
Software Requirements Specification

for

Linux File RAID [SQA Example]

Version 1.0 draft

**Prepared by Sam Siewert
[To be updated, extended and improved by students]**

**Embry Riddle Aeronautical University
Computer, Electrical, Software Engineering
Software Engineering Program**

February, 19, 2018

Table of Contents

Table of Contents	ii
Revision History	iii
1. Introduction.....	1
1.1 Purpose	1
1.2 Document Conventions	1
1.3 Intended Audience and Reading Suggestions	2
1.4 Product Scope	2
1.5 References	3
2. Overall Description	3
2.1 Product Perspective	3
2.2 Product Functions	7
2.3 Use Cases	9
2.4 Use Case and Requirements Tracing	10
2.5 Operating Environment	10
2.6 Design and Implementation Constraints	11
2.7 User Documentation	11
2.8 Assumptions and Dependencies	11
3. External Interface Requirements	12
3.1 User Interfaces	12
3.2 Hardware Interfaces	12
3.3 Software Interfaces	12
3.4 Communications Interfaces	12
4. System Features.....	12
4.1 System Feature 1	13
4.2 System Feature 2 (and so on)	13
5. Test Driven Design Reference Prototypes	13
5.1 Checkout and build	14
5.2 Running Unit tests	16
5.3 Path coverage	18
5.4 Profiling	19
5.5 I&T scripts	19
5.6 System tests and scripts	21
5.7 Regression tests and scripts	22
5.8 Acceptance tests	25
6. Other Nonfunctional Requirements	26
6.1 Performance Requirements	26
6.2 Safety Requirements	26
6.3 Security Requirements	27
6.4 Software Quality Attributes	27
6.5 Business Rules	27
7. Other Requirements	28
Appendix A: Glossary.....	28
Appendix B: Analysis Models.....	28
Appendix C: To Be Determined List.....	29

Revision History

Name	Date	Reason For Changes	Version
Sam Siewert	2/19/2018	Initial creation or draft to provide documentation for prototype.	1.0

1. Introduction

Note that this reference design included reference prototype code, which can be obtained from a non-managed source for convenience as noted in the text below, but the ideal place to obtain the latest reference code corresponding to this documentation is github (<https://github.com/siewertserau/Examples-RAID-Unit-Test>). I will make every attempt to keep the code and documentation in reasonable synchronization, but it is up to the reader and the software engineer re-using this example to “make it their own”. If you want to check in improvements, you can request to be added to my github, but otherwise the intent is for you to clone as a starting point and fork your own project including documentation, analysis, design, and code files.

1.1 Purpose

This SRS provides an overall specification for a Linux File RAID version 1.x.x (Redundant Array of Inexpensive Disk) library of operations as well as example application that can be used to create storage as a service, similar to commercial products such as [Google drive](#), [MS One drive](#), [AWS S3](#), or [Dropbox](#). This specification and the associated software is not intended to be a product or Cloud service, but serves as a pedagogical example of how to analyze, design and build software like these products, starting with the data storage and protection services, to which students can add features such as web access, security, user friendliness (e.g. browsing), and a wide range of additional features to create their own STaaS (Storage as a Service) application. File level RAID was selected rather than block storage DAS (Direct Attach Storage) or SAN (Storage Area Networking) for simplicity and portability so that a derived STaaS application can be hosted almost anywhere on an platform file system. The Linux File RAID software is composed of a library of RAID write (encode), read (decode), rebuild (reconstruction of lost or corrupted file chunk) and basic data compare functions, a file storage and retrieval system with data protection, and unit test code for the library along with basic system testing for the file RAID application. The library code is intended to be re-useable (perhaps with modification and extension) for use in new file RAID applications for new platforms or to provide new STaaS with a web interface. The goal is to support learning objectives related to software analysis, architecture, design, and implementation quality assurance along with testing examples at the unit, I&T (Integration and Test), system, and acceptance testing levels.

1.2 Document Conventions

This document is based upon the IEEE SRS 830-1998 template and documentation standards for software requirements and specification ([IEEE 830-1998](#), [29148-2011](#)).

In this document, UML analysis and design methods are used, where requirements must correlate to use cases, which have hierarchy and priority. Thus, the requirements are prioritized and placed into hierarchy based upon UML use case hierarchy and priority analysis. This is consistent with principles learned in SE 300 (SA/SD - Structured Analysis and Design), SE 310 (Object Oriented Analysis - OOA, OO Design - OOD, OO Programming - OOP) and SE 420 (Software Quality Assurance).

1.3 Intended Audience and Reading Suggestions

The primary readers of this document are anticipated to be upper division undergraduate students working on exercises and projects related to coursework in software engineering analysis, design, and quality assurance. Given the educational goals, this SRS contains both SA/SD as well as OOA/OOD methods of specification such that the analysis and design supports both standard modular procedural programming implementations (e.g. C) and object oriented class structured programming implementations (e.g. C++).

The SRS contains a complete set of:

- 1. Concept goals and objectives,*
- 2. Requirements identification through use case analysis,*
- 3. System, architecture and module analysis and design (SA/SD and OOA/OOD) including:*
 - a. Requirements specification,*
 - b. System use cases and requirements consistency, completeness and correctness analysis*
 - c. Architectural module cohesion and coupling analysis for SA and/or CRC (Class-Responsibility-Collaboration) analysis for OOA*
 - d. Module design (architectural decomposition into abstract and concrete modules or classes and interfaces between them and behavior),*
 - e. Modules detailed design and prototype (detailed structural and behavioral specification of modules – class and/or package diagrams and directory structure as well as flow-charts or activity diagrams, interaction sequence diagrams, and state machines) along with C or C++ module prototype code.*
- 4. Requirements validation and verification with acceptance test plan*
- 5. System design (use cases and high level abstract modules) and end-to-end test plan of a CSCI (Computer Software Configuration Item)*
- 6. I&T test plan for architecture CSC (Computer Software Component)*
- 7. Unit tests for modules or CSUs (Computer Software Units)*

Overall, the SRS is intended to provide a reasonably complete specification of the example Linux File RAID software at the level of a working example that can be improved by students for practice and as a starting point if they wish for a more complete STaaS final project.

1.4 Product Scope

Goals for this SRS and associated, analysis, design, testing validation and verification and prototype software are strictly educational. The learning objectives include experience reviewing, improving, refactoring, finding defects, composing fixes and regression testing, all levels of analysis, design, and implementation of this example.

Specific objectives include learning objectives related to each phase of the “V” model, with an [Agile](#) evolutionary approach to analysis, design, development and test. Specficially, it is assumed that rather than strict waterfall phases of development, students will make quick two week sprints through phases of analysis, design, and implementation with a test-driven design approach (consideration of testing first or at least concurrent with design).

By the time a student is done reviewing and improving this reference SRS and associated software at the unit level (CSU – Computer Software Unit) and test application (CSCI – Computer Software Configuration Item) level, they should have a good working knowledge of test-driven evolutionary software development. This can also be a starting point for a longer term project or

even a capstone design project to build an entire system service (e.g. STaaS) or platform hosted application suitable for deployment.

1.5 References

1. <https://www.dau.mil/glossary/Pages/Default.aspx>
2. International Organization for Standardization/International Electrotechnical Commission/Institute of Electrical and Electronics Engineers (ISO/IEC/IEEE) Standard 24765:2010: Systems and Software Engineering – Vocabulary
3. <https://www.computer.org/web/swebok/v3>, Bourque, Pierre, and Richard E. Fairley. *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0*. IEEE Computer Society Press, 2014.
4. <http://agilemanifesto.org/>
5. <https://www.omg.org/spec/UML/2.5.1/>, UML 2.5.x specification
6. Kendall, Penny A. Introduction to systems analysis and design: a structured approach. Business & Educational Technologies, 1995.
7. Rumbaugh, James, Ivar Jacobson, and Grady Booch. *Unified modeling language reference manual, the*. Pearson Higher Education, 2004.
8. https://raid.wiki.kernel.org/index.php/RAID_setup, Linux software RAID using MDADM.
9. Welch, Brent, et al. "Scalable Performance of the Panasas Parallel File System." *FAST*. Vol. 8. 2008.
10. Libes, Don. *Exploring Expect: a Tcl-based toolkit for automating interactive programs*. "O'Reilly Media, Inc.", 1995.

2. Overall Description

2.1 Product Perspective

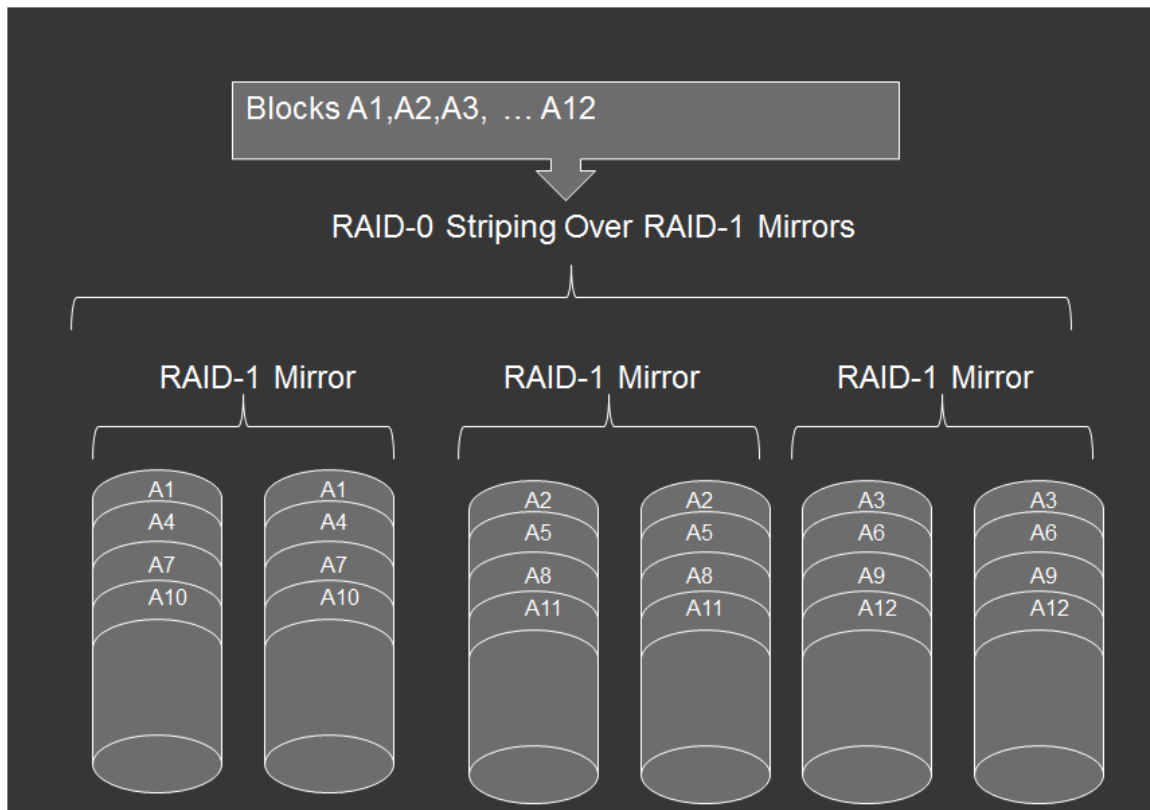
The purpose of the Linux File RAID software libraries and example application is to provide a working example of data storage with data protection and to serve as a starting point for a student wishing to build a more complete application or storage service and to practice QA and analysis and design methods to improve this example. As such, the design includes a basic library of RAID operations and the example File RAID application simply provides ability to store files in chunks that can be mapped to distinct and separate storage subsystems to prevent loss of data when a storage subsystem fails or is lost (a chunk erasure). While RAID is most often implemented at a block and HDD (Hard Disk Drive) or SSD (Solid-State Disk) device level, it can also be implemented at a file level where large files are automatically broken down into smaller chunks (sub-files) that are distributed across multiple file systems to provide data protection equivalent to block level RAID, but with more portability and simplified testing compared to block level RAID.

For students, it is not practical to expect them to have access to SAS/SATA (Serial Attached SCSI or Serial Attached ATA) disk array with root access on a Linux system, so the Linux File RAID allows them to learn about RAID and at the same time learn about software system (service) and

application analysis, design, implementation and test. Students wishing to compare block RAID to file RAID may want to consult Linux MDADM (Multi-Disk Administration) and examples of use with RAM (Random Access Memory) disks for tutorials on configuration, use and operation of software RAID without dedicated HDD arrays. The Linux File RAID is mostly for learning and practice with RAID, but could be used in an actual deployment as long as each file chunk is mapped to a unique file system.

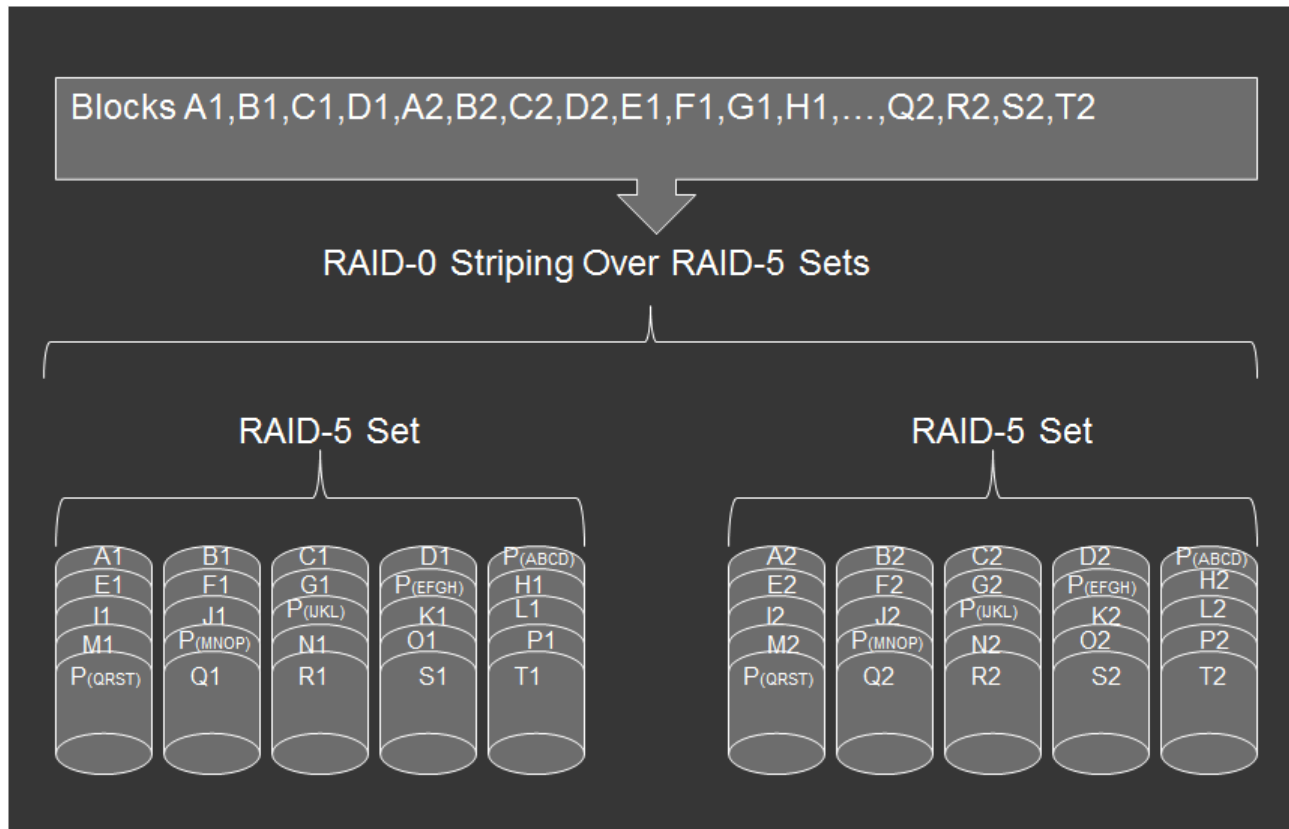
RAID has levels including striping (level 0), mirroring (level 1) and XOR parity (level 5) as well as double XOR parity (level 6). These levels can be combined so that unique block devices are striped and mirrored (level 10 or sometimes called 0+1) or striping of XOR parity sets (level 50). Furthermore, the number of independent disks (or file systems) is variable, but normally ranges from the minimum of a mirror pair (two disks or file systems) up to any number of three or more for XOR parity schemes. The fundamentals of RAID systems are covered in CS 317, file and database systems, and many excellent references and examples such as Linux block level software RAID (MDADM). The same principles used to map blocks (typically 512 bytes of data at a time up to 4 kilobytes) onto a set of independent HDD or SSD storage devices can also be used to break a large file up into file chunks (sub-files) that can be mapped onto multiple independent file systems. The block level is the most common RAID system type where a file system is then built on top of this new RAID volume (virtual block storage device). However, file level is used in real products (e.g. Panasas - <https://www.panasas.com/>) and serves as a convenient alternative to dealing with low-level block devices. All of the same data protection can be achieved as long as file chunks are mapped to independent file systems rather than independent storage devices. This approach is actually taken for scale-out file systems (federated file systems) and is an alternative to SAN (Storage Area Networking) and can work with NAS (Network Attached Storage) or be used as a NAS alternative. It is potentially a great way to provide STaaS (Storage as a Service) where the main goal is to provide a “bit bucket” to share files, back them up, or provide a web based distribution for files. Figure 1 shows a basic RAID 10, where mirror pair block level storage devices (HDD or SSD) are striped.

Figure 1. RAID-10 striping and mirroring of data blocks on 6 storage devices



The idea behind RAID-10 is to provide speed-up for reads and writes by using multiple disk drives in parallel to read portions of a larger set of blocks composing a file or other storage object and at the same time, mirror blocks onto multiple independent disk drives in case one fails. If one does fail, a data erasure, the lost data can simply be read from any device that mirrors the same data. When the failed disk drive is replaced by a disk array technician, the system can restore the mirroring automatically. So, RAID has two main system level design goals – speedup by writing large data sets to many disks in parallel by chunking the original data into blocks as well as providing protection against data loss due to temporary device malfunction or failure. The key to RAID is that the blocks (or more generally chunks) of data must be allocated (mapped) to independent devices or file systems to provide actual speed-up and data loss protection when devices (or file sub-systems) fail. In order to simply learn about RAID and practice with the operations, actual independent devices are not required, but not speed-up and no failure fault-tolerance will actually be realized – for example if file RAID is used with just one file system rather than multiple independent. However, not only can one learn about RAID, the basic features and functions can still be tested, and when a system with “n” file systems or “n” devices is available, then the speed-up and failure fault tolerance can likewise be tested (performance constraints and error recovery requirements). Figure 2 shows a basic RAID 50, where XOR parity block level storage devices are striped.

Figure 2. Striping of data blocks over two 4+1 distributed parity RAID-5 sets

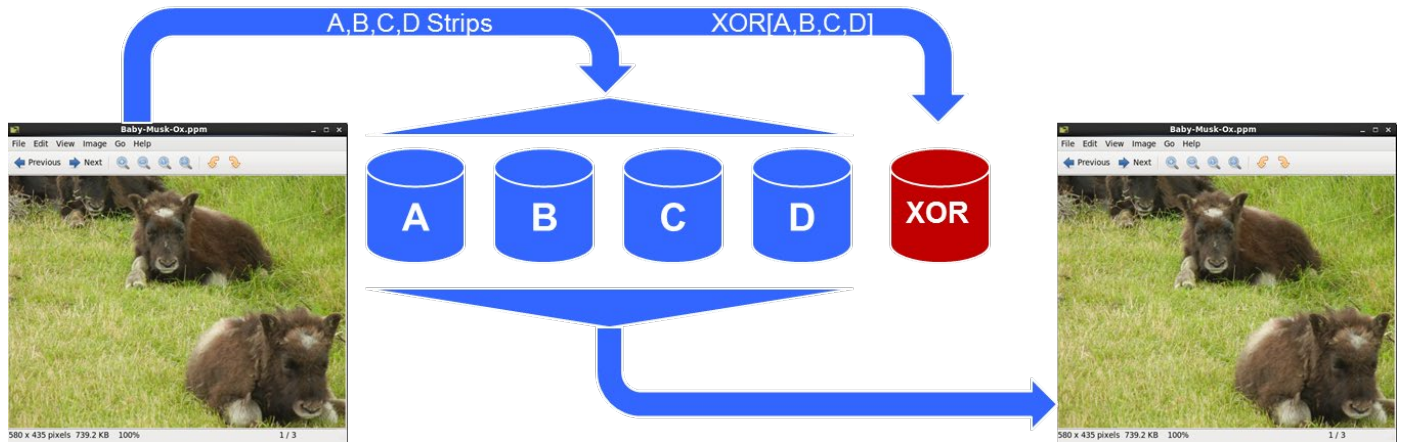


For RAID-50, RAID-5 sets are striped and each RAID-5 set of devices can in fact be any number of devices as long as each set is one parity plus at least one or more data devices – typically $n+1$ (5 shown here). While it's possible to make one of the devices a dedicated parity device, most often the parity is distributed, and parity for any particular write set (n blocks or chunks) must simply be placed on an independent device (+1 device) so that any missing block can be recovered by re-computing the parity of the remaining blocks.

Due to ability to emulate and test a wide range of RAID configurations with file RAID, the design does not provide any block level RAID capability, only file chunk RAID. This supports learning objectives better by allowing students to run code on any Linux system (and easily port the solution to a wide range of POSIX compliant systems such as Mac OS-X, Free BSD, etc.)

To simplify and allow for experimentation with RAID anywhere, the Linux File RAID simply models each independent storage chunk in a RAID mapping as a sub-file that composes a larger file as shown in Figure 3.

Figure 3. Linux File RAID Dataflow



2.2 Product Functions

This section summarizes the major functions the Linux File RAID system or application must perform. Details are provided in Section 3, so only a high level summary list is provided here. The functions are organized into high level capability requirements, test requirements (for test driven design goals) and system constraints (performance, scaling, sizing). A class diagram is provided that goes beyond what is currently prototyped to provide ideas for extending the design to include additional user authentication, access control, and session features as well as file, directory and file date, time, and version information.

The product must perform:

1. Storage of entire file as chunks on file systems A, B, C, D and a dedicated XOR file system
2. Read of chunks contained in a file from A, B, C, D and XOR files to reconstruct the originally written file
3. Recovery of any lost chunk from the original file in order to restore the file
4. Detection of corruption of any chunk and recovery of data if one chunk is corrupted
5. Ability to map chunks to any file system (same for testing) or independent (for safe use)
6. Management of original file names and association with storage chunks for reconstruction and basic file manipulation operations (copy, delete, rename)
7. Error reporting if two or more chunks are lost or corrupted (recovery is not possible)

Test requirements:

1. Show that any chunk, A, B, C, D or XOR can be deleted or corrupted and recovered
2. Show that any file size up to constraint size can be written and stored
3. Show that read and recovery results in an identical file to that written by full data compare or a data digest
4. Test basic number of RAID writes per second possible
5. Test basic number of RAID reads per second possible
6. Test immediate read-after-write RAID operations possible with or without data compare

System constraints:

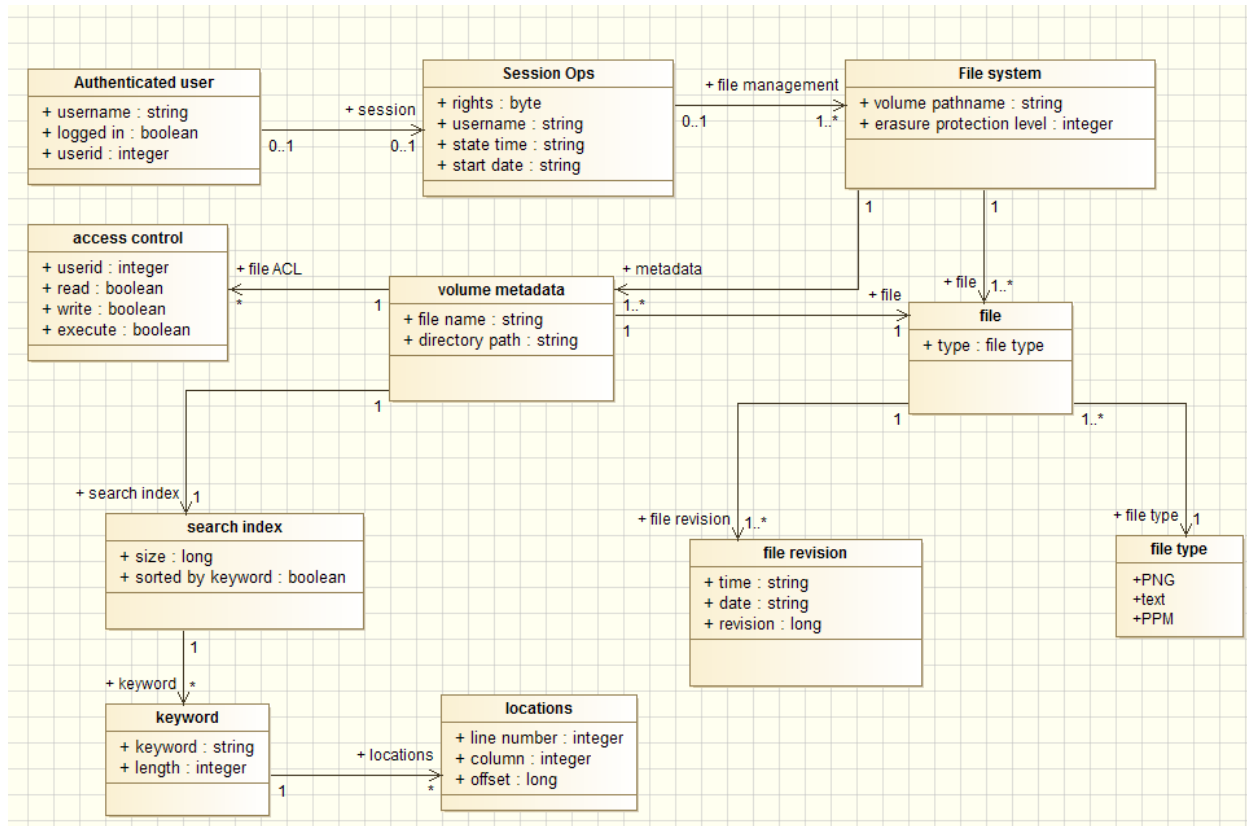
1. Storage of files up to 1 gigabyte in size (e.g. typical documents, image files, code, text, binary files)
2. RAID operations of 100 K read/write operations per second or better in the library such that a 1 gigabyte file could be written at a rate of 50 kilobytes/second or better, completing file transfer of a megabyte in 20 seconds or less

3. The current prototype can only handle files, not directories of multiple files, but this could be an extension pursued for a project or feature addition exercise which would require management of namespace metadata.
4. The current prototype does not implement any file access control or authentication, but this could be pursued as a feature addition exercise or project.

Figure 4 shows a basic class diagram for file RAID. This model can be found as a working example as a UML model for Modelio -

<http://mercury.pr.erau.edu/~siewerts/se310/design/Modelio/3.4-SD/>

Figure 4. More fully featured class diagram for Linux File RAID STaaS or Application



Note that Figure 4 shows class attributes and associations that go beyond what the current prototype has implemented such as:

1. File attributes including time and date, revision, and type
2. File system name space including directory and file path
3. Access control
4. Indexing for search
5. Session management with user authentication

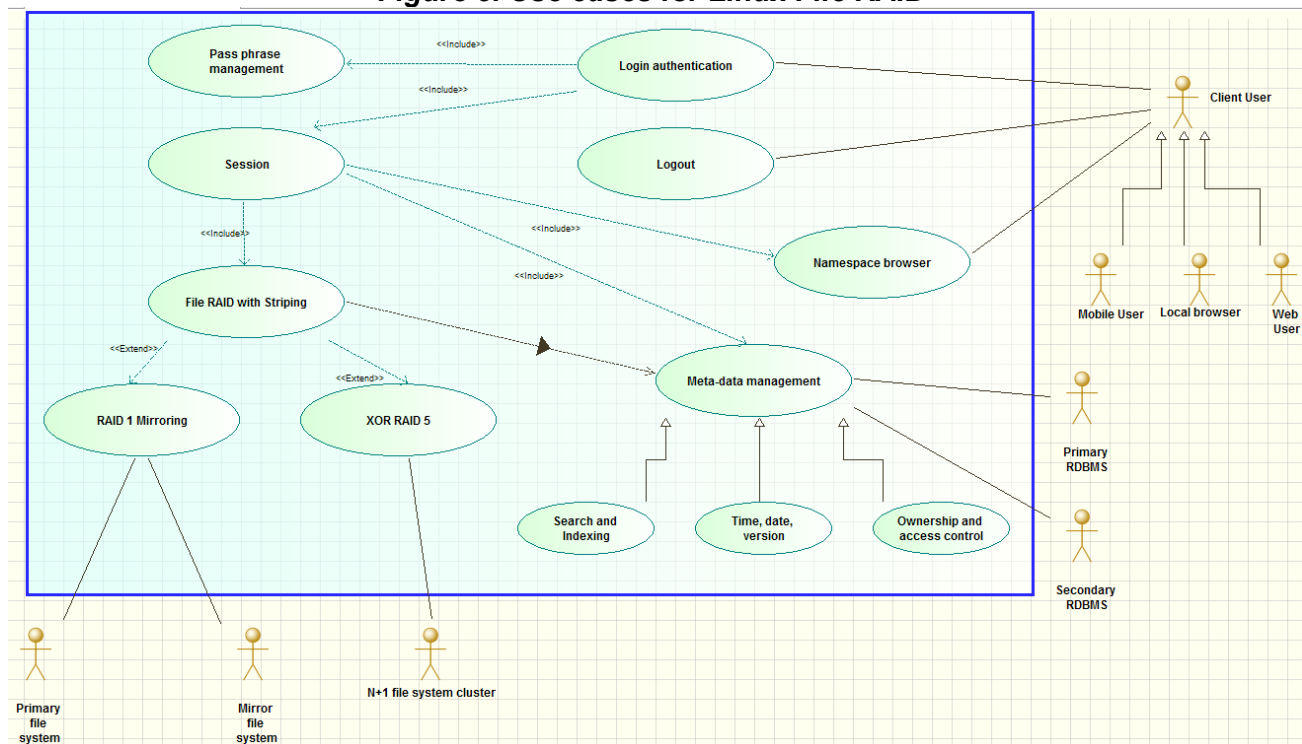
A significant, but useful feature that could be added is journaling, whereby all versions of the same file (same name and path) could be preserved at a chunk level (which therefore works for any file type including binary as well as text) where only modified chunks are re-written on update. This is a complex feature, but many file systems and RAID systems do provide this journaling feature by comparing chunks and writing all changed chunks with time and date metadata. One major

challenge of journaling are the potential read-modify-write operations (and associated workload) for this feature.

2.3 Use Cases

Two main use cases are anticipated for this software including STaaS where the software provides back-end storage for a front-end web interface, much like Dropbox or AWS S3, but with specific options for data protection and possibly journaling features. The second use case is a basic application that provides software RAID on a Linux system with a browser application assuming the Linux system has multiple file systems mounted on multiple storage devices to provide data protection. Figure 5 shows uses cases for both STaaS and browsing locally for RAID-10 and RAID-50.

Figure 5. Use cases for Linux File RAID



For STaaS, it is assumed that users will mostly want a basic archiving capability for files to be used for backup, file sharing, and file distribution. Common uses would be storing, sharing and distributing digital photos, design files, or other file types other than code. Code is most often best stored in a CMVC system, so while source code files could be stored, the system provides no specific advantage and is intended to mostly provide a bit bucket for files.

For a local Linux File RAID browser and data protection application, the user will want the file system to be as easy to use as current command line directory manipulation tools and graphical file explorers commonly available on Ubuntu. Basic access control (user, group and other permissions) and user authentication should be supported.

The key advantage of the Linux File RAID STaaS and browser are the built in speedup and data protection features, which may be sufficient to make the service and applications built for deployment using this software design advantageous compared to block level software RAID (easier to use on a directory basis rather than volume of disks in array). However, additional

features such as journaling, directory level authentication, web access to Linux File RAID and other feature extensions possible could increase value and advantages for users. These use cases could be broken into smaller more specific use cases and expanded based upon the intended application of STaaS or a local browser.

2.4 Use Case and Requirements Tracing

As studied in SE 310, use cases and requirements can be correlated to validate that all requirements support a use case and all use cases are supported by requirements. Furthermore, use cases can be prioritized (or requirements as is often done by the commercial industry if all requirements are not mission critical). For example:

	UC1	UC2	UC3	UC4	UC5	UC6	score
weight	3	3	1	1	2	1	
R1	X	X					6
R2					X		2
R3	X						3
R4		X	X				4
R5				X		X	2
R6		X			X		5

This should be completed as a validation of requirements – are the right requirements and use cases driving what is built so that we build the right thing. Whether the design has been built right (verification) can be determined by steps such as code generation from CASE (Computer Aided Software Engineering) tool code generation (e.g. Modelio class C++ generation, MySQL Workbench schema SQL generation, etc.) as well as prototyping, design walk-throughs, and code walk-throughs along with SQA test plans, scripts, and drivers outlined in section 5 for test-driven design with prototyping.

2.5 Operating Environment

The operating environment for Linux File RAID is a standard Linux platform such as Ubuntu, CentOS, or RHEL Linux running on a scalable server or workstation that is configured to have multiple storage devices and file systems. This could be as simple as two disk drives, each with a standard file system (e.g. ext4) for a RAID-1 configuration, but scaling up to an even number of storage devices and file systems for RAID-10, and scaling up to a number of equally sized RAID-5 sets that can be striped for RAID-50. For data protection each file system must be on an independent device, but for basic testing, the software can be run on one partitioned storage device with multiple file systems or even on one file system with multiple directories (both of which provide no protection, but allow for testing). No specialized hardware is required otherwise, which is a distinct advantage of file RAID compared to block RAID.

2.6 Design and Implementation Constraints

Developers may test on an unsafe system with one file system or fewer file systems than required for the RAID level, but all deployed systems should have a sufficient number of independent file systems for data protection so that a user is not misled into thinking their data is protected when it's not – the unsafe configurations simply allow for emulation testing at lower cost than a safe deployment. Metadata is best managed in an RDBMS that likewise should have dedicated storage, but again, this could also be hosted on common storage for testing.

2.7 User Documentation

Basic RAID configurations are documented by the Linux software RAID project [8]. No specific user documentation for this software has been developed yet for STaaS or a local Linux File RAID browser – this is left as an exercise for a student project along with feature extensions and development of STaaS web services or a browser application.

2.8 Assumptions and Dependencies

It is assumed that students using this software are familiar with basic RAID read, write and rebuild operations and theory.

The design and prototype software should build and run on most any Linux system, without need for root access, but can only be used as a safe deployment with a multi-file scalable system and is primarily intended for educational exploration of Linux File RAID rather than actual deployment.

For RDBMS metadata management the software should be run on a system with MySQL or other commonly available RDBMS that can be accessed locally by the software, ideally with dedicated storage independent of the file systems used to store the chunked file RAID data and parity.

3. External Interface Requirements

External interfaces are any interface to hardware, software outside the scope of the current design and/or users of the design or other systems out of scope.

3.1 User Interfaces

As provided, the software has no user interface. It is just an example and library of basic RAID operations that can be used to build STaaS or a file RAID browser for Linux.

A simple command line interface could be created using a shell such as this mini-shell application code (<http://mercury.pr.erau.edu/~siewerts/cs415/code/minishell/>), or with a GUI built with Qt creator or another similar 2D GUI development tool chain.

The interactive CLI (Command Line Interface) and/or browser GUI (Graphical User Interface) are left as an exercise for the student to explore for practice and/or a project.

3.2 Hardware Interfaces

The example requires a Linux system, typically x86 workstation similar to ERAU PRClab or the SE workstation with commonly found Unix file systems (e.g. ext4, the 4th extended file system for Linux). A network interface is assumed with ability to connect locally or remotely via SSH tunnel. The system is assumed to have an RDBMS like MySQL installed for metadata management.

3.3 Software Interfaces

This prototype requires an interface to Linux file systems and optionally to a Linux SQL compliant RDBMS such as MySQL. The current prototype has been tested on PRClab and the SE workstation at ERAU.

3.4 Communications Interfaces

For web based STaaS, the system must use SSH tunneling to PRClab as documented for other client tools (e.g. <http://mercury.pr.erau.edu/~siewerts/cs317/documents/Lectures/Lecture-MySQL-Workbench.pdf>). For local use it is assumed that the user has a shell interface such as MobaXterm and for a GUI browser that the shell can set a remote graphical display back for remote clients.

4. System Features

The current software is a simple library and example that students are expected to extend into a web service for STaaS or an application for a file RAID browser. As such, the template for detailed features and requirements that follows have been left in the original SWEBOK template format.

<This template illustrates organizing the functional requirements for the product by system features, the major services provided by the product. You may prefer to organize this section by use case, mode of operation, user class, object class, functional hierarchy, or combinations of these, whatever makes the most logical sense for your product.>

4.1 System Feature 1

<Don't really say "System Feature 1." State the feature name in just a few words.>

4.1.1 Description and Priority

<Provide a short description of the feature and indicate whether it is of High, Medium, or Low priority. You could also include specific priority component ratings, such as benefit, penalty, cost, and risk (each rated on a relative scale from a low of 1 to a high of 9).>

4.1.2 Stimulus/Response Sequences

<List the sequences of user actions and system responses that stimulate the behavior defined for this feature. These will correspond to the dialog elements associated with use cases.>

4.1.3 Functional Requirements

<Itemize the detailed functional requirements associated with this feature. These are the software capabilities that must be present in order for the user to carry out the services provided by the feature, or to execute the use case. Include how the product should respond to anticipated error conditions or invalid inputs. Requirements should be concise, complete, unambiguous, verifiable, and necessary. Use "TBD" as a placeholder to indicate when necessary information is not yet available.>

<Each requirement should be uniquely identified with a sequence number or a meaningful tag of some kind.>

REQ-1:

REQ-2:

4.2 System Feature 2 (and so on) ...

5. Test Driven Design Reference Prototypes

A goal for SE 420 and SE 310 is to use evolutionary Agile strategy and process with UML methods and test driven design. As such, the idea is to code in support of requirements analysis and to validate and verify the UML design use cases (cross referenced with requirements), CRC class models, domain model class diagrams and object interaction model message sequencing diagrams, activity diagrams, etc. These methods are covered in class, but this is a chance to practice them with simple exercises related to this reference design and implementation or as a starting point for a student project to build a more complete Linux File RAID STaaS, GUI browser, CLI, or library of re-useable RAID operations for one of these applications or systems.

5.1 Checkout and build

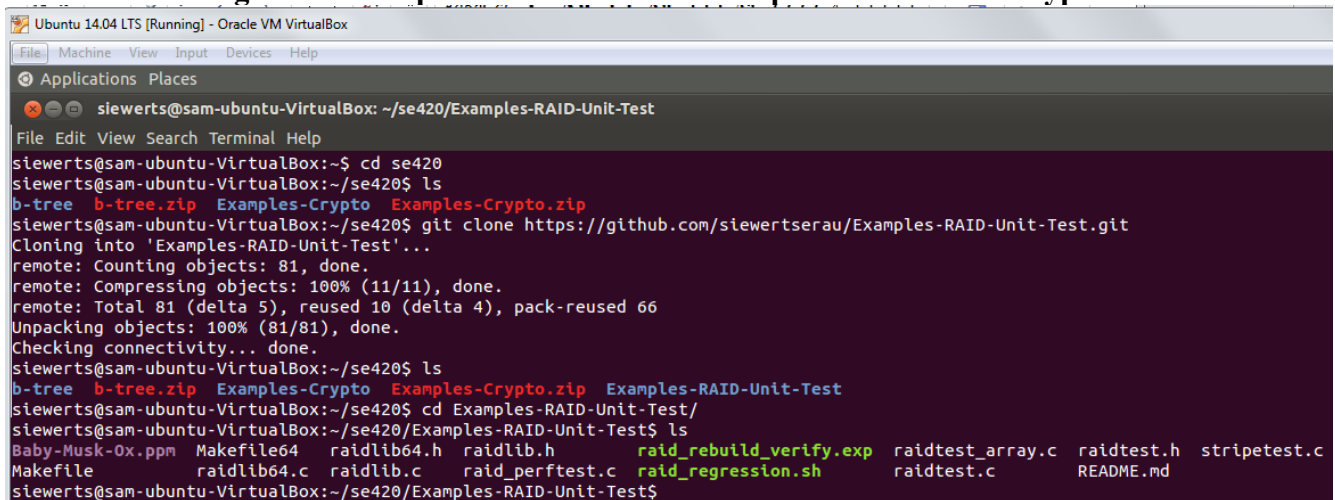
Note that this reference design included reference prototype code, which can be obtained from a non-managed source for convenience as noted in the text below, but the ideal place to obtain the latest reference code corresponding to this documentation is github

(<https://github.com/siewertserau/Examples-RAID-Unit-Test>). On a Linux system, such as the ERAU SE workstation, or your own VB-Linux system, clone the software as follows:

```
git clone https://github.com/siewertserau/Examples-RAID-Unit-Test.git
```

This should look like the clone in Figure 6.

Figure 6. Example Clone of Reference Implementation Prototype



```

siewerts@sam-ubuntu-VirtualBox: ~/se420/Examples-RAID-Unit-Test
File Edit View Search Terminal Help
siewerts@sam-ubuntu-VirtualBox:~$ cd se420
siewerts@sam-ubuntu-VirtualBox:~/se420$ ls
b-tree b-tree.zip Examples-Crypto Examples-Crypto.zip
siewerts@sam-ubuntu-VirtualBox:~/se420$ git clone https://github.com/siewertserau/Examples-RAID-Unit-Test.git
Cloning into 'Examples-RAID-Unit-Test'...
remote: Counting objects: 81, done.
remote: Compressing objects: 100% (11/11), done.
remote: Total 81 (delta 5), reused 10 (delta 4), pack-reused 66
Unpacking objects: 100% (81/81), done.
Checking connectivity... done.
siewerts@sam-ubuntu-VirtualBox:~/se420$ ls
b-tree b-tree.zip Examples-Crypto Examples-Crypto.zip Examples-RAID-Unit-Test
siewerts@sam-ubuntu-VirtualBox:~/se420$ cd Examples-RAID-Unit-Test/
siewerts@sam-ubuntu-VirtualBox:~/se420/Examples-RAID-Unit-Test$ ls
Baby-Musk-Ox.ppm Makefile64 raidlib64.h raidlib.h raid_rebuild_verify.exp raidtest_array.c raidtest.h stripetest.c
Makefile          raidlib64.c raidlib.c  raid_perftest.c raid_regression.sh raidtest.c  README.md
siewerts@sam-ubuntu-VirtualBox:~/se420/Examples-RAID-Unit-Test$

```

If the `raid_regression.sh` script is not “green” indicated it is executable, you can make it so with `chmod a+x raid_regression.sh` as sometimes github cloning with your local “[umask](#)” can wind up modifying file attribute bits such as execute privilege. See associated Linux manual pages for more information (e.g. [umask](#), [chmod](#)).

Now build the examples with “[make](#)” using `make`. If you are new to development and building software in Linux, you may find some of Dr. Siewert’s tutorials useful (<http://mercury.pr.erau.edu/~siewerts/se420/documents/Linux/>), especially [Makefile Basics for Linux](#). All code delivered for SQA should build with a simple top-level “make” and ideally should be warning free when “-Wall” is used. If that’s not true, that should be noted as a build issue. Normally warnings are not considered a build failure, but can indicate code problems that could negatively impact execution tests. The developer should therefore be asked to make code fixes to silence warnings and/or provide explanation and specific build warning options if some warnings can’t be fixed (suppressing warnings can be a dangerous practice, so first preference is fixing code and maintaining -Wall if at all possible).

Figure 7 below shows a make build of the reference code which at the point in time when this code was cloned, was warning free with “-Wall” as can be seen below. Checking warnings may also be supported by code walk-throughs to ensure that they have been properly addressed.

Figure 7. Linux File RAID Unit Test

```

siewerts@sam-ubuntu-VirtualBox: ~/se420/Examples-RAID-Unit-Test
File Edit View Search Terminal Help
siewerts@sam-ubuntu-VirtualBox:~/se420/Examples-RAID-Unit-Test$ make
cc -Wall -O0 -g -c raidlib.c
cc -Wall -O0 -g -o raidtest raidlib.o raidtest.o
cc -Wall -O0 -g -c raid_perftest.c
cc -Wall -O0 -g -o raid_perftest raidlib.o raid_perftest.o
cc -Wall -O0 -g -c stripetest.c
cc -Wall -O0 -g -o stripetest raidlib.o stripetest.o
cc -Wall -O0 -g -c raidtest_array.c
cc -Wall -O0 -g -o raidtest_array raidlib.o raidtest_array.o
make -f Makefile64 clean
make[1]: Entering directory `/home/siewerts/se420/Examples-RAID-Unit-Test'
rm -f *.o *.NEW *~
rm -f raidtest64 raid_perftest64
make[1]: Leaving directory `/home/siewerts/se420/Examples-RAID-Unit-Test'
make -f Makefile64
make[1]: Entering directory `/home/siewerts/se420/Examples-RAID-Unit-Test'
cc -O3 -msse3 -malign-double -g -DRAID64 -c raidlib64.c
cc -O3 -msse3 -malign-double -g -DRAID64 -c raidtest.c
cc -O3 -msse3 -malign-double -g -DRAID64 -o raidtest64 raidlib64.o raidtest.o
cc -O3 -msse3 -malign-double -g -DRAID64 -c raid_perftest.c
cc -O3 -msse3 -malign-double -g -DRAID64 -o raid_perftest64 raidlib64.o raid_perftest.o
make[1]: Leaving directory `/home/siewerts/se420/Examples-RAID-Unit-Test'
siewerts@sam-ubuntu-VirtualBox:~/se420/Examples-RAID-Unit-Test$ 

```

After a successful build, this should be noted (either automatically by a regression script or by the QA tester in a notebook or test plan). As can be seen by the executable scripts and test drivers in “green”, there are eight total tests at the time of the last update of this document. These tests can be described as follows in Table 1.

Table 1. Linux File RAID Unit, I&T and Regression tests

#	Name	Type	Description
1	raidtest	UNIT	Simple RAID operation test with visual verification
2	raidtest64	UNIT	64 bit version of RAID operation test with visual verification
3	raidtest_array	UNIT	Array version of RAID operation test with visual verification requested for better code readability by sustaining engineering as an implementation alternative
4	raid_perftest	UNIT	Read, write and rebuild operations per second 32 bit
5	raid_perftest64	UNIT	Read, write and rebuild operations per second 64 bit
6	stripetest	I&T	Integration test of RAID operations in raidlib with simple application to chunk a file and store it, then read it back for external data compare
7	raid_rebuild_verify.exp	REGRESS	Expect regression script for RAID rebuild operations
8	raid_regression.sh	REGRESS	General regression script for major RAID operations and features

5.2 Running Unit tests

The simple unit tests that provide a positive functional test for three different variants of the RAID library operations and include:

1. *raidtest* – original unsigned char byte array of file chunk data with no consideration taken for processor architecture word size, vector processing features, or efficiency, but based upon pointer arithmetic and dereferencing pointers.
2. *raidtest64* – modification to byte arrays to organize bytes into targets 64-bit words and designed for maximum speed-up on vector processing systems with multi-word XOR (e.g. 256 bit XOR), which in theory should be faster than the simpler default functions called in *raidtest*.
3. *raidtest_array* – modification to operations called by *raidtest* to use arrays and array dereferencing for readability, but perhaps at the cost of performance.

The main point of these three tests is that they all provide the same visual verification of correctness for XOR encoding, rebuild of the fourth block, and then testing with generated patterns using a modulo function for 1000 test cases (or more if provided as an argument) to pattern test the XOR encode and rebuild. This was used to simply ensure first that the XOR and rebuild operations were correct and also to make sure that the 64-bit variant and array variant agree with the original test. For simple visual verification a specific pattern is used in a repeating sequence as follows:

ASCII - #0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ#
 HEX - 3a 47 48 49 4a 4b 4c 4d 4e 4f 50 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24 25
 26 27 28 29 2a 2b 2c 2d 58 59 5a 5b 5c 5d 5e 5f 60 61 62 63 0 1 2 3 4 5 6 7 8 9 a b c d 3a

The second set of test cases uses a new set of patterns and offsets into an array of logical block addresses for chunk data. An assert is called for each test based on a full memory compare difference test, so any difference in the rebuilt buffer from original will cause the test to assert.

The next set of unit tests compares the simple byte array formulation for XOR with pointers and array dereference to determine if both can meet performance constraints. The question we want to answer is whether there is a significant performance difference between use of pointers compared to array dereferencing (seen as easier to read and understand) and it appears there is no appreciable difference for 1,000,000 operations of XOR encode and XOR to rebuild a chunk.

For example, a VB-Linux this produces:

```
siewerts@sam-ubuntu-VirtualBox:~/se420/Examples-RAID-Unit-Test$ ./raid_perftest 1000000
Will start 1000000 test iterations

RAID Operations Performance Test #1 - Pointers
Test Done in 4057491 microsecs for 1000000 iterations
246457.724737 RAID ops computed per second

RAID Operations Performance Test #2 - Arrays
Test Done in 4001900 microsecs for 1000000 iterations
249881.306379 RAID ops computed per second
siewerts@sam-ubuntu-VirtualBox:~/se420/Examples-RAID-Unit-Test$
```

Whereas there is no apparent difference between byte arrays and byte pointers, there is a huge performance difference for the 64-bit version on VB-Linux:

```
siewerts@sam-ubuntu-VirtualBox:~/se420/Examples-RAID-Unit-Test$ ./raid_perftest64 1000000
Will start 1000000 test iterations

RAID Operations Performance Test #1 - Pointers
Test Done in 629704 microsecs for 1000000 iterations
1588047.717658 RAID ops computed per second
siewerts@sam-ubuntu-VirtualBox:~/se420/Examples-RAID-Unit-Test$
```

Which is approximately 6x faster without optimization and without vector instruction code generation (significant).

However, if we recompile both with level 3 optimization and SSE vector instruction code generation, for the first `raid_perftest` UNIT test, we get:

```
siewerts@sam-ubuntu-VirtualBox:~/se420/Examples-RAID-Unit-Test$ ./raid_perftest 1000000
Will start 1000000 test iterations

RAID Operations Performance Test #1 - Pointers
Test Done in 165110 microsecs for 1000000 iterations
6056568.348374 RAID ops computed per second

RAID Operations Performance Test #2 - Arrays
Test Done in 133540 microsecs for 1000000 iterations
7488392.990864 RAID ops computed per second
siewerts@sam-ubuntu-VirtualBox:~/se420/Examples-RAID-Unit-Test$
```

Where arrays are actually 50% faster than pointers (good for readability and performance) and compared to 64-bit, arrays are still the fastest with optimization on:

```
siewerts@sam-ubuntu-VirtualBox:~/se420/Examples-RAID-Unit-Test$ ./raid_perftest64 1000000
Will start 1000000 test iterations

RAID Operations Performance Test #1 - Pointers
Test Done in 143994 microsecs for 1000000 iterations
6944733.808353 RAID ops computed per second
siewerts@sam-ubuntu-VirtualBox:~/se420/Examples-RAID-Unit-Test$
```

If this is a critical performance constraint, we'd want to test the different formulations on platforms that are of most interests such as the STaaS workstation we plan to use.

As a final white-box unit test, the McCabe metric is run on `radilib.c` to rank functions in terms of functional complexity, resulting in:

```
siewerts@sam-ubuntu-VirtualBox:~/se420/Examples-RAID-Unit-Test$ pmccabe -v raidlib.c
Modified McCabe Cyclomatic Complexity
| Traditional McCabe Cyclomatic Complexity
| | # Statements in function
| | | First line of function
| | | # lines in function
| | | filename(definition line number):function
| | | |
5 5 5 38 11 raidlib.c(38): xorLBA
5 5 5 54 16 raidlib.c(54): xorLBAArray
5 5 7 79 19 raidlib.c(79): rebuildLBA
5 5 7 103 21 raidlib.c(103): rebuildLBAArray
3 3 8 127 16 raidlib.c(127): checkEquivLBA
12 12 64 147 108 raidlib.c(147): stripeFile
16 16 95 259 153 raidlib.c(259): restoreFile
siewerts@sam-ubuntu-VirtualBox:~/se420/Examples-RAID-Unit-Test$
```

Which indicates that `stripeFile` and `restoreFile` are the most complex in terms of code structure, so they might be best to walk-through to see if there are simplifications (or single step through with a debugger). The point would be restructuring of the logic and/or algorithm for speed-up, readability, and maintainability (none are functional requirements, but performance is an important constraint or minimum requirement).

Another white-box test might examine the assembly code generated for key code blocks that are performance critical, although performance is well above the constraint (minimum requirement) or 50K operations per second.

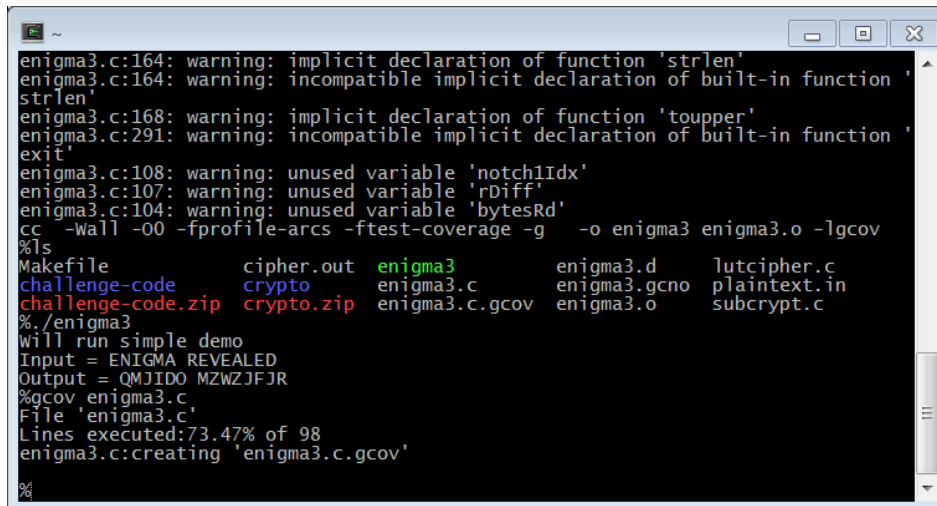
Development of additional unit test cases (corner cases, stress, soak, random pattern, complexity analysis, etc.) are left up to the student.

5.3 Path coverage

Path coverage can be turned on for any test by simply invoking in the Makefile as has been done in previous examples with the following flags noted in bold and “red”:

```
CFLAGS= -Wall -O0 -fprofile-arcs -ftest-coverage -g
```

Furthermore, when compiled, the **-lgcov** linking directive must be added to link the gcov library with the executable as well.



```

enigma3.c:164: warning: implicit declaration of function 'strlen'
enigma3.c:164: warning: incompatible implicit declaration of built-in function '
strlen'
enigma3.c:168: warning: implicit declaration of function 'toupper'
enigma3.c:291: warning: incompatible implicit declaration of built-in function '
exit'
enigma3.c:108: warning: unused variable 'notch1Idx'
enigma3.c:107: warning: unused variable 'rDiff'
enigma3.c:104: warning: unused variable 'bytesRd'
cc -Wall -O0 -fprofile-arcs -ftest-coverage -g -o enigma3 enigma3.o -lgcov
%ls
Makefile          cipher.out  enigma3      enigma3.d      lutcipher.c
challenge-code     crypto     enigma3.c    enigma3.gcov   enigma3.o      plaintext.in
challenge-code.zip crypto.zip  enigma3.c.gcov enigma3.o      subcrypt.c
%./enigma3
Will run simple demo
Input = ENIGMA REVEALED
Output = QMJIDO MZWZJFJR
%gcov enigma3.c
File 'enigma3.c'
Lines executed:73.47% of 98
enigma3.c:creating 'enigma3.c.gcov'

```

Note that the coverage data can also be converted into a browseable web page. This is done after building and running with gcov instrumentation using commands to create HTML from the statistics with:

```
[siewerts@localhost Examples-Crypto-Gcov]$ lcov -t 'Enigma report' -o
enigma3.info -c -d .
```

```
[siewerts@localhost Examples-Crypto-Gcov]$ genhtml -o result enigma3.info
```

Which will produce the browseable coverage similar to that generated for enigma3 in the above example - <http://mercury.pr.erau.edu/~siewerts/se420/code/Enigma-LCOV-results/>

The gcov and lcov analysis is left as an exercise for the student to complete based upon the example provided for `enigma3.c` code. Emphasis should be on `raidlib.c` for this code.

5.4 Profiling

Gnu profiling using `gprof` can be completed as follows by simply adding `-pg` to the compile options that are normally used, so:

```
%make
cc -O3 -Wall -pg -msse3 -malign-double -g -c raidtest.c
raidtest.c: In function 'main':
cc -O3 -Wall -pg -msse3 -malign-double -g -c raidlib.c
cc -O3 -Wall -pg -msse3 -malign-double -g -o raidtest raidtest.o
raidlib.o
%./raidtest
Will default to 1000 iterations
Architecture validation:
sizeof(unsigned long long)=8
RAID Operations Performance Test
Test Done in 453 microsecs for 1000 iterations
2207505.518764 RAID ops computed per second
%ls
Makefile      gmon.out      raidlib.h      raidlib64.c    raidtest      raidtest.o
Makefile64    raidlib.c     raidlib.o      raidlib64.h    raidtest.c    raidtest64
%gprof raidtest gmon.out > raidtest_analysis.txt
```

This profile shows that most of the time is spent in “main” for `raidtest` (which is not surprising as memory copies and memory comparison are very costly) followed by XOR encoding and then rebuild.

Flat profile:

Each sample counts as 0.01 seconds.

%	cumulative	self		self	total	
time	seconds	seconds	calls	ns/call	ns/call	name
82.13	1.54	1.54				main
15.47	1.83	0.29	2000001	145.38	145.38	xorLBA
2.67	1.88	0.05	2000001	25.07	25.07	rebuildLBA

This should be retested with profiling for specific critical performance constraints for STaaS or Linux File RAID browser functionality. A key aspect is to make sure that enough iterations are executed to achieve a ramped-up stable performance metric.

5.5 I&T scripts

One I&T test was created to provide an example of module integration called “stripetest”. This integrated test interfaces and links `raidlib.c` functions for XOR along with functions for chunking a file and coalescing it back into a single file. With more complexity, it might make sense to place the file operations into a `fileraidlib.c` module and to keep the block oriented operations in `raidlib.c` so that `raidlib.c` could support either block or file RAID. Either way, this test integrates file operations and block encode (write), decode (read) and rebuild operations into a single integrated

application that approximates Linux File RAID as an application. To run the stripetest, simply provide an input file to encode into 5 chunks including 4 data and 1 XOR as follows:

```
siewerts@sam-ubuntu-VirtualBox:~/se420/Examples-RAID-Unit-Test$
./stripetest
Usage: stripetest <inputfile> <outputfile>
siewerts@sam-ubuntu-VirtualBox:~/se420/Examples-RAID-Unit-Test$
./stripetest Baby-Musk-Ox.ppm Baby-Musk-Ox.ppm.restored
read full stripe
...
read full stripe
hit end of file
CHUNKED with bytesWritten=756954
FINISHED with bytesRestored=756954
siewerts@sam-ubuntu-VirtualBox:~/se420/Examples-RAID-Unit-Test$
```

Upon completion, to ensure that the I&T driver in fact successfully created a restored version of the original file, a binary difference should be checked. First, we can simply difference the two files as follows:

```
siewerts@sam-ubuntu-VirtualBox:~/se420/Examples-RAID-Unit-Test$ diff Baby-Musk-Ox.ppm Baby-Musk-Ox.ppm.restored
siewerts@sam-ubuntu-VirtualBox:~/se420/Examples-RAID-Unit-Test$ cmp Baby-Musk-Ox.ppm Baby-Musk-Ox.ppm.restored
siewerts@sam-ubuntu-VirtualBox:~/se420/Examples-RAID-Unit-Test$
```

Both are simply silent if there are no differences as is true above, which can be unsatisfying, so to verify more rigorously, we might want to review and make sure there are no differences visually. This can be done by converting the binary into hexadecimal files (which will be much larger) and reviewing visually and performing a diff on the ASCII hexadecimal files as follows:

```
siewerts@sam-ubuntu-VirtualBox:~/se420/Examples-RAID-Unit-Test$ od -t x1 -w32 Baby-Musk-Ox.ppm > Baby-Musk-Ox.ppm.hex
siewerts@sam-ubuntu-VirtualBox:~/se420/Examples-RAID-Unit-Test$ od -t x1 -w32 Baby-Musk-Ox.ppm.restored > Baby-Musk-Ox.ppm.restored.hex
siewerts@sam-ubuntu-VirtualBox:~/se420/Examples-RAID-Unit-Test$ od -t x1 -w32 --read-bytes=512 Baby-Musk-Ox.ppm
00000000 50 36 0a 23 20 43 52 45 41 54 4f 52 3a 20 47 49 4d 50 20 50 4e 4d 20 46 69 6c 74 65 72 20 56 65
00000040 72 73 69 6f 6e 20 31 2e 31 0a 35 38 30 20 34 33 35 0a 32 35 35 0a 27 22 1f 26 21 1d 27 1f 1c 24
00000100 1d 15 26 1c 12 2b 22 13 35 29 1b 3b 2f 1f 47 3f 2c 40 39 26 38 31 21 31 2e 1f 31 2d 21 2f 2f 25
00000140 30 30 28 2e 2f 27 26 23 1a 20 1e 12 1a 17 0e 18 15 0c 1b 18 11 1f 1c 15 1f 1e 1a 1e 1d 19 24 23
00000200 1f 24 23 1f 23 24 1f 24 25 20 26 27 21 29 2a 24 2c 2d 25 2e 2f 27 30 2f 2a 2f 2e 29 34 33 2e 3b
00000240 3a 35 38 37 32 30 2f 2a 31 30 2b 38 37 32 3a 39 34 3b 3a 35 3c 3b 36 3d 3c 37 3d 3c 37 3d 3c 37
00000300 3c 3b 36 3b 3b 33 41 3e 37 41 3e 35 42 3f 36 44 41 38 47 45 39 48 46 3a 48 44 39 46 42 37 3e 3a
00000340 2e 41 3d 31 46 40 34 49 43 37 47 41 35 3f 39 2d 36 30 22 2f 29 1d 2c 25 1d 2d 26 20 33 2a 25 3a
00000400 31 2c 3b 32 2b 39 30 29 3e 34 2b 44 3a 31 36 2e 23 37 31 25 3a 34 26 38 36 27 36 34 25 30 31 21
00000440 2d 30 1d 2c 2f 1e 27 24 1b 36 33 2a 40 3a 2c 42 3b 28 56 4b 35 77 69 4f 86 76 5c 83 71 59 7f 6a
00000500 55 77 62 4f 71 5e 4f 72 5f 50 6a 59 49 5a 4b 38 4f 40 2d 4b 3f 29 53 42 32 4f 3e 2e 4c 3c 2d 4c
00000540 3f 2f 50 42 37 4f 43 37 48 3e 34 42 38 2e 42 39 30 4e 45 3c 50 46 3c 47 3d 33 4d 41 35 5f 53 47
00000600 68 5a 4d 62 55 44 5d 52 3c 60 56 3b 52 46 2e 4f 43 2b 54 48 32 5d 51 3b 6d 60 4f 68 5b 4a 5e 52
00000640 42 4e 45 34 46 3d 2e 44 3e 2e 3a 34 26 26 23 14 1c 1a 0b 1d 1e 10 1c 1c 12 20 21 19 24 25 1d 26
00000700 27 1f 27 27 1f 26 26 1e 27 24 1d 26 23 1c 30 2d 26 36 33 2c 3b 36 30 39 34 2e 33 2c 26 2e 27 21
00000740 2f 28 22 33 2c 26 34 30 27 37 33 2a 38 34 2b 35 31 28 32 2e 23 34 30 25 3a 36 2b 3f 3b 30 3d 39
00010000
siewerts@sam-ubuntu-VirtualBox:~/se420/Examples-RAID-Unit-Test$ od -t x1 -w32 --read-bytes=512 Baby-Musk-Ox.ppm
00000000 50 36 0a 23 20 43 52 45 41 54 4f 52 3a 20 47 49 4d 50 20 50 4e 4d 20 46 69 6c 74 65 72 20 56 65
00000040 72 73 69 6f 6e 20 31 2e 31 0a 35 38 30 20 34 33 35 0a 32 35 35 0a 27 22 1f 26 21 1d 27 1f 1c 24
00000100 1d 15 26 1c 12 2b 22 13 35 29 1b 3b 2f 1f 47 3f 2c 40 39 26 38 31 21 31 2e 1f 31 2d 21 2f 2f 25
00000140 30 30 28 2e 2f 27 26 23 1a 20 1e 12 1a 17 0e 18 15 0c 1b 18 11 1f 1c 15 1f 1e 1a 1e 1d 19 24 23
00000200 1f 24 23 1f 23 24 1f 24 25 20 26 27 21 29 2a 24 2c 2d 25 2e 2f 27 30 2f 2a 2f 2e 29 34 33 2e 3b
00000240 3a 35 38 37 32 30 2f 2a 31 30 2b 38 37 32 3a 39 34 3b 3a 35 3c 3b 36 3d 3c 37 3d 3c 37 3d 3c 37
00000300 3c 3b 36 3b 3b 33 41 3e 37 41 3e 35 42 3f 36 44 41 38 47 45 39 48 46 3a 48 44 39 46 42 37 3e 3a
00000340 2e 41 3d 31 46 40 34 49 43 37 47 41 35 3f 39 2d 36 30 22 2f 29 1d 2c 25 1d 2d 26 20 33 2a 25 3a
00000400 31 2c 3b 32 2b 39 30 29 3e 34 2b 44 3a 31 36 2e 23 37 31 25 3a 34 26 38 36 27 36 34 25 30 31 21
00000440 2d 30 1d 2c 2f 1e 27 24 1b 36 33 2a 40 3a 2c 42 3b 28 56 4b 35 77 69 4f 86 76 5c 83 71 59 7f 6a
00000500 55 77 62 4f 71 5e 4f 72 5f 50 6a 59 49 5a 4b 38 4f 40 2d 4b 3f 29 53 42 32 4f 3e 2e 4c 3c 2d 4c
00000540 3f 2f 50 42 37 4f 43 37 48 3e 34 42 38 2e 42 39 30 4e 45 3c 50 46 3c 47 3d 33 4d 41 35 5f 53 47
00000600 68 5a 4d 62 55 44 5d 52 3c 60 56 3b 52 46 2e 4f 43 2b 54 48 32 5d 51 3b 6d 60 4f 68 5b 4a 5e 52
00000640 42 4e 45 34 46 3d 2e 44 3e 2e 3a 34 26 26 23 14 1c 1a 0b 1d 1e 10 1c 1c 12 20 21 19 24 25 1d 26
00000700 27 1f 27 27 1f 26 26 1e 27 24 1d 26 23 1c 30 2d 26 36 33 2c 3b 36 30 39 34 2e 33 2c 26 2e 27 21
00000740 2f 28 22 33 2c 26 34 30 27 37 33 2a 38 34 2b 35 31 28 32 2e 23 34 30 25 3a 36 2b 3f 3b 30 3d 39
00010000
siewerts@sam-ubuntu-VirtualBox:~/se420/Examples-RAID-Unit-Test$
```

For a more visual review, try “vbindiff” which can be installed with “sudo apt-get install vbindiff”, which produces a review that shows that there are no differences and allows you to hit <RETURN> to skip to any binary difference sections (if there are any).

```
siewerts@sam-ubuntu-VirtualBox: ~/se420/Examples-RAID-Unit-Test
File Edit View Search Terminal Help
Baby-Musk-Ox.ppm
0000 0000: 50 36 0A 23 20 43 52 45 41 54 4F 52 3A 20 47 49 P6.# CRE ATOR: GI
0000 0010: 4D 50 20 50 4E 4D 20 46 69 6C 74 65 72 20 56 65 MP PNM F ilter Ve
0000 0020: 72 73 69 6F 6E 20 31 2E 31 0A 35 38 30 20 34 33 rsion 1. 1.580 43
0000 0030: 35 0A 32 35 35 0A 27 22 1F 26 21 1D 27 1F 1C 24 5.255.'" .&!. '..$
0000 0040: 1D 15 26 1C 12 2B 22 13 35 29 1B 3B 2F 1F 47 3F ..&..+ ". 5).;/.G?
0000 0050: 2C 40 39 26 38 31 21 31 2E 1F 31 2D 21 2F 2F 25 ,@9&81!1 ..1-!//%
0000 0060: 30 30 28 2E 2F 27 26 23 1A 20 1E 12 1A 17 0E 18 00(./'&# . ....
0000 0070: 15 0C 1B 18 11 1F 1C 15 1F 1E 1A 1E 1D 19 24 23 .....$#
0000 0080: 1F 24 23 1F 23 24 1F 24 25 20 26 27 21 29 2A 24 .$.#$. $ % &'!)*$
0000 0090: 2C 2D 25 2E 2F 27 30 2F 2A 2F 2E 29 34 33 2E 3B ,-%./'0/ */.)43.;
0000 00A0: 3A 35 38 37 32 30 2F 2A 31 30 2B 38 37 32 3A 39 :58720/* 10+872:9
0000 00B0: 34 3B 3A 35 3C 3B 36 3D 3C 37 3D 3C 37 3D 3C 37 4;:5<;6= <7=<7=<7
0000 00C0: 3C 3B 36 3B 3B 33 41 3E 37 41 3E 35 42 3F 36 44 <;6;;3A> 7A>5B?6D
0000 00D0: 41 38 47 45 39 48 46 3A 48 44 39 46 42 37 3E 3A A8GE9HF: HD9FB7>:
0000 00E0: 2E 41 3D 31 46 40 34 49 43 37 47 41 35 3F 39 2D .A=1F@4I C7GA5?9-
0000 00F0: 36 30 22 2F 29 1D 2C 25 1D 2D 26 20 33 2A 25 3A 60"/).,% .-& 3*%:
0000 0100: 31 2C 3B 32 2B 39 30 29 3E 34 2B 44 3A 31 36 2E 1,;2+90) >4+D:16.

Baby-Musk-Ox.ppm.restored
0000 0000: 50 36 0A 23 20 43 52 45 41 54 4F 52 3A 20 47 49 P6.# CRE ATOR: GI
0000 0010: 4D 50 20 50 4E 4D 20 46 69 6C 74 65 72 20 56 65 MP PNM F ilter Ve
0000 0020: 72 73 69 6F 6E 20 31 2E 31 0A 35 38 30 20 34 33 rsion 1. 1.580 43
0000 0030: 35 0A 32 35 35 0A 27 22 1F 26 21 1D 27 1F 1C 24 5.255.'" .&!. '..$
0000 0040: 1D 15 26 1C 12 2B 22 13 35 29 1B 3B 2F 1F 47 3F ..&..+ ". 5).;/.G?
0000 0050: 2C 40 39 26 38 31 21 31 2E 1F 31 2D 21 2F 2F 25 ,@9&81!1 ..1-!//%
0000 0060: 30 30 28 2E 2F 27 26 23 1A 20 1E 12 1A 17 0E 18 00(./'&# . ....
0000 0070: 15 0C 1B 18 11 1F 1C 15 1F 1E 1A 1E 1D 19 24 23 .....$#
0000 0080: 1F 24 23 1F 23 24 1F 24 25 20 26 27 21 29 2A 24 .$.#$. $ % &'!)*$
0000 0090: 2C 2D 25 2E 2F 27 30 2F 2A 2F 2E 29 34 33 2E 3B ,-%./'0/ */.)43.;
0000 00A0: 3A 35 38 37 32 30 2F 2A 31 30 2B 38 37 32 3A 39 :58720/* 10+872:9
0000 00B0: 34 3B 3A 35 3C 3B 36 3D 3C 37 3D 3C 37 3D 3C 37 4;:5<;6= <7=<7=<7
0000 00C0: 3C 3B 36 3B 3B 33 41 3E 37 41 3E 35 42 3F 36 44 <;6;;3A> 7A>5B?6D
0000 00D0: 41 38 47 45 39 48 46 3A 48 44 39 46 42 37 3E 3A A8GE9HF: HD9FB7>:
0000 00E0: 2E 41 3D 31 46 40 34 49 43 37 47 41 35 3F 39 2D .A=1F@4I C7GA5?9-
0000 00F0: 36 30 22 2F 29 1D 2C 25 1D 2D 26 20 33 2A 25 3A 60"/).,% .-& 3*%:
0000 0100: 31 2C 3B 32 2B 39 30 29 3E 34 2B 44 3A 31 36 2E 1,;2+90) >4+D:16.

Arrow keys move F find RET next difference ESC quit T move top
C ASCII/EBCDIC E edit file G goto position Q quit B move bottom
```

5.6 System tests and scripts

A system test has not been created and is left as a student exercise, but for example a CLI shell could be added to stripetest so that the CLI would allow for file browsing and listing, file copy, rename, etc. A simple shell such as “minishell” could be adapted as a front end for striping RAID-5

and user interaction (e.g. <http://mercury.pr.erau.edu/~siewerts/cs415/code/minishell/> from CS 415). The idea is to create an end-to-end test from a user (external to our design scope) to the file system (also external to our design scope) and closer to the actual intended use when deployed, but perhaps not yet identical (this might not be realized in all system tests, but should be in at least one and certainly in acceptance testing).

5.7 Regression tests and scripts

The idea of regression is to curate previously created unit, I&T and system tests into one scripted test of all the best tests in terms of finding bugs (defects) in the code so a developer can run this test after making a fix or adding a feature to ensure that they have not broken something else in the process accidentally as a side effect (a regression). The `raid_regression.sh` shell script is a regression that could also be run as a nightly test (another use for regression) to make sure that code pushed back to the github “root” (main code line) is not broken. This is the case for this example which can be run as follows:

```
siewerts@sam-ubuntu-VirtualBox:~/se420/Examples-RAID-Unit-Test$ ./raid_regression.sh
RAID UNIT REGRESSSION TEST
TEST SET 1: checkout and build test
TEST SET 2: simple RAID encode, erase and rebuild test
TEST SET 3: REBUILD DIFF CHECK for Chunk4 rebuild
TEST SET 4: RAID performance test
siewerts@sam-ubuntu-VirtualBox:~/se420/Examples-RAID-Unit-Test$
```

Which doesn't appear to have done much, but we find that this test builds a log file (common with regression for a large batch of smaller github updates, clean and build, test driver executions, scripts, etc. all in one run). The log file shows:

```
siewerts@sam-ubuntu-VirtualBox:~/se420/Examples-RAID-Unit-Test$ cat
testresults.log
RAID UNIT REGRESSSION TEST
```

TEST SET 1: checkout and build test

Successful build

```
rm -f *.o *.NEW *~ Chunk*.bin StripeChunk*.bin
rm -f raidtest raid_perftest stripetest raidtest_array raid64_tests
make -f Makefile64 clean
make[1]: Entering directory `/home/siewerts/se420/Examples-RAID-Unit-Test'
rm -f *.o *.NEW *~
rm -f raidtest64 raid_perftest64
make[1]: Leaving directory `/home/siewerts/se420/Examples-RAID-Unit-Test'
Already up-to-date.
cc -O3 -msse3 -malign-double -g -c raidlib.c
cc -O3 -msse3 -malign-double -g -c raidtest.c
cc -O3 -msse3 -malign-double -g -o raidtest raidlib.o raidtest.o
cc -O3 -msse3 -malign-double -g -c raid_perftest.c
cc -O3 -msse3 -malign-double -g -o raid_perftest raidlib.o
raid_perftest.o
cc -O3 -msse3 -malign-double -g -c stripetest.c
cc -O3 -msse3 -malign-double -g -o stripetest raidlib.o stripetest.o
cc -O3 -msse3 -malign-double -g -c raidtest_array.c
cc -O3 -msse3 -malign-double -g -o raidtest_array raidlib.o
raidtest_array.o
```

```

make -f Makefile64 clean
make[1]: Entering directory `/home/siewerts/se420/Examples-RAID-Unit-Test'
rm -f *.o *.NEW *~
rm -f raidtest64 raid_perftest64
make[1]: Leaving directory `/home/siewerts/se420/Examples-RAID-Unit-Test'
make -f Makefile64
make[1]: Entering directory `/home/siewerts/se420/Examples-RAID-Unit-Test'
cc -O3 -msse3 -malign-double -g -DRAID64 -c raidlib64.c
cc -O3 -msse3 -malign-double -g -DRAID64 -c raidtest.c
cc -O3 -msse3 -malign-double -g -DRAID64 -o raidtest64 raidlib64.o
raidtest.o
cc -O3 -msse3 -malign-double -g -DRAID64 -c raid_perftest.c
cc -O3 -msse3 -malign-double -g -DRAID64 -o raid_perftest64 raidlib64.o
raid_perftest.o
make[1]: Leaving directory `/home/siewerts/se420/Examples-RAID-Unit-Test'

```

TEST SET 2: simple RAID encode, erase and rebuild test

Will start 1000 test iterations

Architecture validation:

sizeof(unsigned long long)=8

Successful RAID
write, read, erase
and rebuild, etc.

TEST CASE 1:

```

3a 47 48 49 4a 4b 4c 4d 4e 4f 50 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20
21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 58 59 5a 5b 5c 5d 5e 5f 60 61 62
63 0 1 2 3 4 5 6 7 8 9 a b c d 3a 3a 47 48 49 4a 4b 4c 4d 4e 4f 50 14 15
16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d
58 59 5a 5b 5c 5d 5e 5f 60 61 62 63 0 1 2 3 4 5 6 7 8 9 a b c d 3a 3a 47
48 49 4a 4b 4c 4d 4e 4f 50 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 22
23 24 25 26 27 28 29 2a 2b 2c 2d 58 59 5a 5b 5c 5d 5e 5f 60 61 62 63 0 1
2 3 4 5 6 7 8 9 a b c d 3a 3a 47 48 49 4a 4b 4c 4d 4e 4f 50 14 15 16 17
18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 58 59
5a 5b 5c 5d 5e 5f 60 61 62 63 0 1 2 3 4 5 6 7 8 9 a b c d 3a 3a 47 48 49
4a 4b 4c 4d 4e 4f 50 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24
25 26 27 28 29 2a 2b 2c 2d 58 59 5a 5b 5c 5d 5e 5f 60 61 62 63 0 1 2 3 4
5 6 7 8 9 a b c d 3a 3a 47 48 49 4a 4b 4c 4d 4e 4f 50 14 15 16 17 18 19
1a 1b 1c 1d 1e 1f 20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 58 59 5a 5b
5c 5d 5e 5f 60 61 62 63 0 1 2 3 4 5 6 7 8 9 a b c d 3a 3a 47 48 49 4a 4b
4c 4d 4e 4f 50 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24 25 26
27 28 29 2a 2b 2c 2d 58 59 5a 5b 5c 5d 5e 5f 60 61 62 63 0 1 2 3 4 5 6 7
8 9 a b c d 3a 3a 47 48 49 4a 4b 4c 4d 4e 4f 50 14 15 16 17 18 19 1a 1b
1c 1d 1e 1f 20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 58 59 5a 5b 5c 5d
5e 5f 60 61 62 63 0 1 2 3 4 5 6 7 8 9 a b c d 3a

```

```

3a 47 48 49 4a 4b 4c 4d 4e 4f 50 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20
21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 58 59 5a 5b 5c 5d 5e 5f 60 61 62
63 0 1 2 3 4 5 6 7 8 9 a b c d 3a 3a 47 48 49 4a 4b 4c 4d 4e 4f 50 14 15
16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d
58 59 5a 5b 5c 5d 5e 5f 60 61 62 63 0 1 2 3 4 5 6 7 8 9 a b c d 3a 3a 47
48 49 4a 4b 4c 4d 4e 4f 50 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 22
23 24 25 26 27 28 29 2a 2b 2c 2d 58 59 5a 5b 5c 5d 5e 5f 60 61 62 63 0 1
2 3 4 5 6 7 8 9 a b c d 3a 3a 47 48 49 4a 4b 4c 4d 4e 4f 50 14 15 16 17
18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 58 59

```

5a 5b 5c 5d 5e 5f 60 61 62 63 0 1 2 3 4 5 6 7 8 9 a b c d 3a 3a 47 48 49
4a 4b 4c 4d 4e 4f 50 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24
25 26 27 28 29 2a 2b 2c 2d 58 59 5a 5b 5c 5d 5e 5f 60 61 62 63 0 1 2 3 4
5 6 7 8 9 a b c d 3a 3a 47 48 49 4a 4b 4c 4d 4e 4f 50 14 15 16 17 18 19
1a 1b 1c 1d 1e 1f 20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 58 59 5a 5b
5c 5d 5e 5f 60 61 62 63 0 1 2 3 4 5 6 7 8 9 a b c d 3a 3a 47 48 49 4a 4b
4c 4d 4e 4f 50 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24 25 26
27 28 29 2a 2b 2c 2d 58 59 5a 5b 5c 5d 5e 5f 60 61 62 63 0 1 2 3 4 5 6 7
8 9 a b c d 3a 3a 47 48 49 4a 4b 4c 4d 4e 4f 50 14 15 16 17 18 19 1a 1b
1c 1d 1e 1f 20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 58 59 5a 5b 5c 5d
5e 5f 60 61 62 63 0 1 2 3 4 5 6 7 8 9 a b c d 3a

TEST CASE 2 (randomized sectors and rebuild):

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51
52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75
76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117
118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135
136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153
154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171
172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189
190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207
208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225
226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243
244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261
262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279
280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297
298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315
316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333
334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351
352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369
370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387
388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405
406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423
424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441
442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459
460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477
478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495
496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513
514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531
532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549
550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567
568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585
586 587 588 589 590 591 592 593 594 595 596 597 598 599 600 601 602 603
604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621
622 623 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639
640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657
658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675
676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693
694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711
712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729
730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747
748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765

```

766 767 768 769 770 771 772 773 774 775 776 777 778 779 780 781 782 783
784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801
802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819
820 821 822 823 824 825 826 827 828 829 830 831 832 833 834 835 836 837
838 839 840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855
856 857 858 859 860 861 862 863 864 865 866 867 868 869 870 871 872 873
874 875 876 877 878 879 880 881 882 883 884 885 886 887 888 889 890 891
892 893 894 895 896 897 898 899 900 901 902 903 904 905 906 907 908 909
910 911 912 913 914 915 916 917 918 919 920 921 922 923 924 925 926 927
928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945
946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963
964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981
982 983 984 985 986 987 988 989 990 991 992 993 994 995 996 997 998 999
FINISHED

```

TEST SET 3: REBUILD DIFF CHECK for Chunk4 rebuild

```

Binary files Chunk1.bin and Chunk4_Rebuilt.bin differ
Binary files Chunk2.bin and Chunk4_Rebuilt.bin differ
Binary files Chunk3.bin and Chunk4_Rebuilt.bin differ

```

Successful re-
build indicated by
zero differences
(silent) for
Chunk4 and
Chunk4 rebuilt

TEST SET 4: RAID performance test

Will start 1000 test iterations

RAID Operations Performance Test #1 - Pointers

Test Done in 261 microsecs for 1000 iterations

3831417.624521 RAID ops computed per second

RAID Operations Performance Test #2 - Arrays

Test Done in 118 microsecs for 1000 iterations

8474576.271186 RAID ops computed per second

Successfully
exceeded
performance
constraint of 50K
operations per
second

siewerts@sam-ubuntu-VirtualBox:~/se420/Examples-RAID-Unit-Test\$

5.8 Acceptance tests

Both regression tests and acceptance testing can be further automated by creating a script which automatically compares actual output to expected output for PASS and with logging for FAIL with a third option of TIMEOUT that also logs information if possible. The three options of PASS, FAIL, TIMEOUT are generally the only accepted options for acceptance testing and automated regression (the log files should be reviewed to determine what happened for FAIL and TIMEOUT cases). The point is that we want a real simple and clear report on tests run, their outcome and basic statistics at the end to chart bug rates and to keep a history of nightly regression testing results.

This functionality can be found in a number of regression automation tools including “expect” and “dejagnu” for Linux, which can be installed if needed with:

```

sudo apt-get install expect
sudo apt-get install expect-dev
sudo apt-get install dejagnu

```

An example “expect” script produces this type of PASS, FAIL and TIMEOUT outcome output. On failure or timeout, one option is to run the specific test again with logging on so that the normal all

PASS scenario is now slowed down by logging when tests are running as expected. Either way, when “raid_rebuild_verify.exp” is run as follows:

```
siewerts@sam-ubuntu-VirtualBox:~/se420/Examples-RAID-Unit-Test$ ./raidtest | ./raid_rebuild_verify.exp
Architecture validation:      [PASS]
Recovered data verification:  [PASS]
Randomized data verification: [PASS]
```

We see three passes as expected from running raidtest and piping the output into “expect” which matches output specification provided in “raid_rebuild_verify.exp” and produces report summary outcomes for each test case. To see what happens when we don’t produce any data for expect, we run it again with nothing piped in and see timeouts for each case:

```
siewerts@sam-ubuntu-VirtualBox:~/se420/Examples-RAID-Unit-Test$ ./raid_rebuild_verify.exp
Architecture validation:      [TIMEOUT]
Recovered data verification:  [TIMEOUT]
Randomized data verification: [TIMEOUT]
siewerts@sam-ubuntu-VirtualBox:~/se420/Examples-RAID-Unit-Test$ █
```

6. Other Nonfunctional Requirements

6.1 Performance Requirements

Examples include read and write operations per second, read-after-write, read-modify-write, rebuild time, and size constraints. A very basic performance test for the RAID operations library example can be found in example code (<http://mercury.pr.erau.edu/~siewerts/se420/code/Examples-RAID/raidops.c>).

<If there are performance requirements for the product under various circumstances, state them here and explain their rationale, to help the developers understand the intent and make suitable design choices. Specify the timing relationships for real time systems. Make such requirements as specific as possible. You may need to state performance requirements for individual functional requirements or features.>

6.2 Safety Requirements

To avoid data loss, the RAID mapping must be properly configured to independent file systems. The software could check this as a safety feature.

<Specify those requirements that are concerned with possible loss, damage, or harm that could result from the use of the product. Define any safeguards or actions that must be taken, as well as actions that must be prevented. Refer to any external policies or regulations that state safety issues that affect the product’s design or use. Define any safety certifications that must be satisfied.>

6.3 Security Requirements

Numerous security features could be added including data privacy, login authentication, access control, metadata security for the RDBMS and protection of metadata from SQL injection attack. The security features are left for the student as a project or as practice exercises for specific features.

<Specify any requirements regarding security or privacy issues surrounding use of the product or protection of the data used or created by the product. Define any user identity authentication requirements. Refer to any external policies or regulations containing security issues that affect the product. Define any security or privacy certifications that must be satisfied.>

6.4 Software Quality Attributes

The software quality assurance tasks to be completed by the student as practice exercises or an extended project include:

1. Unit testing
 - a. Unit test cases for each library function (extended what is provided in the example - <http://mercury.pr.erau.edu/~siewerts/se420/code/Examples-RAID-Unit-Test/>)
 - b. Addition of code coverage to black box unit testing
 - c. Addition of code module profiling to black box performance unit tests
 - d. White-box unit testing as the student sees fit to refactor and improve software source level quality including readability, modularity, maintainability, etc. and other non-functional requirements
2. I&T testing
 - a. Integration of RAID operations modules into a simple application with command line
 - b. Combination of RAID levels including RAID-0, RAID-1, RAID-5
 - c. Fault injection including loss of a mirrored or XOR parity projected chunk of a file
3. System testing
 - a. Integration with a GUI for browsing
 - b. Integration with web services for remote browsing
4. Acceptance testing
 - a. Basic use case scenarios for a bit bucket and associated tests
5. Regression testing and scripting
 - a. Curation of a set of tests from unit, I&T, and system testing that can be re-run manually or via scripted tests to regression test software after bug fixes and feature additions.

<Specify any additional quality characteristics for the product that will be important to either the customers or the developers. Some to consider are: adaptability, availability, correctness, flexibility, interoperability, maintainability, portability, reliability, reusability, robustness, testability, and usability. Write these to be specific, quantitative, and verifiable when possible. At the least, clarify the relative preferences for various attributes, such as ease of use over ease of learning.>

6.5 Business Rules

The basic idea is to improve design and implementation quality based upon this starter example to achieve one of the following final products if selected for a project, including:

1. RAID library that can be re-used, implemented in C or C++ with simple demonstrations and unit tests including all operations required for a STaaS web service or a stand-alone browser
2. Stand-alone file RAID browser with extended features for data protection, data privacy, and general file security.
3. STaaS bit bucket service with data protection and data privacy features.

<List any operating principles about the product, such as which individuals or roles can perform which functions under specific circumstances. These are not functional requirements in themselves, but they may imply certain functional requirements to enforce the rules.>

7. Other Requirements

Rather than focus on complexity in terms of number of features, students are asked to focus on quality of both analysis and design (SA/SD and/or OOA/OOD) as well as code quality in C or C++ (structured program or OOP implementation). For a project, students should adopt a coding standard, for example: [[Ganssle Firmware Style](#), [EC++](#), [Google Style Guide](#), [Stroustrup on Style](#), [ROS Style Guide](#), [Apple Kernel Style Guide](#), [UM Style Guide](#)]. A coding standard is not pre-selected and enforced, but rather one should be adopted and basic styles followed and referenced.

<Define any other requirements not covered elsewhere in the SRS. This might include database requirements, internationalization requirements, legal requirements, reuse objectives for the project, and so on. Add any new sections that are pertinent to the project.>

Appendix A: Glossary

RAID – Redundant Array of Independent (or Inexpensive) Disks

File – collection of bytes with a name and attributes such as last time and date of modification, ownership, access rights, etc.

CLI – Command Line Interface

GUI – Graphical User Interface

STaaS – Storage as a Service

<Define all the terms necessary to properly interpret the SRS, including acronyms and abbreviations. You may wish to build a separate glossary that spans multiple projects or the entire organization, and just include terms specific to a single project in each SRS.>

Appendix B: Analysis Models

Please note example UML models provided

(<http://mercury.pr.erau.edu/~siewerts/se310/design/Modelio/3.4-SD/>). Extend and improve for SE 310 and use updated analysis and design to validate and verify at a unit (library), application shell or GUI (I&T) or system (STaaS) level.

<Optionally, include any pertinent analysis models, such as data flow diagrams, class diagrams, state-transition diagrams, or entity-relationship diagrams.>

Appendix C: To Be Determined List

<Collect a numbered list of the TBD (to be determined) references that remain in the SRS so they can be tracked to closure.>