Emre Uludağ

Arda Arslan

COMP304 Project 3 Report

Part 1)

In the first part of the project we are asked to implement two page replacement algorithms.They are FIFO page replacement method and Least Recent Usage page replacement method. To implement FIFO, we used a queue. Since, there is no built in implementation of queue in c we implemented a queue. And we used that queue for keeping track of page references.

**We implemented a few methods for using a queue:**

Queue:

PageQueue \*initQueue(int numberOfFrames);

void enqueue(PageQueue \*queue, int data);

int dequeue(PageQueue \*queue);

int get\_least\_recently\_used\_element();

**For using least recent usage page replacement method:**

We keep track of elements in an array. Each index represents the element and the value of that index is the usage of that element. The less is the value, less recent that element is used.

**int recentUsages[FRAMES];**

**int get\_least\_recently\_used\_element()**

**{**

**int leastRecentUsedElementIndex = 0;**

**int leastRecent = 10000000;**

**int i;**

**for (i = 0; i < FRAMES; i++)**

**{**

**if (recentUsages[i] < leastRecent)**

**{**

**leastRecent = recentUsages[i];**

**leastRecentUsedElementIndex = i;**

**}**

**}**

**return leastRecentUsedElementIndex;**

}

In the case of a page fault, we need to decide if we must engage a page replacement or not. To decide that we must run a if statement:

**if (free\_page < FRAMES){**

**physical\_page = free\_page;**

**free\_page++;**

**else{**

**if (rp == 0)**

**{ // FIFO**

**physical\_page = dequeue(queue);**

**}**

**else**

**{ // LRU**

**physical\_page = get\_least\_recently\_used\_element();**

**}**

**}**

Also by default, we need to do the following operations in order to use FIFO or LRU page replacement methods in the case of a page fault.

**if (rp == 0)**

**{ // FIFO**

**enqueue(queue, physical\_page);**

**}**

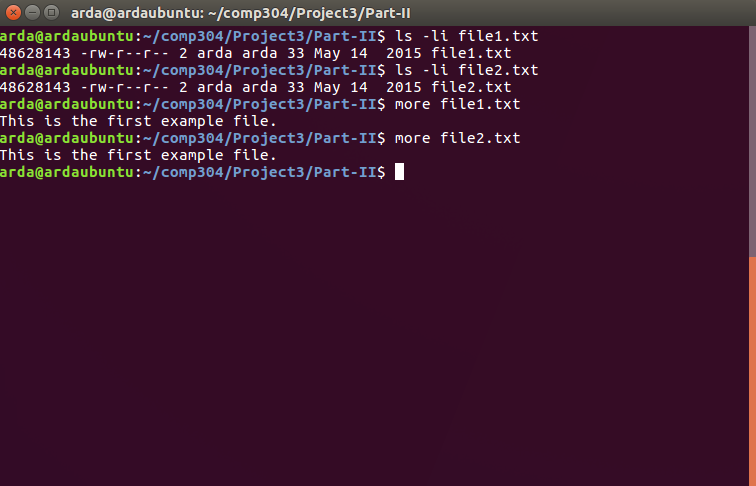
**else**

**{ // LRU**

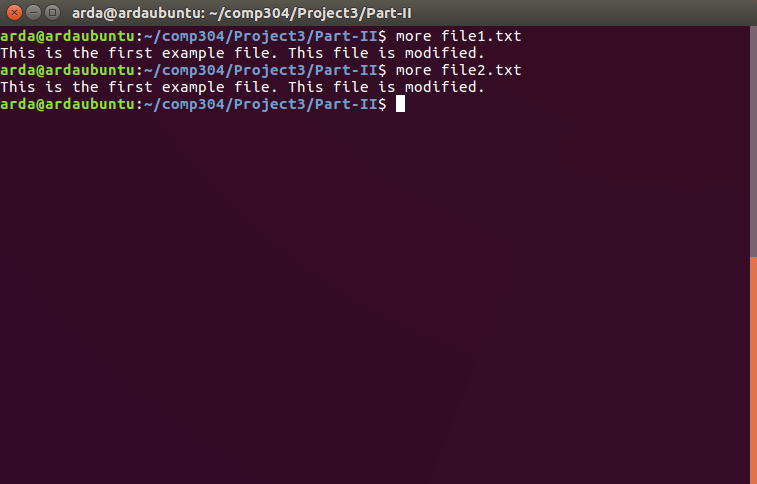
**recentUsages[physical\_page] = time\_count;**

**}**

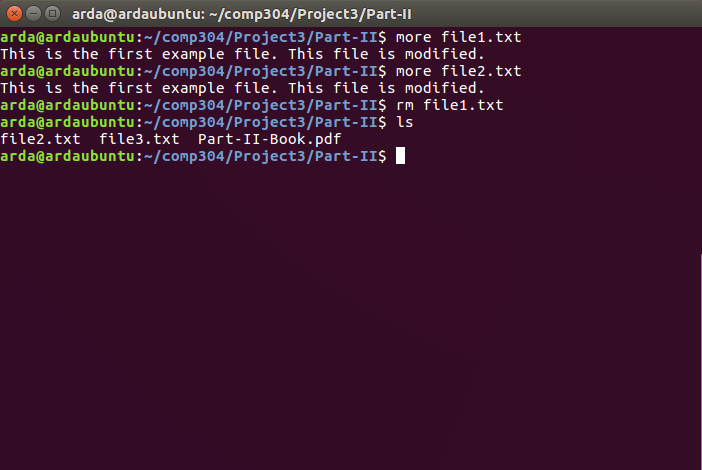
Part 2) Inode value of file1.txt is 48628143 and inode value of file2.txt is 48628143. So they are the same. Their contents are also the same.



After modifying file2.txt, the contents of file1.txt and file2.txt are as following:

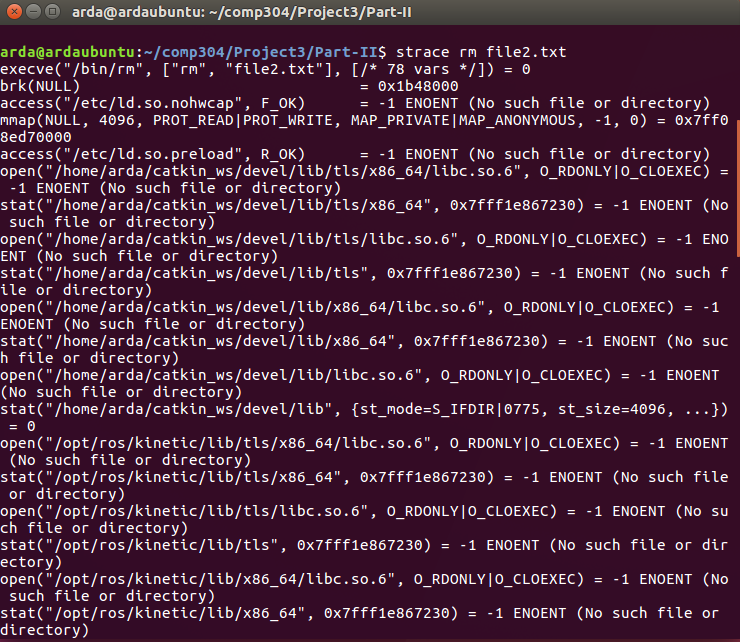
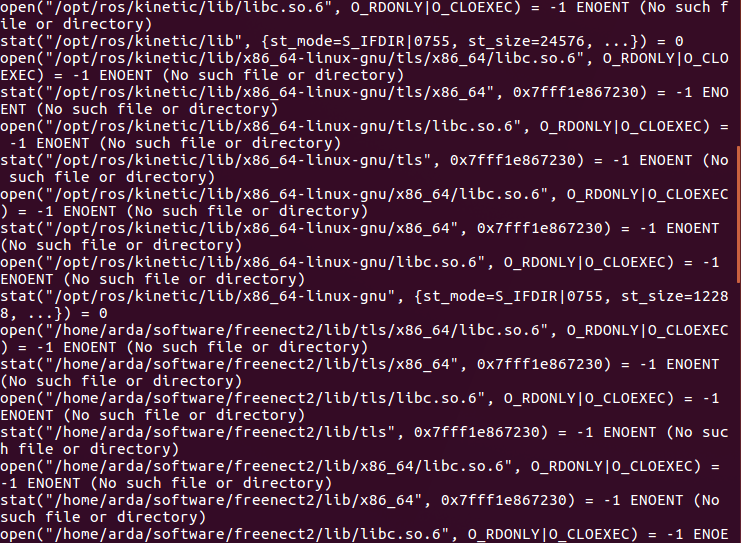


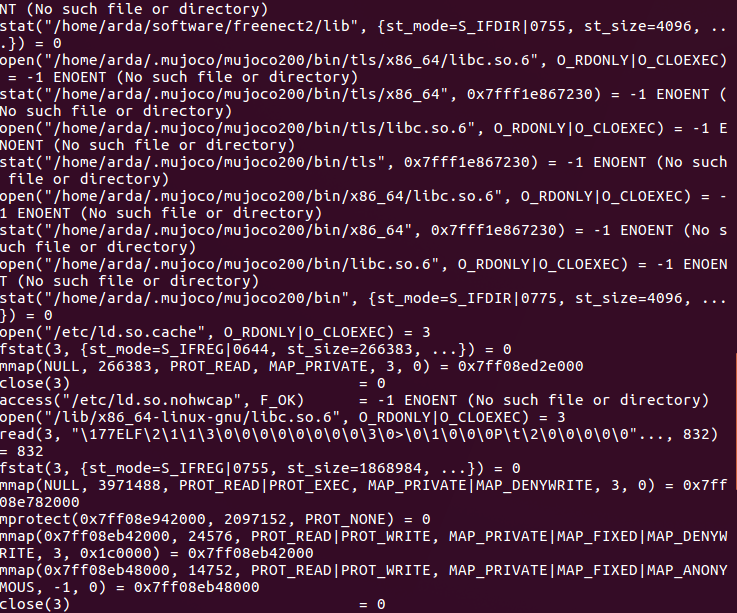
As it can be seen from the screenshot above, after modifying the content of file2.txt, the content of file1.txt was also changed.



As it can be seen from the screenshot above, after deleting file1.txt, file2.txt still exists.

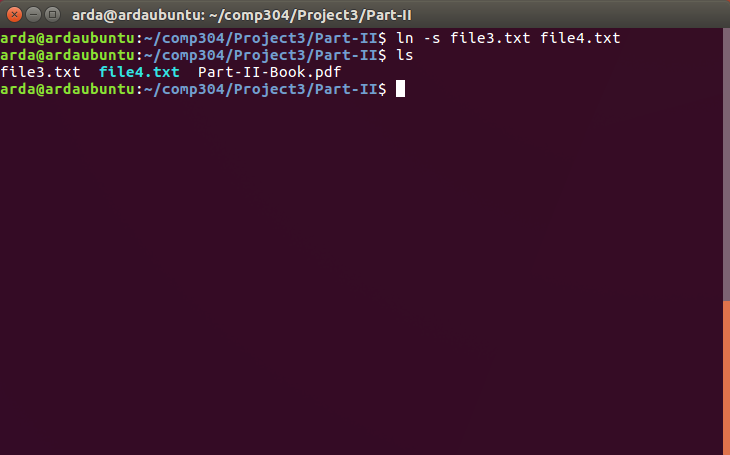
After typing “strace rm file2.txt” the output is as following:

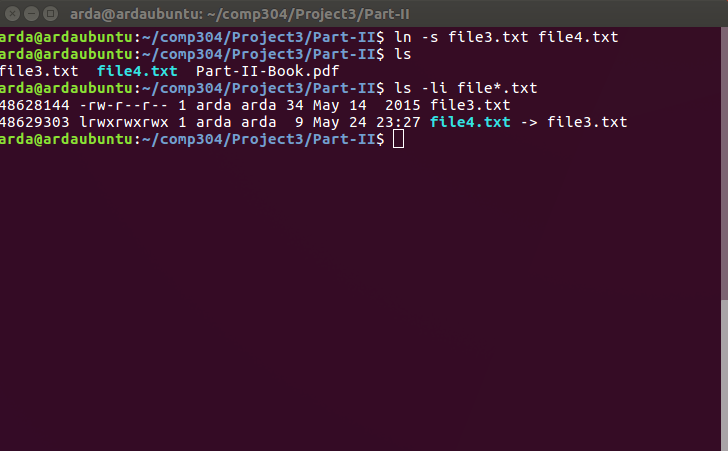


According to this trace, unlinkat(AT\_FDCWD, "file2.txt", 0) system call was used while removing file2.txt.

When we type “ln -s file3.txt file4.txt”, the output is as following:

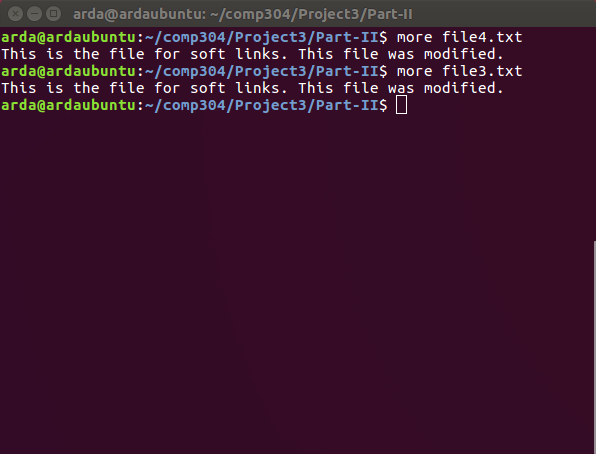


When we type “ls -li file\*.txt”, the output is as following:



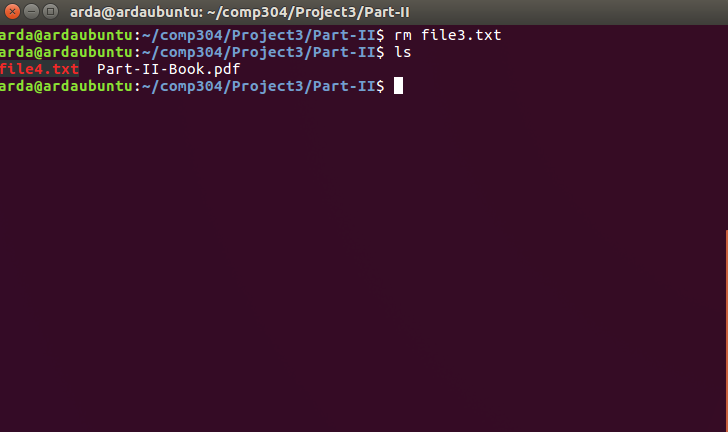
As it can be seen from the screenshot above, inode value of file3.txt is 48628144 and inode value of file4.txt 48628144. So they are different from each other.

After modifying the content of file4.txt, the contents are as following:



As it can be seen from the screenshot above, both contents were modified.

After deleting file3.txt, the directory is as follows:



After deleting file3.txt, we tried to modify file4.txt, however it was a read-only file. We tried to give it “write” permission by “chmod +w file4.txt”, however we got “chmod: cannot operate on dangling symlink 'file4.txt'”. So it could not be modifed.