Bastion: Filesystem and Shell
CS560: Programming Assignment 1

Elliot D. Greenlee and Jared M. Smith

February 28th, 2017

1 Introduction

We implemented a filesystem and shell to access our filesystem for the first programming assignment in CS560: Advanced Operating Systems. We wrote all of the code in Python 2.7, using only the standard library and the extensive os module in Python (`https://docs.python.org/2/library/os.html`). We chose Python for its abilities to write high level code while also interacting on a low level with the operating system through the standard library. We named our system after Bastion, a character from Overwatch (`https://playoverwatch.com/en-us/`), a modern team-oriented combat game, because this particular character is a battle-hardened, resilient, and fault-tolerant fortress for the less resistant players of the team to follow into battle. Though our filesystem and shell are not quite as battle-hardened as systems like the Google File System, Ext4, and HDFS, bastion's ultimate goal is to be the go-to versatile filesystem and shell for computer users.

Figure 1: Bastion: An Overwatch Hero

2 Usage

Bastion is run by typing **python2 run_shell** from the root directory of our code submission, which will put the user into the Bastion prompt loop of our shell, where all available commands can be run. For a list of commands see the section on commands below.

3 Implementation

We implemented Bastion in Python 2.7, with the ability to run the program on the Hydra labs in EECS, a Linux environment. We made extensive use of the os module in Python (`https://docs.python.org/2/library/os.html`), as well as the object-oriented semantics of the language.

Given that the operational use case for this shell is one file system per run, which is overwritten on the next use, we chose to prioritize management speedup and available disk space. To further this, our system is divided into file storage and management. File content storage occurs on disk, with free space checked on every operation to connect contiguous regions and prevent fragmentation. Management of the system occurs in memory, allowing caching of the most used regions. This means that directory and file metadata are available without checking the disk. Overall, we believe these choices allow for rapid development changes to the file system. Larger file systems containing many small files may lead to performance drops due to the in memory management, but in the given use case this seems unlikely.

3.1 Filesystem

We implemented the filesystem with the following set of Python classes:

- **FileSystem**: This class represents the root filesystem and holds references to all the files and directories in the system, as well as the raw file allocation blocks on the physical filesystem on the host machine. During the lifetime of the shell, we represent the filesystem in memory, and serialize the file content to a file on the host machine's disk when commands like **write**, **open**, and **close** are run.
  - This class contains the following attributes:
    - **exists**: True if the filesystem exists on the host machine's disk, false otherwise.
    - **fd**: The file descriptor counter.
    - **available_fds**: A list of pre-used file descriptors.
    - **open_files**: This list holds the files currently open as OpenFiles.
    - **root**: This points to the filesystem root.
    - **free_list**: We use this list of FileSystemAllocations to keep track of the free space in the representation of our virtual file system

when it's serialized to disk. We check this list when allocating more space for creating and writing files on the host machine hard drive file, and when freeing space in that file.

  - This class contains the following functions:
    * **__init__**: This function is the constructor of the class, which is run on class instantiation.
    * **on_disk**: This function checks whether the filesystem exists on the host machine's disk.
    * **get_new_fd**: This function gets an available file descriptor from the filesystem, or creates a new one by simply implementing a counter.
    * **find_open_fd**: This function takes a file descriptor and gets the corresponding OpenFile if it exists in the file system.
    * **find_open_name**: This function takes a file name and gets the corresponding OpenFile if it exists in the file system.
    * **get_free_space**: This function takes a byte size and returns a byte offset in the physical on disk filesystem and returns to a block of free space with that corresponding byte size available.
    * **load_from_disk**: This function takes a byte offset and size in bytes and reads content that size in bytes from the start of the given byte offset, and then returns that content to the calling function.
    * **write_to_disk**: This function takes a byte offset and content string, and writes that content string into the physical filesystem starting at the given byte offset.
    * **free_space**: This function takes a byte offset and byte size and returns that space to the free list.

- **Directory**: This class represents a directory in the virtual filesystem. Directory instances are stored within other Directory instances, starting from the root. The Directory class stores it's children in the **children** instance method of the class.
  - This class contains the following attributes:
    * **name**: The name of the directory.
    * **parent**: The parent of this directory.
    * **children**: This list holds the directories and files under this directory, if any.
  - This class contains the following functions:
    * **__init__**: This function is the constructor of the class, which is run on class instantiation.
    * **add_child**: This function takes the name and instance of either a Directory of File, and adds it to the children list of this directory.

        * **find_child**: This function takes the name of a child and returns a handle to the child.

- **File**: This class represents a file in the virtual filesystem. File instances are stored within Directory instances.
    - This class contains the following attributes:
        * **name**: The name of the directory.
        * **parent**: The parent of this directory.
        * **fd**: The file descriptor of this file while open, which is a unique integer for the current instance of the shell.
        * **content_size**: The byte size of the content within the file.
        * **offset**: The current byte offset of the file, which is used when reading from the file.
        * **date**: When the file was created.
        * **fsa**: The FileSystemAllocation block associated with the file.
    - This class contains the following functions:
        * **__init__**: This function is the constructor of the class, which is run on class instantiation.

- **Child**: This class represents a child of a directory as the directory would reference it.
    - This class contains the following attributes:
        * **name**: The reference name of the child.
        * **child**: The File or Directory child.
    - This class contains the following functions:
        * **__init__**: This function is the constructor of the class, which is run on class instantiation.

- **OpenFile**: This class represents an open file.
    - This class contains the following attributes:
        * **fd**: The file descriptor for the open file.
        * **mode**: The read or write mode of the open file.
        * **file**: The open File.
    - This class contains the following functions:
        * **__init__**: This function is the constructor of the class, which is run on class instantiation.

- **FileSystemAllocation**: This class represents disk space for the file system, both free and used.
    - This class contains the following attributes:

    - **offset**: The location in the disk of the space.
    - **size**: The size of the space on disk.
  - This class contains the following functions:
    - **__init__**: This function is the constructor of the class, which is run on class instantiation.

3.2 Shell

Our shell is a Python class named **Shell** that runs in a loop accepting input, parsing it, and calling the appropriate command, until a Ctrl+c or EOF command is sent through the terminal. The Shell class has the following functions:

- **__init__**: This function is called when the shell is instantiated. It initializes the instance variables of the class, including creating the main FileSystem object, the getting the current line of input, and setting the current directory the shell is within to root.
- **create_filesystem_object**: This function creates the filesystem for the first time.
- **run**: This runs the prompt loop, accepting input and passing it to the parse function to run the correct command based on user input.
- **parse**: This parses input passed from the run function and calls the appropriate commands after error checking, passing their arguments parsed from the input string.

3.2.1 Redirection

We implemented redirection simply by ensuring that the main way to call our shell, the **run_shell** script, supported taking input to stdin through the host filesystem's < symbol, and that it supported sending output to anything after the > symbol.

3.2.2 Commands

The commands for the shell are implemented as Python classes in **commands.py**. Each command class implements the **__init__** method and the **run** method at the very least, and some commands implement other helper functions.

Any file I/O will write over existing files on the system if an existing file is opened for writing. We did this because a mode of writing was not specified for the open command, which meant we have no way knowing whether the user wants to write or append. The simplest case was to overwrite the file if a file was opened and then written to, which is what we did. We implemented the following commands for the shell, including everything in the requirements document.

The commands we implemented are as follows:

- **mkfs**: This command initializes the file system on disk. It is always run when the program is first started.
- **open**: This command opens a file as either read only or write only (no append) and returns a file descriptor from the file system. This adds the file to the open_files variable of the filesystem object and begins tracking the file.
- **read**: This command reads a file in based on a given file descriptor. We read from the file and print it to the shell.
- **write**: This command writes to a file with a given file descriptor. We write the content passed to this command to the disk and update the File instance.
- **seek**: This command sets the current file offset of the file given by a passed file descriptor. We set the offset variable of the File instance for this file in the filesystem.
- **close**: This command closes a give file. When this is done, we remove the file from the open_files variable of the filesystem.
- **mkdir**: This command makes a directory in the filesystem, adding the directory to the children variable of either the root filesystem or the directory we are currently under within the shell.
- **rmdir**: This command removes a directory from the filesyste, recursively deleting everything under it.
- **cd**: This command changes into a directory from the current directory we are in the filesystem.
- **ls**: This command lists all the files and directories in a filesystem by printing out the children of the current directory we are in within the shell.
- **cat**: This command prints out the contents of the given file descriptor.
- **tree**: This command recursively lists the contents from the current directory using the children variable of the current directory and then all of the children of each directory within those children, if that child is a directory, otherwise it simply prints the filenames.
- **import**: This command imports a file from the host filesystem into the Bastion filesystem.
- **export**: This command exports a file from the Bastion filesystem to the host filesystem.

4 Challenges

We faced several challenges while building Bastion.

- Writing the code in Python proved to be a challenge at times due to not having full control over memory allocation, though we were able to get around this by virtualizing our file system and representing it in memory as a set of interrelated Python classes. We then serialized the filesystem contents to disk on the relevant command calls.
- Choosing the file serialization format was a problem, as we originally decided on the Pickle specification, a Python serialization protocol to turn Python objects into binary files on the system. However, we felt this did not meet the on-disk requirement for the project, causing us to switch to our own format.
- Deciding which 3rd-party modules to introduce in our codebase introduced more complexity than what we wanted to tolerate, so we eventually decided to simply write everything in pure Python, and use only the standard library. We believe this paid off in the end as it made our code more readable.

5 Conclusion

In summary, we built Bastion, a filesystem and shell with a wide range of available commands and a fully functional disk format. We implemented the system in Python, and were able to achieve successful results against the testing script. Writing the system in Python sped up our development and enabled us to focus on perfecting the software quality of our system, rather than dealing with arcane memory bugs and security issues.