Technical Report: Project II

Building an Atomic and a Multi-level Persistent Tree File System

Muhammad Aater Suleman and Youssef Hmamouche | 12/02/03

Abstract

This report details the testing, analysis, and discussion of Project 2: Building a Reliable File System.

Table of Contents

1. Introduction

In this project we have written multiple File Systems using the low-level driver Disk implementation provided to us by Dr. Harrick Vin. The File System implementations are accompanied by testing procedures to guarantee the healthiness of the implementation in question. We have provided a test suite for each implementation.

2. Documentation and File Listing

2.1 File Listing

2.1.1 Source Code Files

ADisk.cc: The Atomic Disk implementation
Disk.cc: The low-level disk implementation

PTree.cc: The persistent tree implementation
sthread.cc: The underlying tools needed to build an Atomic file system
testpart0.cc: Stress tests for Part0 of the project
testpart1.cc: Stress tests for Part1 of the project
testpart2.cc: Stress tests for Part2 of the project

2.1.2 Header Code Files

ADisk.h: header file containing subroutine prototypes
Disk.h: header file containing function prototypes of the disk raw operations
FS.h: prototype functions header file for the FlatFS
FlatFS.h: prototype functions of the Flat FS
PTree.h: prototype functions of the Persistent Tree FS
common.h: defined constants
dirent.h: dentry definition
sthread.h: the thread library include file

3. Testing of File System

Our tests range for multiple reads and writes to crash tests and recovery mechanisms. Depending on the file system, we have written specific procedures that target those features that are more likely to cause a problem. However, we have written certain rules that must be met as a minimum.

Sequential "writes" followed by "reads" are a good index of the healthiness of the simple tasks of our file system. However, we have written more specific tests for the PTree, and

the Atomic file system. The former is delicate in a way that we have to make sure data blocks can be dynamically allocated. We ensured that updates to the trees occur atomically and also made sure to update the free list to signal that blocks have been used. We tested the latter in a way that crashes occur frequently and ensuring the file system is able to recover itself. We also ensured that writes do occur entirely as opposed to partially as a feature of an Atomic file system.

We developed command line interfaces that would allow us to communicate with all levels of the disk system hierarchy. Hence, we were able to compare the data actually written on the disk with our expected results.

4. Technical Discussion of Design

4.1 The Atomic File System

The Atomic File System ensures the atomicity of the transactions to avoid any partial reads or writes and therefore corruption in case of a crash. This atomicity is implemented through transaction logs. A redo log consumes a number of sectors specified as a constant. We have also optimized our implementation in a way that it combines multiples writes to the same block. This especially optimizes the PTree to a large extent by reducing the time taken to commit a transaction. The disk layout looks as the following:

ADISK REDO LOG	0 - ADISK_REDO_LOG_SECTORS
DATA	NUM_OF_SECTORS

4.2 The Persistent Tree File System

The PTree file system sits on top of the Atomic Disk abstraction to avoid inter-thread data corruption. The PTree file system is characterized by its tree-like shape with the leaves being the data blocks. Our file system is limited as far as disk space is concerned for illustration purposes. The tree is statically limited to MAX_TREES trees. The internal nodes can grow at will.

ADISK REDO LOG	0 - ADISK_REDO_LOG_SECTORS
PARAMETERS	PARAM_USE
TNODEARRAY	TNODEARRAY_USE
BITMAP	BITMAP_USE
DATA	NUM_OF_SECTORS

5. Conclusion

This project has taught us how to build a reliable file system that is immune to data corruption after unexpected system crashes. Overall, we are quite happy with our implementation since our tests passed successfully.