiunamfs.img" está presente en el directorio actual. Si el archivo no se encuentra, solicita al usuario que ingrese la ruta completa del archivo. Cabe aclarar que si el usuario no ingresa la ruta del archivo correctamente, nunca podrá entrar al programa.

```
def verificar_archivo(self):
    with lock_verificacion:
        # Verificar si el archivo existe en el directorio actual
        if not os.path.exists(self.imagen_archivo):
            print(f"\n\tEl archivo '{self.imagen_archivo}' no se encuentra en el directorio actual.")

        # Solicitar ruta si el archivo no existe en el directorio actual
        while True:
            ruta = input("\n\tPor favor, ingrese la ruta completa del archivo 'fiunamfs.img': ")
            if os.path.exists(ruta):
                self.imagen_archivo = ruta
                print(f"\tArchivo encontrado en la ruta: {self.imagen_archivo}")
                break
            else:
                CLEAR()
                print("\n\tArchivo no encontrado en la ruta especificada. Intente nuevamente.")
```

- **validar_sistema_archivos:** Tiene el propósito de leer el superbloque del sistema de archivos FiUnamFS y verificar que el nombre y la versión del sistema sean correctos. Para su verificación debemos de tener en cuenta que la función leerá el primer cluster que se encuentra en el archivo, ya que ahí se encuentra la información del



superbloque, el programa leerá los primeros 8 bytes para verificar el nombre del sistema de archivos, y luego verificará la versión del mismo de los bytes 10 al 15.

```
def validar_sistema_archivos(self):
    with sem_superbloque: # Acceso controlado al superbloque
        with open(self.imagen_archivo, 'rb') as img:
            img.seek(0)
            superbloque = img.read(TAMANO_CLUSTER)

        # Leer y validar el nombre y versión del sistema de archivos
        nombre = superbloque[0:8].decode().strip('\x00')
        version = superbloque[10:15].decode().strip('\x00')

        if nombre != NOMBRE_SISTEMA_ARCHIVOS:
            raise ValueError("\tNombre de sistema de archivos incorrecto.")
        if version != VERSION_SISTEMA:
            raise ValueError("\tVersión del sistema de archivos no soportada.")
    print("\tValidación del sistema de archivos completada exitosamente.")
```

- **leer_directorio:** Se encarga de extraer y procesar las entradas del directorio del sistema de archivos FiUnamFS. Antes de realizar cualquier operación, llama al método **validar_sistema_archivos** para asegurarse de que el archivo de imagen corresponda al formato esperado del sistema de archivos. Una vez validado, la función itera a través de los clusters que almacenan la información del directorio, leyendo cada entrada de archivo. Durante este proceso, ignora las entradas vacías b' #' y utiliza un método auxiliar para transformar cada entrada válida en una representación estructurada, que se almacena en una lista. Al finalizar, la función devuelve esta lista, que contiene todos los archivos presentes en el directorio, listos para su posterior manipulación o visualización.



```
def leer_directorio(self):
    directorio = []
    with open(self.imagen_archivo, 'rb') as img:
        self.validar_sistema_archivos()

        for cluster in range(CLUSTERS_DIRECTORIO):
            posicion_inicial = (SUPERBLOQUE_CLUSTER + 1 + cluster) * TAMANO_CLUSTER
            img.seek(posicion_inicial)
            cluster_datos = img.read(TAMANO_CLUSTER)

            for entrada in range(0, TAMANO_CLUSTER, TAMANO_ENTRADA):
                entrada_actual = cluster_datos[entrada:entrada + TAMANO_ENTRADA]

                if entrada_actual[0:1] == b'#':
                    continue # Entrada vacía

                archivo_info = self._procesar_entrada_directorio(entrada_actual)
                directorio.append(archivo_info)

    return directorio
```

- **procesar_entrada_directorio:** Toma una entrada del directorio y la convierte en un formato más útil y estructurado para su manipulación posterior. La entrada del directorio se recibe como un bloque de bytes, que contiene información sobre un archivo específico, incluyendo su tipo, nombre, tamaño, cluster inicial y fechas de creación y modificación. Es un `@staticmethod` porque no necesita acceder a ninguna instancia específica de la clase o sus atributos para realizar su tarea.

```
@staticmethod
def _procesar_entrada_directorio(entrada):
    tipo_archivo = chr(entrada[0])
    nombre_archivo = entrada[1:16].decode().strip('\x00')
    tamaño_archivo = struct.unpack('<I', entrada[16:20])[0]
    cluster_inicial = struct.unpack('<I', entrada[20:24])[0]
    fecha_creacion = entrada[24:38].decode()
    fecha_modificacion = entrada[38:52].decode()

    return {
        'Tipo': tipo_archivo,
        'Nombre': nombre_archivo.strip(),
        'Tamaño': tamaño_archivo,
        'Cluster Inicial': cluster_inicial,
        'Fecha Creación': fecha_creacion,
        'Fecha Modificación': fecha_modificacion
    }
```



3.4 Operaciones directorio.

La clase maneja operaciones relacionadas con el directorio del sistema de archivos FiUnamFS. La clase tiene varias funciones que permiten listar, copiar y eliminar archivos, así como gestionar la interacción con el sistema de archivos. Estos son algunos de sus funciones:

- **listar_directorio:** Muestra los archivos contenidos en el sistema de archivos FiUnamFS. Primero, bloquea el acceso al directorio con un lock para que otras operaciones no interfieran mientras se listan los archivos. Luego, utiliza leer_directorio() de SistemaArchivosFiUnamFS para obtener la lista de archivos en el sistema de archivos. Finalmente, muestra la información de cada archivo, como el nombre, tamaño, fecha de creación y fecha de modificación.

```
def listar_directorio(self):
    with directorio_mutex:
        contenido_directorio = self.sistema_archivos.leer_directorio()

        CLEAR()
        listarDirectorioImp() # imprimir las secciones del contenido del directorio

        for archivo in contenido_directorio:
            print(
                f"\t|{archivo['Tipo']}| {archivo['Nombre']}<15|\t | {archivo['Tamaño']} bytes\t"
                f" |\t {archivo['Cluster Inicial']}\t|\t"
                f" |\t {archivo['Fecha Creación']}\t|\t"
                f" |\t {archivo['Fecha Modificación']}| "
            )
        print("\t-----")
        "-----")
```

- **copiar_de_FiUnamFs:** Copia un archivo desde FiUnamFS al sistema local. Bloquea el directorio para evitar conflictos y busca el archivo en el directorio. Si el archivo existe, el usuario puede elegir el directorio de destino; si no se especifica, se copia en el directorio actual. Para evitar sobreescritura, verifica si existe un archivo con el mismo nombre en el destino, y si es



necesario, añade un sufijo. Luego, copia el contenido desde el archivo de imagen de FiUnamFS al sistema local. Cabe aclarar que al asignarle un sufijo numérico, este usara 3 espacios del nombre original, por ejemplo: hola, hola(1), hola(2), hola(3)... hola(n).

```
def copiar_de_FiUnamFs(self, nombre_archivo):
    # Usar un lock para sincronizar el acceso a la función
    with directorio_mutex:
        contenido_directorio = self.sistema_archivos.leer_directorio()
        archivo_encontrado = None

        # Buscar el archivo en el directorio
        for archivo in contenido_directorio:
            if archivo['Nombre'].startswith(nombre_archivo): # Coincidir solo el nombre
                archivo_encontrado = archivo
                break

        if archivo_encontrado:
            # Preguntar si desea copiar en un directorio específico
            while True:
                respuesta = input("\n\tDeseas copiar el archivo a un directorio específico? (s/n): ").strip().lower()

                if respuesta == 's':
                    while True:
                        ruta_destino = input("\n\tIngresa la ruta completa del directorio de destino (escribe 'menu' para regresar): ").strip()

                        if ruta_destino.lower() == 'menu':
                            CLEAR()
                            print("\n\tRegresando al menú principal...")
                            return # Regresa al menú principal

                        # Verificar si el directorio existe
                        if os.path.exists(ruta_destino):
                            break # Salir del ciclo si el directorio existe
                        else:
                            CLEAR()
                            print("\n\tError: El directorio no existe. Por favor, intenta de nuevo.")
                            break # Salir del bucle externo si 's' fue seleccionado

                elif respuesta == 'n':
                    ruta_destino = os.getcwd() # Usar el directorio actual si no se especifica otro
                    break # Salir del bucle externo

            else:
                CLEAR()
                print("\n\tOpción no válida. Por favor ingresa 's' o 'n'.")
```

Crear la ruta completa del archivo

```
nombre_base, extension = os.path.splitext(archivo_encontrado['Nombre'])
ruta_completa = os.path.join(ruta_destino, archivo_encontrado['Nombre'])

# Verificar si ya existe un archivo con el mismo nombre en el destino
contador = 1
while os.path.exists(ruta_completa):
    ruta_completa = os.path.join(ruta_destino, f"{nombre_base}{{contador}}{extension}")
    contador += 1

# Verificar tamaño de lectura
inicio_lectura = (archivo_encontrado['Cluster Inicial']) * TAMANO_CLUSTER

with open(self.sistema_archivos.imagen_archivo, 'rb') as img:
    img.seek(inicio_lectura)
    data = img.read(archivo_encontrado['Tamaño'])
    with open(ruta_completa, 'wb') as nuevo_archivo:
        nuevo_archivo.write(data)

CLEAR()
print(f"\n\tCopiando archivo de tamaño: {archivo_encontrado['Tamaño']} bytes...\n")
print(f"\tArchivo {archivo_encontrado['Nombre']} copiado a {ruta_completa}")
else:
    CLEAR()
    print(f"\tArchivo '{nombre_archivo}' no encontrado en FiUnamFS.")
```



- **copiar_de_local:** Copia un archivo desde el sistema local hacia FiUnamFS. La función bloquea el directorio para evitar conflictos de acceso y lee el archivo de la ruta local. Calcula la cantidad de clusters necesarios y verifica si hay espacio suficiente en FiUnamFS. Si el nombre del archivo ya existe en el sistema de archivos, le asigna un nombre único. Luego, busca clusters libres, escribe la entrada en el directorio y el contenido en los clusters. Muestra un mensaje de error si no hay espacio suficiente.

Abre la ruta del archivo local y lo lee.

```
with open(ruta_archivo_local, 'rb') as archivo_local:
    data = archivo_local.read()
    tamaño_archivo = len(data)

    # Calcular clusters necesarios para almacenar el archivo
    clusters_necesarios = (tamaño_archivo + TAMANO_CLUSTER - 1) // TAMANO_CLUSTER

    # Leer directorio para verificar espacio disponible
    contenido_directorio = self.sistema_archivos.leer_directorio()

    nombre_base, extension = os.path.splitext(os.path.basename(ruta_archivo_local))

    if (len(nombre_base.strip()) + len(extension)) > 16:
        CLEAR()
        print("\tNo puede ingresar nombres mayores a 15 caracteres...")
        print("\tVolviendo al menú principal...\n")
        return # Regresa al menú principal

    # Generar un nombre único en FiUnamFS si el archivo ya existe
    nombre_base, extension = os.path.splitext(os.path.basename(ruta_archivo_local)[:15])
    nombre_archivo = nombre_base[:15 - len(extension)] + extension # Limitar el nombre a 15 caracteres
    contador = 1
```

Busca clusters disponibles en FiUnamFs.

```
# Buscar clusters libres para almacenar el archivo
clusters_disponibles = []
for cluster in range(CLUSTERS_DIRECTORIO + 1, TAMANO_DISQUETO // TAMANO_CLUSTER):
    ocupado = False
    for archivo in contenido_directorio:
        cluster_inicial = archivo['Cluster Inicial']
        clusters_ocupados = range(cluster_inicial, cluster_inicial + (archivo['Tamaño'] + TAMANO_CLUSTER - 1) // TAMANO_CLUSTER)
        if cluster in clusters_ocupados:
            ocupado = True
            break
    if not ocupado:
        clusters_disponibles.append(cluster)
    if len(clusters_disponibles) == clusters_necesarios:
        break
```

Entrada del directorio y conversión de formato.

```
entrada_directorio = (
    b'.' +
    nombre_archivo.encode().ljust(15, b'\x00') +
    struct.pack('<I', tamaño_archivo) +
    struct.pack('<I', clusters_disponibles[0]) +
    fecha_actual.encode().ljust(14, b'\x00') * 2 +
    b'\x00' * (TAMANO_ENTRADA - 64)
)
```



Abre el sistema de archivos FiUnamFS para insertar la entrada_directorio, o sea, el archivo que seleccionamos del sistema local.

```
with open(self.sistema_archivos.imagen_archivo, 'r+b') as img:
    # Buscar entrada libre en el directorio
    for cluster in range(CLUSTERS_DIRECTORIO):
        posicion_inicial = (SUPERBLOQUE_CLUSTER + 1 + cluster) * TAMANO_CLUSTER
        img.seek(posicion_inicial)
        cluster_datos = img.read(TAMANO_CLUSTER)

        for entrada in range(0, TAMANO_CLUSTER, TAMANO_ENTRADA):
            if cluster_datos[entrada:entrada + 1] == b'#': # Espacio libre
                img.seek(posicion_inicial + entrada)
                img.write(entrada_directorio)
                break
            else:
                continue
        break
    else:
        CLEAR()
        print("\tNo se pudo encontrar espacio en el directorio.")
        return

    # Escribir el archivo en clusters disponibles
    for i, cluster in enumerate(clusters_disponibles):
        posicion_cluster = cluster * TAMANO_CLUSTER
        img.seek(posicion_cluster)
        img.write(data[i * TAMANO_CLUSTER:(i + 1) * TAMANO_CLUSTER])

CLEAR()
print(f"\tArchivo '{nombre_archivo}' copiado a FiUnamFS.")
```

- **eliminar_archivo:** Elimina un archivo del sistema de archivos FiUnamFS. La función adquiere el lock del directorio y recorre las entradas buscando el archivo por su nombre. Una vez localizado, marca la entrada como vacía sobre escribiéndola.

```
with directorio_mutex:
    archivo_encontrado = False

    # Buscar el archivo en el directorio
    for cluster in range(CLUSTERS_DIRECTORIO):
        posicion_inicial = (SUPERBLOQUE_CLUSTER + 1 + cluster) * TAMANO_CLUSTER
        with open(self.sistema_archivos.imagen_archivo, 'r+b') as img:
            img.seek(posicion_inicial)
            cluster_datos = img.read(TAMANO_CLUSTER)

            for entrada in range(0, TAMANO_CLUSTER, TAMANO_ENTRADA):
                entrada_actual = cluster_datos[entrada:entrada + TAMANO_ENTRADA]

                # Verifica si el archivo coincide con el nombre buscado
                nombre_archivo_actual = entrada_actual[1:16].decode().strip('\x00').strip()
                if nombre_archivo_actual == nombre_archivo:
                    # Sobrescribir la entrada con un símbolo de archivo vacío '#'
                    nueva_entrada = b'#' + entrada_actual[1:] # Mantener los otros campos
                    cluster_datos = (
                        cluster_datos[:entrada] + nueva_entrada + cluster_datos[entrada + TAMANO_ENTRADA:])
                )
                img.seek(posicion_inicial)
                img.write(cluster_datos) # Guardar cambios en el cluster
                archivo_encontrado = True
                CLEAR()
                print(f"\tArchivo '{nombre_archivo}' ha sido eliminado.")
                break
            if archivo_encontrado:
                break
    if archivo_encontrado:
        break
```



3.5 RutaArchivo.

Esta clase está diseñada para gestionar la obtención de la ruta de un archivo, ya sea preguntando al usuario si el archivo se encuentra en el directorio actual o solicitando la ruta completa.

- **obtener_ruta:** gestiona la interacción con el usuario para determinar la ubicación del archivo. Comienza preguntando si el archivo está en el directorio actual. Si el usuario responde afirmativamente, se le solicita el nombre del archivo. Si responde negativamente, se le pide la ruta completa. Si en cualquier momento el usuario escribe 'v', se regresa al menú principal.

Si el archivo no se encuentra, se muestra un mensaje de error y se le solicita al usuario que lo intente de nuevo.

```
def obtener_ruta(self):
    """
    Solicita al usuario si el archivo está en el directorio actual.
    Si no, permite al usuario ingresar la ruta completa del archivo.
    Devuelve la ruta del archivo si existe o None si se decide volver al menú.
    """
    while True:
        # Solicitar si el archivo está en el directorio actual
        opcion = input("\n\tEl archivo está en el directorio actual? (s/n): ").strip().lower()

        if opcion == 's':
            # Solicitar el nombre del archivo si está en el directorio actual
            nombre_archivo = input("\tIngrese el nombre del archivo (con la extensión) o escriba 'v' para volver: ").strip()

            if nombre_archivo.lower() == 'v':
                CLEAR()
                print("\t Volviendo al menú principal...\n")
                return None # Regresa al menú principal

            # Establecer la ruta del archivo en el directorio actual
            self.ruta_archivo = os.path.join(os.getcwd(), nombre_archivo)

        elif opcion == 'n':
            # Solicitar la ruta completa del archivo si no está en el directorio actual
            ruta_archivo = input("\tIngrese la ruta completa del archivo (con la extensión) o escriba 'v' para volver: ").strip()

            if ruta_archivo.lower() == 'v':
                CLEAR()
                print("\t Volviendo al menú principal...\n")
                return None # Regresa al menú principal

            # Establecer la ruta del archivo ingresada
            self.ruta_archivo = ruta_archivo

        else:
            print("\tOpción no válida. Inténtalo de nuevo.")
            continue

        # Verificar si el archivo existe en la ruta especificada
        if not os.path.isfile(self.ruta_archivo):
            CLEAR()
            print(f"\tEl archivo '{self.ruta_archivo}' no existe. Intenta de nuevo.\n")
        else:
            # Si el archivo existe, devuelve la ruta
            return self.ruta_archivo
```



3.6 ManejoDeHilos.

Facilita la gestión de operaciones relacionadas con archivos mediante el uso de hilos. Su objetivo es permitir que diferentes operaciones de archivo se ejecuten de manera concurrente, mejorando la eficiencia y la capacidad de respuesta del sistema. Esta clase se conforma de las siguientes funciones:

- **ejecutar_hilo:** Ejecuta una operación en un hilo independiente y espera a que termine.
- **listar_directorio:** Inicia un hilo para listar los archivos en el directorio.
- **copiar_de_FiUnamFs:** Inicia un hilo para copiar un archivo desde FiUnamFs a la máquina local.
- **copiar_a_FiUnamFs:** Inicia un hilo para copiar un archivo desde la máquina local a FiUnamFs.
- **eliminar_archivo:** Inicia un hilo para eliminar un archivo de FiUnamFs.

```
class ManejoDeHilos:

    def __init__(self, operaciones_directorio):

        #Inicializa la clase para gestionar los hilos y controlar los semáforos y mutex en cada operación.
        self.operaciones = operaciones_directorio

    def ejecutar_hilo(self, operacion, *args):

        #Ejecuta la operación en un hilo independiente con los argumentos especificados.
        hilo = threading.Thread(target=operacion, args=args)
        hilo.start()
        hilo.join()

    def listar_directorio(self):

        #Crea y maneja el hilo para la operación de listar directorio.
        self.ejecutar_hilo(self.operaciones.listar_directorio)

    def copiar_de_FiUnamFs(self, nombre_archivo):

        #Crea y maneja el hilo para copiar un archivo desde FiUnamFs a nuestro sistema local.
        self.ejecutar_hilo(self.operaciones.copiar_de_FiUnamFs, nombre_archivo)

    def copiar_a_FiUnamFs(self, ruta_archivo_local):

        # Crea y maneja el hilo para copiar un archivo desde el sistema local a FiUnamFs.
        self.ejecutar_hilo(self.operaciones.copiar_de_local, ruta_archivo_local)

    def eliminar_archivo(self, nombre_archivo):

        #Crea y maneja el hilo para eliminar un archivo en FiUnamFs.
        self.ejecutar_hilo(self.operaciones.eliminar_archivo, nombre_archivo)
```



3.8 Función global para verificar el sistema mediante hilos.

VerificarSistema se encarga de coordinar la verificación del sistema de archivos y la lectura del directorio de manera eficiente. Primero, inicia un hilo para ejecutar la verificación del archivo en el sistema de archivos, asegurándose de que esta tarea se complete antes de proceder. Una vez que el hilo de verificación ha terminado, la función inicia otro hilo para listar el directorio.

```
# Función para ejecutar los hilos de verificación
def verificarSistema(sistema_archivos):
    # Hilo para verificar el archivo
    hilo_verificación = threading.Thread(target=sistema_archivos.verificar_archivo)

    # Hilo para listar el directorio, esperando a que el primer hilo termine
    hilo_listar_directorio = threading.Thread(target=sistema_archivos.leer_directorio)

    hilo_verificación.start()
    hilo_verificación.join()

    hilo_listar_directorio.start()
    hilo_listar_directorio.join()
```

3.7 Características adicionales.

- **Uso de hilos en operaciones:** Se inicia un hilo por cada operación seleccionada, para cada uno de las operaciones se usa un mutex para verificar la sincronización de hilos.
- **Limpiar pantalla:** Se asignó una función global CLEAR() en el cual este llama a imprimir a pantalla "\033[2J\033[H", este código ANSI sirve para limpiar la pantalla sin tener que recurrir a funciones adicionales del sistema.
- **Funciones de impresión:** Para no aglomerar el código, se asignaron las funciones de impresión del menú y de la lista de archivos en el directorio hasta el final del código.
- **Borrado de archivos:** Para borrar los archivos del disco, simplemente renombramos el tipo de archivo como vacío b' #' , sin borrar sus datos, esto ayuda a la optimización del sistema.



- **Uso del programa:** No es necesario tener conocimiento técnico ni documentación adicional para poder utilizar nuestro programa.

4. Requisitos.

Para ejecutar este programa, es necesario cumplir con ciertos requisitos que aseguren su correcto funcionamiento. Algunos de los principales requisitos incluyen:

- Contar con Python 3 instalado en el sistema.
- Tener el archivo de imagen fiunamfs.img en su sistema local, no importa si no está en el mismo directorio donde se encuentra el programa (debe especificar la ruta específica donde se encuentra el archivo).
- No es necesario instalar bibliotecas adicionales, ya que el programa utiliza sólo bibliotecas estándar de Python.

5. Descripción de la sincronización empleada.

En el código los hilos se utilizan principalmente para manejar operaciones concurrentes relacionadas con el sistema de archivos. Aquí explicamos la funcionalidad de los hilos en las distintas partes del código:

5.1 Sincronización de Operaciones.

Se utilizan 2 Locks y un Semaphore para sincronizar el acceso a recursos compartidos, como el directorio y el superbloque, evitando así condiciones de carrera y garantizando que las operaciones no interfieran entre sí.

5.2 Clases Relacionadas con Hilos.

ManejoDeHilos: Esta clase es responsable de crear y ejecutar hilos para las operaciones de manejo de archivos. Proporciona métodos como listar_directorio, copiar_de_FiUnamFs, copiar_a_FiUnamFs, y eliminar_archivo, cada uno de los cuales crea un hilo para ejecutar la operación correspondiente.



5.3 Verificación del Sistema de Archivos

En la función verificarSistema, se crea un hilo para verificar la existencia del archivo del sistema de archivos verificar_archivo. Después de que se complete esta verificación con hilo_verificacion.join(), se inicia otro hilo para listar el directorio leer_directorio. Esto asegura que el listado del directorio se realice solo después de que se haya verificado la existencia del archivo.

6. Ejemplos de uso

Este programa no solamente se trata de un simple proyecto para satisfacer una entrega dentro de un tiempo definido, también puede tener múltiples usos en el ámbito educativo y práctico. Aquí podemos tener los siguientes ejemplos más específicos:

6.1 Cursos de sistemas operativos.

Comenzamos primeramente con un ejemplo bastante claro y por el cual se realizó este proyecto en primer lugar: Fines didácticos.

Con este programa se puede simular el comportamiento de los sistemas de archivos para que se pueda observar de manera más clara como es el funcionamiento de estos sin tener que realizar modificaciones al sistema de archivos real del sistema operativo que se tenga en el equipo. Además de que esto implica la enseñanza de otros conocimientos que vienen arraigados con el tema principal como lo son: las estructuras de datos para los archivos en conjunto con los directorios y el gestionar el almacenamiento dentro de un entorno controlado.

6.2 Desarrollo y pruebas de programas en un entorno controlado

Otro ejemplo de uso es el testeo de programas, aquí los desarrolladores pueden probar el manejo de los archivos dentro de un entorno aislado del sistema para evitar daños al sistema local.



Se puede usar el FiUnamFS como disco de prueba para operaciones de copiado o eliminado de archivos, además de poder simular errores a la hora de acceder a los archivos

6.3 Carga y recuperación en memoria limitada.

En este sistema de archivos virtual tenemos una memoria bastante limitada, ya que el manejo de archivos es bastante básico. Esto nos permite observar cómo los archivos se comportan cuando pueden ser almacenados dentro del dispositivo de almacenamiento y, por el contrario, cómo se procede el sistema al no poder almacenar dichos archivos porque no caben dentro de la capacidad establecida.

7. Referencias Electrónicas

1.- Descripción del proyecto en general

sistop-2025-1/proyectos/micro-sist-de-arch-multihilos at main

- *unamfi/sistop-2025-1*. (2024). GitHub.

<https://github.com/unamfi/sistop-2025-1/tree/main/proyectos/micro-sist-de-arch-multihilos>

2.- Instalación e información sobre Python

Python, "Python," Python.org, (29 de mayo de 2019).

<https://www.python.org/>

3.- Conocimientos sobre administración de procesos y sistema de archivos

G. E. .Wolf Iszaevich, "Sistemas operativos Gunnar Wolf", *Sistop.org*, (2024). <http://gwolf.sistop.org/>