# Write-Once File System

George Bernard
Nick Lockett
Ryan St. Pierre
Matthew Wu

# Overview

## Write-Once File System

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

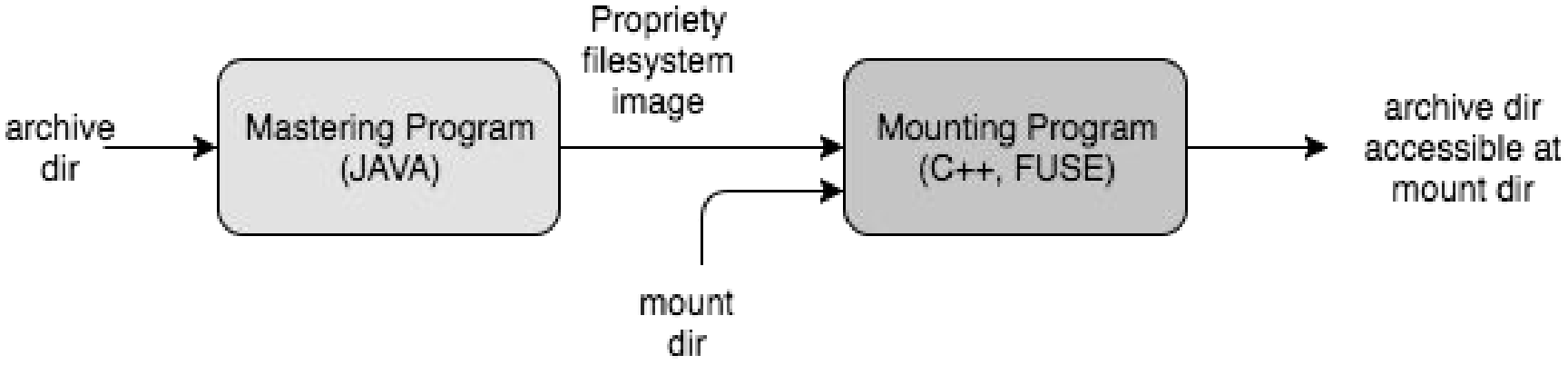
# Major Changes

- 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

Write-Once Filesystem

# General Architecture



# Mastering Utility

- *Language:* Java
- *Platform*: Linux
- *Algorithm:*
  - Two Passes of filesystem '*tree*'
    - First, build graph based on existing filesystem
    - Second, traverse metadata graph, open files, and write them (along with the tree structure) to the image file.
- *File System File Representation:* Proprietary

# Mounting Utility

- *Language:* C/C++
- *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

# Timeline

Write-Once Filesystem

# Milestone 1: Getting Started - October 6

- *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.
- *Server Usage:* Occasional light usage to examine file metadata and Unix protocols.

# Milestone 2: Implement Mastering - October 15

- *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 - October 22

- *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 - October 25

- *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

- *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.

# Testing

- *Unit Testing frameworks:* JUnit (Java), Catch (C++)
- *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.

# Milestone 6: Wrapping Up - November 31

- *Flexible time for any extensions or completion of any other deliverables*

# Milestone 7: Final Presentation - December 7

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

# Criteria of Success

Write-Once Filesystem

# Primary Criteria

- 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.

# Additional Criteria

- 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

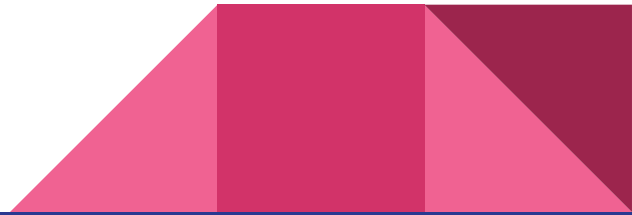
Write-Once Filesystem

# Distributed File System with Fault Tolerant Reads

- 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.

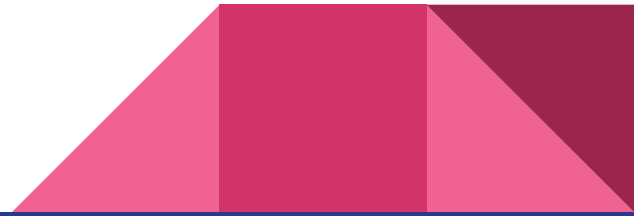
# Encryption

- 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.



# 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.



# Major Risks and Challenges

- 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.
- 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.
- 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.