**Other random notes:**

trans\_t is basically a transaction id, typecasted to int

segname must be unique.

Need a way to keep track of all segments, names, sizes

3) There is no result returned from rvm\_about\_to\_modify. What is the expected behavior on invalid parameters ? (such as offset + size > segment size)

=> We will not test such scenarios in the project.

4) Can a segment be mapped (open) during a truncation?

=> No. You may implement it, but we will not test such cases.

No, no two processes simultaneously map the same segname in the same directory.

Can have multiple threads in a process calling rvm\_init

"directory1" + "seg" and "directory2" + "seg" identify two different segments.

**rvm\_t rvm\_init(const char \*directory)**

Takes a directory name, which is the name of the backing store on local FS. If not present, create it.

**What should rvm\_t contain?**

To Implement:

check if directory exists. If yes, return created struct

If not, create, then return the struct.

Error cases: return -1

From piazza:

Since multiple rvm\_init calls are possible, can they point to the same directory? If so, can they simultaneously map the same segname? If so, are we responsible for coherency between the mappings? I hope there's a 'no' in that chain somewhere :)

=> Yes, it can point the same directory, so you are responsible for the coherency. However, regarding the concurrency, no two processes can simultaneously map the same segname.

**void \*rvm\_map(rvm\_t rvm, const char \*segname, int size\_to\_create)**

Creation and/or mapping of a segment from disk to memory

**Error condition to test for: if the segment already is mapped to memory, what do you return? do you check for size?**

4 cases:

1. segment of size size\_to\_Create already exists, so just map it to memory
2. segment exists, is shorter. extend it to size\_to\_create, then map
3. **segment exists, is longer…? What do you do? shorten it? keep it the same?**
4. segment does not exist. create it with specified size, then map

**void rvm\_unmap(rvm\_t rvm, void \*segbase)**

unmap it. obviously from MEMORY, but it should still exist on disk

1. If it is in memory, free it maybe?
2. **If it is not in memory, can’t actually do an unmap. Is it an error?**

**void rvm\_destroy(rvm\_t rvm, const char \*segname)**

erase it from backing store/disk. cannot be called if it is mapped, so check that condition **before** calling this on a segment

to implement:

before the call, check if it is mapped

if not mapped, call rvm\_destroy()

Delete the corresponding segname from global? segment data structure.

Do we actually have to free the disk space, or can we just mark it as empty using the data structure? Deletes file in disk

**trans\_t rvm\_begin\_trans(rvm\_t rvm, int numsegs, void \*\*segbases)**

contains array of seg ids to be modified. should return error (trans\_t-1) if one of the segments in the array is already being modified by another client.

So this means, there should be some way to know which segments are being modified at any particular instant in time. Some kind of shared data structure (to manage mappings) in the process, where all clients use it to log which segments are being touched.

numsegs is given since it is \*\*segbases.

To implement:

Check the shared data structure if the segments my client needs to modify are already being modified by another client.

If not, add my own segbases array to data structure.

If true, return trans\_t-1

**Why do you need rvm\_t here?**

**void rvm\_about\_to\_modify(trans\_t tid, void \*segbase, int offset, int size)**

after a transaction has started, specify which part of the segment is going to be modified. This happens before modification actually happens

validation: Should have already been specified in begin\_trans, for that specified trans\_t

“Your library needs to ensure that the old memory has been saved, in case an abort is executed. It is legal call rvm\_about\_to\_modify multiple times on the same memory area.”

“old memory needs to be saved”==write to undo log

**So what should undo log contain?**

Probably the address which is going to be modified, the length of data, the old data itself. The undo log should be specific to a tid.

If multiple calls to this are made, check for existence of the log. create if no log, append if it exists.

**void rvm\_commit\_trans(trans\_t tid)**

commit, meaning, copy from memory to disk to make it persistent. once this returns, that means even on restart, the changes must be applied already to the disk.

2 scenarios to make it persistent:

1. copy the memory directly to disk - this could fail, so not a good option
2. write the redo log onto disk-This only writes the changes themselves onto the log, so doesn’t update the segment on disk.

**So what should redo log contain? And when should the contents of the redo log be changed on the disk segment?**

If the redo log is applied before commit returns, that could fail, and commit would return false even if the redo log had been created successfully. It makes sense to keep the commit operation as atomic as possible. **Check what Brubaker said about when to actually commit.**

**The entire transaction succeeds or nothing.. So need a way to revert in case of a crash**

Contents of redo log:

about to modify only has info about addresses which had to be modified, not the new data. So for the redo log, go thru the undo log. Figure out which segments, addresses, and the length of the modifications made. Iterate through the undo log, read from memory, and write the new contents into the redo log. When all lines in undo log have been read, write the redo log onto disk.

**Is this write supposed to be an append operation? Or will there be multiple redo logs?**

Once the write to disk is successful, delete the undo log, since you won’t need to abort anymore. Then return success

If any part of this fails, return fail. **Do you need to undo, or try again? Just undo**

**void rvm\_abort\_trans(trans\_t tid)**

Undo changes with a specific tid. So this can only be called before commit right? from deepti’s readme, yes. because the commit function removes undo logs from memory. this is the last step of commit, after redo log is created.

To actually perform an abort, read the undo log FROM THE REVERSE for the specific tid. iterate through the undo log, and apply changes. Reverse because at the end, memory should be back to the state it was in before the transaction started.

if not undo log for a tid, just return. this will happen if not changes yet, or if tid has already been committed.

if tid doesnt exist, return with error message

**void rvm\_truncate\_log(rvm\_t rvm)**

shrink the log files-remove committed or aborted transactions.

Undo log doesn’t have to be cleaned up, because it will be deleted when committing a transaction.

for the redo log...

<https://piazza.com/class/idfwtpmjm1d4tx?cid=112>

"Now these redo logs **gets flushed** to actual disk data segments only in rvm\_truncate\_log()."

The redo logs are written in the disk when a user calls rvm\_commit\_trans(). When rvm\_trucate\_log() is called, the redo logs are truncated into the corresponding segments.

For the truncation, it's not required to call the truncation method in the rvm\_init, but you should make sure that you have the most recent updates (committed) whether they are in the logs or in the segments.