SE 536 – Operating Systems

Project Report On Journal File SYSTEM PART 2,3,4

**Project Report# 2**

**Assigned Date:** Mar 25, 2015

**Due Date:** Apr 27, 2015

**ASU ID :** 1207663258

|  |  |
| --- | --- |
| *Last Name* | REKALA |
| *First Name* | KRANTHI KUMAR |

**Introduction/Problem Description:**

The Project is aimed at implementing a File System called Journal File System Using the C programming language on Linux Platform. This part of the project takes into assumption that the Environment in which the program runs is an “Error – Prone and uses multithreading”. This project is aimed at testing the concept of All-or-Nothing Atomicity and Before-or-after atomicity in case of a multi-threaded operation. In an Operating System the File System has function of mainly managing the operations of READ, WRITE, ABORT and COMMIT to manage multiple threads of seamlessly. The File System should be able to handle the important issues like race conditions, deadlocks, crashes and appropriate recovery mechanisms are to be implemented. The FS should be able to confirm the atomicity concepts of before-or-after and all-or-nothing to ensure that correct output is given to the user even in case of inappropriate behavior of above operations.

The Following are some of the assumptions that are made to implement this project:

* All the memory allocation to the File System is done using arrays and hence all the required memory is allocated at the start of the program. Hence the modules ALLOCATE and DEALLOCATE are not required in my implementation.
* A log has been implemented for recovery and for testing the Test Cases, it is requested that the log files with names journal\_log.txt and outcome\_record.txt are created for each test case so that the system can handle crash.
* I have implemented dynamic assignment for the threads and hence I sincerely apologize that the output might be little garbled, but I will try to elaborate and explain as better as possible.
* I have implemented two types of crashes wherein abrupt shutdown of the running program is done to ensure that an external event has caused crash. Secondly, a thread cancel has been used which pops a message that thread has faulted and subsequently we have to crash the program once it enters the FAULTY\_WRITE method.
* Input Data\_id’s are generally alphabets and values are integers.
* Thread operations in a multi-threaded systems are difficult to control.

**Implementation/Design Decisions:**

The Following are the methods implemented:

* READ\_CURRENT\_VALUE – returns the last committed value by taking data id
* WRITE\_NEW\_VALUE – takes the data id and value to write to the storage and asks for COMMIT
* NEW\_ACTION – This method can only called by READ or WRITE, FAULTY\_WRITE methods only.
* COMMIT – Only WRITE or FAULTY\_WRITE can call this method, however implementation is provided to commit a particular transaction.
* ABORT – Aborts the transaction by taking transaction id as input
* FAULTY\_WRITE – implementation of Faulty write which eventually crashes after writing to cell storage.
* READ\_FROM\_LOG – after system crash, the values of committed transactions are read back into the memory once the program is started using this routine.
* WRITE\_TO\_LOG – every committed transaction is written to non-volatile memory i.e. text files named journal\_log.txt and outcome\_record.txt which are in CSV format and read back after system recovers from crash.

As the threads are asking for inputs dynamically in my implementation, mutex\_locks are used to achieve synchronization between the threads. Locks are primarily implemented for creating a transaction id and the two WRITE methods to ensure that race conditions for the resources is avoided. Sleep for different operations are implemented with different sleep times and tested to ensure that dead locks are avoided. The READ method is not locked to ensure that parallel reads are possible. Other functions are also implemented without locking as multiple accesses can be allowed.

COMMIT is implemented as to be called only by WRITE methods as I felt it’s logical to commit only post writing some values. Although a COMMIT routine is implemented which takes a transaction id and COMMITS a transaction, its use is not recommended to avoid conflicting values.

Multiple header files like JSM.h, read.h, write.h,global\_variables.h are designed as header files as per the project requirement and multiple variables are declared as global variables and used to ensure that both the threads will be able to access the same values. Presently my system is implemented by taking into consideration two threads and can handle a single crash. However, the implementation can be extended as a future work to handle multiple threads and faults.

The Following are the arrays that are used to visualize the implementation. The Committed values are updated in the outcome\_record and data\_record while temp values are stored in a temp array. The Values are stored in a text file and written to it for every COMMIT operation. Immediately post-crash of the system, these values are read back into the arrays to achieve recovery from crash.

The Arrays in the next page are used for read and write operations. Transaction ID acts the primary key in both the tables.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| data\_record | | | | |
| data\_id | | transaction\_id | | Value |
| a | | 1002 | | 150 |
| b | | 1003 | | 200 |
| c | | 1004 | | 175 |
| d | | 1005 | | 200 |
| a | | 1006 | | 500 |
| outcome\_record | | |
| data\_id | transaction\_id | |
| a | 1002 | |
| b | 1003 | |
| c | 1004 | |
| d | 1005 | |
| a | 1006 | |

**Test Cases, Results and Output Screen shots:**

The Following are the assumptions with respect to the project I implemented:

All the allocations of variables is static and hence the test cases are to be implemented in one go

Let’s start the Test Cases **(kindly ignore the warnings, they are due to C language type casting and doesn’t impact the working of application, the implementation is error-free):**

Part 2: Error Free Environment with multi-threading (before or after atomicity)

**ID: Test Case 1**

Purpose: Before-or-After Atomicity

Prerequisite:

1. Read, Write, Commit Procedures have been implemented.

Steps:

0. Read the information at a certain location in a file.

1. Write information to the disk (using one thread)

2. Optional: Sleep in that thread for a time being.

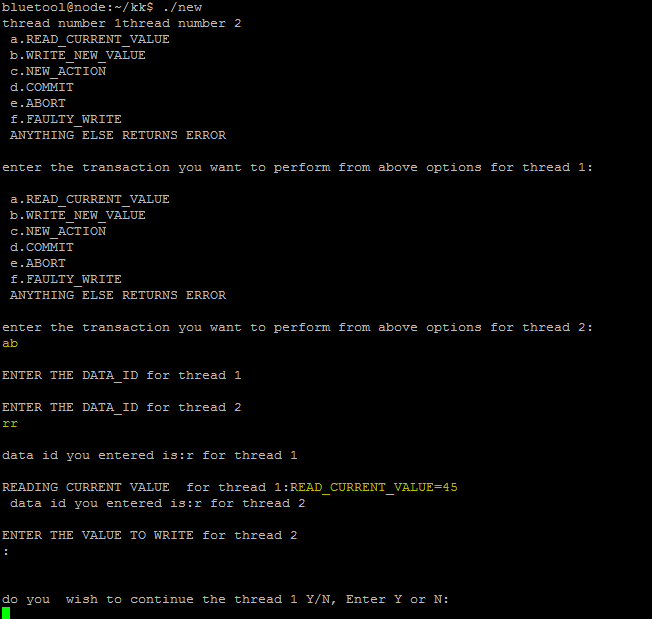
3. Try to write to that location in the FS, using another thread(s)

Expected:

1. The second and subsequent threads should return unsuccessful and call your programs appropriate handler (which might decide to wait for a certain time and try to reacquire the lock).

0. Read the information at a certain location in a file (assuming that the previous program has written some values already)

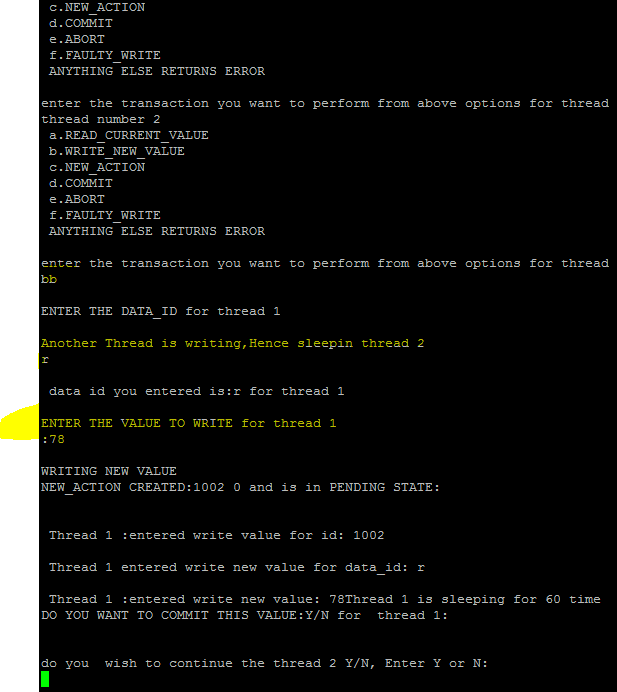
1, 2, 3 are explained with screenshots for better visibility.



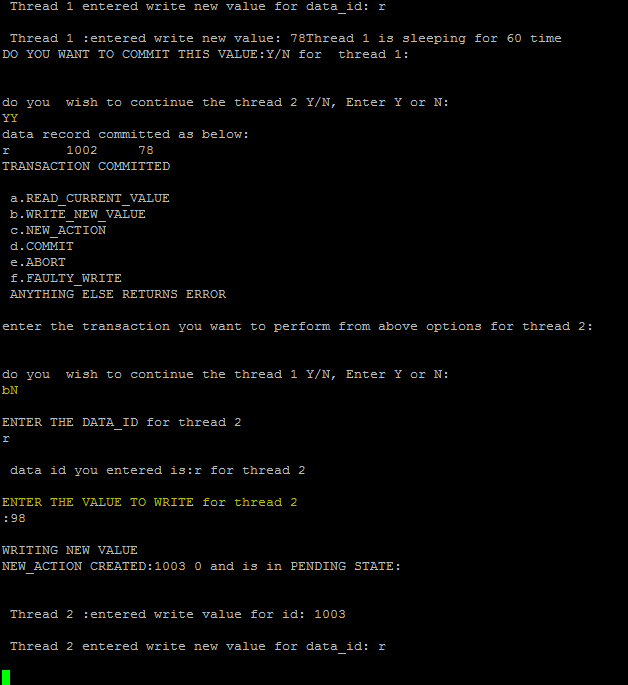
The Option for threads are given as a. read and b. Write

Thread reads a value of 45 for the data\_id ‘r’. Now we will try to write another value to r to same location and sleep in that thread. Please start the program again.

Now we will try performing two writes simultaneously such that the second thread pops out a message that another thread is accessing the WRITE and goes to sleep for some time and starts back again.



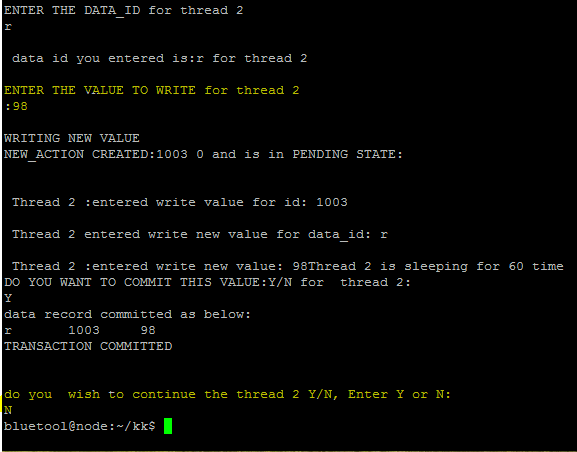
The Scenario can be explained as below. Thread 1 has acquired the lock and hence is asking for data values. Thread 2 was unable to acquire a lock and hence went into sleep for 70 ms as per my program while Thread 1 is sleeping post writing values to outcome\_record and hence wouldn’t give lock back unless we mention commit and ask it to stop its operation which will be done in next subsequent steps.



The option of YY in above screenshot gives a Yes to commit for thread and yes for thread 2 to retry its operation.

The input bN ensures that thread 2 would retry the WRITE and thread 1 is exited

Now the values for thread2 are to be committed and exited as a single threaded operation.



Outcome:

PASS/FAIL.

PASS

**ID: Test Case [3]**

Purpose: Test Error-Prone Environment

Prerequisite:

1. Read, Write, Commit Procedures have been implemented.

2. Error prone read/write procedures

Steps:

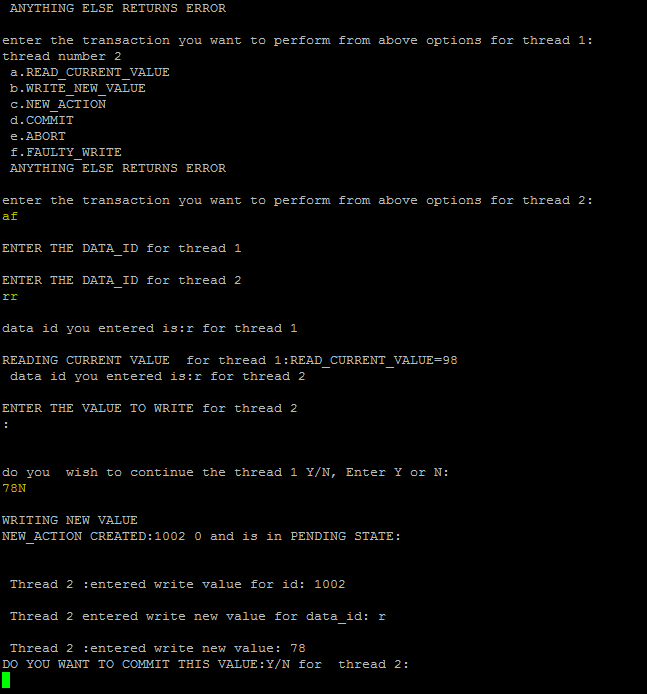
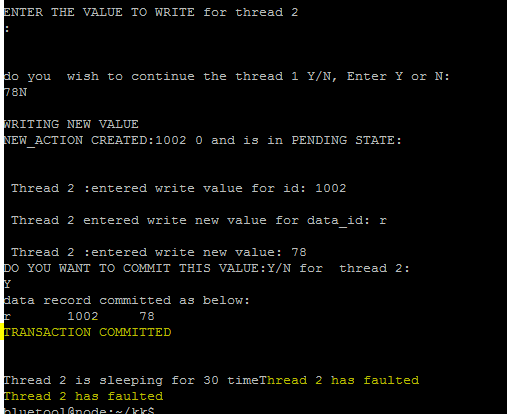
0. Faulty\_Write to a location in disc.

1. Read from that location on disc.

2. Detect (somehow) that the last write was not completed correctly

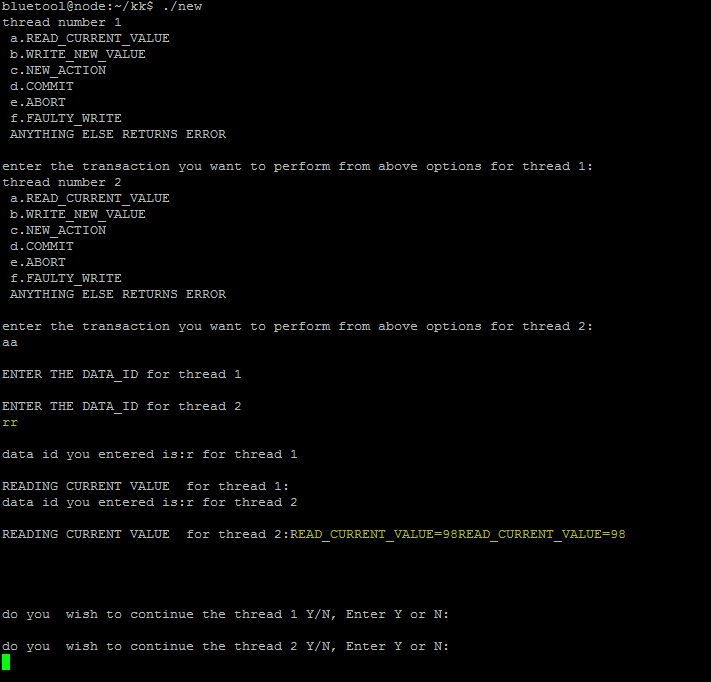
3. Determine what to do best depending on where the fail occurred.

4. Return either the Old correctly committed value or the new committed value based on step 3.

After writing value of 78 the program has crashed after Committing the value. To ensure all-or-nothing atomicity the value of either 78 or previous value of 98 is to be returned for the data\_id.

We wil try reading the value of after restarting the program:



Expected:

1. Good value is returned and not scrambled information from the faulty\_write.

Outcome:

PASS/FAIL.

PASS

ID: Test Case [4]

Purpose: Error-Prone environment with multi-threads

Prerequisite:

1. Read, Write, Commit Procedures have been implemented.

2. Multi-thread support

3. Error Prone files

Thread 1- proper write of a value 78

Thread 2- Faulty write of value 108

**Steps:**

1. Write to a location for a data id

2. Faulty write to the same location using same data id and different thread for a different value

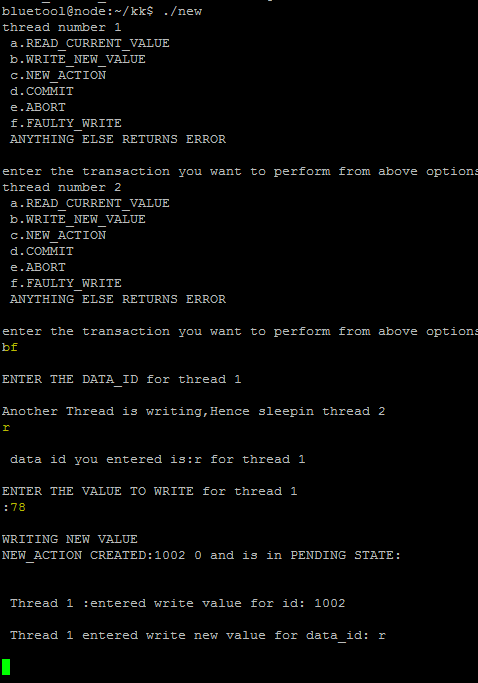
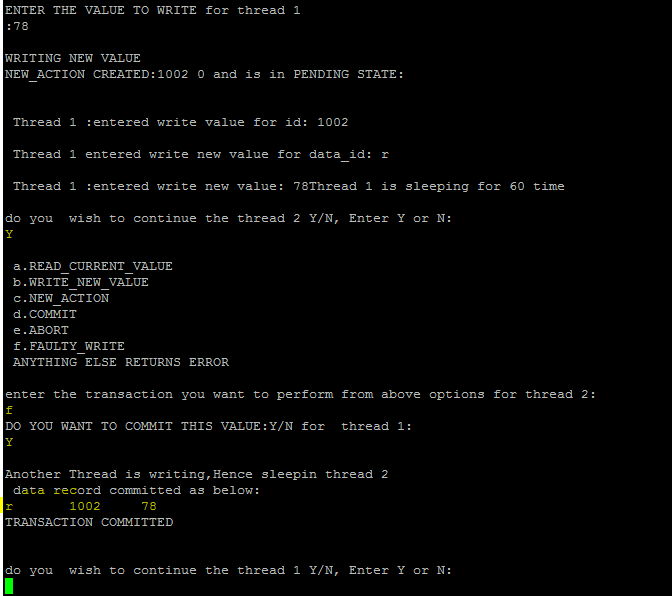
3. Commit thread1 - trying to write 78 to data id r

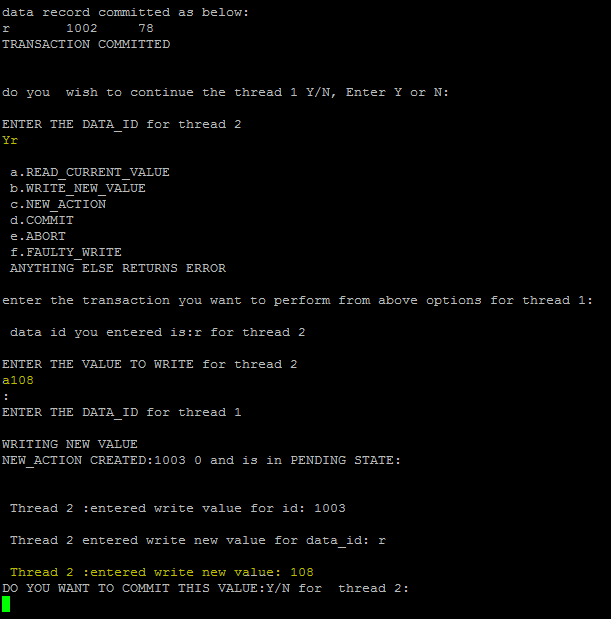
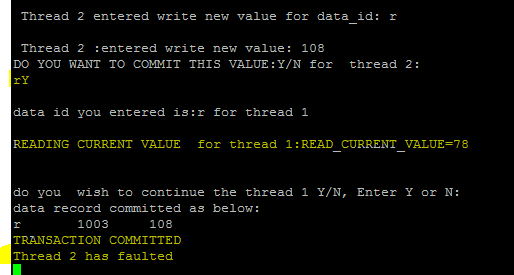
4. Commit thread2 - trying to write 108 to data id r

5. Read the value of data id

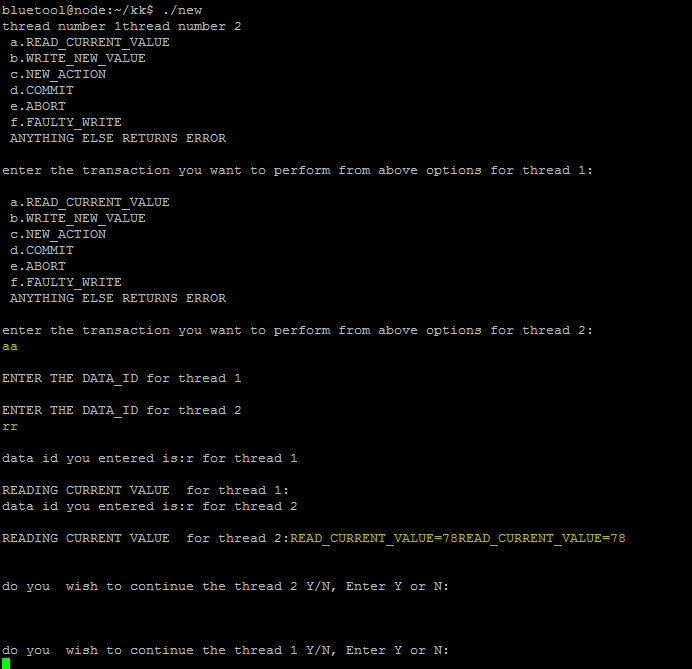
Output:

The o/p should give the last committed value by thread 1 i.e. 78

Thread 1 was able to read the value of last committ i.e 78. Post that the program has crashed.

Now we will try reading the value of r after restarting 

Outcome:

PASS/FAIL.

PASS

Output Screenshots:

Output Screenshots are provided in above test cases.

**Procedure for Installing/Running my application:**

1. Copy all the files in the ZIP file to a location on Linux box
2. Compile the header files separately using gcc <filename>
3. Ensure that the files journal\_log.txt and outcome\_record.txt are created and saved in case they are not available

Compile the whole program using the following command gcc -pthread -o output multi\_thread\_mutex.c

1. Ignore the warnings (as it’s a c type casting warnings and doesn’t impact the execution
2. Once all set please type ./output and try test cases

**Note:** My program is dynamic in nature and is heavily dependent on lock acquisition by mutex, hence the test cases may need to tried a couple of times to ensure proper functioning. I would please request you to test this program manually as I have not implemented static variable allocation to threads.

**Summary:**

* An in depth understanding of JFS File System has been understood
* The Procedures for READ, WRITE, and FAULTY WRITE have been successfully implemented and tested. The program was able to avoid race conditions for variables and also deadlocks were prevented using mutex locking.
* Recovery for program crash has been implemented and the implementation is able to handle one crash which is induced by external event (ctrl+c) and also by thread cancellation.
* The mutex locking mechanism was tried with multiple sleep times between two threads and optimal values have been included in the program.
* Allocation and de-allocation has been deprecated so that all the memory is assigned at the start of the program which allows an easy implementation and fast read and write responses.
* Data ids are taken as alphabets only as it allows us to use their ASCII values while storing and retrieval which would have its advantages as no separate arrays are to be maintained for data ids.
* Log recovery will overwrite the existing entries and rewrite them and hence only one crash can be handled at a time, however this can be extended to handle multiple crashes with more time investing and code.
* Cell Storage Working has been understood

**Improvements and Future Work:**

* Multiple crash handling can be effectively implemented using multiple log files for persistence
* Thread values can be passed in static mode instead of dynamic for more control and clarity in operation
* A WAL(Write Ahead Log) may be implemented to include each and every operation performed like COMMIT,ABORT etc. and to include all the states like PENDING COMMITTED and perform more sophisticated recovery.
* Dynamic memory allocation using Linked Lists and Heap may be implemented to improve scalability

**Conclusion & lessons learned:**

* The Challenges in handling race conditions for resources while developing an operating system or File System is understood
* Deadlock prevention techniques and the effective use of sleep for preventing dead lock and live locks is understood
* Recovery methods in case of system crash are looked at and the importance of synchronization between volatile and non-volatile storage systems is understood.
* Recovery using Log is implemented which gave an insight into how logs are used in case of crash.
* Various types of thread functions and the subtle nuances in parallel programming are understood

**Deliverables:**

Code, Report, Output

**References:**

[www.stackoverflow.com](http://www.stackoverflow.com)

[www.cplusplus.com](http://www.cplusplus.com)