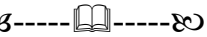
ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN



HỆ ĐIỀU HÀNH

BÁO CÁO ĐỒ ÁN 01: QUẢN LÝ TẬP TIN

GIẢNG VIÊN : ***CAO XUÂN NAM***

LỚP : ***21CLC09***

THÀNH VIÊN :

21127635 – Nguyễn Khánh Anh Kiệt

21127013 – Nguyễn Phú Minh Bảo

21127168 – Bùi Phước Thiện

Hồ Chí Minh, 2023

MỤC LỤC

I. THÔNG TIN THÀNH VIÊN

STT	MSSV	Họ và tên	Email
1	21127635	Nguyễn Khánh Anh Kiệt	nkakiet21@clc.fitus.edu.vn
2	21127013	Nguyễn Phú Minh Bảo	npmbao21@clc.fitus.edu.vn
3	21127168	Bùi Phước Thiện	bpthien21@clc.fitus.edu.vn

II. BẢNG PHÂN CÔNG CÔNG VIỆC

STT	Người thực hiện	Nội dung công việc	Mức độ hoàn thành
1	Nguyễn Khánh Anh Kiệt	Tìm hiểu và xây dựng thành công chương trình đọc thông tin trên phân vùng ổ đĩa FAT32 và NTFS	100%
2	Nguyễn Phú Minh Bảo	Xây cây thư mục của FAT32 và NTFS Định dạng lại code	100%
3	Bùi Phước Thiện	Xây dựng giao diện GUI cho người dùng	100%

III. CÁC CHỨC NĂNG ĐÃ THỰC HIỆN

STT	Chức năng
1	Tự phát hiện ổ đĩa là FAT32 hay NTFS
2	Hiển thị cây thư mục
3	Hiện đầy đủ tên của thư mục hay tập tin
4	Thư mục có thể mở rộng/gom các thư mục con
5	Hiện các thông tin cần thiết khi chọn 1 thư mục/tập tin

IV. MÔ TẢ CÁC HÀM SỬ DỤNG

a. FAT32

Định nghĩa các class cần đọc trong NTFS:

- Class BootSector: chứa các thông tin về loại partition FAT32 đang chọn với các giá trị: BytePerSector, BytePerCluster...

```
class BootSectorFAT32:

    # Initiate Value, Atrribute
    def __init__(self) -> None:
        self.bytePerSector = 0
        self.sectorPerCluster = 0
        self.sectorBeforeFAT = 0
        self.cntFAT = 0
        self.sizeVol = 0
        self.sectorPerFAT = 0
        self.firstClusterinRDET = 0
        self.FATtypeEach = 0
        self.FATtype = ""
```

- Class Entry: Chứa thông tin về 1 Entry (Tập tin, Folder, ...)

```
class Entry: # These just Object to store data of each Entry

    def __init__(self) -> None:
        self.name = ""
        self.attr = ["NULL", "NULL", "NULL", "NULL", "NULL", "NULL", "NULL",
"NULL"]
        #              X       X      ARCH    DIR     VOL    SYSTEM   HIDDEN  REA
DONLY
        self.attr_Bin = 0
        self.createTime = 0
        self.createDate = 0
        self.size = 0
        self.startCluster = 0
        self.tempName = ""
        self.ListEntry = [] # List of Entry
```

- Class RDET: Chứa thông tin về 1 ROOT ENTRY

```
class RDET:

    # We Read Each Entry And Store It In List
    def __init__(self) -> None:
        self.RootEntry = Entry()

    def PrintRDET(self):

        for i  in self.RootEntry.ListEntry:
            i.PrintAttribute()

    def ReadRDET(self,address, drive):

        # Read subEntry of the ROOT
        self.RootEntry.ReadDET(address, drive)

        return 0
```

Các bước đọc FAT32:

- Bước 1: Dò ra các ổ đĩa trên máy tính. Bắt đầu tiến hành kiểm tra:
  Nếu nó là FAT32. Chạy các hàm của FAT32
  Nếu nó là NTFS. Chạy các hàm của NTFS
- Bước 2: Đọc thông tin từ vùng BOOTSECTOR. Ta sử dụng seek(byte,1) để di chuyển giữa các offset nhằm lấy được giá trị ta mong muốn. Các dữ liệu thông thường sẽ được lưu dưới dạng int, dưới sự hỗ trợ của "From_Bytes" chuyển đổi từ byte sang int có tích hợp đọc byte theo kiểu Big/Little Endian

```
# Read Boot Sector - Read 26 Bytes At First Sector
    def ReadBootSector(self, drive):

        with open(drive, 'rb') as fp:

            # First we move to Offset 0x0B
            fp.read(11)

            # Read: Byte per sector, Sector per cluster, Sector before FAT, Count
of FAT
            self.bytePerSector = int.from_bytes(fp.read(2), byteorder='little')
            self.sectorPerCluster = int.from_bytes(fp.read(1),
byteorder='little')
            self.sectorBeforeFAT = int.from_bytes(fp.read(2), byteorder='little')
```

```
            self.cntFAT = int.from_bytes(fp.read(1), byteorder='little') # It End
At 0x11

            # Next, we move to Offset 0x20
            fp.seek(15,1)

            # Read: Size of volume, Sector per FAT
            self.sizeVol = int.from_bytes(fp.read(4), byteorder='little')
            self.sectorPerFAT = int.from_bytes(fp.read(4), byteorder='little') #
End At 0x28

            # Then we move to Offset 0x30
            fp.seek(4,1)

            # Read: First cluster in RDET
            self.firstClusterinRDET = int.from_bytes(fp.read(4),
byteorder='little') #End At 0x30

            # Final, we move to Offset 0x52
            fp.seek(34,1)

            # Read: Each Byte of FAT type then convert to ASCII
            for i in range(8):
                self.FATtypeEach = int.from_bytes(fp.read(1), byteorder='little')
                self.FATtype += chr(self.FATtypeEach)

        return 0
```

- Bước 3: Với những thông tin có được từ BootSector, ta sẽ tính toán được địa chỉ Cluster đầu tiên trong vùng DATA và địa chỉ RDET

```
 """ Address of FAT1, FAT2, DATA, RDET (Including Data part) """
            FAT1_Address = BOOT.sectorBeforeFAT * BOOT.bytePerSector
            FAT2_Address = (BOOT.sectorBeforeFAT + BOOT.sectorPerFAT) *
BOOT.bytePerSector
            FirstCluster_Data_Address = (BOOT.sectorBeforeFAT + BOOT.sectorPerFAT
* 2) * BOOT.bytePerSector
            RDET_Address = (BOOT.firstClusterinRDET- 2 ) * BOOT.sectorPerCluster
* BOOT.bytePerSector + FirstCluster_Data_Address
```

Kế tiếp, ta sẽ thực hiện đọc RDET:

```
def ReadInfoRDET(drive, BOOT, FirstClusterDATA):
```

```
    # Read Root Directory Entry Table
    RDET = Data.RDET()
    res = RDET.ReadRDET(FirstClusterDATA, drive)

    """ Read all directory in RDET """
    for x in RDET.RootEntry.ListEntry:

        # 1 Of Entry in RDET contain the information of the disk, so we need to
get it
        str = x.name.split('\x00')[0]
        if (str == 'System Volume Information'):
            RDET.RootEntry.attr = x.attr
            RDET.RootEntry.createDate = x.createDate
            RDET.RootEntry.createTime = x.createTime

        # Besides, We still continute to the full Information of Each Entry in
Root Directory Entry Table
        ReadAllDirectory_FromRDET(x, FirstClusterDATA, BOOT, drive)

    return RDET # Return Value
```

Tuy nhiên với những "Entry" trong RDET thôi là chưa đủ, vì thế ta dùng kĩ thuật đệ qui: Đi vào các cluster trong vùng DATA và đọc tất cả (nếu có thể) và lặp lại quá trình đến khi đã hoàn tất đọc được các file hoặc đạt đến 1 "Độ sâu" nhất định (tùy chỉnh)

```
def ReadAllDirectory_FromRDET(Entry, FirstClusterDATA, bootSector, drive, depth =
0):
    if depth >= 10: return None # The Depth can be adjusted to fit the
requirement

    # Goal: Read data in the Directory Entry
    # We just use Recursive to go further in the DataZone If only the Entry is a
Regular Directory with no extra attributes
    if Entry.attr[3] == 'DIRECTORY' and Entry.attr[4] == 'NULL' and Entry.attr[5]
== 'NULL' and Entry.attr[6] == 'NULL' and Entry.attr[7] == 'NULL':

        # Locate the Cluster contain the chosen Entry data
        EntryInsideDir_Address = (Entry.startCluster - 2 ) *
bootSector.sectorPerCluster * bootSector.bytePerSector + FirstClusterDATA
        Entry.ReadDET(EntryInsideDir_Address, drive) # Read it

        for x in Entry.ListEntry:
```

```
            if x.attr[3] == 'DIRECTORY' and x.attr[4] == 'NULL' and x.attr[5] ==
'NULL' and x.attr[6] == 'NULL' and x.attr[7] == 'NULL':
                    ReadAllDirectory_FromRDET(x, FirstClusterDATA, bootSector,
drive, depth + 1)

    return None
```

Bên trong hàm RDET: Mỗi địa chỉ Cluster bắt đầu của 1 Entry nào đó, ta sẽ xem Entry đó là "ROOT" và những Entry (Mỗi khi đọc 32 byte) được đọc tiếp theo đây sẽ là Node con của nó

```
class RDET:

    # We Read Each Entry And Store It In List
    def __init__(self) -> None:
        self.RootEntry = Entry()

    def PrintRDET(self):

        for i  in self.RootEntry.ListEntry:
            i.PrintAttribute()

    def ReadRDET(self,address, drive):

        # Read subEntry of the ROOT
        self.RootEntry.ReadDET(address, drive)

        return 0
```

Cụ thể hàm đọc thông tin "ReadDET" trong Class Entry:

```
def ReadDET(self,address, drive):

        # Step_1: Move to offset xxxB (1 byte): check the type of Entry
        # Step_2: If it is Main Entry, read 32 bytes
        # Otherwise, read Extra Entry
        # Step_3: Move another 32 Byte with Address and repeat Step_1

        with open (drive, 'rb') as fp:
            TempName = ""

            while True:
                # New Entry Each Time
```

```
                EachEntry = Entry()

                # Seek each Entry - 32 bytes pattern
                fp.seek(address,0)

                checkFirstByte = int.from_bytes(fp.read(1), byteorder='little')
                if checkFirstByte == 0x00: break # Empty Entry -> End Of
Directory
                if checkFirstByte == 0xE5: # If it is deleted Entry
                    address += 32 # Move to next Entry (+32bytes)
                    continue

                # Move to offset 11B (1 byte): check the type of Entry
                fp.read(10)

                getbinary = lambda x, n: format(x, 'b').zfill(n) # Full Fill The
Binary Pattern with n bit
                Entry_Type_Byte = int.from_bytes(fp.read(1), byteorder='little')

                # if Entry_Type_Byte == 0: break # Empty Entry -> End Of
Directory
                if  Entry_Type_Byte == 15: # This is the Extra Entry
                    EachEntry.ReadExtraEntry(address, drive, fp)

                    EachEntry.name = EachEntry.tempName + EachEntry.name
                    TempName = EachEntry.name + TempName # Save the name of Extra
Entry for the next Main Entry if needed

                else:
                    # Read Main Entry
                    EachEntry.ReadMainEntry(address, drive, fp)

                    if len(TempName) > 8: # If the name Main Entry size > 8, we
need to add the name of Extra Entry
                        EachEntry.name = TempName

                    TempName = "" # Reset the name of Extra Entry
                    if (EachEntry.name[0] != '.' and EachEntry.name[1] != '.'): #
Usually in each Directory Entry, it has 2 Entry: '.' and '..'
                        self.ListEntry.append(EachEntry)

                address += 32 # Move to next Entry (+32bytes)

        return None
```

Các hàm cụ thể "ReadMainEntry" và "ReadExtraEntry" trong ReadDET. MainEntry có các thông tin như Tên, Thuộc tính, Ngày Tạo, Thời gian tạo, kích thước nhưng ExtraEntry chỉ dùng để lưu tên trong trường hợp tên Entry vượt mức cho phép.

```
def ReadMainEntry(self, address, drive, fp):

        # Seek to address
        fp.seek(address,0)

        if(self.name == ""):
            #Read 8 first character
            for i in range(8): # We read one by one byte for 8 bytes
                eachName  = int.from_bytes(fp.read(1), byteorder = 'little') #
Read 1 byte
                if chr(eachName) != ' ': # If not space
                    self.name += chr(eachName)

            #Read 3 last character (if exist)
            checkBlank = True
            for i in range(3):
                eachName  = int.from_bytes(fp.read(1), byteorder = 'little')
                if (chr(eachName) != ' ' and checkBlank == True):
                    self.name += "."
                    checkBlank = False
                self.name += chr(eachName)
        else:
            fp.seek(11,1) # Seek 11 bytes to skip name, start read attribute

        #Read 1 byte for attribute
        getbinary = lambda x, n: format(x, 'b').zfill(n)
        self.attr_Bin = getbinary(int.from_bytes(fp.read(1), byteorder =
'little'),8)
        bi = self.attr_Bin # Convert to the bit pattern

        # Check the ith of bit pattern
        for i in range(len(bi)):
            if bi[i] == '1':
                if i==2:
                    self.attr[i] = "ARCHIVE"
                elif i==3:
                    self.attr[i] = "DIRECTORY"
                elif i==4:
                    self.attr[i] = "VOLUME LABEL"
```

```
                elif i==5:
                    self.attr[i] = "SYSTEM FILE"
                elif i == 6:
                    self.attr[i] = "HIDDEN FILE"
                elif i==7:
                    self.attr[i] = "READ ONLY"

        #Move the Created Time Part
        fp.seek(1,1)
        time = getbinary((int.from_bytes(fp.read(3),'little')), 24) # read 3
bytes
        self.createTime = str(int(time[0:5],2)) + ":" + str(int(time[5:11],2)) +
":" + str(int(time[11:16],2)) # 0->5: second, 5->11: minute, 11->16: hour

        #The Created Date Part
        date = getbinary((int.from_bytes(fp.read(2),'little')), 16) # read 2
bytes
        self.createDate = str(int(date[11:16],2)) + "/"+ str(int(date[7:11],2)) +
"/" + str(int(date[0:7],2)+1980) # 0->7: year, 7->11: month, 11->16: day

        # To Determine the exactly start cluster: Need the high word and low word
of Start Cluster bit pattern, Both are 2 bytes
        fp.seek(2,1)
        highword = int.from_bytes(fp.read(2),byteorder='little') << 16
        fp.seek(4,1)
        lowword = int.from_bytes(fp.read(2),byteorder='little')
        self.startCluster = highword + lowword

        # Size
        self.size = int.from_bytes(fp.read(4),byteorder='little') #Read 4 bytes

    def ReadExtraEntry(self, address, drive, fp):
        # Seek to address
        fp.seek(address,0)
        fp.seek(1,1) # Skip the first byte, get in the Name

        #Read 5 first character of the Name
        for i in range(5): #We read one by one byte for 5 times, each times read
2 bytes
            eachName = int.from_bytes(fp.read(2), byteorder = 'little')
            if eachName != 65535: # If not space
                self.tempName += chr(eachName)

        fp.seek(3,1) # Skip 3 more byte
```

```
        for i in range(6):
            eachName = int.from_bytes(fp.read(2), byteorder = 'little')
            if eachName != 65535:
                self.tempName += chr(eachName)

        fp.seek(2,1) # Skip 2 more byte

        #Read 2 last character of the Name
        for i in range(2): # We read one by one byte for 2 times, each times read
2 bytes
            eachName = int.from_bytes(fp.read(2), byteorder = 'little')
            if eachName != 65535:
                self.tempName += chr(eachName)
```

b. NTFS

Định nghĩa các class cần đọc trong NTFS:

- Class Content: Class chứa tất cả nội dung của phần Attribute. Chỉ quan tâm 2 attributes chính là $STANDARD_INFORMATION và $FILE_NAME

```
class Content:
    def __init__(self) -> None:
        self.standard_information = ContentOfStandardInformation()
        self.file_name = ContentOfFileName()
```

  - Class Content của $STANDARD_INFORMATION: chứa các thông tin

```
class ContentOfStandardInformation:
    def __init__(self) -> None:
        # self.major_version = 0
        # self.minor_version = 0
        # self.flags = 0
        self.create_time = 0
        self.last_modification_time = 0
        self.last_mft_modification_time = 0
        self.last_access_time = 0
        # self.file_attribute = 0
        # self.max_version = 0
        # self.version = 0
        # self.class_id = 0
        # self.owner_id = 0
        # self.security_id = 0
        # self.quota_charged = 0
        # self.update_sequence_number = 0
        # self.reserved = 0
```

  - Class Content của $FILE_NAME: chứa các thông tin

```
class ContentOfFileName:
    def __init__(self) -> None:
        self.IdRootParentDirectory = 0
        self.attr = ["NULL", "NULL", "NULL", "NULL", "NULL"]
        self.NameLength = 0
        self.Name = ''
        # self.parent_directory = 0
        # self.create_time = 0
        # self.last_modification_time = 0
        # self.last_mft_modification_time = 0
        # self.last_access_time = 0
        # self.logical_size = 0
        # self.logical_cluster_number = 0
        # self.flags = 0
        # self.real_size = 0
        # self.real_cluster_number = 0
        # self.file_name = 0
```

- Class Attribute: Bao gồm các thông tin phần Header của 1 attribute và phần content như trên

```
class Attribute: #type,size,
    def __init__(self) -> None:
        self.typeHeader = ''
        self.SizeOfAttributeIncludeHeader = 0 #bytesperAttributes
        self.NonResidentFlag = 0
        self.LengthOfContent = 0
        self.OffsetToContent = 0
        self.content = Content()
    # def __init__(self, type, name, size, data):
    #     self.type = type
    #     self.name = name
    #     self.size = size
    #     self.data = data
```

- Class MFT Entry: Bao gồm các thông tin phần Header của 1 MFT entry và tập hợp những Attributes

```
class MFTEntry:
    def __init__(self) -> None:
        self.MFTEntry = ''
        self.OffSetFirstAttri = 0
        self.Flag = ''
        self.SizeofusedMFTE = 0
        self.SizeofMFTE = 0
        self.IDofMFTEntry = 0
        self.sizeMFT = 0
```

```
        self.isROOT = False
        self.attributes = []
        self.listEntry = []
```

- Class MFT: Bao gồm tập hợp các MFT khác nhau

```
class MFT:
    def __init__(self) -> None:
        self.MFT = []
        self.Dictionary = {}
        self.MFTsize = 0
```

- Class VBR:

```
class VBR:
    def __init__(self) -> None:
        self.BytesPerSector = 0
        self.SectorsPerCluster = 0
        self.SectorsPerTrack = 0
        self.NumberOfHead = 0
        self.TotalSector = 0
        self.FirstClusterInMFT = 0
        self.FirstClusterInMFTMirr = 0
        self.BytesPerEntryMFT = 0 #Byte per MFT Entry
```

Các bước đọc NTFS:

- Bước 1: Dò ra các ổ đĩa trên máy tính. Bắt đầu tiến hành kiểm tra:
  Nếu nó là FAT32. Chạy các hàm của FAT32
  Nếu nó là NTFS. Chạy các hàm của NTFS
- Bước 2: Đọc các thông tin vùng VBR (Volume Boot Record):

```
def ReadVBR(self,drive,fp):
            #BytesPerSector in offset 0B -> read or seek 11 bytes -> 0B 00
            fp.read(11)
            #BytesPerSector in offset 0B -> read 2 bytes, fp pointer after read at
0D
            self.BytesPerSector = int.from_bytes(fp.read(2),byteorder='little')
            #SectorsPerCluster in offset 0D -> read 1 byte, fp pointer after read
at 0E
            self.SectorPerCluster = int.from_bytes(fp.read(1),byteorder='little')
            #SectorsPerTrack in offset 18 -> from 0E to 18 seek 10 bytes
            fp.seek(10,1)
            #SectorsPerTrack in offset 18 -> read 2 bytes, fp pointer after read
at 1A
            self.SectorPerTrack = int.from_bytes(fp.read(2),byteorder='little')
            #NumberOfHead in offset 1A -> read 2 bytes, fp pointer after read at
1C
            self.NumberOfHead = int.from_bytes(fp.read(2),byteorder='little')
```

```
            #TotalSector in offset 28 -> from 1C to 28 seek 12 bytes
            fp.seek(12,1)
            #TotalSector in offset 28 -> read 8 bytes, fp pointer after read at 30
            self.TotalSector = int.from_bytes(fp.read(8),byteorder='little')
            #FirstClusterInMFT in offset 30 -> from 30 to 38 seek 8 bytes
           self.FirstClusterInMFT = int.from_bytes(fp.read(8),byteorder='little')
            #FirstClusterInMFTMirr in offset 38 -> from 38 to 40 seek 8 bytes
                                                self.FirstClusterInMFTMirr       =
int.from_bytes(fp.read(8),byteorder='little')
            #BytesPerEntryMFT in offset 40 -> from 40 to 48 seek 8 bytes
                                                              BinPerMFTEntry         =
int.from_bytes(fp.read(1),byteorder='little',signed=True)
            #move to 2^bù 2
             self.BytesPerEntryMFT = 2**abs(BinPerMFTEntry) #so byte cua 1 entry
MFT
```

- Bước 3: Đọc đến phân vùng MFT từ cluster bắt đầu `FirstClusterInMFT` đã tìm được ở **Bước 2**

```
    def ReadMFT(self,drive,fp,offset,bytePerCluster):
        #seek to First Cluster in MFT
        fp.seek(offset)
        #Read MFT entry[0] to get the size of MFT
        temp = MFTEntry()
        temp.ReadMFTEntry(fp, drive[4:])
         self.MFTsize = SizeMFT #size of MFT is VCN in attribute data (virtual
cluster number)
        #Read each MFT entry, IF fp pointer > size of MFT, break
        while(True and fp.tell() <= (offset + self.MFTsize*bytePerCluster)):
            temp = MFTEntry()
            temp.ReadMFTEntry(fp, drive[4:])
            #IF MFT entry is FILE, add to MFT and Dictionary
            if(temp.MFTEntry == 'FILE'):
                self.MFT.append(temp)
                self.Dictionary[temp.IDofMFTEntry] = temp
                continue
            elif(temp.MFTEntry == 'BAAD'):
                self.MFT.append(temp)
                continue
        return self
```

- Bước 4: Khi đọc MFT. Ta đọc từng MFT entry. Hàm đọc MFT entry như sau:

```
def ReadMFTEntry(self,fp, driveName):
        #each MFT Entry has 1024 bytes but MFT Entry can have 1024 bytes 00
        #check if MFT Entry has 1024 bytes 00 -> seek to next MFT Entry
        #else -> seek back to the beginning of MFT Entry and read
        if(int.from_bytes(fp.read(1024), byteorder = 'little') != 0):
```

```
            fp.seek(-1024,1)
        else:
            return
        #read MFT Entry is FILE or BAAD
        for i in range(4):
            temp = fp.read(1)
            self.MFTEntry += temp.decode('ascii')
        self.MFTEntry = self.MFTEntry.replace('\x00','')
        #seek to 16 bytes from the beginning of MFT Entry to offset 14
        fp.seek(16,1)
        #read offset to first attribute
        self.OffSetFirstAttri = int.from_bytes(fp.read(2),byteorder='little') #luu
tru = bytes
        #read 2 bytes flag
        flag = hex(int.from_bytes(fp.read(2),byteorder='little'))
        if(flag == '0x0'):
            self.Flag = "File has already deleted"
        elif(flag == '0x1'):
            self.Flag = "File is in use"
        elif(flag == '0x2'):
            self.Flag = "Directory has already deleted"
        elif(flag == '0x3'):
            self.Flag = "Directory is in use"
        #read 4 bytes Size of used MFT Entry
        self.SizeofusedMFTE = int.from_bytes(fp.read(4),byteorder='little')
        #read 4 bytes Size of MFT Entry
        self.SizeofMFTE = int.from_bytes(fp.read(4),byteorder='little')
        fp.seek(12,1)
        #read 4 bytes ID of MFT Entry
        self.IDofMFTEntry = int.from_bytes(fp.read(4),byteorder='little')
         #seek to offfset first attribute but minus 48 bytes (48 bytes which we
have already read over)
        fp.seek(self.OffSetFirstAttri-48,1)
        while(True):
            temp = Attribute()
            #read attribute
            temp.ReadAttribute(fp)
              if(temp.typeHeader == 'STANDARD_INFORMATION' or temp.typeHeader ==
'FILE_NAME'):
                #add attribute to list attribute
                self.attributes.append(temp)
                #check if MFT Entry is ROOT
                            if  temp.content.file_name.IdRootParentDirectory  ==
self.IDofMFTEntry:
                    self.isROOT = True
```

```
                    #is ROOT = true -> this is MFT of Drive -> change name from
'.' to drivename
                    for i in range(len(self.attributes)):
                        if self.attributes[i].typeHeader == 'FILE_NAME':
                            self.attributes[i].content.file_name.Name = driveName
            if(temp.typeHeader == "END"): #if attribute is END -> break
                break
        fp.seek(self.SizeofMFTE - self.SizeofusedMFTE+4,1) #seek to next MFT Entry
        return self
```

- Bước 5: Khi đọc MFT Entry, ta đọc các attributes của MFT entry. Hàm đọc Attributes như sau

```
    def ReadAttribute(self,fp):
            #read type of attribute
            HeaderTypeHex = hex(int.from_bytes(fp.read(4),byteorder='little'))
            if(HeaderTypeHex == '0x10'):
                self.typeHeader = "STANDARD_INFORMATION"
            elif(HeaderTypeHex == '0x30'):
                self.typeHeader = "FILE_NAME"
            elif(HeaderTypeHex == '0x80'):
                self.typeHeader = "DATA"
            elif(HeaderTypeHex == '0x90'):
                self.typeHeader = "INDEX_ROOT"
            elif(HeaderTypeHex == '0xa0'):
                self.typeHeader = "INDEX_ALLOCATION"
            elif(HeaderTypeHex == '0xb0'):
                self.typeHeader = "BITMAP"
            elif(HeaderTypeHex == '0x100'):
                self.typeHeader = "VOLUME_NAME"
            elif(HeaderTypeHex == '0x120'):
                self.typeHeader = "VOLUME_INFORMATION"
            elif(HeaderTypeHex == '0xffffffff'):
                self.typeHeader = "END"
                return
            #read size of attribute
                                        self.SizeOfAttributeIncludeHeader      =
int.from_bytes(fp.read(4),byteorder='little')
            if(self.typeHeader == "DATA"):
                #if data attribute, read size of MFT (only in MFTentry[0])
                global SizeMFT
                fp.seek(16,1)
                SizeMFT = int.from_bytes(fp.read(8),byteorder='little')
                #seek to end of attribute
                fp.seek(self.SizeOfAttributeIncludeHeader-32,1)
                return
```

```
            #if not standard_information, file_name, end, data attribute -> seek
to end of attribute
            elif(self.typeHeader != "STANDARD_INFORMATION" and self.typeHeader !=
"FILE_NAME" and self.typeHeader != "END" and self.typeHeader != "DATA"):
                fp.seek(self.SizeOfAttributeIncludeHeader-8,1)
                return
            #read NonResidentFlag
            self.NonResidentFlag = int.from_bytes(fp.read(1),byteorder='little')
            fp.seek(7,1)
            #read length of content
            self.LengthOfContent = int.from_bytes(fp.read(4),byteorder='little')
            #read offset to content
            self.OffsetToContent = int.from_bytes(fp.read(2),byteorder='little')
            #header 16 bytes, size of content 4 bytes, offset to content 2 bytes
            #fp dang o vi tri offset cuoi content
            #fp phai seek qua backup -> seek(vitri content -(16+4+2))
            fp.seek(self.OffsetToContent-22,1)
            self.content = Content()
            if(self.typeHeader == "STANDARD_INFORMATION"):
                #read information of standard_information
                                self.content.standard_information.create_time   =
as_datetime(int.from_bytes(fp.read(8),byteorder='little'))
                                self.content.standard_information.create_time   =
self.content.standard_information.create_time.strftime("%d/%m/%Y %H:%M:%S")
                        self.content.standard_information.last_modification_time  =
as_datetime(int.from_bytes(fp.read(8),byteorder='little'))
                        self.content.standard_information.last_modification_time  =
self.content.standard_information.last_modification_time.strftime("%d/%m/%Y
%H:%M:%S")
                    self.content.standard_information.last_mft_modification_time =
as_datetime(int.from_bytes(fp.read(8),byteorder='little'))
                    self.content.standard_information.last_mft_modification_time =
self.content.standard_information.last_mft_modification_time.strftime("%d/%m/%Y
%H:%M:%S")
                            self.content.standard_information.last_access_time   =
as_datetime(int.from_bytes(fp.read(8),byteorder='little'))
                            self.content.standard_information.last_access_time   =
self.content.standard_information.last_access_time.strftime("%d/%m/%Y %H:%M:%S")
                fp.seek(self.LengthOfContent-32,1)
            elif(self.typeHeader == "FILE_NAME"):
                #read information of file_name
                                self.content.file_name.IdRootParentDirectory   =
int.from_bytes(fp.read(6),byteorder='little')
                fp.seek(50,1)
                getbinary = lambda x, n: format(x, 'b').zfill(n)
```

```
                attr = getbinary(int.from_bytes(fp.read(4),byteorder='little'),32)
                attr = attr[::-1]
                for i in range(len(attr)):
                    if(attr[i] == '1'):
                        if(i == 0):
                            self.content.file_name.attr[0] = "READ_ONLY"
                        elif(i == 1):
                            self.content.file_name.attr[1] = "HIDDEN"
                        elif(i == 2):
                            self.content.file_name.attr[2] = "SYSTEM"
                        elif(i == 5):
                            self.content.file_name.attr[3] = "ARCHIVE"
                        elif(i== 28):
                            self.content.file_name.attr[4] = "DIRECTORY"
                fp.seek(4,1)
                                            self.content.file_name.NameLength    =
int.from_bytes(fp.read(1),byteorder='little')
                fp.seek(1,1)
                for i in range(self.content.file_name.NameLength):
                    eachName = fp.read(2)
                    if eachName != b'\xff\xff':
                        self.content.file_name.Name += eachName.decode('utf-16')
                                                    self.content.file_name.Name     =
self.content.file_name.Name.replace('\x00','')
                                        fp.seek(self.SizeOfAttributeIncludeHeader    -
(self.OffsetToContent+self.LengthOfContent),1)
            else:
                fp.seek(self.SizeOfAttributeIncludeHeader-4,1)
```

c. Một số Hàm Bổ Sung:

- Sau khi hoàn tất dữ liệu việc tiếp theo là hiển thị lên GUI, Hàm Push_To_GUI với tham số truyền vào là TypePartition sẽ quyết định nạp dữ liệu lên theo kiểu nào “FAT32” or “NTFS”

```
def Push_To_GUI(Entry, TypePartition, file_path):

    if TypePartition == 'FAT32':
        file_path.append(Load_FAT_DATA(Entry,True))
    else:
        file_path.append(Load_NTFS_DATA(Entry,True))
```

- Đối với hàm Load_FAT_DATA:

```
def Load_FAT_DATA(Entry, isROOT = False):
```

```
    path = [] # List of Children

    if isROOT == True: # If the Entry is the Root Directory, we load all info of
it even it is a hidden file or system file
        for x in Entry.ListEntry:
            res =Load_FAT_DATA(x)
            if res != '': # If the Entry is not a hidden file or system file, we
load it
                path.append(res)

        dict_path = {} # Dictionary of the Entry, Help located the data easier
with calling the key
        dict_path["Name"] = Entry.name

        str = ''
        for i in range(len(Entry.attr)):
            if Entry.attr[i] != "NULL" and i != 4 and i != 5 and i != 6: # Only
load the attribute that is not Hidden File or System File
                if str != '': str += ',' + Entry.attr[i]
                else: str += Entry.attr[i]

        dict_path["Attribute"] = str
        dict_path["Date_Created"] = Entry.createDate
        dict_path["Time_Created"] = Entry.createTime
        dict_path["Size"] = Entry.size
        dict_path["Children"] = path

    else:
        if Entry.attr[3] == 'DIRECTORY' and Entry.attr[4] == 'NULL' and
Entry.attr[5] == 'NULL' and Entry.attr[6] == 'NULL' and Entry.attr[7] == 'NULL' :

            for x in Entry.ListEntry:
                res =Load_FAT_DATA(x)
                if res != '': # If the Entry is not a hidden file or system file,
we load it
                    path.append(res)

            dict_path = {} #Dictionary of the Entry, Help located the data easier
with calling the key
            dict_path["Name"] = Entry.name

            str = ''
            for i in range(len(Entry.attr)):
```

```
                if Entry.attr[i] != "NULL": #Only load the attribute that is not
Hidden File or System File
                    if str != '': str += ',' + Entry.attr[i]
                    else: str += Entry.attr[i]

            dict_path["Attribute"] = str
            dict_path["Date_Created"] = Entry.createDate
            dict_path["Time_Created"] = Entry.createTime
            dict_path["Size"] = Entry.size
            dict_path["Children"] = path

        elif Entry.attr[4] == 'VOLUME LABEL' or Entry.attr[5] == 'SYSTEM FILE' or
Entry.attr[6] == 'HIDDEN FILE': #If the Entry is a Hidden File or System File
                return ''
        else: #If the Entry is a File
            dict_path = {} #Dictionary of the Entry, Help located the data easier
with calling the key
            dict_path["Name"] = Entry.name

            str = ''
            for i in range(len(Entry.attr)):
                if Entry.attr[i] != "NULL": #Only load the attribute that is not
Hidden File or System File
                    if str != '': str += ',' + Entry.attr[i]
                    else: str += Entry.attr[i]

            dict_path["Attribute"] = str
            dict_path["Size"] = Entry.size
            dict_path["Date_Created"] = Entry.createDate
            dict_path["Time_Created"] = Entry.createTime
            dict_path["Size"] = Entry.size

            return dict_path

    return dict_path
```

- Đối với hàm Load_NTFS_DATA:

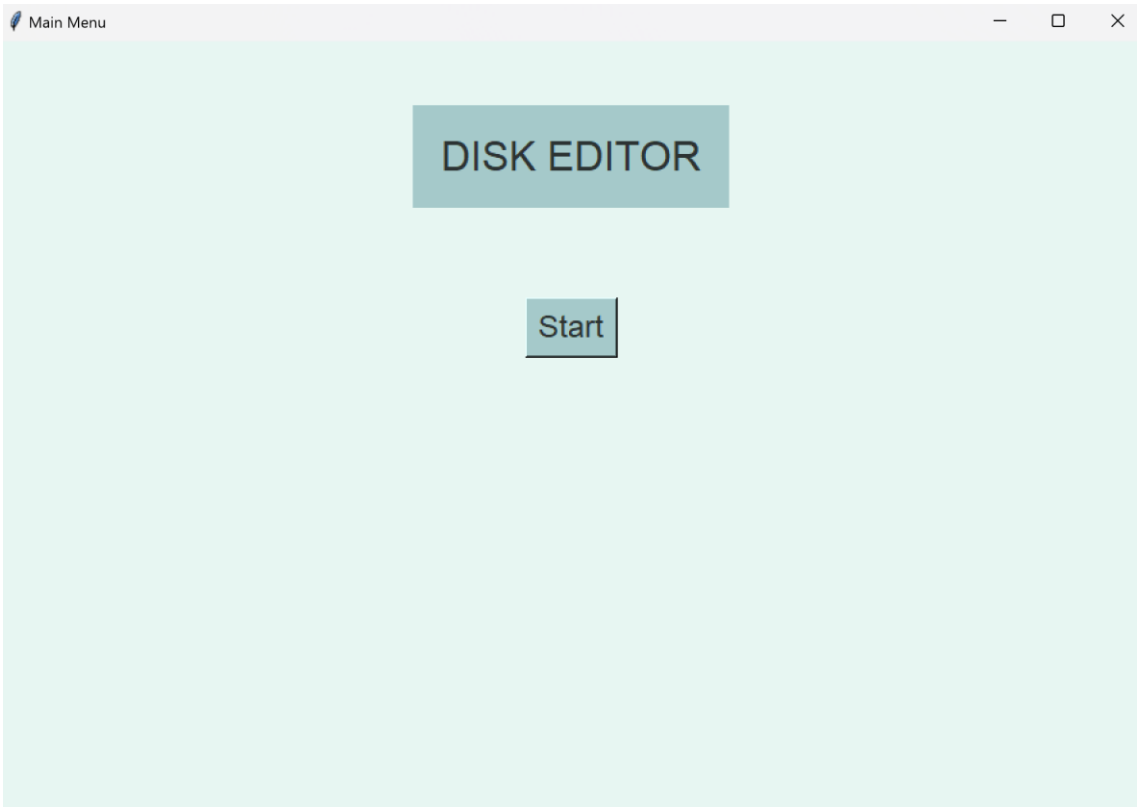
```
def Load_NTFS_DATA(Entry, isROOT = False):

   path = [] #List of Children
    NTFS_CreateTime = "" #Create Time of the Entry
    Check_Is_Folder = False #Check if the Entry is a Folder
```

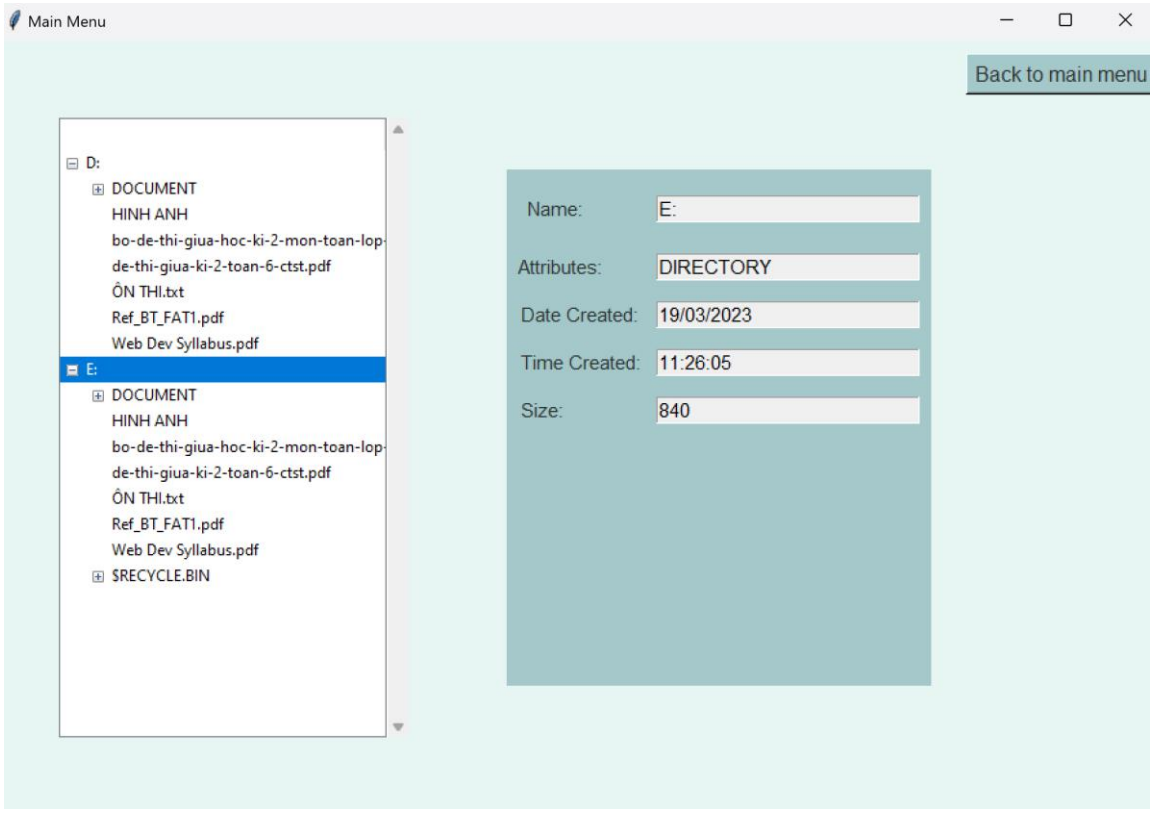
```
    for j in range(len(Entry.attributes)): #Scan all the attribute of the Entry
        if(Entry.attributes[j].typeHeader == 'STANDARD_INFORMATION' ):
            NTFS_CreateTime =
Entry.attributes[j].content.standard_information.create_time

        if(Entry.attributes[j].typeHeader == 'FILE_NAME'):#If the attribute
contain important information about the Entry
            if(Entry.attributes[j].content.file_name.attr[1] != "NULL" or
Entry.attributes[j].content.file_name.attr[2] != "NULL"):
                if(isROOT != True): return ""

                dict_path = {} # Dictionary of the Entry, Help located the data
easier with calling the key
                dict_path["Name"] = Entry.attributes[j].content.file_name.Name
#Name of the Entry

                str = ''
                for x in range(len(Entry.attributes[j].content.file_name.attr)):
                    if Entry.attributes[j].content.file_name.attr[x] != "NULL"
and x != 1 and x != 2: #Only load the attribute that is not Hidden File or System
File
                        if str != '': str += ',' +
Entry.attributes[j].content.file_name.attr[x]
                        else : str +=
Entry.attributes[j].content.file_name.attr[x]

                dict_path["Attribute"] = str #Attribute of the Entry
                dict_path["Date_Created"] = NTFS_CreateTime.split(" ")[0] #Date
of the Entry
                dict_path["Time_Created"] = NTFS_CreateTime.split(" ")[1] #Time
of the Entry
                dict_path["Size"] = Entry.SizeofusedMFTE #Size of the Entry
                Check_Is_Folder = True

            else:
                dict_path = {} # Dictionary of the Entry, Help located the data
easier with calling the key
                dict_path["Name"] = Entry.attributes[j].content.file_name.Name
#Name of the Entry

                str = '' #Attribute of the Entry
                for x in range(len(Entry.attributes[j].content.file_name.attr)):
                    if Entry.attributes[j].content.file_name.attr[x] != "NULL":
#Only load the attribute that is not Hidden File or System File
```

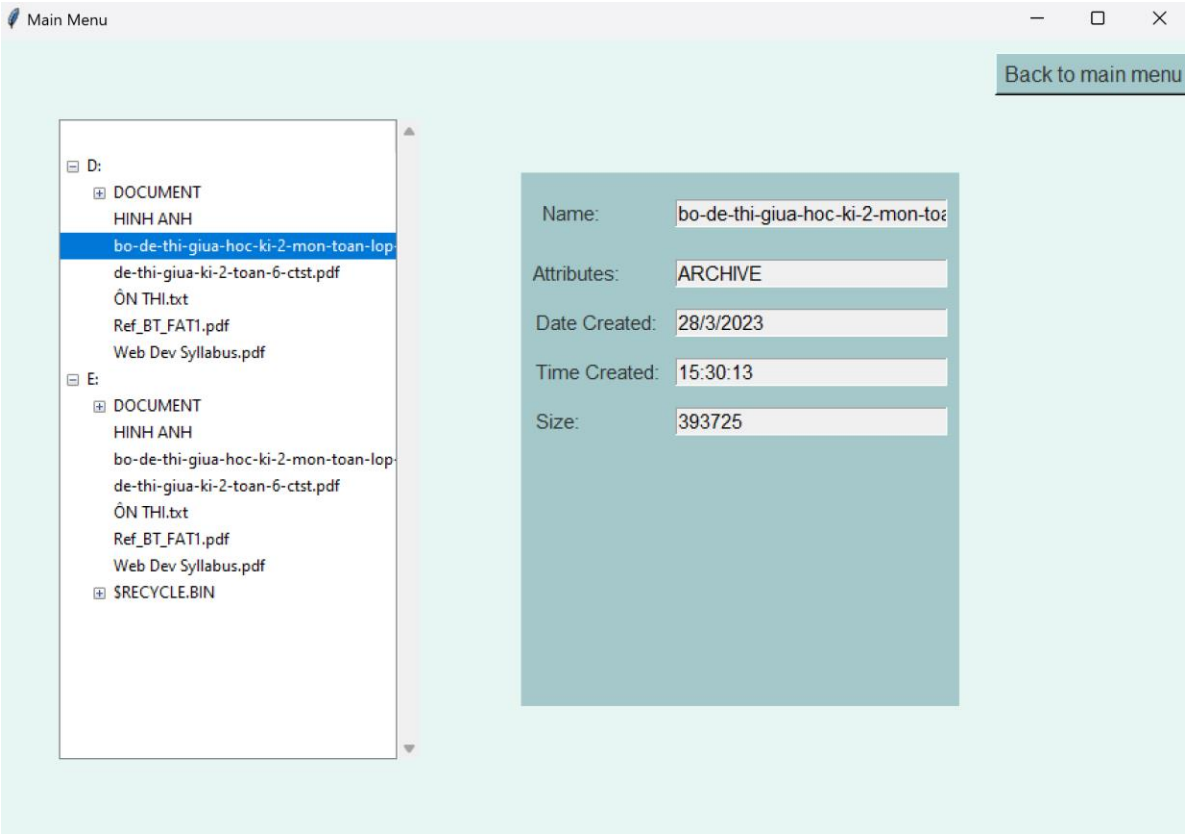
```
                        if str != '': str += ',' +
Entry.attributes[j].content.file_name.attr[x]
                        else : str +=
Entry.attributes[j].content.file_name.attr[x]

                dict_path["Attribute"] = str #Attribute of the Entry
                dict_path["Date_Created"] = NTFS_CreateTime.split(" ")[0] #Date
of the Entry
                dict_path["Time_Created"] = NTFS_CreateTime.split(" ")[1] #Time
of the Entry
                dict_path["Size"] = Entry.SizeofusedMFTE #Size of the Entry

                if Entry.attributes[j].content.file_name.attr[4] != "NULL":
Check_Is_Folder = True #Check if the Entry is a Folder
            break


    for x in Entry.listEntry:
        res = Load_NTFS_DATA(x)
        if res != '': #If data is not empty
            path.append(res)

    if len(path) == 0: path = ""
    if Check_Is_Folder == True: #If the Entry is a Folder
        dict_path["Children"] = path

    return dict_path
```

Sau đó, các dữ liệu sẽ được lưu ở File_Path và ta chỉ cần biểu diễn nó lên GUI

V. CHẠY THỬ TRÊN TERMINAL







VI. NGUỒN THAM KHẢO

[i] https://drive.google.com/drive/folders/1QimR-ok34rm4QyrPvGID3Lxnx2pxWlkq

[ii] Clusters - Concept - NTFS Documentation (pucp.edu.pe)

[iii] 13: Introduction to NTFS | COMPSCI 365 | Digital Forensics (Spring 2019) (umass.edu)

[iv] Python - Tkinter Button (tutorialspoint.com)

[v] Python - Tkinter Entry (tutorialspoint.com)

[vi] Python - Tkinter pack() Method (tutorialspoint.com)

[vi] Python - Tkinter Label (tutorialspoint.com)

[vii] Tkinter Layouts, designing Python GUI (pythonguis.com)