Assignment - In progress

Add attachment(s), then choose the appropriate button at the bottom.

Title	Project 4: Recoverable Virtual Memory
Due	Apr 19, 2013 11:55 pm
Status	Not Started
Grade Scale	Points (max 10.0)
Modified by instructor	Mar 29, 2013 1:56 pm

Instructions

Project 4: Recoverable Virtual Memory

Introduction

In this project you will implement a recoverable virtual memory system like the ones described in the LRVM and Rio Vista papers. Users of your library can create persistent segments of memory and then access them in a sequence of transactions.

You may work in groups of two.

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.

The API

Make a static library exactly named librvm.a and a header file exactly named rvm.h. If you don't know how to make a static library, man ar and look at the 'r', 's', and 'v' flags. Your library must implement the following functions exactly:

- 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.
- 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. **Note that trant_t needs to be able to be typecasted to an integer type.**

- 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.
- 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.

Test Cases

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 withfork() or by starting programs from a shell script. You may also want to simulate crashes within the program; the exit() andabort() functions are useful for this.

Each test case should print out either "OK" or "ERROR"; if it prints "ERROR" then it should give some description of what went wrong.

I won't grade your test cases, but you are very likely to have fewer bugs in your program the more tests you write. Thus more tests have an indirect effect of increasing your score.

Attached are some (but not all) tests the TA will use.

Deliverables

You must make your code work on Fedora 13.

- Please submit a compressed file including:
  - Your library source code (including rvm.h)
  - Makefile (that builds librvm.a)
  - A README file including:
    - How to compile your library.
    - How you use logfiles to accomplish persistency plus transaction semantics. How many files do you have? What goes in them? How do the files get cleaned up, so that they do not expand indefinitely?
    - Any thoughts you have on the project, including things that work especially well or which don't work.
- *How*: If two students worked as a team, one should submit all materials mentioned above, and another student should submit a textfile including their names.

Additional resources for assignment

- abort.c (2 KB; Jan 22, 2013 11:53 am)
- basic.c (1 KB; Jan 22, 2013 11:53 am)
- multi.c (2 KB; Jan 22, 2013 11:54 am)
- multi-abort.c (2 KB; Jan 22, 2013 11:54 am)
- truncate.c (1 KB; Jan 22, 2013 11:54 am)

Submission

This assignment allows submissions by attaching documents only.

Attachments

No attachments yet

Select a file from computer Choose File No file chosen