G. Bernard, N. Lockett, R. St. Pierre, M. Wu
(ghb5, nsl8, ras70, mlw55)
Group 5

Project Outline: Archival Write-Once File System

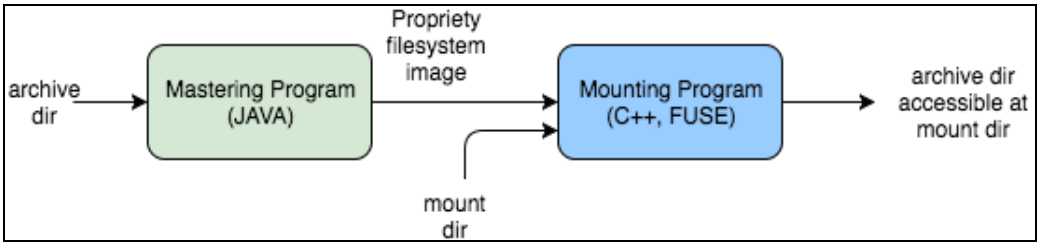
Overview/Changes Since Proposal

Our project goal is to implement an archival, highly available write-once file system. Major changes since the proposal include:

- Decision to use a proprietary image format for our filesystem rather than to meet *.iso* or other archive standards.
- Changes to our extension goals:
  - Previously, our stretch goals were compression and bit level error correcting code.
  - During our proposal we found this does not make sense: space efficiency should not be an issue and bit level error correcting code should be handled by our program.
  - Instead, we will focus on block level failures, availability, information security, and ease of data extraction.

Project Architecture

Our project will consist of two main components – mastering and mounting. A description of our architecture is included below. The mastering program will accept a directory and convert it to a file of proprietary type that can be leveraged by our mastering program.



WOFS pipeline overview

Mastering program

- *Language:* Java
  - Benefits:
    - A familiar language will minimize the development learning curve as well as time to market.
    - *File* class gives functionality to recursively traverse the directory.
  - Con: Performance Hit.
- *Platform*: Linux
- *Algorithm:*

- ➢ Two Passes of filesystem '*tree*'
  - ■ First, build graph based on existing filesystem
    - Store file metadata for each node
    - Link from directory node to sub nodes
    - Link from soft or hard links to intended address
  - ■ Second, traverse metadata graph, open files, and write them (along with the tree structure) to the image file.
- ❖ *File System File Representation:* Proprietary – aspects of FAT and ext2 maximized for write-once file system.

Mounting program

- ❖ *Language:* C/C++
  - ➢ Benefits:
    - ■ Natural access to libfuse
    - ■ Since FUSE likely requires traversing from node of filesystem tree each call, will need fast pointer traversal to recoup performance cost
  - ➢ Cons:
    - ■ More difficult to develop in
- ❖ *Platform:* Linux
- ❖ *Libraries:* FUSE (Filesystem in userspace)
- ❖ *Algorithm:* Pass in the mastered file, mount using FUSE, implement the read and directory methods.
  - ➢ Each call to FUSE traverses from the root of the mastered image file along the path given to the function – **not** bulk loading and making tree structure at mount time.

Timeline

Below are our project milestones. By the date given for each milestone, we expect to have the deliverables described completed.

Milestone 1: Getting Started **6 October**

By the point, we will have a well defined interface between the two programs – the structure within the filesystem image. This will allow us to work as concurrently as possible and realize potential implementation issues before we begin real development.

- ❖ *Deliverables:*
  - ➢ Well defined structure for the filesystem image file.
  - ➢ Data structure to house the mastering information.

- *Group Members:* Since design is essential to the success of this project, all members will work together on this.
- *New Skills*: Understand the Unix file system and representation better.
  - What metadata information does Unix maintain and what do we want to include?
  - What Unix capabilities complicate the tree structure of the file system?
- *Server Usage:* Occasional light usage to examine file metadata and Unix protocols.

Milestone 2: Implement Mastering **15 October**

By this point, we will have implemented the basic mastering utility. This must come early in the project timeline, since it is the major dependency for the project.

- *Deliverable:*
  - Alpha implementation of WOFS mastering program
- *Group Members:* Leads – Nick and Matthew
- *New Skills:*
  - Java *File* class
- *Server Usage:* Light server usage for testing, most likely in the evening from 6 to midnight. Most development will be done in Java on local machine.

Milestone 3: Implement Mounting **22 October**

By this point we will have implemented the basic mounting utility.

- *Deliverable:*
  - Alpha implementation of WOFS mounting program
- *Group Members:* Leads – Ryan and George
- *New Skills:*
  - Libfuse, FUSE
- *Server Usage:* Significant usage for testing purposes, most likely in the evening from 6 to midnight.

Milestone 4: Milestone Presentation **25 October**

By the time of the presentation stage, we should have a working prototype, and be ready to present.

- *Deliverable:*
  - An integrated, demonstrable implementation of WOFS (mastering and mounting)

- ➢ 5 minute demo highlighting our current status
- ❖ *Group Members:* All

Milestone 5: Testing
Mid-November
This milestone exists to polish and maximize test coverage for our utility.

- ❖ *Deliverables:*
  - ➢ Integration test pipeline for maintaining compatibility between both mastering and mounting programs, but also legacy versions of both programs
  - ➢ Unit test suite for mastering and mounting programs.
  - ➢ Beta version of mastering and mounting programs with bugs removed from alpha version.
- ❖ *Division of Labor:*
  - ➢ Ryan and George: mounting unit and integration testing.
  - ➢ Matthew and Nick: mastering unit testing.
- ❖ *New Skills:*
  - ➢ JUnit
  - ➢ Catch

Milestone 6: Wrapping Up **31**
November
We have left a week to complete any deliverables that are not completed by their desired date. Additionally, this time will give us time to explore possible extensions (discussed in further detail in a later section).

Milestone 7: Final Demo **7**
December

- ❖ *Deliverables:*
  - ➢ Finished, fully polished final product with accompanying test suit
  - ➢ 15-minute rehearsed presentation

Division of labor

We will attempt to create the mastering and mounting programs as concurrently as possible. For this reason we will appoint two different teams, detailed below. However, the mounting program has a heavy dependence on the mastering program. Until the mastering program is complete, the mounting program team will have to create filesystem manually, which limits the complexity of what can be tested. For this reason, we will put an emphasis on the mastering program until it is complete.

- Mastering team: Nick and Matthew
- Mounting team: George and Ryan

Testing

- *Unit Testing frameworks:* JUnit (Java), Catch (C++)
- *Building:* Makefile for each utility.
- *Integration Tests:* We will need to test the interface between the master and mounting program.
  - This will likely be implemented as a *bash* script, or set of *bash* scripts, that will automatically input a test directory into the mastering program, mount the resulting file system image, and monitor the output of different linux commands on the mounted file system.
- *Performance:* We anticipate manually benchmarking our mastering and mounting utilities through the development process as user experience is impacted.

Updated Criteria of Success

- Functional correctness (i.e. it works). More formally, this means the file system is capable of mastering, mounting, and reading.
  - Master program creates a consistent and independent image file.
  - Master program capable of mastering directories with any file types and with tree structures including the following linux constructs: soft and hard links.
  - Mounting program is successfully able to mount any properly generated image file from the mastering program.
  - Correctly notify users of warning, errors, and failures of the program.
- Completing all milestones in a timely manner.
- Space and time efficiency of the file system.
  - No observable latency while using mounted file system.
- Full testing suite:
  - 80% code coverage.
  - Unit and integration tests.
- Extremely successful if we implement one stretch goal

Extensions

1. *Distributed File System with fault tolerant read calls*

Archive systems should be highly available. This means archives should be fault tolerant in the long term. A good archival system should withstand not only a block or drive failure, but a full server failure, perhaps even a full datacenter failure.

- A fitting extension for an archival system like this could be to distribute the image file across multiple machines with a corresponding checksum.
- If an error is encountered, or if checksums differ across image files, the failed image can be reconstructed across the network.
- In a complete implementation, if a failure is encountered while parsing the image file, the data could be requested across the network, ensuring availability with a performance hit.
- The images could then have their checksums compared and the failed one could be copied from valid images.
- **Note:** this will not stop malicious attackers. That fault model would likely require something akin to blockchain or cryptographically secure hashing and is unfeasible to implement on our timescale.

2. *Encryption*

Archive file systems may hold sensitive data, e.g. HIPAA. For these use cases, a strong measure of data security is required. Archival encryption, based on would achieve two things: hard to read, and hard to edit.

- Following a Encrypt-Then-MAC procedure, the file system would be authenticated before it was unencrypted, keeping the underlying filesystem secure.
- Since any enterprise system should never roll their own encryption services, interfacing with a field tested and verified encryption library would be necessary.
- **Note:** Encryption schemes would have to be weighed on merit. You don't want to decrypt the entire tree every time you traverse it.
    - This could require having the tree structure be separate from the actual underlying data in its one set of well known encrypted blocks.
    - But now, attackers know what section of your filesystem they need to brute-force in order to find out the smaller section of data they need to brute-force.
    - Perhaps decrypting as they traverse could be viable, but this is still a difficult problem that would need to be implemented perfectly if at all.

3. *Tree Command*

This extension would be smaller than the other two and involves writing a program that traverses our file image and displays its structure before mounting. This would be a useful program for the user and also serve as a testing tool for the mastering program.

Risks and Challenges

1. Learning Fuse/other tools: With any project, learning new tools essential to the project's success, such as Fuse, presents a risk when attempting to meet a strict deadline.
2. Dependency between mounting and mastering programs could present difficulties in the case that mastering takes longer than expected or does not work as expected, since mounting depends on a fully functioning mastering program.
3. Significant reliance on a single, early defined interface presents challenges if it turns out that the interface is actually not that well-suited for the project.