

File system Implementation

Mukund Manikarnike
120859725
Computer Science, MCS

Advanced Operating Systems - Class Project
CSE 536
Spring 2015

Contents

1.	Introduction	5
2.	Requirements Specification	6
2.1	Implementation Requirements	6
2.1.1	File system	6
2.1.2	Cell Storage System.....	7
2.1.3	Journal Storage Manager.....	7
2.1.4	Error Injection	7
3.	Detailed Design	8
3.1	Assumptions	8
3.2	Overview.....	8
3.3	Data Structures	10
3.3.1	Shared Memory.....	10
3.3.2	File Structure memory.....	11
3.4	Operations.....	12
3.4.1	Initialization.....	12
3.4.2	Read.....	12
3.4.3	Write	13
3.4.4	Allocate.....	13
3.4.5	De-allocate	13
4.	Implementation	14
4.1	Overview.....	14
4.2	Pthreads	15
4.3	Shared Memory IPC and Semaphores	15
4.4	Locks	16
4.5	Requirements Mapping	16
5.	Testing	17
5.1	Overview.....	17
5.2	Test Results	17
5.2.1	ALL_OR_NOTHING_TEST.....	17
5.2.2	GEN_RD_WR_TEST_1	19
5.2.3	GEN_RD_WR_TEST_2	21
5.2.4	MULTI_THRD_TEST_1	23
5.2.5	ERR_PRN_TEST_1	25
6.	Summary.....	28

6.1	Key Features	28
6.2	Drawbacks.....	28
6.3	Learning experiences	29
6.4	Future work	29
	References	30

Tables and Figures

Figure 2-1 File System Schematic (Extract from [COMP_SYS_DESIGN])	6
Figure 3-1 CSS and JSM Inter-process communication Schematic	9
Figure 3-2 Shared Memory Structure	11
Figure 3-3 File Structure Memory	11

1. Introduction

The aim of the project is to implement the functionalities of a file system in the application layer only using C programming. The focus of the project is to simulate functionalities of a file system in environments with and without multithreading, introduce errors in the system and test how the file system performs.

This document contains a description of each of the following phases of the project

1. Requirements Specification
2. Design
3. Implementation
4. Testing

2. Requirements Specification

2.1 Implementation Requirements

2.1.1 File system

- [REQ_1] The file system shall implement a cell storage system and a journal storage manager as shown in **Figure 2-1** the detailed functionalities of which will be explained in further sections of the document.

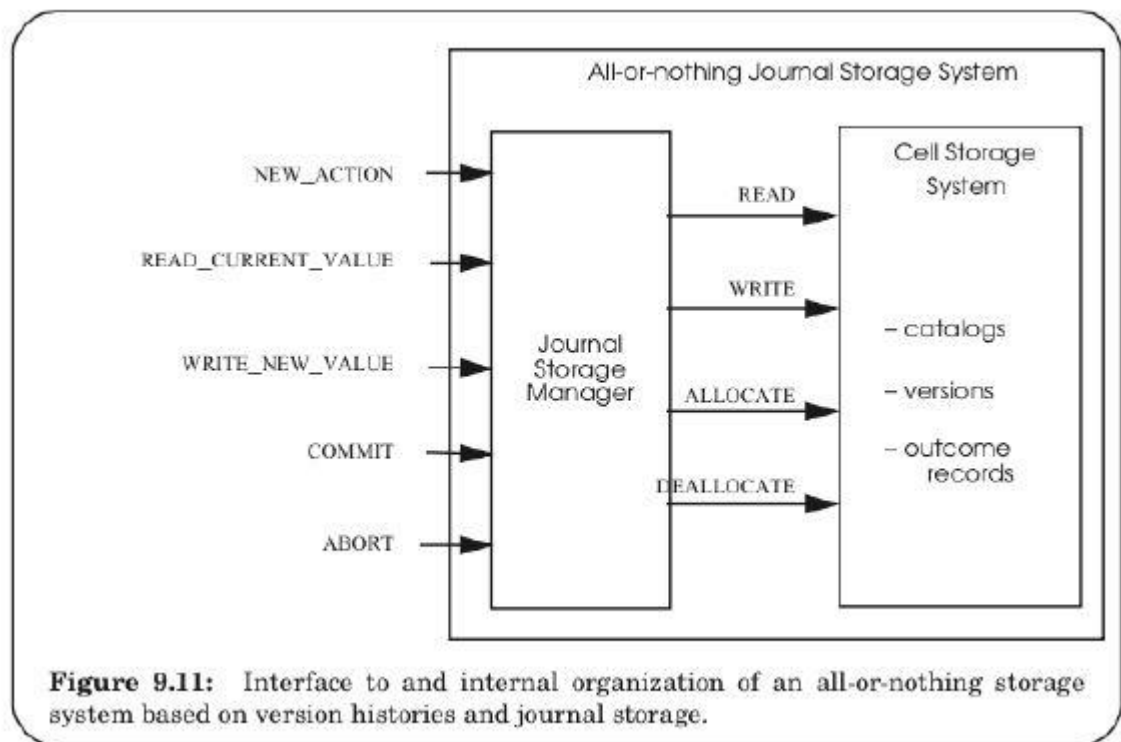


Figure 2-1 File System Schematic (Extract from [COMP_SYS_DESIGN])

- [REQ_2] The file system shall provide functionalities of NEW_ACTION, READ_CURRENT_VALUE, WRITE_NEW_VALUE, COMMIT and ABORT, the exact definitions of which will be taken up at the design phase.

- [REQ_3]** The file system shall handle the operations mentioned in **[REQ_2]** in an error free environment.
- [REQ_4]** The file system shall handle the operations mentioned in **[REQ_2]** in an error free environment with multi-threading.
- [REQ_5]** The file system shall handle the operations mentioned in **[REQ_2]** in an error prone environment without multi-threading.
- [REQ_6]** The file system shall handle the operations mentioned in **[REQ_2]** in an error-prone environment with multi-threading.

2.1.2 Cell Storage System

- [REQ_7]** The cell storage system shall provide an abstraction to an application like a journal storage system in writing data to the files.
- [REQ_8]** The cell storage system shall be implemented as a single file with multiple sectors.
- [REQ_9]** The cell storage system shall implement the READ, WRITE, ALLOCATE and DEALLOCATE procedures for files.

2.1.3 Journal Storage Manager

- [REQ_10]** The Journal storage manager shall be built as an application which uses the underlying functionalities provided by the Cell Storage System as described in **2.1.2**
- [REQ_11]** The journal storage manager shall provide all functionalities as mentioned in **[REQ_2]**

2.1.4 Error Injection

- [REQ_12]** Error injection to simulate error-prone conditions shall be provided by writing programs with certain faults, the exact details of which shall be explained in the design phase.

3. Detailed Design

This section will describe the overall design of the architecture chosen for the implementation of the file system and also details of the exact structures chosen in the design of the cell storage system and journal storage manager.

3.1 Assumptions

This section provides a brief overview of the assumptions in the design

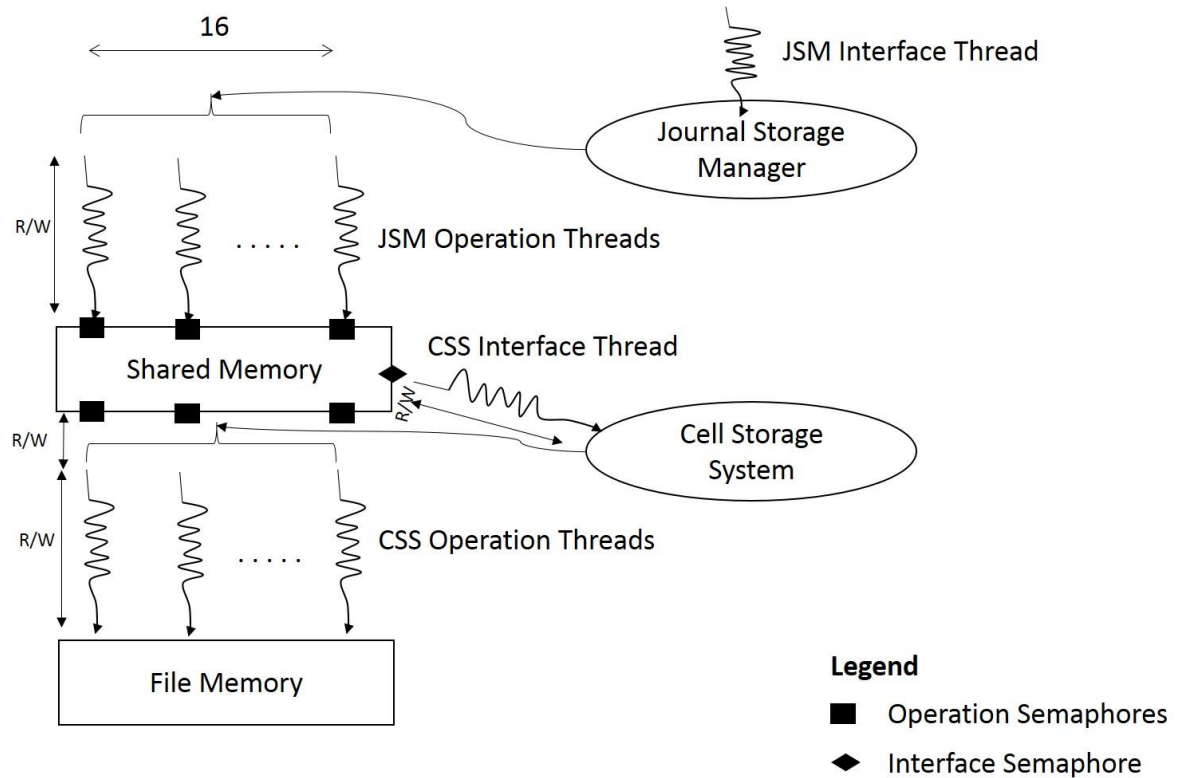
1. The number of files and the file size supported by the file system can be limited for purposes of this project
2. The number of concurrent operations can be limited to a certain maximum number.

3.2 Overview

The cell storage system acts as the back-end to the file system which stores a certain number of files of a certain size, the maximum limit on both of which are discussed in latter parts of the design.

The journal storage manager acts as the front-end interfacing with the user. The journal storage manager interacts with the cell storage system to extract/write data from/into the cell storage system structure.

As depicted in **Figure 3-1** the cell storage system and the journal storage manager run as 2 separate processes communicating with each other through a shared memory location. The shared memory allocation is performed using Linux system programming functions made available in C.

Architecture Overview**Figure 3-1** CSS and JSM Inter-process communication Schematic

The features supported by the Cell storage system are

1. Read, Write, Allocate and De-allocate operations for files
2. A maximum of 16 concurrent operations on files
3. A maximum storage of 1024 files
4. A maximum file size of 512 bytes
5. A maximum file name length of 27 characters

Some of the key concepts that the design employs are

1. The Cell Storage System (CSS) and the Journal Storage manager (JSM) are implemented as two separate processes.
 - a. The JSM acts as an interface to the user to support, Read, Write, Commit and Abort operations.
 - b. The CSS internally maintains the file data in a file memory and provides atomic APIs to perform these operations to the JSM.
2. The CSS and JSM communicate with each other using a shared memory which sits in between the CSS and the JSM. In order to synchronize the communication

mechanism between the two, they use semaphores to ensure that both processes aren't modifying the same shared memory location at the same time.

3. In order to support multi-threading, the CSS and JSM run up to a maximum of 16 concurrent threads through which desired file operations can be performed. The reason for 16 concurrent operations being the limit will be discussed in the next section.
 - a. Since multiple threads have access to shared memory, one semaphore per operation has been employed. Hence, there are 16 operation semaphores.
 - b. In order to have a one to one synchronization mapping between the JSM threads and the CSS threads, an interface semaphore is employed by waiting and signalling on which ensures that the JSM thread is synchronized with the CSS thread.
 - c. The interface thread on the JSM spawns new threads for each operation and provides an external user interface through command-line to the user. The interface thread on the CSS spawns new threads for each operation and serves the purpose of one to one thread synchronization mapping with JSM threads along with the use of the interface semaphore.
4. In order to ensure thread synchronization between threads of a CSS to avoid writes to file memory when another write is in progress, locks are used. When a write or a de-allocation is being performed, a lock is acquired and hence, no other write or de-allocation operation can be performed on the file at the same time.
 - a. Read operations, at this time, however will go through at this time. However, there is no guarantee of correct data being given out because it is not the responsibility of the file system to ensure correctness of read data when a write is performed. An application that uses these operations should ensure correctness by performing reads at specific intervals to detect changes in data and update the same accordingly. This is in accordance with the end-to-end argument which talks about keeping the lowest layer simple.

The following sections describe the data structure formats used and operations supported to make the above mentioned features available.

3.3 Data Structures

This section describes the format for structures created by the Cell storage system. The exact byte-structure indicated in this section is a result of what is stated in **3.1**. The reasons for the byte-structure chosen for fields that aren't explained by the above mentioned section are self-explanatory.

3.3.1 Shared Memory

The shared memory is partitioned as shown in **Figure 3-2**. This memory region is accessible by both the cell storage system as well as the journal storage manager.

Operation Type	Operation Number	Operation Bitmap	File - 1 Size	File - 1 Name	File - 1 Data	...	File - 16 Size	File - 16 Name	File - 16 Data
1 Byte	1 Byte	2 Bytes	2 Bytes	28 Bytes	512 Bytes		2 Bytes	28 Bytes	512 Bytes

Figure 3-2 Shared Memory Structure

The fields mentioned in the shared memory structure are defined as shown below

1. Operation Type

- This is used by the Journal storage manager to indicate the kind of operation to be done, i.e. Read/Write/Allocate/De-allocate.

2. Operation Number

- In order to support communication between the two components in the system during concurrent operations, an operation number is provided which will indicate which file structure in the shared memory to access.

3. Operation Bitmap

- This is used to indicate the status of the operation for each concurrent operation in progress.

4. File Structure

- Since 16 concurrent operations are supported, 16 File structures are supported which contain the size at the start of the structure followed by the file name and then the file data.

3.3.2 File Structure memory

The file structure memory is partitioned as shown in **Figure 3-3**. This memory region is accessible only to the Cell Storage system.

File - 1 Name	File - 2 Name	...	File - N Name	File Use Bitmap Address	File - 1 Size	File - 1 Data	...	File - N Size	File - N Name
28 Bytes	28 Bytes		28 Bytes	N/8 Bytes	2 Bytes	512 Bytes		2 Bytes	512 Bytes

Figure 3-3 File Structure Memory

3.4 Operations

All signaling between the Journal Storage manager and the cell storage system are performed through the use of semaphores, the details of which are discussed in **Implementation**

3.4.1 Initialization

The cell storage system on initialization allocates

1. The shared memory location for communication between the Journal storage manager
2. The file structure memory to store file data and metadata

The cell storage system needs to run in the background for the Journal storage manager to interact with it.

All other operations are done as follows

1. The Journal storage manager writes the operation type into the shared memory
2. The Cell storage system reads the operation type in an eternal while loop and performs the requested operation on the file structure memory that it internally maintains.
3. The Cell storage system then returns the result to the Journal storage manager through the shared memory in the case where a return to the Journal storage manager is required.

3.4.2 Read

The read operation is performed as follows

1. The cell storage system, extracts the file name from shared memory
2. It traverses the file structure memory to obtain the file data for the file mentioned.
3. It extracts the data from the file structure memory and copies the data to the shared memory
4. It then indicates to the Journal storage manager using semaphore signaling method.
5. The journal storage manager on requesting a read would wait on the semaphore that the Cell storage system signals. On detecting a success, the Journal storage manager has access to the data.

3.4.3 Write

The first 2 steps of a write operation are same as that in the read operation. The write operation is then performed as follows

1. The cell storage system then extracts the data from the shared memory and writes it into the corresponding entry for the file in the file structure memory.

3.4.4 Allocate

The allocate operation is performed as follows

1. The cell storage system finds the first free entry in the file inode in the file structure memory.
2. It then writes the file name in the file structure memory into the file inode. It also sets the usage bit in the File Use bitmap field to indicate that the file structure is being used.

3.4.5 De-allocate

The de-allocate operation is similar to the allocate operation in finding the inode entry. It resets the file usage bitmap to ensure that the link is lost and hence behaves like a de-allocation.

4. Implementation

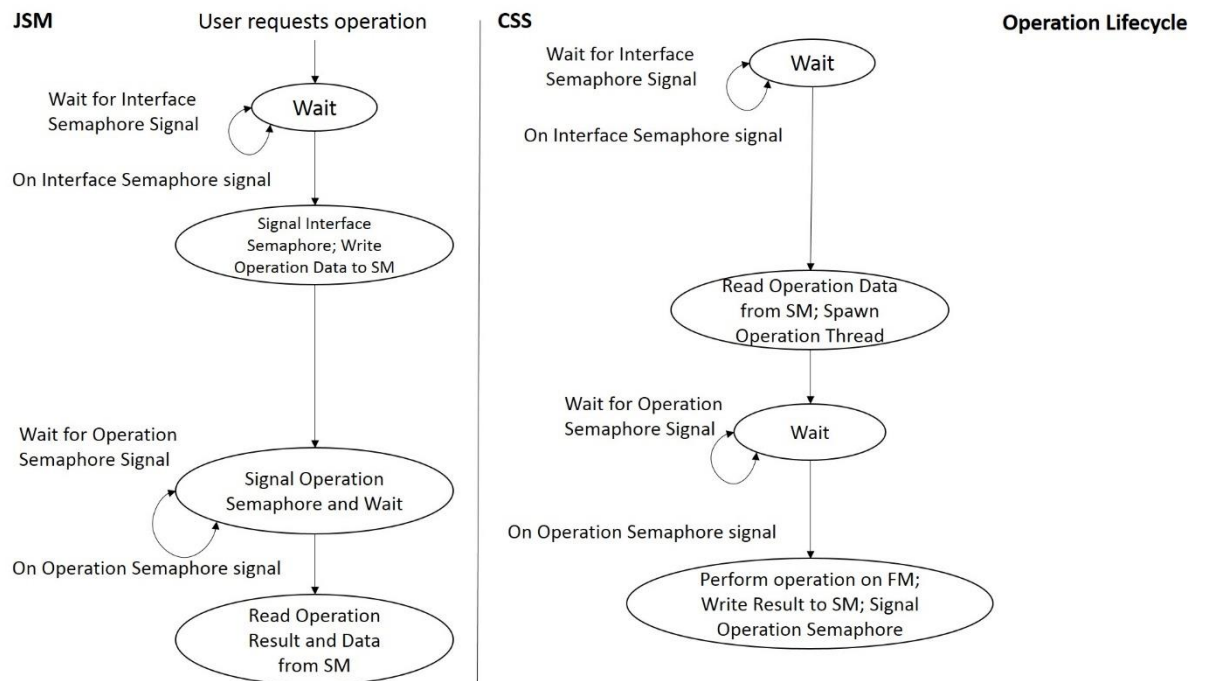
4.1 Overview

This section takes one through the several components used in the implementation of the JSM and CSS. Some of the components used in implementation are

1. Pthreads
2. Shared memory IPC
3. Semaphores
4. Locks

Lifecycle of an operation

This section depicts the lifecycle of an operation in the system in the form of a picture as shown below.



One of the things that might seem like a problem with the design is that the JSM and the CSS are both waiting on the same semaphore and might be eternally in a wait state. But,

this is not so. This case is handled by not waiting on the Interface semaphore if it's the first operation on start-up. Once the JSM gets past that step, the whole wait-signal lifecycle works smoothly like an FSM as depicted in the picture shown above.

The following sections describe the implementation method used in each of the components

4.2 Pthreads

The JSM and the CSS together can support 16 concurrent operations as mentioned in the design. These concurrent operations are supported using Pthreads. The basic procedure followed in the usage of Pthreads is described as follows

1. When the JSM receives a request from a user to perform an operation, it schedules a thread.
2. This thread will initiate the request on the CSS through the use of Shared memory.
3. The CSS on getting a request through the shared memory will then initiate another thread for each operation request that it receives.

4.3 Shared Memory IPC and Semaphores

Shared memory IPC and Semaphores are achieved using linux system calls. The requirement of semaphores is for the following reasons

1. To synchronize reads and writes from and to the shared memory since this memory is accessible to both the processes.

The project uses 2 sets of semaphores, 1 set containing 1 semaphore to initiate a request between the JSM and the CSS and another set which contains 16 semaphores to support concurrent operations. The reason for the usage of 2 sets of semaphores is

1. In order to support concurrent operations, both the processes need to run continuously and receive inputs from the user. But, they also need to communicate between each other.
2. In order to achieve this, the 2 processes halt user operations only for a short period in what is called an operation initialization phase. The 1st semaphore set is used in the operation initialization phase. Once the operation is initialized, the CSS and JSM are independent of each other. The read/write threads of the CSS and the JSM communicate with each other using the 2nd semaphore set on initialization.

4.4 Locks

Locks are used to facilitate locking the file memory when writes are being performed. This is to ensure that different operation threads which can be initiated concurrently are synchronized.

4.5 Requirements Mapping

The implementation has ensured that all requirements mentioned in **Requirements Specification** have been ensured. Following are a few details of the mapping

1. The Journal Storage manager and the Cell storage system being written as 2 separate processes ensures the abstraction required between the 2.
2. The operations mentioned as inputs to each of these systems have been implemented as they are
3. Multi-threading support and error scenario support have been provided. Demonstration of the same will need different builds using the makefile provided. A detailed description can be found in the Readme file.

5. Testing

5.1 Overview

This section gives a brief description of the testing strategy followed and the results for each of the tests carried out. Please refer to the readme file for exact commands to run each test.

1. The CSS and JSM are run in 2 separate terminals.
2. The CSS has to be run first. It will output a CSS ID which needs to be taken note of.
3. The JSM will read the CSS ID and allow the user to perform operations on the CSS through the shared memory.
4. The CSS and JSM run in an eternal while loop and will have to be stopped using an interrupt signal from the keyboard if needed.

5.2 Test Results

Following are some of the test cases that have been executed, the test results and their screenshots.

5.2.1 ALL_OR_NOTHING_TEST

Pre-requisite

1. Should have created a file with a certain name
2. Should have written and committed data to the file

Steps	Expected Result	Implementation Mapping
<ol style="list-style-type: none">1. Read information at a certain location from the disc2. Write some information to that position in disc3. Abort the write request (i.e. do not commit)4. Read from the write	Information read at Step 4 should be same information read at Step 1	<ol style="list-style-type: none">1. Read data from the file created2. Perform a write operation to the file3. Perform an abort operation on the write operation number

Steps	Expected Result	Implementation Mapping
location		4. Read data from the file

```

powershell - Shortcut
vagrant@vagrant-ubuntu-trusty-32:/vagrant/Courses/CSE536/ClassProject-Filesystem$ ./css.out
Start Semaphore, key = 262153 initialized.
Status Semaphore No 0, key = 262154 initialized.
Status Semaphore No 1, key = 262154 initialized.
Status Semaphore No 2, key = 262154 initialized.
Status Semaphore No 3, key = 262154 initialized.
Status Semaphore No 4, key = 262154 initialized.
Status Semaphore No 5, key = 262154 initialized.
Status Semaphore No 6, key = 262154 initialized.
Status Semaphore No 7, key = 262154 initialized.
Status Semaphore No 8, key = 262154 initialized.
Status Semaphore No 9, key = 262154 initialized.
Status Semaphore No 10, key = 262154 initialized.
Status Semaphore No 11, key = 262154 initialized.
Status Semaphore No 12, key = 262154 initialized.
Status Semaphore No 13, key = 262154 initialized.
Status Semaphore No 14, key = 262154 initialized.
Status Semaphore No 15, key = 262154 initialized.
vagrant@vagrant-ubuntu-trusty-32:/vagrant/Courses/CSE536/ClassProject-Filesystem$ ./css.out
Start Semaphore, key = 32770 initialized.
Status Semaphore No 0, key = 32771 initialized.
Status Semaphore No 1, key = 32771 initialized.
Status Semaphore No 2, key = 32771 initialized.
Status Semaphore No 3, key = 32771 initialized.
vagrant@vagrant-ubuntu-trusty-32:/vagrant/Courses/CSE536/ClassProject-Filesystem$ ./css.out
Start Semaphore, key = 65539 initialized.
Status Semaphore No 0, key = 65540 initialized.
Status Semaphore No 1, key = 65540 initialized.
Status Semaphore No 2, key = 65540 initialized.
Status Semaphore No 3, key = 65540 initialized.
Status Semaphore No 4, key = 65540 initialized.
Status Semaphore No 5, key = 65540 initialized.
Status Semaphore No 6, key = 65540 initialized.
Status Semaphore No 7, key = 65540 initialized.
Status Semaphore No 8, key = 65540 initialized.
Status Semaphore No 9, key = 65540 initialized.
Status Semaphore No 10, key = 65540 initialized.
Status Semaphore No 11, key = 65540 initialized.
Status Semaphore No 12, key = 65540 initialized.
Status Semaphore No 13, key = 65540 initialized.
Status Semaphore No 14, key = 65540 initialized.
Status Semaphore No 15, key = 65540 initialized.
Cell storage system ID is: 65538
Please keep note of this ID. The Journal storage manager
will need this as an input to access the cell storage
system
INFO: Allocate Attempted - Operation Number: 0
INFO: Read Attempted - Operation Number: 0

```

Outcome: PASS

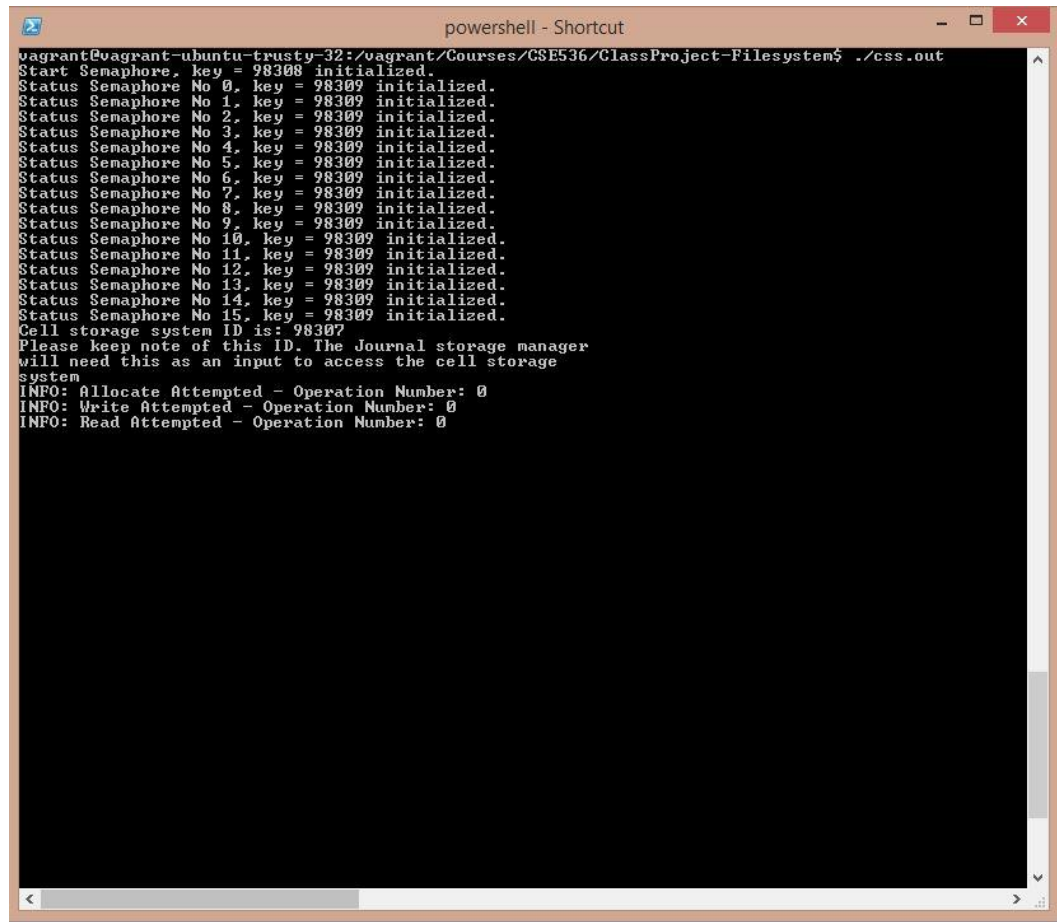
5.2.2 GEN_RD_WR_TEST_1

Pre-requisite

1. Should have created a file with a certain name
2. Should have written and committed data to the file

Steps	Expected Result	Implementation Mapping
1. Read information at a certain location from the disc 2. Write some information to that position in disc 3. Commit the write request 4. Read from the write	Information read at Step 4 should be same information written at Step 2	1. Read data from the file created 2. Perform a write operation to the file 3. Perform a commit operation on the write operation 4. Read data from the file

Steps	Expected Result	Implementation Mapping
location		



```
powershell - Shortcut
vagrant@vagrant-ubuntu-trusty-32:/vagrant/Courses/CSE536/ClassProject-Filesystem$ ./css.out
Start Semaphore, key = 98308 initialized.
Status Semaphore No 0, key = 98309 initialized.
Status Semaphore No 1, key = 98309 initialized.
Status Semaphore No 2, key = 98309 initialized.
Status Semaphore No 3, key = 98309 initialized.
Status Semaphore No 4, key = 98309 initialized.
Status Semaphore No 5, key = 98309 initialized.
Status Semaphore No 6, key = 98309 initialized.
Status Semaphore No 7, key = 98309 initialized.
Status Semaphore No 8, key = 98309 initialized.
Status Semaphore No 9, key = 98309 initialized.
Status Semaphore No 10, key = 98309 initialized.
Status Semaphore No 11, key = 98309 initialized.
Status Semaphore No 12, key = 98309 initialized.
Status Semaphore No 13, key = 98309 initialized.
Status Semaphore No 14, key = 98309 initialized.
Status Semaphore No 15, key = 98309 initialized.
Cell storage system ID is: 98307
Please keep note of this ID. The Journal storage manager
will need this as an input to access the cell storage
system
INFO: Allocate Attempted - Operation Number: 0
INFO: Write Attempted - Operation Number: 0
INFO: Read Attempted - Operation Number: 0
```

```

powershell - Shortcut
vagrant@vagrant-ubuntu-trusty-32:/vagrant/Courses/CSE536/ClassProject-Filesystem$ ./jsm.out
=====
NOTE: Please note that file names and file data are to be entered
without any spaces in them. The program is not written to
handle such inputs
=====
Enter the Cell Storage System ID:98307

What do you want to do?
1.Allocate 2.De-allocate 3.Read 4.Write 5.Commit
6.Faulty Commit 7.Abort 8.Exit
1
Enter the file name <Max 28 characters/bytes>: newfile

What do you want to do?
1.Allocate 2.De-allocate 3.Read 4.Write 5.Commit
6.Faulty Commit 7.Abort 8.Exit
4
Operation Num: 0 ! Alloc successful

Enter the file name: newfile

Enter file data <Max 512 characters/bytes>: newfiledata

What do you want to do?
1.Allocate 2.De-allocate 3.Read 4.Write 5.Commit
6.Faulty Commit 7.Abort 8.Exit
5
Operation Num: 0 ! Write successful

Enter operation number to commit: 0

What do you want to do?
1.Allocate 2.De-allocate 3.Read 4.Write 5.Commit
6.Faulty Commit 7.Abort 8.Exit
3
Operation Num: 0 ! Commit successful

Enter the file name: newfile

What do you want to do?
1.Allocate 2.De-allocate 3.Read 4.Write 5.Commit
6.Faulty Commit 7.Abort 8.Exit
0
Operation Num: 0 ! Read successful, <Hit Enter if you don't see the data>
File data is as follows:

newfiledata

```

Outcome: PASS

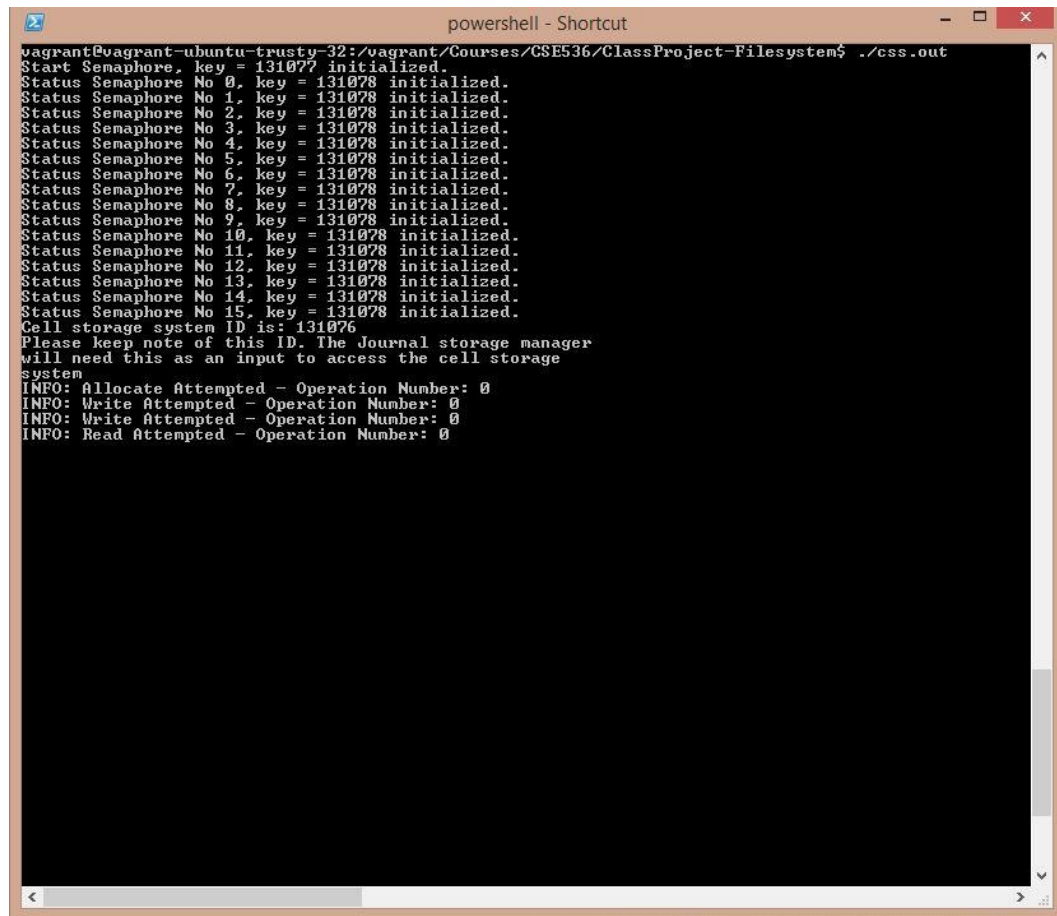
5.2.3 GEN_RD_WR_TEST_2

Pre-requisite

1. Should have created a file with a certain name

Steps	Expected Result	Implementation Mapping
1. Write some information to a position in the disc	Information read at Step 4 should be same information written at Step 3	1. Perform a write operation on the file created
2. Commit the write		2. Perform a commit operation on the write operation number
3. Write another information to the same position (same sector and position if implemented as such)		3. Perform another write operation on the file created.
4. Read from the write		4. Perform another

Steps	Expected Result	Implementation Mapping
location		commit operation on the write operation number. Read from the file.



```
vagrant@vagrant-ubuntu-trusty-32:/vagrant/Courses/CSE536/ClassProject-Filesystem$ ./css.out
Start Semaphore, key = 131077 initialized.
Status Semaphore No 0, key = 131078 initialized.
Status Semaphore No 1, key = 131078 initialized.
Status Semaphore No 2, key = 131078 initialized.
Status Semaphore No 3, key = 131078 initialized.
Status Semaphore No 4, key = 131078 initialized.
Status Semaphore No 5, key = 131078 initialized.
Status Semaphore No 6, key = 131078 initialized.
Status Semaphore No 7, key = 131078 initialized.
Status Semaphore No 8, key = 131078 initialized.
Status Semaphore No 9, key = 131078 initialized.
Status Semaphore No 10, key = 131078 initialized.
Status Semaphore No 11, key = 131078 initialized.
Status Semaphore No 12, key = 131078 initialized.
Status Semaphore No 13, key = 131078 initialized.
Status Semaphore No 14, key = 131078 initialized.
Status Semaphore No 15, key = 131078 initialized.
Cell storage system ID is: 131076
Please keep note of this ID. The Journal storage manager
will need this as an input to access the cell storage
system
INFO: Allocate Attempted - Operation Number: 0
INFO: Write Attempted - Operation Number: 0
INFO: Write Attempted - Operation Number: 0
INFO: Read Attempted - Operation Number: 0
```

```

powershell - Shortcut
=====
Enter the Cell Storage System ID:131076

What do you want to do?
1.Allocate 2.De-Allocate 3.Read 4.Write 5.Commit
6.Faulty Commit 7.Abort 8.Exit
1
Enter the file name <Max 28 characters/bytes>: newfile

What do you want to do?
1.Allocate 2.De-Allocate 3.Read 4.Write 5.Commit
6.Faulty Commit 7.Abort 8.Exit
Operation Num: 0 : Alloc successful
4
Enter the file name: newfile

Enter file data <Max 512 characters/bytes>: newdata1

What do you want to do?
1.Allocate 2.De-Allocate 3.Read 4.Write 5.Commit
6.Faulty Commit 7.Abort 8.Exit
Operation Num: 0 : Write successful
5
Enter operation number to commit: 0

What do you want to do?
1.Allocate 2.De-Allocate 3.Read 4.Write 5.Commit
6.Faulty Commit 7.Abort 8.Exit
Operation Num: 0 : Commit successful
4
Enter the file name: newfile

Enter file data <Max 512 characters/bytes>: newdata2

What do you want to do?
1.Allocate 2.De-Allocate 3.Read 4.Write 5.Commit
6.Faulty Commit 7.Abort 8.Exit
Operation Num: 0 : Write successful
5
Enter operation number to commit: 0

What do you want to do?
1.Allocate 2.De-Allocate 3.Read 4.Write 5.Commit
6.Faulty Commit 7.Abort 8.Exit
Operation Num: 0 : Commit successful
3
Enter the file name: newfile

What do you want to do?
1.Allocate 2.De-Allocate 3.Read 4.Write 5.Commit
6.Faulty Commit 7.Abort 8.Exit
Operation Num: 0 : Read successful, <Hit Enter if you don't see the data>
File data is as follows:
newdata2

```

Outcome: PASS

5.2.4 MULTI_THRD_TEST_1

MULTI_THRD_TEST_1

Pre-requisite

1. Should have created a file with a certain name

Steps	Expected Result	Implementation Mapping
1. Read the information at a certain location in a file	The write in the first thread should block the write in the second thread, i.e. the first thread should execute first and the second should execute next.	1. Perform a write operation on the file created
2. Write information using one thread		2. Perform another write operation on the file created
3. Optionally sleep in the thread.		3. Perform commit operation for each of
4. Write information		

Steps	Expected Result	Implementation Mapping
using another thread		<p>the writes one after the other. When the first commit is being taking place, the second commit can be initiated through another thread.</p> <p>4. After both the threads complete execution, check the order of execution of the threads</p>

```

powershell - Shortcut
vagrant@vagrant-ubuntu-trusty-32:/vagrant/Courses/CSE536/ClassProject-Filesystem$ ./css.out
Start Semaphore, key = 196616 initialized.
Status Semaphore No 0, key = 196616 initialized.
Status Semaphore No 1, key = 196616 initialized.
Status Semaphore No 2, key = 196616 initialized.
Status Semaphore No 3, key = 196616 initialized.
Status Semaphore No 4, key = 196616 initialized.
Status Semaphore No 5, key = 196616 initialized.
Status Semaphore No 6, key = 196616 initialized.
Status Semaphore No 7, key = 196616 initialized.
Status Semaphore No 8, key = 196616 initialized.
Status Semaphore No 9, key = 196616 initialized.
Status Semaphore No 10, key = 196616 initialized.
Status Semaphore No 11, key = 196616 initialized.
Status Semaphore No 12, key = 196616 initialized.
Status Semaphore No 13, key = 196616 initialized.
Status Semaphore No 14, key = 196616 initialized.
Status Semaphore No 15, key = 196616 initialized.
Cell storage system ID is: 196614
Please keep note of this ID. The Journal storage manager
will need this as an input to access the cell storage
system
INFO: Allocate Attempted - Operation Number: 0
INFO: Write Attempted - Operation Number: 0
INFO: Write Attempted - Operation Number: 1
INFO: Read Attempted - Operation Number: 0

```



```

powershell - Shortcut
=====
Enter the Cell Storage System ID:196614
What do you want to do?
1.Allocate 2.De-allocate 3.Read 4.Write 5.Commit
6.Faulty Commit 7.Abort 8.Exit
1
Enter the file name <Max 28 characters/bytes>: newfile
What do you want to do?
1.Allocate 2.De-allocate 3.Read 4.Write 5.Commit
6.Faulty Commit 7.Abort 8.Exit
Operation Num: 0 ! Alloc successful
4
Enter the file name: newfile
Enter file data <Max 512 characters/bytes>: data1
What do you want to do?
1.Allocate 2.De-allocate 3.Read 4.Write 5.Commit
6.Faulty Commit 7.Abort 8.Exit
Operation Num: 0 ! Write successful
4
Enter the file name: newfile
Enter file data <Max 512 characters/bytes>: data1data2
What do you want to do?
1.Allocate 2.De-allocate 3.Read 4.Write 5.Commit
6.Faulty Commit 7.Abort 8.Exit
Operation Num: 1 ! Write successful
5
Enter operation number to commit: 0
What do you want to do?
1.Allocate 2.De-allocate 3.Read 4.Write 5.Commit
6.Faulty Commit 7.Abort 8.Exit
5
Enter operation number to commit: 1
What do you want to do?
1.Allocate 2.De-allocate 3.Read 4.Write 5.Commit
6.Faulty Commit 7.Abort 8.Exit
Operation Num: 0 ! Commit successful
Operation Num: 1 ! Commit successful
3
Enter the file name: newfile
What do you want to do?
1.Allocate 2.De-allocate 3.Read 4.Write 5.Commit
6.Faulty Commit 7.Abort 8.Exit
Operation Num: 0 ! Read successful, <Hit Enter if you don't see the data>
File data is as follows:
data1data2

```

Outcome: PASS

Note - To show the effectiveness of locks, sleep has been implemented in the write procedure of the cell storage system. The sleep kicks in for random threads. So, multiple runs might be needed to see the demonstration of the test.

5.2.5 ERR_PRN_TEST_1

Pre-requisite

1. Should have created a file with a certain name

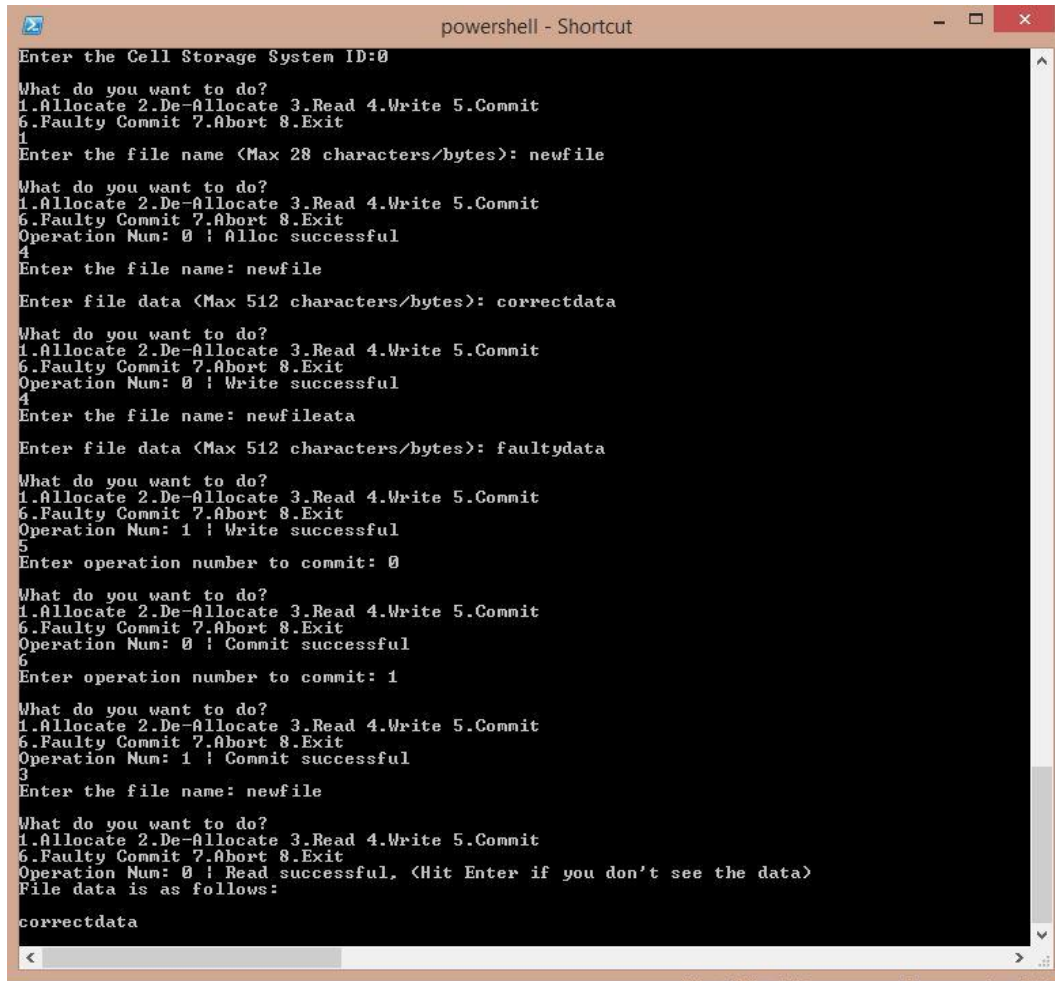
Steps	Expected Result	Implementation Mapping
1. Write data to the file created 2. Perform a faulty write to the file created	The data read out at the end should be the same as the data written in the first write.	1. Perform a write operation on the file created 2. Perform another write operation on the file

Steps	Expected Result	Implementation Mapping
3. Read data from the file		<p>created</p> <p>3. Perform a commit operation on the first write operation</p> <p>4. Perform a faulty commit operation on the second write operation</p> <p>5. Perform a Read</p>

```

powershell - Shortcut
vagrant@vagrant-ubuntu-trusty-32:/vagrant/Courses/CSE536/ClassProject-Filesystem$ ./css.out
Start Semaphore, key = 1 initialized.
Status Semaphore No 0, key = 2 initialized.
Status Semaphore No 1, key = 2 initialized.
Status Semaphore No 2, key = 2 initialized.
Status Semaphore No 3, key = 2 initialized.
Status Semaphore No 4, key = 2 initialized.
Status Semaphore No 5, key = 2 initialized.
Status Semaphore No 6, key = 2 initialized.
Status Semaphore No 7, key = 2 initialized.
Status Semaphore No 8, key = 2 initialized.
Status Semaphore No 9, key = 2 initialized.
Status Semaphore No 10, key = 2 initialized.
Status Semaphore No 11, key = 2 initialized.
Status Semaphore No 12, key = 2 initialized.
Status Semaphore No 13, key = 2 initialized.
Status Semaphore No 14, key = 2 initialized.
Status Semaphore No 15, key = 2 initialized.
Cell storage system ID is: 0
Please keep note of this ID. The Journal storage manager
will need this as an input to access the cell storage
system
INFO: Allocate Attempted - Operation Number: 0
INFO: Write Attempted - Operation Number: 0
INFO: Write Attempted - Operation Number: 1
INFO: Read Attempted - Operation Number: 0
INFO: Read Recovery - File Index: 0

```



```
powershell - Shortcut
Enter the Cell Storage System ID:0
What do you want to do?
1.Allocate 2.De-Allocate 3.Read 4.Write 5.Commit
6.Faulty Commit 7.Abort 8.Exit
1
Enter the file name <Max 28 characters/bytes>: newfile
What do you want to do?
1.Allocate 2.De-Allocate 3.Read 4.Write 5.Commit
6.Faulty Commit 7.Abort 8.Exit
Operation Num: 0 : Alloc successful
4
Enter the file name: newfile
Enter file data <Max 512 characters/bytes>: correctdata
What do you want to do?
1.Allocate 2.De-Allocate 3.Read 4.Write 5.Commit
6.Faulty Commit 7.Abort 8.Exit
Operation Num: 0 : Write successful
4
Enter the file name: newfileata
Enter file data <Max 512 characters/bytes>: faultydata
What do you want to do?
1.Allocate 2.De-Allocate 3.Read 4.Write 5.Commit
6.Faulty Commit 7.Abort 8.Exit
Operation Num: 1 : Write successful
5
Enter operation number to commit: 0
What do you want to do?
1.Allocate 2.De-Allocate 3.Read 4.Write 5.Commit
6.Faulty Commit 7.Abort 8.Exit
Operation Num: 0 : Commit successful
6
Enter operation number to commit: 1
What do you want to do?
1.Allocate 2.De-Allocate 3.Read 4.Write 5.Commit
6.Faulty Commit 7.Abort 8.Exit
Operation Num: 1 : Commit successful
3
Enter the file name: newfile
What do you want to do?
1.Allocate 2.De-Allocate 3.Read 4.Write 5.Commit
6.Faulty Commit 7.Abort 8.Exit
Operation Num: 0 : Read successful, <Hit Enter if you don't see the data>
File data is as follows:
correctdata
```

Outcome: PASS

6. Summary

6.1 Key Features

The key features of the project are

1. The implementation methodology chosen to implement the CSS and the JSM as 2 separate processes is a very scalable design. The CSS can be used as a RAM Disk on it's own.
2. The inter-process communication strategy chosen using shared memory and semaphores is a very complex mechanism and the implementation shines in this area.
3. The implementation itself has been very carefully architected to ensure that changes in the file sizes and number of files supported will only need minimal changes in source code.

Reasons for Choices

The reason for the numbers mentioned in the above table were that the system would maintain all data and metadata on the RAM and hence acts like a RAM disk. Since, the system uses up memory on the RAM, there needs to be some limit on the usage of the memory. So, certain minimal numbers were chosen to keep memory allocated to a minimum. However, the sizes can easily be changed according to the RAM available on the machine because the implementation is such that all sizes and offsets for the file and shared memory have been implemented as macros and all that would be needed would be to change the macros without affecting any code.

Since, the sizes have been limited, the operation bitmap is limited and this is the reason for limiting concurrent operations. The sizes of shared memory and file memory that the system allocates are as follows

1. Shared Memory – 8.47 KB
2. File Memory – 542.125 KB

6.2 Drawbacks

The drawbacks of this project are

1. The memory allocation done on heap is never freed. This can lead to a lot of memory leaks. The reason for not concentrating on memory free is because of the amount of time available.

6.3 Learning experiences

Some of the things learnt from this project are

1. Implementation of inter-process communication using
 - a. Shared Memory
 - b. Semaphores
 - c. Usage of locks
2. Implementation of multiple threads
3. File system design strategies

6.4 Future work

Some of the things that could be done to extend work on this project are

1. Ensuring no memory leaks
2. Extensively testing the complete power of the project and resolve issues seen. If this project can be extensively tested and issues resolved, it can serve as a very easily scalable RAM disk.

References

[1]	Principles of Computer System Design: An Introduction by Jerome H. Saltzer and M. Frans Kaashoek
[2]	Linux man pages for IPC
[3]	Project Description document