**VIRTUAL FILE**

**MANAGEMENT SYSTEM ‎**

**DOCUMENTATION &**

**USER-GUIDE**

**Page:**

1. **Introduction, System Structure, Classes**
2. **System Design Modification**
3. **Memory Map, Main/Main Loop and System Functionalities**
4. **Limitations**
5. **Outputs/Results**

**By: Ayesh Ahmad**

**Introduction**

This Python script consists of 3 classes, a terminal, and a main that work in tandem to represent a multithreaded virtual file system. The file system is based on directories and files, with the ability to create, delete, read and write files and directories. The purpose of the system is to enable multiple users to interact with the file system at the same time.

**System Structure**

The system structure consists of 3 classes working in composition. Once the VirtualFileSystem object is created in the terminal, it immediately creates a Directory object within itself as the “root” and builds further on it. Directory objects store content as dictionaries, key-value pairs that cannot be repeated (hence allowing the usual unique naming convention) and hold associated data with the key. When shifting and working between directories, we simply keep track of the current working directory and its parent and swap them around as needed.

**Classes**

Note: All methods use a str type return in order to allow writing to an output\_thread<x>.txt file

File

# Attributes

**name(str)**: the name of the file

**type(str)**: the file extension

**content(str)**: the contents of the file

**size(int)**: the size of the file in bytes

**created\_at(datetime.datetime)**: the datetime when the file was created

**modified\_at(datetime.datetime)**: the datetime when the file was last modified

**open\_mode(str)**: the mode in which the file is currently open ('r', 'w')

**lock(threading.Lock)**: a lock used to ensure thread safety when accessing the file's attributes

# Methods

**\_\_init\_\_(self, name, content='')**: initializes a new **File** object with the specified name and content (defaults to empty string)

**open(self, mode)**: opens the file in the specified mode ('r', 'w')

**close(self)**: closes the file if it is currently open

**write(self, data**: writes the specified data to the file if it is currently open in write mode

**write\_at(self, offset, data)**: writes the specified data to the file at the specified offset if it is currently open in write mode

**read(self)**: reads the contents of the file if it is currently open in read mode

**read\_at(self, offset, length)**: reads the contents of the file starting from the specified offset up to the specified length if it is currently open in read mode

**truncate(self, size=None)**: truncates the file to the specified size (defaults to 0) if it is not currently open

Directory

# Attributes

**name(str)**: the name of the directory

**parent(Directory)**: the parent directory (None if this is the root directory)

**created\_at(datetime.datetime)**: the datetime when the directory was created

**modified\_at(datetime.datetime)**: the datetime when the directory was last modified

**contents(dict)**: a dictionary containing the files and sub-directories in the directory

**lock(threading.Lock)**: lock used to ensure thread safety when accessing the directory's attributes

# Methods

**\_\_init\_\_(self, name, parent=None)**: initializes a new **Directory** object with the specified name and parent directory (defaults to None for root directory)

**add\_file(self, file)**: adds the specified file to the directory

**add\_directory(self, directory)**: adds the specified sub-directory to the directory

**get\_file(self, name)**: returns the file with the specified name in the directory (None if not found)

**get\_directory(self, name)**: returns the sub-directory with the specified name in the directory (None if not found)

**remove\_file(self, file)**: removes the specified file from the directory

**remove\_directory(self, directory)**: removes the specified sub-directory from the directory

VirtualFileSystem

# Attributes

**root(Directory)**: the root directory

**current\_directory(Directory)**: the current working directory (initially set to root)

**memory(list of dicts)**: 8\*8 grid meant to simulate a 64 byte memory; each block (list) has 8 bytes and each byte (dict) has 1 word and its metadata

**lock(threading.Lock)**: lock used to ensure thread safety when accessing the system’s attributes

# Methods

**\_\_init\_\_(self)**: Initializes the virtual file system by creating a root directory named **root**, and setting the current directory to the root directory. It also initializes a lock using the **threading.Lock()** method, and creates a 2D list called **memory** which is used to simulate the blocks of memory in the file system.

**create\_file(self, name)**: Creates a new file with the specified **name** in the current directory. If the file already exists in the current directory, it returns an error message.

**delete\_file(self, name)**: Deletes the file with the specified **name** from the current directory. If the file does not exist in the current directory, it returns an error message.

**create\_directory(self, name)**: Creates a new directory with the specified **name** in the current directory. If the directory already exists in the current directory, it returns an error message.

**delete\_directory(self, name)**: Deletes the directory with the specified **name** from the current directory. If the directory does not exist in the current directory, it returns an error message.

**change\_directory(self, path)**: Changes the current directory to the specified **path**. The **path** can be either absolute or relative. If the **path** is absolute, it starts at the root directory. If the **path** is relative, it starts at the current directory. It returns an error message if the directory does not exist.

**move\_file(self, file\_name, target\_directory\_name)**: Moves the file with the specified **file\_name** to the directory with the specified **target\_directory\_name**. If the file does not exist or the target directory does not exist, it returns an error message.

**calc\_free\_memory(self)**: Calculates the number of free blocks of memory in the virtual file system.

**update\_mmap(self, \_file)**: Updates the memory map of the file system with the specified **\_file**.

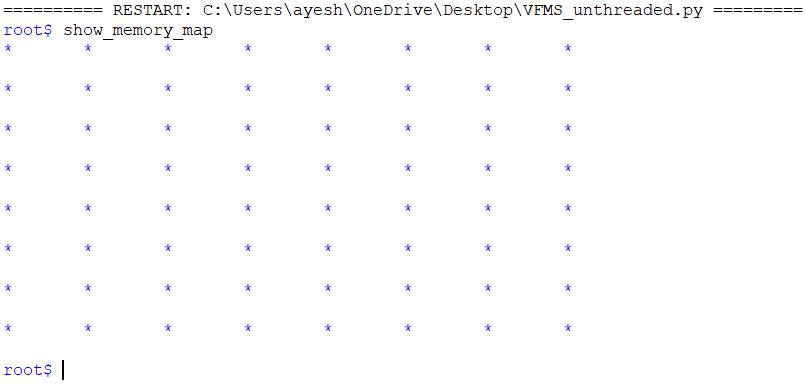
**SYSTEM DESIGN MODIFCATION**

The overall design for the system has largely remained the same. The new functionalities added are relatively minor and simple such as:

* **self.lock** as an attribute for all classes to prevent deadlocks and ensure atomicity **import threading**
* **import sys** to work with system files (.txt and .json)
* a **.json** file to save and load old system states, if any **import pickle, import json pickle**
* new minor portion of code discussed later in the code related to creating and managing threads (lines 255-262 and 449-461)
* **import builtins, import patch** to automatically run the threads

**MEMORY MAP**

This system’s memory map is designed using a 2D list consisting of dictionaries. For test purposes, the memory size was limited to 64 bytes which translates to 8 blocks (lists) and 8 words (dicts). Visualizing an empty memory map in the system:



word1 word2 word3 word4 word5 word6 word7 word8

block1

block2

block3

block4

block5

block6

block7

block8

As the memory map uses the conventional, simple, “first fit” method, an example output of it after creating and writing to some files will be:

A picture containing table

Description automatically generated

Each part of the block holds the word, its metadata, and the block number.

**MAIN/MAIN LOOP & SYS FUNCTIONALITIES**

The multithreaded system’s main is:

|  |
| --- |
| if \_\_name\_\_ == "\_\_main\_\_":  if len(sys.argv) < 2:  print("Usage: python program\_name.py <number\_of\_threads>")  sys.exit(1)  k = int(sys.argv[1])  threads = []  for i in range(k):  t = threading.Thread(target=terminal)  threads.append(t)  t.start()  # Wait for all threads to complete  for t in threads:  t.join() |

‘k’ number of threads is taken as an input from the user and the main loop method **terminal()** is then assigned to each thread

**terminal()**: A main interactive loop that takes input from the user and performs the corresponding operation on the virtual file system. It includes the following commands:

**ls**: list contents of the current directory

**mkdir <name>**: create new directory in current directory

**rmdir <name>**: directory from current directory

**chdir <path>**: change current directory. Set path as:

.. ==> Move up directory

/ ==> Return to root

/d/d ==> Absolute path

d/d ==> Relative path

**create <name>**: create new file in current directory

**delete <name>**: remove file from current directory

**open <name> <mode>**: open file in r or w mode

**close <name>**: close file

**write\_to\_file <name> <date> <offset>**: write to file at a specific offset (optional)

**read\_from\_file <name> <offset> <length>**: read from file from a specific offset (optional)

**truncate <name> <size>**: truncate file to a specified size (or all of it if not specified)

**show\_memory\_map**: display memory map

**help**: display this help message

**exit**: exit the program

terminal initialization portion:

|  |
| --- |
| try:  with open("VFMS.json") as jf:  temp = json.load(jf)  vfs = jsonpickle.decode(temp)  except:  vfs = VirtualFileSystem()  t\_no = threads.index(t)+1  with open(f"input\_thread{t\_no}.txt", "r") as fin:  lines = []  for line in fin:  line = line.strip()  lines.append(line)  fout = ''  with patch('builtins.input') as input\_mock:  input\_mock.side\_effect = lines |

The terminal attempts to open an older save of the system at first. If an older save does not exist, it will create one from scratch. As the script is run automatically by invoking input\_thread<x>.txt input files and producing output\_thread<x>.txt output files, this initialization portion opens the appropriate input file for the thread and automatically feeds all of the commands into the terminal. All methods which modify the system return strings to indicate a success or failure, which are appended to **fout**. At the end an output file is created and the contents of **fout** are written into it.

terminal graceful shutdown portion:

|  |
| --- |
| elif parts[0] == "exit":  with open(f"output\_thread{t\_no}.txt", "w") as f:  f.write(fout)  temp = jsonpickle.encode(vfs, unpicklable=False)  with open("VFMS.json","w") as f:  json.dump(temp,f)  break |

Once the exit command is called, all output thread files are created, and the system state is saved.

**LIMITATIONS**

The main limitations of the system include:

* Memory size
* Lock availability
* Number of threads

**OUTPUTS/RESULTS**

**Input files:**

Graphical user interface

Description automatically generated

**Output files:**

A screenshot of a computer

Description automatically generated with medium confidence

**.json file**

Graphical user interface, text, application, email

Description automatically generated

**Terminal upon execution (all outputs are displayed in output files only):**  
