Project 3: Recoverable Virtual Memory

Team Members

Janardhan Haryadi Ramesh | 903271464
Rahul Jayakrishnan | 903281837

Introduction:

In this project we have implemented a recoverable virtual memory system like LRVM as described in Lightweight Recoverable Virtual Memory Paper. Users of the library can create persistent segments of memory and then access them in a sequence of transactions. Making the memory persistent is achieved by mirroring each segment of memory to a backing file on disk. If the client crashes, or if the client explicitly requests an abort, then the memory is returned to the state it was in before the transaction started.

Design Overview:

We have provided a set of APIs to map segments from disk into memory, make changes to the segment and subsequently commit or abort these changes. The library guarantees that all changes made to these segments are persisted on to the disk after a successful commit.
The changes during the transaction are recorded in a log file which will be processed when the segment is mapped the next time, or when truncate API is called explicitly. This ensures the user gets the most up to date committed data when a segment is mapped. Abort restores the segment to a state prior to the beginning of the transaction.Each log entry consists of two parts, metadata and payload. Metadata which acts as the header to the log entry houses information required to patch the bytes stored in payload section of the log file into the persistent data file.

API Implementation:

Initialization & Mapping APIs:

rvm_t rvm_init(const char *directory)

Initialize the library with the specified directory as backing store.

rvm_t is a structure that stores the the directory that the user wants to configure as the backing store. rvm_init takes the directory name as argument and creates the directory if it doesn't already exist.

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

rvm_map maps a segment for a size **size_to_create** from disk into virtual memory. We use mmap system call to map a file from disk to memory. If the segment doesn't exist, a segment is created by generating two segment specific files - log<segname>, data<segname>. log<segname> stores the committed changes that will be processed during

truncate phase, which will then be written to data<segname>. We sever the mapping by calling munmap once the local memory copy has been created. The copy is pushed into a vector of structure **local_store.**

This structure has elements to store metadata about the mapped segment like segment name, segment size, transaction ID, segbase, etc.

The API returns a pointer to the in-memory copy of the mapped segment. The user operates on this memory space.

The API returns error if an attempt is made to map an already mapped segment.

void rvm_unmap(rvm_t rvm, void *segbase)

Unmap a segment from memory. We clear the in memory copy of the mapped segments. All metadata stored that is associated with the segment is also cleared from memory. If the segment has to be accessed again, it must be remapped from disk using rvm_map.

void rvm_destroy(rvm_t rvm, const char *segname)

Destroys a segment completely, erasing its backing store. Returns error if a mapped segment is attempted to be destroyed.

Transactional Operations

trans_t rvm_begin_trans(rvm_t rvm, int numsegs, void **segbases)

Begin a transaction that will modify the segments listed in segbases. **trans_t** is an integer type which holds the transaction ID returned by **rvm_begin_trans**. **rvm_begin_trans** checks if all the segbases specified are mapped by cycling through the local_store vector and searching for all segbases passed as the argument. It then generates a transaction number by incrementing a library local counter. If any of the specified segments is not mapped or is already being modified by a transaction, then the call should fail and return (trans_t) -1.

void rvm_about_to_modify(trans_t tid, void *segbase, int offset, int size)

Declare that the library is about to modify a specified range of memory in the specified segment. The segment must be one of the segments specified in the call to rvm_begin_trans. The library ensures that the old memory and its metadata has been saved into a separate vector called **undo**, in case an abort is executed. rvm_about_to_modify can be called multiple times on the same memory area.

Every call to rvm_about_to_modify iterates through the local_store vector for the segbase and stores the offset and size to the area being modified, it also sets a flag that marks the segment as about to be modified to true. The user then modifies this area through the pointer returned through rvm_map.

void rvm_commit_trans(trans_t tid)

Commit all changes that have been made within the specified transaction. When the call returns, then enough information should have been saved to disk so that, even if the program crashes, the changes will be seen by the program when it restarts.

The local_store vector is traversed. Every entry for change in local_store corresponding to tid is written to the logfile along with the metadata required to update the persistent copy during truncation. The offset, size and pointer to the data gives the location and size of the change and these are recorded into the corresponding segment's log file. A valid byte is written after the payload entry into the log file is complete. This prevents patialy written log entries(due to crash during commit) from being processed as a log record is only truncated if this valid byte if found to be equal to '1'. Once commit returns successfully, the library guarantees the persistence of the changes of the transaction. It also sets the flag that marks the segment is about to be modified to false by iterating through the local_store.

void rvm_abort_trans(trans_t tid)

undo all changes that have happened within the specified transaction. **undo** vector is traversed to find the first instance of **tid** for a particular segment and the data in the vector is patched back into the local copy. Following this the flag that marks the segment as about to be modified is set to false in the local_store.

Log Control Operations

void rvm_truncate_log(rvm_t rvm)

plays through any committed or aborted items in the log files and shrink the log files as much as possible. rvm_truncate is called whenever rvm_map is called. The API walks through the log files in the specified directory and applies to the datafile. Each log record is isolated using delimiters, and metadata like offset and data size. These are used to write the log data into the persistent data file after the validity of the log entry is checked(valid byte='1'). The log file of each segment is processed line by line, so crashes during the truncation will not produce any undesirable effects as the log record is only removed from the log file when its corresponding changes have already been applied to disk.

Structure of data on the log file:

Delimiter	Offset, Path	Size of Data	Data	Valid

LIbrary Output

void rvm_verbose(int enable_fag)

If the value passed to this function is nonzero, then verbose output from your library is enabled. If the value passed to this function is zero, then verbose output from your library is disabled. When verbose output is enabled, the library prints information about what it is doing.

Result/Test Cases:

The library passes all the following mandatory test cases:

basic.c - Two separate processes commit two transactions and call abort and exit.
abort.c - A commit is aborted and the result is checked to make sure that the previous value is intact.
multi.c - Complete a transaction in one process and check that the transaction completed correctly in another process.
multi-abort.c - Show that transactions can be completed in two separate segments.
truncate.c – Truncate log

Additionally, the library passes the following test case written by us:

tree.c - The user tries to store and modify a 3-level deep nested tree structure, modifying data at various levels. These changes are persisted correctly.

music.c - This test passes two music files to concatenate in memory and the persisted music file is played back to check concatenation without file or header corruption. This demonstrates that the design is agnostic to the type of data.

uncommitted.c - This test is used to check that changes that are not committed are not persisted to disk.

Division of Labor:

Design and development of local store structure - Janardhan
Design and development of hierarchical disk directory - Rahul
Design and development of change persistence/log generation mechanism - Janardhan
Design and development of log truncation/processing mechanism - Rahul
tree.c - Janardhan
music.c - Janardhan
uncommitted.c - Rahul
Debugging mandatory test cases - Rahul and Janardhan
Report - Rahul & Janardhan

Conclusion:

We have successfully implemented and tested the lightweight recoverable virtual memory by providing various APIs to perform transactions on a segment of data by providing persistent memory. The user can make use of the rvm library to design an application and persist critical data and easily recover the data in the event of a crash.