CS-6210-A -Advanced Operating Systems

Project 4

Nidish Nair | GT ID# 903055730 **Akshat Harit | GT ID# 903090915**

How to Compile and Run

- make clean - To clean the project folder and restore it to an uncompiled state
- Ensure that all test case files are in the folder “testcases”
- make - To compile and build all test cases
- make <test_file.c> - To compile a particular <test_file.c> ,for example, basic.c, with the rvm library and generate the executable file of the same name, namely <test_file>
- ./<test_file> - To run the above compiled test case code.

Implementation

Description

All the requested library functions have been implemented in C++, in the file rvm.cpp. There are 2 kinds of files that are created - log files and backing store files.

The general flow of our implementation is thus -

1) The RVM is first initialized and the backing store directory is created.
2) A segment map request is made. Here, there are 2 cases :
   a) If the segment doesn’t exist on disk, we simply create a new backing segment of the given size and also create an in memory segment.
   b) If the segment exists on the disk, then after creating and in-memory mapping of that segment, we first load the data from it into memory and then also play through the logs and apply any relevant changes to that segment in memory.

   We feel that this is a good design decision because writing out the log entries to the backing store on disk at every mapping seemed less efficient.
3) A transaction is started which registers a unique transaction id for each given segment.
4) An about_to_modify declaration will basically store 3 things in a linked list – the offset, the size and the current data (as backup). In case of an abort or program failure, we simply traverse the linked list and at each node, restore the backed up data at the appropriate offset.
5) The transaction is committed, which basically writes out to the log file all the changes stored in the linked list mentioned in the previous point. That list is then cleared since we now have it committed to disk. This is also the point where we check if the log file is too big and truncate it if necessary.
6) The segments can now be unmapped (removed from memory) and then destroyed (which deletes the backing store of that segment).

It must be noted that there can be a case where after destroying the backing store of the segment, there may be some entries of that segment in the log. One fix would be to read the whole log and remove those entries but that will be very inefficient. Instead, we choose to let them lie in the log, because the design of the rest of our system is such that whenever the log is being read (during rvm_truncate_log and while mapping a segment), and if encounter an entry for a segment that doesn't exist, then we simply skip that entry and move on. In short, orphaned segment log entries are taken care of automatically by our design.

Log file

The log file forms the foundation of this library, because it stores the record of all the committed transactions. We have designed our log file to have a structure where every unique record is spread across multiple lines between the keywords "START" and "END". One sample record is -

```
START
testseg
0
100
hello, world
END
```

Here the 2nd line is the segment name, the 3rd line is the offset, the 4th line is the size and the 5th line to just before END is the actual data written.

We perform a periodic trimming of our log file by calling rvm_truncate_log whenever the log file size exceeds a certain threshold value. That value can be tuned.

Backing Store

The backing store is a directory containing .seg files. These files are the backing segments on disk and they contain the byte representation of the segment data. If this file is opened with a hex editor, one can verify that the actual data in byte form is written at the appropriate offsets.
For example, in our custom test case basic_int, after writing integers 1,2,3 and hello in the segment, the log file contents are as follows:

53 54 41 52 54 0a 74 65 73 74 73 65 67 0a 30 0a 32 32 0a **01** 00 00 00 **02** 00 00 00 **03** 00 00 00
48 65 6c 6c 6f 00 00 00 00 00 0a 45 4e 44 0a

which corresponds to the log file looking like this in a normal text editor -

```
------------------------------------LOG ENTRY STARTS------------------------------------------------------------
START
testseg
0
22
Hello
END
------------------------------------LOG ENTRY ENDS-------------------------------------------------------------
```

A hex editor will basically show that the proper data has been written in the log entry – 01 for 1, 02, for 2, 03 for 3 and **48 65 6c 6c 6f** for hello

API function descriptions

rvm_t rvm_init(const char* directory)

A backing directory is created with the given name and the log name is initialized.

void* rvm_map(rvm_t rvm, const char* seg_name, int size_to_create)

Here we first create check if the backing store file exists and create it if it doesn't. If a file exists but is of smaller size, it is enlarged with a call to ftruncate(). It then loads data from the backing segment file into memory and then applies the log file entries to the in-memory segment.

void rvm_unmap(rvm_t rvm, void* seg_base);

This function removes the segment from the segment list vector, thereby, effectively unmapping it from memory.

void rvm_destroy(rvm_t rvm, const char* seg_name);

This function removes the backing store file of the segment only if it is unmapped. otherwise it prints an error message and does nothing.

trans_t rvm_begin_trans(rvm_t rvm, int num_segs, void seg_bases);**

It checks if the segment doesn't exist already in the segment list. If not, it assigns it a unique transaction id.

void rvm_about_to_modify(trans_t tid, void* seg_base, int offset, int size);

This function stores the old values of the segment to an internal in memory vector. This is done in case of a call to rvm_abort_trans, we can return the data segment to its previous state

void rvm_commit_trans(trans_t tid);

Here, the in memory segments are written to the log file using the offset list vector.

void rvm_abort_trans(trans_t tid);

This function undos the changes made to the segment. The function iterates over the segment list and identifies the correct segment to undo based on transaction id. It then copies the data from the old value stored in the rvm_begin_trans function to the current data segment at the correct offset. Then it clears the offset list that contains the partial values modified in that transaction and terminates.

void rvm_truncate_log(rvm_t rvm);

This function reads the log file and writes the corresponding data into the backing store file. The data

Important Library Helper functions

save_seg_file(segment_node seg, FILE* file)

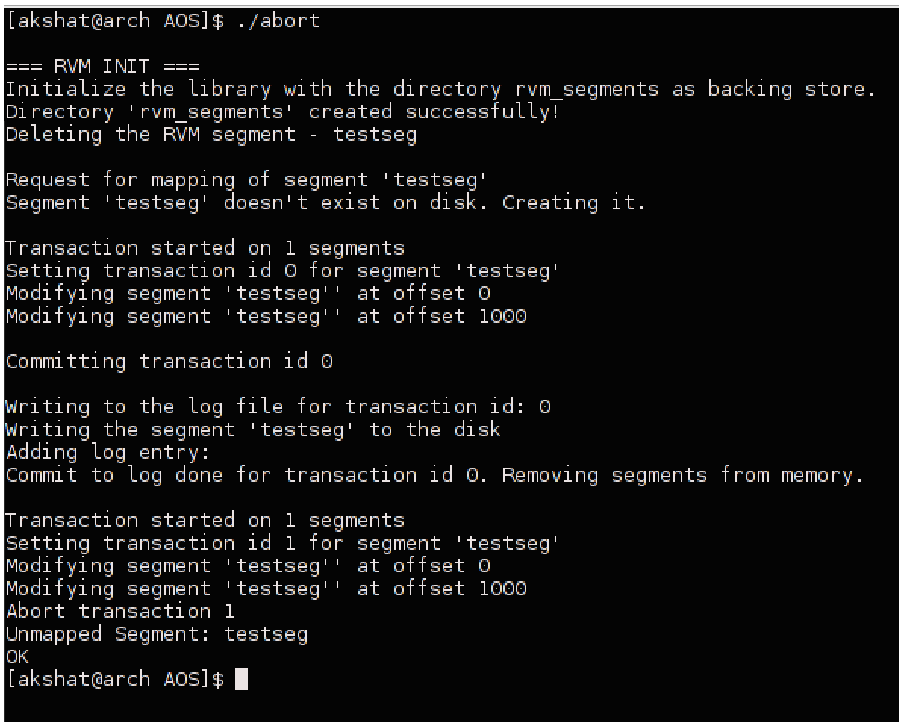
This function takes an in-memory segment and writes out log entries for all its transactions into the given file pointer (a log file). These series of transactions (like an in-memory log) are the result of data writes that happened after each call to about_to_modify. The format of each log entry has been mentioned in the beginning of this report.

read_seg_log(const char* seg_name, segment_t* seg)

This important function reads the log file and plays through its entries to find the entries pertaining to the input segment, and then applies those changes to the in-memory segment. This function is only called after the backing store of the segment has already been loaded from disk.

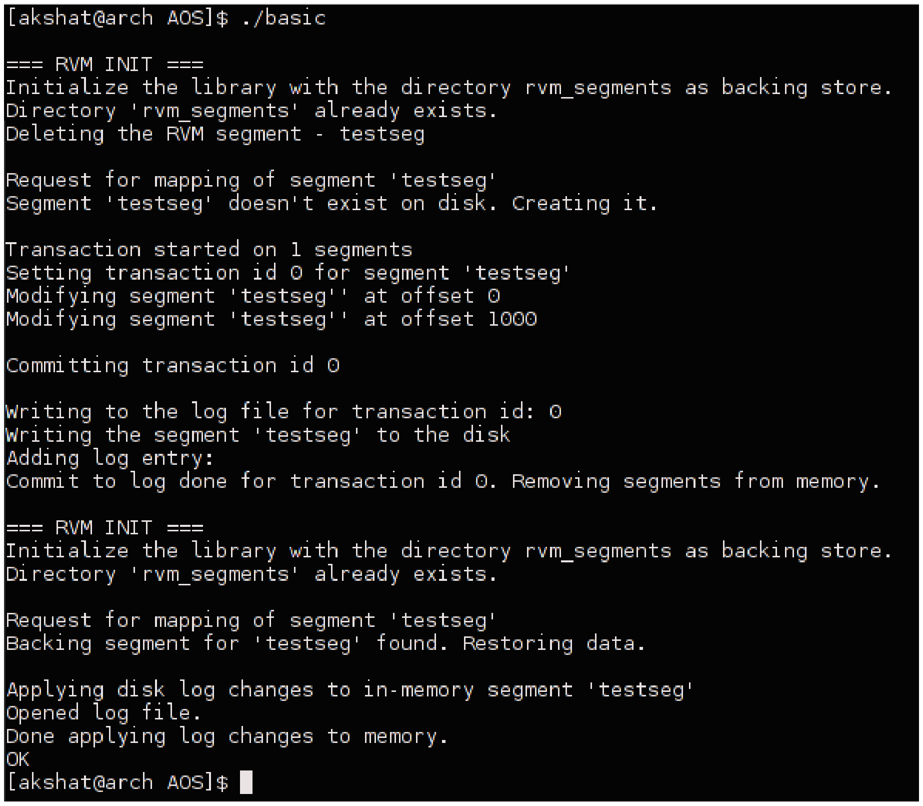
Provided Test Case Output

Abort



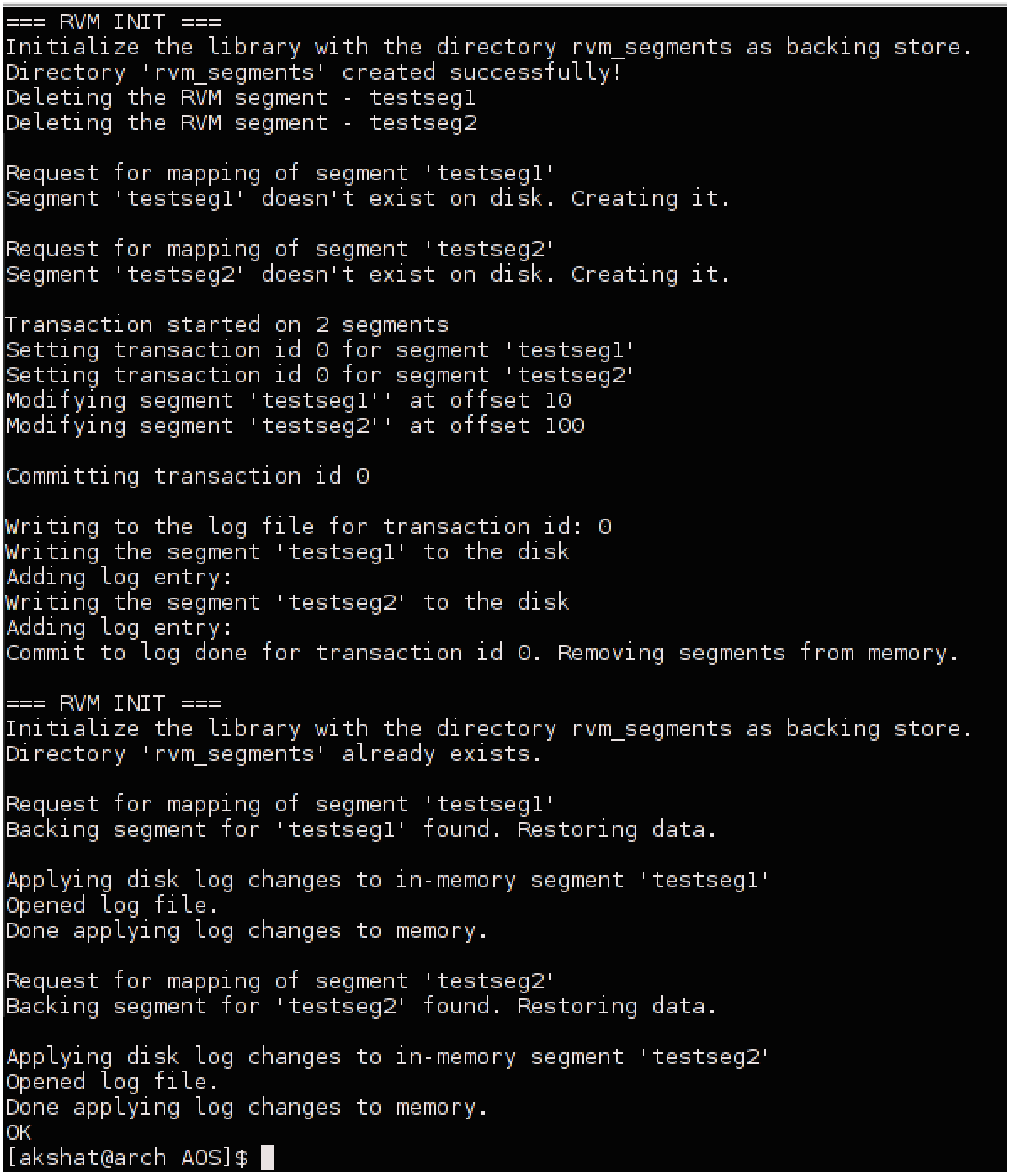
```
[akshat@arch AOS]$ ./abort

=== RVM INIT ===
Initialize the library with the directory rvm_segments as backing store.
Directory 'rvm_segments' created successfully!
Deleting the RVM segment - testseg

Request for mapping of segment 'testseg'
Segment 'testseg' doesn't exist on disk. Creating it.

Transaction started on 1 segments
Setting transaction id 0 for segment 'testseg'
Modifying segment 'testseg'' at offset 0
Modifying segment 'testseg'' at offset 1000

Committing transaction id 0

Writing to the log file for transaction id: 0
Writing the segment 'testseg' to the disk
Adding log entry:
Commit to log done for transaction id 0. Removing segments from memory.

Transaction started on 1 segments
Setting transaction id 1 for segment 'testseg'
Modifying segment 'testseg'' at offset 0
Modifying segment 'testseg'' at offset 1000
Abort transaction 1
Unmapped Segment: testseg
OK
[akshat@arch AOS]$
```

Basic



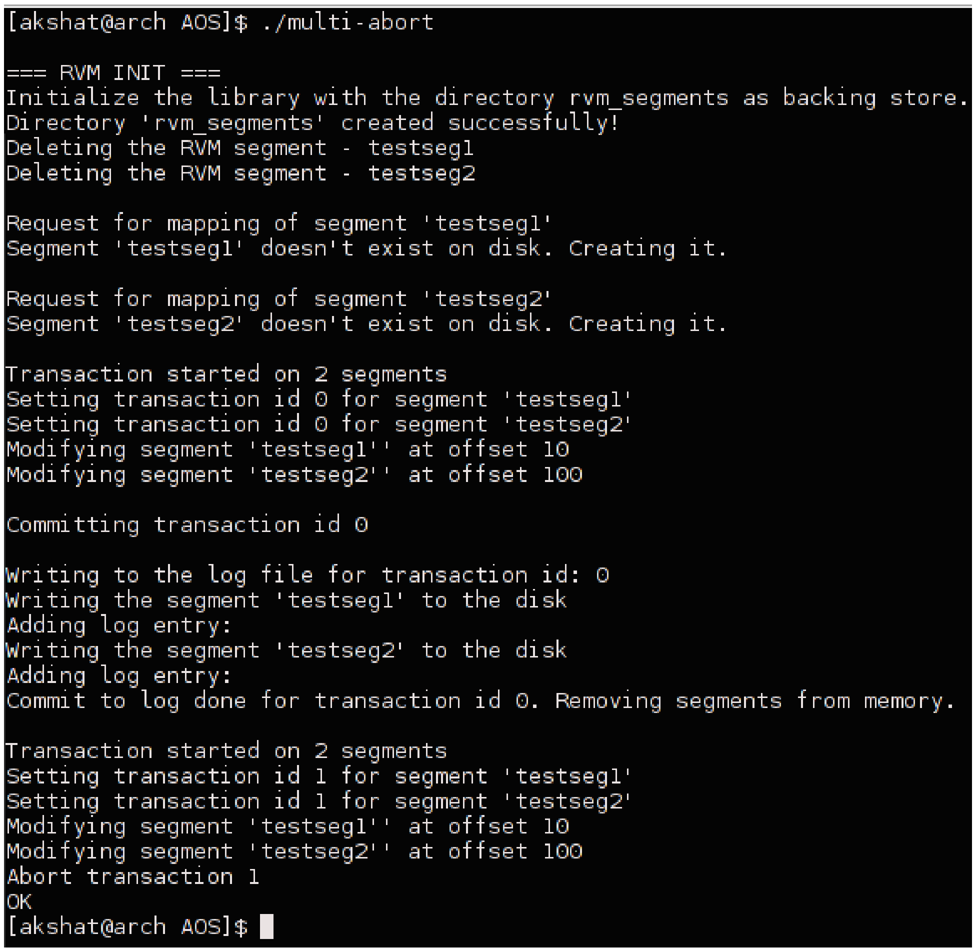
```
[akshat@arch AOS]$ ./basic

=== RVM INIT ===
Initialize the library with the directory rvm_segments as backing store.
Directory 'rvm_segments' already exists.
Deleting the RVM segment - testseg

Request for mapping of segment 'testseg'
Segment 'testseg' doesn't exist on disk. Creating it.

Transaction started on 1 segments
Setting transaction id 0 for segment 'testseg'
Modifying segment 'testseg'' at offset 0
Modifying segment 'testseg'' at offset 1000

Committing transaction id 0

Writing to the log file for transaction id: 0
Writing the segment 'testseg' to the disk
Adding log entry:
Commit to log done for transaction id 0. Removing segments from memory.

=== RVM INIT ===
Initialize the library with the directory rvm_segments as backing store.
Directory 'rvm_segments' already exists.

Request for mapping of segment 'testseg'
Backing segment for 'testseg' found. Restoring data.

Applying disk log changes to in-memory segment 'testseg'
Opened log file.
Done applying log changes to memory.
OK
[akshat@arch AOS]$
```

Multi



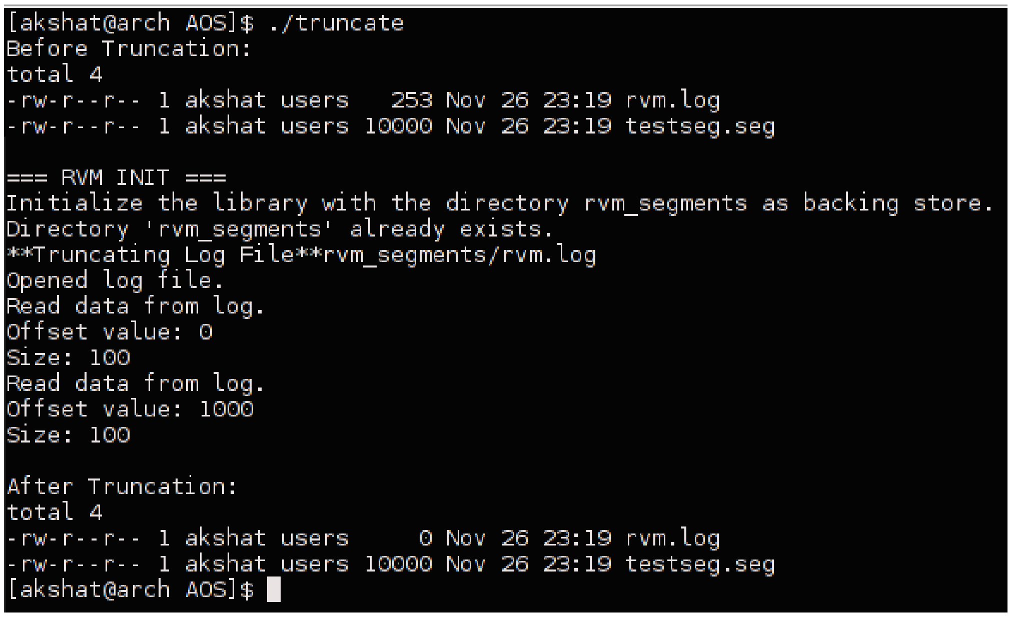
```
=== RVM INIT ===
Initialize the library with the directory rvm_segments as backing store.
Directory 'rvm_segments' created successfully!
Deleting the RVM segment - testseg1
Deleting the RVM segment - testseg2

Request for mapping of segment 'testseg1'
Segment 'testseg1' doesn't exist on disk. Creating it.

Request for mapping of segment 'testseg2'
Segment 'testseg2' doesn't exist on disk. Creating it.

Transaction started on 2 segments
Setting transaction id 0 for segment 'testseg1'
Setting transaction id 0 for segment 'testseg2'
Modifying segment 'testseg1'' at offset 10
Modifying segment 'testseg2'' at offset 100

Committing transaction id 0

Writing to the log file for transaction id: 0
Writing the segment 'testseg1' to the disk
Adding log entry:
Writing the segment 'testseg2' to the disk
Adding log entry:
Commit to log done for transaction id 0. Removing segments from memory.

=== RVM INIT ===
Initialize the library with the directory rvm_segments as backing store.
Directory 'rvm_segments' already exists.

Request for mapping of segment 'testseg1'
Backing segment for 'testseg1' found. Restoring data.

Applying disk log changes to in-memory segment 'testseg1'
Opened log file.
Done applying log changes to memory.

Request for mapping of segment 'testseg2'
Backing segment for 'testseg2' found. Restoring data.

Applying disk log changes to in-memory segment 'testseg2'
Opened log file.
Done applying log changes to memory.
OK
[akshat@arch AOS]$ 
```

Muti_abort



```
[akshat@arch AOS]$ ./multi-abort

=== RVM INIT ===
Initialize the library with the directory rvm_segments as backing store.
Directory 'rvm_segments' created successfully!
Deleting the RVM segment - testseg1
Deleting the RVM segment - testseg2

Request for mapping of segment 'testseg1'
Segment 'testseg1' doesn't exist on disk. Creating it.

Request for mapping of segment 'testseg2'
Segment 'testseg2' doesn't exist on disk. Creating it.

Transaction started on 2 segments
Setting transaction id 0 for segment 'testseg1'
Setting transaction id 0 for segment 'testseg2'
Modifying segment 'testseg1'' at offset 10
Modifying segment 'testseg2'' at offset 100

Committing transaction id 0

Writing to the log file for transaction id: 0
Writing the segment 'testseg1' to the disk
Adding log entry:
Writing the segment 'testseg2' to the disk
Adding log entry:
Commit to log done for transaction id 0. Removing segments from memory.

Transaction started on 2 segments
Setting transaction id 1 for segment 'testseg1'
Setting transaction id 1 for segment 'testseg2'
Modifying segment 'testseg1'' at offset 10
Modifying segment 'testseg2'' at offset 100
Abort transaction 1
OK
[akshat@arch AOS]$
```

Truncate



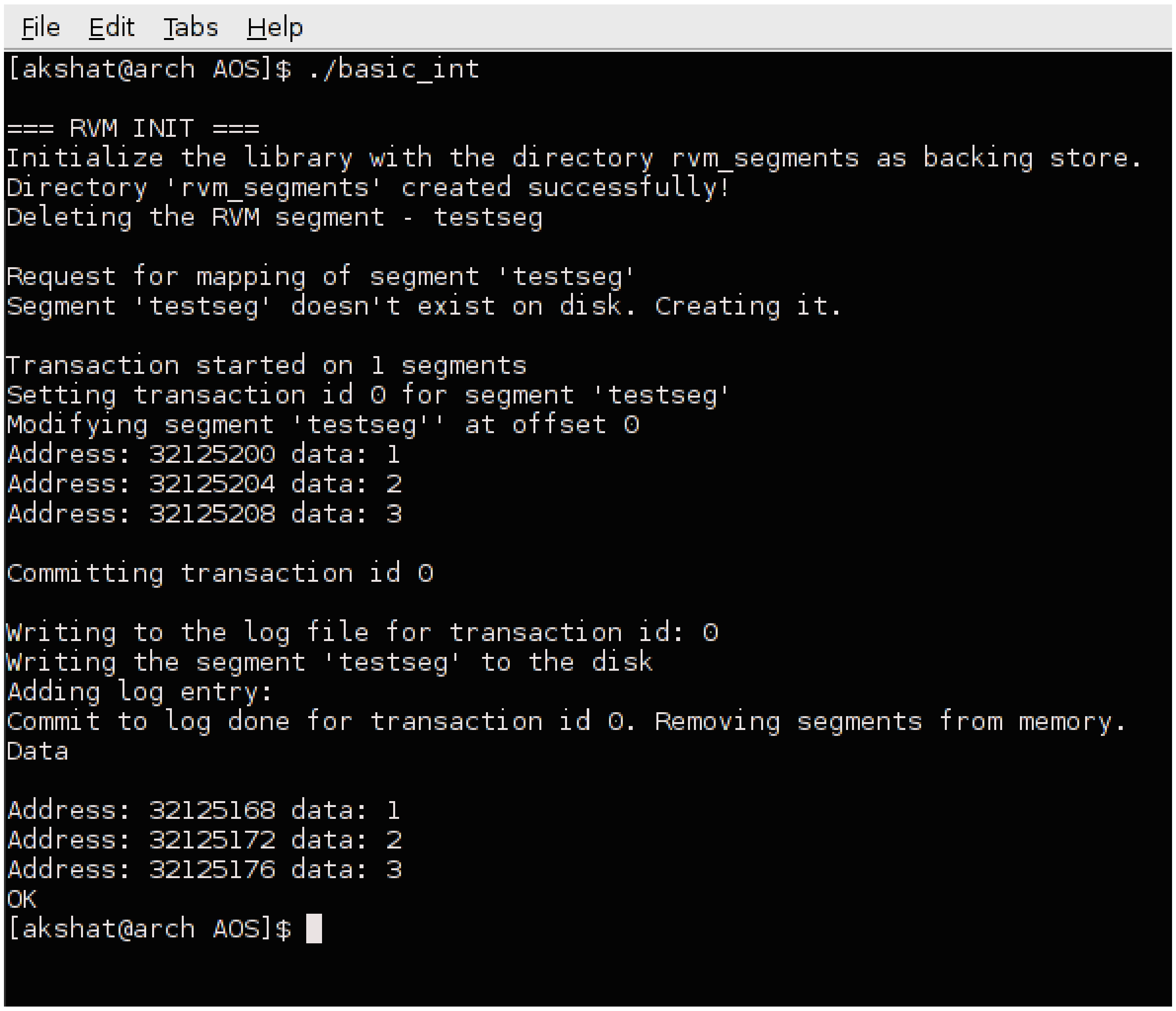
```
[akshat@arch AOS]$ ./truncate
Before Truncation:
total 4
-rw-r--r-- 1 akshat users   253 Nov 26 23:19 rvm.log
-rw-r--r-- 1 akshat users 10000 Nov 26 23:19 testseg.seg

=== RVM INIT ===
Initialize the library with the directory rvm_segments as backing store.
Directory 'rvm_segments' already exists.
**Truncating Log File**rvm_segments/rvm.log
Opened log file.
Read data from log.
Offset value: 0
Size: 100
Read data from log.
Offset value: 1000
Size: 100

After Truncation:
total 4
-rw-r--r-- 1 akshat users     0 Nov 26 23:19 rvm.log
-rw-r--r-- 1 akshat users 10000 Nov 26 23:19 testseg.seg
[akshat@arch AOS]$
```

Custom Test Cases

The following are the custom test cases we wrote for testing our library -

basic_int

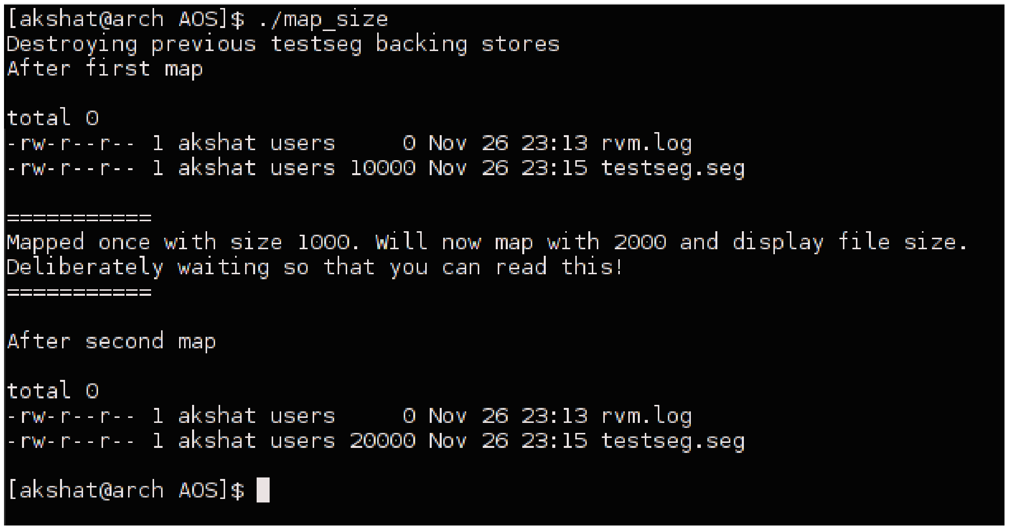
This function tests that our backing store works for binary formats as well as text data. To do this, in the transaction, we put the values of an integer array into the segment. The first function proc1() does this, commits the transaction and exits. After this function proc2() runs, which maps the same segment again. The data from the previous log is written to the in memory segment from the disk, and verified with the array.



```
[akshat@arch AOS]$ ./basic_int

=== RVM INIT ===
Initialize the library with the directory rvm_segments as backing store.
Directory 'rvm_segments' created successfully!
Deleting the RVM segment - testseg

Request for mapping of segment 'testseg'
Segment 'testseg' doesn't exist on disk. Creating it.

Transaction started on 1 segments
Setting transaction id 0 for segment 'testseg'
Modifying segment 'testseg'' at offset 0
Address: 32125200 data: 1
Address: 32125204 data: 2
Address: 32125208 data: 3

Committing transaction id 0

Writing to the log file for transaction id: 0
Writing the segment 'testseg' to the disk
Adding log entry:
Commit to log done for transaction id 0. Removing segments from memory.
Data

Address: 32125168 data: 1
Address: 32125172 data: 2
Address: 32125176 data: 3
OK
[akshat@arch AOS]$
```

map_size

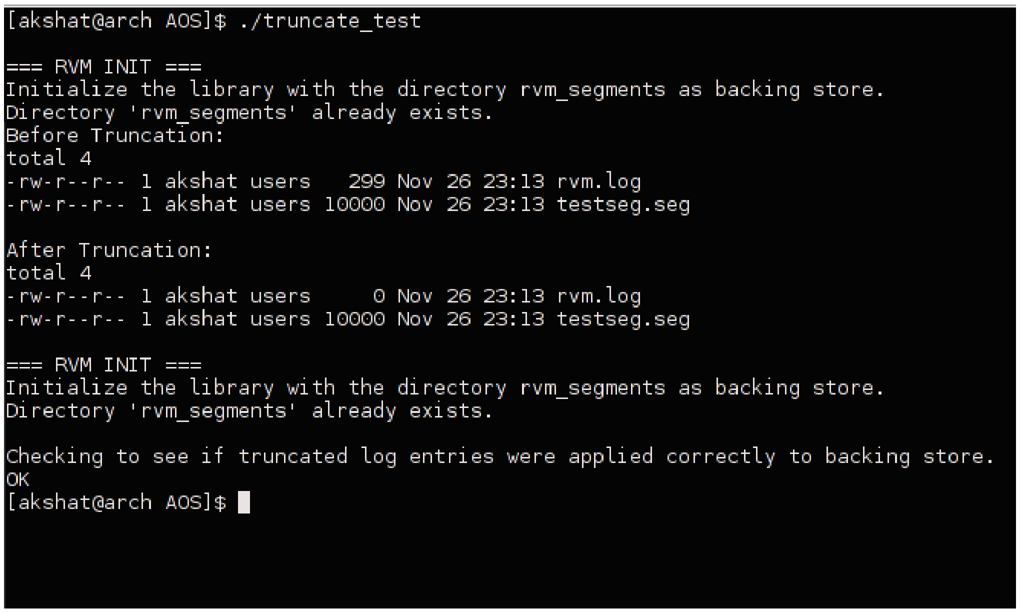
This test is to test the functionality of map function expanding the size of the backing storage depending on the input parameter of size. This maps a segment to size 1000 and then unmaps it and then maps it to 2000 size.It displays the output of ls command on the directory to demonstrate the size.



```
[akshat@arch AOS]$ ./map_size
Destroying previous testseg backing stores
After first map

total 0
-rw-r--r-- 1 akshat users     0 Nov 26 23:13 rvm.log
-rw-r--r-- 1 akshat users 10000 Nov 26 23:15 testseg.seg

===========
Mapped once with size 1000. Will now map with 2000 and display file size.
Deliberately waiting so that you can read this!
===========

After second map

total 0
-rw-r--r-- 1 akshat users     0 Nov 26 23:13 rvm.log
-rw-r--r-- 1 akshat users 20000 Nov 26 23:15 testseg.seg

[akshat@arch AOS]$
```

truncate_test

This test basically verifies if rvm_truncate_log works properly, i.e. check to see if log enries are applied properly. Here, one thread function maps a segment and then writes 2 strings to it and commits the changes. Then rvm_truncate_log is called which would ideally process those entries and apply them to the backing store. After that in another thread function, the same segment is mapped to memory. Since this involves the loading of the backing store to memory, we should now have our 2 strings in memory at the appropriate offsets. There are 2 “if conditions” that verify the same.



```
[akshat@arch AOS]$ ./truncate_test

=== RVM INIT ===
Initialize the library with the directory rvm_segments as backing store.
Directory 'rvm_segments' already exists.
Before Truncation:
total 4
-rw-r--r-- 1 akshat users   299 Nov 26 23:13 rvm.log
-rw-r--r-- 1 akshat users 10000 Nov 26 23:13 testseg.seg

After Truncation:
total 4
-rw-r--r-- 1 akshat users     0 Nov 26 23:13 rvm.log
-rw-r--r-- 1 akshat users 10000 Nov 26 23:13 testseg.seg

=== RVM INIT ===
Initialize the library with the directory rvm_segments as backing store.
Directory 'rvm_segments' already exists.

Checking to see if truncated log entries were applied correctly to backing store.
OK
[akshat@arch AOS]$
```

automatic_truncate

This test 100 transactions wherein a test string is written to different locations(0, 10, 20 ….) in the same segment. This functions tests whether automatic truncation occurs when the log file increases above a certain size set on LOG_LIMIT. This is currently set to 1024 bytes i.e 1 kB. We find that the rvm_truncate_log is called multiple times based on whenever the file size limit is exceeded.

Final Notes and Observations

The test case **abort.c** gives this compilation error -
testcases/abort.c:26:29: error: lvalue required as left operand of assignment

```
seg = (char *) segs[0] = (char *) rvm_map(rvm, "testseg", 10000);
```

We've made the following simple fix, which doesn't otherwise change the test file:

```
segs[0] = (char *) rvm_map(rvm, "testseg", 10000);
seg = (char *)segs[0];
```