Project 4: Recoverable Virtual Memory

Due Nov 26, 11:59pm

Introduction

In this project you will implement a recoverable virtual memory system like LRVM as described in Lightweight Recoverable Virtual Memory 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.

The API

Your library must implement the following functions:

Initialization & 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.

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 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.

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.

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.

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 an rvm.h file. You must also generate a library file named librvm.a for your RVM library.

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 TA may run other test cases not listed here against your library.

Following are the minimal test cases that your library should pass:

- 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

Write Up

You must deliver some documentation, in the form of a document or README file in your source code, which includes directions on how to compile and run your code. You must also include a short description of your design that includes information on how your library uses logfiles and how it manages transactions. The write up should also include information on things that do not work well.

Deliverables

You must deliver your project via T-Square by Nov 26th,

11:59pm. 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;