Swansea University
Prifysgol Abertawe

College of Science

Coursework Submission

Module	**CS-354**
Coursework	**Interim Document**
Lecturer	**Mr. Christopher Whyley**
Deadline	**13 Dec 2013 11:00 AM**
Student Number	**644475**

RCU File System: An Investigation into the performance characteristics of Read Copy Update in a File System

Interim Document

Callum Massey - 644475

December 13, 2013

Contents

1 Introduction

In the initial document I attempted outline the background and motivation for this project to create a file system that implements Read-Copy-Update as it's internal locking mechanism. In this document we discuss the progress made since then and discuss deeper the technical design choices that have been made so far. We also discuss further the background surrounding the project specifically in relation to the Read-Copy-Update synchronization mechanism. In section 3 we discuss the remaining tasks that need to be completed that are in the areas of code implementation and testing. In the final section we attempt to draw conclusions and report on how the project is progressing in general.

2 Background

2.1 Technical Details

2.1.1 FUSE

Filesystem in User Space is what I am using as a basis of my FS to allow me to focus on the complex implementation of RCU without having to worry about the kernel level implementation of the file system. In order to implement FUSE you need at a minimum the following FUSE operations;

```
static int getattr(const char *path, struct stat *stbuf)

static int open(const char *path, struct fuse_file_info *fi)

static int read(const char *path, char *buf, size_t size,
                      off_t offset, struct fuse_file_info *fi)

static int write(const char *path, const char *buf, size_t size,
                      off_t offset, struct fuse_file_info *fi)
```

These are at a minimum the functions I will need to implement, however as time permits I intend to implement the following functions as time permits in approximate order of relevance:

```
static int mkdir(const char *path, mode_t mode)

static int rename(const char *from, const char *to)

static int truncate(const char *path, off_t size)

static int statfs(struct fuse_statfs *fst)
```

Most of these functions will be simply passing their commands through to the underlying file system library for example I do not expect to change getattr and statfs from their standard implementation.

2.1.2 FAT File System

For the underlying file system which I will be building the wrapper around I will be using FAT, this is because FAT is the simplest of any viable file system for this project. This stems from FAT being a very simple file system and while it is rather too simple for serious modern use, it should allow for the purest comparison of performance without other FS features masking the difference. As an example the implementation of the write function for FAT in FUSE, without error checking, is as simple as;

```
static int fusefat_write(const char *path, const char *buf, size_t size,
    off_t offset, struct fuse_file_info *fi)
{
  int res;
  File_t *F;
  Volume_t *V;
  fusefat_getvolume(V);
  F = (File_t *)fi->fh;
  fat_lock(V);
  fat_seek(F, offset, SEEK_SET);
  fat_write_data(V, F,&(F->CurClus), &(F->CurOff), buf, size );
  fat_update_file(F);
  fat_unlock(V);
  return size;
}
```

To this end I will be basing my code upon fusefat mainly adding the RCU code.

2.1.3 URCU

As this code is being designed to run at a userspace level I will not be able to use the kernel level RCU implementation, as such I will be using the Userspace Read Copy Update (URCU) library for my RCU implementation. This has major advantages over attempting to create my own implementation as it is a tried and tested implementation of the read copy update library.

2.2 Further Background

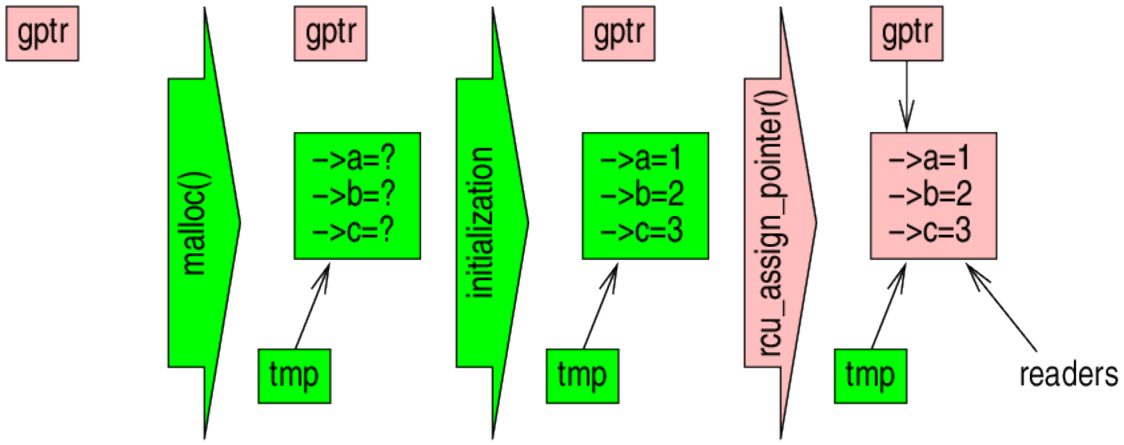
2.2.1 RCU

The Read Copy Update algorithm allows you to have coherent access to data structures even while they are undergoing updates by another process. It does this by, before making any changes, copy the structure you intend to modify to a new location and then perform the modifications to this struct while other processes can still access the old copy until the write operation is finished, then an internal record of where each structure maps to is updated to point at the new location. Once the read operations are completed on the old copy the space is marked as empty.

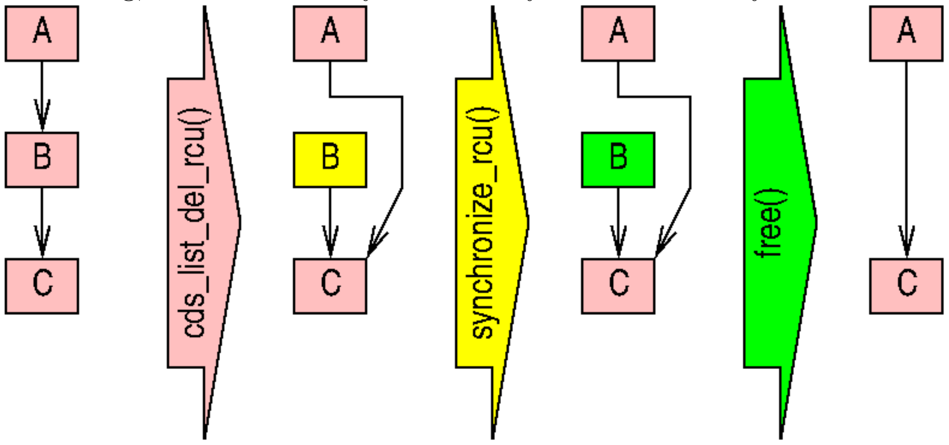
This allows for a improved read performance under real world loads where multiple processes are reading and writing the data at the same time. Especially under more read heavy loads as there is a very minor cost in the copying of the data though on modern hardware this is negligible.

This performance increase mainly comes from the absolute lack of blocking for all readers, which is implemented at a fundamental level within RCU.

To insert an element into an RCU structure, as shown below, you start with a global pointer set to a NULL value, you then malloc some memory, initialise that memory, and then use rcu_assign_pointer to assign the global pointer to the address of the tmp pointer. You use this method to prevent the compiler changing the order of execution and making the pointer return invalid values.



To remove an element from an RCU list you first use cds_list_del_rcu to remove the pointer to B so that no new access to that pointer can be made as it no longer exists. Now you must wait for the current readers to finish, you can tell your code to wait until all readers that could possibly have access to that memory address using synchronize_rcu, this functions waits until all readers that started before it was executed to finish before continuing, then afterwards you can safely free this memory.



3 Revised Project Plan

3.1 Progress

In the initial document it was planned that I would be much very along in my project however this was a very optimistic plan as I have account significant issues with find the available time to invest as well as the required amount of work being much higher than expected. I have performed extensive research into the inner workings of the RCU implementation as well as how, at a fundamental level, file systems are defined and implemented. Another major source of difficulty has been understanding of advanced C programming, especially regarding how block device access is handled. I have overcome this through reading source code as well as literature sources.

3.2 Remaining Tasks

3.2.1 Develop Testing Framework

The most significant part of this project will be running performance tests and creating those test cases. The usefulness of this file system will ultimately be determined by it's performance in real world scenarios, as such I intend to create several viable test to run against it. To this end I intend to create a database benchmark that will involve running a standard database engine on the file system and then performing various simultaneous task on it. I will also create a benchmark test using dedicated benchmarking tools i.e. sysbench.

3.2.2 RCU Implementation

The bulk of the coding work relates around how I am going to be handling writes and what method I will use to handle tracking the current and previous data locations. I plan on implementing this using, at first, an in memory structure containing original and current file descriptors that I will use to lookup the real address of any anything I need to access before doing so.

4 Summary

Learning the underlying workings of file systems and block device access took significantly longer than I had anticipated. Hopefully I should not have to do any further significant research during my project. The biggest challenge going forward will be creating the test suite to benchmark the new file system.