**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: mirror each segment of memory to a backup file on disk. The difficult part is implementing transactions. If the client crashes, or if the client explicitly requests an abort, 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 backup file, you must 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**

Fill out rvm.cpp and rvm\_internal.h from a given code template to implement the following functions exactly:

* rvm\_t rvm\_init(const char \*directory) - Initialize the library with the specified directory as backup 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 to the given size. It is an error to call rvm\_map on the same segment twice -- return (void \*) -1 for this case.
* 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 -- return immediately for this case.
* 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, 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 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, 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) - apply 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 check how the above API is used, you should write some test cases. 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 may also want to simulate crashes within the program; the exit() and abort() functions are useful for this.

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

The test cases will not be graded directly, but they do help your debugging.

The attached code template includes some test cases (but not all) that TA will use.

**Deliverables**

You should make your code work on a common Ubuntu release. Note that some linux distributions and different version of kernel may show different behaviors for the functions mentioned above. If you want to use different platform, please discuss with TA and be prepared to demo your work.  
  
For this assignment, you need to submit the following files.

* rvm\_internal.h (this defines your own data structures including rvm\_t and trans\_t)
* rvm.cpp (this implements the rvm interfaces that are previously described)
* README file including
  + How log files are used to accomplish persistency and transaction semantics.
  + What is written in them? How do the files get cleaned up, so that they do not expand indefinitely?

You need to make sure that your code works with the given api. You can simply fill out the given rvm.cpp and rvm\_internal.h in the template to ensure this.  
  
If two students worked as a team, one should submit all materials mentioned above, and another student should submit a textfile including their names.

DO START EARLY and have fun!