CS 6210: Advanced Operating Systems

Project 4: Lightweight Recoverable Virtual Memory

Fall 2010

1. Goal

In this project you will implement a recoverable virtual memory system like LRVM as described in Lightweight Recoverable Virtual Memory paper. There are other papers available which you may consider reading, including Rio Vista. Users of your library can create persistent segments of memory and then access them in a sequence of transactions.

Making the memory persistent is simple: simply mirror each segment of memory to a backing file on disk. The difficult part is implementing transactions. If the client crashes, or if the client explicitly requests an abort, then the memory should be returned to the state it was in before the transaction started.

To implement a recoverable virtual memory system you should use one or more log files. Before writing changes directly to the backing file, you can first write the intended changes to a log file. Then, if the program crashes, it is possible to read the log and see what changes were in progress.

More information is available in the above-mentioned papers. It is up to you how many log files to use and what specific information to write to them. You may again work in pairs for this project. Your submission must provide a Makefile.

2. RVM Library API

Your library must implement the following functions:

a. Initialization and Mapping Operations

- ***rvm_t rvm init(const char *directory)*** - Initialize the library with the specified directory as backing store.

- ***void *rvm_map(rvm_t rvm, const char *segname, int size_to_create)*** – map a segment from disk into memory. If the segment does not already exist, then create it and give it size size_to_create. If the segment exists but is shorter than size_to_create, then extend it until it is long enough. It is an error to try to map the same segment twice.
- ***void rvm_unmap(rvm_t rvm, void *segbase)*** - unmap a segment from memory.
- ***void rvm_destroy(rvm_t rvm, const char *segname)*** - destroy a segment completely, erasing its backing store. This function should not be called on a segment that is currently mapped.

b. 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. If any of the specified segments 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. Your library needs to ensure that the old memory has been saved, in case an abort is executed. It is legal to call rvm_about_to_modify multiple times on the same memory area.
- ***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.
- ***void rvm_abort_trans(trans_t tid)*** - undo all changes that have happened within the specified transaction.
- ***int rvm_query_uncomm(rvm_t rvm, const char* segname, trans_t **tids) –*** return number of uncommitted transactions in the segment. Stores all uncommitted transactions in the tids array.

c. Log Control Operations

- ***void rvm_truncate_log(rvm_t rvm)*** - play through any committed or aborted items in the log file(s) and shrink the log file(s) as much as possible. You also have to implement the “epoch-based truncation” as mentioned in the LRVM paper. The LRVM library should begin truncation whenever the logsize exceeds a fraction of its total size. What the value of the fraction is upto you. You need to take into consideration how often do you want to do the truncation and how that would impact current application performance. Please mention the fraction you have chosen and its tradeoffs in your report.

d. Library Output

- ***void rvm_verbose(int enable_flag)*** - 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, your library should print information about what it is doing.

3. Crash Recovery

Whenever a program using RVM fails either to commit its previous transactions or crashes during execution, it should be able to recover its previous data once it starts again. When restarted the program will perform rvm_map on the same segment it requires. Your version of rvm_map should also need to perform crash recovery if there are pending commit transactions in the log file but not in the data segment. You will need to create a recovery tree of commits in the log from tail to head then applying the changes to the segment/disk file to make them permanent. Then you would need to update the log to reflect it as empty.

4. Test Cases/Demo

In order to get a feel for how the above API is used, you should write some test cases that use the above functions and check whether they worked correctly. To implement your test cases, you will probably want to use multiple processes, started either with fork() or by starting programs from a shell script. You must also simulate crashes and demonstrate recovery; exit() and abort() functions are useful for simulating the crashes.
Your RVM library must minimally pass the tests that are given below. You must also demonstrate at least three other tests that you believe are relevant.

The TAs may run other test cases not listed here against your library. Here are some tests the TAs will use:

- 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

Source code for above test cases is attached along with the assignment (on T-Square).

You'll notice that there is a file named "rvm.h" that is referenced in each one of test cases below. You must place all of the previously described function declarations in the rvm.h file. You must also generate a library file named librvm.a for your RVM library.

5. Write Up

You must have a README file in your source code (which includes directions on how to compile and run your code). You must also write a report (in PDF only) for the project which includes a short description of your design, how your library uses logfiles, performs recovery and how it manages transactions. The write up should also include information on things that do not work well.

6. Deliverables

You must submit the project on T-Square (strictly before the deadline). Your deliverable should include the following:

- Your RVM library source code
- A Makefile for compiling your code and generating your library
- Test cases for your library and related testing code
- Documentation of the contributions of each team member
- Your write up for the project