

# **Implementation of File System with Journal Storage Manager**

Report for Phase -2

(Error Free environment with multi-threading -  
Before or After atomicity)

---

**ADVANCED OPERATING SYSTEM  
CSE 536  
ARIZONA STATE UNIVERSITY**

---

**SHARAD  
1210268966**

## Idea

Phase 2 of the project is to implement the file system with a Journal Storage Manager which serves as an interface between client and the Cell Storage System. This phase of the project introduces the concept of 'Before-or-After' atomicity in the Version History approach. Having chosen that approach, I have used the mark point discipline to show the before-or-after action among the various File ID's that would be updated by the client. Various assumptions (as discussed later) has been made to correctly implement the discipline in a very restrictive yet powerful way. The procedures are analogous to the ones provided by the textbook Saltzer & Kaashoek, Chapter 9, Atomicity.

## The Mark-Point Discipline

In simple words, this discipline demands that before any read could take place by any client with a given transaction id, the previous client should have created all the versions of File ID's it is going to affect(or update). The way to do that is to modify read\_value of previous phase to wait for, rather than skip over, pending versions created by transactions that are earlier in the sequential ordering (that is, they have a smaller transaction\_id).

Each transaction should create new, pending versions of every variable it intends to modify, and announce when it is finished doing so. Creating a pending version has the effect of marking those variables that are not ready for reading by later transactions, so we will call the point at which a transaction has created them all the *mark point* of the transaction. These features has been achieved by the functions

1. new\_version()
2. mark\_point\_announce(transaction id)

When finished marking, the transaction calls `mark_point_announce()`. It may then go about its business, reading and writing values as appropriate to its purpose.

The other procedure of concern is the one which creates new outcome record.  
`new_outcome_record()`

The procedure must itself be a before-or-after action because it may be invoked concurrently by several different threads and it must be careful to give out different serial numbers to each of them. It must also create completely initialised outcome records, with value and `mark_state` set to `PENDING` and `NULL`, respectively, because a concurrent thread may immediately need to look at one of those fields.

## **No Deadlock**

A useful property of the mark-point discipline is that it never creates deadlocks. Whenever a wait occurs, it is a wait for some transaction *earlier* in the serialization. (The wait has been implemented by while loop which keeps the client asking to retry until the other client declares all the mark points.) That transaction may in turn be waiting for a still earlier transaction, but since no one ever waits for a transaction later in the ordering, progress is guaranteed. The reason is that at all times there must be some earliest pending transaction. The ordering property guarantees that this earliest pending transaction will encounter no waits for other transactions to complete, so it, at least, can make progress. When it completes, some other transaction in the ordering becomes earliest, and it now can make progress. Eventually, by this argument, every transaction will be able to make progress. This kind of reasoning about progress is a helpful element of a before-or-after atomicity discipline.

## Assumptions

1. As mentioned in the book, the data objects has been represented as File ID's. The client has an option to select any one of the three pre-defined ID's which it wants to update.
2. The storage for File ID's is created at Journal Storage Manager when the client chooses to create a version for that ID.
3. Every client is automatically provided with an unique (the sequencer that generates this id is locked before being called) which is used by it for further transactions.
4. Once committed, the state of the outcome record for a given transaction ID can not be changed.
5. Any number of reads and writes are possible by a client, as long as it follow the constraint that reads and writes should only take place from and to the File ID's that it already declared to update. Any try to access ID's outside its domain would result in an error.
6. The data to be written on the file should be of length 15 bytes or less and should not contain space.

## Operations

The following operations can be performed by the client.

1. On entering the system, declare the File ID's it wants to update. The File ID's should be selected from the list of pre defined ID's (which is limited to 3 in this case).
  2. Say 'Yes' when asked if it wants to create a version for the already selected ID's. While it has not created all the versions, any further transaction would be asked to wait.
  3. Mark Point is declared once client accepts to create all the versions for the ID's it wants to update. Further transactions can now access the system and declare their own versions.
  4. When presented with a menu containing choices to
    - read
    - write
    - commit
    - abortthe client should select one of the options.
- 
- I. At first transaction, the client should write a value to a selected File ID and then it has an option to commit it.
  - II. Once committed, the client can read the value. The most recent value which has been committed for the File ID (it could be from a different previous transaction ID as per the outcome record) would be displayed.
  - III. Client can choose to abort the system once all the transaction wished has been taken place.

## Procedure Descriptions

### **begin\_transaction()**

The program starts with a call to `begin_transaction()`. The procedure calls *new\_outcome\_record*, which returns a unique `transaction_id` for the client. Before giving the client an option to create its own versions, a check is made for the previous transaction ID's mark status. If 'marked', it returns to calling procedure for further steps. If 'unmarked', the client is asked to wait till the previous transaction ID is marked. Client can keep retrying in the meanwhile.

### **new\_outcome\_record()**

The `new_outcome_record` is responsible for creating unique transaction ID's for the client. Since more than two threads would be trying to create a new transaction ID, it is necessary and sufficient condition to implement a single lock around the call to sequencer which provides the new ID. In our case, the sequence is implemented using a single test file called *t\_id.txt*. Every call to *t\_id.txt* is locked and is incremented by one and writes back to the file. Thus the uniqueness of ID is maintained.

The above action of acquiring lock is a **before-or-after** action.

### **new\_version()**

The `new_version` creates a new version for a given transaction ID and data ID. The new version is appended to the version history list in the *version\_file.bin*. The file contains a list of structure containing the Data associated with the given transaction ID and data ID. Initially the data field of the structure is stored as 'NULL', which can be later changed by using the `write_value()` operation.

### **mark\_point\_announce(current\_transaction\_id)**

The `mark_point_announce` is called when the client is done creating all the version for File ID's it has selected to update. The procedure looks for the outcome record of the transaction ID in the *outcome\_file.bin*. If the record is

found to be 'unmarked' (the *mark\_status of the structure*), it is updated as 'marked'.

### **display\_menu()**

Once the marked point discipline has been taken care of, the client is presented with a menu of actions it can perform on the File ID's it has selected for update. It contains the option for

- read
- write
- commit
- abort

Choosing among the choices calls the appropriate procedure associated with the choice.

### **write\_value(this\_transaction\_id, file\_id)**

The *write\_value* is passed the transaction ID and the ID of the file which needs to be updated. The *version\_file.bin* is opened to look for the corresponding File ID bounded to the transaction ID. The client is then asked for the data to be written in the File ID structure. The provided data is propagated to the version structure and the default 'NULL' is replaced by the data provided.

### **read\_current\_value(this\_transaction\_id, file\_id)**

The *read\_current\_value* is passed the current transaction ID and the ID of the file which needs to be read. The first step is to confirm the status of the transaction id. The procedure *if\_committed()* serves this purpose. If not found to be committed, the previous ID to it is seeded and checked for the status. After all the checks, the most recent ID which is found to have both the 'COMMITTED' status as well as an update to the File ID, returns its associated data and that is displayed to the client.

### **if\_committed(transaction\_id)**

The *if\_committed* checks the *outcome\_file.bin* for the state of the transaction. If found COMMITTED, it returns true, else returns false.

## Test cases and reason for the choice

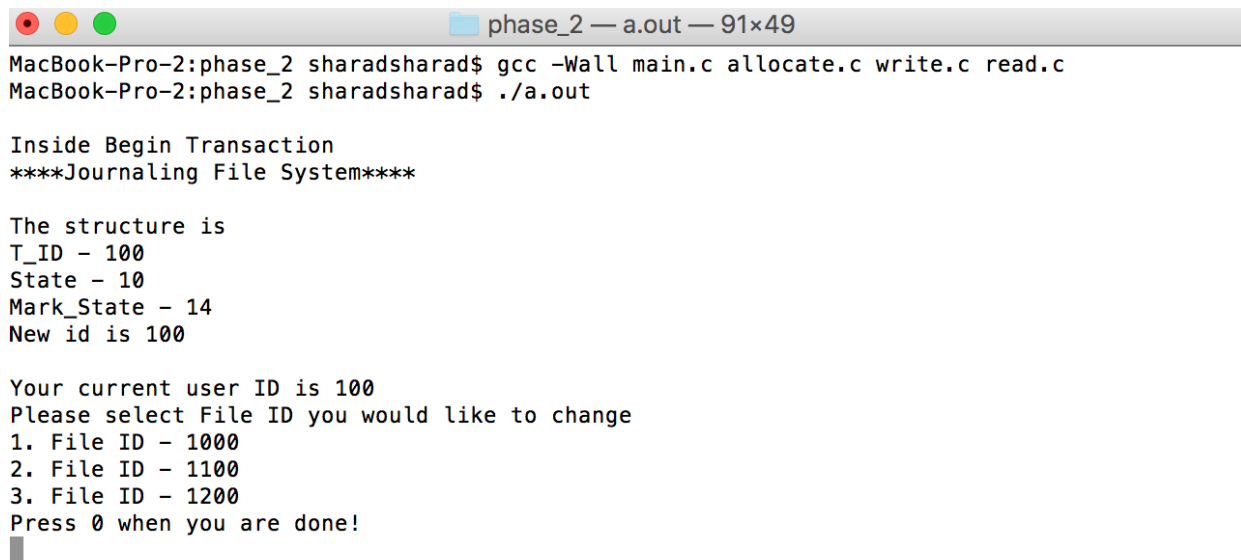
1. We open the main program in two different terminals. The first opened program receives a unique transaction ID and is asked to choose the File ID's which it is going to update. Meanwhile the program in the 2nd terminal is asked to wait till the 1st client is done creating all the versions for selected data ID's. Once the first client is done creating all the new versions, the client on second terminal, if retry, is allowed to choose its own file ID's. This assures the 'Before-and-After' mechanism among various threads trying to use same File ID's.
2. We try to start two threads at a same time which tries to allocate transaction ID's to both the clients. It program assures that allocated transaction ID's are sequential and unique.
3. Once the Commit has been called for given File ID by a transaction ID, all further reads assures that the most recent COMMITTED version of that File ID is displayed to the client.

All the cases are sufficient to test the Journaling File System in the error free environment for 'Before-and-After' atomicity requirement. Since we are not worried about any unexpected abort, a test for abort case sounds unnecessary and futile.



## Screenshots for Test Cases and Outcomes

1. On start, the **begin\_transaction** procedure is called. The client is given an option to select from 3 pre defined file ID's to which it is going to perform changes. The client should choose at least one and at most three of those ID's to proceed.



```
MacBook-Pro-2:phase_2 sharadsharad$ gcc -Wall main.c allocate.c write.c read.c
MacBook-Pro-2:phase_2 sharadsharad$ ./a.out

Inside Begin Transaction
****Journaling File System****

The structure is
T_ID - 100
State - 10
Mark_State - 14
New id is 100

Your current user ID is 100
Please select File ID you would like to change
1. File ID - 1000
2. File ID - 1100
3. File ID - 1200
Press 0 when you are done!
```

2. Then the client proceeds with entering the name of the ID's it plans to update. It ends with a '0' to finish entering its choices of ID's.

```
phase_2 — a.out — 91x49
MacBook-Pro-2:phase_2 sharadsharad$ gcc -Wall main.c allocate.c write.c read.c
MacBook-Pro-2:phase_2 sharadsharad$ ./a.out

Inside Begin Transaction
****Journaling File System****

The structure is
T_ID - 100
State - 10
Mark_State - 14
New id is 100

Your current user ID is 100
Please select File ID you would like to change
1. File ID - 1000
2. File ID - 1100
3. File ID - 1200
Press 0 when you are done!
1100
1200
0
```

3. The client is then asked to create a version for first File ID it selected.

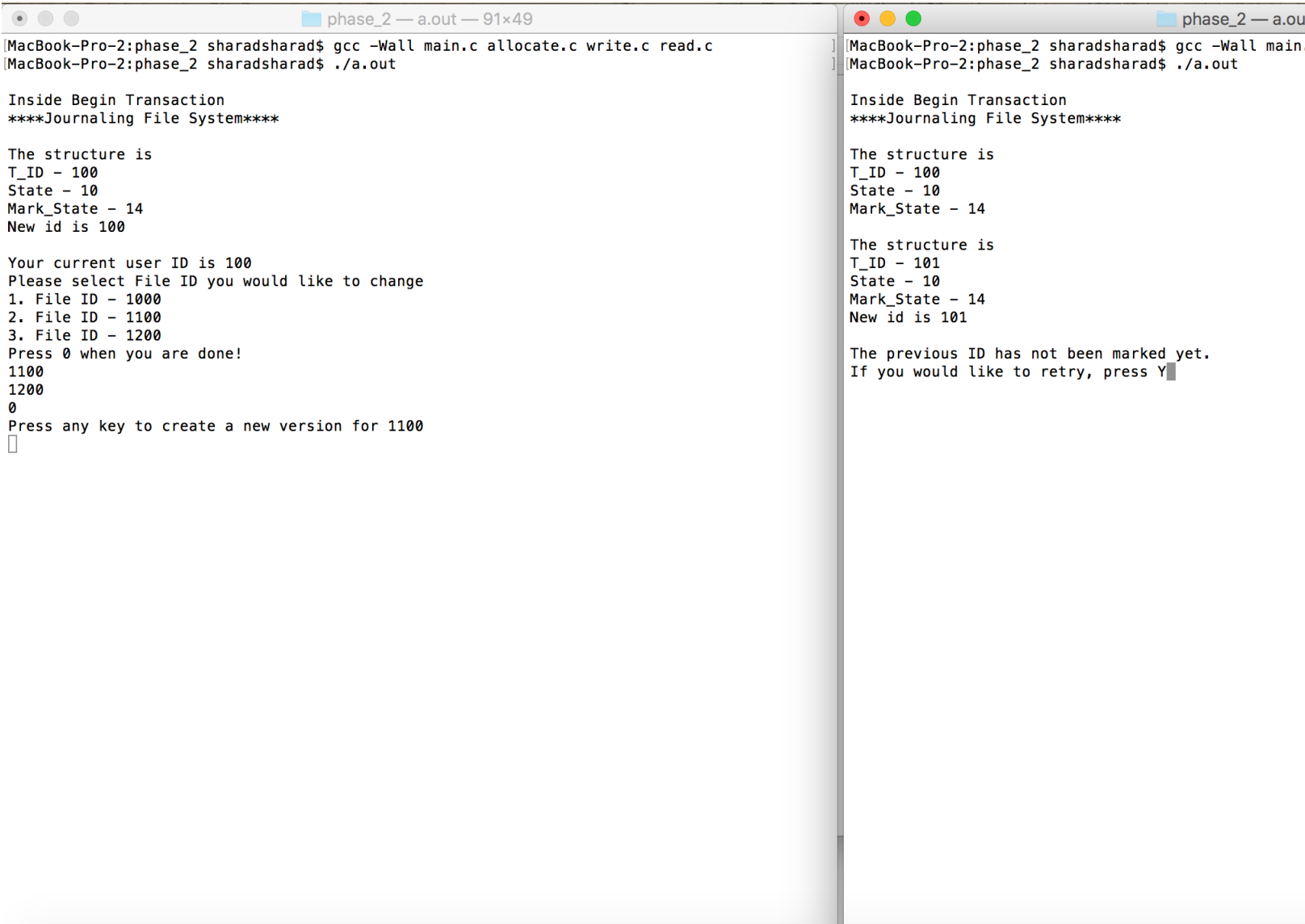
```
phase_2 — a.out — 91x49
MacBook-Pro-2:phase_2 sharadsharad$ gcc -Wall main.c allocate.c write.c read.c
MacBook-Pro-2:phase_2 sharadsharad$ ./a.out

Inside Begin Transaction
****Journaling File System****

The structure is
T_ID - 100
State - 10
Mark_State - 14
New id is 100

Your current user ID is 100
Please select File ID you would like to change
1. File ID - 1000
2. File ID - 1100
3. File ID - 1200
Press 0 when you are done!
1100
1200
0
Press any key to create a new version for 1100
```

4. Meanwhile, in a second terminal, we start a different thread of the program. The new client is asked to wait till the previous client is done marking the transaction ID by creating all the selected ID's versions.



```
MacBook-Pro-2:phase_2 sharadsharad$ gcc -Wall main.c allocate.c write.c read.c
MacBook-Pro-2:phase_2 sharadsharad$ ./a.out

Inside Begin Transaction
****Journaling File System****

The structure is
T_ID - 100
State - 10
Mark_State - 14
New id is 100

Your current user ID is 100
Please select File ID you would like to change
1. File ID - 1000
2. File ID - 1100
3. File ID - 1200
Press 0 when you are done!
1100
1200
0
Press any key to create a new version for 1100

```

```
MacBook-Pro-2:phase_2 sharadsharad$ gcc -Wall main.c
MacBook-Pro-2:phase_2 sharadsharad$ ./a.out

Inside Begin Transaction
****Journaling File System****

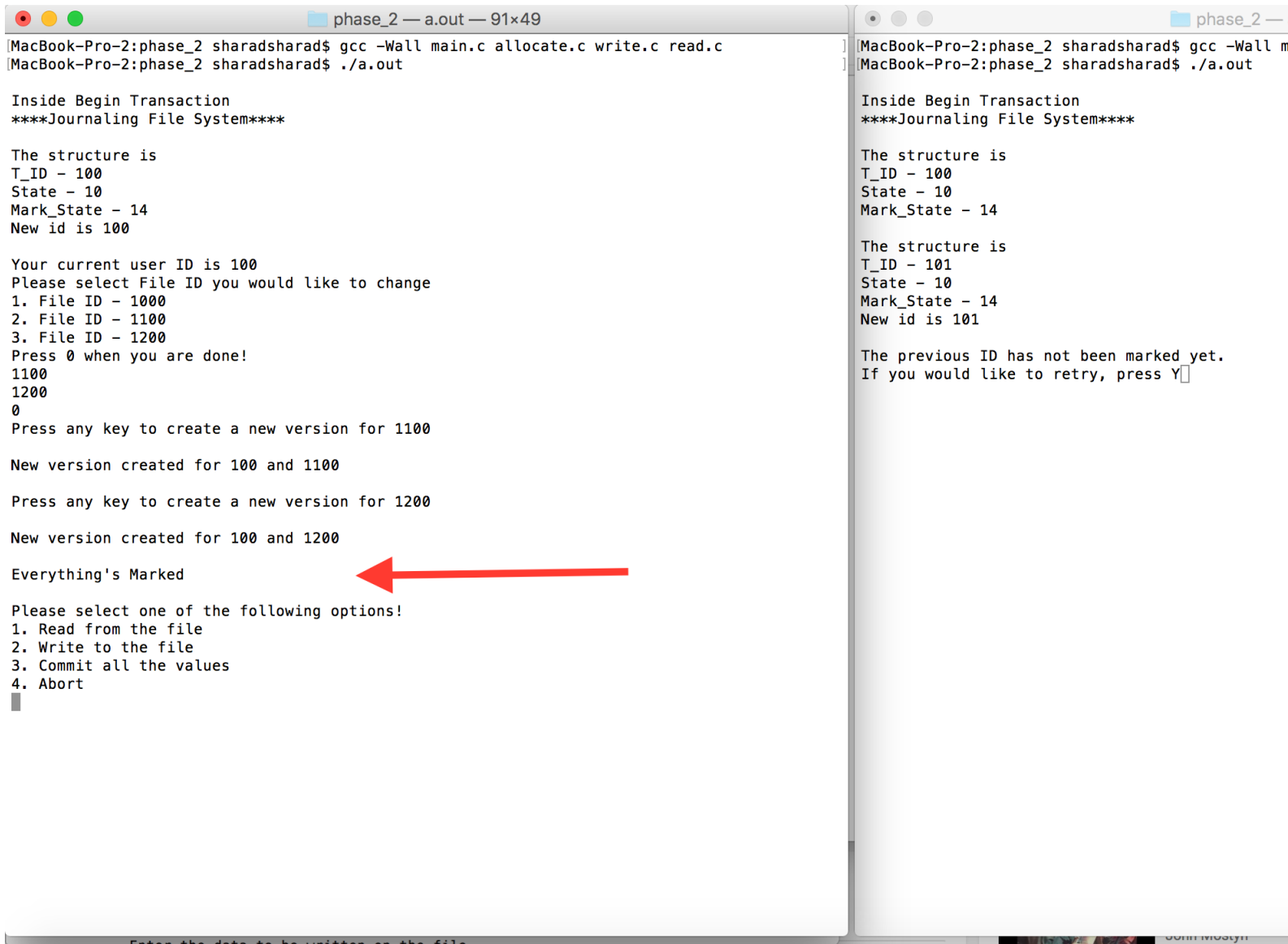
The structure is
T_ID - 100
State - 10
Mark_State - 14
New id is 100

The structure is
T_ID - 101
State - 10
Mark_State - 14
New id is 101

The previous ID has not been marked yet.
If you would like to retry, press Y

```

5. The 1st thread is done creating all the versions for the selected ID's. At this point, mark\_point\_announce is called. In turn it displays the mark point has been updated.



```
MacBook-Pro-2:phase_2 sharadsharad$ gcc -Wall main.c allocate.c write.c read.c
MacBook-Pro-2:phase_2 sharadsharad$ ./a.out

Inside Begin Transaction
****Journaling File System****

The structure is
T_ID - 100
State - 10
Mark_State - 14
New_id is 100

Your current user ID is 100
Please select File ID you would like to change
1. File ID - 1000
2. File ID - 1100
3. File ID - 1200
Press 0 when you are done!
1100
1200
0
Press any key to create a new version for 1100

New version created for 100 and 1100

Press any key to create a new version for 1200

New version created for 100 and 1200

Everything's Marked
Please select one of the following options!
1. Read from the file
2. Write to the file
3. Commit all the values
4. Abort
█

MacBook-Pro-2:phase_2 sharadsharad$ gcc -Wall main.c allocate.c write.c read.c
MacBook-Pro-2:phase_2 sharadsharad$ ./a.out

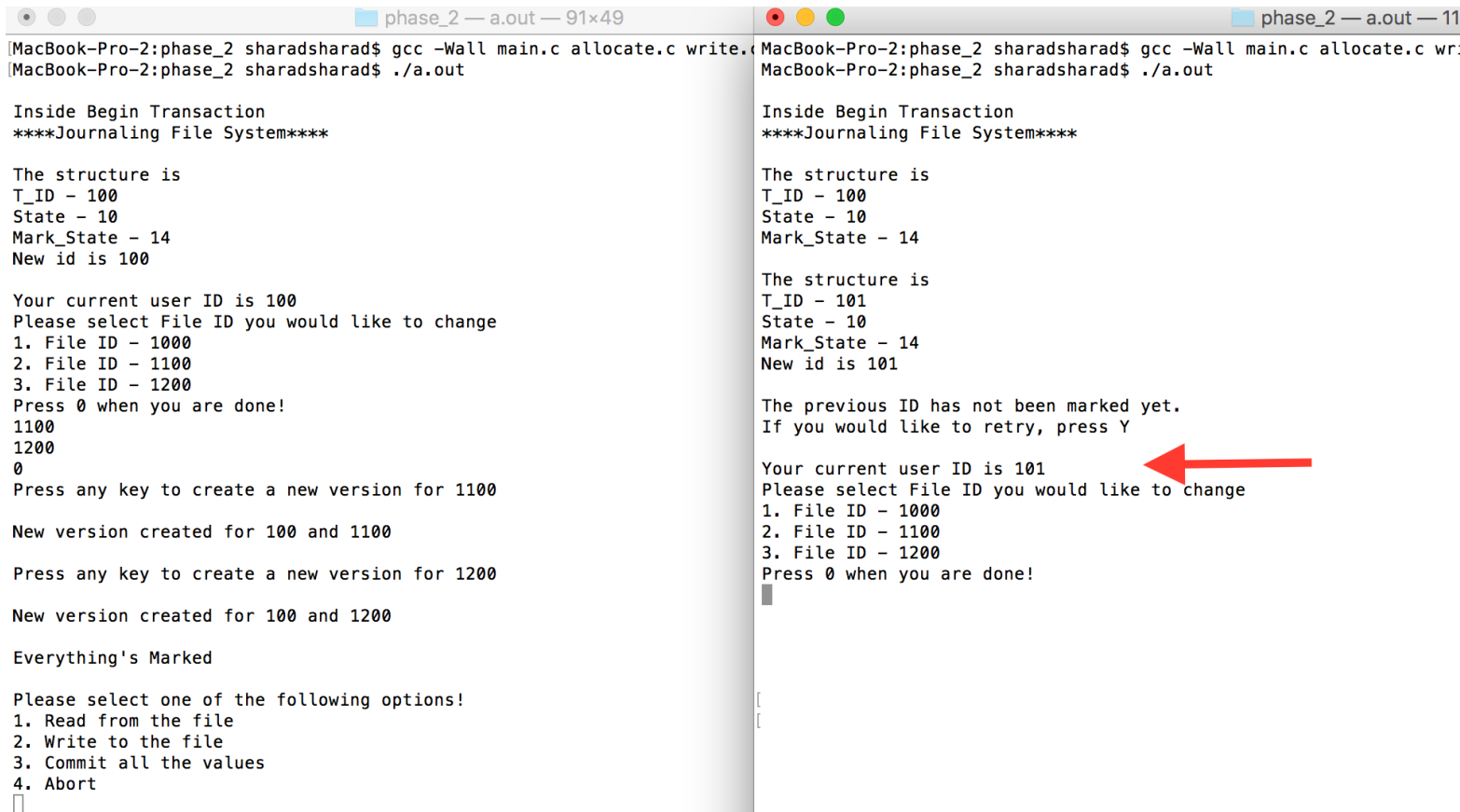
Inside Begin Transaction
****Journaling File System****

The structure is
T_ID - 100
State - 10
Mark_State - 14
New_id is 100

The structure is
T_ID - 101
State - 10
Mark_State - 14
New_id is 101

The previous ID has not been marked yet.
If you would like to retry, press Y
```

6. The second thread now retries to receive an ID. Since the first thread has declared the *marked status*, the second thread can acquire a new ID.



```
MacBook-Pro-2:phase_2 sharadsharad$ gcc -Wall main.c allocate.c write.c
MacBook-Pro-2:phase_2 sharadsharad$ ./a.out

Inside Begin Transaction
****Journaling File System****

The structure is
T_ID - 100
State - 10
Mark_State - 14
New id is 100

Your current user ID is 100
Please select File ID you would like to change
1. File ID - 1000
2. File ID - 1100
3. File ID - 1200
Press 0 when you are done!
1100
1200
0
Press any key to create a new version for 1100

New version created for 100 and 1100

Press any key to create a new version for 1200

New version created for 100 and 1200

Everything's Marked

Please select one of the following options!
1. Read from the file
2. Write to the file
3. Commit all the values
4. Abort
[

MacBook-Pro-2:phase_2 sharadsharad$ gcc -Wall main.c allocate.c write.c
MacBook-Pro-2:phase_2 sharadsharad$ ./a.out

Inside Begin Transaction
****Journaling File System****

The structure is
T_ID - 100
State - 10
Mark_State - 14
New id is 100

The structure is
T_ID - 101
State - 10
Mark_State - 14
New id is 101

The previous ID has not been marked yet.
If you would like to retry, press Y

Your current user ID is 101
Please select File ID you would like to change
1. File ID - 1000
2. File ID - 1100
3. File ID - 1200
Press 0 when you are done!
[
```

7. The client using the first thread can now select from the menu. Being the first time, it decides to 'write' to a provided file ID.

```
phase_2 — a.out — 91x49
MacBook-Pro-2:phase_2 sharadsharad$ gcc -Wall main.c allocate.c write.c read.c
MacBook-Pro-2:phase_2 sharadsharad$ ./a.out

Inside Begin Transaction
****Journaling File System****

The structure is
T_ID - 100
State - 10
Mark_State - 14
New id is 100

Your current user ID is 100
Please select File ID you would like to change
1. File ID - 1000
2. File ID - 1100
3. File ID - 1200
Press 0 when you are done!
1100
1200
0
Press any key to create a new version for 1100

New version created for 100 and 1100

Press any key to create a new version for 1200

New version created for 100 and 1200

Everything's Marked

Please select one of the following options!
1. Read from the file
2. Write to the file
3. Commit all the values
4. Abort
2
Enter the File ID you would like to change
1000
```

8. Since it has entered an id which it has not previously declared as the one it would like to update, an error is thrown and it would be asked to retry the action.

```
phase_2 — a.out — 91x49
MacBook-Pro-2:phase_2 sharadsharad$ gcc -Wall main.c allocate.c write.c read.c
MacBook-Pro-2:phase_2 sharadsharad$ ./a.out

Inside Begin Transaction
****Journaling File System****

The structure is
T_ID - 100
State - 10
Mark_State - 14
New id is 100

Your current user ID is 100
Please select File ID you would like to change
1. File ID - 1000
2. File ID - 1100
3. File ID - 1200
Press 0 when you are done!
1100
1200
0
Press any key to create a new version for 1100

New version created for 100 and 1100

Press any key to create a new version for 1200

New version created for 100 and 1200

Everything's Marked

Please select one of the following options!
1. Read from the file
2. Write to the file
3. Commit all the values
4. Abort
2
Enter the File ID you would like to change
1000

Please enter the ID that you have declared to change!
Enter the File ID you would like to change
█
```



9. On entering the correct File ID, it is asked to enter the data to be stored in the File ID. On entering the correct ID, the File ID has (stored as data\_id in the structure) been updated to the new value, as can be seen from the structure state.

```
phase_2 — a.out — 91x49
2. File ID - 1100
3. File ID - 1200
Press 0 when you are done!
1100
1200
0
Press any key to create a new version for 1100

New version created for 100 and 1100

Press any key to create a new version for 1200

New version created for 100 and 1200

Everything's Marked

Please select one of the following options!
1. Read from the file
2. Write to the file
3. Commit all the values
4. Abort
2
Enter the File ID you would like to change
1000

Please enter the ID that you have declared to change!
Enter the File ID you would like to change
1200

Enter the value to be written to the file id 1200
data_1200
The data has been succesfully written to 1200

The structure is
Data - NULL
Transaction ID - 100
Data ID - 1100

The structure is
Data - data_1200
Transaction ID - 100
Data ID - 1200

Please select one of the following options!
1. Read from the file
2. Write to the file
3. Commit all the values
4. Abort
```



10. The 1st thread now decides to commit it's state as *COMMITTED*. This is successfully done by the Commit procedure.

```
phase_2 — a.out — 91x49

Press any key to create a new version for 1200

New version created for 100 and 1200

Everything's Marked

Please select one of the following options!
1. Read from the file
2. Write to the file
3. Commit all the values
4. Abort
2
Enter the File ID you would like to change
1000

Please enter the ID that you have declared to change!
Enter the File ID you would like to change
1200

Enter the value to be written to the file id 1200
data_1200
The data has been succesfully written to 1200

The structure is
Data - NULL
Transaction ID - 100
Data ID - 1100

The structure is
Data - data_1200
Transaction ID - 100
Data ID - 1200

Please select one of the following options!
1. Read from the file
2. Write to the file
3. Commit all the values
4. Abort
3

Transaction ID 100 has been succesfully committed!
Please select one of the following options!
1. Read from the file
2. Write to the file
3. Commit all the values
4. Abort
```

11. The client now decides to read the committed value. The most recent committed value stored in Version History by the Journal Manager is displayed.

```
phase_2 — a.out — 91x49
3. Commit all the values
4. Abort
2
Enter the File ID you would like to change
1000

Please enter the ID that you have declared to change!
Enter the File ID you would like to change
1200

Enter the value to be written to the file id 1200
data_1200
The data has been successfully written to 1200

The structure is
Data - NULL
Transaction ID - 100
Data ID - 1100

The structure is
Data - data_1200
Transaction ID - 100
Data ID - 1200

Please select one of the following options!
1. Read from the file
2. Write to the file
3. Commit all the values
4. Abort
3

Transaction ID 100 has been successfully committed!

Please select one of the following options!
1. Read from the file
2. Write to the file
3. Commit all the values
4. Abort
1
Enter the File ID you would like to read from
1200
The data in data id 1200 for action id 100 is data_1200

Please select one of the following options!
1. Read from the file
2. Write to the file
3. Commit all the values
4. Abort
```

12. When the client selects to abort the system, all actions are aborted.

```
phase_2 — -bash — 91x49

4. Abort
2
Enter the File ID you would like to change
1000

Please enter the ID that you have declared to change!
Enter the File ID you would like to change
1200

Enter the value to be written to the file id 1200
data_1200
The data has been succesfully written to 1200

The structure is
Data - NULL
Transaction ID - 100
Data ID - 1100

The structure is
Data - data_1200
Transaction ID - 100
Data ID - 1200

Please select one of the following options!
1. Read from the file
2. Write to the file
3. Commit all the values
4. Abort
3

Transaction ID 100 has been succesfully committed!

Please select one of the following options!
1. Read from the file
2. Write to the file
3. Commit all the values
4. Abort
1
Enter the File ID you would like to read from
1200
The data in data id 1200 for action id 100 is data_1200

Please select one of the following options!
1. Read from the file
2. Write to the file
3. Commit all the values
4. Abort
4
MacBook-Pro-2:phase_2 sharadsharad$
```

## Summary

- This file system introduces journal management system where maintains all the versions of the data.
- Journal file system ensures All-or-nothing and Before-or-After atomicity is maintained.
- All-or-nothing atomicity ensures that if data is written into cell storage it should either get the new committed value but not the value which is in pending state or abort state.
- Before-orAfter atomicity ensures that all the versions are created before new transaction ID could be issued.
- The creation of unique transaction ID is also a 'Before-or-After' action.
- At any point of time, committed data can be retrieved from the Version History maintained by the Journal File System.

## Conclusion

- In Journal File System, writing of data follow the atomicity principle and is consistently maintained.
- Journal storage manager expects to receive tentative values, but ignores them unless the all-or-nothing action commits.
- Journal storage manager makes sure that all new transactions happens either completely before or completely after any other transaction under process.

## Lesson Learnt

- Any system that consists of multiple operations, to work atomic in nature, requires it to follow a set of principles and methods that confirms that its non-atomic nature is not visible to the upper layer.
- File Systems in most of the operating systems are not implemented with journaling system as other methods may be used. Journaling makes it easier for a file system to store all the changes made in the file.
- The 'Before-or-After' atomicity principle is highly valuable in transactions where multiple clients tries to access a critical resource at the same time. Previously declaring the required data id's is beneficial for the concurrency.